Product Sequencing on a Mixed-Model Assembly Line using Deep Learning

Christoffer Lindkvist

January 30, 2026

Abstract

A mixed-model assembly line manufactures different product variants on a single line, where variations in tasks can create imbalances across the workstations. When several labour-intensive models appear consecutively, some stations may exceed their capacity, leading to overloads that require halting the entire assembly line to resolve the issues.

This challenge is formalized as *the Product Sequencing Problem*, an NP-hard optimization task concerned with arranging production orders into efficient sequences. This thesis investigates whether deep learning can be used for solving this problem. Using a Transformer-, Pointer Network-, and Sequence to Sequence model trained to emulate tacit scheduling knowledge captured in historical data, and using its predictions to mimic rule-based approaches whilst still providing time-based improvements.

Acknowledgements

Funding: Vinnova grant 2023-00970 (EUREKA ITEA4 ArtWork), Vinnova grant 2023-00450 (Arrowhead fPVN, Swedish funding), and KDT JU grant 2023-000450 (Arrowhead fPVN, EU funding).

Contents

1. Introduction

1.1 Background

This thesis addresses machine learning applied to the Product Sequencing Problem, an NP-Hard optimization problem which arises in the planning of mixed-model assembly lines [6]. Traditionally, product sequences are determined manually by management staff, relying primarily on tacit knowledge accumulated through experience. While this often produces feasible solutions with relatively few scheduling conflicts, it remains ad hoc and sporadic.

The main constraint for this problem is that the assembly line moves synchronously; if a single station exceeds its cycle time, often depicted as an overlap (see 1.7), then the *entire* assembly line halts. Thus, the goal of this thesis is to minimize the number of production halts.

The methods used in assembly today are strictly rules-based, and due to a large number of rules that have accumulated over the years a lot of these subtleties can prove highly restrictive in search of better solutions.

1.2 Research Problem

Current approaches in practice rely entirely on human expertise and tacit knowledge, which limits scalability and consistency. If this knowledge could be systematically emulated using a model that mimics historical sequencing data, and in effect tacit knowledge, it may provide stronger starting points for any given adjustments to be made later on. Consequently, switching from a strictly rules-based to a time-based approach can help reduce the number of station overloads (1.7), and conflicts.

1.3 Objectives

The objectives of this thesis are:

1. To design a deep learning model capable of emulating the tacit knowledge of management workers using historical data.

2. To investigate which model performs the best in the given task of permuting input data to an acceptable solution.

3. To provide a visualization tool that intuitively illustrates the scheduling flow, highlighting overlaps, borrowed time, and bottlenecks across stations and time.

1.4 Visualization of the Problem

The user interface (UI) will visualize the flow of the assembly line along two axes: one representing stations and one representing clockcycles. A clockcycle is defined as the time required for an item to move from one station to the next. In the visualization, items are displayed in a way that reflects the relative duration of processing at each station. In Figure 1.1, this is illustrated by stretching items along the timeline to better represent the clockcycles. Note that a clockcycle is an arbitrary unit of time and does not correspond to real-world durations. For the purpose of this thesis, one clockcycle is defined as the time it takes for an arbitrary item X to move from station S_n to station S_{n+1}.

Each entry, whose size represents the time it needs to complete its cycle, may borrow time from a previous or upcoming station, this is refered to as the *"drift area"* (see 1.6). Unfortunately, this is where the problems related to the Mixed-Model Assembly Lines start to arise. If time-intensive items are placed consecutively, we will experience an overlap, as the time-allocations will not fit given the constraints of the station and drift areas.

Issues in visualizing this way start to appearing when we start to consider that different stations S_n and S_m may take different times to complete. If we then step a clockcycle for each possible item, then we can never keep our items in sync. The main issue is that; if we compare the station S_n and S_m, then we'll see that each station have a different time to finish, then the clockcycle system will not be perfect or even realistic as stations with differing times will each finish in different times and thus an item X might make it to the station S_{n+2} from S_n in the same time it takes item Y to make it to S_{m+1} from S_m.

Thus we find the difficulty in displaying it properly in an intuitive graphical user interface. If we wish to display each station as uniform sizes, then we also have to stretch the items to make up for it visually. But doing this we have no intuitive way of knowing that S_4 could be 200 seconds long in real life, while S_3 could be 300. *As luck would have*

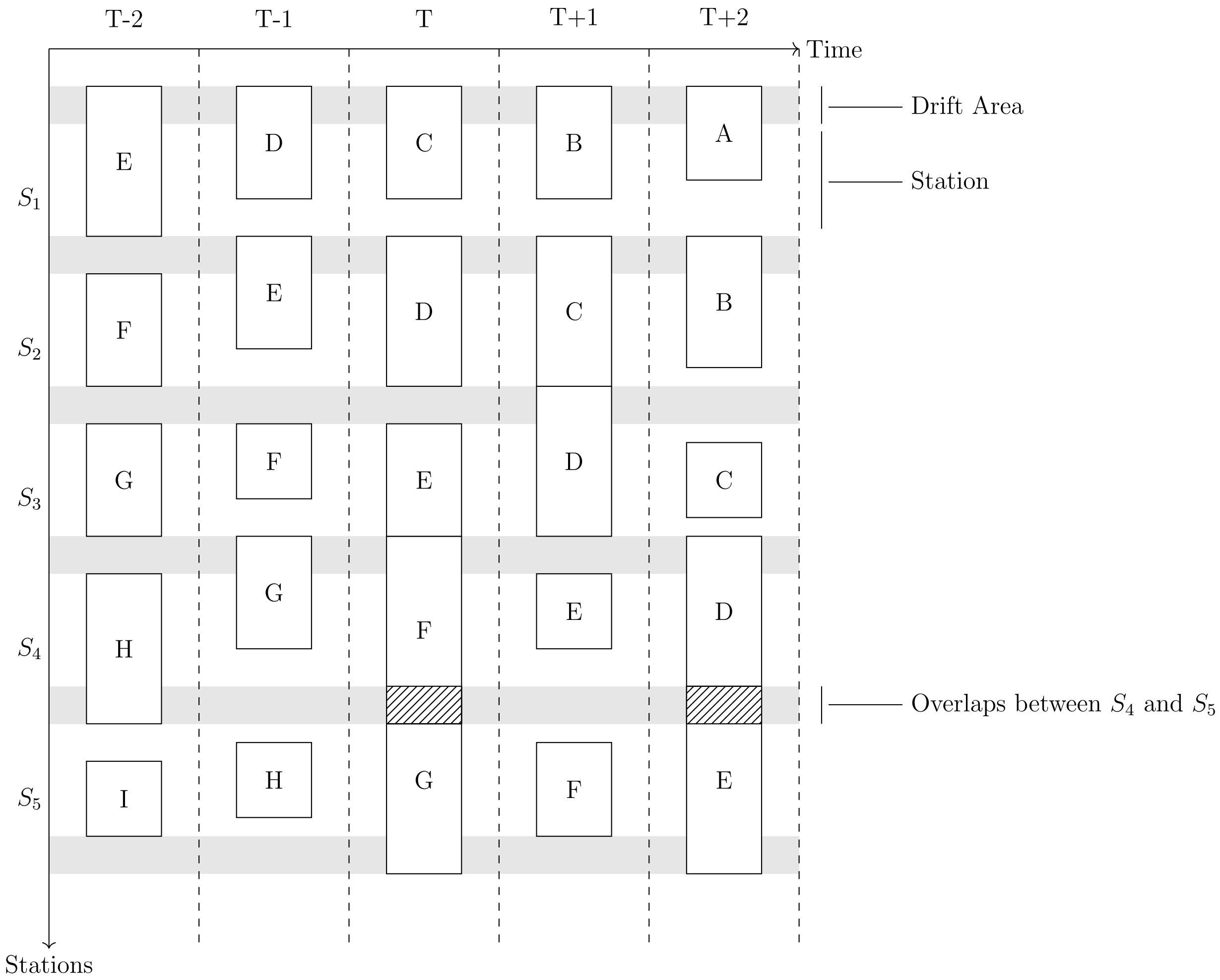

Figure 1.1: Assembly Line Example with Uniform Station and clockcycles

it, each station in this specific case are each roughly 7 minutes long, 700cmin, thus we will not run into any major desync problems using clockcycles on these stations.

Between each station lies a buffer zone refered to a "drift area". A drift area in this case is a transitional area between any given station S_n and S_{n+1}. Both of the stations can borrow time from each other within this area, but only one station may utilize that area at the time. This proves useful to help fit items that take longer on some stations onto the assembly line, but these conveniences which at first glance makes this problem easier are also the source of problems that may ensue in production.

As pictured in Figure 1.1, D will take a lot of time on S_4 and is forced to utilize time from S_3 and S_5. While this works well in a vacuum, the problems start to arise when E also has to utilize additional time from its neighbouring stations, causing an overlap between D and E at $T + 2$ as they both require the use of the drift area.

The same problem can be seen at T with F and G as both items need to borrow time from the stations before and after. Thus we run into an overlap, as the items *cannot* fit.

Do note that on T, E does not utilize the drift area which results in it sitting flush with F on the timeline, this may look good on paper but can result in overlap in practice due to the human workers at the assembly line occasionally taking a bit longer than presumed. This can be resolved by borrowing some time from S_2 and moving E into the drift area.

The same issue occurs at $T + 1$ where C and D just *barely* get enough time, but it cannot get resolved by simply moving D forward, as D on $T + 2$ will require all of the time it can get on S_4.

1.5 Jump-in and Jump-out

In modern production, product sequences are typically finalized and frozen in batches before entering the assembly line. Ideally, the order remains unchanged, but last-minute adjustments may occasionally be necessary. When a product is missing critical components and cannot be built, it cannot proceed on the assembly line. In such cases, the product must be temporarily removed from the sequence and held until all parts are available, a process referred to as *jump-out*.

A related challenge occurs when the missing components finally arrive. At this point, the product occupies space on the assembly station. Ideally, it should be reintegrated into the line as soon as possible, preferably before the next batch begins production. Currently, however, reintegration is delayed, and the suspended product may not re-enter the assembly line until several batches later. [6]

From an algorithmic perspective, a *jump-in* can be treated as an $O(n)$ problem: one needs only to inspect the n positions in a sequence of length n to determine the best insertion point in the upcoming batch. Implemented correctly, a jump-in could occur as soon as the following batch. Nevertheless, depending on operational constraints, it may not be feasible in every batch.

Jump-out, on the other hand, can be considered an $O(1)$ problem, though it presents additional practical challenges. Removing a product from the assembly line creates a gap that must be managed. This gap may require shifting a portion of the line by a clock cycle, potentially causing overloads. Alternatively, the gap could be left in place, which avoids overloading but may result in lost revenue.

1.6 Defining Drift Area

The Drift Area is visualized in this thesis as a gray zone intersecting two neighbouring stations, it is also the source of our scheduling conflicts. The Drift Area allows for a lot of flexibility due to the fact that *instead* of slowing down the assembly line to 1100 Cmin and decreasing the production output by roughly 40%, the assembly line instead moves at a pace of 700 Cmin per station (Standard Takt), with 200Cmin available as *drift* on both sides.

For a station S_n (where $Takt = 700$ Cmin), we can use up to 200 Cmin from S_{n-1} and up to 200 Cmin from S_{n+1}. This is refered to in this thesis as **Borrowing**. Borrowing this way will give us up to a total time of 1100 Cmin for any given task, with the caveat of S_{n-1} and S_{n+1} being reduced to a minimum of 500 Cmin.

It's also important to note that while Object n is borrowing from S_{n+1}, then work cannot begin on $n+1$ until it has finished.

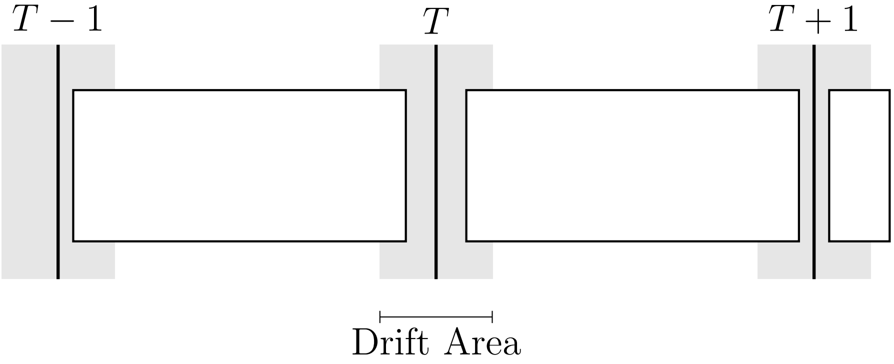

Figure 1.2: Two objects whose $Size_{T_n} < Takt$. This is the most common case, as opposed to the more extreme cases shown in this thesis.

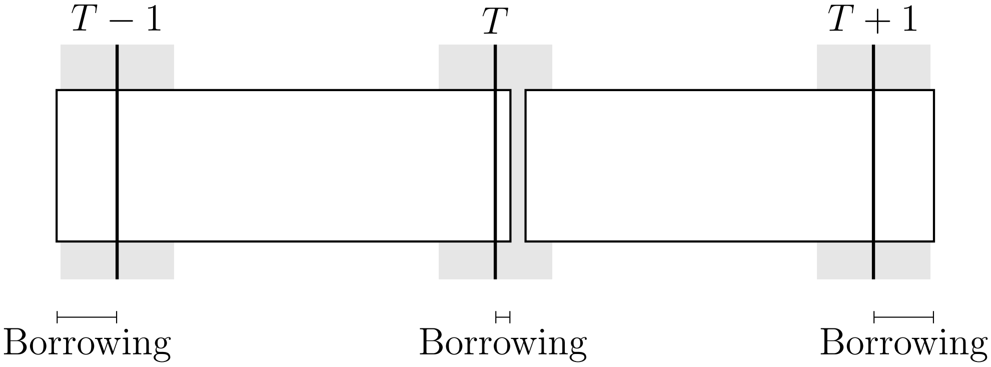

Figure 1.3: Two objects whose $Size_{T_n} > Takt$. Both need to borrow time from their neighbouring stations in order to finish in time.

1.7 Defining Overlaps

Overlaps stem from timing dependencies: each item's "size" on a station S_n depends on how long it takes for the item to process at the station. In other words, the "size" of the task is the duration of the task relative the duration of the station. If a task exceeds **Takt** in duration, then the use of the drift area is mandatory.

However, if two items are processed consecutively at station S_n, and both of those items exceed **Takt** in S_n to the point of requiring the usage of both of its neighbouring stations (S_{n-1} and S_{n+1}), then there *will* not be sufficient time for all operations to complete if both of those objects need to borrow time. These conflicting dependencies result in what's visualized as (and defined in this thesis as) an overlap, there simply isn't enough time for assembly to finish in time.

In practice the items do not literally stack or collide. Rather, the assembly line must halt in order to allow the operation to complete before the assembly line may continue again.

Overlaps are how station overloads are visualized in this thesis, as we follow the constraint that no matter what, an allocation can never exceed $Takt + 2 * Drift$ in size, and an allocation can never go past the right-most limit of the drift area on the coming station.

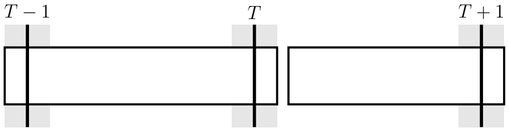

Figure 1.4: An example of two allocations, where there is enough room to borrow time for the larger one.

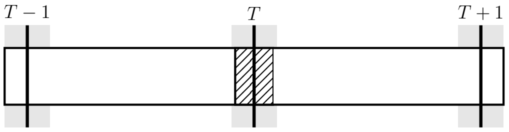

Figure 1.5: An example of two allocations, where there is not enough room as both require borrowing, causing an overlap.

2. State of the Art Analysis

The Mixed-Model Assembly Line (MMAL) problem is an optimization problem, formally classified as NP-Hard. Traditionally, these optimization problems are addressed using heuristic or meta-heuristic algorithms, rather than using machine learning.

In recent years, Deep Reinforcement Learning (DRL) has become a dominant approach for solving dynamic scheduling problems. Unlike supervised methods, DRL agents learn through trial-and-error interaction with a simulation environment, aiming to maximize a cumulative reward signal.

For example, Chen et al. proposed an Adaptive Deep Q-Network, which builds on the Q-Learning Algorithm in order to address scheduling in Cloud Manufacturing environments characterized by stochastic task arrivals. Their approach utilizes a resizable network structure to adapt to changing machine availability and employs a complex reward mechanism to balance multiple objectives, such as minimizing work time, and optimizing machine load. [10]

However, reinforcement learning requires trying and failing repeatedly in order to learn, and applying such trial and error Reinforcement Learning methods to a Mixed-Model Assembly Line (MMAL), especially in a production environment, will pose significant risks. In a cloud environment, a scheduling error typically results in a slowdown in the form of increased latency or reduced throughput. In contrast, a scheduling error in a physical MMAL situation often results in scheduling overlaps that force the entire assembly line to halt in order for each station to be able to finish in time. Given these higher stakes, the exploratory nature of reinforcement learning agents may be prohibitively costly compared to imitating proven human strategies, hence the use of Deep Supervised Learning was chosen instead.

While DRL is effective for dynamic environments where rules change frequently, it presents significant implementation challenges. It requires the construction of a high-fidelity simulation environment and the careful engineering of state and action spaces. Furthermore, the "black box" nature of the reward signal makes it difficult to capture the nuanced, unwritten knowledge of the human servicemen, which is the primary objective of this thesis.

Current manual approaches rely heavily on the "tacit knowledge" of management staff, knowledge accumulated through experience that is difficult to articulate as explicit rules, especially if rules have accumulated over time. This thesis proposes capturing this knowledge using Supervised Learning, formally known in this context as Imitation Learning.

2.1 Recurrent Architectures

To emulate the sequential decision-making process of human schedulers, we first investigate architectures adapted from Natural Language Processing (NLP) that process data sequentially.

2.1.1 Long Short-Term Memory (LSTM)

In previous works addressing similar scheduling problems, researchers have applied Recurrent Neural Networks (RNNs), often utilizing Long Short-Term Memory (LSTM) units within a sequence-to-sequence (Seq2Seq) framework [2].

A defining characteristic of LSTMs is their ability to selectively forget irrelevant or outdated information via the "forget gate". This mechanism allows the model to focus on relevant patterns over time, mitigating the vanishing gradient problem inherent in standard RNNs and improving the modeling of long-term dependencies [4, 9].

2.1.2 Sequence-to-Sequence (Seq2Seq) Models

Seq2Seq models, typically built upon encoder-decoder RNN architectures, are designed to map an input sequence to an output sequence of a different length or order [2]. While traditionally applied to machine translation (e.g., transforming English to Swedish), this architecture is adaptable to the assembly line problem. In this context, the input is a sequence of vectorized product orders, and the output is the permuted production sequence [3].

However, applying standard Seq2Seq models to permutation problems presents a specific challenge: **vocabulary definition**. Standard NLP models select outputs from a fixed, pre-defined vocabulary, similar to a dictionary. In a manufacturing context, where every day's product list is unique, a fixed vocabulary is insufficient. The model must instead learn to select from the dynamic input available that day, a task that is non-trivial for standard Seq2Seq architectures unless they strictly rely on abstract sequence indices [2, 3].

One way to remedy this limitation is to modify the decoder to point directly to the input elements, an approach that leads to the development of Pointer Networks,

2.1.3 Limitations of Recurrent Architectures

Despite their historical success, RNN and LSTM-based models suffer from a significant bottleneck: **sequential processing**. This is because these models must process the input sequence step-by-step, so that item t depends on item $t-1$, they cannot parallelize computation either. This results in slower training times and, more importantly, a limited ability to capture a lot of global context across the entire schedule simultaneously [4].

This limitation motivates the shift toward **Attention-based approaches**, which process the entire sequence in parallel, and has a larger global context scope.

2.2 Attention-Based Architectures

2.2.1 The Transformer Model

To address the sequential bottlenecks of RNNs, Transformer-based architectures discard recurrence entirely, relying instead on a self-attention mechanism [5].

The defining characteristic of the Transformer is the use of scaled dot-product attention. This allows the model to weigh the importance of each element in a given sequence relative to all other elements in that same sequence, regardless of distance. [14] This parallelized computation not only accelerates training but enables the model to capture global dependencies more effectively than RNN-based methods. Since Transformers are order-agnostic, positional encodings are added to the input vectors to retain sequence order information [5].

For combinatorial scheduling problems, this translates into the ability to model complex interactions across the entire planning horizon. [13] The placement of one item can be directly conditioned on all others in the same day's sequence, providing the context awareness necessary for effective assembly line scheduling [8].

2.2.2 Pointer Networks

While Transformers excel at context, they still typically rely on generating tokens from a fixed vocabulary. The **Pointer Network** is the only architecture in this analysis specifically designed for combinatorial optimization and permutation problems [7].

Implemented using an encoder-decoder structure of the sequence-to-sequence model, the Pointer Network utilizes a specialized attention mechanism to generate context vectors that "point" to specific elements in the input sequence rather than predicting a value from a dictionary.

This model is explicitly designed with combinatorial problems in mind, such as the traveling salesman problem, convex hull, and other combinatorial optimization problems. [7] It is thus well-suited for the mixed-model assembly line problem (MMAL), as the object attributes (such as station order, or time to complete) are immutable, and the only thing that matters for our case is the permutation/rearrangement.

3. Methodology

3.1 Data generation

3.1.1 Constants

For data generation the following constants are used:

Table 3.1: Table of Constants used in Data Generation

Constant	Default Value	Description
Objects	100	Total number of objects in the testing sequence
Stations	39	Number of stations
Takt	700	The cycle time (*Cmin*)
Drift	200	The size of the drift area (*Cmin*)
Gap	10	Time-buffer (*Cmin*) between generated objects
min_size	100	Smallest possible Object (*Cmin*)
max_size	$Takt + 2 \times Drift$	Largest possible Object (*Cmin*)
Multiplier	5000	Constant value to multiply *Objects* with, used for Training.

3.1.2 Data Preprocessing and Abstraction

For the purpose of training optimization, the raw information is abstracted into a concise JSON schema. Each entry in the dataset is reduced to the following three attributes:

- **Object ID** (`object`): An integer representing the unique identifier of the item.
- **Station Data** (`data`): A mapping of station keys ($s_1, \dots, s_{35}$) to their respective time requirement values.
- **Station Offsets** (`offsets`): A mapping of station keys to their relative offset values.

By utilizing these values, we can train a model to prioritize time-based decision-making over rigid rule-based systems. It is important to note that while the model operates without explicit constraints, the use of a sufficiently large dataset derived from expert tacit knowledge ensures that it implicitly adheres to fundamental ground rules, albeit with greater flexibility.

3.1.3 Offset Calculation and Centering

The positioning of an object within a station is determined by its *offset* relative to the **Takt** start (0 locally). The script employs a dynamic centering logic that positions tasks within the remaining available capacity of the standard Takt window. This approach balances the empty space (or buffer) on either side of the allocation, adapting to the finish time of the preceding task.

To determine the position, the script first calculates the local start time, t_{start}, which corresponds to the completion of the previous allocation relative to the current Takt. The remaining buffer, B, is calculated as the difference between the standard Takt duration T and the occupied time, clamped to zero to ensure non-negative spacing:

$$B = \max(0, T - t_{start} - S) \tag{3.1}$$

Where S is the size (duration) of the new task. The tentative offset, O_{calc}, is then derived by placing the task at the start position plus half of the calculated buffer:

$$O_{calc} = t_{start} + \frac{B}{2} \tag{3.2}$$

This formula ensures that if the previous task finishes early, the new task is centered in the wide gap; if the previous task finishes late, the buffer shrinks, effectively pulling the new task closer to the previous one.

Finally, to ensure the object adheres to physical station boundaries and drift limitations (D), the calculated offset is clamped. The maximum permissible offset is defined such that the task end does not exceed the drift limit ($T + D$):

$$O_{calc} = \max(-D, \min(O_{calc}, T + D - S)) \tag{3.3}$$

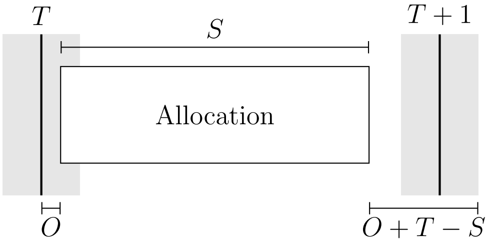

Figure 3.1: Allocation positioning before the centering process

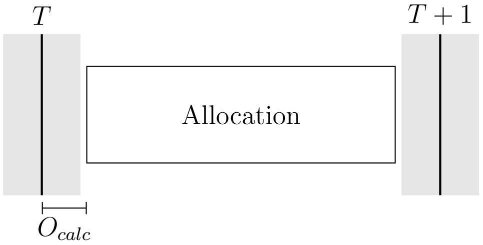

Figure 3.2: Allocation after centering.

3.1.4 Constructive Data Generation as a Solved Puzzle

To train the machine learning model, a synthetic dataset is required that represents valid, collision-free production schedules. Rather than placing tasks stochastically and checking for validity, the system employs a constructive generation algorithm which builds one object at the time. This approach acts as a forward simulation, placing allocations sequentially to inherently create a ”solved puzzle” which serves as the ground truth for training. Due to how the entire sequence generates left to right the risk for overlaps is nonexistent.

Dependency-Based Positioning

The placement of any given allocation $A_{i,j}$ (Object i at Station j) is not independent. It is strictly constrained by two spatial-temporal boundaries derived from the existing grid state:

1. **Station Availability:** The station j must be free. Therefore, $A_{i,j}$ cannot start until the previous object $i-1$ has departed station j.
2. **Object Availability:** The object i must be ready. Therefore, $A_{i,j}$ cannot start until object i has finished processing at the previous station $j-1$.
3. **The Gap** To control the density of the generated data a gap is added (dependent on the values found in 3.1).

Mathematically, the earliest valid start time (t_{start}) for the new allocation is defined as the maximum of these two completion timestamps, plus a configured safety gap:

$$t_{start} = \max(End_{i-1,j}, \quad End_{i,j-1}) + Gap \tag{3.4}$$

This logic ensures that the generated schedule is causally valid—no object effectively ”teleports” between stations, and no two objects overlap on a single station.

Context-Aware Scaling

The dimensions of the tasks are also generated dependently. While a desired duration is chosen stochastically, the actual assigned size (S) is clamped by the remaining available time in the Takt window. The generator calculates the available space between the calculated t_{start} and the maximum drift boundary. If the random size exceeds this space, it is truncated to fit.

This process results in a dataset where every input (the sequence of objects) has a perfectly matching output (the offsets and sizes) that is guaranteed to be solvable, providing a robust target for supervised learning.

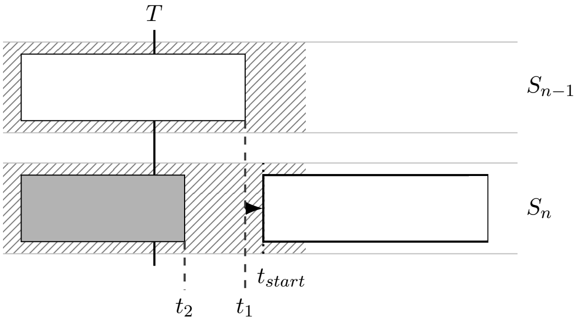

Figure 3.3: Case 1: The white object in S_{n-1} is borrowing time from itself at S_n, and thus cannot begin assembly at S_n until it has finished the task at S_{n-1}.

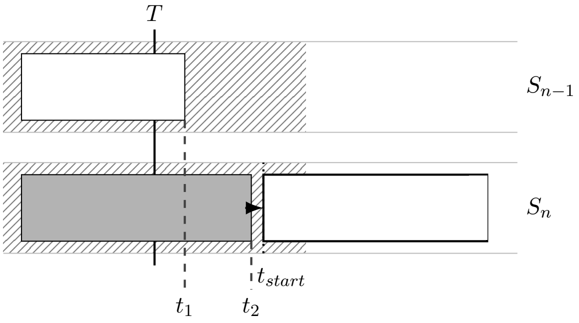

Figure 3.4: Case 2: The gray object is borrowing time from the white object in S_n. Even though it finished in S_{n-1}, the white object cannot begin assembly right away in S_n as it's dependent on the gray object finishing at S_n.

3.2 Adapting Language-Based Models for Sequencing

Two of the models examined in this thesis, the Sequence-To-Sequence model, and Transformer model, were originally designed for natural language processing (NLP) tasks, such as machine translation. These models typically rely on a predefined vocabulary.

This reliance proves problematic when the data does not consist of frequently occurring words, but rather *unique* objects with distinct properties. This raises the question of whether a list of products can be treated analogously to a collection of words, where each product variation constitutes a distinct element within the vocabulary. While this approach allows the model to replicate historical patterns, it risks learning inherent data biases or flaws.

Furthermore, specific constraints must be imposed. Language models tend to overproduce common tokens (e.g., “the”) because they are statistically probable in general English text. To avoid this bias in a combinatorial context, the model must be adapted to generate valid permutations relevant to the specific application (see 3.3).

3.3 Workarounds

The Transformer and Sequence-To-Sequence models have been adapted from their original NLP architectures to a permutation architecture in order to address combinatorial problems. The Pointer Network remains unaffected by these modifications.

3.3.1 Direct Continuous Input

The Sequence-To-Sequence model is implemented here without an Embedding layer. Instead of projecting vocabulary indices into a latent embedding space, this module is omitted to allow raw feature vectors to be processed directly. This enables the model to interpret the specific magnitudes and geometric properties of input elements as continuous values rather than abstract symbols. This approach is particularly well-suited for combinatorial tasks, as it preserves intrinsic numerical relationships in the data that would otherwise be obscured by tokenization.

3.3.2 Soft Decoding

The Sequence-To-Sequence model employs a ”soft” feedback loop during sequence generation. In traditional language processing, the decoder typically commits to a single discrete choice by selecting the most probable word and treating it as the ground truth for the subsequent step. In this context, however, rigid early choices can trap the model in sub-optimal paths. To mitigate this, the model feeds the full probability distribution into the next step rather than a single discrete selection. By propagating uncertainty rather than discarding it, the system can dynamically refine its strategy as the output sequence is constructed.

3.3.3 The Pointer Head

The greatest deviation from the canonical Transformers is the removal of the vocabulary projection layer. It is replaced by a `PointerHead` module designed to select elements from the input sequence, inspired by the Pointer Network. 4.1.3

Instead of predicting a token, the Pointer Head computes a similarity score between the current decoder state, and the encoder outputs.

$$\text{Scores} = \frac{(Dec_{out}W_q) \cdot (Enc_{out}W_k)^T}{\sqrt{d_{model}}} \tag{3.5}$$

These raw logits scores represent the compatibility between the current decoding step and every element in the input sequence. When passed through a Cross-Entropy loss function during training, they effectively train the model to ”point” to the correct input index for the next position in the sequence.

4. System Architecture

4.1 Machine Learning Models

The models in this thesis are implemented using PyTorch.

Table 4.1: Hyperparameters and Model Constants

Constant	**Transformer**	**Sequence-To-Sequence**	**Pointer Network**
batch_size	16	16	8
num_epochs	10	10	10
lr	$1e^{-3}$	$1e^{-4}$	$1e^{-4}$
d_model	256	256	256
dropout	0.1	0.2	
hidden_dim		128	128
d_ff	512		
n_heads	8		
n_layers	3		

4.1.1 The Transformer Model

The implemented model adapts the standard Transformer architecture originally proposed by Vaswani et al. [5] to handle continuous combinatorial data. Unlike standard NLP models that process discrete tokens, this implementation operates on continuous feature vectors and utilizes a pointer mechanism for output generation.

Continuous Input Embeddings

Standard Transformers typically utilize a lookup table to map discrete vocabulary indices to dense vectors. Since the input data for this problem consists of continuous immutable features (station order (always 1 - n), station times, dimensions), the embedding layer is replaced by a learnable linear projection.

Positional Encoding

As the Transformer architecture contains no recurrence or convolution, it can thus not distinguish the order of the input objects, essentially treating it as an unordered set. To address this, a fixed sinusoidal Positional Encoding is added to the input embeddings. This provides each element with a unique geometric signature, allowing the model to determine the relative positions of objects within the sequence. The encodings are calculated as [5]:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}}) \tag{4.1}$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}}) \tag{4.2}$$

This mechanism allows the model to distinguish between identical tasks placed at different positions in the sequence.

Layer Normalization and Pre-Norm Formulation

The system employs Layer Normalization to stabilize training [1], specifically adopting "Pre-Normalization" where the input is normalized *before* passing it to the sub-layer, rather than after. This configuration, implemented in the `ResidualConnection` module, allows gradients to flow more directly through the network, generally improving convergence in deeper models.

Multi-Head Attention

The core mechanism of the model is Multi-Head Attention (MHA), which allows the model to jointly attend to information from different representation subspaces. The implementation utilizes h parallel heads. For each head, the input is projected into Query (Q), Key (K), and Value (V) matrices. The attention scores are computed using Scaled Dot-Product Attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4.3}$$

Feed-Forward Block

Each encoder and decoder layer contains a position-wise Feed-Forward Network (FFN). This consists of two linear transformations with a ReLU activation function in between:

$$FFN(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2 \tag{4.4}$$

The inner layer projects the input to a higher dimension (d_{ff}, typically $4 \times d_{model}$) before projecting it back, introducing additional non-linearity to the model.

The Pointer Head

This implementation of the Transformer replaces the vocabulary projection layers in favour of a Pointer Head in order to handle combinatorial tasks rather than NLP-based tasks. See 3.3.3

4.1.2 Sequence-to-Sequence

The implemented model is a standard Sequence-To-Sequence model with modifications clarified in 3.3.

Continuous Input Encoder

The `EncoderRNN` processes the input sequence to generate a latent representation of the problem state. Unlike standard NLP encoders that require an embedding layer to project discrete tokens, this module accepts raw feature vectors directly.

Attention Mechanism

To overcome the bottleneck of compressing the entire sequence into a single fixed-length vector, the model employs an Attention mechanism, referred to as Bahdanau Attention [16]. This approach allows the decoder to ”search” the entire input sequence and dynamically focus only on the information relevant to the current step.

Rather than relying on a static summary of the input, the mechanism calculates a dynamic ”relevance score” for every item in the input sequence. These scores effectively function as a spotlight, determining how much focus the model should place on each specific input element when generating the next part of the output.

These scores are then converted into probabilities that sum to 100%. Finally, the model computes a ”Context Vector,” which is a weighted average of the inputs based on these probabilities. This ensures that the model ”mixes” the most important information together to guide the next prediction, prioritizing the relevant inputs while ignoring irrelevant ones.

Context-Aware Decoder

The decoder (`DecoderRNN`) generates the output sequence step-by-step. To maximize the utility of the attention mechanism, the context vector c_t is injected at two distinct points in the decoding step:

1. **Input Feeding:** The context vector is concatenated with the input of the current time step before being passed to the LSTM.

2. **Output Projection:** The context vector is concatenated with the LSTM output before the final linear projection layer.

This structure ensures that both the recurrent processing and the final prediction are directly conditioned on the relevant parts of the source input.

4.1.3 The Pointer Network Model

The proposed solution implements a Pointer Network architecture adapted from Vinyals et al. (2015). [7] The model follows an Encoder/Decoder paradigm enhanced with an attention mechanism, specifically designed to handle variable-length input sequences and output a permutation of the input indices.

Encoder Mechanism

The encoder processes the input sequence to generate a latent representation of the problem state.

1. **Input Projection:** Raw input features are first passed through a `Projector` block. This module consists of a Linear Transformation, an Tanh Activation, and Layer Normalization. [1]

2. **Bidirectional LSTM:** The projected features are fed into a bidirectional LSTM. This allows the model to capture context from both past and future states in the sequence.

Decoder and Attention Mechanism

The decoder generates the output permutation one index at a time. Unlike standard NLP models that generate new tokens from a fixed vocabulary, this decoder selects indices directly from the input sequence.

- **Glimpse Mechanism:** To enhance the pointing accuracy, the model employs a "glimpse" or "multi-hop" attention step. At every time step t, the decoder's current state s_t queries the encoder outputs to calculate a context vector c_t:

$$c_t = \sum_{i=1}^{T} \text{Attention}(s_t, h_i) \cdot h_i \tag{4.5}$$

  This vector summarizes the most relevant parts of the input sequence for the current decision.

- **Pointer Selection:** The final selection is performed by a second attention layer. To fuse the information, the context vector c_t is projected and added to the current decoder state to form a *combined query*:

  $$u_t = s_t + \tanh(W_{proj} \cdot c_t) \tag{4.6}$$

  This combined query u_t is then used to compute the final attention scores (logits) over the input elements. The index with the highest probability is selected as the next element in the permutation.

Masking and Inference

To ensure the validity of the resulting permutation (i.e., the Traveling Salesman constraint that every city is visited exactly once), a masking mechanism is applied during inference.

Once an input index k is selected, it is masked out by setting its log-probability to $-\infty$ ($1e^{-9}$) for all subsequent steps. This hard constraint prevents the model from selecting the same element twice. Furthermore, the model is auto-regressive: the feature vector of the selected element h_k is used as the input to the decoder for the next time step $t + 1$.

4.2 The Dataset

To train the models, a custom PyTorch Dataset was implemented to handle combinatorial data. This pipeline is responsible for converting raw input jsons into tensors. The dataset looks more or less the same for all models.

4.2.1 Data Processing

The dataset is initialized with raw JSON files containing sequences of station data. The pipeline flattens these structures into a stack of tensors, where each tensor represents a single task sequence with the shape:

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,N} \\ x_{2,1} & x_{2,2} & \dots & x_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M,1} & x_{M,2} & \dots & x_{L,N} \end{bmatrix} \tag{4.7}$$

Where N is the number of stations and M is the the number of objects.

The feature extraction is performed by selecting specific keys $(s_1 \dots s_N)$ from the raw data objects. To ensure numerical stability and faster converges the data is then normalized using:

$$x_{norm} = \frac{x_{raw}}{1000.0} \tag{4.8}$$

Due to memory constraints, the sequences exceeding the maximum defined length are chunked into smaller subsequences to manage memory constraints effectively.

4.2.2 Shuffled Task Generation

The dataset stores the ground truth sequences in their correct, sorted order. However, the data is effectively augmented during the training phase by randomly shuffling the input sequence at every iteration. This stochastic approach ensures that the model encounters the same sequence twice.

To accommodate this shuffling, the target vector is dynamically generated to represent the specific indices required to reconstruct the original order from the disordered input. If the shuffled input is denoted as X' and the original sorted sequence as X, the target index y_t for time step t is calculated to satisfy the condition:

$$X'[y_t] = X_t \tag{4.9}$$

This formulation specifically trains the Pointer Mechanism of the Transformer and Pointer Network to identify the location of the next correct element within the provided input set. (And then ”point” to it)

4.3 The Training Loop

To ensure a fair and consistent evaluation, a standardized training pipeline is implemented across all three architectures (Seq2Seq, Pointer Network, and Transformer). The training logic is built using the PyTorch framework and orchestrates data loading, gradient optimization, and model persistence.

4.3.1 Optimization and Objective Function

Model parameters are optimized using the **AdamW** algorithm [15]. This optimizer was selected for its ability to decouple weight decay from adaptive gradient updates, a modification that consistently yields better generalization than the standard Adam implementation. The learning rate is set to 1×10^{-4} and the weight decay is set to 1×10^{-2}.

The model is trained to minimize the **Cross-Entropy Loss** between the output logits and the target sequence. A masking constraint (`ignore_index=-1`) is applied to handle variable sequence lengths within batches. This effectively isolates the padding tokens from the objective function, ensuring the optimization focuses solely on valid data elements rather than padding.

4.3.2 Regularization and Stabilization

Training deep neural networks, particularly those involving recurrence (RNNs), can be unstable due to the exploding gradient problem. [17] To mitigate this, Gradient Clipping is applied to the model parameters. This acts as a safety mechanism that caps the maximum possible step size during backpropagation. By preventing drastic weight updates, it ensures that outliers in the training data or instabilities in the recurrent layers do not cause the optimization process to diverge.

4.3.3 Transformer Specifics: Masking strategy

Unlike the recurrent models, which process data sequentially by definition, the Transformer processes the entire sequence in parallel. To replicate the sequential nature of the problem and prevent information leakage, a dual-masking strategy is implemented within the training loop via the `make_masks` function:

1. **Padding Mask (Source):** This mask identifies padding tokens in the input batch. It forces the self-attention mechanism to assign zero attention weights to these positions, ensuring the model focuses only on valid tasks.

2. **Look-Ahead Mask (Target):** To preserve the auto-regressive property, a triangular "no-peak" mask is applied to the decoder. This ensures that when generating the prediction for position t, the model can only attend to positions 0 through $t-1$, effectively hiding future ground-truth tokens.

4.4 The Application Programming Interface (API)

The system exposes its core functionalities through a RESTful API, facilitating programmatic interaction with the implemented neural architectures. This interface serves as the primary gateway for data ingestion and inference execution.

4.4.1 Endpoints

/run/{model_type}

This POST endpoint provides a unified interface for sequence processing across different model architectures (**Transformer**, **Pointer Network**, or **Seq2Seq**). Once selected it returns a processed sequence made from the input data, including refitting.

/run/{model_type}/$int : n$/

This POST endpoint extends the interface to support multiple sequence generation. It accepts an integer parameter n, executing the selected model architecture n times to return a list of n alternative processed sequences.

/check/

This POST endpoint will count the number of **overlaps** on a given sequence, based on configuration data.

5. Experiments and results

5.1 Task Performance

Performance is measured by how well the model can take an input with overlapping data and rearrange the tasks to minimize overlaps. The reduction metric is defined at 5.3.

Although the operational constraints of the target problem typically limit sequences to batches of roughly 100 objects, the following results also evaluate the model's scalability beyond this range.

The values in the tables below is based on the ratio found at 5.3 $\frac{O_P}{O_S}$, where O_S is the number of overlaps found in a freshly generated shuffled dataset, and O_P is the number of overlaps found after predicting and processing the items in said dataset. With the improvement rate being defined as

$$Improvement = (1 - \frac{O_P}{O_S}) \cdot 100\% \tag{5.1}$$

he testing data has an optimal solution before getting shuffled where $O_P = 0$. The data is generated using a random number generator, using no seed, to ensure variability across test runs.

5.1.1 Pointer Network Performance

Table 5.1: Test-size of 10

Run	**Ratio (R)**	**Red.**
Run #1	14/72	80.56%
Run #2	15/68	77.94%
Run #3	21/67	68.66%
Avg		75.72%

Table 5.2: Test-size of 100

Run	**Ratio (R)**	**Red.**
Run #1	188/713	73.63%
Run #2	183/741	75.30%
Run #3	180/719	74.97%
Avg		74.63%

Table 5.3: Test-size of 1000

Run	**Ratio (R)**	**Red.**
Run #1	2008/7208	72.14%
Run #2	2013/7234	72.17%
Run #3	2031/7266	72.05%
Avg		72.12%

Table 5.4: Test-size of 10000

Run	**Ratio (R)**	**Red.**
Run #1	20329/72702	72.04%
Run #2	20001/72463	72.40%
Run #3	20151/72567	72.23%
Avg		72.22%

5.1.2 Transformer Performance

Table 5.5: Test-size of 10

Run	**Ratio (R)**	**Red.**
Run #1	7/80	91.25%
Run #2	4/58	93.10%
Run #3	8/64	87.50%
Avg		90.62%

Table 5.6: Test-size of 100

Run	**Ratio (R)**	**Red.**
Run #1	73/721	89.88%
Run #2	76/734	89.85%
Run #3	71/706	89.94%
Avg		89.89%

Table 5.7: Test-size of 1000

Run	**Ratio (R)**	**Red.**
Run #1	693/7276	90.48%
Run #2	686/7137	90.39%
Run #3	656/7268	90.97%
Avg		90.61%

Table 5.8: Test-size of 10000

Run	**Ratio (R)**	**Red.**
Run #1	5259/72219	92.72%
Run #2	5553/72584	92.36%
Run #3	5518/72422	92.38%
Avg		92.49%

5.1.3 Seq2Seq performance

Table 5.9: Test-size of 10

Run	**Ratio (R)**	**Red.**
Run #1	23/78	70.51%
Run #2	16/71	77.46%
Run #3	19/71	73.24%
Avg		73.74%

Table 5.10: Test-size of 100

Run	**Ratio (R)**	**Red.**
Run #1	200/708	71.75%
Run #2	190/760	75.00%
Run #3	203/717	71.69%
Avg		72.81%

Table 5.11: Test-size of 1000

Run	**Ratio (R)**	**Red.**
Run #1	2098/7171	70.74%
Run #2	1987/7236	72.54%
Run #3	2063/7362	71.96%
Avg		71.75%

Table 5.12: Test-size of 10000

Run	**Ratio (R)**	**Red.**
Run #1	20495/72675	71.80%
Run #2	20542/72830	71.79%
Run #3	20262/72554	72.07%
Avg		71.89%

5.2 Dataset

We obtained an official dataset from a production run. However, the raw data contained several inconsistencies that required resolution before it could be used for training. Hence a script for generated dummy dataset based on the original was made for simulating this task, where borrowing and tighter fits are more commonplace than actual production data. (At least from what we saw from it) In theory this would be beneficial as the model is then trained on "harder" sequences, thus sequences with more room for error and wider margins should perform better. How the data was generated is further discussed in 3.1.4

5.3 Evaluation Metrics

To objectively measure the performance of the model, we monitor performance of the models in overlap reduction (see 1.7) using a simple ratio.

$$\text{ratio} = \frac{O_{predicted}}{O_{source}} \tag{5.2}$$

where O_{source} is the total number of overlaps of the shuffled input sequence, and $O_{predicted}$ is the total number of overlaps of the predicted sequence. The closer to 0 it gets, the better it performs, and if it goes above 1 we're yielding worse results than before.

6. Discussion

6.1 Interpretation of Results

The evaluated models demonstrated strong overall performance, with the Transformer architecture notably exceeding initial projections. The Transformer achieved an improvement rate of 93%; while this figure is exceptionally high, it has been verified against a stochastically generated test set that should thus distinct from the training data apart from the predefined limits 3.1. This suggests the model has successfully generalized the underlying logic rather than memorizing patterns. The remaining models exhibited performance metrics in the 70–80% range, falling within a more expected distributions. Crucially, all architectures consistently maintained valid allocations within the defined operational constraints.

6.2 Limitations

6.2.1 Testing and Training Data

Due to the limited size and inconsistency of available production data, this study relies on generated synthetic data (see 5.2). Although this dataset was rigorously designed to adhere to operational constraints, the absence of real-world validation, and edge-cases not accounted for within sequencing, limits our ability to empirically confirm the models' performance in a live production environment.

6.2.2 Hardware and Resource Limitations

Hardware memory constraints restricted the training set to $500,000$ samples, falling short of the millions typically required for optimal Transformer generalization. This limitation likely constrained the model's ability to capture complex patterns. The primary bottleneck was identified within the data parsing implementation rather than the GPU memory capacity itself; future optimizations in data serialization could resolve this to allow for larger batch sizes and more extensive training sets.

Even though Transformer-based architectures generally demonstrate improved generalization with larger datasets, the training size in this study had to be restricted to $500,000$ due to hardware memory constraints. This could have greatly impacted the performance, and feasibility of the Transformer model in this thesis. Preliminary analysis suggests that this bottleneck may stem from the current data parsing implementation, which could be optimized in future work to allow for larger batch sizes and more extensive training sets.

6.3 Future Work

Future iterations of this research should prioritize larger datasets and utilize high performance computing clusters to overcome the memory bottlenecks encountered in this study. This would enable the training of models on datasets significantly larger than the current limit of $500,000$ objects. Furthermore, the current transformer model exhibits signs of underfitting, suggesting that the training loop configuration requires further refinement. While an exhaustive optimization of the training regimen was outside the temporal scope of this work, future efforts should focus on tuning these dynamics to fully exploit the model's capacity.

Additionally, it would be valuable to investigate the feasibility of Online Learning paradigms to see if the model could adapt in real-time using Human-in-the-Loop (HITL) strategies to expand the training scope to include operator overrides. Doing so would incorporate further tacit knowledge, complex reasoning, and constraints that are not captured in the standard training data.

Bibliography

[1] J. L. Ba, J. R. Kiros, and G. E. Hinton, *Layer Normalization*, arXiv preprint arXiv:1607.06450, 2016.

[2] A. Dupuis, C. Dadouchi, and B. Agard, *A decision support system for sequencing production in the manufacturing industry*, *Computers & Industrial Engineering*, vol. 185, p. 109686, 2023.

[3] A. Bay, B. Sengupta, *Approximating meta-heuristics with homotopic recurrent neural networks*, arXiv preprint arXiv:1709.02194, 2017.

[4] I. Pointer, *Programming PyTorch for Deep Learning, ISBN: 9781492045359*, O'Reilly Media, 2019.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, *Attention is All You Need*, Advances in Neural Information Processing Systems (NeurIPS), vol. 30, 2017.

[6] C. Fink, O. Schelén, and U. Bodin, *Work in progress: Decision support system for rescheduling blocked orders*, Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, 2023.

[7] O. Vinyals, M. Fortunato, and N. Jaitly, *Pointer Networks*, Department of Mathematics, University of California Berkeley, 2015.

[8] E. Stevens, L. Antiga, T. Ciehmann, *Deep Learning with PyTorch, ISBN: 9781617295263*, Manning Publications, 2020.

[9] S. Hochreiter, *The vanishing gradient problem during learning recurrent neural nets and problem solutions*, Institut für Informatik, Technische Universität München, D-80290, 1998.

[10] Author, Title, Journal, Year.

[11] T. Mitchell, *Machine Learning*, 9780070428072, McGraw-Hill, 1997.

[12] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, Proceedings of NAACL-HLT, 2019.

[13] P. Giannoulis, Y. Pantis, C. Tzamos, *Teaching Transformers to Solve Combinatorial Problems through Efficient Trial & Error*, Advances in Neural Information Processing Systems (NeurIPS), 2025.

[14] Y. Youssef, P. R. M. de Araujo, A. Noureldin, S. Givigi, *TRATSS: Transformer-Based Task Scheduling System for Autonomous Vehicles*, arXiv preprint arXiv:2504.05407, 2025.

[15] D. Kingma, J. L. Ba, *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*, In ICLR, 2015.

[16] D. Bahdanau, K. Cho, Y. Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*, arXiv preprint arXiv:1409.0473, 2016.

[17] R. Pascanu, T. Mikolov, Y. Bengio, *On the difficulty of training Recurrent Neural Networks*, In ICML, 2013.