Applying Deep Learning on Sequential Data within the Mixed-Model Assembly Line problem

Christoffer Lindkvist

January 18, 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 which requires halting the entire assemblyline to remedy 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 complement heuristic methods 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 initialize a heuristic algorithm. By providing informed starting points, this approach aims to reduce the computation time of the algorithm by presenting an adequate starting point.

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.

To improve upon this, a heuristic algorithm has been proposed. The algorithm first generates a feasible baseline solution from pre-defined constraints, and then it refines the baseline until an acceptable sequence is reached. The central question of this thesis is whether the runtime of such an algorithm can be reduced by initializing it with a baseline informed by historical sequencing data, rather than relying solely on rule-based object placements. The baseline in question will be generated using a deep-learning model.

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 heuristic method. Such informed baselines could potentially reduce the runtime required to reach high-quality solutions, especially for complex sequencing instances.

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 integrate the model as a preprocessing step in the existing heuristic algorithm in order to reduce its runtime.

4. 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"*. 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 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

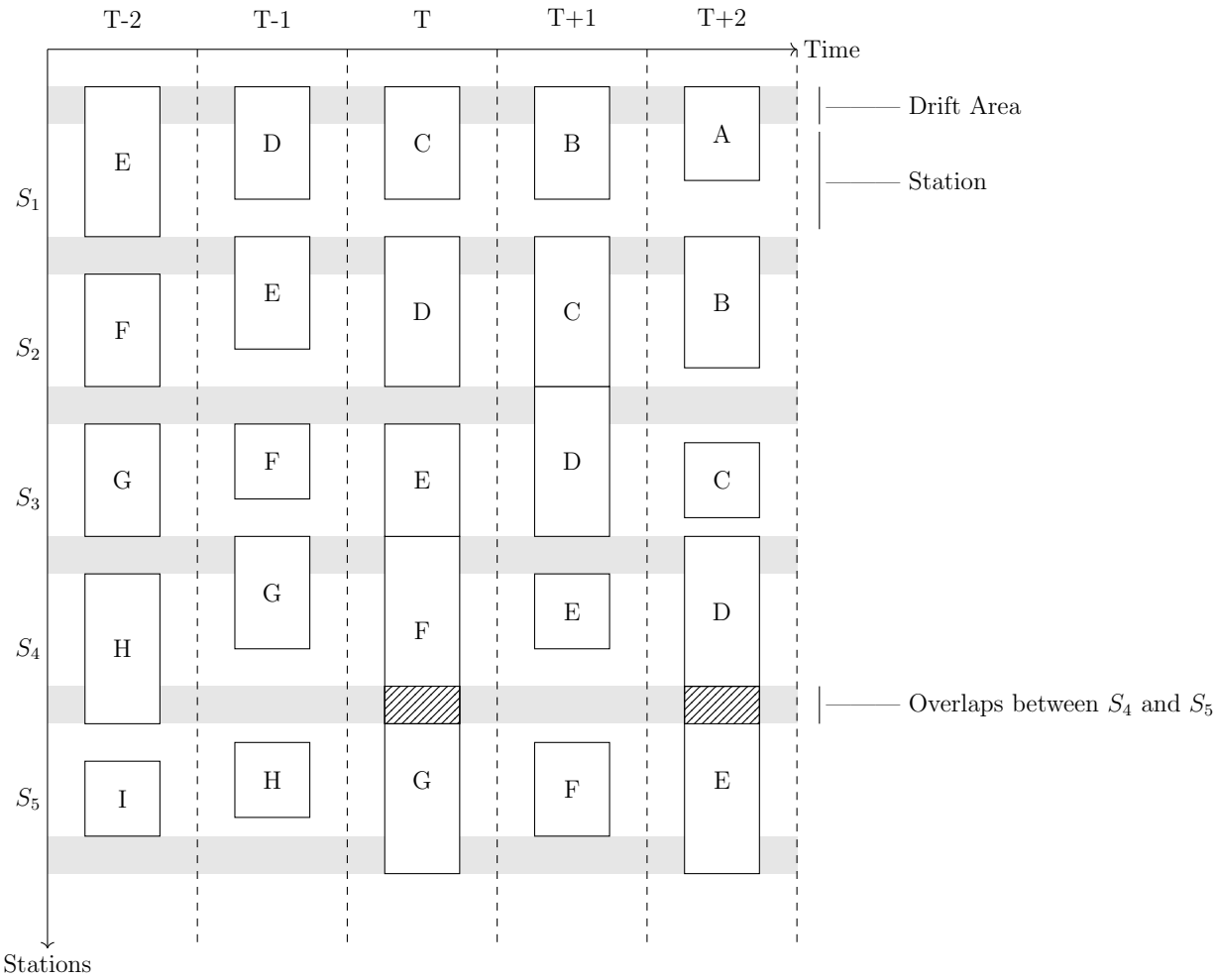

Figure 1.1: Assembly Line Example with Uniform Station and clockcycles

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 derives 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 Overlaps

Overlaps in this case arise from timing dependencies: each item's progress depends on how long it takes to process at its respective station. If two items are processed consecutively at station s_n, and both of those items require additional time from its neighbouring stations s_{n-1} and s_{n+1} there will not be sufficient time for all operations to complete. This results in what's visualized as an overlap. In practice the items do not literally stack or collide. Rather, the assembly line must halt in order to allow these operations to complete before production may continue.

2. State of the Art Analysis

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

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 Agorithm in order to address scheduling in Cloud Manufacturing environments characterized by random 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. [**?**]

However, reinfocement 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, 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 provene human strategies, hence I've chosen to use Deep Unsupervised Learning 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. 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 (discussed in Section 2.2.2).

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. 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 scheduling problems, this translates into the ability to model complex interactions across the entire planning horizon. 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 (often LSTM-based), 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 explicitly solves the fixed vocabulary problem: the model selects which station to place next directly from the available input pool. This significantly reduces the risk of repeating or hallucinating items, a common failure mode in standard Seq2Seq models [7].

3. Methodology

3.1 The Heuristic Approach

The problem to properly order manufacturing assembly lines with as few overlaps as possible is considered an NP-Hard problem, as it is an optimization problem.

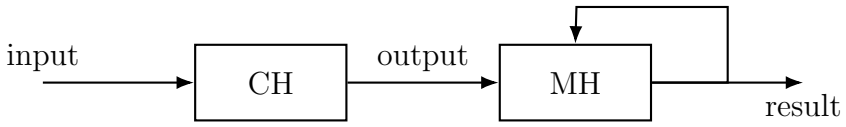

Figure 3.1: Heuristic solution

The Algorithm designed to solve this problem is a heuristic solution that will be made out of a Construction Hieuristic (*CH*) that produces a starting point based on pre-defined constraints, that feeds into a Meta Heuristic (*MH*) that finds a better solution starting from the output of the construction heuristic and self-improving until an acceptable result is returned.

3.2 Complementing the Heuristic Approach using Machine Learning

Due to the fact that management workers today place the items manually using tacit knowledge that they have accumulated over the years, and in some cases can tell at first glance if a sequence may create problems, then what this thesis proposes is to emulate that exact same knowledge by learning which placement patterns tend to work together and which do not.

The idea is that; if these workers have knowledge of an adequate solution from the get-go with some risk of overlap, then we can train a Deep Learning Model (*ML*) on such previous data to give the algorithm a better starting point, thus (in theory) reducing the runtime of that algorithm.

However it is worth to consider that such an approach can prove redundant or yield worse results if the problem at hand is an ”easy” problem where many solutions can be

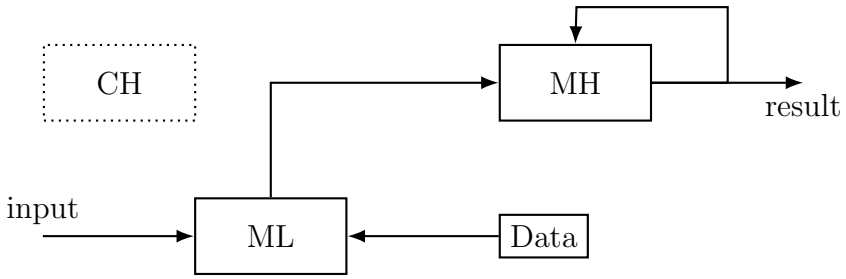

Figure 3.2: ML solution

found quickly, as opposed to a "hard" problem where a desired solution may not even be found.

3.3 JSON to Vector Methodology

Machine learning models operate on numerical vector data, often represented as *tensors*, rather than on raw JSON structures. Given a list of JSON objects on an assembly line, each JSON can be encoded into a fixed-length vector x_t by extracting and normalizing its features. This process transforms the entire list into a sequence of vectors $[x_1, x_2, \ldots, x_N]$. Using sequences from historical data, a model can then be trained to learn a mapping from an input order of JSONs to a desired output order.

3.4 Transforming a language-based model into a sequence based model

The models examined in this thesis were originally designed for natural language processing tasks, such as translation between languages. Both sequence-to-sequence and transformer-based architectures rely on a predefined vocabulary. This raises the question of whether a list of products can be treated analogously to a collection of words where each product variation forms a distinct element within the vocabulary. Doing so could allow the model to more accurately replicate historical data, along with its flaws.

However, certain constraints must be imposed as language models often overproduce common tokens such as "the," since these words are statistically more likely to appear in most positions in any given English sentence. To avoid this bias, the model must instead operate on genuine rearrangements or permutations relevant to our specific application.

3.4.1 Workarounds

To adapt the Transformer architecture for product sequencing without suffering from the previously mentioned limitations, two specific architectural modifications were im-

plemented:

Hybrid Continuous-Discrete Embedding Standard NLP models map a finite set of words to static vectors. Since product variations (specifically size and weight) are continuous rather than discrete, a traditional vocabulary approach would require an infinite vocabulary size to capture every possible variation.

To resolve this, the implementation utilizes a hybrid *ObjectEmbedding* layer. The discrete station identifier is mapped via a standard lookup table, while the continuous size value is projected through a linear layer. These two feature sets are concatenated to form the input vector:

$$E_{input} = \text{Concat}(\text{Embedding}(ID), \text{Linear}(Size))$$

This allows the attention mechanism to process the semantic identity of the station alongside the precise magnitude of the load, enabling the model to generalize to unseen product sizes without expanding the vocabulary.

Decoupling Prediction from Sequencing To strictly prevent the repetition of tokens common in generative language models, the sequence generation is decoupled from the neural network entirely.

The Transformer is configured as a standalone Encoder, serving purely as a regressor to predict the behavioral characteristics (drift offsets) of the items. It does not output a probability distribution over the next token. Instead, the final reordering is performed by a combinatorial optimizer (Simulated Annealing) which utilizes the Transformer's predictions to minimize a conflict cost matrix:

$$\text{Cost}_{ij} = \sum(\max(0, |\text{Drift}_i| + |\text{Drift}_j| - \text{Limit}))^2$$

By using the Transformer solely for high-fidelity feature extraction and delegating the sorting logic to a mathematical heuristic, the system ensures that the output is always a valid permutation of the inputwhich will guarantuee that no items are duplicated or omitted.

4. System Architecture

4.1 Machine Learning Models

4.1.1 The Transformer Model

TODO

4.1.2 The Pointer Network Model

TODO

4.1.3 The Dataset

TODO

4.1.4 The Training Loop

TODO, (see 3.4.1)

4.1.5 The Configuration

4.1.6 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.

4.1.7 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.

4.1.8 Bruh

5. Experiments and results

5.1 The Transformer Model

5.1.1 Unhooking the decoder

While testing the feasibility of the transformer model, I found that it yielded better results by switching to a BERT-like model rather than a Sequence-to-Sequence based approach.

5.1.2 Task Performance

Table 5.1: Comparison model performance in overlap reduction for $S = e^3$ (TODO)

Training Size	**Overlap Ratio (R)**	
$1e^3$	0.45	110
$1e^6$	0.62	125
$1e^9$	0.78	140
$1e^{12}$	**0.85**	138

Table 5.2: Comparison model performance in overlap reduction for $S = e^6$ (TODO)

Training Size	**Overlap Ratio (R)**	
$1e^3$	0.45	110
$1e^6$	0.62	125
$1e^9$	0.78	140
$1e^{12}$	**0.85**	138

Table 5.3: Comparison model performance in overlap reduction for $S = e^9$ (TODO)

Training Size	**Overlap Ratio (R)**	
$1e^3$	0.45	110
$1e^6$	0.62	125
$1e^9$	0.78	140
$1e^{12}$	**0.85**	138

5.2 The Pointer Network

5.2.1 Adding the offset handler

While the Transformer model used a greedy algorithm for adjusting offsets on runtime, the same idea was applied to the pointer network. (See [TODO])

5.3 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 dummy dataset was generated for simulating this task, where borrowing and tighter fits are more commonplace than actual production data. (At least from what we saw from it)

5.4 Evaluation Metrics

To objectively measure the performance of the model, we monitor performance of the models in overlap reduction (see 1.6) using a simple ratio. The ratio is:

$$\text{ratio} = \frac{O_{predicted}}{O_{source}} \tag{5.1}$$

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.

5.5 Hardware Limitations

While training the models on a larger set of data (as Transformers generally perform better with more training data) I ran out of available memory so the size of the training data had to be restricted to 500,000 objects at most.

5.6 Runtime Analysis

6. Discussion

6.1 Interpretation of Results

6.2 Limitations

As is the case in many machine learning projects the lack of data to train on led to the downfall of this project.

6.3 Future Work

Bibliography

[1] J. Abbasi, *Predictive Maintenance in Industrial Machinery using Machine Learning*, Master's thesis, Luleå University of Technology, Department of Computer Science, Electrical and Space Engineering, 2021.

[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] J. Lindén, *Understand and Utilise Unformatted Text Documents by Natural Language Processing Algorithms*, Master's thesis, Mid Sweden University, Department of Information and Communication Systems (IST), Spring 2017.

[4] I. Pointer, *Programming PyTorch for Deep Learning, ISBN: I pointer deep learning*, 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.