Applying Machine Learning on Sequential Data

Christoffer Lindkvist

December 1, 2025

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 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.

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

Chapter 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.

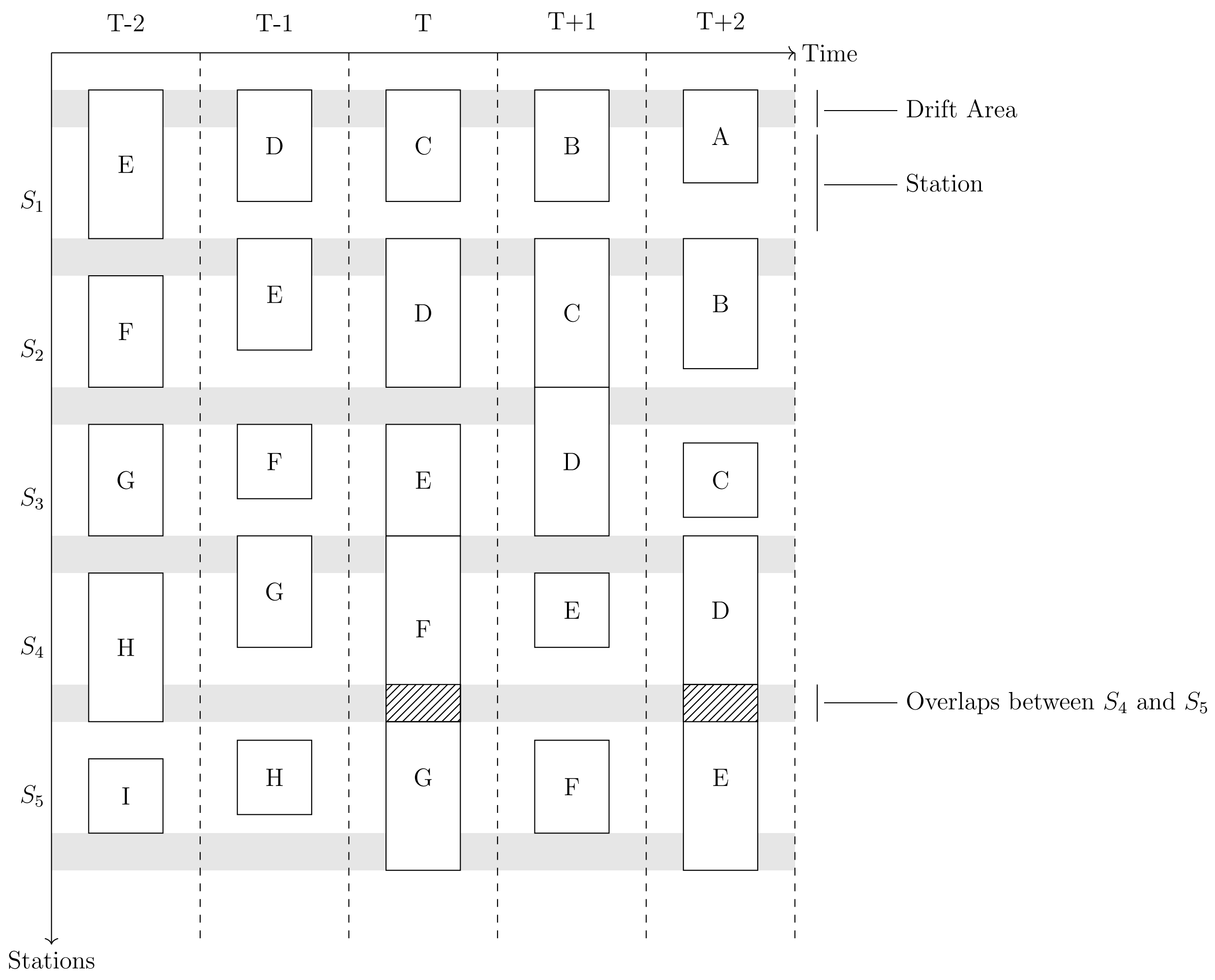


Figure 1.1: Assembly Line Example with Uniform Station and clockcycles

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 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.

Chapter 2

State of the art analysis

In this analysis I examine several deep-learning models with respect to their ability to perform permutation-based tasks, and evaluate whether training these networks on tensors with the shape...

```
[batch_size, stations, 2]
```

...can yield desired outcomes. Due to the fact that the majority of these model architectures originate from Natural Language Processing, the recurring challenge will be their tendency to genereate repeat tokens rather than producing proper permutations of the input sequence. This issue can be mitigated through the usage of penalty-functions in training, and input masking dependent on previously predicted tokens.

Permutation-based prediction tasks introduce constraints that standard Natural Language Processing architectures are not explicitly designed to handle. The central mismatch lies in the use of a fixed vocabulary. Typical NLP models operate by mapping inputs and outputs to discrete token indices drawn from a predefined vocabulary, which suits language modelling but not tasks where the output must be a one-to-one transformation of the input sequence. In permutation problems, each output element should refer directly to an input element rather than to a token from a learned vocabulary, effectively removing the need for a vocabulary altogether. This distinction highlights why conventional sequence models struggle with such tasks without previously mentioned mechanisms such as masking or pointer-based attention.

2.1 LSTM

In previous works addressing similar scheduling problems, researchers have applied Recurrent Neural Networks (RNNs), often with Long Short-Term Memory (LSTM) units, in a sequence-to-sequence (Seq2Seq) framework. Seq2Seq models are traditionally used in language-based tasks such as machine translation or text summarization, but the same architecture can be adapted to the assembly line problem.[2] Each item can be represented as a vector, and a day's worth of incoming items forms an input sequence of vectors. The Seq2Seq model can then be trained to output a corresponding placement sequence, effectively learning to replicate the tacit knowledge of the management workers in arranging the production items to minimize overlaps.[3]

The interesting part about LSTMs is that they can selectively forget irrelevant or outdated information through their forget gate. This helps them focus on more relevant patterns in the data over time, improving their ability to model long-term dependencies. [4]

Despite its age LSTM models are still in use today, and a common issue in LSTM-based models is that each entry is handled one-by-one in sequence, rather than in parallel, which increases the training time, and performance in general. This problem is addressed in the creation of the Transformer model, using an Attention mechanism which handles the sequence in parallel, and can differentiate words based on their position in sentences.

2.2 Seq2Seq / Sequence to Sequence Models

Seq2Seq models, commonly based on encoder/decoder architectures of RNNs such as LSTMs, are designed to transform one sequence into another using a prebaked vocabulary, hence the name. Usually these are found in language translation models, but can be as applicable in this thesis as we want to take one sequence of items and transform it into another. In our case, the input sequence represents the original order of vectorized JSONs, and the output sequence represents the target (reordered) sequence. However it may prove tricky to dynamically define a vocabulary from input data unless it solely looks at the sequence number. [2][3] This approach allows the model to learn patterns in reordering and can assist the rearrangement process for new data when the tacit knowledge is emulated.

2.3 Transformer-based Approaches

Similarly to Seq2Seq models, Transformer models are used for Natural Language Processing as well. However, while Seq2Seq models have been effective for sequence learning tasks, they suffer from limitations in handling very long sequences due to vanishing gradients and sequential processing bottlenecks. Transformer-based architectures address these shortcomings by discarding recurrence entirely and instead relying on a self-attention mechanism as an alternative to the sequential approach.

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

For scheduling problems, this could translate into the ability to model complex interactions between items across the entire planning horizon. For example, the placement of one item can be directly conditioned on all others in the same day's sequence, not only its immediate neighbours. Such encompassing context awareness can prove particularly valuable in assisting assembly line scheduling. [8]

2.4 Pointer Networks

Pointer Networks is the only model in this analysis which are specifically designed for assisting in combinatorial optimization problems, and sequence permutations. Encoding and Decoding is made using an LSTM architecture, similar to the Seq2Seq model, but with an Attention-mechanism similar to the Transformer network. The attention mechanism allows the model to generate context vectors that point to specific elements from an input sequence, hence the name. By explicitly selecting elements from a fixed vocabulary rather than generating tokens, the Pointer Network runs a lower risk of repeating tokens which otherwise prove a tricky problem to mitigate for the other models listed above. [7]

Chapter 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.

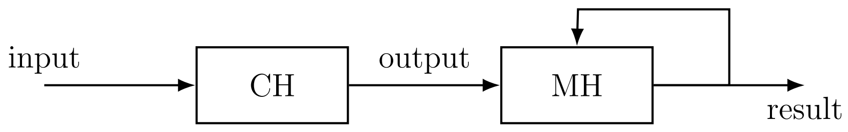


Figure 3.1: Heuristic solution

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, then what this thesis proposes is to emulate that same knowledge by learning which placement patterns tend to work together and which do not.

The idea is that if they 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.

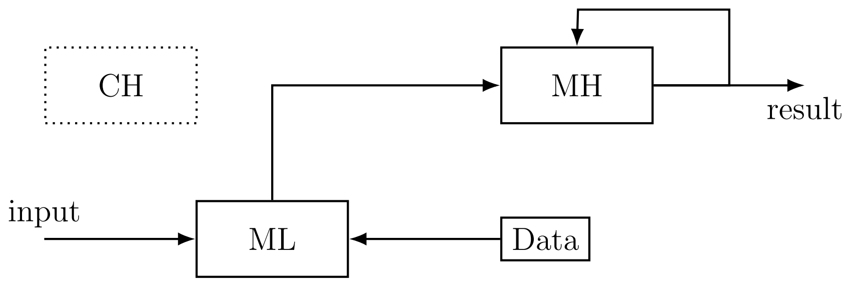


Figure 3.2: ML solution

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 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 System Architecture

The system will be implemented using PyTorch. Historical assembly line data will serve as input to the machine learning model, but this data must first be vectorized, as most ML frameworks, including PyTorch, operate on numerical vectors rather than structured formats like JSON.

The model architecture is inspired by sequence-to-sequence (Seq2Seq) models, which are commonly used in language processing tasks, such as machine translation. However, the traditional Seq2Seq architecture must be adapted to handle mixed-model sequencing rather than natural language. One challenge is that Seq2Seq models have a tendency to repeat "safe" tokens; in the context of a Mixed-Model Assembly Line (MMAL), this could manifest as repeatedly placing the easiest product in the sequence. To address this, constraints will be added to prevent repetition and encourage diverse permutations.

Alternative architectures, such as Transformer networks or Pointer Networks, will also be explored to determine the most effective approach for sequencing tasks.

To evaluate model performance before using historical data, a set of synthetic "puzzles" will be created. These puzzles will be scrambled product sequences with known solutions, allowing the model to learn placement strategies in a controlled environment. After successful training on these synthetic sequences, the model will be tested on real historical data to assess its ability to generalize to practical assembly line scenarios.

Finally, once trained, the model parameters will be stored to avoid retraining for each new sequence, ensuring the system can provide predictions efficiently during production.

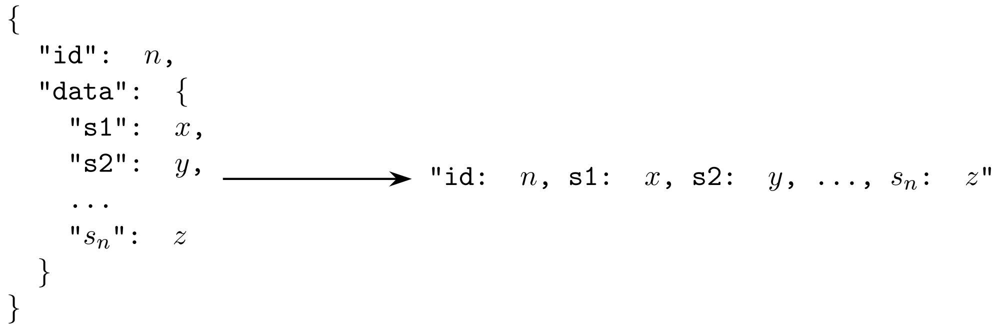
3.5 Transforming a language-based model into a sequence based model

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.

3.5.1 The Tokenizer

As one may expect when working with models designed for language, we find that most ready-made tokenizers are made to handle words in natural sentences rather than json-data or vectorized data. One way to circumvent this hurdle is to simply flatten the JSON data so that it becomes a sequence of numerical data – a string!



```
{
  "id": n,
  "data": {
    "s1": x,
    "s2": y,
    ...
    "s_n": z
  }
}
```

$\longrightarrow$

```
"id: n, s1: x, s2: y, ..., s_n: z"
```

Figure 3.3: Pre-processing JSON data by flattening nested fields.

3.5.2 Training the Model

Most of the models explored in this thesis are traditionally applied to translation tasks, where the goal is to map text from one language to another. The training data in such a case might look like:

[sv: ”Jag glömde min plånbok”, en: ”I forgot my wallet”]

Which will allow the model to learn how a sentence in the source language corresponds to one in the target language.

In our case, the challenge is different as we're not dealing with linguistic sequences or semantic transformations. Instead, our objective is to learn how to *rearrange* items rather than generating new content.

If we take a look at how Large Language Models are trained, we see that they are exposed to vast amounts of text data and learn statistical relationships between tokens through iterative training. By analogy, we can train a model on large sets of structured examples to learn how to map one ordering of items to another. With appropriate constraints or loss functions to enforce rearrangement behavior, this approach provides a viable framework for our training objective.

3.5.3 The Training Loop

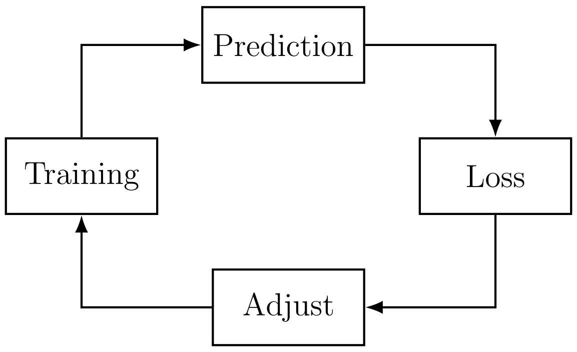


Figure 3.4: Training Loop

Chapter 4

Experiments and results

4.1 Dataset

We did get an official dataset from a production run, the dataset however contained a few issues that had to be resolved.

4.1.1 Fitting the data

The data we got contained everything we needed to visualize it on a timeline similar to Figure 1.1 where timeslots had to be manually offset to better fit between allocations. The algorithm to fit the alloctions was: [blablabla] with the following criteria: [blablabla]. The reason that the data needed additional processing was due to the fact that some allocations exceeded the given 700 cmin window, and needed to borrow time from neighbouring stations.

Without offsetting any timeslots roughly 2,000 overlaps were found.

After processing the data we were left with 300. Given that this is production data which supposedly ran without any issues we can assume that something is fishy.

4.1.2 Custom Loss Function

In the training loop used for this thesis a custom loss function was used that penalized overlaps. In the first attempts of training using 10 epochs, we reduced the number of overlaps in our shuffled dataset by roughly $20 percent$ before processing. What I did wrong was having it train on offsets as well rather than just the size of the data, as the offsets would need to be recalculated after predicting as they would else not properly align to their new locations. Thus we need to incorporate the recalculation of offsets into the training loop in order to properly spot and punish violations such as overdrafts and overlaps.

4.2 Evaluation Metrics

4.3 Runtime Analysis

Chapter 5

Discussion

5.1 Interpretation of Results

5.2 Limitations

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

5.3 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] Author, Title, Journal, Year.