Department of Industrial Engineering

Senior Design Project

Machine Learning Based Scheduling

Supervisors
Zeki Caner TAŞKIN
Ali Tamer ÜNAL

Project Team
Hasancan Cebeci – 2019402036
Ahmet Çapar - 2019402183
Halil İbrahim Türkmen - 2016402006

Table of Contents

Abstract

Scheduling is a repetitive process that almost every organization commits to make whether daily, weekly or any other period. Besides, while a schedule is started to be implemented, organizations face real life unpredictabilities, which results in a need for rescheduling. Scheduling and rescheduling also requires some time to make, depending on the mathematical model used, this time can be long and can be counted as idle time, which reduces productivity. Therefore, the aim of this project is to reduce the time wasted for

(re)scheduling efforts by shifting the focus from (re)scheduling to dispatching and by implementing a machine learning model that is trained offline with past optimal schedules and, in result, benefitting the fast online execution of Machine Learning models.

Introduction

Scheduling is the process of assigning resources to tasks over specific time periods to achieve one or more objectives. This process is crucial in many industries, including manufacturing, healthcare, transportation, education, and services. For example, in manufacturing, scheduling helps to assign jobs to machines in an order; in healthcare, it organizes patient appointments while considering medical staff availability; and in transportation, it helps plan vehicle routes. In this paper, we will use manufacturing terminology, where resources are called machines and tasks are referred to as jobs. It's worth mentioning that these terms can vary depending on the industry or context.

Scheduling is a key topic for Industrial Engineers and can be broadly described as solving an assignment problem. Essentially, this means figuring out the best way to assign resources to tasks to meet objectives like minimizing costs, maximizing throughput, or meeting deadlines. **A schedule** represents one specific way of arranging tasks out of all the possible sequences. These sequences include every potential order of jobs (all possible permutations), but only a few (or just one) will satisfy the desired objective function. *Appendix Table 1* provides an overview of common objective functions used in scheduling. Note that there may be one or more goals that need to be achieved simultaneously. Generally, there are two main methods to solve scheduling problems: optimization models and heuristics.

Optimization models are designed to find the exact best solution to a scheduling problem. These models typically use mathematical programming methods, such as mixed-integer programming, and work well for problems that can be solved within polynomial time. However, many real-world scheduling problems are considered NP-hard or NP-complete. This means that as the problem size increases, finding the exact best solution becomes computationally unrealistic.

Heuristics, on the other hand, offer a more practical approach by providing near-optimal solutions within a reasonable amount of time. While heuristics don't guarantee the exact best solution, they target specific problems and can deliver high-quality results efficiently. Some heuristics are even capable of finding optimal solutions. For example, the Earliest Due Date (EDD) rule is optimal for minimizing the maximum lateness (L_{max}) in single-machine scheduling environments. This means that if jobs are ordered by their due dates, the resulting schedule will have the smallest possible maximum lateness among all jobs. However, most heuristics are designed to provide approximate solutions. Table 2 gives an overview of different heuristic methods. [1]

***Table 2**: Heuristic Approaches in Scheduling*

Category	Heuristic Type	Description	Examples
Rule-Based Heuristics	Simple, static rules for decision-making	These heuristics rely on predefined rules to make decisions.	First Come, First Serve (FCFS): Schedule tasks in the order they arrive. Shortest Job First (SJF): Prioritize tasks with the shortest processing times. Earliest Due Date (EDD): Sequence tasks to minimize maximum lateness.
Adaptive or Feedback-Based Heuristics	Dynamic adjustment using feedback	These heuristics adapt based on feedback to improve scheduling performance.	Greedy Algorithms: Iteratively build a solution by selecting the best local option at each step.
Simulation-Based Heuristics	Use of probabilistic or iterative strategies	These heuristics explore the solution space through probabilistic or iterative methods.	Genetic Algorithms: Use principles of natural selection to evolve solutions over iterations. Particle Swarm Optimization: Simulate the behavior of a flock of birds to optimize solutions.

Machine learning-based models are increasingly being applied to scheduling problems, though this area hasn't been studied as much as optimization models or heuristics. In general, machine learning models are great at identifying patterns and relationships in historical data, which makes them applicable for dynamic and complex situations. In the context of scheduling, machine learning approaches often involve architectures like reinforcement learning models, recurrent neural networks (RNNs), convolutional neural networks (CNNs), and graph neural networks (GNNs). Once these models are trained, they provide significant benefits, such as very short execution times during the online phase. This feature helps solve many of the challenges typically faced in scheduling.

Table 3 summarizes the methods to solve scheduling problems.

	Optimization	Heuristics	ML Based
Duration of Execution	Exponential Time	Polynomial Time	Faster than both
Approach	Algorithmic	Rule-based or Iterative optimization-based	Data-driven and predictive
Required Time Before Execution	No time required	No time required	Requires Training
Optimality	Exact	Near-optimal	Near-optimal with generalization
Scalability	Limited for large problems	Handles larger problems	Highly scalable post-training

***Table 3**: Comparison of Methods*

Once a scheduling problem is solved, the schedule found is often shown onto a Gantt Chart. Gantt Chart includes two axis, horizontal one shows time horizon and the vertical one shows the available resources/machines. From the Gantt Chart, the start and end times of tasks, resource usage and the sequence of tasks can be read. A Gantt Chart not only indicates when tasks begin and end but also illustrates which machine is handling which job and for how long. Additionally, it shows the precedence relationships between jobs so that tasks are carried out in the correct order based on their dependencies. Below, in Figure 1, an example of a Gantt Chart can be seen.

***Figure 1**: Example of Gantt Chart*

Scheduling is the first (and arguably the most crucial) step in production operations. This step is essential because how well tasks are completed on time and how consistently due dates are met have a direct impact on an organization's reputation and its ability to stay competitive. In general, it can be said that scheduling is an ongoing process. In other words, as long as an organization continues to operate or take on new tasks, it will always need to engage in scheduling. This need exists regardless of what type of mathematical model they use, which industry they are in, or which objective functions they use. Each time new tasks arise (typically these tasks are aggregated on a daily or weekly basis and often including

carryovers from previous periods) the organization must decide how to allocate its resources and organize these tasks over time.

Problem Definition

As long as an organization continues its operations, it needs to schedule its waiting jobs on a daily or weekly basis (as discussed in the previous section). During these periods, the organization goes through what we name it as the general scheduling cycle, shown in *Figure 2.1*.

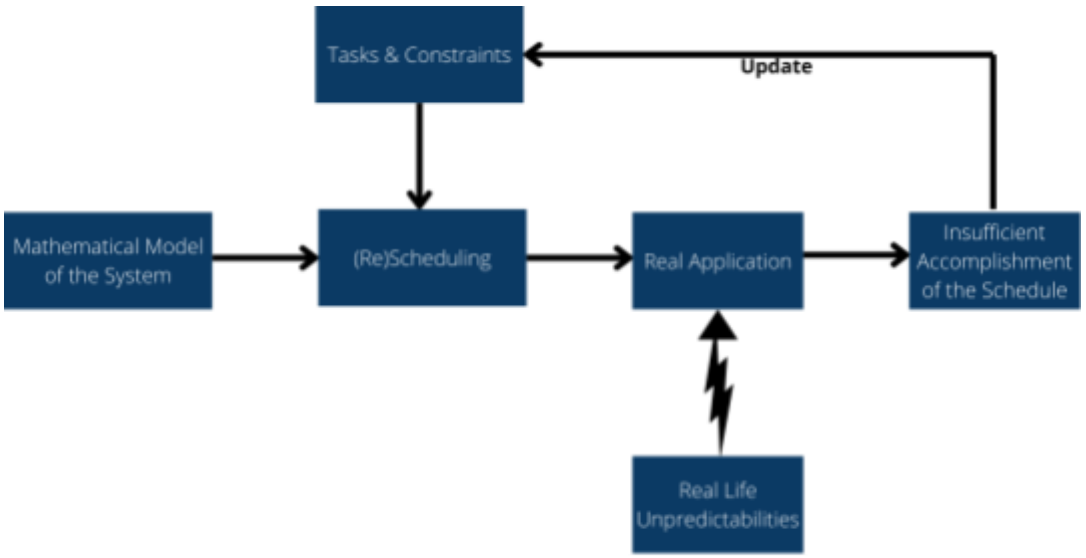


***Figure 2.1**: General (Re)Scheduling Cycle*

The process starts with the organization developing a mathematical model that represents its production system. This model defines resources and processes and is constructed on the day they decide to run the organisation, and is not changed frequently by the decision makers. On any operation period, specific tasks, their constraints, and other details are added to the model, which is then executed to have a schedule. Once the schedule is created, it moves to the implementation stage.

However, during the implementation, external noise interferes with the schedule. This can be referred to as *real-life unpredictability*, which may include:

- The arrival of new, urgent tasks that require immediate processing,
- The cancellation of one or more jobs,
- Changes to the due dates of certain jobs,
- Processing times for some jobs taking longer than expected,
- Machine breakdowns, and many other unpredictable situations.

Such unpredictability makes it difficult (and sometimes impossible) to follow the originally created schedule. As conditions and current time change, the schedule often loses its optimality/near-optimality. These disruptions force the scheduler or organization to update

tasks and constraints, which leads to the need for creation of a new schedule. We refer to this process as *rescheduling*.

In the *general (re)scheduling cycle*, sections beyond the mathematical model of the system fall within the scope of operational planning. This means that scheduling or rescheduling needs to be quick and responsive because it is on the operation level. However, when optimality is prioritized, optimization models are often chosen over heuristics. But it's well known that most scheduling problems are NP-hard, which means that the complexity of solving them grows exponentially as the number of jobs increases. As a result, it becomes practically impossible to generate optimal solutions within a short time. Therefore, our project arguments are:

1. Scheduling is a repetitive process that organizations perform daily or weekly.
2. This repetitive process inherently includes the *general rescheduling cycle*.

In other words, organizations often create optimal or near-optimal schedules that they are unable to follow and that require significant time to construct.

Proposed Solution

To overcome these challenges, our project suggests a shift in focus from rescheduling to **dispatching**. The key idea is to dynamically determine only the next job to process, rather than attempting to create an entire schedule. But finding the next job to do is still an NP-Hard problem and it does not reduce the complexity of the problem. For that purpose, we developed a machine learning model trained on past optimal schedules. This model is designed to dispatch jobs dynamically while benefiting from the fast online execution time of ML models. By doing that, reducing the time typically spent on scheduling and rescheduling is achieved.

Stakeholders

We have categorized the stakeholders for this project into three primary groups:

1. Organizations Struggling with Unpredictability
   This category includes organizations in industries where real life unpredictabilities are high while application of schedule. These organizations often face disruptions like unexpected delays, urgent new tasks, or resource unavailability. They require dynamic, real-time updates to adapt to these challenges while maintaining operational efficiency so our fast dispatching approach would benefit them.
2. Efficiency-Driven Professionals
   This group consists of operations managers, planners, and decision-makers who rely on heuristics or traditional methods for scheduling. These professionals often question whether their current operations are truly optimal and seek solutions to improve efficiency. By adopting machine learning models trained on past optimal schedules,

they can make more informed and reliable decisions, reducing inefficiencies and achieving higher productivity.

3. Automation Seekers
   Automation seekers are organizations and professionals looking to minimize manual scheduling efforts and reduce dependence on human intervention. They aim to achieve cost-effectiveness through just an initial investment in training machine learning models, which eliminates the recurring costs associated with manual or traditional scheduling. Additionally, they are concerned with addressing the hidden costs of inefficient scheduling, such as wasted time, financial losses, operational disruptions, and the frustration caused by enforcing impractical schedules. Therefore, reduction of such costs can be achieved via the application of our proposed solution.

Supporting Technical Teams

The success of this project also depends on the involvement of various technical teams that would implement and manage the proposed solution:

- Data Scientists and Machine Learning Engineers: They will collect historical scheduling data, design, train, and refine the machine learning model. Their job also involves monitoring how the model performs over time and updating it with new data as needed.
- Operations Managers: These professionals ensure the machine learning model is aligned with the organization's specific goals, constraints, and priorities. They also validate the model's output to confirm it meets real-world needs.
- Software Developers: They are responsible for integrating the machine learning model into existing scheduling tools or creating a standalone system with an easy-to-use interface.

By addressing the needs of these stakeholders and maintaining active collaboration among technical and managerial teams, this project has the potential to transform scheduling processes across a wide range of industries where such solutions are especially impactful.

Benchmark Problem Selection

As part of this project, we needed to create a hypothetical environment to construct and test our machine learning model. Developing a high-level, conceptual model that encompasses all scheduling problems was neither feasible within the scope of this project nor aligned with our objectives. Instead, we opted for a **single-machine environment** where the objective function is **total weighted tardiness**.

The criteria for selecting this problem were as follows:

1. **Low Data Burden**
   Since the project involves conducting large-scale experiments, the selected problem needed to ensure a manageable data burden. Machine learning models require rich training data to achieve sufficient learning, and hyperparameter tuning necessitates

conducting numerous experiments. Therefore, we prioritized selecting a problem with relatively small data dimensions. For instance, in more complex scenarios such as flexible job shop scheduling with multiple parallel machines, the data dimensions include factors like the sequence-dependent setup times and machine compatibility constraints, resulting in need for higher dimensional data representations. In contrast, the single-machine environment simplifies these dimensions by focusing on one machine and assuming no process restrictions, which significantly reduces data complexity. This made single-machine problems more practical and aligned with the project's scope.

2. **Generalizability**
   The single-machine scheduling problem is widely applicable because many scheduling problems can either:
   - **Be represented as a single-machine problem**: For example, parallel machine scheduling can be decomposed into independent single-machine problems when jobs are pre-assigned to specific machines.
   - **Be expressed as nonlinear combinations of single-machine problems**: For instance, **flow shop scheduling** and **job shop scheduling** often have components that can be reduced to single-machine subproblems in a hierarchical structure.

By focusing on the single-machine environment, we aimed to make our model adaptable to a wide range of scheduling problems. If other problems can be reduced to a single-machine format, they could also benefit from the machine learning model we developed. This approach makes the generalizability and potential utility of our solution possible across different domains.

3. **NP-Hard Nature of the Problem**
   It was essential to select an NP-hard problem to justify the use of advanced methods like machine learning. If the problem were not NP-hard, it would be computationally feasible to find optimal schedules within polynomial time. Organizations could simply rerun their optimization models each time they needed to schedule or reschedule. However, most scheduling problems are NP-hard, even for single-machine cases. For example, the single-machine total weighted tardiness problem (SMTWTP) is a classic NP-hard problem. This makes that finding optimal or near-optimal solutions is computationally hard, making it a suitable benchmark for evaluating the effectiveness of machine learning models in this context.

Based on these criteria, we selected the **Single-Machine Total Weighted Tardiness Problem (SMTWTP)** as the benchmark for our project. This problem not only aligns with our technical and practical requirements but also serves as a well-established challenge in the field of scheduling. The problem is represented by

$$1||\sum w_j T_j$$

In this representation:

- 1 denotes a single-machine environment, meaning there is only one resource (machine) available for processing all jobs.
- Within the vertical bars, there is nothing, which stands for that there are no process restrictions. This means that jobs can be scheduled without additional constraints such as precedence relationships, machine breakdowns, preemptive rules, or machine compatibility requirements.
- The objective function minimizes the total weighted tardiness, represented as $\sum w_j T_j$, where:

$T_j = max(C_j - d_j, 0)$): Tardiness is the difference between a job's completion time

(C_j) and its due date (d_j), but only if it is late (otherwise, tardiness is 0).

w_j: Weight assigned to job j, reflecting its priority or importance.

Learning Model Selection

Why Neural Networks?

Among machine learning methods, Neural Networks (NN) is chosen to solve the optimization problem. The advantages and reasons of choosing Neural Networks are as following:

- **Well Studied Technology**

One of the widely researched machine learning models are neural networks. Also, it is one of most common technologies that is used in scheduling problems.

- **Good Performance**

Neural networks are known for their high accuracy. They have proved their capability to solve complex and non-linear problems.

- **Availability of Tools And Sources**

Tools such as PyTorch and TensorFlow provide a user-friendly environment for our project. Implementation of the Neural Networks becomes easy using these tools and it allows us to focus on the problem, rather than creating a neural network from scratch. They reduce complexity of model and allows us to experiment rapidly with different architectures and configurations. Moreover, the educational materials such as videos, open source codes, research papers fasten the troubleshooting processes and implementation.

How Does Neural Networks Work?

Neural Networks are machine learning models that are inspired by the human brain structure. It consists of interconnected layers of nodes (neurons). Neurons process input data to predict output data.

Basic Components of a Neural Network

1. Input Layer
   Input layers take the raw data as input. In our problem, input layer takes inputs as processing time, due time and weights.

2. Hidden Layer
   Hidden layers transform input data and introduce non-linearity using activation functions.

3. Output Layer
   Output layer produces the final output. In our problem, this is the probability of each job being first.

Forward Pass

This process allows a neural network to produce an output. Each neuron in the hidden layer, calculates a weighted sum of its inputs, adds a bias term and applies an activation function:

$$z_j = Activation(\sum_i (w_{ij} x_i + b_j))$$

Where:
w_{ij}: Weight of neuron connecting neuron i to neuron j
x_i: Input
b_j: bias for neuron j
z_j: Result passed through activation function

Activation functions introduce non-linearity to model, allowing them to learn complex cases. For example ReLU and SoftMax. ReLU gives 0 for negative values, normal output for positive values. Softmax convert logits (raw data) to probabilities, ensuring outputs sum to 1.

Backpropagation

Backpropagation part is the process of computing gradients of output with respect to each node. Backpropagation allows the model to learn from its wrong predictions. Deviations of predictions from target values are calculated using loss functions and loss values are obtained. Common examples of loss functions are Cross-Entropy Loss and Mean Squared Error (MSE).

After calculating the loss values, chain rule is used to propagate error backward through the network. Then the weights are updated using an optimization algorithm such as Stochastic Gradient Descent (SGD) or Adam.

Data Generation

To create training and testing datasets, we generated synthetic job collections and their respective schedules based on predefined rules. The process is illustrated in *Figure 4.1*.

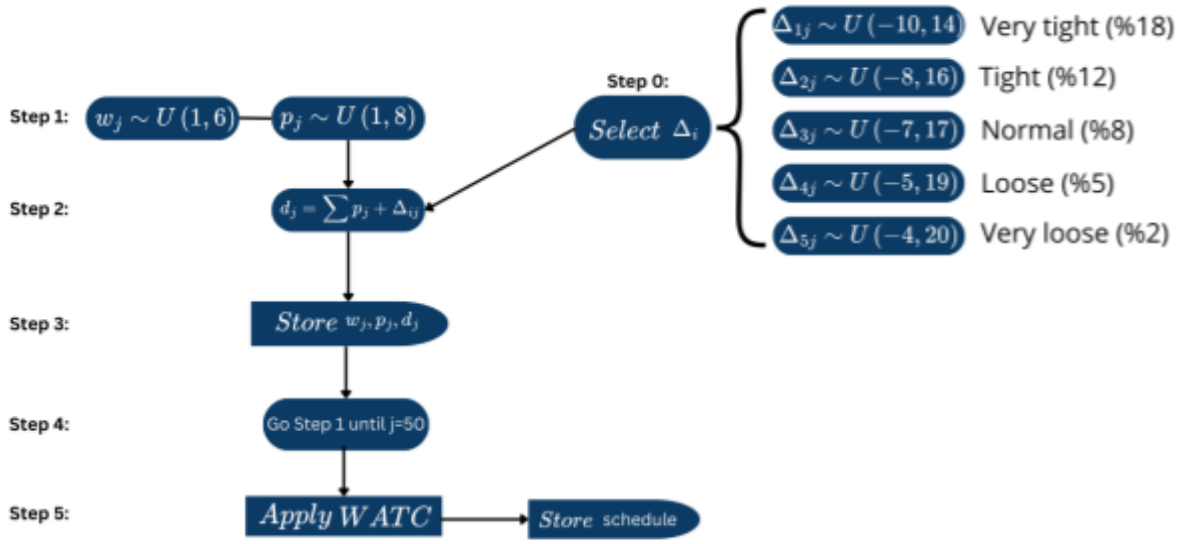


***Figure 4.1**: Data Generation Diagram*

Before running the data generation code, we defined five different delta ranges to represent various scheduling behaviors of an organization, namely, very tight schedules, tight schedules, normal schedules, loose schedules, very loose schedules. For each delta range, we generated 100,000 schedules, each containing 50 jobs. This provides sufficient diversity for training and testing our machine learning model under different scheduling scenarios. The data generation code can be accessed via this github link in *Appendix 3*, section *Data_Generation* in the repository.

Detailed Steps

Step 0:
Among five, one delta value is chosen depending on how tight the schedule is going to be. (Determination of ranges explained on Selection of Delta Ranges)

Step 1:
The process begins with assigning weights (w_j) and processing times (p_j) for each job. Both parameters follow discrete uniform distributions with specified ranges (as shown in *Figure 4.1*). *Note*: This approach is consistent with methodologies commonly used in relevant literature.

Step 2:
The due date (d_j) for each job is calculated based on cumulative processing times and a delta value (Δ_{ij}), which varies depending on the selected delta range. Here's an example for clarity:

- For Job 1, if $p_1 = 4$ and $\Delta_{11} = 3$, then $d_1 = 7$.
- For Job 2, if $p_2 = 5$ and $\Delta_{12} = -8$, then $d_2 = 1$.
- For Job 3, if $p_3 = 2$ and $\Delta_{13} = 10$, then $d_3 = 21$.

This calculation shows how due dates are cumulative sums of processing times and adjusted by the delta range values.

Step 3:
The weights (w_j), processing times (p_j), and due dates (d_j) for all jobs are stored.

Step 4:
Steps 1 through 3 are repeated 50 times to generate a complete set of 50 jobs.

Step 5:

For the generated set of 50 jobs, we applied a heuristic scheduling method known as Weighted Apparent Tardiness Cost (WATC) to create schedules. The WATC heuristic is further explained in its respective section.

Note: Although we could have used an optimization model to generate optimal schedules, this was not feasible due to resource limitations. However, using an optimization model would provide that generated schedules are optimal. It is worth noting that the optimization process is conducted offline, meaning it occurs during ML model training rather than during real-time (re)scheduling (offline procedure). For obtaining optimal schedules, we implemented a Mixed-Integer Programming (MIP) model using Gurobi to generate optimal schedules, which can be accessed via github link in *Appendix 3*, section *Single_Machine_Weigthed_Tardiness_Scheduling_with_Gurobi* in the repository. Refer to for a mathematical representation of the MIP model at *Appendix [2]*.

Selection of Delta Ranges

To simulate realistic organizational scheduling behaviors, we defined five delta ranges, each corresponding to a unique scenario. In very tight scenario 18%, in tight scenario 12%, in

normal scenario 8%, in loose scenario 5%, in very loose tight scenario 2% of the jobs are tardy. Delta ranges were determined as follows:

1. We assigned arbitrary delta values to jobs with weights and processing times to create due dates for 10 jobs. (Additionally, there is an assumption here that the percentage of tardy jobs remains consistent when scaling from 10 jobs to 50 jobs. This assumption allows us to generalize the percentage of tardiness across different job sizes while maintaining computational feasibility during the data generation process.)
2. Using the previously mentioned MIP model, we ran 100 iterations for each delta range, and we analyzed the percentage of tardy jobs. The results are shown in *Figure 4.1*.

For better understanding:

- Without a delta range, all jobs are in a steady state. In this state, their due dates have not yet been determined, but the jobs are arranged in the order they were generated. All jobs have "*zero temperature*".
- The delta range acts as "*additional temperature (can be +/-)*" imposed on each job in the sequence, introducing variability to the due dates.
- As the average delta range becomes more positive, jobs gain more "wiggle room," meaning their due dates are pushed further out which creates looser schedules. This reduces the likelihood of tardiness across jobs.

Rationale for This Approach

Our primary motivation was to emulate how organizations set due dates in practice. Because due dates are not assigned completely at random, they are typically based on average feasibility. Additionally, different organizations exhibit varying tendencies when setting due dates. By modeling this behavior, we aimed to reflect these tendencies while maintaining greater control over tardiness levels.

WATC (Weighted Apparent Tardiness Cost)

The WATC heuristic, also referred to as AU (Apparent Urgency) in the relevant literature, combines elements of the MS (Minimum Slack First) and SPT (Shortest Processing Time First) dispatching rules. The priority index for scheduling is calculated as follows:

$$I_j(t) = \frac{w_j}{p_j} e^{\left(\frac{-max\left(d_j - p_j - t, 0\right)}{K \times P_{avg}}\right)}$$

Where

- jth job slack: $max\left(d_j - p_j - t, 0\right)$
- K is scaling parameter
- Average processing time of the jobs: P_{avg}

This heuristic balances the trade-off between jobs with high weights (WATC favors jobs with high importance or penalties) and those with short processing times (minimizing overall schedule length).

For additional details, refer to [2] [3].The implementation of this heuristic can be seen in the "Data Generation with WATC" section from the github link Appendix 3.

Model Construction

During our project, we have developed two machine learning models. Main idea of the first model is creating schedules, rather than dispatching. Our second model better reflects our main idea.

First ML model

Due to time restrictions, we were worried about the long training time. Therefore, we only considered 10 jobs with reduced dimensions. The meaning of the reduced dimensions is reducing complexity of the ML Model. For that purpose, we define a parameter called a_i which is determined by

a_i =(process time of job i * due date of job *release time of job i /weight of job i)

By doing this the dimensions of the jobs are reduced to 1, which decrease our ML model's input layers to 10. By doing so, we expected training time to be lower.
There was one hidden layer which contained 50 neurons. We used Sigmoid as an activation function. The target layer consists of the optimal schedules created by our MIP model.
Training data sets include 1000 pair of a_is (coefficient of jobs) and corresponding schedules.

In the training phase, our model used Mean Squared Error loss function and as optimizer, Adaptive Moment Estimation. We had not created a test function at that time so we tested our model manually. Our model produces a sequence of 10 float numbers as an output. Then, the model uses "*argsort"* these values to determine the schedule. From the github link (*Appendix.3*) section "*First_ML_Model.ipynb*", related code can be seen.
However, we abandoned this model due to the change of our focus from scheduling to dispatching but we got familiarized with the environment, pytorch.

Final ML Model

The main focus of constructing this model is that we do not need to generate a complete schedule; we only need to find the first job to be processed. This ML model's design is centered around this idea. Furthermore, although we were initially worried about handling large amounts of data, input size, and high input dimensions, these concerns have diminished as we became more familiar with the environment and PyTorch.

First, the input layer of our model takes 50 jobs of their associated processing times, weights, due date. Therefore, input layer consists of one dimensional 50*3=150 nodes.

Input vector is in the following form:

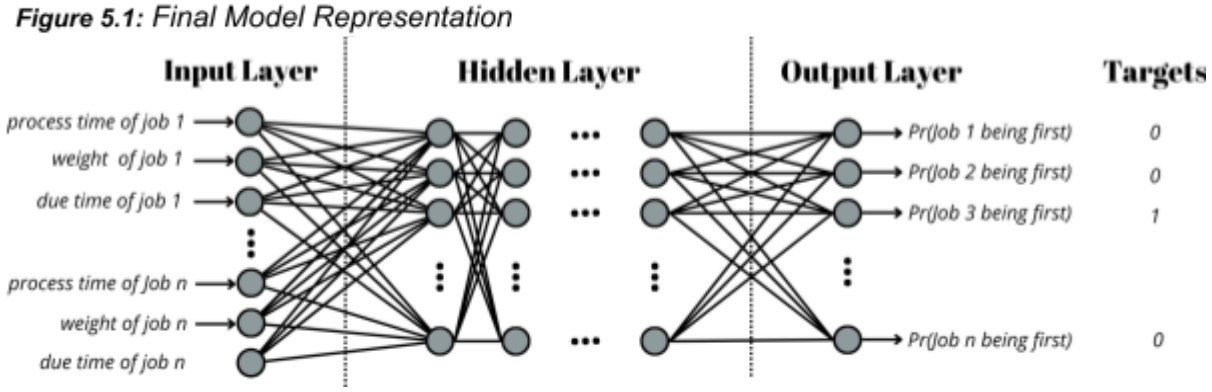
x = [Processing Time_1, Weight_1, Due Time_1, …, Processing Time_n, Weight_n, Due Time_n]

In the training period, the target layer takes a sequence of 1 and 0's. 1 represents the first job, 0's represents the other 49 jobs. (i.e if job 3 is the first job to be processed in the actual schedule, then the target layer would look like [0,0,1,0,0…]. To represent it directly, target vector v is:

(Vj = 1 if job j is the first job; 0 otherwise)

In our model we have 3 hidden layers as default. These layers include 150, 100 and 50 neurons respectively. The ANN representation of this model can be seen in the *Figure 5.1*:

Figure 5.1: *Final Model Representation*



We created datasets which contained 100.000 schedules for each scheduling behaviour as mentioned in the Data Generation section. The model trained separately for each scheduling behavior. Then, we split %80 of the dataset as training and the remaining %20 of them as test sets.

In the training phase, our model used Cross Entropy loss function and as optimizer, Adaptive Moment Estimation is used. In pytorch, as embedded CrossEntropyLoss has softmax in itself and softmax normalizes the vector and each number represents a probability of a job being the first job on the real schedule.

After the training phase, the trained model takes jobs' information from the test set.Then, output layer looks like the following array when we do not use softmax:

[35.0892, 35.3803, 34.6428, ..., -13.6928, -22.9848, 1.4545].

When we use softmax, the output looks like the following:

[1.6839e-38, 3.2630e-37, 0.0000e+00, ..., 5.7733e-43, 4.1008e-15, 1.0604e-39],

containing 50 values and their sum is 1, which can be interpreted as probabilities of each job being first to process, and the sum of the probabilities of all jobs selected as the first job to process adds up to 1. Then, for both cases (with or without softmax), we use argmax to find the most likely job to be processed first. The position of the neuron found by argmax will show the job predicted by our model.

In the testing phase, we evaluate our model performance based on the following rule; identifying jobs in the actual schedule corresponds to the model's first prediction, and calculate the average of these values across the entire test set which refers to average prediction order.

The rest of the report is valid for the Final ML Model.

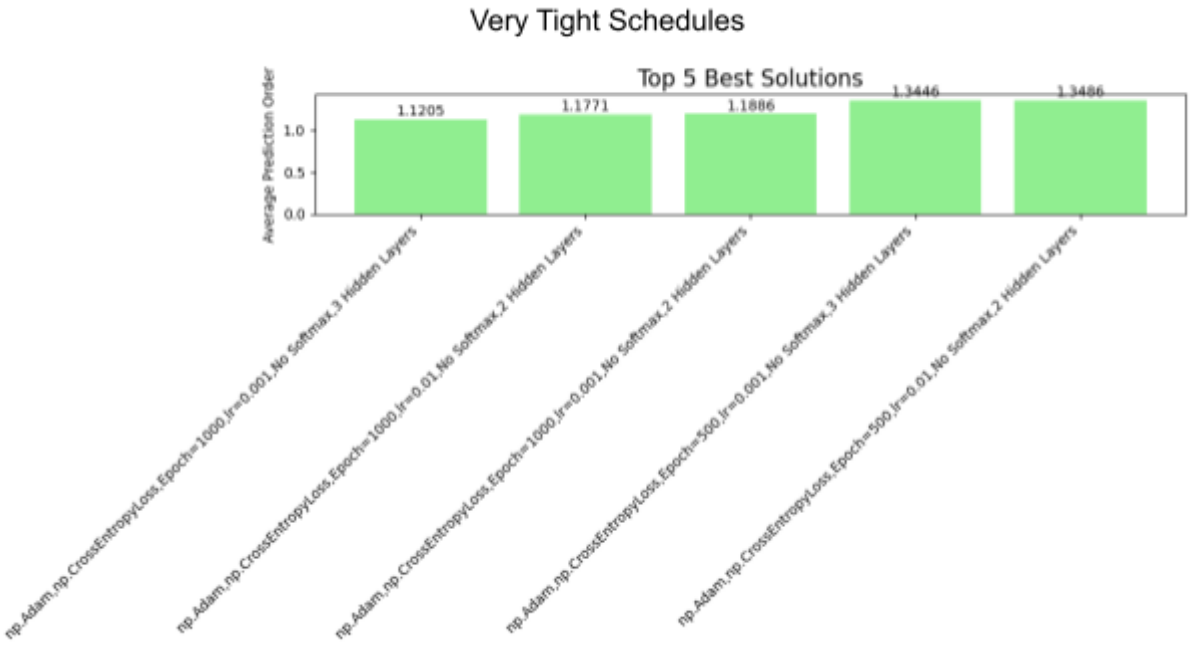
Hyperparameter Tuning

As mentioned in the data generation section, we have 5 distinct scheduling behaviors. For each scenario, we create 5 separate neural networks. To determine optimal performance of each artificial neural network, we experimented with 64 different hyperparameter combinations based on the parameters below to see which configuration works the best. The goal was to find the configuration that gives the best result.

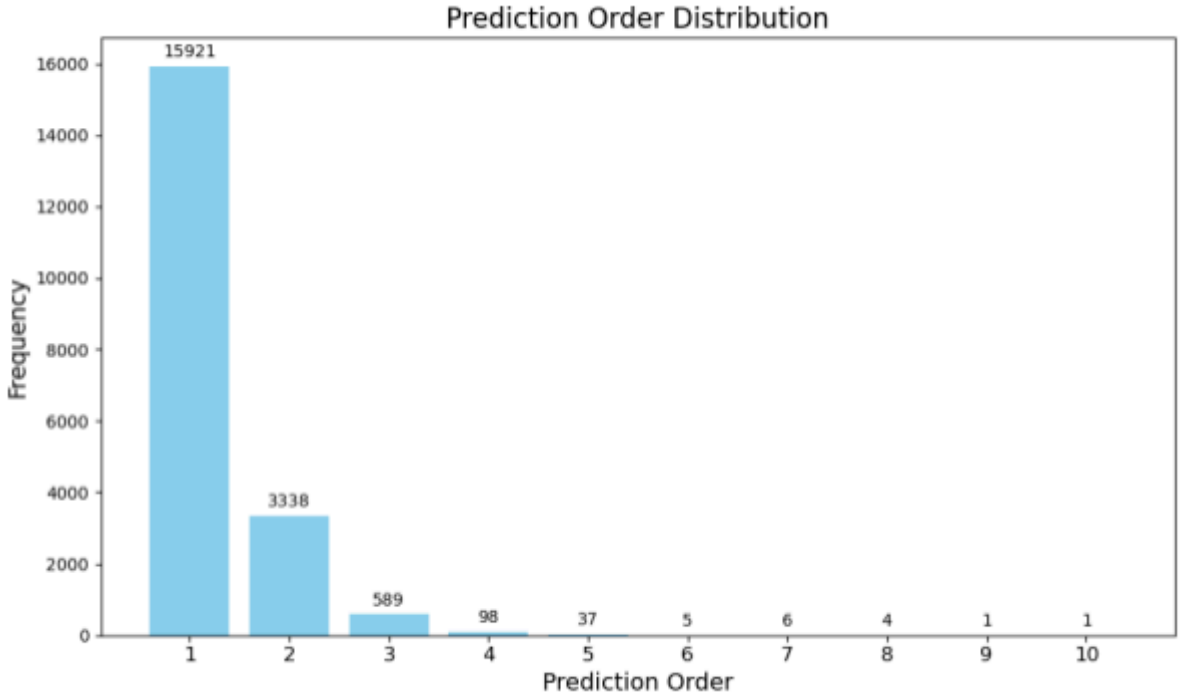
1. **Optimizer Function:** As an optimizer function we used Adam(Adaptive Moment Estimator) and SGD(Stochastic Gradient Descent)
2. **Loss Function:** For calculating the loss, we test MSELoss Function(Mean Squared Error) and CrossEntropyLoss Function of the pytorch.
3. **Epoch Number:** We trained the model with 500 and 1000 epochs.
4. **Learning Rate:** The ML model is trained with learning rates 0.01 and 0.001 seperately.
5. **Softmax:** We thought the softmax function could be efficient for predicting the first job so we tested the model both with and without Softmax.
6. **Hidden Layer Configuration:** We either use 2 hidden layers or 3 hidden layers with each having (150,100) and (150,100,50) neurons respectively.

There are countless other possible configurations as expected but we only tried what we consider as meaningful and doable within the scope of our project. There are advanced search methods developed, such as grid search and Bayesian optimization, for finding the best configurations but we do not explore them.

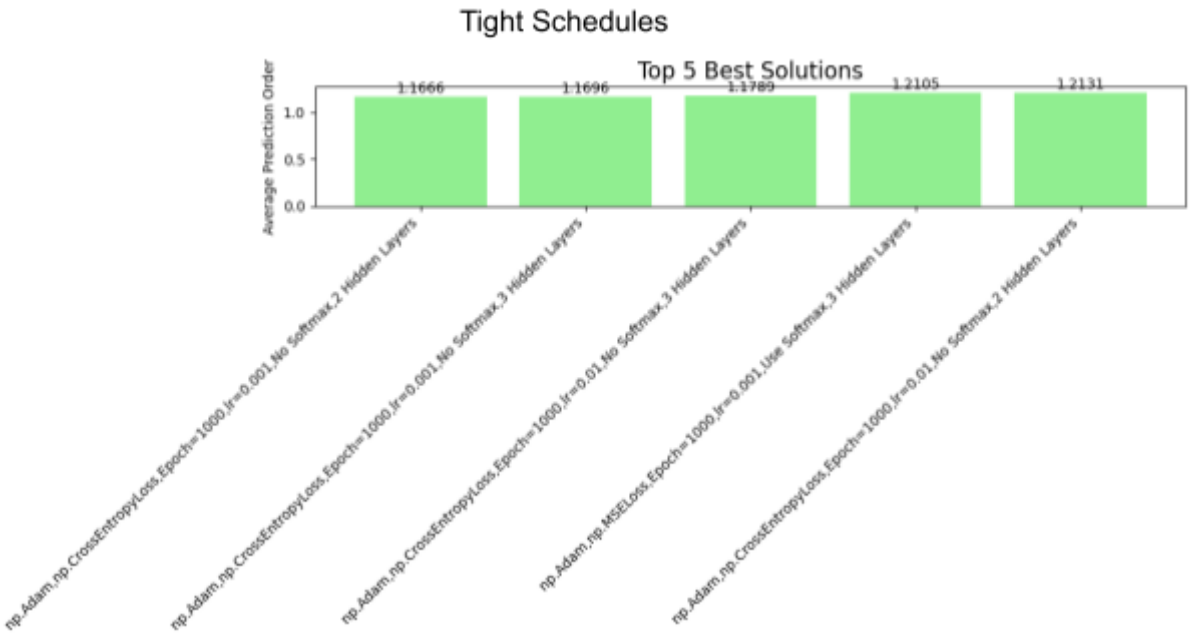
Model Performance



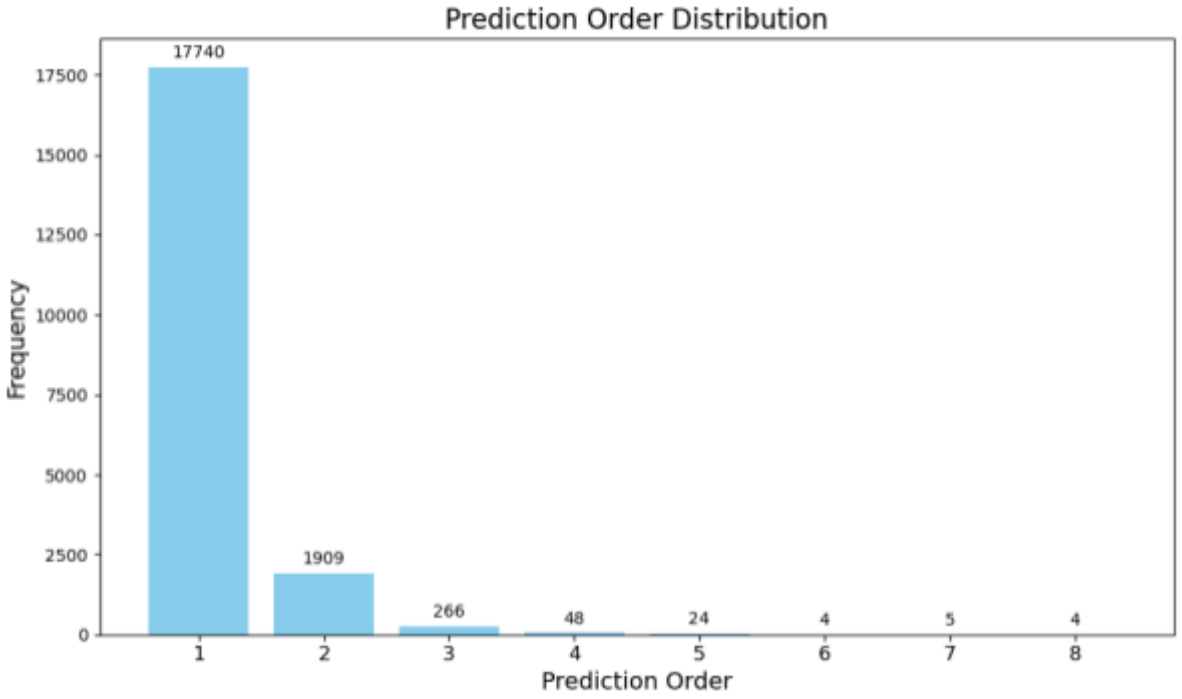
For very tight schedules, hyperparameters with (Adam,CrossEntropyLoss, 1000 Epoch, 0.001 learning rate, without softmax, 3 Hidden Layers) gives the best result. Distribution of the best configuration's predictions are as follows (If the model predicts the second job in the actual schedule as first instead of predicting as first , the plot reflects this by showing a value of two):



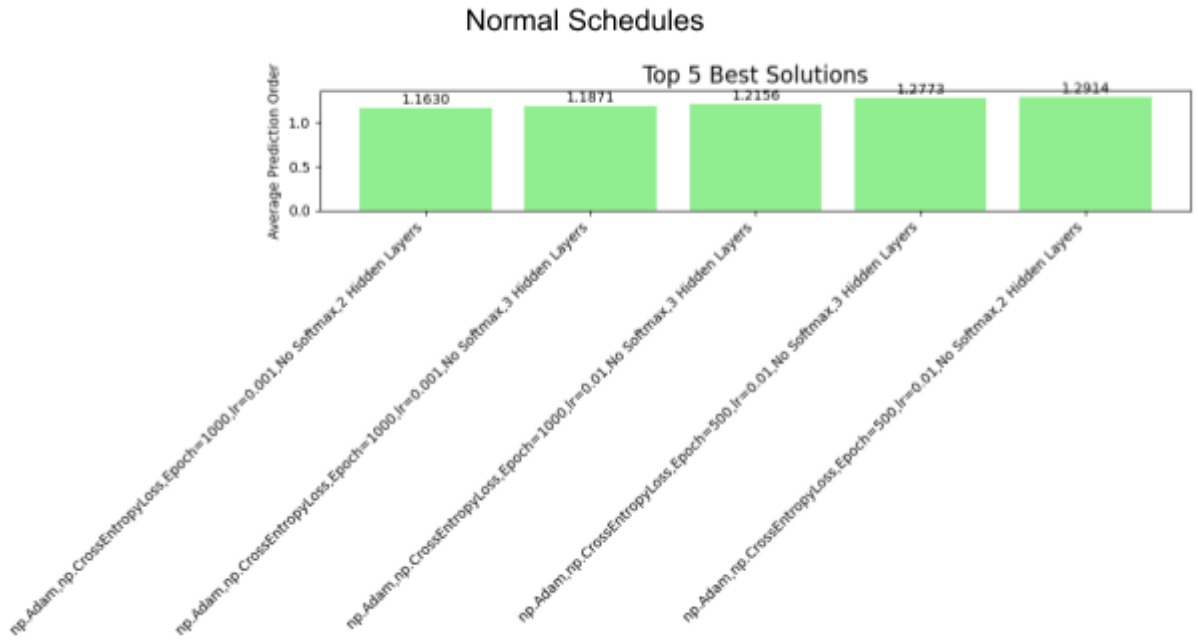
Accuracy of our model with best configuration for *very tight schedules* predicts the first job correctly is %79,6.



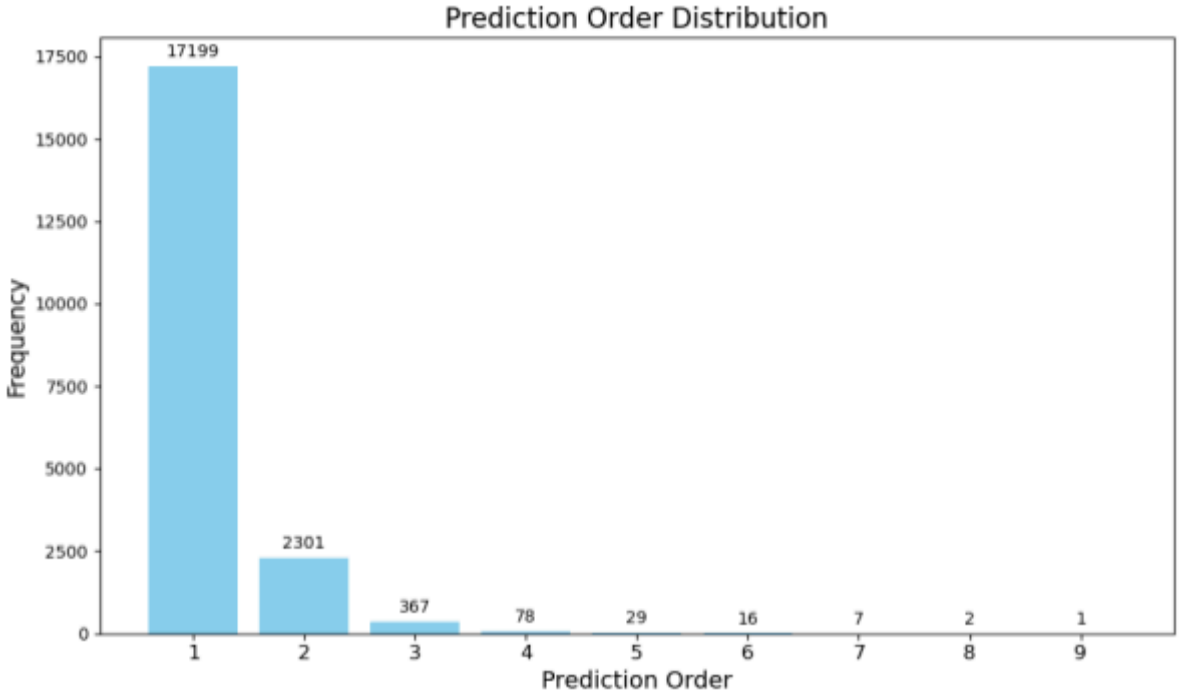
For tight schedules, hyperparameters with (Adam,CrossEntropyLoss, 1000 Epoch, 0.001 learning rate, without softmax, 2 Hidden Layers) gives the best result. Distribution of the best configuration's predictions are as follows:



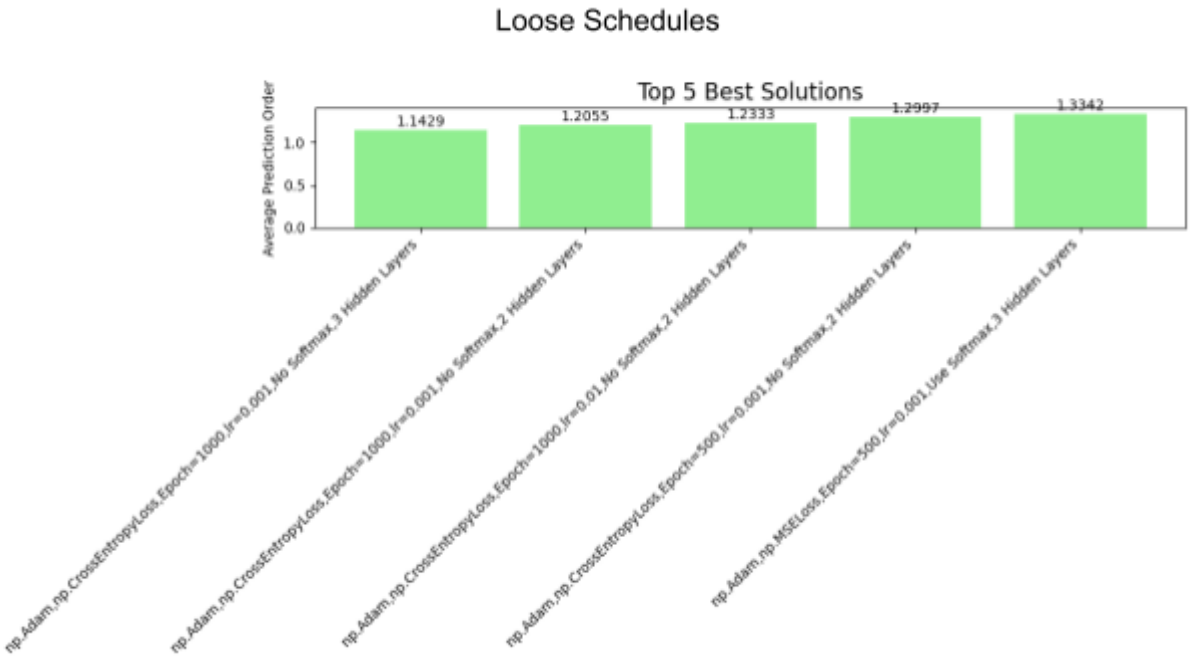
Accuracy of our model with best configuration for *tight schedules* predicts the first job correctly is %88,7.



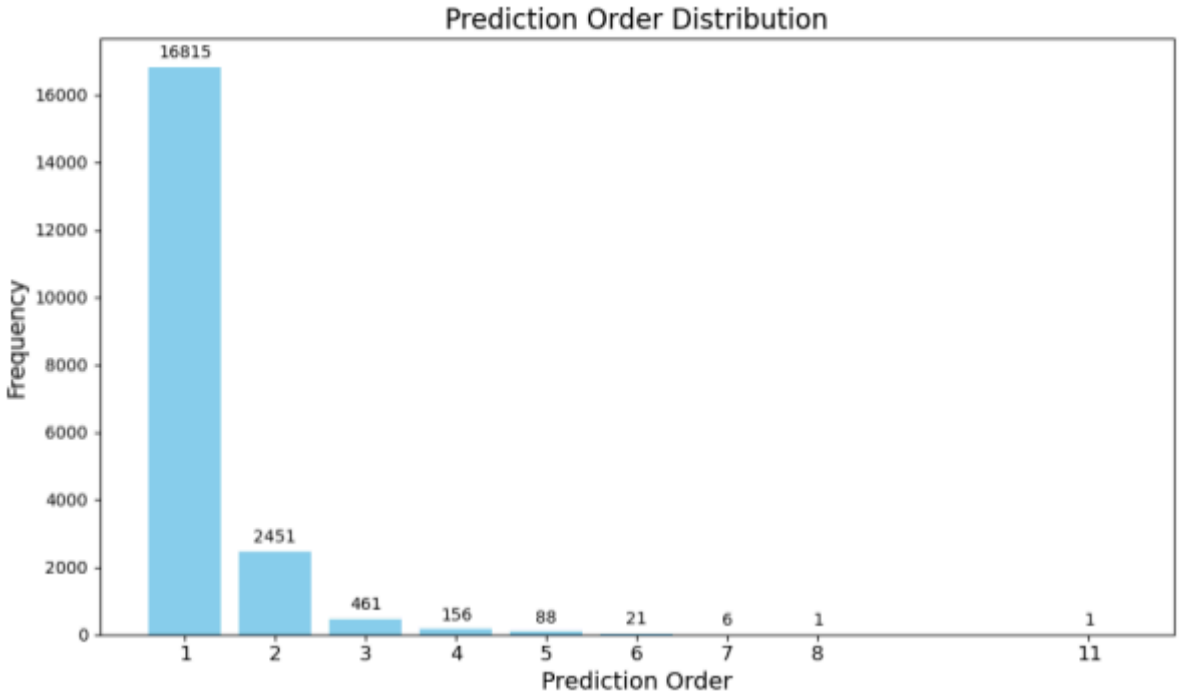
For normal schedules, hyperparameters with (Adam,CrossEntropyLoss, 1000 Epoch, 0.001 learning rate, without softmax, 2 Hidden Layers) gives the best result. Distribution of the best configuration's predictions are as follows:



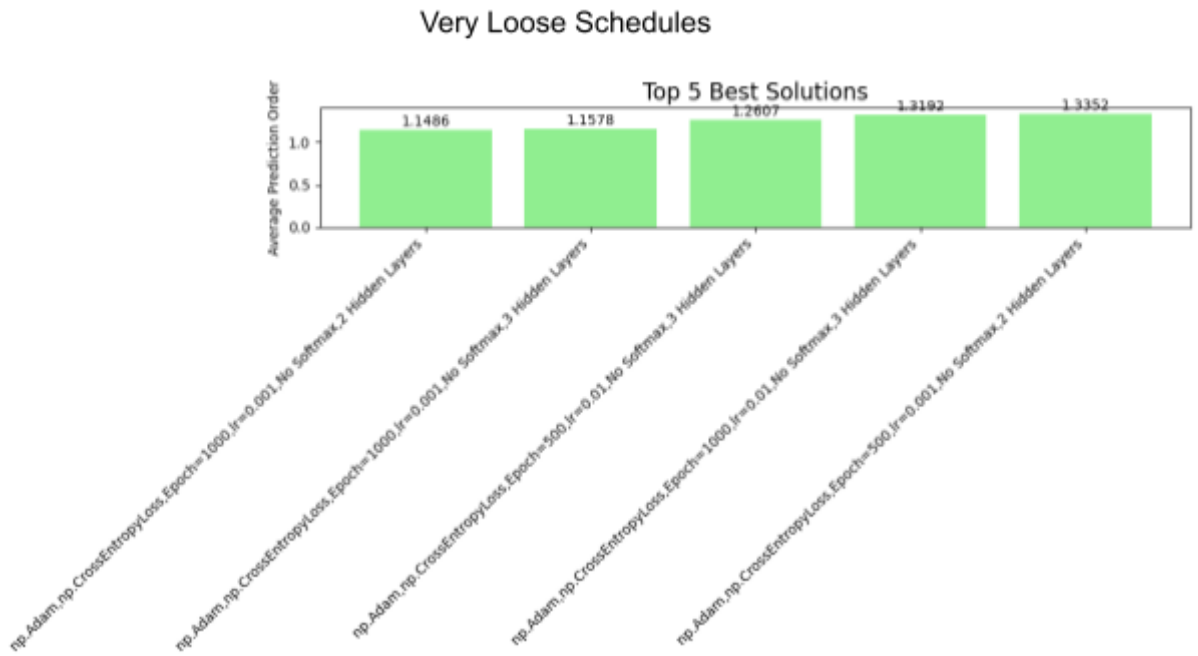
Accuracy of our model with best configuration for *normal schedules* predicts the first job correctly is %86,0.



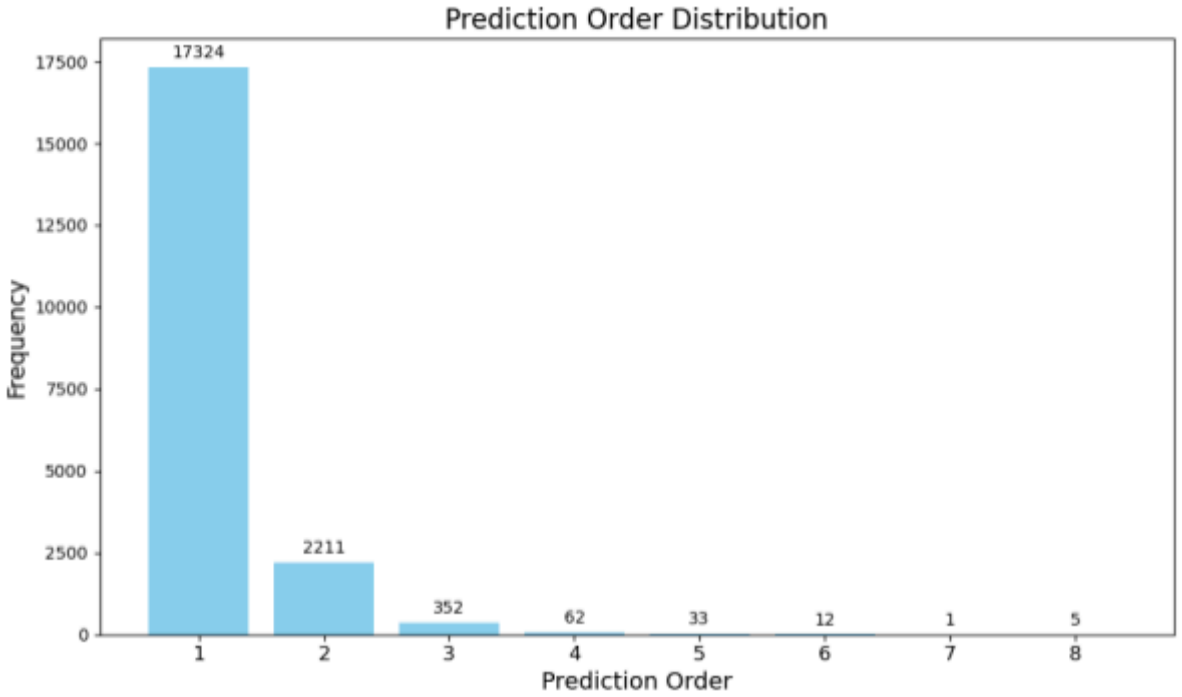
For loose schedules, hyperparameters with (Adam,CrossEntropyLoss, 1000 Epoch, 0.001 learning rate, without softmax, 3 Hidden Layers) gives the best result. Distribution of the best configuration's predictions are as follows:



Accuracy of our model with best configuration for *loose schedules* predicts the first job correctly is %84,1.



For very loose schedules, hyperparameters with (Adam,CrossEntropyLoss, 1000 Epoch, 0.001 learning rate, without softmax, 2 Hidden Layers) gives the best result. Distribution of the best configuration's predictions are as follows:

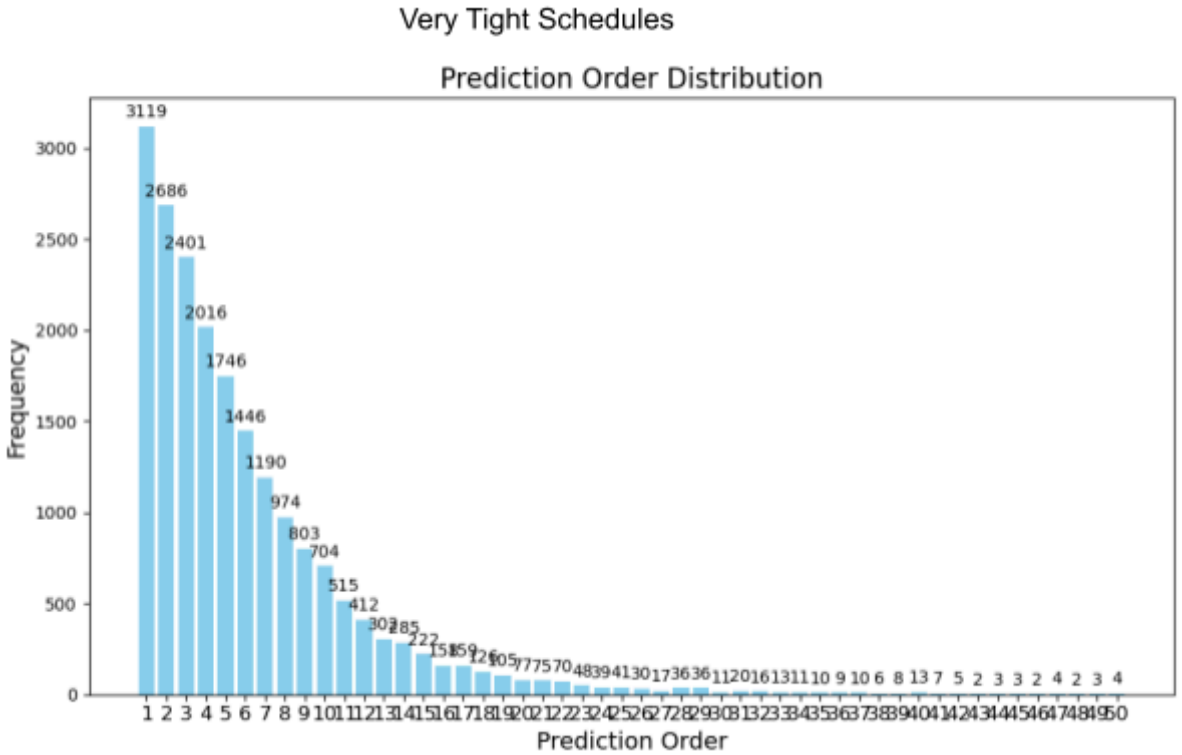


Accuracy of our model with best configuration for *very loose schedules* predicts the first job correctly is %86,6.

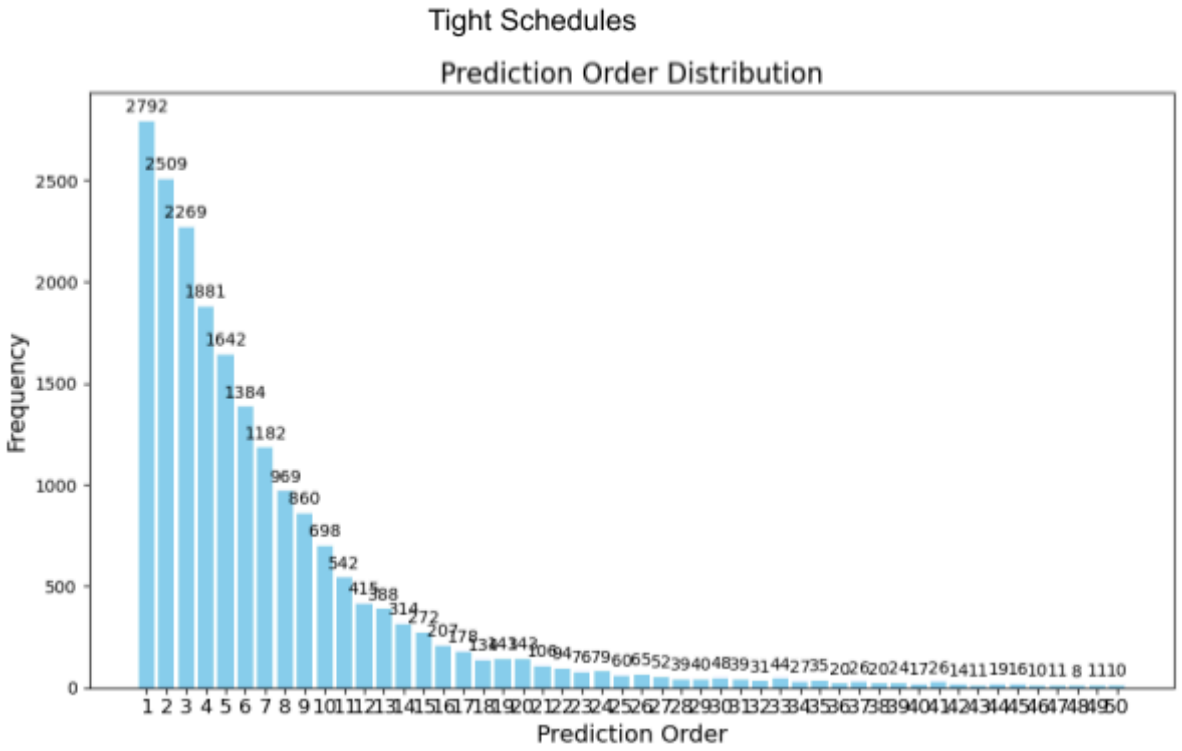
Shuffled Case

We put jobs in the input layer in the order of the sequence generated during data generation. Here, we aimed to analyze the following: if we provide the job order to the input layer in a shuffled sequence rather than the data generation order, will there be any change in the model's performance? Note: this shuffling only changes the position of the jobs, while their processing times, weights, and due dates remain the same.

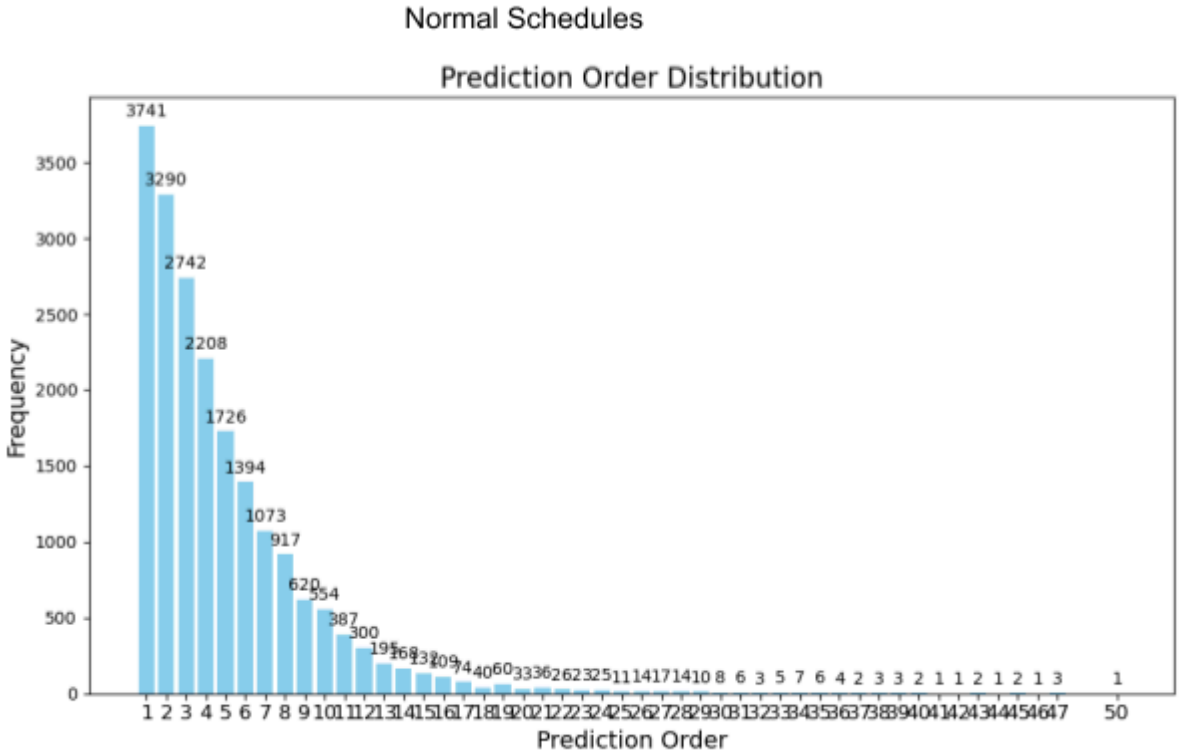
Then, we tested the model using the same hyperparameter combinations mentioned in the previous section and plotted the configurations that performed the best for each scheduling behavior based on our test function. We observed that all the best-performing configurations included the use of softmax. Configurations without softmax had an average prediction order of 17.49 (in other words, the job that should be processed nearly 17th in the actual schedule is predicted by our model without softmax as the first job to be processed) among all scheduling behaviors, which we can conclude that performance is very poor. Additionally, configurations with 3 hidden layers consistently result in the best performance. For the shuffled case, the accuracies across different scheduling behaviors are similar, and the best performing configurations result in an average prediction order of 5.31. See the results below for each scheduling behavior with shuffled data.



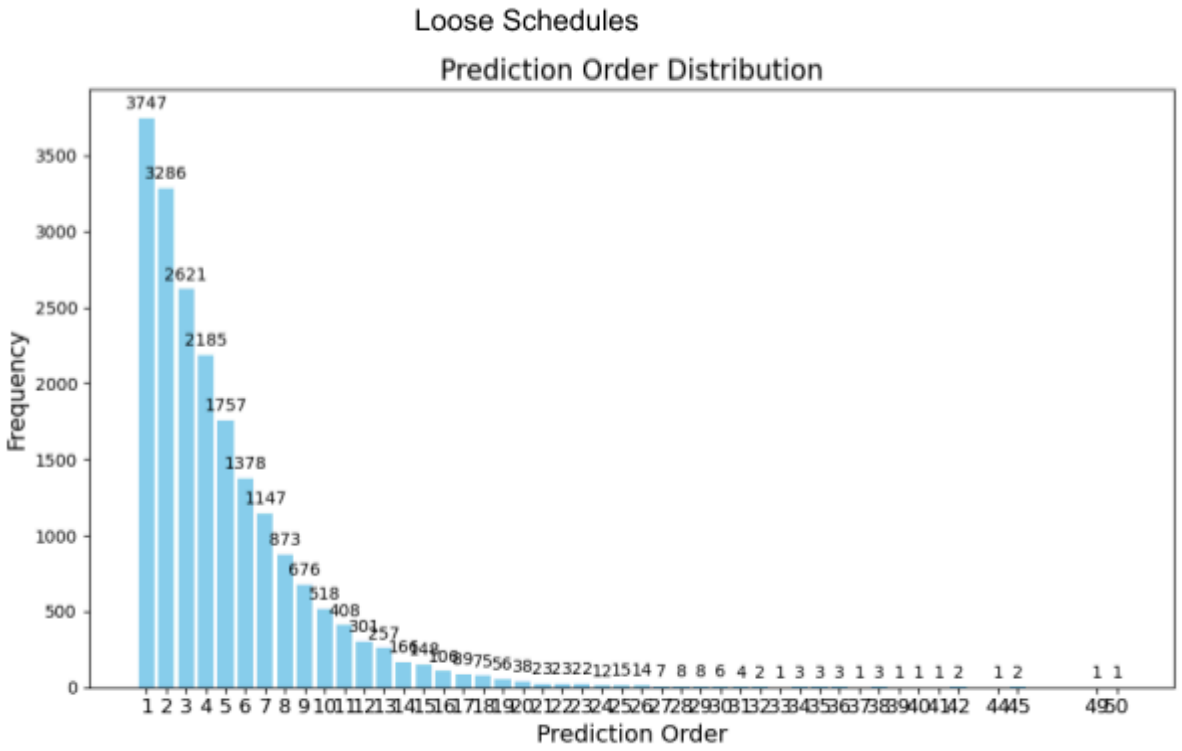
The accuracy of the model that is trained on shuffled data for Very Tight Scheduling behavior is 15.6%.



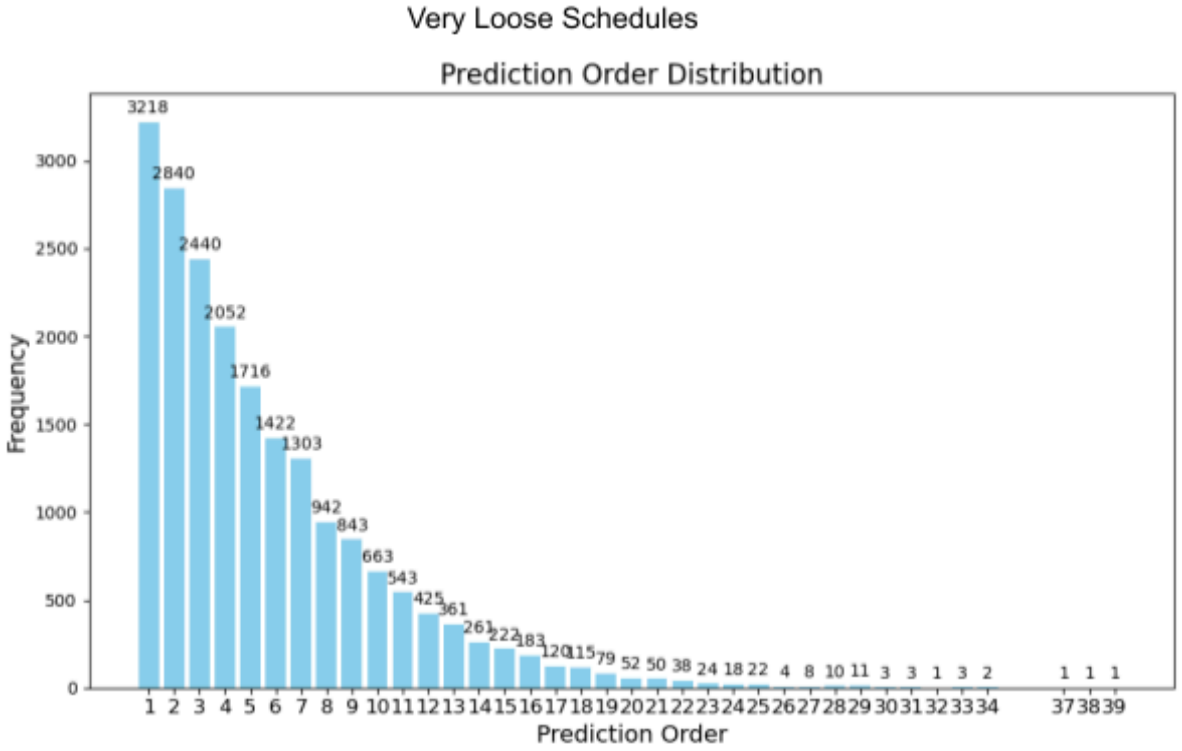
The accuracy of the model that is trained on shuffled data for Tight Scheduling behavior is 13.9%.



The accuracy of the model that is trained on shuffled data for Normal Scheduling behavior is 18.7%.



The accuracy of the model that is trained on shuffled data for Loose Scheduling behavior is 18.7%.



The accuracy of the model that is trained on shuffled data for Very Tight Scheduling behavior is 16.1%.

Conclusions

1. The first model we created for the project was quite naive, as it was our first experience with artificial neural networks. Additionally, we shifted our focus from generating full schedules to predicting the first job to process, as this is more relevant in a dynamic environment. Over time, we gradually increased our knowledge and confidence in the subject, and with our final model, we developed a well-performing solution.
2. Our model's prediction is first job of the real schedule in 85% of the tests, one of the first 2 jobs in the real schedule in 97.5% of the tests and one of the first 3 jobs in the real schedule in more than 99% of the tests so we can say that Neural Network could be efficient approach for solving scheduling problem.
3. Making a test took only several seconds so it performs faster than Heuristic and Optimization in the execution phase.
4. For our problem, some parameters such as Stochastic Gradient Descent(SGD) optimizer function, Mean Squared Error(MSE) Loss Function performed poorly. When we used SGD and shuffled data, our model always predicts the job #1 as the first job, so we can conclude that it may be stuck in local minima as mentioned in [4]. For the loss function we see that CrossEntropyLoss performed better when the output is a set of probabilities as mentioned in [5].
5. A smaller learning rate (0.001) results in improved convergence compared to larger rate (0.01) and higher number of epoch rates (1000) performed better than lower number of epochs (500).
6. Our model is trained and tested by heuristic solutions because finding an optimal schedule takes too much time. But based on our results, we can say that if enough time is spent to create data which contains optimal schedules then the model may be trained by optimal schedules and it can predict the jobs close to the optimal. Again, the data generation process is offline, so in the execution phase, it works as fast as a neural network which is trained by heuristic data.
7. Model is adaptable to schedules that have different tightness levels.

Future Work

We don't how the overfitting affects our solutions. Since the epoch numbers are relatively high, we expect the model to stop improving itself continuously in some cases, especially for different learning rates. Although, in some results, higher epoch rates performed better, there still some ambiguity. We might need an early stopping for better generalization.

Moreover, having different results on shuffled data, shows that learning is not perfect. It was expected to have the same results when we shuffled the data. This is also investigated for the future work.

Lastly, to improve the accuracy of predicting the first job, an attention layer could be added to the model. Since all job information, such as processing times, due dates, weights, is closely interconnected. The schedule heavily depends on the relationships between these values. Adding an attention layer could increase the accuracy by considering these dependencies. However, this approach has a downside—it increases the complexity of the model, which results in longer training time. Nevertheless, it's worth noting that training is performed offline, so this added complexity does not impact the model's online execution time. Therefore, an attention layer could be added to our model for further studies.

Appendix

Table 1*: Objective Functions in Scheduling*

Category	Objective (Performance Measure)	Notation	Description	Reference
Completion Time-Based Measures	Makespan	Cmax	Total time to complete all jobs	Conway et al. (1967)
	Total Completion Time	∑Cj	Sum of completion times	Smith (1956)
	Weighted Total Completion Time	∑wjCj	Sum of weighted completion times	Lawler et al. (1982)
Due Date-Based Measures	Maximum Tardiness	Tmax	Maximum delay beyond due dates	Kanet (1982)
	Number of Tardy Jobs	T	Count of jobs completed late	Moore & Hardy (1965)
	Total Tardiness	∑Tj	Sum of job tardiness	Baker & Trietsch (1984)
	Weighted Total Tardiness	∑wjTj	Sum of weighted job tardiness	Lenstra & Rinnooy Kan (1978)
Earliness and Tardiness Measures	Total Absolute Deviation	∑|Ej| + ∑|Tj|	The sum of the absolute deviations of all jobs from their due dates.	Hall et al., 1997
	Weighted Total Absolute Deviation	∑wj|Ej| + ∑wj|Tj|	The sum of the weighted absolute deviations of all jobs from their due dates, where wj is the weight of job j	Baker & Trietsch, 1984
Other Measures	Average Flow Time	∑Cj/n	Average time jobs spend in the system	Conway et al. (1967)

	Maximum Flow Time	Cmax - Pi	Maximum time a job spends in the system	Smith (1956)
	Number of Jobs Completed by Due Date	ND	Count of jobs completed on time	Moore & Hardy (1965)
	Weighted Number of Jobs Completed by Due Date	∑wjND	Weighted count of on-time jobs	Lenstra & Rinnooy Kan (1978)
Multiple Objective Measures	Lexicographic Ordering		Prioritizing one objective over another	Lawler et al. (1982)
	Weighted Sum		Combining multiple objectives with weights	Baker & Trietsch (1984)
	Goal Programming		Setting target values and minimizing deviations	Charnes et al. (1978)

2. MIP application to SMTWT

Indices

I = Jobs

Parameters

P_i = Processing time of job i

W_i = Weight of one unit tardiness penalty of job i

D_i = Due date of job i

Decision Variables

S_i = Starting time of job i

C_i = Completion time of job i

T_i = Tardiness of job i

$Y_{i,j}$ = {1 if job i processed before job j, 0 otherwise}

Model

$$minz = \sum_{i \in I} T_i \times W_i$$

$$C_i = S_i + P_i$$

$$T_i \geqslant 0$$

$$T_i \geqslant C_i - D_i$$

$$S_j \geqslant C_i - M \times (1 - Y_{i,j})$$

$$S_i \geqslant C_j - M \times Y_{i,j}$$

$$C_i, S_i, T_i : integer,$$

$$Y_{i,j} : boolean$$

3. Github Link to the project:

https://github.com/trkmnhll/IE492_Graduation_Project

4. Final ML Model

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import torch.nn.functional as F
from sklearn.model_selection import train_test_split
import pandas as pd
import csv
import matplotlib.pyplot as plt

def init_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_uniform_(m.weight)
        nn.init.zeros_(m.bias)


def load_data_from_csv(filename):

    inputs, targets, heuristics = [], [], []

    with open(filename, 'r') as file:
        csv_reader = csv.reader(file)
        data = list(csv_reader)
        for i in range(0, len(data), 3):
            try:
                input_matrix = eval(data[i][0])
                input_array = np.array(input_matrix, dtype=np.float32)
                inputs.append(input_array
                target_list = eval(data[i + 1][0])
                target_array = np.array(target_list, dtype=np.float32)
                targets.append(target_array)
                heuristic_sequence = eval(data[i + 2][0])
                heuristics.append(heuristic_sequence)
            except Exception as e:
                print(f"Error processing rows {i}, {i + 1}, and {i + 2}: {e}")
                continue

    return inputs, targets, heuristics

filename = 'loose_data.csv'
inputs, targets, heuristics = load_data_from_csv(filename)

inputs = np.stack(inputs)
targets = np.stack(targets)
heuristics = np.stack(heuristics)
inputs = torch.tensor(inputs, dtype=torch.float32)
targets = torch.tensor(targets, dtype=torch.float32)
heuristics = torch.tensor(heuristics, dtype=torch.float32)

train_inputs, test_inputs, train_targets, test_targets, train_heuristics, test_heuristics = train_test_split(
    inputs, targets, heuristics, test_size=0.2, random_state=42)
```

```
class RankNet(nn.Module):
    def _init_(self):
        super(RankNet, self)._init_()
        self.hidden1 = nn.Linear(50 * 3, 150)
        self.hidden2 = nn.Linear(150,100)
        self.hidden3 = nn.Linear(100,50)
        self.output = nn.Linear(50, 50)
        self.activation = nn.ReLU()

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.activation(self.hidden1(x))
        x = self.activation(self.hidden2(x))
        x = self.activation(self.hidden3(x))
        x = self.output(x)
        return x

model = RankNet()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
model.apply(init_weights)
train_targets = torch.argmax(train_targets, dim=-1)

epochs = 1000
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(train_inputs)
    loss = criterion(outputs, train_targets)
    loss.backward()
    optimizer.step()
    if (epoch + 1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
model.eval()
total_prediction_order = 0
predicted_jobs = []
correct_predictions = 0
order_counts = {}

print("Predicted Jobs:")
with torch.no_grad():
    for idx, (test_input, heuristic) in enumerate(zip(test_inputs, test_heuristics)):
        test_input = test_input.unsqueeze(0)
        test_output = model(test_input)
        predicted_job = test_output.argmax(dim=-1).item() + 1
        prediction_order = (heuristic == predicted_job).nonzero(as_tuple=True)[0].item() + 1
        total_prediction_order += prediction_order
        if prediction_order in order_counts:
            order_counts[prediction_order] += 1
        else:
            order_counts[prediction_order] = 1
```

```
        actual_first_job = heuristic[0].item()
        if predicted_job == actual_first_job:
            correct_predictions += 1


        predicted_jobs.append(predicted_job)

        print(f"Test Sample {idx + 1}: Predicted Job = {predicted_job} Order = {prediction_order} Heuristic =
{heuristic.tolist()}") # Optional

average_prediction_order = total_prediction_order / len(test_inputs)
print(f"\nAverage Prediction Order: {average_prediction_order:.4f}")
test_accuracy = correct_predictions / len(test_inputs) * 100
print(f"Test Accuracy: {test_accuracy:.2f}%")
print("\nPrediction Order Counts:")
for order, count in sorted(order_counts.items()):
    print(f"Order {order}: {count} times")
total_predictions = sum(order_counts.values())
print(f"\nTotal Predictions: {total_predictions}")

def plot_prediction_order_counts(order_counts):
    plt.figure(figsize=(10, 6))
    orders = sorted(order_counts.keys())
    counts = [order_counts[order] for order in orders]
    plt.bar(orders, counts, color='skyblue')
    plt.xlabel('Prediction Order', fontsize=14)
    plt.ylabel('Frequency', fontsize=14)
    plt.title('Prediction Order Distribution', fontsize=16)

    for i, count in enumerate(counts):
        plt.text(orders[i], count + 0.01 * max(counts), str(count), ha='center', va='bottom', fontsize=10

    plt.xticks(orders, fontsize=12)
    plt.tight_layout()
    plt.savefig('loose_prediction_order_bar_chart.png')
    plt.show()

plot_prediction_order_counts(order_counts)
```

References

[1] Smith, J. K., & Jones, R. (2020). "Heuristic Scheduling Techniques in Practice."

[2] Demir, H. I., Erden, C., Kökçam, A. H., & Uygun, Ö. (2019). "Concurrent solution of WATC scheduling with WPPW due date assignment for environmentally weighted customers, jobs and services using SA and its hybrid."

[3] Brown, A., & Wilson, P. (2018). “*Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, p. 430.”
[4] Goodfellow, Ian. "Deep learning." (2016).
[5] Bishop, C. M., & Nasrabadi, N. M. (2006). Pattern recognition and machine learning (Vol. 4, No. 4, p. 738). New York: springer.

All the content of this report is written by the project team except if the information is gathered from a resource, it is specified. AI tools like ChatGPT, or Gemini are only used in some parts not to create content but for language improvement of already established content just to increase readability of this report.