





A special thank you to Professor Dirk Hovy for taking me under his wing,  
and for Lorenzo Lupo for putting up with me the whole time



# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Contributions . . . . .	7
<b>2</b>	<b>Review of literature</b>	<b>9</b>
<b>3</b>	<b>Setting up the DFJSS problem</b>	<b>11</b>
3.1	Priority functions: the main object of study . . . . .	11
3.2	The inherent complexity of the problem . . . . .	12
3.2.1	Should the schedule wait, sometimes? . . . . .	13
3.2.2	Mechanics implemented . . . . .	14
3.3	Plots of 'patient' priority functions . . . . .	15
3.4	Features of the simulation . . . . .	16
3.4.1	Features of the simulation . . . . .	16
3.4.2	Mechanics of the simulation . . . . .	19
<b>4</b>	<b>Genetic algorithm</b>	<b>21</b>
4.1	Overview . . . . .	22
4.2	Methodology . . . . .	22
4.3	Results . . . . .	25
4.3.1	Possible improvements . . . . .	26

<b>5</b>	<b>Auto-Encoder</b>	<b>27</b>
5.1	The Idea . . . . .	27
5.2	General setup . . . . .	28
5.2.1	Definitions . . . . .	28
5.2.2	The Data . . . . .	29
5.2.3	Metrics . . . . .	29
5.2.4	Optimizer . . . . .	30
5.2.5	The "Anti-decoder" . . . . .	30
5.3	The Model . . . . .	31
5.3.1	Fixing the structure of the trees . . . . .	31
5.3.2	The network . . . . .	32
5.3.3	Training Results . . . . .	33
5.4	Other results . . . . .	38
<b>6</b>	<b>Reward model</b>	<b>39</b>
6.1	The Idea . . . . .	39
6.2	General setup . . . . .	40
6.2.1	The Data . . . . .	40
6.2.2	Metrics . . . . .	41
6.2.3	Optimizer . . . . .	42
6.3	The Model . . . . .	42
6.4	Training results . . . . .	43
6.5	Optimizing the reward . . . . .	49

<b>7</b>	<b>Discussion and Future Work</b>	<b>51</b>
7.1	Main Takeaways . . . . .	51
7.2	Algebraically Aware Encoding . . . . .	52
7.3	Training the autoencoder and reward model in tandem . . . . .	53
7.4	Changing the parameters of the DFJSS problem . . . . .	53
<b>8</b>	<b>Conclusion</b>	<b>54</b>
<b>A</b>	<b>Unused concepts</b>	<b>56</b>
A.1	Syntax Score . . . . .	56
A.1.1	Definition . . . . .	57
A.1.2	Problems . . . . .	59

# Chapter 1

## Introduction

Job Shop Scheduling (JSS) is a class of problems that concern *jobs* (or orders<sup>1</sup>) that need to be executed by *machines*, where we wish to optimize some objective function about the process. For instance, we would like to know, given a set of jobs and machines in a manufacturing plant, how to fulfill all jobs as quickly as possible, or to make sure they are done as close to the deadline as possible (Just-In-Time manufacturing).

Table 1.1: Some common (DF)JSS objective functions. In part from Xie, Gao, Peng, Li, Li [11]

Objective	Interpretation
maximum completion time	How long it takes for all jobs to be finished
maximum/total/mean tardiness	How late a job is completed relative to its deadline. Early jobs have 0 tardiness
maximum/total/mean earliness	How early a job is completed relative to its deadline. Late jobs have 0 earliness
maximum/total/mean net lateness	How late a job is completed relative to its deadline. Can be negative
”Just-In-Time” penalty	Combines tardiness and earliness. Each job may have different penalty weights for each
total idle time	For how much time machines have been unused throughout their lifetime
total/mean flow time	For how much time jobs have stayed in the shop
maximum/total workload	For how much time machines have been used throughout their lifetime
total operation cost	Monetary cost of the shop
total energy consumption	Energy consumption of the shop

Formally, each job  $J_i \in J$  has an ordered set of operations  $O_j = (O_{j1}, O_{j2}, \dots)$ , which need

---

<sup>1</sup>’jobs’ or ’orders’ may be used interchangeably.



to be executed in order and one at a time, and the job is finished once all operations have finished.

Operations are executed by machines  $M_i \in M$ . Not all machines may be able to execute all operations, though. We say that each operation follows a recipe  $r \in R$ , and each machine belongs to a family  $f \in F$ , and each family contains recipes that a machine can execute.

For instance, the job of making a cup of tea may have the operations  $O_j = (\text{Boil water}, \text{Add teabag})$ , while a microwave oven may belong to a family with the recipes  $f = \{\text{Heat up}, \text{Boil water}\}$ .

If the job and the machine are available, we would start the job by assigning it to the microwave oven.





































	TOASTER	DISHWASHER	MICROWAVE	WASHING MACHINE	STOVE/OVEN	DRYER
MAKE TOAST						
WASH DISHES					 STERILIZED AT LEAST.	
COOK A FROZEN DINNER		 FISH MIGHT BE COOKED				
WASH CLOTHES						
COOK EGGS			 (POACHED)			
DRY CLOTHES						

Figure 1.1: xkcd [9]. Some of your home appliances belong to a family with multiple recipes. Some. "If you had an oven bag and a dryer that runs unusually hot, I guess you could in theory make tumbled eggs."

Machines, jobs and operations may have parameters that determine their behavior. A machine may be faster at heating up water than another, a job may have a deadline, a 'Boil water' operation may take longer than another if there is more water to boil.

The solution of a JSS problem is something that determines in what order the jobs' operations are assigned (*operation sequencing*) and to what machines (*machine assignment*).

In baseline JSS, which has a *fixed* number of jobs and machines, and each machine only has one recipe, these solutions may be solved for and explicitly described using graphs [8]. However, in practice, this is too rigid of a setup to be useful.

Flexible Job Shop Scheduling (FJSS) [11] assumes machines may have more than one recipe, while keeping the machines and jobs fixed. Dynamic Flexible Job Shop Scheduling (DFJSS) assumes that they might change over time, for example with machines breaking down and/or being replaced, or jobs being added over time ((random) job arrival).

FJSS and beyond not only are NP-hard [11], but even if we were to solve one particular snapshot of the problem, at some point in time that solution will become obsolete as the environment changes. A different approach would be to find a heuristic which quickly computes a good enough solution in a set of benchmark environments.

Regardless of the class of the problem, it is a common trait that an "incremental improvement" in the solution is not trivial, as the solutions are not objects that can be differentiated. This means that proposing a better solution is a process of trial and error, for instance by perturbing a given schedule and seeing if it improves ("local search"), or by using a genetic algorithm.

There have been past approaches to move the DFJSS problem into a continuous space. Chang [1] is a recent article that uses deep reinforcement learning to solve the DFJSS problem with random job arrival. With their use of neural networks, double-deep Q-networks (DDQNs) in particular, it's possible to move the problem from a discrete space to a continuous one. However, DFJSS has a varying number of possible actions and the feature space is large, which need to be limited to fit in fixed-size feed-forward neural networks.

The focus of this dissertation will be on syntax trees, as described in Zhang [12] and Poli, Langdon, McPhee [10]. Rather than containing the entire solution of the problem, they

are instead *priority functions* with a binary tree construction, which take all possible combinations of machine assignment and operation sequencing, and output a *priority value* depending on the value of the features of the jobs, their next operation, and of the machines. The machine-operation pair with the highest priority is executed. Compared to the DDQN approach by Chang [1], syntax trees are more flexible in the number of features they can have as input and the number of actions that they can suggest is not limited by construction. In addition, a syntax tree solution is fast to use, compared to a GPU-intensive neural network.

Syntax trees, by their very nature, are a sequence of words (or "tokens") which need to follow a particular syntax. In parallel, the bleeding edge of neural networks has been turning sequences of words into differentiable embeddings, which is something that I will explore.

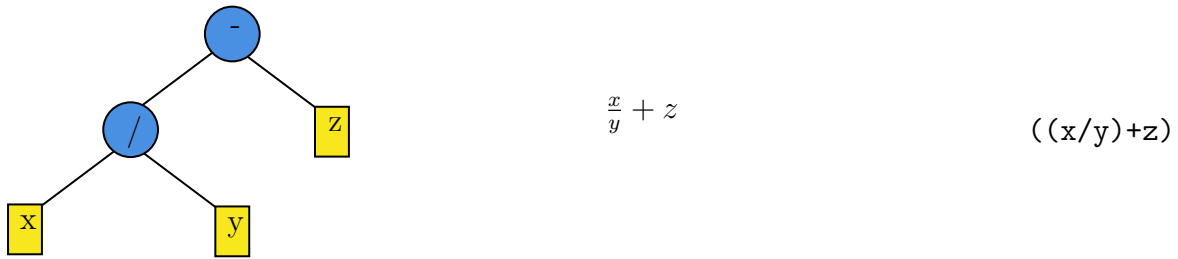


Figure 1.2: Equivalent representations of a syntax tree (full binary tree, mathematical expression, character string)

## 1.1 Contributions

- I will consider a formulation of the DFJSS problem with random job arrival and machine breakdown, where the solution space are priority functions, which are syntax trees which assign a priority value to machine assignments/operation sequencing. The DFJSS implementation was written by me in Python, and its code is made available on GitHub, at [https://github.com/MicheleScarnera/ms\\_dfjss](https://github.com/MicheleScarnera/ms_dfjss).
- As for the optimization of the problem, I will implement a genetic algorithm as a baseline, by following the guidelines in Poli, Langdon, McPhee [10].
- Using the properties of neural networks, I will try and solve the same DFJSS problem

by first synthesizing the syntax trees into a continuous space, adding differentiability to the problem, using an autoencoder. I will then use this autoencoder’s compression to train a reward model, aiming to approximate the fitness values of the priority functions using only their compressed form from the autoencoder’s encoder.

- I will then explore if the reward model is capable of proposing feasible and optimal syntax tree solutions.

While the training of neural networks in general is GPU-intensive, the goal is to obtain a syntax tree out of the setup, which means the networks need not be used anymore, unless more training is desired.

**This is a negative result** It’s important to mention right from the beginning that, while the dimensionality reduction from the autoencoder was successful, the reward model could not be trained to an acceptable standard. Therefore, anything beyond that requires the reward model cannot be done. Nonetheless, by laying out the methodology of all the steps, I have found many potential openings for improvement, which I will highlight as the context for them is explained.

# Chapter 2

## Review of literature

Production scheduling has been studied for decades, as it is an evergreen topic in a world with intricate manufacturing facilities and complex supply chains.

Depending on whether the paper tackles JSS, FJSS, or DFJSS, there are significant differences on what the study of the solution looks like. For instance, Kuhpfahl and Bierwirth [8] tackle the baseline JSS problem, in particular they expand the possible local search operations of the schedules. Given that JSS schedule graphs describe everything about the process, they are objects that are hard to read and that could become non-feasible (meaning, that they would no longer be a valid schedule) with a wrong perturbation. Therefore, these new operations are multi-step, and most of the paper tackles how they always preserve feasibility.

Since FJSS still has a known, finite amount of machines and jobs, it can still be solved with similar graphs, with Chen [2] being an example. Rather than using local search, genetic algorithms are used.

As previously mentioned, Zhang [12] tackles DFJSS by using a priority function heuristic, which are made of syntax trees which output values using the features of jobs, their operations, and the machines, with higher priority actions being done first. Compared to graph-based solutions, a syntax tree formulation makes it trivial to define perturbation operations which preserve feasibility.

Since the evaluation of the fitness of these priority functions, or of any other approach for that matter, can be time consuming due to having to run simulations, ideally many

of them for each candidate, the same paper makes use of a *phenotypic characterization* heuristic from Hildebrandt [5], which allows to have a lossy precompute table to avoid calculations which are deemed redundant. While Zhang uses this for feature selection, Hildebrandt’s idea can be used to speed up anything involving evaluating the fitness of a priority function.

The DDQN approach of Chang [1] is the latest attempt at using the differentiability of neural networks to solve the DFJSS problem. As alluded in Chapter 1, DFJSS has a varying number of actions and features to consider, which need to be limited to fit in fixed-size feed-forward neural networks. Hence, the paper fixes 4 global-state features as their input for the neural networks, and 4 output nodes that denote the reward of 4 different actions. These 4 actions are a set of ”common sense” actions that are deemed appropriate by the authors, for instance working on the job that is closest to being done, or the job that is contributing the most to the penalty.

# Chapter 3

## Setting up the DFJSS problem

In this chapter, I will be explicitly writing down the DFJSS problem, and the objects that I will use to solve it: priority functions.

As it will become clearer later, it's important to keep in mind that the DFJSS problem is not *the* logistical problem, but *one* logistical problem, as not all logistical problems can be neatly be put into the DFJSS box.

In Section 3.4, I will also be explicit about what the DFJSS environment is, by listing all of its features and by writing down the simulation in pseudocode.

### 3.1 Priority functions: the main object of study

As previously mentioned, the solution space will be priority functions. I will now define it more explicitly.

Consider the formal setup of jobs  $J_i \in J$ , operations  $O_j = (O_{j1}, O_{j2}, \dots)$  and machines  $M_i \in M$  from the introduction. Additionally, let  $\dot{O}_j$  be the current operation of job  $j$  that needs to be completed.

Each of these objects may have a fixed set of features. For example, machines may have the set of features {Machine's work power, Machine cooldown}. Each object's feature has a value. If any of the symbols above are written in **bold**, they mean the vector of features of that object.

A priority function  $P : (J, M) \rightarrow \mathbb{R}$  takes as input a job, its current operation, and a

(compatible) machine, and returns a number. Note that by choosing a job  $J$ , we also have access to its current operation  $\dot{O}$ . The priority functions of study in the thesis are made exclusively of binary operations on the jobs, their current operations, and machines' features, and are occasionally referred to as "syntax trees".

For example, one priority function might be:

$$P(J, M) = P(J, \dot{O}, M) = \text{Machine's work power} - \text{Operation's work to be completed}$$

The machine assignment/operation schedule to be done is whichever  $(J, M)$  pair, out of the jobs/operations/machines that are compatible with each other and available, has the highest value.

By making decisions over time on what to do first, by the end of the simulation, the priority function's fitness is computed. See Table 1.1 for examples. The fitness function of choice  $\Phi : (P, s) \rightarrow \mathbb{R}$  takes the priority function and the simulation (or simulation seed, or environment, depending on the context), and returns a number.

Finally, the DFJSS problem at hand is:

$$\min_P \Phi(P, s)$$

## 3.2 The inherent complexity of the problem

Even though defining the problem is relatively straight forward, the implementation of simulations of DFJSS have the potential of being wildly different from each other. Depending on the field DFJSS is being applied to, one might need to add features to jobs or machines that are essential but not usually discussed, while there might be common features that are not relevant. Even the basic assumption that all JSS papers start with, the one that jobs have a list of operations which need to be done in order and one at a time, could not hold in some fields. The operations may not need to be done in a certain order, the set of operations may resemble a tree more than a sequential line, the features of *all* operations may be made accessible (rather than just the current one), et cetera.



Additionally, the more "general" one wants to make their DFJSS simulation, the bigger the parameter space becomes. It's not surprising that most if not all of the cited papers limit themselves to at most a dozen features, and unusual mechanics are considered one at a time. The purpose of those papers is usually not the direct application of the presented solution in a field, but simply discussing the optimization strategy, so this is understandable. I will be listing the mechanics of note of my simulations, with the knowledge they might not necessarily apply to all fields.

### 3.2.1 Should the schedule wait, sometimes?

Concerning job shop problems where finishing a job earlier is penalized, it might be the case that a job shop schedule might finish jobs too early and get a penalty. At first, we might think of these schedules as being worse, but we might be incorrectly assuming that every time an action *can* be done, that an action *must* be done. In my review of the literature, I could not find any explicit mention of job shop schedules that are allowed to do nothing when queried to do something. I suspect there might be some cases where there are very few jobs in the queue, and that doing nothing for some time would improve the earliness penalty, but schedules are obligated to work on them immediately. This might make some job shops, the ones with few jobs at a time especially, artificially harder to optimize.

For this reason, I have decided to implement a wait mechanic. When the priority value of an action is strictly less than 0, it's not considered in the scheduling decision. If all actions that can be done have a negative value, the scheduling waits for the next soonest operation/machine to become available to try making a decision again (with a fallback value of 5 seconds if there are none). To avoid infinite loops (i.e. a constant, negative priority function), there is a "wait debt" counter  $d$ , initialized at 0, which goes up by 1 for every wait, and down by 1 for every non-wait action. If the wait debt counter is greater than 0, all priority values are offset by  $2^{d-1}$ . For example, a constant priority function of  $-2$  would be queried two times and it would always wait, being  $-2$  the first time and  $-1$  the second, then on the third query it would make an arbitrary decision, as it would return

1 for every argument. While the choice of the offset and its growth are hyperparameters, I suspect the effect on the end result would be marginal, as any optimizing algorithm would internalize them. This setup was chosen because it was robust against pathological priority functions<sup>1</sup>. In Section 3.3 there are histogram plots for the fitness of an example priority function, with and without a negative offset, which allows for occasional waiting.

### 3.2.2 Mechanics implemented

In my implementation of DFJSS, I have chosen to implement the following mechanics:

- Random job arrival
- Random machine breakdown
- The capability of machines of working on more than one operation at a time, with a penalty on per-operation speed
- "Just-In-Time" penalties for both earliness and lateness, delivery relaxation factors
- Schedules can choose to "wait"

---

<sup>1</sup>The reason the offset grows exponentially, rather than linearly, is that some pathological priority functions might output large, negative values, which would not be a problem if not for this wait mechanic. For the same reason, the offset doesn't immediately reset to 0 when a non-wait action is taken, but still stays relatively high. Coincidentally, the first time I tried running a genetic algorithm with a linearly increasing offset which would reset to 0, the first individual it generated was *very* pathological, routinely in the negative tens of thousands, and I never got to fully evaluate that individual because it would have taken too long.

### 3.3 Plots of 'patient' priority functions

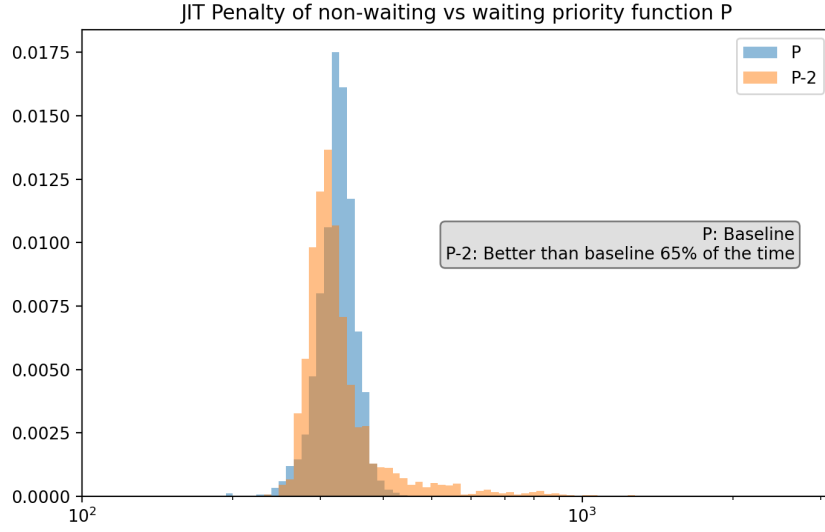


Figure 3.1: JIT penalty of the following priority function  $P$

$$P(\cdot) = \frac{\text{Job Flow Time}}{(\text{JobRelativeDeadline} \vee 0) \times \text{JobRemainingNumberOfOperations}}$$
 without and with a negative offset. The former never waits, while the latter might. The simulations seeds for the two groups are the same ( $N = 1500$ , 100 histogram bins). Note how the offset function has a much long tail.

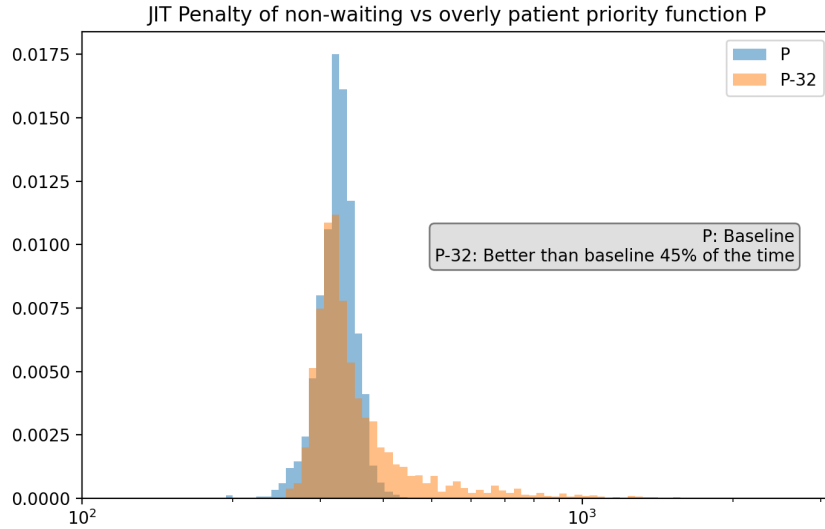


Figure 3.2: The same priority function, but made "overly patient" with an excessive offset. Moving any arbitrary priority function down is not necessarily an improvement.

## 3.4 Features of the simulation

As mentioned before in Section 3.2, the specifics of the DFJSS simulation depend heavily on the field they’re going to apply to. With that said, this section summarizes what features are implemented, and provides their distribution when appropriate. Some features do not affect some objective functions, like monetary cost not affecting lateness (while most other time-based features might), but their implementation might be useful if a multi-objective extension of this implementation is pursued. For instance, how Yuguang et al. [13] optimize for multiple objectives by finding the Pareto fronts. The features the optimization strategies are exposed to will be in **bold**.

### 3.4.1 Features of the simulation

**Recipes and the families they belong to** *Inspired from xkcd [9]*

- Make toast  $\ni$  {Toaster, Oven}
- Cook frozen dinner  $\ni$  {Microwave, Oven}
- Cook eggs  $\ni$  {Microwave, Oven}
- Wash dishes  $\ni$  {Dishwasher}
- Wash clothes  $\ni$  {Washing machine}
- Dry clothes  $\ni$  {Dryer}

#### Operations’ features

- **Work required** — Amount of work to finish the operation [joules, seconds, etc.]  $[U(10, 500)]$
- **Windup** — Amount of time right after machine assignment before the operation starts being worked on. Neither the machine nor the job are available during the process. [seconds]  $[U(0, 60)]$
- **Cooldown** — Amount of time after finishing the operation before the job can continue being worked on. [seconds]  $[U(0, 60)]$
- **Start time** — Absolute time at which the operation was started being worked on, pre-windup. [seconds]

## Jobs' features

- **Starting number of operations** — Number of operations the job starts with. [integer]  $[U(2, 12)]$
- **Remaining number of operations** — Number of operations the job currently has. [integer]
- **Deadline** — Time at which the job is requested to be finished. Internally, there is a '**relative**' deadline (relative to job arrival), and an '**absolute**' deadline. [seconds]  $[U(120, 720)$  (relative)]
- **Arrival time** — Time at which the job entered the simulation. [seconds]
- **Current flow time** — For how long the job has been in the shop. [seconds]
- **Earliness/Lateness penalty** — Per-second penalty of finishing the job early/late. [seconds/second]  $[U(0.5, 1.5)/U(0.5, 2)]$
- **Delivery relaxation factor** — Upon finishing the job, its flow time is scaled by this value. If  $> 1$ , the finished job will be treated as if it was finished earlier. If  $< 1$ , it will be treated as if it was finished later. [seconds/second]  $[U(0.5, 2)]$
- **Remaining work to complete** — Amount of work of all remaining operations. [joules, seconds, etc.]

## Machines' features

- (Nominal) **Work power** — Amount of power (work per second) this machine has. If the machine is working on multiple operations at once, the actual work power may be lower. [watts, seconds/second, etc.]  $[U(5, 200)]$
- **Capacity** — How many operations can the machine do at once. [integer]  $[U(\{1, 2, 3\})]$
- **Capacity scaling** — How the machine scales its work power, depending on the current number of operations it's working on. [function decreasing on no. of concurrent operations  $c$ ]  $[U(\{100\%, \frac{100\%}{c}, \frac{100\%}{c^{0.5}}\})]$
- **Cooldown** — Amount of time after finishing an operation before the machine can be used again. [seconds]  $[U(0, 60)]$
- **Machine Breakdown Rate** — While a machine is in the simulation, there is a chance it will break down. Upon breaking down, the machine is removed, and all operations under it go on their cooldown, but without being completed, and need to be started again from scratch. Breakdowns happen using a Poisson Process regimen. [breakdowns/second]  $[U(0.0001, 0.0005)]$
- **Replacement cooldown** — When a machine breaks down, it's replaced with another one with the same recipe, and it starts on cooldown. The cooldown is determined by the broken down machine. [seconds]  $[U(60, 120)]$
- **Monetary cost (fixed/per second)** — Amount of monetary cost upon starting a job/per second of use. [money(/second)]  $[U(0.01, 0.02)/U(0.005, 0.01)]$
- **Energy cost (fixed/per second)** — Amount of energy consumed upon starting a job/per second of use. [energy(/second)]  $[U(1, 5)/U(10, 50)]$

### Machine-Operation pairs' features

- **Number of alternative machines** — How many machines could also fulfill this operation, including the machine in the pair. [integer]
- **Number of alternative operations** — How many available operations could also be fulfilled by this machine, including the operation in the pair. [integer]
- **Expected work power** — Given the current situation, the work power that the machine will provide to the operation. A machine's power can be reduced if it is working on more than 1 operation at a time. [watts, seconds/second, etc.]
- **Nominal/Expected processing time** — The expected time to finish an operation, depending on nominal/actual work required by the operation and the machine's work power. [seconds]

### Shop's features/Global state features

- **Utilization rate** — How many machines, in percentage, are currently being used. Machine with capacity  $N > 1$  are counted as if the machine is split into  $N$  copies with capacity 1. [percent]

### Simulation's features

- **Number of starting machines over essential** — Number of machines to generate at the start, beyond the "essential" ones (one for each family) [integer] [ $U(0, 15)$ ]
- **Number of starting jobs** — Number of jobs to generate at the start. [integer] [10]
- **Job arrival average amount** — Amount of jobs that randomly arrive. Job arrivals happen under a Poisson Process regimen. [jobs] [ $U(20, 300)$ ]
- **Simulation time window** — Random job arrivals happen only inside this window, after which the simulation will exhaust all of its jobs and end. [seconds] [43200]

### 3.4.2 Mechanics of the simulation

**Definitions** Before describing the simulation, here are some needed definitions:

- **Busy Couple** – An operation/machine pair that is currently being worked on/working. Each of them has a processing time, which needs to reach 0 before the work is considered done. One machine may be in more than one busy couple if it has *Capacity* higher than 1.
- **Wait** – The job/operation/machine cannot be assigned to anything, and does not take part in the decision making until it stops waiting. For machines with *Capacity* higher than 1, a machine cannot be used only if the amount of busy couples it's in is greater or equal to its capacity.
- **Release** – The action of making a job/operation/machine stop waiting;
- **Compatible Pair** – An operation/machine pair where the machine can potentially be assigned to the operation at this step, i.e. a machine whose recipe is an element of the operation's family, and neither are waiting;
- **Decision Rule** – The logic that dictates which compatible pairs are actually assigned. The decision rule may choose to do nothing. Only priority function decision rules are considered, but other objects are possible;
- **Time(s) Passed** – The (simulated) time passed between the start of the previous routine step and the current routine step;

**Initial state** Before the simulation can begin, the following is done:

1. The essential machines are generated, generating more depending on *Number of starting machines over essential*;
2. The essential jobs are generated, generating more depending on *Job arrival average amount* and *Simulation time window*. The latter jobs are not available at time 0, and wait such that they are released uniformly throughout the simulation time.

**Progression of states** The simulation's discrete steps are called "routine steps". In each routine step, the followings things are done in order:

1. **Machine breakdown** – Each machine is tested for breakdown. With probability  $\text{Machine Breakdown Rate} \times \text{Time Passed}$ , the machine breaks down, meaning all operations under it go on their *Cooldown* and must be started from scratch. A new machine which uses the same recipe is created, and waits for the duration specified by the broken down machine's *Replacement cooldown*.
2. **Check operations that are finished** – Check all busy couples. For all of them whose processing time is  $\leq 0$ , make the machine wait its *Cooldown* and remove the busy couple. If the operation was the job's last, remove the job. If not, make the job wait its *Cooldown* and expose its next operation.
3. **Waiting jobs/operations/machines** – Check all waiting jobs/operations/machines. If any of them's waiting time has passed, release them.

4. Utilization rate – Calculate the *Utilization rate*, as the current number of busy couples divided by the maximum number of busy couples (that all machines can accomplish).
5. Real time features – Features that change over time are calculated. Notably, machines' breakdown rate changes according to a uniform distribution, between 0 and *(Machine's Maximum) Breakdown Rate*.
6. Assign operations and machines – Get all compatible pairs, and run them through the decision rule. The pairs selected by the decision rule are assigned and turn into busy couples, whose processing time depends on the operation and machine's relevant features and state.
7. Check end state – If there are no jobs left, end the simulation. Note that a simulation can go for longer than *Simulation time window*, but usually not much longer as the number of jobs is finite.
8. Make time pass – Check all things that are waiting for time to pass (waits, windups, busy couples, etc). The routine step's time passed is the minimum time found. That time is subtracted from all time-waiting objects and the routine step is over.



# Chapter 4

## Genetic algorithm

In this chapter, I will be setting up the genetic algorithm that looks for optimal priority functions for the DFJSS problem. Genetic algorithms are a natural choice, because priority functions are made of discrete elements, and mutations, be them pointwise (one element of the tree changes to another) or crossovers (one part of a tree is replaced with another part from another tree), are easy to implement in a way that doesn't make the tree unusable for the problem.

The goal of a genetic algorithm is to mimic evolution: a set of trees is tested against the problem, the better performing ones are kept (they "survive"), and new copies are made from the successful population (the "next generation"). For an exhaustive explanation of genetic algorithms, I recommend the GP Field Guide by Poli, Langdon, McPhee [10] ("The GP Field Guide" from this point on).

## 4.1 Overview

Table 4.1: Overview of GP

Objective	Find a syntax tree DFJSS schedule that minimizes the loss function
Function set:	Addition, subtraction, multiplication, safe division, min ( $\wedge$ ), max ( $\vee$ )
Terminal set:	A subset of features described in Section 3.4, and random constants in $\{-15, -14.5, \dots, 15\}$
Loss function:	Mean Just-In-Time (JIT) penalty

Table 4.2: Parameter settings of GP

Parameter	Value
Population size	1000
Number of generations	40
Number of simulations per individual, per generation $\dagger$	50
Number of simulations seeds $\dagger$	200
Summary of simulations' fitness $\ddagger$	$0.75 \times \text{Median} + 0.25 \times \text{Mean}$
Method for initializing population	ramped-half-and-half
Initial maximum depth	2
Maximum depth of programs	8
Crossover/Reproduction/Mutation rate	80%/10%/10%
Parent selection	Tournament selection
Tournament selection size	5% of population

## 4.2 Methodology

The following paragraphs describe various tricks that were implemented in the genetic algorithm. Some of these approaches were inspired by the GP Field Guide [10].

**$\dagger$ Overfitting and Local Optima** According to the GP Field Guide [10], if the fitness function is static, meaning that, for instance, the training data stays the same, the algorithm might evolve into niches that overfit for that training data, be it with a local or almost global optimum. In order to prevent overfitting, the simulations that are used to compute the fitness of individuals are changed every generation. There is a set of possible simulations (of size *Number of simulations seeds*), and each generation there is a random sampling of the set, without replacement, of size *Number of simulations per individual*,

*per generation*, which define the loss function for that generation. This prevents the algorithm from selecting individuals which overspecialize to a limited number of simulations, and also allows the closest local optimum to "move around", preventing the algorithm from getting stuck in one. Additionally, evaluating the fitness of one individual in one simulation is expensive, and this approach allows to somehow reduce the number of simulations per generation while still keeping the overall fitness evaluation quite expressive over time.

‡**Summary of simulations** The fitness of these different simulations needs to be reduced to a single number for the genetic algorithm to rank individuals. While the median of a sample is a robust summary when dealing with unknown distributions, especially when working symbolically, empirically it might not be sensitive enough for GP to use. According to the GP Field Guide [10], it helps if the fitness function has as many small steps as possible, a "gradient" loosely speaking. This is why some of the mean value is used to provide such gradient.

**Small sample bias** Due to the nature of changing the simulations each generation, some individuals might be evaluated on less simulations overall than others. This might result in individuals with few data points that overperform, or underperform, due to chance. To compensate, the probability of one given simulation being sampled in a generation is inversely proportional to how many unique individuals it has values for, which allows simulations that haven't been used a lot to be more likely to be sampled in the future. If appropriate, I will discard individuals with too few simulations if they are seemingly the best performing ones. Normally, you would address this by, for instance, centering the metric of fitness around a null hypothesis value and dividing by its standard error. However, evaluating one individual in one simulation is expensive, hence the sample size for such standard error is limited, and I cannot make any assumption on the distribution of fitness across different simulations, making the naive standard error potentially incorrect. The naive standard error might be "correct enough" if a given individual has been evaluated in the hundreds of times, where things like the Central Limit Theorem

apply. However, the rate of convergence depends on the heaviness of the tails, which in turn might depend on the parameter space and the quality of the schedule. Figure 4.1 is an example of changing the parameter space which influence the heaviness of tails.

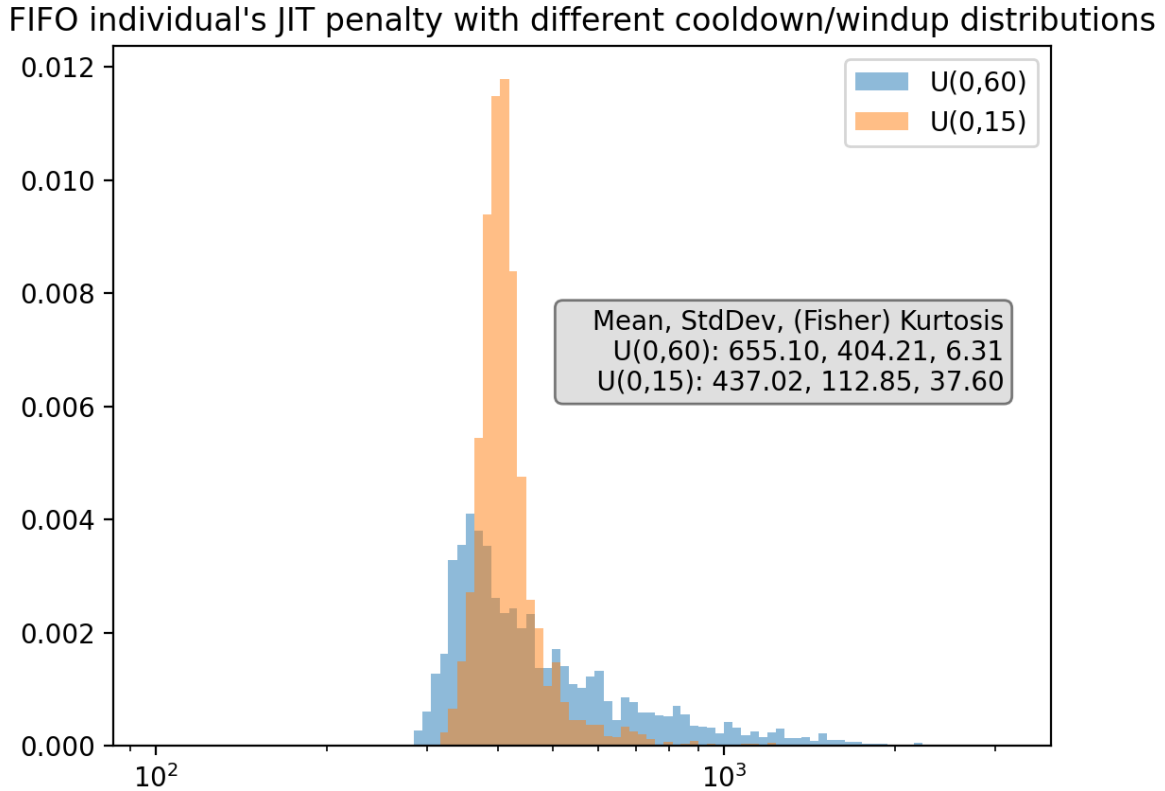


Figure 4.1: JIT penalty of a FIFO (First-In-First-Out) individual on 1500 simulations, with different distributions for Machine Cooldown, Operation Cooldown, and Operation Windup. The simulations seeds for the two groups are the same ( $N = 1500$ , 100 histogram bins). Mean, standard deviation and kurtosis are also shown.

While making cooldowns and windups smaller might make most job queues easier most of the time, a FIFO individual, which is not able to wait, might occasionally finish jobs consistently too early in some simulations. Even if better suited individuals have consistently thin tails, applying standardization to the metric with varying kurtosis might be unreliable. Hence, I will only make use of the strategy of sampling simulations that haven't been sampled much in the past, which was described above.

## 4.3 Results

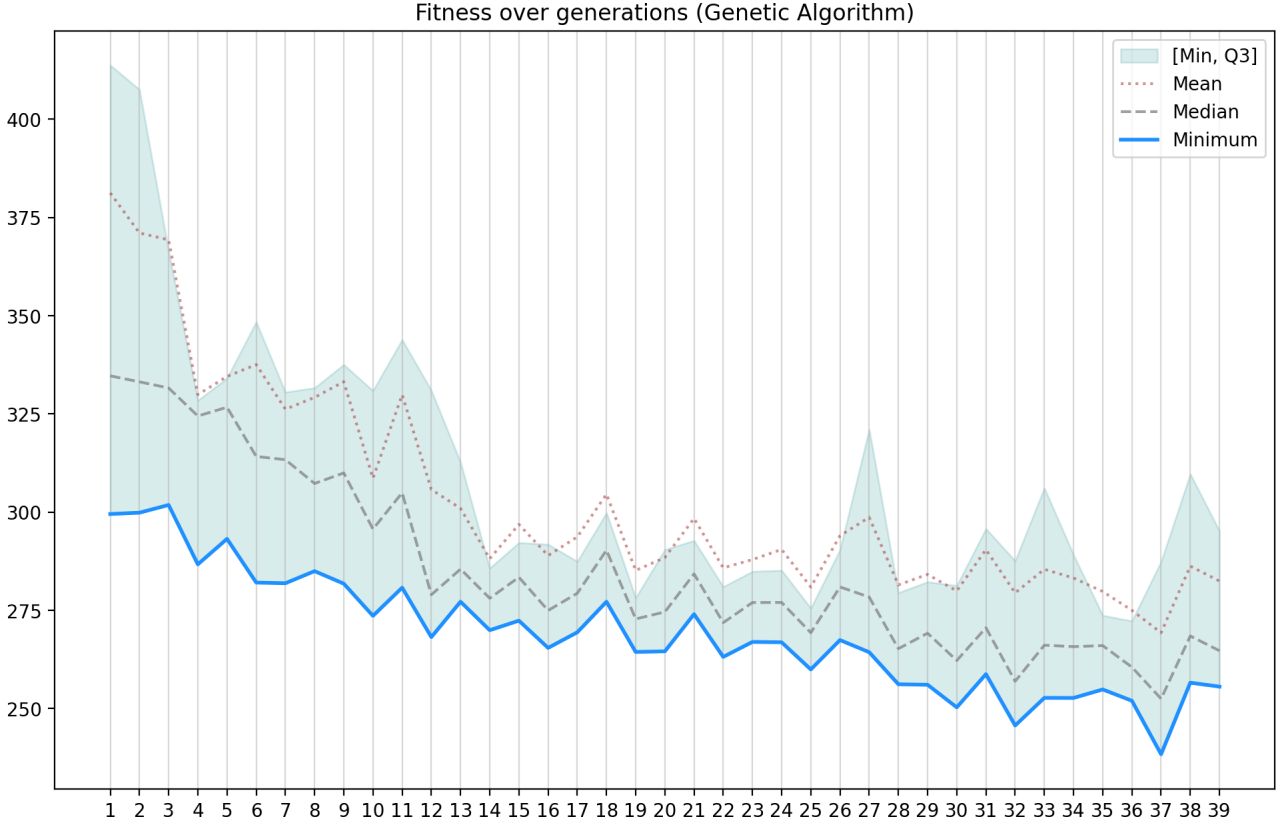


Figure 4.2: Distribution of fitness at every step of the genetic algorithm. At each step, the seeds considered are changed according to Section 4.2.

The genetic algorithm is capable of selecting for better performing individuals. Keep in mind that this plot shows the distributions *as the genetic algorithm sees them*, meaning that, at every step, the simulations seeds change. This likely makes the graph more volatile. Nonetheless, there is a tendency of improvement.

However, as it was alluded in Section 1.1, the reward model could not be used meaningfully, thus this baseline genetic algorithm is not useful. For this reason, I avoid a thorough analysis of it, while not shying away from thinking about low-hanging fruit improvements.

### 4.3.1 Possible improvements

Looking at the individuals from the population, there is likely a bloat problem, meaning that the syntax trees' depth tends to grow without bound. A possible solution to this, which would also greatly simplify the mutation operations, is to only consider populations of full trees of a given depth, rather than free-form trees.

The advantage of full trees will be explained in better detail in subsection 5.3.1, but in summary, there is no expression in a non-full tree that cannot also be expressed identically in a full tree. For what concerns genetic mutations specifically, individual generation from scratch, crossovers and point mutations become much easier to do without accidentally skewing distributions while running down the recursive structure of non-full trees.

# Chapter 5

## Auto-Encoder

In this chapter, it begins the "machine learning" part of the thesis. Auto-encoders are the first of the two neural networks that are used, and their goal is to take the input, reduce the amount of numbers used to represent it, and then reconstructing it whole from the reduced information.

Auto-encoders are a very common choice when you want to relate objects with each other, using an abstract (or "latent") space. This is usually done by supplying the auto-encoder task with additional objective functions that work on the abstract space directly.

I will discuss my implementation of the auto-encoder and its results.

### 5.1 The Idea

**Internal representation** Inspired by Hildebrandt [5], which used a middleman numerical representation to synthesize the fitness of individuals, I wanted to explore to what extent this numerical representation can be done in a differentiable context. Namely, since syntax trees are an established representation, and those are essentially strings, which can trivially be tokenized, I will consider the internal representation of an auto-encoder.

**Representation-to-fitness mapping** While Hildebrandt [5] would use discrete-friendly techniques like nearest neighbor for such mapping, a differentiable environment provided by the auto-encoder allows this mapping to be pretty much anything. After learning the

auto-encoder, a *feed-forward network* is used to model the mapping between the encoding of an individual and its fitness.

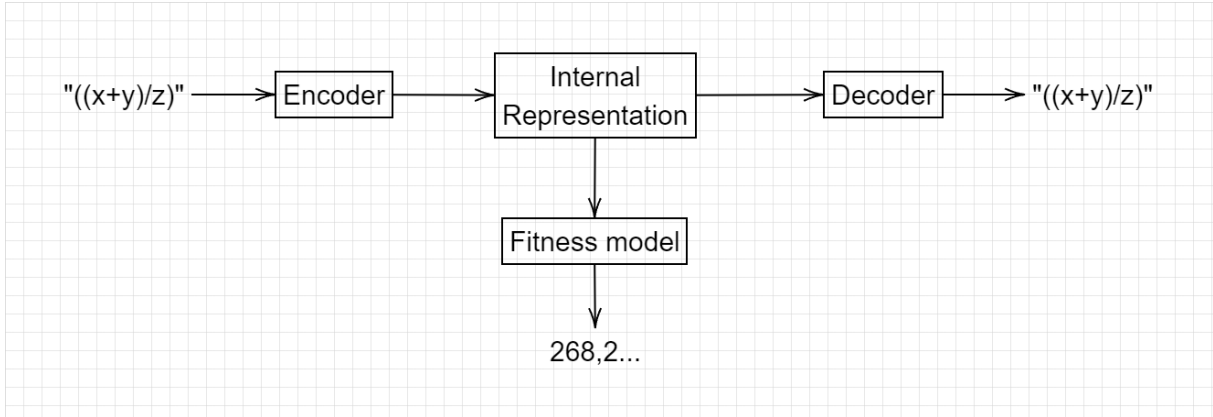


Figure 5.1: Sketch of the model.

**Optimization** After training the representation-to-fitness mapping, we would have what is essentially a *reward model*. At this point, we would optimize the value of this function.

## 5.2 General setup

### 5.2.1 Definitions

- Vocabulary  $V$ : list of all the language model's characters. Contains functional characters, parentheses, all operation characters, and all features.

$$V = \{\text{NULL}, \text{EOS}, (, ), \dots \text{operations} \dots, \dots \text{features} \dots\}$$

- Reduced Vocabulary  $V^*$ : Contains functional characters, parentheses, a placeholder operation character, and a placeholder feature character.

$$V^* = \{\text{NULL}, \text{EOS}, (, ), \text{o}, \text{F}\}$$

- Distribution(/Confidence) of characters of a sentence, at the  $i$ -th position  $p_i$ : Each  $p_i$  is a  $|V|$ -dimensional vector, and its values sum up to 1. The probability of a specific character is  $p_{ij}$ . This similarly applies to the reduced vocabulary. It is trivial to turn any  $V$ -space distribution  $p_i$  into a  $V^*$ -space distribution  $q_i$  with a matrix of size  $|V^*| \times |V|$ . I will call it *the reduction matrix*  $\mathcal{R}$ .



### 5.2.2 The Data

Since the auto-encoder only needs syntax trees and nothing else, generating data for the them to train on is trivial and inexpensive. Simply generate a syntax tree and turn it into a one-hot encoding.

The training dataset is 16384 individuals, which are completely replaced every epoch. The validation set is 16384 individuals as well, but they are never replaced. The training and validation set generators start from two different seeds.

Since the data for the autoencoder is essentially infinite, and the model never learns on the same datapoint twice, there is little risk of overfitting, which seems to be the case for the auto-encoder configurations that work.

### 5.2.3 Metrics

**Loss Function** One part of the loss function is the negative log likelihood against the true token (how confident the model is that the token is, say *the* operation +), which will be referred to as "raw criterion". Additionally, we apply the reduction matrix  $\mathcal{R}$  to the token, and consider the negative log likelihood against the true reduced token (how confident the model is that the token is, say *an* operation o), which will be referred to as "reduced criterion". The final loss function is a convex combination of the raw and reduced criterion. The weight of the reduced criterion starts higher than the raw's in the first epoch, and gradually becomes lower epoch by epoch through a rational function.

I have chosen to implement the reduced criterion to condition the learning of the model. If it's tested against getting the category right, it's essentially learning the syntax of syntax trees first, and the specific tokens later. (This approach to condition on syntax replaces a "syntax score" metric that was tried, but was found to be too hard to optimize. See Section A.1 for details.)

Depending on the network structure, the reduced criterion is learned very quickly, and gradient computation is stopped if its value is too low, to prevent numerical instability of the gradients. In these cases, the raw criterion's weight is increased faster than otherwise.

**Non-differentiable Metrics** Other metrics include:

- *Accuracy*, how many tokens are correct, in percentage;
- *Perfected*, how many sequences are 100% correct, in percentage;
- *Valid*, how many sequences are syntactically valid, in percentage.

Depending on the network structure, the *Valid* metric may not apply. Ultimately, what we care about is whether or not the *Perfected* metric is high.

### 5.2.4 Optimizer

The Adam [7] optimizer was used, with a learning rate of  $10^{-3}$  to start. The learning rate is decreased by a factor of  $\sqrt[4]{0.1}$  if the "Accuracy" metric over the validation set does not improve by  $1 + 10^{-8}$  times its highest, for 10 epochs in a row. No weight decay was used. The batch size is 64.

### 5.2.5 The "Anti-decoder"

Assume that we have a trained auto-encoder which has good performance, but does not perfectly auto-encode everything. Ultimately, we are not interested in the encoder's efficacy, but only in the numerical space it creates, and on the decoder. Since the individuals are not a complex object like images, but simply consecutive classification tasks, we may try to change the encoding to get the decoder output we want.

If we have a sequence, and its auto-encoded output does not match it, we may take its encoding and optimize it against the raw criterion.

The setup starts from the slightly incorrect encoding, and uses an Adam [7] optimizer, with learning rate of 0.001, to nudge it towards a correct one. The learning rate decreases at every step using a rational function. The optimization stops if the decoded argument matches the desired sequence. To prevent infinite loops, this optimization routine may only be done up to 50 times.

Using a feed-forward auto-encoder with a  $\approx 95\%$  Perfected metric, it takes usually less than 8 steps to find an encoding that produces the correct output. For this network, usually

only one token needs to change. This process seems to work even if the start encoding is always the zero vector, however it obviously takes much longer to get to the result.

However, even with so few steps, the process is quite slow, since it essentially has to go through the decoder multiple times. It is wise to pre-compute the anti-decoder whenever possible.

It's worth noting that such an anti-decoder is effective only because we are dealing with a relatively forgiving discrete space. If we were to use an anti-decoder on an auto-encoder trained on real-world images, for example, I have no reason to believe it will ever get to an acceptable result.

## 5.3 The Model

The simplest approach to make an auto-encoder would be to use a standard feed-forward network. Feed-forward networks are the easiest to work with, but they are also limited by the fact that they are of fixed size. To deal with this, syntax trees must also be made of fixed size. Given a fixed, maximum size, one could either fill the strings of smaller trees with NULL tokens until it matches the fixed size, or you could *fix the structure of the trees*. This latter approach was used.

### 5.3.1 Fixing the structure of the trees

Consider the tree  $((x+y)/z)$ , and say we want to bring it to a fixed size. A binary-like tree, like syntax trees, has a predictable number of elements if it's full, meaning that all bifurcations end at the same depth level. The depth of  $((x+y)/z)$  is 2, but the  $"/z$  side" of the tree ends at depth 1. In our context, we can easily increase the depth of a subtree by adding a  $+0$  to the expression, like so  $((x+y)/(z+0))$ . Now the syntax tree is full, meaning that we get to its maximal depth, 2, every way we traverse down the tree.

Similarly, we can take any syntax tree of depth  $\leq D$ , and turn it into a full syntax tree of depth  $D$ .

It's worth pointing out that, for full syntax trees of a given depth, parentheses are redundant. Since we know that  $((x+y)/(z+0))$  is full, we lose no information on the order of

operations if we strip it down to  $x+y/z+0$ . This would not be true for the non-full tree, which would become  $x+y/z$ , which might mean  $((x+y)/z)$  as well as  $(x+(y/z))$ .

It's also very simple to sample different syntax trees, if we only care about the full ones. Simply start from a template,  $((0+0)+(0+0))$  for a tree of depth 2, and change each feature or operation for a random one. This obviously also works if we remove the parentheses. Sampling only trees that are full is not a significant constraint, as it's easy to tell from the example that, for any non-full syntax tree, there is a full syntax tree whose expression is equivalent.

The facts above make it trivial to set up the autoencoder data for a feed-forward network.

### 5.3.2 The network

As input, the feed-forward network takes a full syntax tree of depth 4. Since the tree is full, I have chosen to not include parentheses, but, as it will become clear soon, they can be just as well kept and it would not increase complexity.

There is an "encoder" part and a "decoder" part. Each of these parts has hidden layers. The encoder part takes the input, has hidden layers, then a final layer, which is the output of the encoder. In the decoder, we take the encoder output as input, has hidden layers as well, and finally the output layer, which is of the same dimension as the original input. Each layer in the network has its own batch normalization layer, which is applied right after the layer's initial linear transformation. The final layer of the encoder has a tanh activation function, while all other layers between the input and output have a PReLU activation function<sup>1</sup>, one different instance for each layer. The final decoder layer also unflattens into the shape of the original input, and applies a log softmax normalization for each distribution of tokens. Any non-final layer of the encoder or decoder could apply dropout, but it has been unused since the risk of overfitting is minimal, given that training data is effectively never seen more than once.

All setups which only involve one hidden layer per side will be referred to as "shallow", while if there are any more it will be referred as "deep". In writing, the network structure

---

<sup>1</sup> $\text{PReLU}(x) = \max(x, 0) + a \cdot \min(x, 0)$ , where  $a$  is a learnable parameter. It behaves like a leaky ReLU, where the slope of the negative side is learnable. With this setup, there is one  $a$  parameter for each layer.

will be formatted as, for example, a shallow "input  $\rightarrow 2400 \rightarrow 1200$ , 1200  $\rightarrow 2400 \rightarrow$  output" autoencoder, or a deep "input  $\rightarrow 2400 \rightarrow 1200 \rightarrow 600$ , 600  $\rightarrow 1200 \rightarrow 2400 \rightarrow$  output" autoencoder.

For simplicity, the width of the hidden layers of the encoder and decoder is symmetric. For instance, a "input  $\rightarrow 2400 \rightarrow 1200$ " encoder will always have a "1200  $\rightarrow 2400 \rightarrow$  output" decoder.

### 5.3.3 Training Results

The reduced criterion is perfectly learned very quickly. This is not surprising, since the sequences all have the same structure, and the model probably "remembers" the structure as high bias terms in the last layer. Having parentheses or not in the input sequence should not affect this.

When it comes to the raw criterion, the network steadily learns to get the tokens right. Without batch normalization, smaller networks would plateau faster, but at a higher value. With batch normalization, it seems that almost all sizes of networks converge to near perfect performance. The validation loss is always below the training loss, probably due to the different modes of batch normalization between training and validation.

The feed-forward network can effectively have a high "Perfects" metric. While the input-2400-1200 network can hit 100% perfects, the input-200-100 network comes pretty close at a drastically smaller size. The latter model was ran for longer to see if reaching an exact 100% perfects was possible. The input-200-100 network was kept for further use, due to having near identical performance with orders of magnitude less parameters.

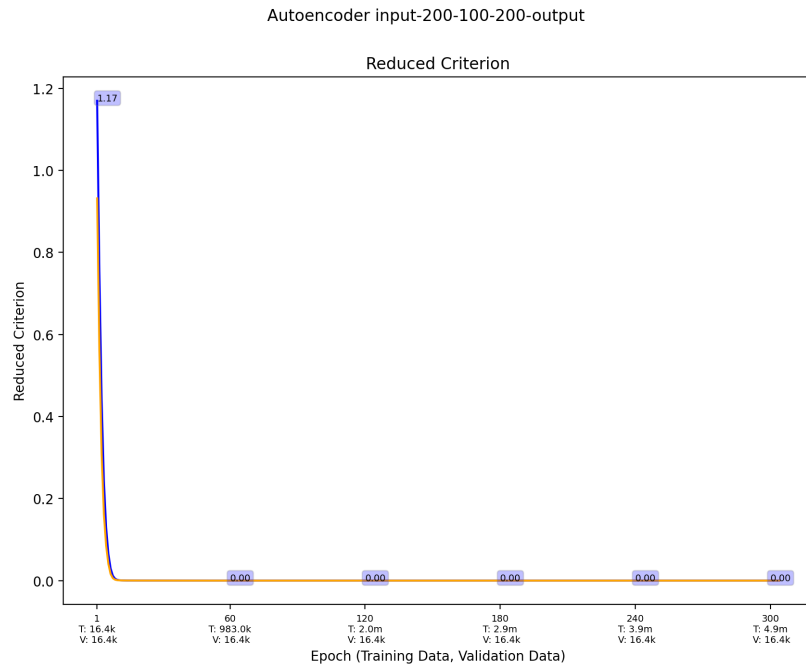


Figure 5.2: Reduced criterion of the "input  $\rightarrow$  200  $\rightarrow$  100" autoencoder.

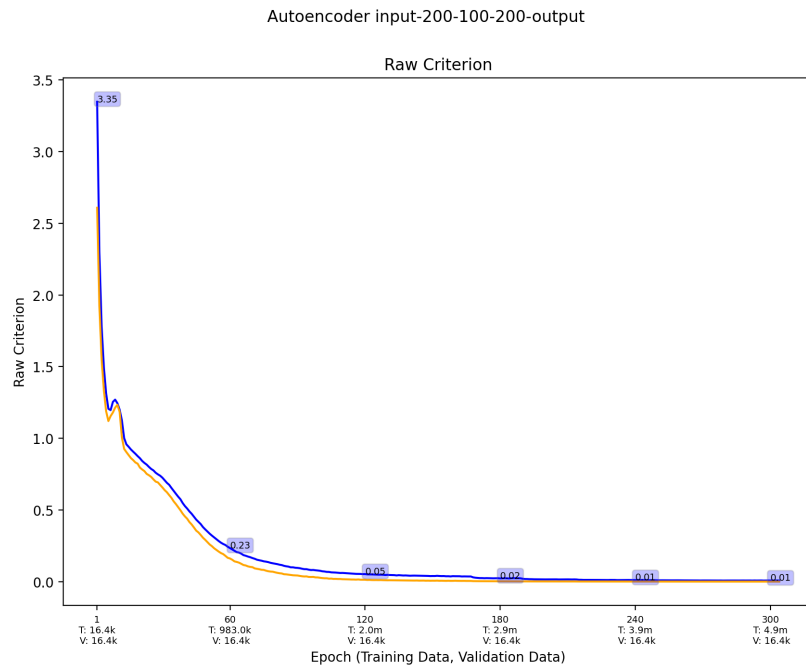


Figure 5.3: Raw criterion of the "input  $\rightarrow$  200  $\rightarrow$  100" autoencoder.

# Autoencoder input-200-100-200-output

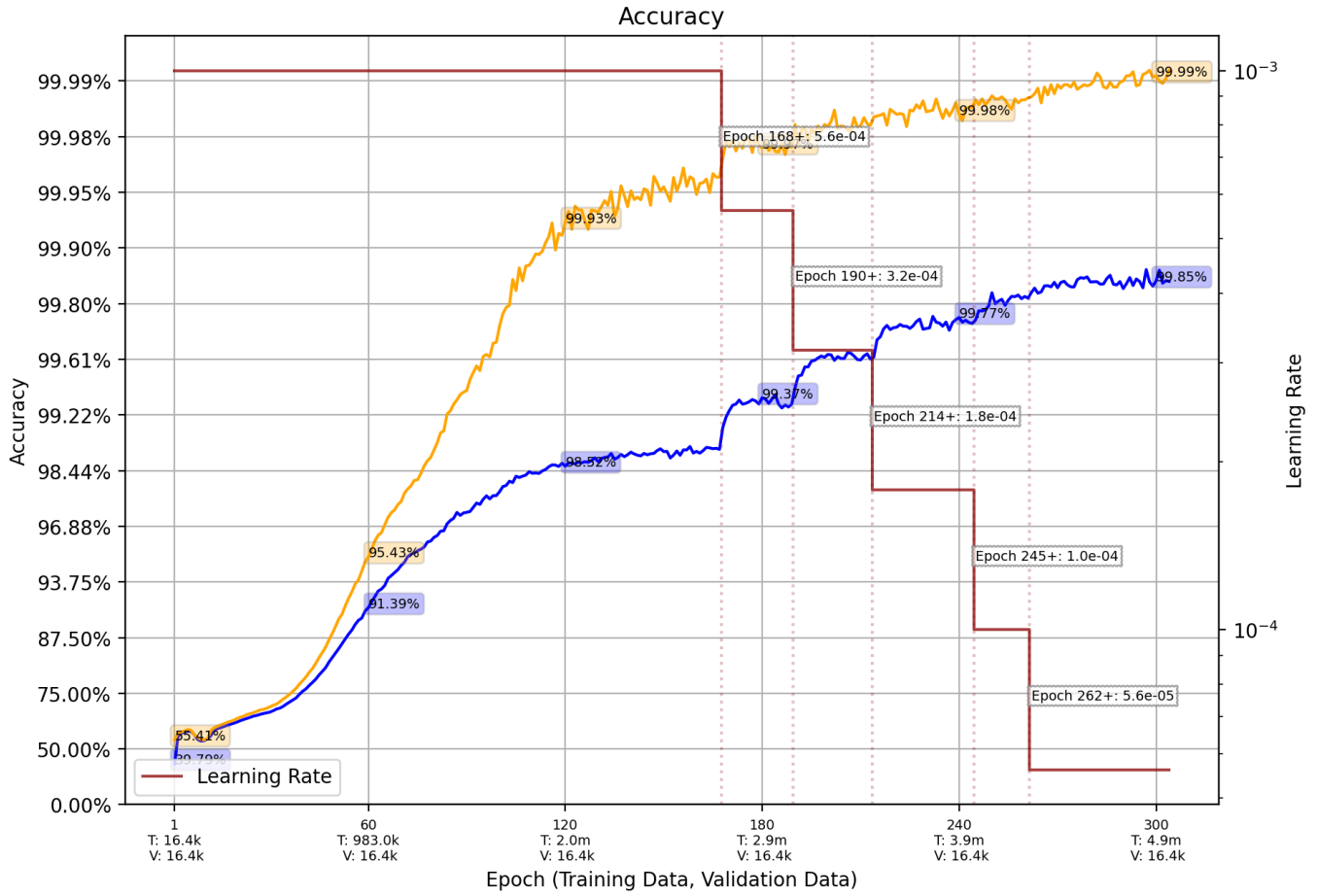


Figure 5.4: Accuracy of the "input  $\rightarrow$  200  $\rightarrow$  100" autoencoder. The learning rate is also shown. Note how right before significant improvements in learning, there was a learning rate reduction.

# Autoencoder input-200-100-200-output

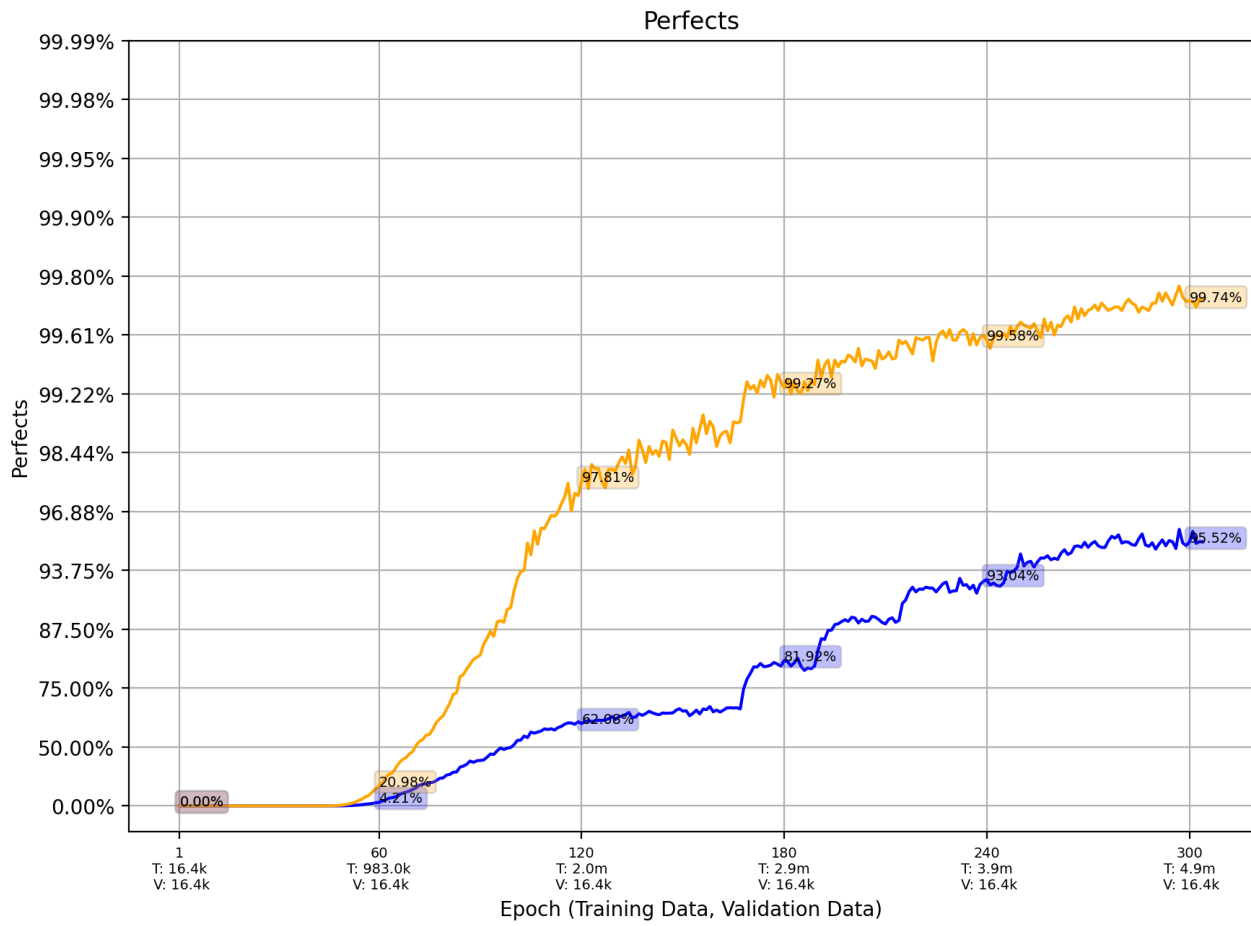


Figure 5.5: Perfects of the "input  $\rightarrow$  200  $\rightarrow$  100" autoencoder.



# Autoencoder input-2400-1200-2400-output

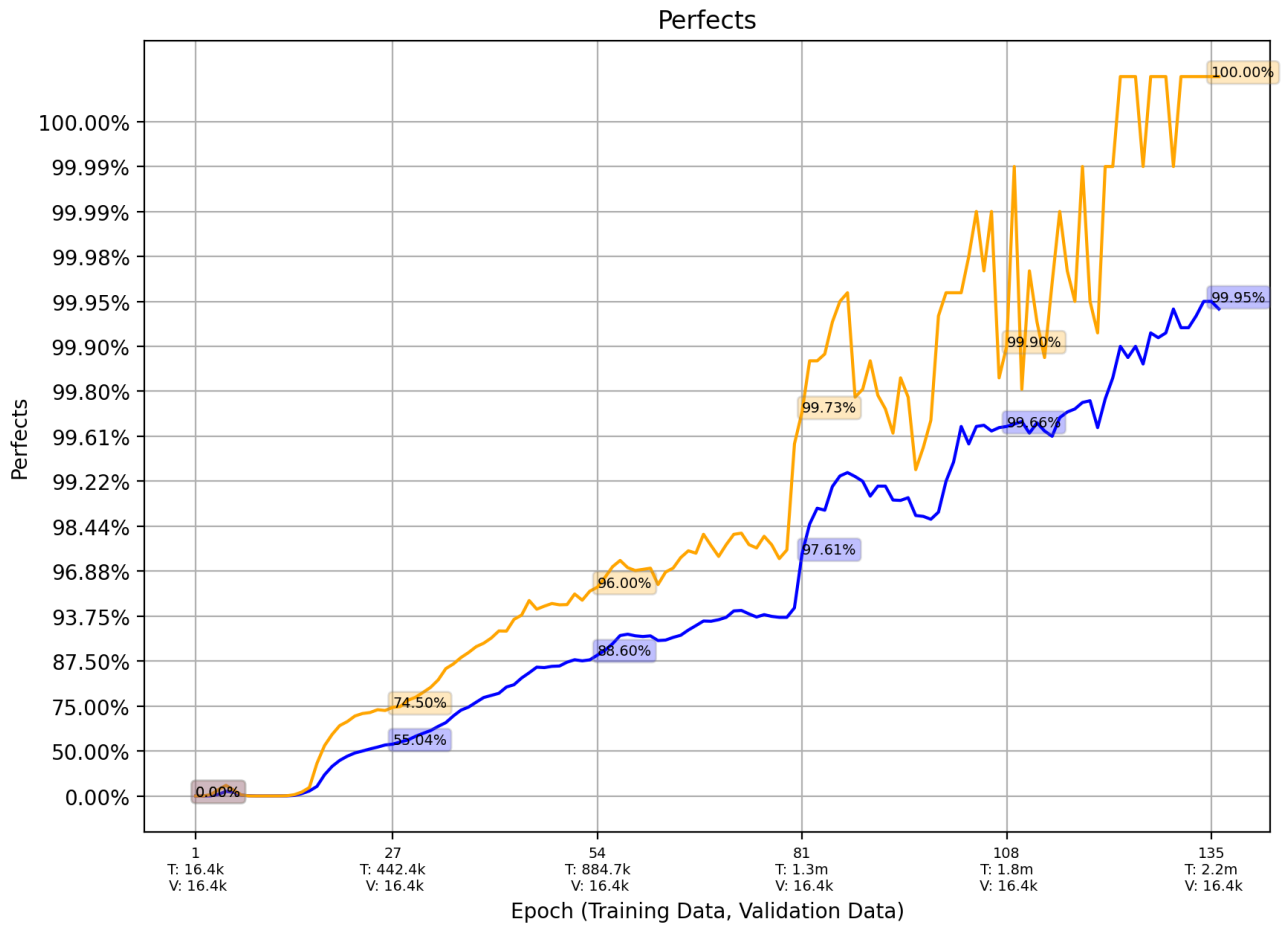


Figure 5.6: Perfects of the "input  $\rightarrow$  2400  $\rightarrow$  1200" autoencoder. The training was stopped earlier due to the validation set having been fully perfected.

## 5.4 Other results

**Test set** The model was evaluated on a test set, similarly structured with 16384 individuals and yet another seed. Its loss is similar to the validation set, with raw criterion equal to 0.0012337 and reduced criterion being smaller than  $10^{-7}$ . While the raw criterion is technically several orders of magnitude higher than the validation set’s, it is still well within acceptable values.

**Random encodings** To test the decoder’s expressiveness, random encodings were generated, to see how many collisions there would be, or if they would make a valid individual at all. 100k vectors of length 100 were generated, using independent  $U(-1, 1)$ .

- The average reduced criterion was 0.0000701. This supports the idea that the syntax is learned ”up the chain”, like in bias terms inside the decoder, and the choice of operations and features is taken from its input. While technically being several orders of magnitude higher than ”genuine” encodings, it is still very low;
- When collapsing the distributions of tokens into sequences, *all* of them differed from each other. If there is a possibility of collision, it is probably below 1 in 100k.

# Chapter 6

## Reward model

This chapter is for the other machine learning model used. Its goal is to predict a priority function's fitness in the DFJSS problem, which could then be used to find optimal priority functions. Its input is the auto-encoder's encoding of the priority functions.

If the prediction is good enough, you could use the reward model to find optimal syntax trees. Unfortunately, it fails in the task of predicting fitness, hence the latter doesn't follow. However, its failure is a necessary step to understand how one could iron out problems.

### 6.1 The Idea

The purpose of the autoencoder of the last chapter is to reduce the dimensions of syntax trees, in order to compress the sparse signals of the one-hot encodings. Another feed-forward network is made to predict the syntax trees' fitness.

One crucial feature of the reward model is that it's trained *after* the autoencoder, and not *along* with the autoencoder. This can be useful if you want the same autoencoder to be used with different reward models (if you are predicting different rewards, for instance), but it's possible that the reward model's task becomes harder as a result. If the autoencoder and reward model were trained in tandem, assuming that both can succeed in their task, the autoencoder could have trouble being used to train another, appendaged reward model.

## 6.2 General setup

### 6.2.1 The Data

In order for a reward model to exist, it needs the following:

- A trained auto-encoder;
- A dataset of individuals, for which fitnesses have been (pre-)computed.

The trained auto-encoder is available and well performant, as explained earlier in the paper. As for the dataset, here's how it was constructed:

- Fix a set of seeds for the DFJSS simulations;
- Generate a random priority function;
- Compute that function's fitness across all possible DFJSS seeds.

More formally, the dataset is structured like panel data:

- $y_{i,j}$ : Fitness  $y$  of an individual  $i$  in simulation of seed  $j$ ;
- $\overline{y_{\cdot,j}}$ : Average fitness across individuals and a fixed seed  $j$ ;
- $\overline{y_{i,\cdot}}$ : Average fitness across seeds and a fixed individual  $i$ ;
- Each individual has its fitness value computed for all seeds, so the dataset is balanced. The total average  $\overline{y}$  can be obtained by averaging out  $\overline{y_{\cdot,j}}$ , or  $\overline{y_{i,\cdot}}$ , or by unconditionally taking the average.
- $\mathbf{x}_i$ : The (one-hot) discrete representation of the individual  $i$ ;
- $\xi(\mathbf{x}_i)$ : The encoded representation of the individual  $i$ ;
- $\mathbf{s}_j$ : Features of the simulation of seed  $j$ . These features are learnable embeddings. See Section 6.3 for details.

There are about 120k individuals, and 8 seeds. Including the across-seed averages, that is the equivalent of 9 seeds. Two thirds of the dataset includes randomly generated individuals, while the other third is individuals made up of only constants. The fitnesses of the latter group are computed only once per seed, and copied throughout. This was done because constant individuals are a significant portion of the space, and are also cheap to compute.

While it is possible, depending on the goal, to compute a function’s fitness for only a subset of the seeds, all priority functions in the dataset had their fitness computed on every seed.

The encoder of the autoencoder is used to turn the discrete-space priority function into a compressed representation. This representation is the input of the model. The response variable could either be the average fitness across all seeds, or a single seed’s fitness. In the latter case, the seed is also an input.

For the sake of flow of writing, I have said that the encoded sequence is the input. More precisely, it’s the [anti-decode] of the input, as previously introduced. Since computing the anti-decode can be slow, the representation of all the priority functions is computed in advance of training, and kept in storage, rather than being computed every time it’s required during training or validation.

Additionally, every epoch, these representations have some noise added to them (centered around the original, pre-computed representation). This noise is from a uniform distribution, ranging from  $-0.002$  to  $0.002$ . This range was chosen because it empirically did not change the resulting decode, nearly 100% of the time. Noise is added in the first place in hopes of making the model more robust by exposing it to the same datapoint from multiple locations.

### 6.2.2 Metrics

Being a regression problem, the metric(s) must be a function of the distance between the prediction and the true value. The metrics considered are the L1 loss, the L2 loss, and the smooth L1 loss [4] (which is just a reparametrization of the Huber loss [6]), which is

set to L2 behavior in a neighborhood of 50 around 0. The optimization is done on the smooth L1 loss, while the other metrics are just computed.

### 6.2.3 Optimizer

The optimizer used is pretty similar to the autoencoder training, being an Adam [7] optimizer, with a learning rate of  $10^{-3}$ . After 5 consecutive non-decreases of the validation loss, the learning rate is reduced by a factor of  $\sqrt[4]{0.1}$ .

## 6.3 The Model

The model is a simple feed-forward network. The input size depends on the autoencoder that is used, and whether or not the response variable is the across-seed average fitness or the per-seed fitness:

- If the encoder layer of such autoencoder is of width  $k$ , then the input layer is of width  $k$  to start;
- If the response variable is the across-seed average fitness, the input layer is only  $k$  wide. If not, the seed is included in the input through learned embeddings  $\mathbf{s}_j$ , and each seed will have its unique representation of width  $d$ . The input layer will be  $k + d$  wide in this case. The zero vector of this seed embedding space is reserved for the across-seed average fitness, in case it is still of interest.

**The model’s structure** There can be a variable number of hidden layers, all which have batch normalization right after the linear transformation. Consecutive layers that have the same width are residual layers, meaning that after batch normalization, the hidden layer’s input is added to the result. The final operation of each hidden layer is the PReLU activation function<sup>1</sup>, one instance for each layer. The final layer is a single neuron, followed by only the ELU [3] activation function, and adding 1 (since fitness is always positive)

---

<sup>1</sup>PReLU( $x$ ) =  $\max(x, 0) + a \cdot \min(x, 0)$ , where  $a$  is a learnable parameter. It behaves like a leaky ReLU, where the slope of the negative side is learnable. With this setup, there is one  $a$  parameter for each layer.

**Why embeddings of seeds?** The embeddings of the seeds are used in lieu of having explicit features of the simulations. You could have, for instance, the initial number of machines and jobs as a descriptor of the simulation, but any finite list of these features might be incomplete. This reward model starts from time 0 and needs to predict the fitness of the simulation at its final time, which any limited set of features may have limited capability of predicting, especially since the priority function will influence the simulation’s status over time, depending on what operations and jobs are done first, and what machines are used at what time. And last but not least, the number of features might be variable (each operation has its features, each job has a variable number of operations, there can be a variable number of jobs, etc.), which might require specialized networks to deal with. As such, an arbitrary, learnable vector representing the seed of the simulation is used, so that the network hopefully ”invents” its own useful features for the specific seed.

## 6.4 Training results

Unfortunately, the training has been unfruitful. While there is some learning, the performance which the models converge to, regardless of width and depth of the model, is not satisfactory.

**Baselines** It’s worth comparing the model(s)’ results to just the average value baseline for each seed:

Compared to just taking the mean fitness for each seed, no model performs particularly better than just predicting with the mean value. While performance is better when more than one seed is used to train the model, it is still not good enough.

There could be many factors that make this task not possible.

**The amount of data is not sufficient** A short-sighted decision that was made in the beginning was to use Python to write the DFJSS simulator. Efficiency of the code aside (even though it probably still is a factor), Python is across the board a slower language to run, taking about 3 seconds to get a single result from one priority function and one

Seed	Baseline L1	Baseline L2
<i>Averaged across seeds</i>	158.06	51351.3
0	121.27	25278.66
25	690.8	993670.83
50	23.81	1566.28
75	295.78	172319.98
100	30.53	2400.9
125	76	13695.62
150	25.4	2025.32
175	44.45	5326.12
<i>Average loss</i>	162.9	140848.33

Table 6.1: Baseline loss values of the reward model. The baseline is taking the average fitness of a particular seed,  $\overline{y_{\cdot,j}}$ , and using it to predict every instance within that seed. The data points of *Averaged across seeds* are the across-seed average of each individual,  $\overline{y_{i,\cdot}}$ , meaning that the baseline is the total average  $\overline{y}$ . The last row takes the average of the rest of columns above it.

seed. If the same program was written in any compiled language, the time required to run simulations would definitely decrease enormously, obviously increasing the availability of data.

### **The encoder’s outputs are not structured in a useful way for the reward model**

This will be unfolded in greater detail in Section 7.2, but in summary, the encoder part of the autoencoder was not constrained in any particular manner during training, resulting in an arbitrary structure that did not necessarily benefit any abstraction of the underlying algebra of the expressions.

### **Predicting the fitness from time 0 until the final time is exceptionally hard**

While Chang’s [1] DDQN method is not versatile in some ways, it’s in a position where it learns from bite-sized pieces of the problem, as opposed to having to learn the entire development of a simulation from just its end result.

### **The DFJSS environments might be too constraining**

The are a lot of parameters involved in making a DFJSS simulation. It is entirely possible that some of them might have been set half-mindedly, without realizing how much they would restrict the ”free energy” of the system.



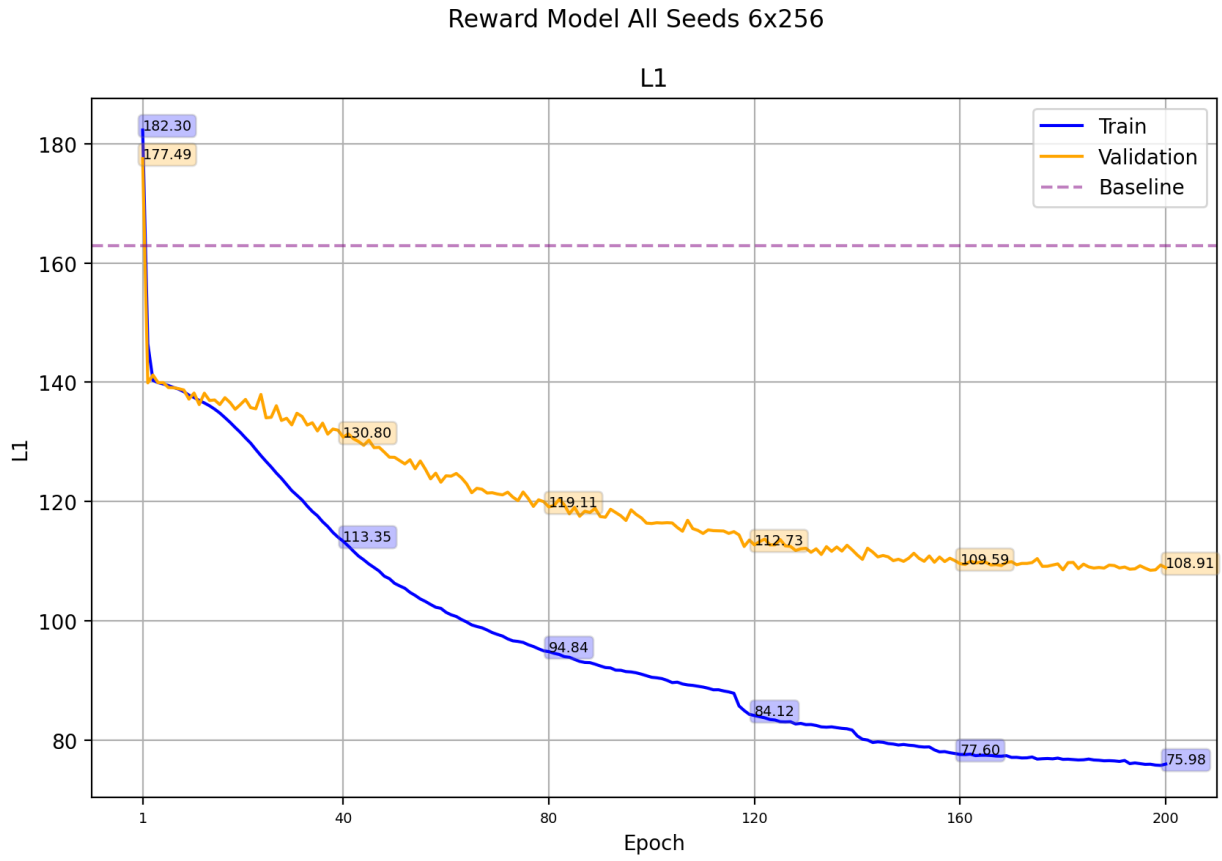


Figure 6.1: L1 Loss of a reward model with 6 layers of width 256, trained on all seeds, with an embedding width of 256. The baseline is the average loss from a random seed (including the across-seed average).

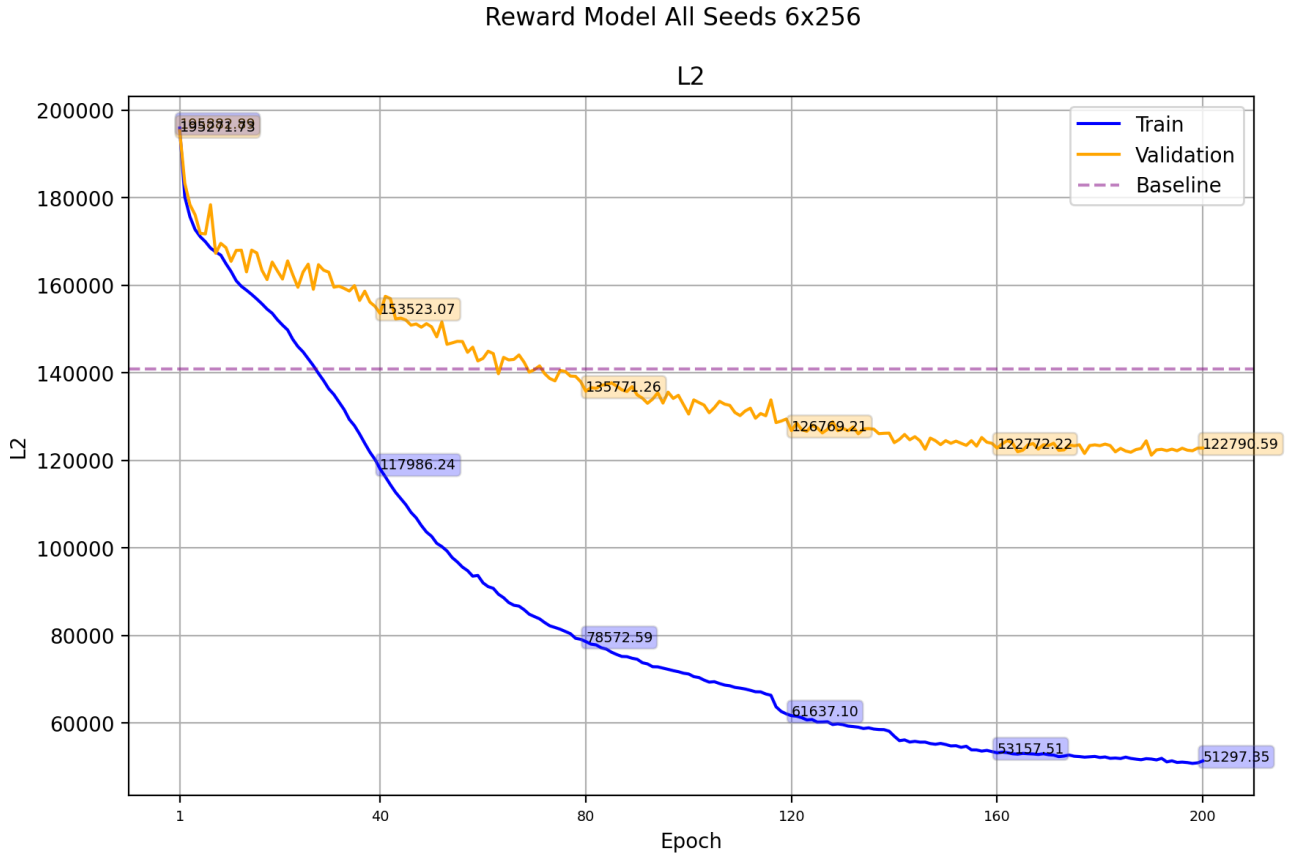


Figure 6.2: L2 Loss of a reward model with 6 layers of width 256, with an embedding width of 256. The baseline is the average loss from a random seed (including the across-seed average).

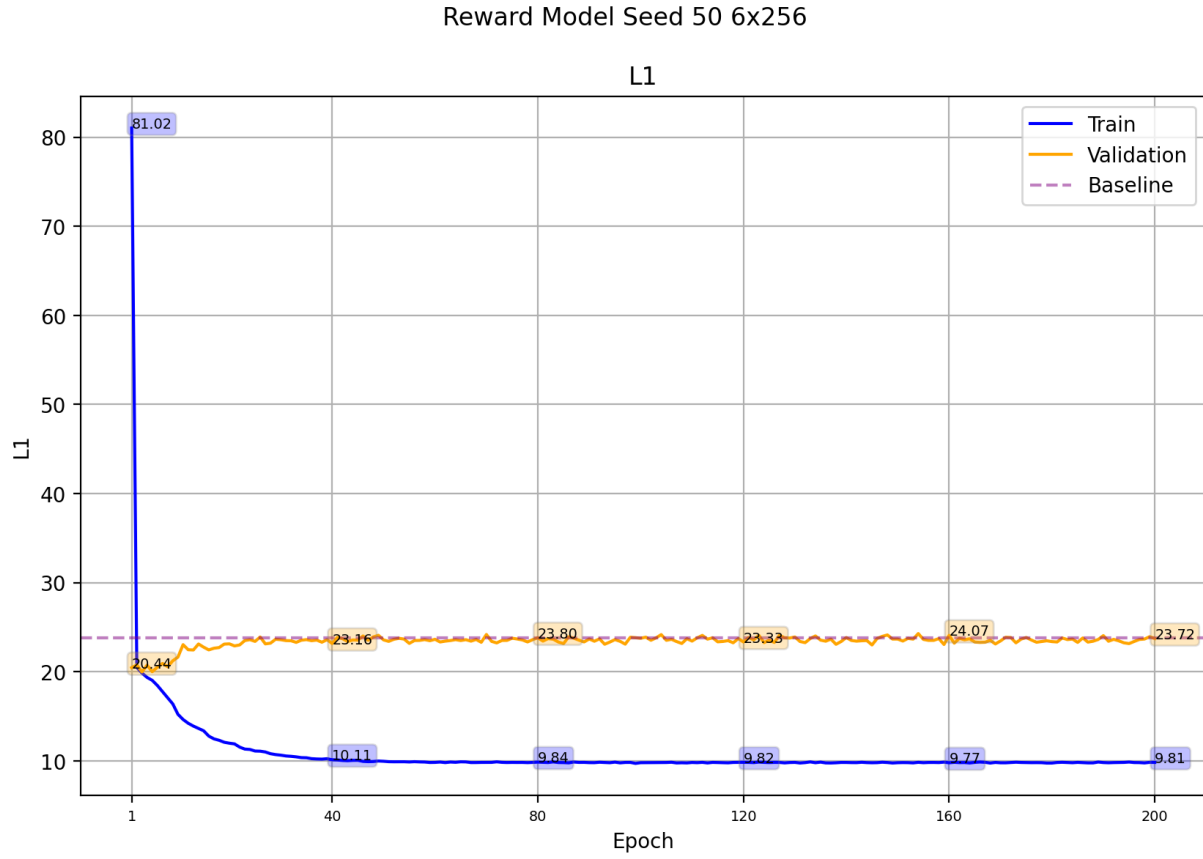


Figure 6.3: L1 Loss of a reward model with 6 layers of width 256, trained on only the seed 50, with no embeddings. The baseline is the loss resulting from using just the average of the seed 50,  $\overline{y_{\cdot,50}}$ .

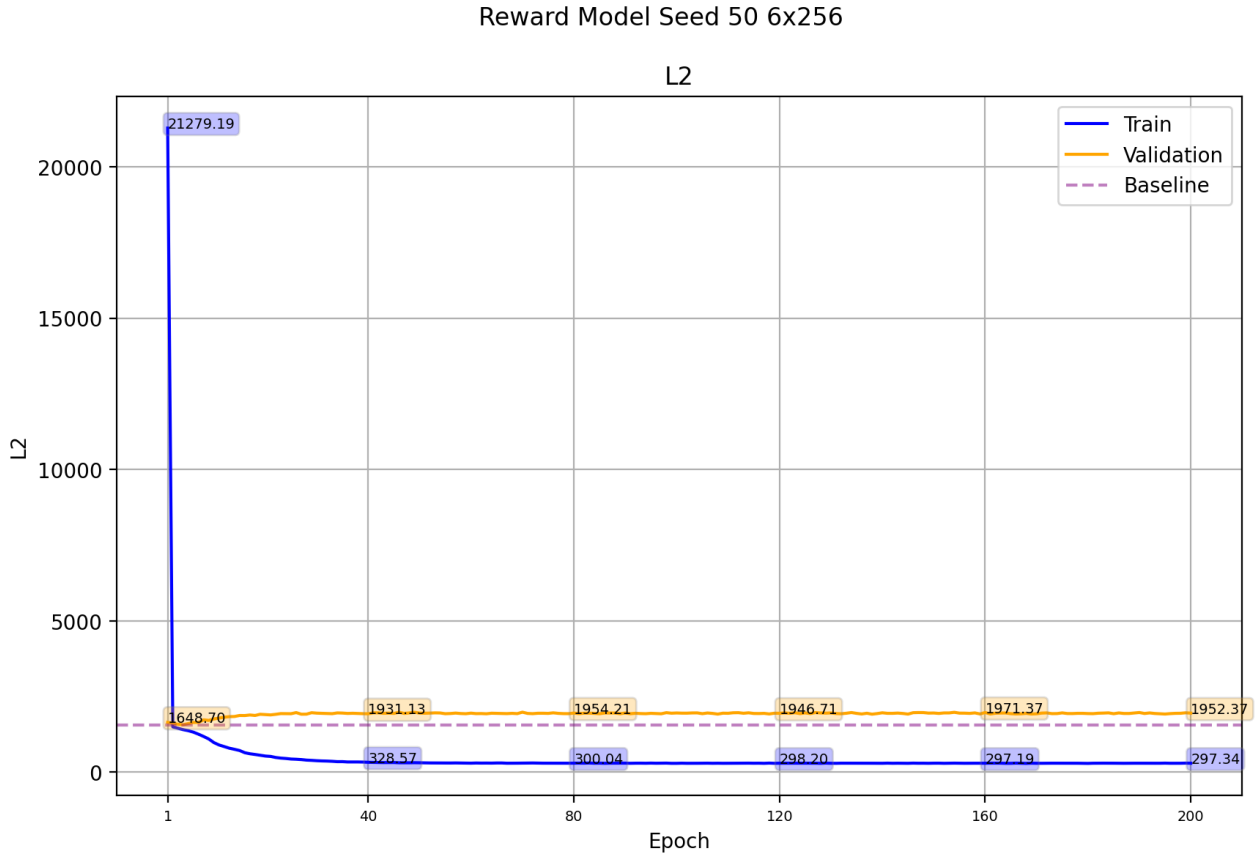


Figure 6.4: L2 Loss of a reward model with 6 layers of width 256, trained on only the seed 50, with no embeddings. The baseline is the loss resulting from using just the average of the seed 50,  $\overline{y_{\cdot,50}}$ .

## 6.5 Optimizing the reward

Since the reward model could not converge to low values for the loss functions, it's not hard to think that any optimization of its space would not be useful. Indeed, it's not.

Two optimization strategies were implemented, and the reward model used is from the epoch with smallest Smooth L1 validation loss. Since the model is not satisfactory, the optimization strategies fail.

**Optimization by gradient** One approach is to select a starting input, compute the gradient of the reward model against the input, and change the input in the opposite direction of the gradient. This is not different from the usual gradient descent in machine learning, indeed it was done with an Adam [7] optimizer. The chosen starting input was the global minimum from the reward model dataset. Upon starting the optimization steps, the model's output goes down (arbitrarily down, in fact), but the decoded input does not necessarily change. More formally:

- Let  $\mathbf{x}_0$  be the initial input,  $\xi_0 = \xi(\mathbf{x}_0)$  its encode, and  $\Delta_0 = \Delta(\xi_0)$  its decode. Let  $\mathbf{x}_i$ ,  $\xi_i$  and  $\Delta_i$  respectively be the same things at the optimization step  $i$ ;
- Let the reward model be  $R : \xi \rightarrow \mathbb{R}$ , and  $R_i = R(\xi_i)$ . Importantly, assume  $R_0 \ggg 1$ ;
- At every optimization step  $i$ , compute  $\frac{\partial R}{\partial \xi_i}$  and optimize according to the Adam optimizer. At every optimization step, the proposed solution is  $\Delta_i = \Delta(\xi_i)$ ;
- Over time, for a given optimization step  $I$ , we have  $R_I \lll 1$ ,  $|\xi_0 - \xi_I| > 0$  and  $\Delta_0 = \Delta_I$  simultaneously.

This phenomenon is not isolated to  $\mathbf{x}_0$  being the global minimum of the dataset. The best way I can describe this behavior is by saying the reward model is "spongy": like a sponge, while there is solid space near every point you look, it's also mostly holes you can fall into.

**Optimization by Simulated Annealing (SA)** SA is an optimization technique which behaves like stochastic hill climbing. Rather than deterministically going in the direction determined by the gradient, a direction is proposed, and the better it is, the more likely it is to be followed.

- Let  $\hat{y}_i$  be the value of the reward model at an SA optimization step  $i$ , and  $\xi_i$  its argument.
- There are a finite amount of steps  $I = 50000$ , and a "temperature" parameter  $T_i = 1 - \frac{i}{I}$ . The role of this parameter is to make the optimization steps less erratic as it is running out of steps.
- At every step, there is a "proposal"  $\xi_i^P$ , with value  $\hat{y}_i^P$ . The proposal is "accepted" with a probability, given by  $P(\text{Accepting proposal}) = \min\left(\exp\left(\frac{\hat{y}_i - \hat{y}_i^P}{T_i}\right), 1\right)$ . If the proposal's value is lower than the current one, the proposal is always accepted. If not, it's accepted with a probability. If it's not accepted, the argument does not change. As  $T_i$  gets closer to 0, the acceptance probability will become more extreme, getting closer to a pure "hill-climb / stay-still" schedule.
- The proposal  $\xi_i^P$  is of the form  $\xi_i^P = \alpha\xi_i + (1 - \alpha)\tilde{\xi}_i$ , where  $\alpha = 0.5$  and  $\tilde{\xi}_i$  is equal to  $\xi_i$ , but with one value of the vector changed to a random value from the distribution  $U(-1, 1)$ .

This optimization schedule is more robust than gradient-based hill climbing, as it doesn't care about steep gradients on "rough" objective functions. Given the reward model at hand, however, it suffers from the same problem as the optimization by gradient.

# Chapter 7

## Discussion and Future Work

### 7.1 Main Takeaways

While the result has been negative, there is no denying that a lot has been done. In order of appearance:

- I have written a DFJSS library from scratch. As opposed to finding a pre-made solution, doing it myself made it abundantly clear to me how there is no *one* "logistics problem", but even the most subtle changes of assumptions can have branching consequences;
- I have written a genetic algorithm to select through priority functions. Frankly, having to write it on my own made me wonder how it's even possible that genetic algorithms ever work at all!
- Had to autonomously learn PyTorch in order to make the autoencoder and reward model. Looking at how I wrote my networks at first, there is a considerable improvement in quality;
- Learned the hard lesson that just because it has a gradient, it doesn't mean that it's useful. Section A.1 defines a differentiable score function that is high when a sequence of tokens was close to, or exactly, a syntactically correct tree. Between its inefficiency and complex path to the final number, it caused more harm than good.

- Having intimately dealt with those two networks, while ultimately not being able to make the concept work within the time frame of the thesis, was a trial of fire for many different ideas. I will discuss the most solid idea among them in Section 7.2, which I hope I'll be able to develop in the future.

## 7.2 Algebraically Aware Encoding

The likely underlying problem with the approach, at least on the modeling side, is that the autoencoder has no incentive to have an algebraically coherent encoding space. For instance, the encoding of the tree  $(a+b)$  should be related to the encoding of  $(a+(0.8*b))$ , since the fitness of these two syntax trees are probably similar, but the autoencoder, as it is set up, has no reason to conceptualize those two trees as "close" in the encoding space. Instead, the easiest representation for the autoencoder to learn would be one that places trees with similar characters in similar positions closer to each other. This probably makes the task of the reward model incredibly harder.

**Encoder loss functions** There have been attempts to do self-supervised training on the encoder. One objective of this training was to make the cosine similarity of two trees' encodings match the correlation of the priority values of the two trees. For instance,  $(a+b)$  and  $(a+(0.8*b))$  would have highly correlated priority values, 0.95 for example, and the training objective would be that the cosine similarity of the two encodings should be close to 0.95. However, the feed-forward encoder was essentially unable to learn, not performing much better than an uninitialized encoder.

**Many more questions than answers** Many other questions came up during this naive, first approach:

- Is a simple feed forward network enough? Does this problem require a more advanced network?
- Is the loss function chosen a good choice? Whether or not it is, would you be able to answer if the network chosen is not the right one?



- Is the "linear" Pearson correlation expressive enough as a measure of "true" similarity to begin with?

### 7.3 Training the autoencoder and reward model in tandem

An easier approach which might not need self-supervised learning of the encoder is to train both models at once. As previously mentioned, that comes with the disadvantage that you cannot then use the autoencoder to train another reward model, but since *no* reward model can be trained from the lone-trained autoencoder to begin with, *one* trained reward model is a better outcome.

Speaking of different network structures, since syntax trees are a sequence of tokens, one could use a transformer as an autoencoder. An autoencoder with no further constraint has no reason to have a coherent structure, but by constraining a transformer with the fitness predictions, it could be possible for the transformer's attention to learn some algebra-like associations that would help with the problem in Section 7.2.

### 7.4 Changing the parameters of the DFJSS problem

Another possibility is that the parameters that determine the possible ranges of all the features, when combined, result in a DFJSS problem that is hard to optimize to begin with. For instance, the xkcd [9] setup for families and recipes might be too constrained, allowing too few alternatives to machine assignments, and forces even the best performing priority functions to have little control most of the time. To be sure about this, it would require a thorough analysis of not only the current (and alternative) parameters of the simulations, but also of other fitness metrics that were not really looked at.

With that said, even if the current setup results in a hard problem, it doesn't mean that optimization of the problem is useless. What it means is that it would potentially be harder to tell better optimization strategies apart.

# Chapter 8

## Conclusion

In this paper, I considered the DFJSS problem, being explicit in its assumptions and implementation. The decision object that was considered for the problem were priority functions, which look at all the features of jobs, operations, and machines, and decide how machine assignment/operation sequencing looks like by combining them into a function where the highest-value arguments are the tasks done first.

After a discussion on what assumptions on the logistics problem, and the decision object, might entail, I introduced a genetic algorithm as a baseline optimization to the problem. I then focused on priority functions as a sequence of tokens, constructing a feed-forward auto-encoder, to reduce their dimensionality and to map them into a continuous space. While the way in which the autoencoder organises the latent space is probably not useful, the reconstruction by the autoencoder is successful.

On top of the auto-encoder's latent space, I trained a reward model to map a priority's function latent representation into its fitness of the logistics problem. This training was not fruitful, for possible reasons ranging from not enough data, unusable latent space, or harsh conditions for the DFJSS problem.

The final idea would have been to navigate the space of this reward model, looking for the best performing priority function. However, any optimization strategy for this reward model could not, understandably, accomplish much, since the reward model itself could not model the task at hand.

A lot of possible opportunities for future work were found. The one that piqued my

interest the most was a way of arranging the auto-encoder’s latent space in such a way that it relates algebraically correlated priority functions, rather than arbitrarily. There is also the possibility of training the auto-encoder and reward model together, especially if a transformer is used for the former, as it might be able to show its muscles when also tasked with the prediction of fitness of the priority functions. Marginally, exploring different DFJSS environments, to see whether or not I was dealing with a harsh one the whole time, would also enrichen the analysis of the various strategies used.

# Appendix A

## Unused concepts

### A.1 Syntax Score

Since the auto-encoder need to be able to produce strings with a strict syntax, it is worth considering implementing a metric of how well they are able to do so. While this could be done by simply checking if a string can be converted to a priority function object in my Python implementation, it would be an "out-of-network" metric<sup>1</sup>, not too dissimilar to a naive accuracy metric in a classification setting. Not to mention, even if a language model outputs a valid syntax tree, it might be helpful to distinguish between a confident generation of that syntax tree (very high confidence for each token) and a non-confident one (valid sequence, but the correct tokens were barely the ones with highest confidence). Finally, making use of the [0,1] range and gradients, we could even inform the network that syntax is important, and not just the autoencoding. Granted, a good autoencode would respect syntax anyway, but hopefully making the network learn syntax outside of its training data might improve its performance on test data. In the same way accuracy was generalized into cross-entropy for classification problems, to what extent can we turn a yes-no metric of valid syntax into a differentiable one?

First of all, for what concerns syntax, the specific operation or feature that is used does not matter, so we use a "reduced" vocabulary  $V^* = \{\text{NULL}, \text{EOS}, (, ), o, F\}$ , where operations

---

<sup>1</sup>By "out-of-network metric" I mean any metric that informs the human on model performance, but it doesn't play a role in the optimization process. It might be by design, but it might also be because the metric itself cannot communicate with a neural network directly.

and features are replaced with standins  $\circ$  and  $F$ . For instance,  $((x+y)/z)$  turns into  $((FoF)\circ F)$ . It is trivial to turn any  $V$ -space distribution  $p_i$  into a  $V^*$ -space distribution  $q_i$  with a matrix of size  $|V^*| \times |V|$ .

An important idea for syntax evaluation is the recursive construction of syntax trees. Differentiability aside, if we can replace any instance of  $(FoF)$  with  $F$ , until all that is left of the sentence is  $F$ , then the sentence has valid syntax. If there are no  $(FoF)$  left in the sentence and the sentence is not  $F$ , the sentence is not valid.

Table A.1: Example of a valid and invalid syntax tree, evaluated by replacing any  $(FoF)$  with  $F$ .

$((FoF)\circ F)$	$((FoF)\circ(FoF))$
$(FoF)$	$((FoF)\circ F)$
$F$	$(FoF)$
$\Rightarrow$ Valid	$\Rightarrow$ Not Valid

### A.1.1 Definition

To translate this in a differentiable context, we would like a way to detect sequences of characters that are  $(FoF)$  with confidence. Consider the following mask:

	NULL	EOS	(	)	$\circ$	$F$
0	0	0	<b>1</b>	0	0	0
1	0	0	0	0	0	<b>1</b>
2	0	0	0	0	<b>1</b>	0
3	0	0	0	0	0	<b>1</b>
4	0	0	0	<b>1</b>	0	0

Table A.2: Convolution-like mask that is sensitive to  $(FoF)$ .

If we take a sequence of five characters and use the following mask (of the 0-th character, we take the confidence of  $($ , of the 1-st character we take the confidence of  $F$ , et cetera), we obtain a vector of five numbers, independently between 0 and 1. If we were to reduce these five numbers into one somehow, we would have a differentiable measure of "how

much (FoF)” a quintet of characters is.

This is the chosen reduce function, nicknamed ”pegged geometric mean”:

$$r(x = \{x_s\}) = \sqrt[|x|]{\prod_{x_s} (x_s + \epsilon)} - \epsilon, \epsilon := 0.1 \quad (\text{A.1})$$

Figure A.1: Definition of the pegged geometric mean.

If the confidence of each character is close to 1, the final number is close to 1. If any of them is close to 0, the final number is decreased significantly. The  $\epsilon$  ”pegs” the geometric mean to avoid infinite gradients. For simplicity, we will refer to this reduced quantity as ”the confidence of (FoF)”.

We can now apply said operation to an entire sequence, five characters at a time, from the beginning to the end. If the sequence is  $k$  tokens long, we would obtain  $k - 4$  numbers, independently between 0 and 1.

$$\begin{aligned} & ((\text{FoF}) \circ (\text{FoF})) \\ & \Downarrow \\ & [0.01.., \mathbf{0.93}.., 0.02.., 0.01.., 0.03.., 0.04.., 0.01.., \mathbf{0.97}.., 0.01..] \end{aligned}$$

Figure A.2: Example of running the (FoF) confidence across a sequence of highly confident tokens.

Now we would like to replace the detected (FoF)s with F. This will be done one at a time, and will be done at the most confident instance of (FoF). The confidence of the replacing F is equal to the confidence of the (FoF) it’s replacing. In parallel, we store said value in a list  $L$ . From this point on, we just repeatedly apply the previous steps, and store the values along the way, until we are left with a lone F and its confidence. At this point,  $L$  stores the confidence scores all the (FoF) we replaced. *The syntax score is the pegged geometric mean of  $L$ .*

$$\begin{aligned}
((\text{FoF}) \circ (\text{FoF})) &\rightarrow [0.01\ldots, 0.93\ldots, \ldots, \mathbf{0.97\ldots}, 0.01\ldots] & L &= [0.97\ldots] \\
\rightarrow ((\text{FoF}) \circ \text{F}) &\rightarrow [0.01\ldots, \mathbf{0.93\ldots}, 0.02\ldots, 0.01\ldots, 0.03\ldots] & L &= [0.97\ldots, 0.93\ldots] \\
\rightarrow (\text{FoF}) &\rightarrow [\mathbf{0.96\ldots}] & L &= [0.97\ldots, 0.93\ldots, 0.96\ldots] \\
\implies \text{SyntaxScore} &= r(L) = 0.95\ldots
\end{aligned}$$

Figure A.3: Example of computing the syntax score for a highly confident  $((\text{FoF}) \circ (\text{FoF}))$ .

**Remarks** Note that we assume that a  $(\text{FoF})$  is always present, and by doing so we might replace a non- $(\text{FoF})$  sequence with  $\text{F}$ . However, this would only happen if there are no  $(\text{FoF})$ , and whatever non- $(\text{FoF})$  quintet of characters we replace will have a low confidence for  $(\text{FoF})$ , which will be carried along, lowering the final score.

The syntax score is  $r(L)$ , rather than being, for instance, just the confidence of the last  $(\text{FoF})$  we encounter, because it is more robust. If at any point we made a non-confident replacement, it is remembered, and it will lower the score.

To avoid undefined behavior, the length of the sequence must be  $1 \bmod 4$ . In practice, if a sequence is not of that length, it is truncated to be that length. Valid sequences are always  $1 \bmod 4$  (this is trivial to prove by induction), so if any truncation is involved, it's most likely from an invalid sequence. At most, we might have a sequence of length  $2 \bmod 4$ , where the last character is  $\text{EOS}$ , but that character indicates truncation anyway, so it does not matter.

### A.1.2 Problems

While this idea might be interesting, it was eventually scrapped, for the following reasons:

- It was hard to write in an efficient fashion;
- The reduced criterion (as described in subsection 5.2.3) is much more efficient and serves the same purpose;
- Gradient descent could not meaningfully optimize it.

# Bibliography

- [1] Jingru Chang et al. “Deep Reinforcement Learning for Dynamic Flexible Job Shop Scheduling with Random Job Arrival”. In: *Processes* 10.4 (2022). ISSN: 2227-9717. DOI: 10.3390/pr10040760. URL: <https://www.mdpi.com/2227-9717/10/4/760>.
- [2] James C. Chen et al. “Flexible job shop scheduling with parallel machines using Genetic Algorithm and Grouping Genetic Algorithm”. In: *Expert Systems with Applications* 39.11 (2012), pp. 10016–10021. ISSN: 0957-4174. DOI: <https://doi.org/10.1016/j.eswa.2012.01.211>. URL: <https://www.sciencedirect.com/science/article/pii/S0957417412002394>.
- [3] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. 2016. arXiv: 1511.07289 [cs.LG].
- [4] Ross Girshick. *Fast R-CNN*. 2015. arXiv: 1504.08083 [cs.CV].
- [5] Torsten Hildebrandt and Juergen Branke. “On Using Surrogates with Genetic Programming”. In: *Evolutionary computation* 23 (June 2014). DOI: 10.1162/EVC0\_a\_00133.
- [6] Peter J. Huber. “Robust Estimation of a Location Parameter”. In: *The Annals of Mathematical Statistics* 35.1 (1964), pp. 73 –101. DOI: 10.1214/aoms/1177703732. URL: <https://doi.org/10.1214/aoms/1177703732>.
- [7] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].



- [8] J. Kuhpfahl and C. Bierwirth. “A study on local search neighborhoods for the job shop scheduling problem with total weighted tardiness objective”. In: *Computers & Operations Research* 66 (2016), pp. 44–57. ISSN: 0305-0548. DOI: <https://doi.org/10.1016/j.cor.2015.07.011>. URL: <https://www.sciencedirect.com/science/article/pii/S0305054815001860>.
- [9] Randall Munroe. *xkcd/2420*. Feb. 2021. URL: <https://xkcd.com/2420/>.
- [10] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. (With contributions by J. R. Koza). Published via <http://lulu.com> and freely available at <http://www.gp-field-guide.org.uk>, 2008. URL: <http://www.gp-field-guide.org.uk>.
- [11] Jin Xie et al. “Review on flexible job shop scheduling”. In: *IET Collaborative Intelligent Manufacturing* 1.3 (2019), pp. 67–77. DOI: <https://doi.org/10.1049/iet-cim.2018.0009>. eprint: <https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-cim.2018.0009>. URL: <https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-cim.2018.0009>.
- [12] Fangfang Zhang et al. “Evolving Scheduling Heuristics via Genetic Programming With Feature Selection in Dynamic Flexible Job-Shop Scheduling”. In: *IEEE Transactions on Cybernetics* 51.4 (2021), pp. 1797–1811. DOI: 10.1109/TCYB.2020.3024849.
- [13] Liu Feng Zhong Yuguang Yang Fan. “Solving multi-objective fuzzy flexible job shop scheduling problem using MABC algorithm”. In: *Journal of Intelligent and Fuzzy Systems* 36 (Mar. 2019). DOI: 10.3233/JIFS-181152.