Technische Universität Berlin

MASTER THESIS

A novel approach to Job Scheduling applying Deep Reinforcement Learning and Neural Network Architectures of Natural Language Processing

Diego de Oliveira Hitzges

Mathematics
364296

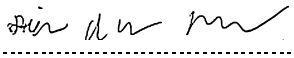
01. April 2023

Faculty II | Mathematics and Science
Institute of Mathematics
Research Group for Algorithmic and Discrete Mathematics

Supervision by:
Dr. Guillaume Sagnol
Prof. Dr. Martin Skutella

Affidavit

I hereby declare that I have prepared this thesis independently and on my own, without any unauthorized external assistance and exclusively using the sources and aids listed.



Diego de Oliveira Hitzges

Berlin, 01.04.2023

Contents

1 Introduction **7**

2 Job Scheduling Problems **12**
2.1 Problem formulation 15
2.2 Related Work 17
2.3 Our Approach 18

3 Deep Reinforcement Learning **21**
3.1 Markovian decision problems 21
3.2 Q-learning and its convergence 23
3.3 Deep Q-learning 27

4 Neural Network Architectures of Natural Language Processing **30**
4.1 Word Embedding 30
4.2 Recurrent Neural Networks 32
4.3 Vanishing/Exploding Gradients 41
4.4 Long Short-Term Memory 43
4.5 Attention 48
4.6 Pointer Networks 50
4.7 Transformers 52

5 Problem Embedding **58**
5.1 States and Actions 59
5.2 Policies 63
5.3 Action Values 65
5.4 Target Values 69
5.5 Normalization 71

6 Network Architecture **77**
6.1 Overview 77
6.2 Preprocessing 77
6.3 Input 79
6.4 Sequential Embedding 79
6.5 Transformer Encoder 82
6.6 Action-Pointer Decoder 83

7 Data Creation **87**
7.1 Supervised Job Scheduling Problems 87
7.2 Deeply Reinforced Job Scheduling Problems 90

8 Training and Testing **94**
8.1 Supervised Training 94
8.2 Supervised Testing 95
8.3 Deeply Reinforced Training 96
8.4 Deeply Reinforced Testing 97

9 Results **99**

10 Conclusion and Future Work **101**

11 References **104**

List of Figures

1 Schedule minimizing Makespan on Identical Machines 15
2 Target Network in Deep Q-learning [15] 29
3 Neural Network with Word Embedding 32
4 One-to-One Mapping 32
5 Zero-Padding 33
6 Output Sequence 34
7 Feeding Network one-by-one 35
8 Simple Recurrent Neural Network 37
9 Bidirectional RNN 39
10 Stacked RNN 40
11 LSTM Unit 45
12 Forget Gate 46
13 Input Gate and Candidate Memory 46
14 Output Gate 47
15 Pointer Network 50
16 Encoder of Transformer [47] 53
17 Transformer [47] 55
18 State to Data 61
19 Normalization of Target Values 72
20 Normalization of Data 73

21 Data to Input 79
22 Sequential Embedding of a Job's Machine Resource Environment 80
23 Embed Urgencies of Jobs 81
24 Action-Pointer Decoder 85

List of Tables

1 Embedding Layers 82
2 Encoder Layers 83
3 Decoder Layers 86
4 Data Results of Supervised Learning 95
5 Policy Cost Results from Supervised Learning 96
6 Data Results of Deep Reinforcement Learning 97
7 Comparing Policy Costs from Supervised Learning to DRL . 98
8 Comparing Policy Costs from NN to Scheduling Algorithm . 99

Abstract

This master thesis presents a novel approach to Job Scheduling based on Deep Reinforcement Learning. It treats the Job Scheduling Problems of deterministically and non-preemptively scheduling on Unrelated Parallel Machines. These are allowed to be initially occupied and can be deactivated during the process. The objective is to minimize the sum of the makespan together with the weighted tardinesses of the Jobs and Machines. The problem structure is embedded into a Deep Reinforcement Learning environment. Policies are identified as scheduling algorithms. Any ensuing sequence of state-action pairs denotes a feasible schedule. The sum over their transition costs equals the respective scheduling costs. Each state is associated with an instance in which a Machine becomes unoccupied. The assignments of pending Jobs form the set of feasible actions, together with the decision to turn it off in case that there is at least one more Machine operating. A sophisticated Neural Network is presented that induces an estimation of the optimal policy, corresponding to a least-cost schedule. It is based on Natural Language Processing Architectures. The Jobs and Machines get processed as time series by an LSTM and subsequently encoded by a Transformer Encoder. An Action-Pointer Decoder then points to the feasible action estimated to be optimal. Consequently, the Neural Network is able to process any number of Jobs and Machines. It is trained and tested supervisedly by computing the scheduling costs for Job Scheduling Problems of 8 Jobs and 4 Machines. Applying it to unseen problems of the same conditions induces schedules that yield additional costs of 2.51% relative to the optimal ones. When comparing it to standard heuristics on Job Scheduling Problems with up to 100 Jobs and 15 Machines, the Neural Network significantly outperforms for up to 100 Jobs and 10 Machines and vastly outperforms in the range of 3 to 8 Machines and up to 30 Jobs. Under these circumstances, applying a combinatorial scheduling algorithm in general results in average costs 20% to 40% higher than with the use of the Neural Network. Further Deep Reinforcement Learning based techniques are presented to enable the training of the Neural Network on higher numbers of Jobs.

1 Introduction

Mathematical optimization problems are often not only interesting from a theoretical perspective but also from a practical one, since they might resemble situations occuring in the real world where one would want to take decisions based on the goal of optimizing some cost or reward related objective. Job Scheduling Problems, trying to assign tasks or *Jobs* to the associated resources, namely *Machines* in this context, certainly fit this description. Scheduling is needed when deciding which goods in a fabric shall be produced by which unit in which order, which airplane shall land on which runway at what time, which deeds shall be done when and by which employee in a company or even in every day situations like in which sequence every team member should take care of which parts of the work within a group project. On a personal level, one could think of it as a time-related plan of how to tackle a To-Do-List. While in reality we often use simple heuristics, one might benefit from introducing more sophisticated approaches yielding costs or rewards closer to the optimal ones. Although this might not be strictly necessary in simple cases with little information to consider, it becomes more advantegous the more Jobs have to be handled, the more Machines are available and the less they are related in the times they need to process the Jobs as well as the more additional conditions and constraints having to be considered, like possible deadlines, priorities, dependencies etc. Unfortunately, the more complex the scheduling environment becomes, the more challenging it also gets to find exact or even approximative solutions guaranteed to stay within a certain range of the theoretical optimal solution. In fact, these problems get NP-hard very quickly [1, p. 26] and even approximation algorithms often only have been found for narrowly tailored circumstances and specifially defined objective functions, lacking flexibility for many real world applications for which, varying from situation to situation, some additional cost-contributing factors might have to be considered or can be neglected.

Nowadays, a popular approach to tackle pattern related challenges that might be too complex to be solved intuitively or with traditional mathematical methods is to apply some form of machine learning. In fact, some problems of related nature like Resource Management or Job Shop Problems, where every Job has to be processed by every Machine at some time, have been addressed this way in [2, 3, 4]. However, there is an evident lack of available research on solving Job Scheduling Problems with any form of artifical intelligence in the scientific literature. This thesis therefore represents a novel approach to solving Job Scheduling Problems on Unrelated Machines by using a Neural Network, derived from a version of Deep Reinforcement Learning slightly modified to the specific needs of our problem

environment, which is further specified below. This Neural Network will be used to solve any Job Scheduling Problem meeting said criteria, but could also easily be modified to solve similiar types of problems, turning it a more polyvalent approach. Since knowledge from several different fields will be required for this method, a brief overview of all the necessary theoretical background will be given, focussing on the parts relevant to our specific application to enable the reader to understand the problem formulation as well as the successive construction of its solution.

We will start by giving some mathematical insight to Job Scheduling Problems in section 2 by introducing some basic definitions, problem environments and objective funtioncs along with known combinatorial approaches to them, mostly in form of approximation algorithms. We will analyze their limitations, hence legitimizing the motivation behind this thesis and introducing the special case that we wish to solve, for which no satisfying solution is established in the scientific literature. Especifically, we are trying to find an efficient way of solving the strongly NP-hard problem of non-preemptively scheduling Jobs with deadlines and weights on unrelated parallel Machines, having deadlines and weights as well together with initial occupations, with the objective of minimizing the costs arising by the makespan, i.e. the time at which the process finishes, alongside the total total weighted tardiness of the Jobs and Machines. We will restrain to the case of deterministic processing times but discuss briefly how our approach could be generalized to the stochastic formulation at the end of the thesis.

Due to the infeasibility of constructing sufficiently simple mathematical solutions to the stated problem, we turn to machine learning and the tools it has to offer. We will therefore give a short resumee of Reinforcement Learning in section 3, starting with Q-learning whose convergence property we will mathematically prove and which will serve as ground-laying structure for the computation of the costs. For every feasible action, these Q-values will consist of the immediate costs of taking said action of shutting a currently unoccupied Machine down or assigning a Job to it that has not been processed yet, and the future costs, derived from the assumption that every subsequent decision will be optimal, or respectively what is estimated to be optimal at a given point of the learning process.
Since we do not wish to solve a single Job Scheduling Problem but any one that coheres with the given specifications, for higher numbers of Jobs and Machines the amount of states to consider becomes significantly too vast to compute the true costs to every feasible action in every possibly occuring state for any given set of Jobs and Machines. Consequently, we expend our approach to include Deep Reinforcement Learning, where the agent is given in form of a Neural Network, learning the mapping of any state to the costs

associated with any feasible action, represented by the associated unit in its output layer. This Neural Network is the one that we will finally use to successively create schedules by following its greedily implied policy of always taking the action it estimates to generate the least costs at any point of decision in the scheduling process.

With regards to the architecture of said Neural Network we will face a lot of structural challenges induced by the nature of the data. Vanilla Neural Networks require data of static dimension as input and produce outputs of static dimension as well. In scheduling tasks on the other side, the number of remaining Jobs and usable Machines varies for every problem. Moreover, to guarantee the Markovian property needed in Deep Reinforcement Learning, an architecture is required that only considers the set of remaining Jobs and still working Machines at any given time point in a scheduling process. As a consequence, the number of Jobs and Machines changes even within any scheduling process whenever an action is taken. Accordingly, the number of feasible actions differs for every state as well. We therefore need an architecture capable of processing inputs of varying sizes as well as producing ouputs of dynamic dimensions.
One important field of machine learning that has to deal with data of variable size and that has developed a lot over the last years is the one of Natural Language Processing. Thus, in section 4 we will give an overview of the current architectures of said area that are suited to our needs and discuss their benefits and weaknesses from a mathematical as well as an empirical standpoint to properly incorporate them into our Neural Network.

After providing the reader with all the required theoretical knowledge, we will explain in section 5 how to interprete our Job Scheduling Problems in a way that we can apply our approach, i.e how we interprete any decision-demanding moment of a scheduling process as a state and how we extract the data from its pertinent information to feed it to our Neural Network. Subsequently, in section 6 we present the architecture of our Neural Network in detail, stating the motivation behind every part of its architecture by analyzing which of the difficulties inherent in the nature of our data it is supposed to address.

The creation of the data for the learning process will be divided into two parts in section 7. We will start by training the Neural Network in a supervised maneer. For this, we construct the training, validation and test data ourselves by repeatedly creating Job Scheduling Problems consisting of 8 Jobs and 4 Machines, generated randomly within some value restrictions. Then, for every possible state of every scheduling problem, we exhaustively compute the ground-true costs associated with every action. We then select

some of these states at random, following a distribution ensuring balancing in the data with regards to which of the actions is optimal. From these states we then extract the data to train the Neural Network. We do, however, only consider states with at least 3 Jobs and 2 Machines remaining, since it would elsewise be too cheap to just compute the optimal action. The states used to generate the validation and test set do not get balanced. Naturally, this approach would be applicable for higher Job and Machine numbers as well when greater computational power is at hand.
In the second part we want to increase the number of Jobs. This time, we make use of Deep Reinforcement Learning techniques. We create scheduling problems consisting of 9 Jobs and 2 to 4 Machines. Then, we approximate the costs of assigning a Job in this state by computing the immediate costs and estimating the optimal future costs by feeding the successor state of 8 Jobs to our Neural Network and greedily selecting the smallest produced value. Hence, our pretrained Neural Network serves as Target Network. This way, we can create training data for Job Scheduling Problems with 9 Jobs and 2,3 and 4 Machines, with which we continue to train our Neural Network, acting as Q-Network. We then update the Target Network and repeat this process, always increasing the number of Jobs by one, until we have trained our Neural Network on up to 12 Jobs. Note that due to the drastically reduced computational costs of this approach, we do not need to select the actions ϵ-greedily or follow any other behaviour policy, but can efficiently compute the costs for every action at any given initial state.

We will use the Neural Network to solve Job Scheduling Problems by always taking the action corresponding to the estimated optimal value in any given state of the scheduling process. In section 8 we will compare the costs obtained by using our Neural Network as an implicit policy for several Job Scheduling Problems to the optimal costs as well as the ones produced by an heuristic algorithm which will serve as comparative and competetive metric. Afterwards, we will investigate if applying the updated parameters obtained by the deeply reinforced learning phase improves its performance .

In section 9 we will then see how our Neural Network compares to the scheduling algorithm when applying both to Job Scheduling Problems with a higher number of Jobs and Machines than the Neural Network has been trained on. We will test them on up to 100 Jobs and 15 Machines.

Throughout this thesis, we will use the letters j and m to index the Jobs and Machines respectively and the belonging capital versions J and M to denote their number inside a Job Scheduling Problem. This slightly deviates from the standard notation, where the Machines usually are indexed with i and their number is referenced with m while n refers to the number of

Jobs [1, 5]. Since we however cover a lot of different theoretical fields, we decided for this adjustment to guarantee notational coherence throughout our elaboration and to minimize the total deviation from standard notation.

A code for the entire procedure, from creating the states and extracting the data over constructing the Neural Network up to comparing its results, is given in form of Notebooks, viewable on Github under [6]:

Dieguinho1612/Job-Scheduling-Deep-Reinforcement-Learning

2 Job Scheduling Problems

Adult life can be hard and everybody probably knows the sensation of feeling overwhelmed by the quantity of tasks that have to be done at times. This holds especially true when there are *deadlines* involved that should not be exceeded (as in writing this master thesis). A common type of advise for these situations is to first create a *schedule* in which order one wants to tackle the tasks. So let us say there are $J \in \mathbb{N}$ tasks (or *Jobs*) that have to be done. Some of them are shorter than others, so let p_j be the units of time it takes to complete Job j and $d_j \geq$ the date it is due. We start at time 0. So if we complete the Job at date C_j, its *lateness* is denoted by $L_j := C_j - d_j$. We hope that being a little late on a Job may be pardoned, but being very late on any of them could cause us trouble, so our objective is to minimize the maximum lateness $L_{max} := \max_{j=1,\dots,J} L_j$. We quickly realize that the number of possible schedules is equal to the number of permutations of J, i.e. $J!$. Unfortunately, this problem turns out to be strictly NP-hard [5, p. 36], so computing all of them to then pick the one with the minimum lateness becomes infeasible for greater J. As a consequence, we do not even know if there exists a solution for us to be on time, expressly to guarantee $L_{max} \leq 0$, which really bothers the mathematician in us. But what about an approximation, providing us with a solution not worse than a certain threshold? So let L^*_{max} be the minimum maximum lateness that could possibly be achieved. An algorithm that grants us a solution whose value is assured to not be worse than α times the value of the optimal solution for an $\alpha > 1$ is called an *α-approximation-algorithm*. In our case, that would mean

$$L_{max} \leq \alpha * L^*_{max}$$

As constructed, no such near-optimal solution can exist, since for any $\alpha > 0$ for every set of Jobs that can completely be finished on time, we would obtain a schedule with exact optimal maximum lateness $0 \leq L_{max} \leq \alpha * 0$, contradicting the NP-hardness of the problem. This can be fixed with the technical tweak of forcing L^*_{max} to always be strictly positive by defining the deadlines to be negative. An intuitive heuristic approach would now be to just start with the job with the earliest deadline. This actually turns out to be a 2-approximation-algorithm.

The above is a very simple example for a general type of decision-making-processes called **Job Scheduling Problems**. They are a common challenge in the field of combinatorial optimization and still the object of a lot of research [1, p. 13]. Let us now extend the simple example from above a little further:

Let us say, we do realize that the work is just too much to be handled by us

alone. Lucky for us, a lot of tasks that might have required hard physical labor in the past can these days be automated, given one has the right resources to do so. Therefore, we decide to buy some *Machines* to get the work done for us, let their number be denoted by $M \in \mathbb{N}$. We now have the big advantage of being able to process the Jobs in *parallel* by assigning them to the Machines. Naturally, we want to schedule them in a way that is beneficial to whatever our objective is. For this, we have to consider the *processing time* p_{jm} of how long it would take Machine m to to complete Job j. We distinguish between the following three main cases of:

- **Identical Machines** (PM): Every Job $j = 1, \ldots, J$ has a processing time p_j that is identical on every Machine $m = 1, \ldots, M$:

$$p_{jm} = p_j$$

- **Uniform Machines** (QM): Every Job $j = 1, \ldots, J$ has a processing time p_j and every Machine $m = 1, \ldots, M$ has a *speed* $s_m \in \mathbb{R}_+$. The processing time of Job j on Machine m is the product of:

$$p_{jm} = p_j * s_m$$

- **Unrelated Machines** (RM): The processing times of any two Machines $m_1, m_2 = 1, \ldots, M$ are not related with each other for any Job $j = 1, \ldots, J$ in any way:

$$p_{jm_1} \not\sim p_{jm_2}$$

These options allow us to extend our application of Job Scheduling to several real world problems. Our Jobs and Machines might be products that have to be produced in a fabric by certain production units, patients in a hospital that have to be attended by doctors, airplanes that have to be distributed to runways, orders in a restaurant that have to be prepared by the cooks working there, cars that have to be fixed my mechanics in a workshop and so on. In all of these situations, it is desirable to find a schedule that is optimally efficient with regards to certain conditions. Until now, these have been restricted to the deterministic processing time of a Job on every Machine as well as its deadline, but these can be furhter customized to the specific needs of many problems. The Jobs might be of difference importance and therefore have different *weights* w_j. These weights function as a factor to the costs that have to be paid due to its makespan or an exceedance of its deadline. It is also possible that not all Jobs are available from the start, which would be represented by their *release date*. All the features stated so far can also apply to the Machines, so a Machine might not be available from the start, due to it having an *initial occupation* of runtime r_m analogous to the release date of a job, has a date δ_m by which it should get turned off (which is equivalent

to the decision of not assigning any further Jobs to it) and some penalty cost factor in form of a weight ω_m. Further possible determining factors are whether there are any precedence constraints, demanding for certain Jobs to be finished before processing other ones, whether the scheduling allows for *preemptiveness*, meaning that the processing procedure of a Job can be interrupted before it is completed and then continued later on, if a Machine can be *idle* at any time point, if certain Jobs can only be done by certain Machines, if a Machine can break down at some moment and so on. Moreover, in many applications the processing time cannot be expteced to be exactly known, but often only follows an estimated distribution. Therefore, in *Stochastic Job Scheduling* the processing times of Jobs are not given as a deterministic value, yet instead in form of such a probability distribution.

Alongside all these options for problem formulation, there also arise many specific goals in form of different objective functions to them. The following lists gives an overview of some possible choices:

- **Maximum lateness**: As already mentioned, this is defined as the lateness of the Job whose completion date exceeds its deadline the most.
$$L_{max} := \max(L_1, \ldots, L_J)$$
- **Makespan**: The makespan refers to the time point when the last Job is finished. Minimizing the makespan therefore is equivalent to trying to terminate the workload the earliest possible.
$$C_{max} = \max(C_1, \ldots, C_J)$$
- **Total weighted completion time**: The sum over all the completion times indicates how much times the Machines had to be working in total. Weighting the Jobs adds incentive to be working less time on some compared to others, for instance because the process might be more expensive per time unit.
$$\sum_{j=1}^{J} w_j C_j$$
- **Total weighted tardiness**: Similiar to the total weighted completion time, we wish to minimize the sum over the exceedence of all Jobs with regards to their deadline. The tardiness here is therefore defined as $T_j := max(0, C_j - d_j)$. Note that in contrast to the lateness of a Job, its tardiness only refers to whether or not it gets finished on time, not providing us with any additional reward for doing so even earlier.

Analogously, weighting every Job gives more importance to not exceed certain deadlines over others.

$$\sum_{j=1}^{J} w_j T_j$$

- **Weighted number of tardy Jobs**: Even though being related to the total weighted tardiness, in this case we only care about how many Jobs have exceeded their deadline, yet not about by how much. Weighting them again gives stronger incentive to being on time for certain Jobs than for others. For this purpose, we define the binary variable U_j, indicating whether a Job has been tardy or not.

$$\sum_{j=1}^{J} w_j U_j$$

Of course, combinations of these objective functions are valid, too. A gantt chart of an optimal scheduling example to minimize the makespan on Identical Machines with initial occupations is given by the following figure 1:

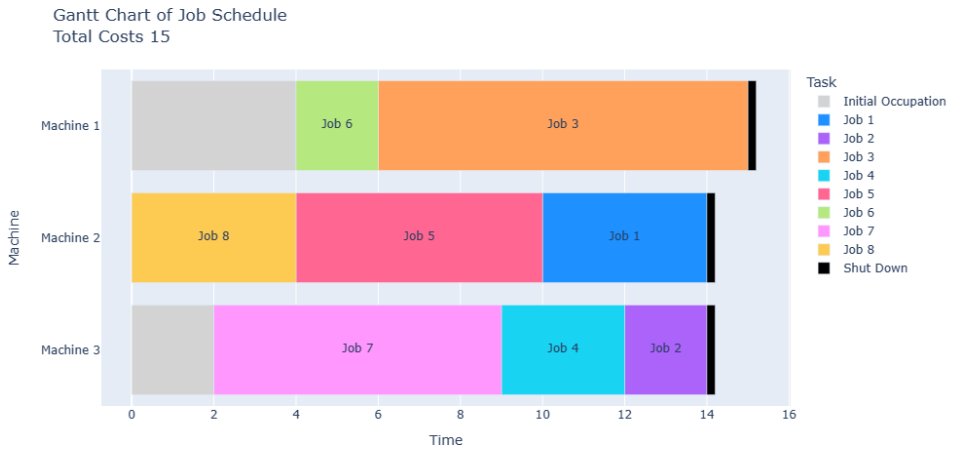


Figure 1: Schedule minimizing Makespan on Identical Machines

2.1 Problem formulation

In this master thesis, we will treat the problem of scheduling on Unrelated Machines with initial occupations, not allowing for preemptions and aiming for the goal of minimizing the sum of the makespan together with the total

weighted tardiness of the Jobs as well as of the Machines, the latter denoted by τ_m:

$$C_{max} + \sum_{j=1}^{J} w_j T_j + \sum_{m=1}^{M} \omega_m \tau_m \tag{1}$$

Since all Jobs are available from the start, deciding to leave an available Machine unoccupied at some moment and appointing a Job to it later on would always be suboptimal compared to doing so immediately. Therefore, we do not allow for the Machines to be idle either, so not assigning any Job to a Machine as soon as it finishes its last occupation is equivalent to turning it off.

Thus, for $j = 1, \ldots, J$ and $m = 1, \ldots, M$ such a Job Scheduling Problem can mathematically be described by an input consisting of the values:

- $p_{ij} > 0$
- $d_j \geq 0$
- $\delta_m \geq 0$
- $w_j > 0$
- $\omega_m > 0$
- r_m

Becasue we do not allow for preemptive solutions, every Job has to get assigned exactly once. Consequently, a *schedule* is a sequence of 2-tuples

$$(st_1, m_1), \ldots, (st_J, m_J)$$

which are composed of the starting times $st_j > 0$ for every Job $j = 1, \ldots, J$ and the Machine $m_j = 1, \ldots, M$ to which it gets assigned, hence also inducing its completion time:

$$C_j = st_j + p_{jm_j}$$

Let

$$A_m := \{j \mid m_j = j\}$$

denote the set of all Jobs j that get processed by Machine m. Then, the time ζ_m at which a Machine m gets shut down can be derived as:

$$\zeta_m = \max_{j \in A_m} C_j$$

Such a schedule is *feasible* if:

1. As soon as it finishes its initial occupation, every Machine either gets its first Job assigned or is turned off immediately:

$$\begin{gathered} \forall m = 1, \ldots, M : \\ \min_{j \in A_m} st_j = r_m \vee \ \zeta_m = r_m \end{gathered} \tag{2}$$

2. No Machine has more than one Job assigned at a time:

$$\begin{gathered} \forall m = 1, \ldots, M : \\ \forall j_1 \neq j_2 \in A_m : \\ C_{j_2} \leq st_{j_1} \vee C_{j_1} \leq st_{j_2} \end{gathered} \tag{3}$$

3. No Machine is ever idle between two Jobs:

$$\begin{gathered} \forall m = 1, \ldots, M : \\ \forall j_1 \in A_m : \\ (\exists j_2 \in A_m : C_{j_1} = st_{j_2} \vee \ C_{j_1} = \zeta_m) \end{gathered} \tag{4}$$

Requirement (2) ensures that no Machine receives a Job assignment before finishing its initial occupation. Combined with (4) it prevents any idleness. The remaining conditions are guaranteed by construction. Hence, we treat the Job Scheduling Problem:

$$RM \mid d_j, \delta_m, w_j, \omega_m, r_m \mid C_{max} + \sum_{j=1}^{J} w_j T_j + \sum_{m=1}^{M} \omega_m \tau_m \tag{5}$$

2.2 Related Work

Being a combination of both, before tackling the task of minimizing our objective function stated in (1), we first want to take a look at how well pre-existing combinatorial algorithms approximate the two separate objectives of minimizing the makespan and the total weighted tardiness. We start with the former.

Minimizing the Makespan C_{max} is already NP-hard when scheduling on two Identical Machines, what can be derived from the partition problem [1, p. 112]. However, reasonable approximations can be achieved even through simple greedy or local search algorithms, for which a set of rules determine the movement from one local feasible solution to the next one [5, p. 40]. For scheduling on Unrelated Machines, there have been proposed a

2-approximation algorithm by Versache and Wiese [7] as well as by Lenstra, Shmoys and Tardos [8]. The latter team has also shown that no approximation algorithm running within polynomial time with a factor α smaller than 1.5 can exist unless $P = NP$.

For minimizing the total weighted tardiness of Jobs on Unrelated Machines, also strongly NP-hard, not many known solutions exist. This problem is extremly hard to solve even for the case of Uniform Machine scheduling and in an environment of only about 5 Machines and 30 Jobs [1, p. 143]. One probabilistic solution was proposed in the paper [9] where they used an Ant Colony Optimization Algorithm derived from an heuristic rule from single Machine scheduling, outperforming the same in computational experiments on Unrelated Machines.

2.3 Our Approach

Due to the complexity and difficulty of our scheduling problem as well as the the lack of options within the theoretical literature to solve even only parts of it, we decide to take a novel approach by applying Deep Learning with immanent structures of Deep Reinforcement Learning with the objective to construct a Neural Network that provides us with satisfying solutions. The exact method will be unfolded over the next sections. Deep Learning has already been used to solve Job-Shop Scheduling Problems as in [4]. However, no such approach successfully tackling a Job Scheduling Problem similiar to ours is known to this point.

To measure our results, we will use a combinatorial algorithm as a comparative metric to the scheduling costs we obtain by following the estimations of our Neural Network. The greedy algorithm of list scheduling, where the Jobs get sorted according to a feature and then iteratively assigned to the next Machine that becomes available, has shown to provide a reasonable approximation for several objective functions when scheduling on Single or Identical Machines [5]. Since this simple implementation might neglect too much information in our more complex case, we extend this algorithm to the following version:
Let $\Pi_m : \{1, \dots, J\} \mapsto \{1, \dots, J\}$ be a permutation, ordering the list of Jobs by some feature depending on the machine m. So whenever a Machine m becomes free during the scheduling process, check whether for the first remaining Job of that sorted list there exists another machine m^* on which it would finish sooner, i.e. if the processing time of said Job added to the time until Machine m^* finishes its current occupation is shorter than its processing time on the free Machine m. If no such Machine m^* exists, assign the

Job to Machine m. Otherwise, repeat the process for the next remaining Job from the sorted list until a Job gets assigned to Machine m. If the iteration completes the entire list of remaining Jobs without assigning any one to the currently free Machine m, turn it off. Algorithm 1 describes this decision process:

Algorithm 1: Comparative Scheduling Algorithm

Input: list of remaining Jobs LJ
list of working Machines LM
free Machine m
machine occupation runtimes $r \in \mathbb{R}^M$

```
1  LJ = Π_m(LJ) ;                                   // sort Jobs
2  for Job j in LJ do
3      assign = True;
4      for Machine m* in LM do
5          if r_{m*} + p_{jm*} < p_{jm};        // compare completion times
6          then
7              assign = False ;                  // do not assign Job j
8              break;
9      if assign = True ;                        // no faster Machine
10     then
11         assign Job j to Machine m;
12         remove Job j from LJ;
13         break
14 if assignment = False ;                       // no Job got assigned
15 then
16     turn off Machine m;
17     remove m from LM;
```

To enable a dynamic sorting of the list of remaining Jobs, taking into account the development of the schedule over time, they will get reordered whenever a Machine m becomes free. As for the features, Π_m will sort the Jobs in ascending order with regards to their processing time on Machine m. If two processing times match, the Job with the greater weight will be put first. If these are equal, too, the Job with the lower deadline will be ordered before the other one. The pseudo code for ordering any two remaining Jobs j_1 and j_2 with respect to Π_m is given by the following algorithm 2:

Algorithm 2: Π_m: Sorting Jobs for Machine m

Input : Jobs j_1, j_2
Output: Jobs sorted by Π_m

```
1  if p_{j1m} < p_{j2m} ;                          // shorter processing time
2   then
3   |  return (j1, j2);
4  else if p_{j1m} > p_{j2m} then
5   |  return (j2, j1);
6  else                                          // processing times are equal
7   |  if w_{j1} > w_{j2} ;                                  // higher weight
8   |   then
9   |   |  return (j1, j2);
10  |  else if w_{j1} < w_{j2} then
11  |   |  return (j2, j1);
12  |  else                                       // weights equal as well
13  |   |  if d_{j1} ≤ d_{j2} ;                          // earlier deadline
14  |   |   then
15  |   |   |  return (j1, j2);
16  |   |  else
17  |   |   |  return (j2, j1);
```

3 Deep Reinforcement Learning

When it comes to Machine Learning, there exist three basic paradigms. On the one hand, there is Supervised Learning, which uses labeled data sets to train an algorithm according to a task, usually to classify an instance or to predict an outcome. In the context of Neural Networks, this means that there exist target values which the Network tries to achieve and that can be used to measure its success in doing so. On the other hand, there is Unsupervised Learning, where no such targets are known and the data is unlabeled. Pattern Classification is an example, where the Network shall divide the data into clusters of data points with similiar characteristics. **Reinforcement Learning** can be seen as somewhere inbetween these two, since target values are not given, but shall be deducted by an agent throughout his learning phase. In Job Scheduling, theoretically the costs of all possible assignment of Jobs to Machines could be calculated, in particular the optimal one. Due to many of these problems being NP-hard, doing so is usually computationally infeasible. However, computing the costs for certain chosen assignments might be possible, hence turning the problem into an observable environment that might be suited for Reinforcement Learning, especially for the subgenre of **Deep Reinforcement Learning** (**DRL**), where the agent gets replaced by a Neural Network.

Therefore, in this section we will give an overview of Reinforcement Learning by introducing *markovian decision problems* together with the method of *Q-learning* and its associated *temporal difference learning algorithm* in stochastic iterative processes. We will mathematically proof that the Q-learning relaxation does indeed provide convergence to the minimal costs, since especially the contraction property has been explained only very briefly in the original paper of [10]. Subsequently, we will present the extension to Deep Reinforcement Learning and how to apply it.

3.1 Markovian decision problems

This subsection is based on [11]. Basic Reinforcement Learning is modeled as a markovian decision problem, in which an agent tries to pay the minimum cost (often also referred as the highest reward). He moves through a set of *states*, choosing from a set of *actions* which will take him to a succesor state. The possible actions are determined by the state he is in. To which state the transition will occur is not necessarily deterministic, yet the probabilites only depend on the current state and the choosen action. The arising *immediate costs* are defined by the current and the succesor state as well as by the chosen action. The agent starts in a beginning state, there

can be diverse final states. After reaching one of them, there are no possible actions anymore. The goal is it to find a *policy*, mapping an action to every possible state such that the expected value of the *total costs*, being the sum over all the immediate costs along the way, is minimal. We will use the following notation:

- states $S = \{s_1, \ldots, s_n\}$
- actions $A = \{a_1, \ldots, a_l\}$
- transition probabilty $\rho_{ss'}(a)$:
  The probabilty of reaching state s' by chossing action a in state s.
  It is derived from the conditional probability mass function $p(\cdot \mid s, a)$ of the discrete probability distribution $P : S \times A \times S \mapsto [0, 1]$:
  $$\rho_{ss'}(a) := p(s' \mid s, a)$$
- immediate costs $c_s(a)$:
  The mean of the *reward function* $R : S \times A \times S \mapsto \mathbb{R}$ over all the possible succesor states:
  $$c_s(a) := \sum_{s' \in S} \rho_{ss'}(a) R(s, a, s')$$
- policy $\mu : S \to A$
- Value function $V_\mu(s) = \lim_{N \to \infty} E\{\sum_{t=0}^{N-1} \gamma^t c_{s_t}(\mu(s_t)) \mid s_0 = s\}$

The value function indicates the expected total cost of playing policy μ when starting in state s with the *discount rate* $\gamma \in [0, 1]$. If $\gamma = 0$, we only worry about the next immediate cost, if $\gamma = 1$, we have no time pressure of reaching a certain final state.
Clearly, our goal is to find an optimal policy μ^*, such that:

$$V^*(s) := V_{\mu^*}(s) = \min_\mu V_\mu(s)$$

For this, we use the recursevly defined Bellman Equation

$$V^*(s) = \min_{a \in A(s)} \{c_s(a) + \gamma \sum_{s' \in S} \rho_{ss'}(a) V^*(s')\} \tag{6}$$

where the idea is to pretend for every state $s \in S$, that we are starting in this state, taking the optimal action there and then continue playing an optimal

policy from the successor state on reached by this action.

In order to find such a V^*, we use an interative process:

$$V^{(k+1)}(s) = \min_{a \in A(s)} \{c_s(a) + \gamma \sum_{s' \in S} \rho_{ss'}(a) V^{(k)}(s')\}$$

The iteration from $V^{(k+1)}(s)$ to $V^*(s)$ can be shown to be a contraction mapping, the proof is analogous to the one in the following subsection. Therefore:

$$\max_s \mid V^{(k+1)}(s) - V^*(s) \mid \leq \gamma \max_s \mid V^{(k)}(s) - V^*(s) \mid$$

which implies that $V^{(k)}$ converges to V^* for an arbitrary inicial $V^{(0)}$.

3.2 Q-learning and its convergence

Until now we assumed that the transition probabilites and expected costs were known. Since this is not always the case, we opt to slightly change the Bellman Eqaution to

$$V^*(s) = \min_{a \in A(s)} Q^*(s, a)$$

by introducing the Q-values:

$$Q^*(s, a) := \bar{c}_s(a) + \gamma \sum_{s' \in S} \rho_{ss'}(a) V^*(s') \tag{7}$$

This has the advantage that we can iteravely update the value of each action instead of just the action with the minimal expected value. In this case the Q-learning algorithm is a stochastic form of value iteration.

To perform a value iteration step, for any V the quantity $\sum_{s' \in S} \rho_{ss'}(a) V(s')$ and the expected cost $c_s(a)$ have to be known. Said quantities can be *estimated* by the quantities $V(s')$, if successor state s' is chosen with probability $\rho_{ss'}(a)$, which is assured by simply following the transitions of the actual markovian environment, giving us an unbiased estimate of the sum. Redefining $c_s(a)$ as an unbiased estimate of the actual expected cost leads to the following relaxation of the Q-learning algorithm, where $Q_t(s, a)$ and $V_t(s)$ are the learners estimates of the Q function and V function at time t:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t)[c_{s_t}(a_t) + \gamma V_t(s_{t+1}) - Q_t(s_t, a_t)] \tag{8}$$

The difference between Q_t and Q_{t+1} is called *temporal difference*. The objective now therefore is to denote conditions under which an arbitrary Q_0

converges to the optimal action-value-mapping Q^* when applying this *temporal difference learning algorithm* and to give an elaborated proof of this.

To do so, we use the following theorem stated in [10]:

Theorem 1 *A random iterative process* $\Delta_{n+1}(x) = (1 - \alpha_n(x))\Delta_n(x) + \beta_n(x)F_n(x)$ *converges to zero w.p.1 under the following assumptions:*

1. *The state space is finite*
2. $\sum_n \alpha_n(x) = \infty$, $\sum_n \alpha_n^2(x) < \infty$, $\sum_n \beta_n(x) = \infty$, $\sum_n \beta_n^2(x) < \infty$, *and* $E\{\beta_n(x) \mid P_n\} \leq E\{\alpha_n(x) \mid P_n\}$ *uniformly w.p.1*
3. $|| E\{F_n(x) \mid P_n\} ||_W < \gamma || \Delta_n ||_W$, *where* $\gamma \in (0, 1)$
4. $Var\{F_n(x) \mid P_n\} \leq C(1 + || \Delta_n ||_W)^2$

We want to apply this theorem to our Q-learning algorithm. For this, we reorganize the iteration term to

$$Q_{t+1}(s_t, a_t) = (1 - \alpha_t(s_t, a_t))Q_t(s_t, a_t) + \alpha_t(s_t, a_t)[c_{s_t}(a_t) + \gamma V_t(s_{t+1})]$$

Furthermore, we mark that the optimal Q^* function is a fixpoint of the following contraction operator $\mathbf{H}$, defined for a generic function $q : S \times A \mapsto \mathbb{R}$ [12] as

$$\mathbf{H}q(s, a) := \sum_{s' \in S} \rho_{ss'}(a)[c_s(a) + \gamma \min_{a' \in A} q(s', a')]$$

since

$$\begin{aligned} \mathbf{H}Q^*(s, a) &= \sum_{s' \in S} \rho_{ss'}(a)[c_s(a) + \gamma \min_{a' \in A} Q^*(s', a')] \\ &= \sum_{s' \in S} \rho_{ss'}(a)[c_s(a) + \gamma V^*(s')] = Q^*(s, a) \end{aligned}$$

$\mathbf{H}$ is indeed a contraction operator in the sup-norm:

$$
\begin{aligned}
&|| \mathbf{H}q_1 - \mathbf{H}q_2 ||_\infty \\
&= \max_{(s,a)} | \sum_{s' \in S} \rho_{ss'}(a)[c_s(a) + \gamma \min_{a' \in A} q_1(s', a) - c_s(a) - \gamma \min_{a' \in A} q_2(s', a')] | \\
&= \max_{(s,a)} \gamma | \sum_{s' \in S} \rho_{ss'}(a)[\min_{a' \in A} q_1(s', a') - \min_{a' \in A} q_2(s', a')] | \\
&\leq \max_{(s,a)} \gamma \sum_{s' \in S} \rho_{ss'}(a) | \min_{a' \in A} q_1(s', a') - \min_{a' \in A} q_2(s', a') | \\
&= \max_{(s,a)} \gamma \sum_{s' \in S} \rho_{ss'}(a) | - \max_{a' \in A} -q_1(s', a') + \max_{a' \in A} -q_2(s', a') | \\
&\leq \max_{(s,a)} \gamma \sum_{s' \in S} \rho_{ss'}(a) \max_{(i,a')} | q_1(i, a') - q_2(i, a') | \\
&= \max_{(s,a)} \gamma \sum_{s' \in S} \rho_{ss'}(a) || q_1 - q_2 ||_\infty \\
&= \gamma || q_1 - q_2 ||_\infty
\end{aligned}
$$

Finally, the following theorem of [10] determines conditions under which convergence is garantueed for our Q-learning relaxation algorithm:

Theorem 2 *The Q-learning algorithm given by*

$$Q_{t+1}(s_t, a_t) = (1 - \alpha_t(s_t, a_t))Q_t(s_t, a_t) + \alpha_t(s_t, a_t)[c_{s_t}(a_t) + \gamma V_t(s_{t+1})]$$

converges to the optimal $Q^*(s, a)$ values if:

1. *The state and action space is finite*
2. $\sum_n \alpha_n(x) = \infty$, $\sum_n \alpha_n^2(x) < \infty$ *uniformly w.p.1*
3. $Var\{c_s(a)\}$ *is bounded*
4. *If $\gamma = 1$ all policies lead to a cost free terminal state w.p.1*

Proof.

To proof this theorem, we have to bring our algorithm into the form of Theorem 1 to be able to aply it. For this, we do the following:

- substract $Q^*(s, a)$ from both sides

- define $\Delta_t(s,a) := Q_t(s,a) - Q^*(s,a)$
- define $F_t(s,a) := c_s(a) + \gamma \min_{a' \in A} Q_t(s_{next}, a') - Q^*(s,a)$
- define $\beta_t(s,a) := \alpha_t(s,a)$

So finally, we get:

$$\Delta_{t+1}(s,a) = (1 - \alpha_t(s,a))\Delta_t(s,a) + \beta_t(s,a)F_t(s,a)$$

With this, the first and second condition of Theorem 1 are fullfilled given the assumptions 1. and 2. of Theorem 2. With our knowledge about **H** we can show now that the third and fourth condition of Theorem 1 are fullfilled as well [10, 12].

$$E\{F_t(s,a) \mid P_t\}$$

$$= \sum_{s' \in S} \rho_{ss'}(a)[c_s(a) + \gamma \min_{a' \in A(s')} Q_t(s', a')] - Q^*(s,a)$$

$$= (\mathbf{H}Q_t)(s,a) - Q^*(s,a)$$

$$= (\mathbf{H}Q_t)(s,a) - (\mathbf{H}Q^*)(s,a)$$

Since **H** is a contraction mapping, we get:

$$|| \; E\{F_t(s,a) \mid P_t\} \; ||_\infty = || \; (\mathbf{H}Q_t) - (\mathbf{H}Q^*) \; ||_\infty$$

$$\leq \gamma \; || \; Q_t - Q^* \; ||_\infty = \gamma \; || \; \Delta_t \; ||_\infty$$

In the undiscounted case, where $\gamma = 1$, if the chain is absorbing and all policies lead to the terminal state w.p.1., there still exists a weighted maximum norm that gives us the desired contraction mapping property.

For the variance of F_t we conclude:

$$
\begin{aligned}
&Var\{F_t(s,a) \mid P_t\} \\
&= E[F_t(s,a) - E\{F_t(s,a) \mid P_t\}]^2 \\
&= E[c_s(a) + \gamma \min_{a' \in A} Q_t(s_{next}, a') - Q^*(s,a) - (\mathbf{H}Q_t)(s,a) + Q^*(s,a)]^2 \\
&= E[c_s(a) + \gamma \min_{a' \in A} Q_t(s_{next}, a') - (\mathbf{H}Q_t)(s,a)]^2 \\
&= Var\{c_s(a) + \gamma \min_{a' \in A} Q_t(s_{next}, a')\}
\end{aligned}
$$

We know by condition that $Var\{c_s(a)\}$ is bounded and from the properties of the variance itself we also know that:

$$
\begin{aligned}
&Var\{\gamma \min_{a' \in A} Q_t(s_{next}, a')\} \\
&\leq E[\gamma \min_{a' \in A} Q_t(s_{next}, a') - Q^*(s,a)]^2 \\
&\leq E[\gamma \mid\mid Q_t - Q^* \mid\mid_\infty]^2
\end{aligned}
$$

Considering this, we finally get that there exists a $C \in \mathbb{R}_{>0}$ such that:

$$
Var\{c_s(a) + \gamma \min_{a' \in A} Q_t(s_{next}, a')\} \leq C(1 + \mid\mid \Delta_t \mid\mid_\infty)^2
$$

□

3.3 Deep Q-learning

Since we have proven the convergence property of the temporal difference learning algorithm from (8), we know that we have found the true action-value for the state s_t and the action a_t given by $Q^*(s_t, a_t)$ as soon as $Q_t(s_t, a_t)$ equals its *temporal difference target* $Q_{t+1}(s_t, a_t)$. To compute both these values, we need a policy π that decides which action a_t to take for state s_t at step t of the iteration, as well as a policy μ_t that is applied from state s_{t+1} on to compute $V_t(s_{t+1})$. So μ_t is called the *target policy*, a greedy-policy derived from the currently learned estimator Q_t of Q^* and therefore a learned

estimator itself of the optimal policy μ^*, while π is called the *behaviour policy* and is used only during the temporal difference learning algorithm to find this true action-value mapping Q^* by determining the probability distribution of the actions $a_t \sim \pi(s_t)$ for s_t [13]. Algorithms that use the target policy also as behaviour policy are denoted as *on-policy-algorithms*, otherwise refered to as *off-policy-algorithms*. To learn these state-action-values $Q^*(s, a)$ by applying (8), every state-action-combination has to be experienced several times. Since the number of possible combinations increases exponentially, this can quickly become computationally infeasible, which is especially the case for NP-hard problems like the problem of Job Scheduling with a higher number of Jobs and Machines, that to solve is our ultimate goal.

This leads us to the reformulation of the task of constructing Q^* with the approximative approach of finding the parameters Θ so that the action-value mapping $Q(\cdot, \cdot, \Theta)$ minimizes the temporal differences over all states $s \in S$ with regards to the state-dependent probability distribution $\pi(s)$ over the action space A. For a given state s_t, this minimization can therefore be expressed with the mean-squared-error function

$$L_\vartheta(\Theta) := E_{a_t \sim \pi(s_t)}[(y_\vartheta^{(s_t)}(a_t) - Q(s_t, a_t, \Theta))^2] \tag{9}$$

with the target values

$$y_\vartheta^{(s_t)}(a_t) := E_{a_{t+1} \sim \mu(s_{t+1})}[c_{s_t}(a_t) + \gamma V_t(s_{t+1}, a_{t+1}, \vartheta) \mid s_t, a_t] \tag{10}$$

whose minimization can be achieved through the common gradient-descent-method. In **Deep Q-learning**, these temporal-difference targets y_ϑ are estimated by a *Target Network*, while the Q-values from (7) are predicted by a *Q-Network*, both identical in their architecture but not in their weights ϑ and Θ [14]. The parameters ϑ of the Target Network correspond to a former version Θ_{i-1} of the parameters of the Q-Network, where i stands for the next update step. The Target Network itself is therefore identical to a previous version of the Q-Network and gets updated to the current one after every k iterations. Figure 2 summarizes this entire approach:

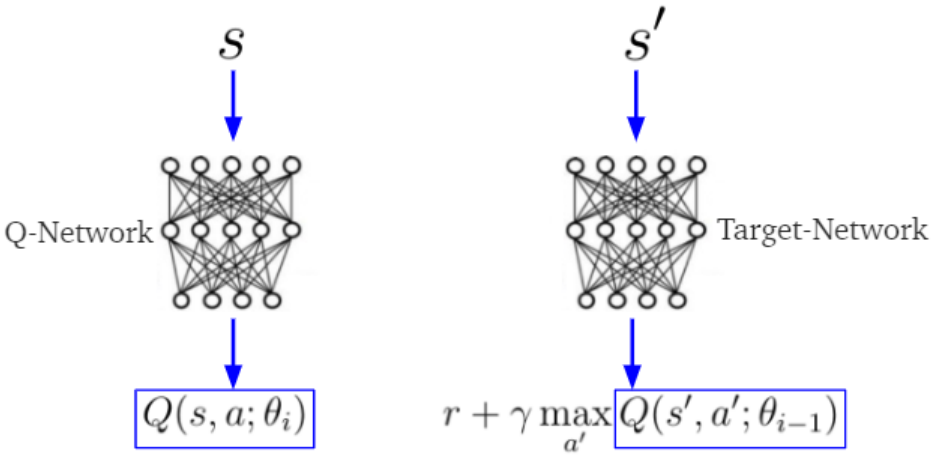


Figure 2: Target Network in Deep Q-learning [15]

Using such a time-delayed version of the Q-Network to compute the temporal difference targets $y_{\Theta_{i-1}}$ has shown to lead to an improved stability of the entire model [16].

4 Neural Network Architectures of Natural Language Processing

As constructed in the previous section, in Deep Reinforcement Learning a Neural Network takes the role of the agent in order to learn a mapping from states to their action-values. We therefore are in need of a *Network Architecture* that fits the structural challenges induced by embedding Job Scheduling Problems into such environments. Since we have to deal with different numbers of Jobs and Machines for different problems and even within the progressing of a schedule, we require our Network to be able to deal with input sequences of varying length as well as dynamically sized outputs. One field of artificial intelligence whose data often comes with such properties is **Natural Language Processing** (**NLP**). It involves enabling the computer to deal with human languages like English, German or Spanish in a way that certain tasks can be performed and dealt with. These tasks come from a broad spectrum such as machine translation, speech-to-text and text-to-speech convertion, text classification, text summarization, sentiment analysis, text generation, image captioning up until fake news detection [17]. Since Human language on itself is complex, finding a numeric representation of it that can be computationally implemented while loosing as little of relevant information as possible is very challenging. Thus, using *Deep Learning* became a popular approach, where a Neural Networks gets trained upon extracting linguistic patterns by using optimizing algorithms such as gradient descent to minimize a defined loss function, which mathematically enforces the desired requirements [18]. Therefore, in this section we will introduce some Network Architectures of this field that deal with sequential data of varying dimensionality like senteces or text blocks and analyze their strenghts and weaknesses to potentially implement them into our Neural Network later on. We start by explaining how idiomatic corpora are converted into data of according form.

4.1 Word Embedding

This subsection is based on the original paper [19] as well as on [20], an in depth mathematical formulation of the belonging structures.
For a Neural Network to be able to interprete given inputs as well as the targets to which the created outputs shall be compared to, these have to be passed in a numerical form. In NLP, a common method is to represent every word or expression with one (or many) vector(s). These vectors should contain the semantic, syntactic and often also contextual information of that word or expression within their geometric embedding into the vector space. This might manifest in words belonging to a group, for example colors, lying closely together in a subspace, or algebraic operations such as adding and

substracting being applicable, for instance something like:

'Berlin' - 'Germany' + 'France' = 'Paris'

where the operation is called on the vector linked to the word. Hence, the resulting vector of 'Berlin' - 'Germany' should resemble the vector of 'capital city'. Another example would be:

'composer' - 'Mozart' + 'Messi' = 'footballer'

where the difference of the first two words stands for 'profession'.
One possible requirement could also be to understand two words being opposite to each other, among other relations:

'high' - 'low' + 'old' = 'young'

For syntactics, recognizing comparatives and superlatives might be desired:

'taller'- 'tall' + 'nice' = 'nicer'

Of course, these relations can change depending on the context of the data.

This approach is called **Word Embedding**, where Deep Learning is applied to learn these vector representations as well. One well known technique to do so is *Word2Vec*, where a Neural Network receives large amounts of text data for training and iterates over every word in the corpus, trying to either predict the central word with the surrounding ones (*Continuoues Bag of Words*) or to use the current word to name the contextual words around it (*Skip-Gram*). Either way results in two vector representations for each word, a context-word and a center-word representation. Both should meet the desired requirements listed above.

As shown in figure 3, these embeddings are then used to feed the inputs and targets to the Neural Network that was designed for a certain task, as well as to interpret its outputs if any of them are ment to resemble text data.

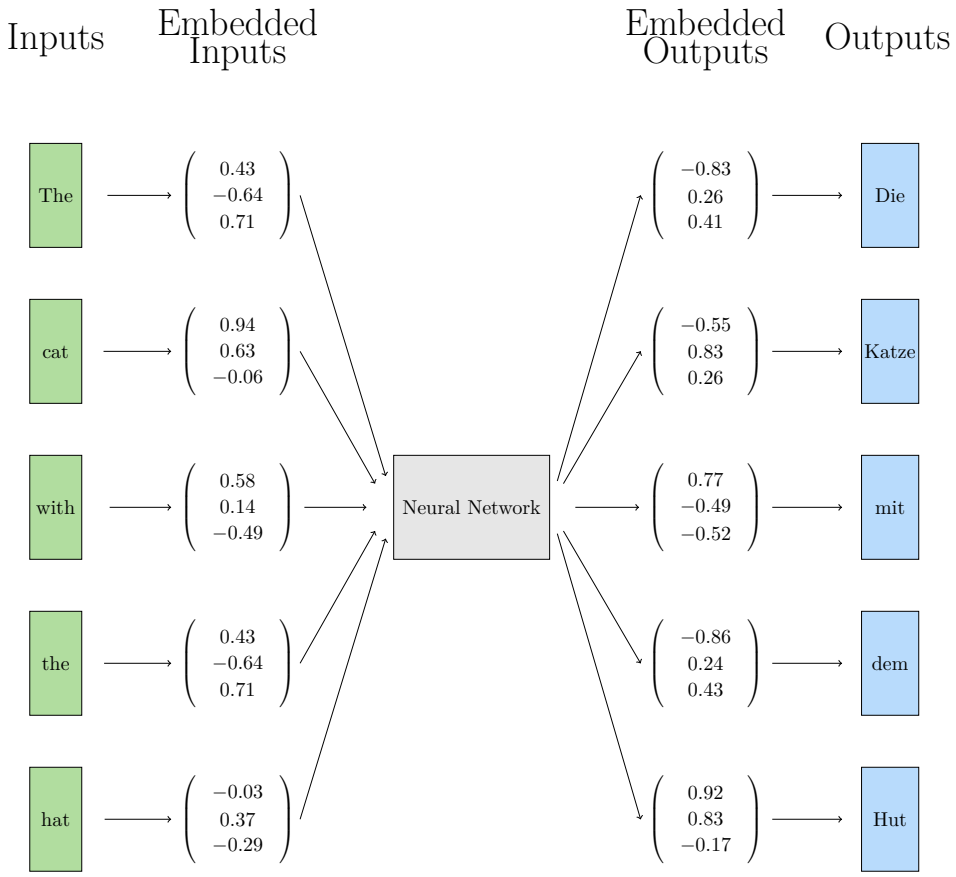


Figure 3: Neural Network with Word Embedding

4.2 Recurrent Neural Networks

The purpose of a Neural Network is to resemble a ground-truth mapping of inputs to outputs given a certain task. In the most basic scenario, every input and output consist of exactly one element of fixed size. This type of mapping is called one-to-one, shown in figure 4:

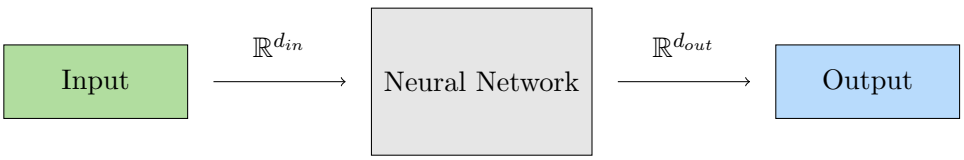


Figure 4: One-to-One Mapping

In NLP however, the size of each sample often varies within one corpus of data, since these can contain structures like sentences or paragraphs which

may differ in the number of words they consist of. Thus, applying Word Embedding transforms them into vector-sequences of varying length as seen in figure 3. Since Neural Networks with a basic architecture like the Multilayer Perceptron have a fixed number of neurons in every layer, we seem to be in the need of a more sophisticated approach.

Let in the following $x^{(1)}, \ldots, x^{(n)} \in \mathbb{R}^d$ denote a sequence of vectors, where n is dynamic. A simple way of handling such an input with a static layer would be to define a maximum permitted sequence length of N_{inp}/d for any instance by setting the number of neurons in the input layer of our Neural Network to $N_{inp} \in \mathbb{N}$ and then applying *zero-padding* [21]. As illustated in figure 5 (with $N_{inp} = 12$, $d = 3$, $n = 3$), for every sample we would concatenate all n vectors and subsequently add zeros-values until we obtain a single vector of dimension N_{inp}.

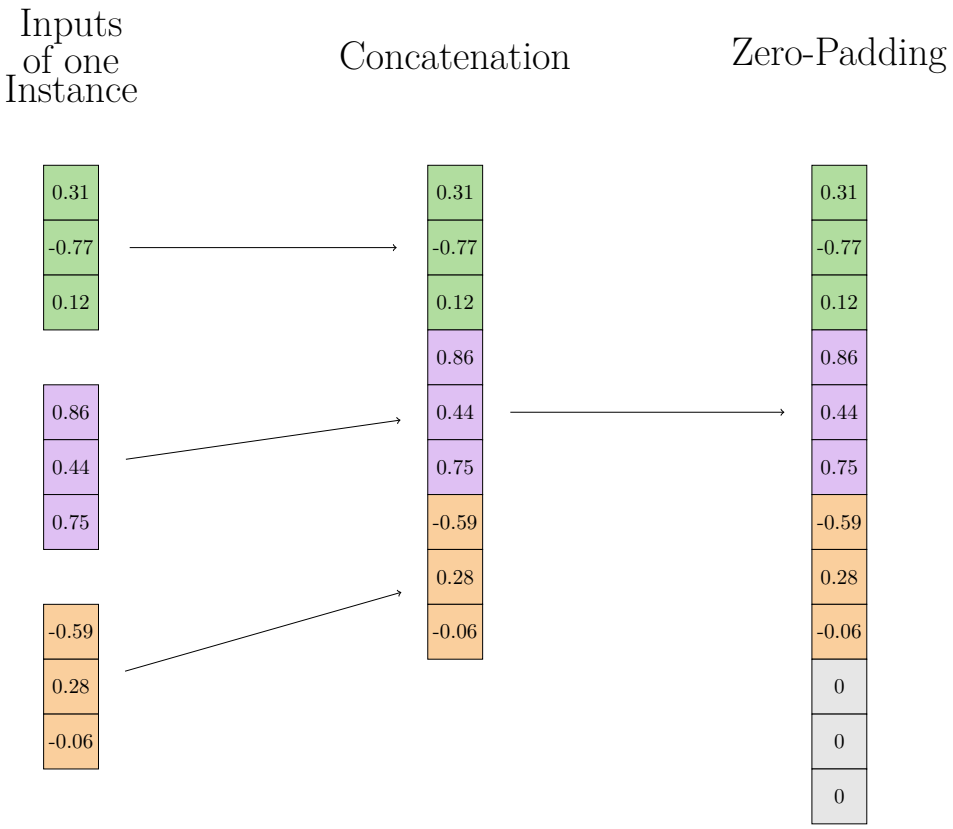


Figure 5: Zero-Padding

Analogously, if the output shall be of seqeuntial form as well, we could create an output layer of fixed size $N_{out} \in \mathbb{N}$ and then split the output into

a sequence of N_{out}/d vectors, hence likewise defining a maximum sequence length. As soon as one of these vectors resembles an end-token, the rest of these vectors get discarded like in figure 6.

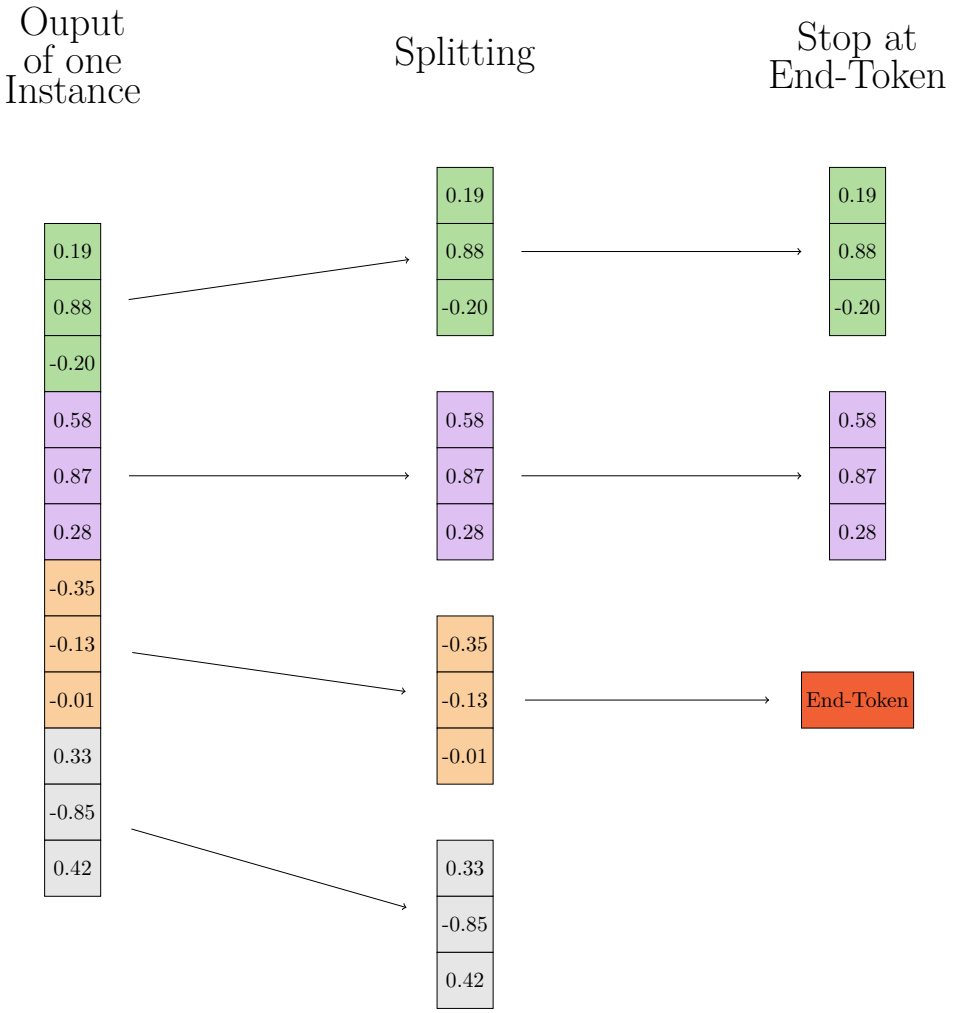


Figure 6: Output Sequence

One obvious downside would be that we could not process any observations with a sequence longer than the defined maximum.
If, on the other hand, we decide to increase N_{in} or N_{out} respectively when constructing our Network, we would also increase the computational cost for every instance. We therefore would always have to consider this trade-off.

Another possibility would be to feed each of the n input vectors of an instance separately through the same Network one by one as shown in figure 7. In language translation this would be equivalent to the idea of translating

a sentence word by word.

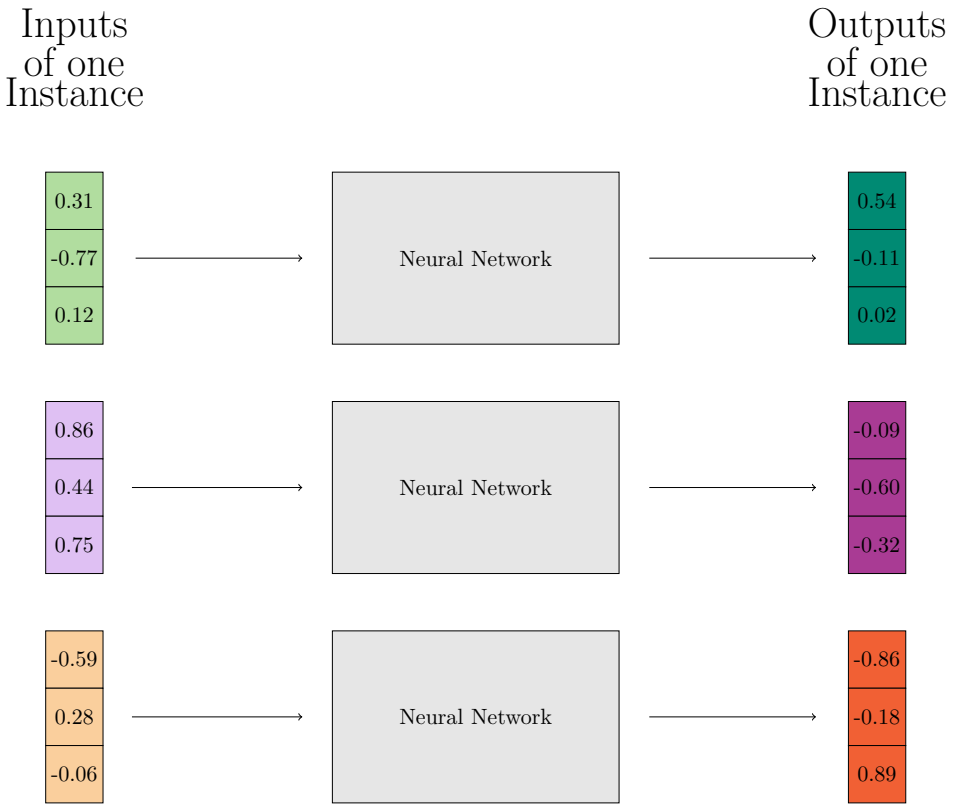


Figure 7: Feeding Network one-by-one

Unfortunately, this approach would be limited in the variability of its structure, since the output sequence would have to be of the same length as the input, as well as especially in the amount of contextual information it could grasp, since the relation between the vectors of one sequence would be strictly neglected [22]. In NLP however, context is of huge importance, so the meaning of a word heavily depends on the ones surrounding it.
Furthermore, the order in which words are presented in a sentence has a decisive impact, too. 'I prefer cats over dogs' is significantly different from 'I prefer dogs over cats'.
This does not hold true exclusively for translations task at all. In sentiment analysis for example, the statement 'This is not bad at all, in fact I do like it!' should be labeled as positive, in contrast to its word permutation 'This is bad, in fact I do not like it at all!'. The above mentioned technique, however, would not be able to distinguish between these two cases.

We therefore wish for an architecture that is able to deal with sequences of varying length while at the same time passing information about the context and the relation of the vectors to each other in every sequence.

These requirements can be met by interpreting instances of language data, such as sentences, as dependent time series consisting of a dynamic number $n \in \mathbb{N}$ of time steps which then get fed through the Neural Network in a recurrent way [23].
This means that, starting at the beginning of the series, every time step iteravely passes the entire Neural Network together with the information obtained at previous time steps.

Mathematically speaken, if an input consists of n vectors $x^{(1)}, \ldots, x^{(n)}$, we want at any given time step $t = 1, \ldots n$ our Network f to be of the form:

$$y_t = f(x^{(t)}, h^{(t-1)}) \tag{11}$$

where $y^{(t)}$ is the output at the t-th time step and $h^{(t-1)}$ some information computed at the previous time step $t - 1$.

This is exactly the way in which a **Recurrent Neural Network (RNN)** operates [24]. After receiving an *initial hidden state* $h^{(0)}$ together with the input sequence $x^{(1)}, \ldots, x^{(n)}$, it iteravely computes all $y^{(t)}$ by considering the previous, so called *hidden state* $h^{(t-1)}$ and repeatedly applying (11).

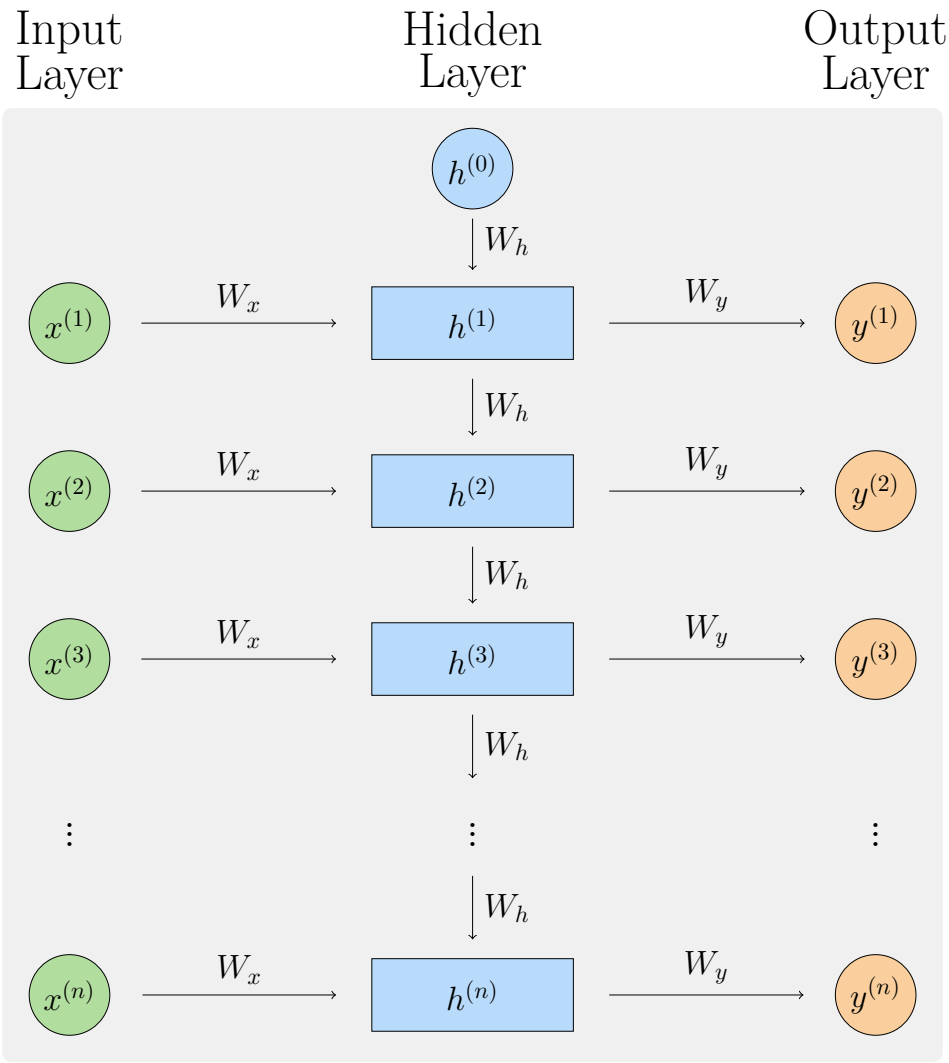


Figure 8: Simple Recurrent Neural Network

Figure 8 is the groundlaying architecture of a **Simple Recurrent Neural Network (Simple RNN)** [25]. The calculation of the hidden states h_t in the blue blox as well as the weight matrices W_x, W_h, W_y remain the same throughout every time step $t = 1, \ldots, n$. There are also bias vectors b_h and b_y that get added in the hidden layer and output layer respectively, ommited in the figure for sake of clarity. Terminologywise, from here on, the computation that happens in one line of figure 8 will be called a *cell* of the RNN.
The activation function for the hidden states is usually chosen to be a sigmoid or the tanh [26], where the latter tends to perform more stable during backpropagation for reasons we will see in the next subsection about van-

ishing and exploding gradients.
Depending on the task, the final output is then given either by the last output $y^{(n)}$ as in sentiment analysis [27] or, as in language translation tasks, by the entire sequence $y^{(1)}, \ldots, y^{(n)}$ [26]. If the i-th entry of the output layer shall represent the probabilty that said output corresponds to the i-th word of a dictionary, $softmax$ usually gets chosen as outer activation funtion and therefore applied to every output vector $y^{(t)}$ [28].
To perform a Logistic Regression, the loss at time step t is naturally given by

$$L_t(y^{(t)}, y^{(t)}_{true}) = y^{(t)}_{true} log(y^{(t)})$$

Since we need to consider the losses throughout all time steps, we derive the cumulated Loss function

$$L(Y, Y_{true}) = -\sum_{t}^{n} L_t(y^{(t)}, y^{(t)}_{true}) \tag{12}$$

$$= -\sum_{t}^{n} y^{(t)}_{true} log(y^{(t)}) \tag{13}$$

where $y^{(t)}_{true}$ is the target vector at time step t, Y the matrix consisting of all $y^{(1)}, \ldots, y^{(n)}$ and Y_{true} the matrix consisting of all $y^{(t)}_{true}$ respectively [29].

Other combinations of sequence lengths regarding input and output are possible, too. In image description for example, the structure is reversed: Our input is a single image that we want to describe with a sequence of words [30]. To use the presented RNN architecture, we could define our input sequence as $x^{(t)} := x$ and to be of the same length as the desired output sequence by repeating the single input vector x accordingly many times if the length is known beforehand or until an output resembles a predefined end-token. Another approach is to feed the input x to a Convolutional Neural Network first and use its output for the Recurrent Neural Network. [31].
In general, input and output sequence may differ in length. For instance, a translated sentence may not consist of the same amount of words as the original one. A popular solution is to use an Endcoder-Decoder structure, where the input sequence gets fed in an RNN denoted to be the Encoder. The last hidden state is then passed to initialize another RNN, the Decoder, which then uses its own previous output as input for any current time step [32]. Nevertheless, since we will not encounter these scenarios in Job Scheduling Problems, we will refrain from a more detailed explanation at this point.

One big disadvantage of the time-dependent-series approach is that whenever a RNN is seeing the t-th input $x^{(t)}$ of its input sequence, it has already seen the inputs of all preceding time steps, yet none of the succeeding. The

context of a word, however, is not only defined by the previous ones in a sentence, but also by the ones following it. To integrate this into our architecture and improve the ability of our Neural Network to interprete the context of each input step, we can use a Bidirectional Wrapper for our RNN as in figure 9:

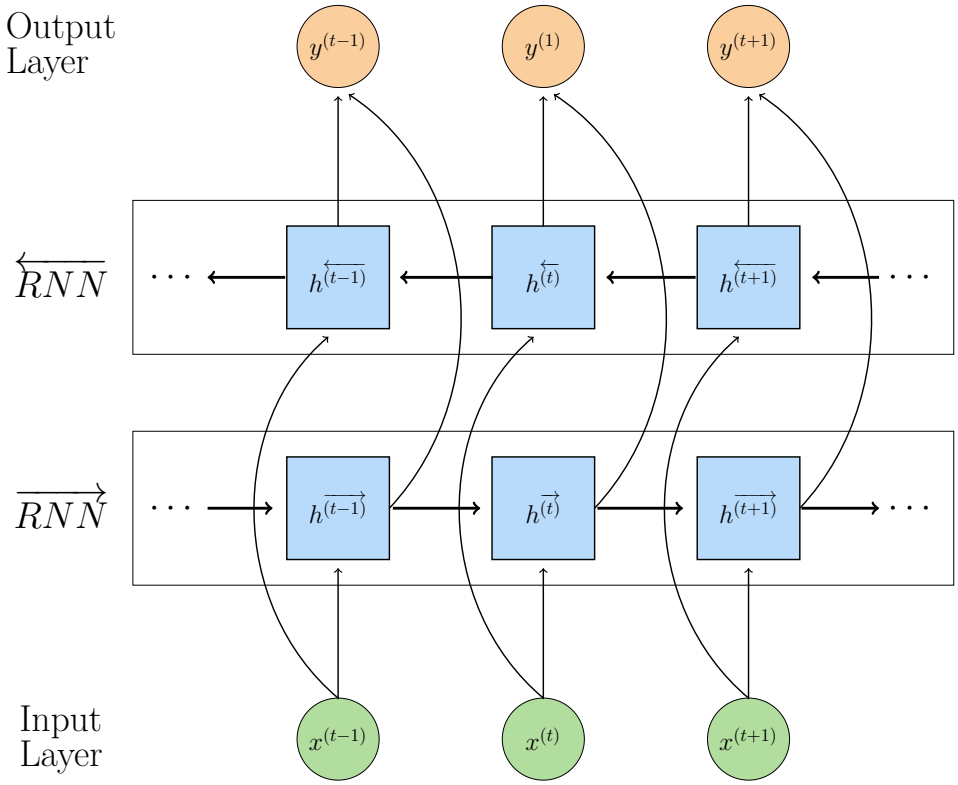


Figure 9: Bidirectional RNN

Here, the input sequence $x^{(1)}, \ldots, x^{(n)}$ gets fed into our RNN normally, creating the forward hidden states $h^{\overrightarrow{1}}, \ldots, h^{\overrightarrow{n}}$. In addition, the same input sequence gets fed parallely into the same RNN in reversed order $x^{(n)}, \ldots, x^{(1)}$, thus chronologically creating the backward hidden states $h^{\overleftarrow{n}}, \ldots, h^{\overleftarrow{1}}$. All hidden states get then multiplied by W_y as in figure 8, resulting in the forward output sequence $y^{\overrightarrow{1}}, \ldots, y^{\overrightarrow{n}}$ and the backward output sequence $y^{\overleftarrow{n}}, \ldots, y^{\overleftarrow{1}}$. By concatening the according vectors, i.e. by defining

$$y^{(t)} := concat(y^{\overrightarrow{t}}, y^{\overleftarrow{t}}) \tag{14}$$

for every time step $t = 1, \ldots, n$, we obtain the resulting output sequence $y^{(1)}, \ldots, y^{(n)}$ [33].

Another way of providing the Neural Network with more context about each time step of the input sequence is to stack RNNs onto each other [34]. Here, the hidden states $h^{(1)}, \ldots, h^{(n)}$ of one *bottom* RNN are fed as input sequence to a second *top* one. Figure 10 illustrates this architecture:

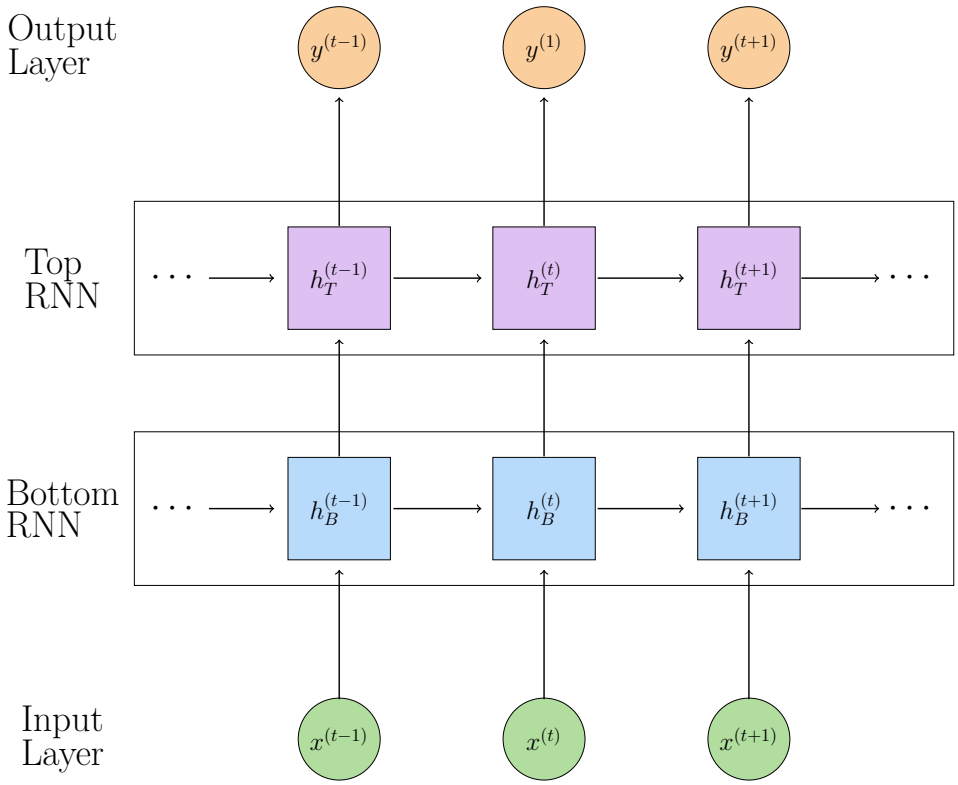


Figure 10: Stacked RNN

An arbitrary amount of RNNs can be stacked. This has the advantage that every input from the second RNN on has already be computed by considering the entire input sequence, hence providing it with some context from the start. A similiar approach would be to use the original input sequence for every stacked RNN as well, but set its initiate hidden state to be equal to the final hidden state of the preceding one.
Naturally, to take advantage of both approaches, one can combine them by stacking Bidirectional RNNs onto each other. Nevertheless, one mayor structural disadvantage remains for Simple RNNs due to their learning process, as will be discussed in the following subsection.

4.3 Vanishing/Exploding Gradients

We will now take a more technical look at how the Simple RNNs learn, which will confront us with the problem of the vanishing/exploding gradient [35] and, as a result, lead us to the improved architecture of *Long Short-Term Memory* (*LSTM*) cells that will be presented as a solution subsequently.

Let us first further split the equation (11) into two parts:

$$y^{(t)} = f_y(h^{(t)}) = \phi_y(W_y * h^{(t)} + b_y) \tag{15}$$
$$h^{(t)} = f_h(x^{(t)}, h^{(t-1)}) = \phi_h(W_x * x^{(t)} + W_h * h^{(t-1)} + b_h) \tag{16}$$

Here, ϕ_y is the activation function in the output layer, usually a $softmax$, and ϕ_h the activation function for the hidden layer, usually chosen to be $tanh$ or the $sigmoid$-function.

To understand how the Network learns, we have to take a look at its back-propagation. We will focus on the gradients for the weights of W_h.

Due to equation (12), we have:

$$\frac{\partial L(Y, Y_{true})}{\partial W_h} = -\sum_{t}^{n} \frac{\partial L_t(y^{(t)}, y_{true}^{(t)})}{\partial W_h}$$

For better readability, in the following we will write L_t short for $L_t(y^{(t)}, y_{true}^{(t)})$. If we further look at the gradient of the $t+1$-th time step, we get:

$$\frac{\partial L_{t+1}}{\partial W_h} = \frac{\partial L_{t+1}}{\partial y^{(t+1)}} \frac{\partial y^{(t+1)}}{\partial h^{(t+1)}} \frac{\partial h^{(t+1)}}{\partial W_h}$$

The hidden state $h^{(t+1)}$, however, also depends on the previous hidden state $h^{(t)}$ due to the recursive structure of the Neural Network. Hence, applying the chain rule gives us:

$$\frac{\partial L_{t+1}}{\partial W_h} = \frac{\partial L_{t+1}}{\partial y^{(t+1)}} \frac{\partial y^{(t+1)}}{\partial h^{(t+1)}} \frac{\partial h^{(t+1)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial W_h}$$

Repeating this for every foregoing time step produces the equation

$$\frac{\partial L_t}{\partial W_h} = \sum_{k}^{t} \frac{\partial L_{t+1}}{\partial y^{(t+1)}} \frac{\partial y^{(t+1)}}{\partial h^{(t+1)}} \frac{\partial h^{(t+1)}}{\partial h^{(k)}} \frac{h^{(k)}}{W_h}$$

Again due to the recursive dependency of the hidden states and the chain rule, we can rewrite the term $\frac{\partial h^{(t+1)}}{\partial h^{(k)}}$ as:

$$\frac{\partial h^{(t+1)}}{\partial h^{(k)}} = \prod_{j=k}^{t} \frac{\partial h^{(j+1)}}{\partial h^{(j)}}$$

This leaves us with the expression:

$$\frac{\partial L_t}{\partial W_h} = \sum_{k}^{t} \frac{\partial L_{t+1}}{\partial y^{(t+1)}} \frac{\partial y^{(t+1)}}{\partial h^{(t+1)}} (\prod_{j=k}^{t} \frac{\partial h^{(j+1)}}{\partial h^{(j)}}) \frac{h^{(k)}}{W_h}$$

The part of this term that will therefore make the gradient vanish or explode by exponentially decreasing or growing is $\prod_{j=k}^{t} \frac{\partial h^{(j+1)}}{\partial h^{(j)}}$. Due to equation (16) we can rewrite this as:

$$\prod_{j=k}^{t} \frac{\partial h^{(j+1)}}{\partial h^{(j)}} = \prod_{j=k}^{t} diag(\phi_h^{'}(h^{(j)}) * W_h \tag{17}$$

Let us now take a look at the norms of the Jacobian Matrices of equation (17). We denote λ to be the greatest eigenvalue of W_h. If we assume its norm to be upper bounded by a $\gamma_j \in \mathbb{R}_+$ for every $j = k, \ldots, t$, i.e.

$$|| \; diag(\phi_h^{'}(h^{(j)}) \; || \leq \gamma_j \tag{18}$$

we can give an upper bound to the entire term:

$$\begin{aligned}
|| \prod_{j=k}^{t} \frac{\partial h^{(j+1)}}{\partial h^{(j)}} || &\leq \prod_{j=k}^{t} || \frac{\partial h^{(j+1)}}{\partial h^{(j)}} || && (19)\\
&\leq \prod_{j=k}^{t} || \; diag(\phi_h^{'}(h^{(j)}) \; || * || \; W_h \; || && (20)\\
&\leq \prod_{j=k}^{t} \gamma_j * \lambda && (21)\\
&\leq (\gamma * \lambda)^{t-k} && (22)
\end{aligned}$$

for $\gamma := \max_j(\gamma_j)$. If now $\lambda < \frac{1}{\gamma}$, its upper bound of (22) and therefore the term $|| \prod_{j=k}^{t} \frac{\partial h^{(j+1)}}{\partial h^{(j)}} ||$ itself converges to zero, resulting in a vanishing gradient for any sequence long enough.

As mentioned, ϕ_h usually gets chosen to be *tanh* or sigmoid. The derivative of *tanh* is given by

$$\frac{\partial tanh(z)}{\partial z} = 1 - tanh(z)$$

and its absolute value therefore upper bounded by 1.
For sigmoid, the absolute value is even already upper bounded by 0.25, since its derivative is given by:

$$\frac{\partial sigmoid(z)}{\partial z} = sigmoid(z) * (1 - sigmoid(z))$$

Therefore, the assumption of (18) would be fulfilled in both cases and the vanishing of the gradient would be ensured in case that $\lambda < 1$ for *tanh* and $\lambda < 0.25$ for *sigmoid*, thus making *sigmoid* the less stable choice for an activation function in regards to this problem.

Reversing this proof gives the *necessary* condition of $\lambda > \frac{1}{\gamma}$ for the gradient to explode. However, vanishing gradients are the bigger issue for Simple RNNs. Also, gradient clipping provides a stable solution to exploding gradients by taking the minimum of a fixed value and the calculated gradient for any gradient descent step (hence clipping it above a certain value) [36].

4.4 Long Short-Term Memory

As seen in the prvious subsection, Simple RNNs struggle with the problem of vanishing gradients when it comes to parameter optimization for longer sequences and may therefore fail to give a correct context in NLP applications for words that are far apart from each other. Lets take a look at the following sentence:

"The student, which already put a lot of thought into how to explain the challenges and solutionary architectures needed for the master thesis in an illustrative way, still found himself struggling to think of a random examplary sentence."

and compare it to:

"The students, which already put a lot of thought into how to explain the challenges and solutionary architectures needed for the master thesis in an illustrative way, still found themselves struggling to think of a random examplary sentence."

These are identical but for two words: In the first sentence, the singular of *student* is used, while the subject of the second one is the plural form. Because of this, the reflexive pronoun "*himself/themselves*" gets conjugated differently towards the end of the sentence (or more precisely, after the inserted subordinate clause). Hence, the Neural Network has to remember the numerical order of the object from the beginning of the sentence to

chose the right persona for the reflexive pronoun towards the end of it. This can be challenging for Simple RNNs due to the mentioned problem of the vanishing gradients. To use more intuitive expressions, they rather rely on a kind of short-term memory due to their tendency to focus stronger on parts of the input sequence recently seen before, while having trouble to remember connections from longer ago [37]. We therefore might wish for some modifications in the architecture to implement additional information that tends to resemble some kind of long-term memory as well. This is were Neural Networks called **Long-Short-Term-Memory** (**LSTM**) come into play. These are an extension of RNNs, designed to tackle the mentioned problem of vanishing (and also exploding) gradients and therefore enabling the processing of larger sequences. The following information is based on [38].

Just like Simple RNNs, the LSTM is a type of recurrent Neural Network, consisting of one *unit*, refered to as the *memory unit* or the *LSTM unit* in the literature, that gets run through iteratively by every part of an input sequence, while also taking in some information created at the previous time step (which again depends on information of the foregoing time step etc). The difference to Simple RNNs is that these only receive the hidden state $h^{(t-1)}$ as additional input from the previous time step. In an LSTM however, there also is a vector $c^{(t-1)}$ that gets fed to the LSTM unit at time step t, in addition to the hidden state $h^{(t-1)}$ and the input $x^{(t)}$. This additional vector is called the *cell state* and is meant to resemble some type of long-term memory that gets passed along and modified at every time step. Lets now take a look at how these LSTM units are constructed in detail:

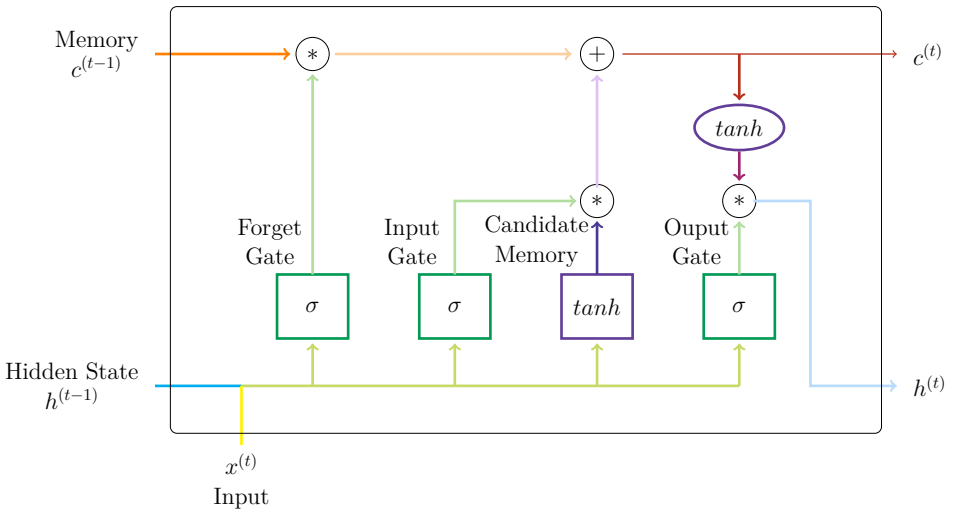


Figure 11: LSTM Unit

Figure 11 shows that an LSTM unit takes $c^{(t-1)}$, $h^{(t-1)}$ and $x^{(t)}$ as inputs and is built of 3 gates and a *Candidate Memory*. The gates gate how much information should be let through, while the Candidate Memory is a potential addition to the already existing information based on the current time step. Each of these are Feedforward Neural Networks on their own, consisting of an activation function ϕ_i and two trainable weight matrices W_i and V_i, $i = 1, 2, 3, 4$. Any of these Networks takes $h^{(t-1)}$ and $x^{(t)}$ as inputs, multiplies them by W_i or V_i respectively, sums up the two resulting vectors and then applies the activation function ϕ_i:

$$(x^{(t)}, h^{(t-1)}) \mapsto \phi_i(W_i x^{(t)} + V_i h^{(t-1)})$$

We will now go through each of these gates, one at a time, to explain their function in the unit. To better illustrate the idea behind the architecture, we often will use the more intuitive terms long-term memory and short-term memory instead of the technical terms cell state and hidden state.

First, there is the *Forget Gate* of figure 12:

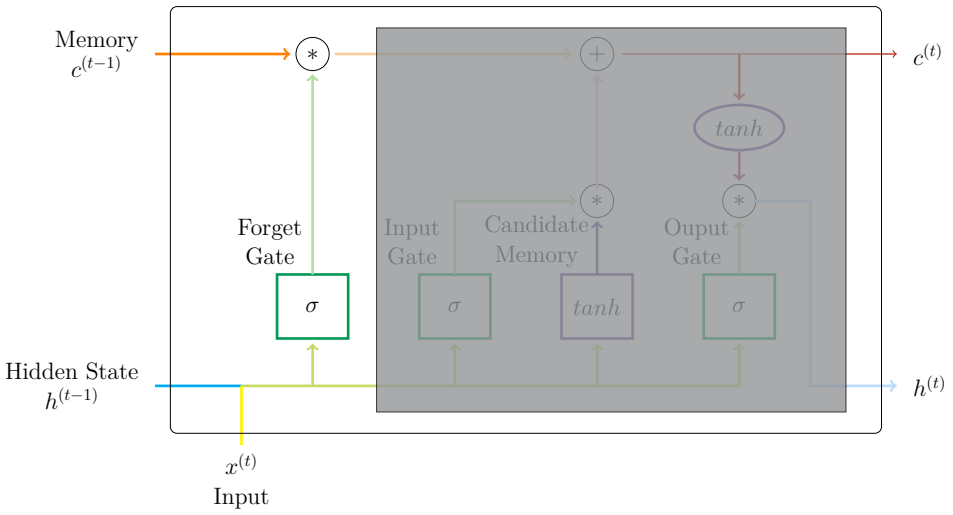


Figure 12: Forget Gate

Its activation function is *sigmoid*, therefore it outputs a vector with values ranging between 0 and 1. These values then get multiplied entrywise with the cell state $c^{(t-1)}$. Hence, the Forget Gate determines what percentage of each entry of the long-term memory shall be remembered (and therefore how much should be "forgotten").

Now we want to update our remaining long-term memory with the information of the current time step:

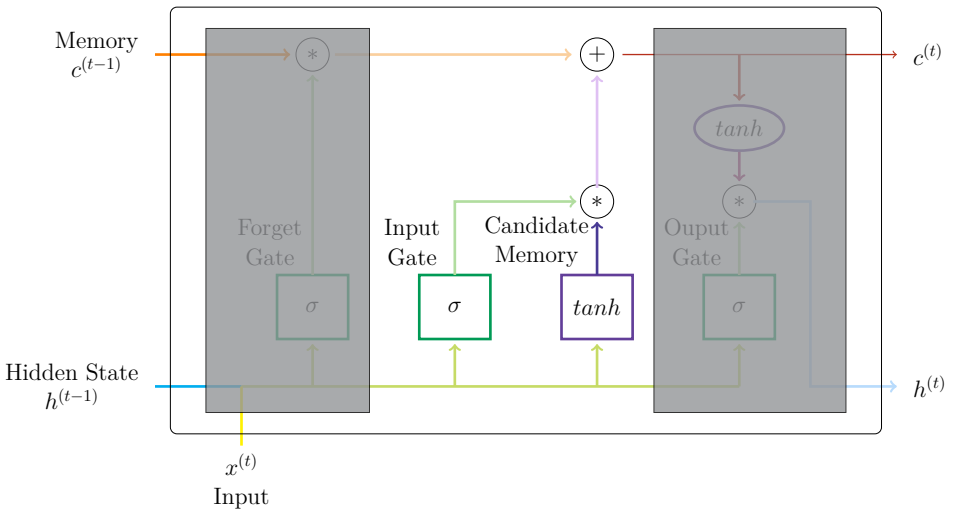


Figure 13: Input Gate and Candidate Memory

The Candiate Memory of figure 13 resembles this information. Since its activation function is *tanh*, it will output a vector with values between -1 and 1. Before adding them to the long-term memory, these values get multiplied with the ones produced by the *Input Gate*. Having a *sigmoid* as activation function, it therefore decides what percentage of each information of the current time step shall be added to the long-term memory.

The vector that we obtain for the long-term memory after applying these operations will be the updated long-term memory, the cell state $c^{(t)}$. Regarding this new context, we now also want to update our short-term memory $h^{(t-1)}$:

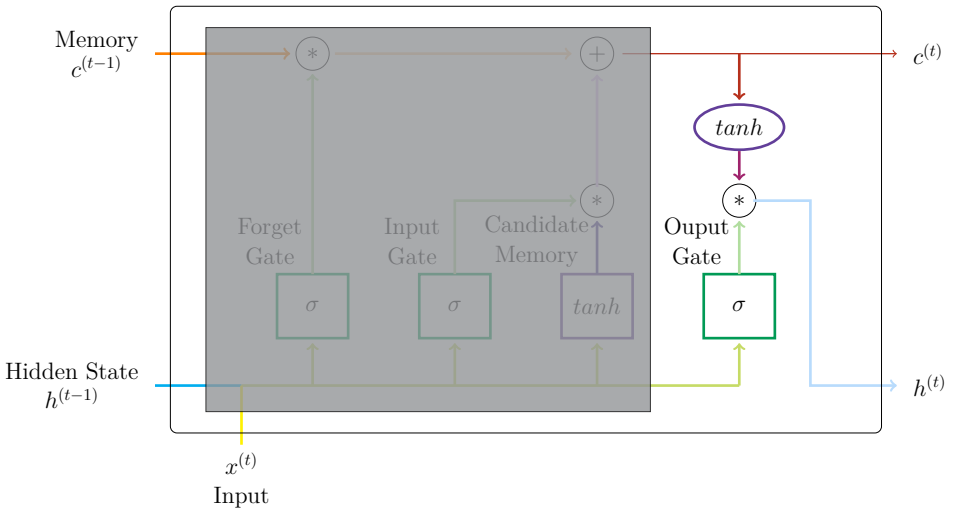


Figure 14: Output Gate

For this, we apply *tanh* to $c^{(t)}$ as shown in figure 14, receiving a vector ranging between -1 and 1, therefore being the Candidate Memory for the new short-term memory. Having a *sigmoid* as activation function, the *Output Gate* then decides what percentage of these values shall be outputted to be the updated short-term memory $h^{(t)}$ based on the previous short-term memory $h^{(t-1)}$ and the new information $x^{(t)}$, revealed at the current time step t.

The advantage of these gates is that they enable the LSTM to learn how to control the flow of information. When applying the backpropagation through time in the gradient descent method, the ratio of growth $\frac{\partial c^{(t)}}{\partial c^{(t-1)}}$ between consecutive cell states are part of the chain of factors of derivates created by the chain rule, acting as learnable scalings to the other factors [39]. Thus, they work as controlling instances to the growth of the gradients, preventing them from vanishing or exploding for larger sequences. As a con-

sequence, LSTMs are able to process instances of a sequence not only based mainly on information from inputs seen shortly before, but by considering parts of the sequence that has been fed to it at time steps longer ago as well, therefore also exhibiting long-term memory-based behaviours [28].
Just like Simple RNNs, LSTMs can be integrated into bidirectional wrappers or stacked onto each other as well [40]. Note that in this case in addition to the final hidden state $h^{(n)}$, the final cell state $c^{(n)}$ gets passed to initiate the succeeding LSTM, too.

4.5 Attention

With the presented architectures to this point, we are now able to implement context in the form of short-term and long-term memory. These representations of context are a result of all the information that the Neural Network has seen up to the present time step, of which it is independent of. However, certain parts of the input sequence may be more important than others to determine the context for the current instance, while the others might be more significant at a different time step. Let us say we want to translate the sentence:

"I am measuring the length of my bathroom to know how many feet of carpet I should buy."

To output the correct translation for "bathroom", the Neural Network mainly has to consider the word itself, while the information of what is being done in there can most completely be neglected. However, when it comes to translating "feet", it does have to take into account that the context is given by "measuring" and "length", so it does not output the translation of the respective body part. Hence, every instance should pay **Attention** to different parts of the input sequence to gather the needed context. We therefore would like to expand our recurrent architecture by some type of attention-mechanism that tells the Network on which of the presented information it should focus given the current input to correctly and efficiently grasp its context. Since, typicial for Deep Learning Problems, we do not know beforehand which values should ideally trigger which attention-response to which extend, we want to constrcut the general architecture, but make the precise weighting of the parameters a part of the learning process of the Neural Network.

The general idea is to implement such an attention-mechanism based on the structure of a query request [41]. Traditionally, these work the following way: We have a query $q \in \mathbb{R}^{d_q}$ and a dictionary of keys $k_1, \dots, k_n \in \mathbb{R}^{d_k}$ with associated values $v_1, \dots, v_n \in \mathbb{R}^{d_v}$. We check which key fits the query best and output the corresponding value.

The difference now is that we do not decide for the best fitting key, but instead score every key by *how well* it fits the given query.
There are different ways to do so. Here, we will use the one presented in the paper [42], which has been a milestone in the field of NLP. To do so, we need two weight matrices W_q and W_k. The attention scores $u \in \mathbb{R}^n$ are then computed proportionally to the mathematical similarity of the weighted query and the weighted keys, scaled by the square root of their dimension $\sqrt{d_k}$:

$$u_i := \frac{< W_q * q, W_k * k_i >}{\sqrt{d_k}} \tag{23}$$

These weight matrices are the learnable parameters for the Neural Network. Next, we apply $softmax$ to these scores to obtain a distribution $a \in D^{'}(\{k_1, \ldots, k_n\})$ of the attention over the keys, corresponding to how fitting any of them is compared to the others regarding the current query:

$$a := softmax(u) \tag{24}$$

Consequently, for every key k_i, its entry for the attention distribution a_i indicates how much the associated value v_i should contribute to the final value $z \in \mathbb{R}^{d_v}$, which therefore will be the weighted sum:

$$z := \sum_{i=1}^{n} a_i * v_i \tag{25}$$

In the environment of NLP, the keys usually are identical to their values (i.e. $k_i = v_i$) and correspond to the information available for the Network to define the context of the current query.
In case that the query is an element of this sequence $q \in \{v_1, \ldots, v_n\}$, too, we call this procedure **Self-Attention**. Self-Attention can be used to determine the meaning of a word within the context of its sentence like in our example stated above [43].
Otherwise, we are applying **Cross-Attention**. If, for example, we have a label for a sentence as query in sentiment analysis, the attention score can indicate the importance of each word of the sentence for the Neural Network in deciding for said label [44].

Attention can also be integrated when stacking RNNs or when using an Encoder-Decoder architecture [45]. For instance, Self-Attention can be applied to the sequence of hidden states $h_B^{(1)}, \ldots, h_B^{(n)}$ from figure 10 that gets passed from one Network to the next one, while a succeeding LSTM can be initiated with the vector obtained by using the final hidden state or cell of the preceding one as query and applying Cross-Attention over the input

sequence $x^{(1)}, \ldots, x^{(n)}$. A slight variation of the latter technique will be the central aspect of the following subsection.

4.6 Pointer Networks

We already have analyzed multi-variate mapping problems and how to use the RNN architecture to create output sequences of dynamic lengths in this section. Yet, every vector of this output sequence still has to be of a predefined, fixed dimension. When $softmax$ is used as outer activation function in the RNN, the i-th entry of any of its outputs usually denotes the probability to chose the corresponding dictionary entry. So this architecture suffices when working with static dictionaries, meaning the vocabulary that the Neural Network can map to consists of exactly N elements for any given input.

However, we might want to expand the set of mapping problems we can address by those requiring a dynamic dictionary size, where the vocabulary does depend on the respective input. Let us postulate that we have a sequence of words and want to point to the one which does not fit the others like in figure 15:

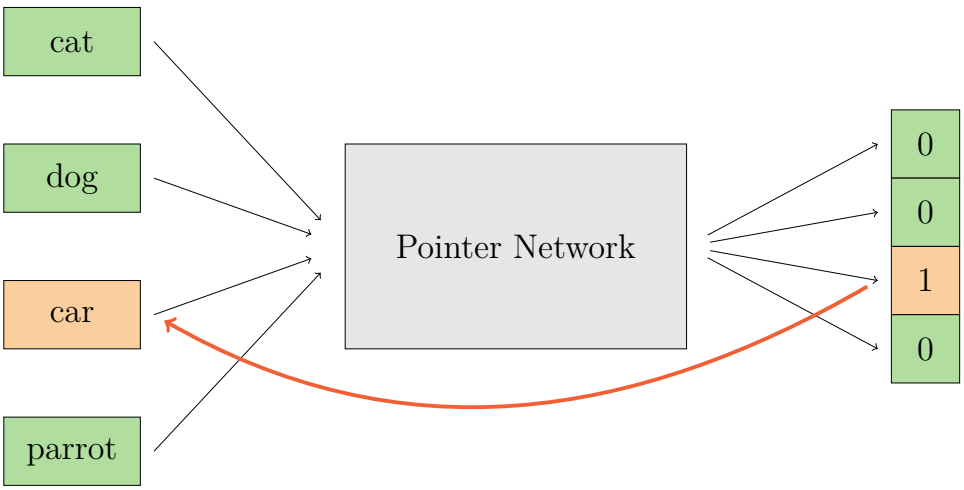


Figure 15: Pointer Network

In this case, the vocabulary would be equal to the input sequence whose length would therefore define the dictionary size. Consequently, if input sequences differ in the number of words they consist of, the size of our output vector has to as well. Using Attention enables us to implement such an architecture:

As usual, we run our input sequence through our RNN time step by time step, until we receive the overarching context in form of our final hidden state $h^{(n)}$. We now use $h^{(n)}$ as query and the instances of the input sequence $x^{(1)}, \ldots, x^{(n)}$ as keys to apply Cross-Attention. To create the scores $\hat{u} \in \mathbb{R}^n$ of the keys, the paper [46], being the one that originally introduced this approach, uses a computation method slightly different to (23). It completely omits to assign associated value vectors $v^{(1)}, \ldots, v^{(n)}$ and therefore to compute the final weighted sum of (25). Instead, it introduces a single value vector $v \in \mathbb{R}^{d_v}$, consisting of d_v *learnable* parameters, suiting the dimensions of the weight matrices $W_q \in \mathbb{R}^{d_v \times d_q}$ and $W_k \in \mathbb{R}^{d_v \times d_k}$. By applying $tanh$ to the sum of the weighted vectors $W_q * q$ and $W_k * k_i$, the resulting term indicates how much of each entry of v should be added or substracted to obtain the corresponding attention score:

$$\hat{u}_i := < v, tanh(W_q * q + W_k * k_i) > \tag{26}$$

The attention distribution $\hat{a} \in D^{'}(\{k_1, \ldots, k_n\})$ is then computed analogously to (24):

$$\hat{a} := softmax(\hat{u}) \tag{27}$$

Since we chose our keys to be equal to the input sequence, in particular $\hat{a} \in D^{'}(\{x_1, \ldots, x_n\})$ is a probability distribution over the instances of the input sequence. Hence, by defining the output $y \in \mathbb{R}^n$ of the Neural Network to be

$$y_i := \hat{a}_i \tag{28}$$

we created an architecture in which it maps back to the input sequence by *pointing* at its instances. Neural Networks with this kind of achitecture are therefore refered to as **Pointer Networks**. Due to our RNN being able to process input sequences of dynamic length, the corresponding vocabulary is now dynamic in size and content as well, thus fulfilling our stated requirements. This architecture does not work exclusively with RNNs but also with other, non-recurrent Neural Networks that are able to process sequences as input like the Transformer, which will be introduced in the next subsection.

Stated as motivation in their creation in [46], Pointer Networks can in particular be used for tasks in which there are several possible actions and the Neural Network shall decide which one to take. The input sequence is then the sequence of (feasible) actions and the output points towards which of them may be optimal with regards to a certain objective. Therefore, they also get called **Action-Pointers**. We will take advantage of this property

in our task of scheduling Jobs by applying such an Action-Pointer to a sequence of numerical representations of the Jobs to map to the set of actions of feasible Job assignments to the currently free machine, appended by the action of turning it off. From this mapping we will then deduce a schedule by taking the action it estimates to be optimal, being the one corresponding to the entry with the highest probability/attention.

4.7 Transformers

In the previous subsections, we have explained how to implement the attention mechanism of the paper [42] from 2017 into Neural Networks that are constructed based on a recurrent architecture to improve their interpretation of the individual context of instances to which it is applied. However, one decisive disadvantage of RNNs remains, since any output $y^{(t)}$ depends on all the previous hidden states $h^{(1)}, \ldots, h^{(t-1)}$, which consequently first have to be sequentially computed. The same paper therefore had a big impact on the further development of the field of NLP by presenting the **Transformer** for language translation, an Attention based Neural Network where during the training phase all instances of the output sequence are created independently from each other, hence allowing for much faster computation by making use of parallelization instead of relying on recurrent processing. They typically consist of an Encoder and a Decoder. We will now analyze the composure of all their layers in detail, starting with the Encoder, illustrated in figure 16:

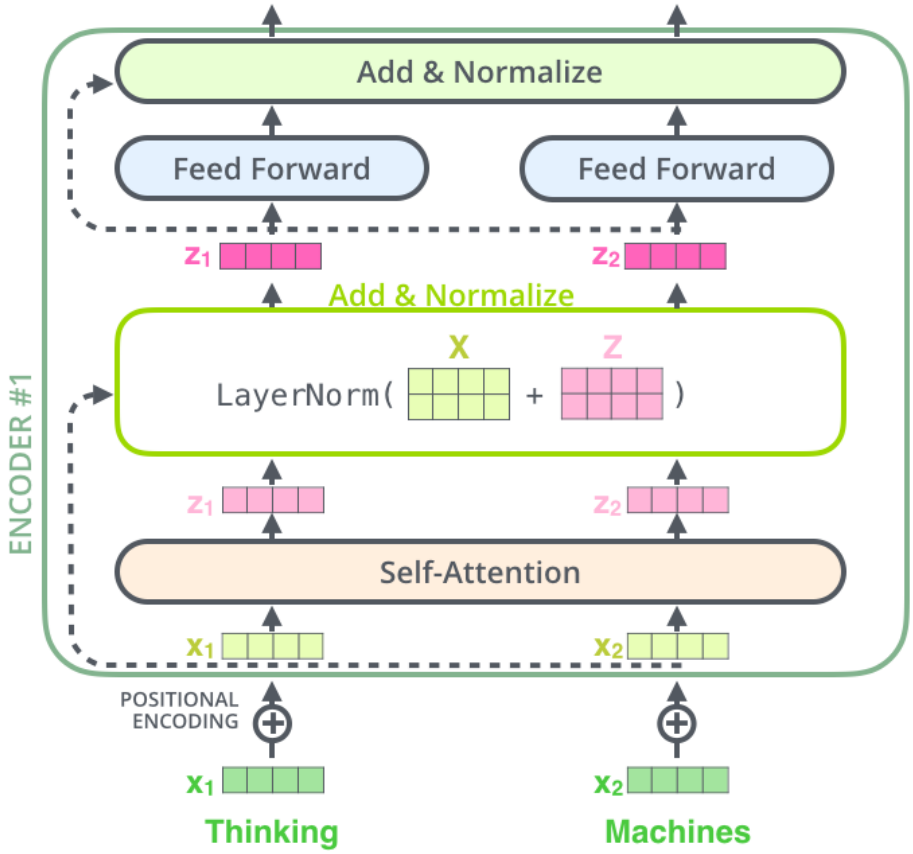


Figure 16: Encoder of Transformer [47]

The Encoder consists of several so called Attention-Heads. Their exact number $H \in \mathbb{N}$ is set to 8 in the mentioned paper, however any number is possible. The entire input sequence $x^{(1)}, \ldots, x^{(n)}$ gets fed to the Encoder, where each of the Attention-Heads separately applies Self-Attention to every instance of this sequence, producing the sequences $z_h^{(1)}, \ldots, z_h^{(n)}$, with h being the index of the respective Attention-Head. Therefore, any of these $z_h^{(t)}$ for $h = 1, \ldots, H$ is an encoding of the associated word $x^{(t)}$ with regards to its meaning in the context of all words $x^{(1)}, \ldots, x^{(n)}$ from the input sequence. Since it uses several Attention-Heads initialized to distinct values, the Transformer can encode different aspects of the contextual meaning of every word. Note that all the computations within an Attention-Head as well as inbetween them happen in parallel, since they do not depend on each other.

After applying Self-Attention, for every $t = 1, \ldots, n$ we concatenate all H respective vectors $z_1^{(t)}, \ldots, z_H^{(t)}$ and apply a learnable weight matrix $W_0 \in \mathbb{R}^{d_x \times (H * d_z)}$, where d_x is the dimension of any input instance $x^{(t)}$ and d_z of any $z_h^{(t)}$ respectively, so that

$$z_{att}^{(t)} := W_0 * \begin{pmatrix} z_1^{(t)} \\ \vdots \\ z_H^{(t)} \end{pmatrix} \tag{29}$$

is of dimension d_x again. To enhance the stability of the learning process, a Residual Connection is added by summing $x^{(t)}$ and $z_{att}^{(t)}$ and then Layer Normalization is applied:

$$z_{norm}^{(1)}, \ldots, z_{norm}^{(n)} := LayerNorm(x^{(1)} + z_{att}^{(1)}\ , \ldots,\ x^{(n)} + z_{att}^{(n)}) \tag{30}$$

Subsequently, the normalized vectors get fed parallely into the same Feed-Forward layer FF, consisting of an inner Weight matrix W_{in} with an activation funtion ϕ_F, usually the *Rectified Linear Unit (ReLU)*, and an outer weight matrix W_{out} as well as bias vectors b_{out} and b_{in}:

$$z_{forw}^{(t)} := FF(z_{norm}^{(t)}) = W_{out} * \phi_F(W_{in} * z_{norm}^{(t)} + b_{in}) + b_{out} \tag{31}$$

Afterwards, another Residual Connection and Layer Normalization is added:

$$z^{(1)}, \ldots, z^{(n)} := LayerNorm(z_{norm}^{(1)} + z_{forw}^{(1)},\ \ldots,\ z_{norm}^{(n)} + z_{forw}^{(n)}) \tag{32}$$

These $z^{(1)}, \ldots, z^{(n)}$ are the final output sequence of the Encoder. They can then be passed to the Decoder or to the next Encoder, in case we have several of them stacked onto each other, analogously to the way we stacked RNNs in figure 10. All Encoders would then be identical in their architecture, yet not share weights. The first (or *bottom*) Encoder would receive the input sequence $x^{(1)}, \ldots, x^{(n)}$, while the latter ones would use the output sequence $z^{(1)}, \ldots, z^{(n)}$ of the former Encoder as input, until the final (or *top*) Encoder sends its output sequence to the Decoder. In case that there is a stack of Decoders as well, said sequence is send to all of them, while the output of any Decoder is used as input for the following one. Figure 17 illustrates the entire architecture of such a Transformer:

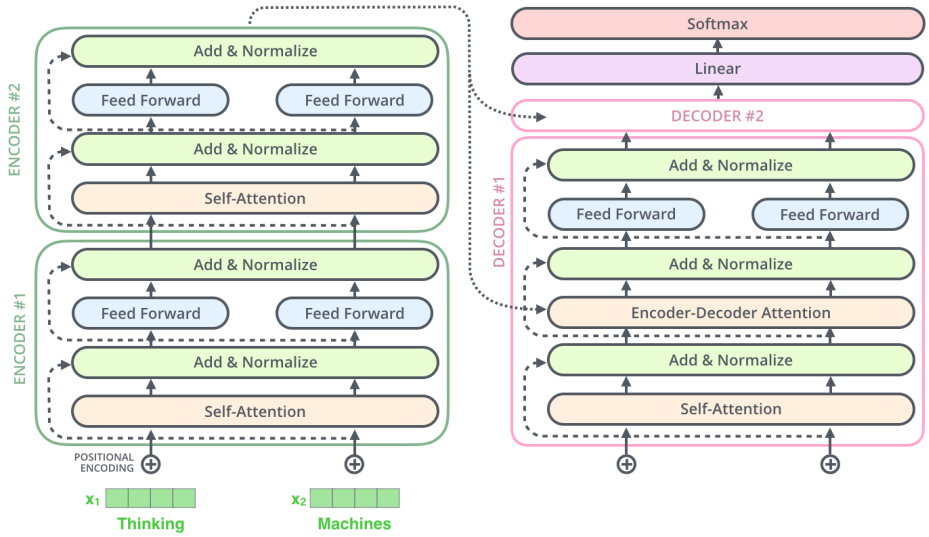


Figure 17: Transformer [47]

It is worth mentioning that, due to construction, the ordering of the instances of the input sequence does not matter anymore, which is why a positional encoding gets added to the embedded representation of any word before passing them to the bottom Encoder. Since this is not relevant for our task, we will not focus further on it.

We will now analyze the Decoder. As figure 17 shows, architecturewise it is similiar to the Encoder, but with one more Attention layer added. Analogous to the Encoder, after every Attention and Feed-Forward layer there will be applied a Residual Connection, which sums the vector representation from before that layer with the one obtained after it, and subsequently a Layer Normalization.

To define the input of the Decoder, we first have to look at what its output is supposed to look like. The output of the top Decoder passes a Linear layer before finally $softmax$ gets applied to it as outer activation function. The final output of the Transformer is hence a probability distribution over the vocabulary dictionary, indicating which entry it estimates to be the correct one. In language translation, the desired output often is a sentence, therefore a sequence of length $N \in \mathbb{N}$ of these distributions over all possible words. To produce the correct word for any given time step $T = 1, \dots, N$, we ideally want to use as much information about its context as possible. That means taking into account not only the encoded input sequence $z^{(1)}, \dots, z^{(n)}$ passed by the Encoder, but also the previous words of the translated sentence, mak-

ing their vector representation a natural choice as inputs for the Decoder. However, this would result in a recurrent structure again, for which in order to compute the estimate $y^{(T+1)}$ of the next word we would first need to have created all antecedant estimates $y^{(1)}, \dots, y^{(T)}$. To get rid of this sequential dependency in the training phase, as well as to speed it up by providing the Transformer with additional information, there is often something applied called *Teacher Forcing*. This means that instead of feeding the sequence of previous estimates as input to the Decoder, its input is set to be the desired output, i.e. the target vectors $y^{(1)}_{target}, \dots, y^{(N)}_{target}$, provided by the training data. So if we are dealing with a language translation task from English to German and the input sequence $x^{(1)}, \dots, x^{(n)}$ of the Encoder is the embedding of the sentence

"What do you want to do?"

the sentence we want to obtain as a translation would be:

"Was möchtest du machen?"

The embedding of this translated sentence is the sequence we would therefore use as input for the Decoder, shifted by one time step (therefore "Was" would be the input for the Decoder at time step $T = 2$), with the very first input being an initialization token. This allows the Transformer to compute all outputs $y^{(1)}, \dots, y^{(N)}$ in parallel during its training phase. So in order to create $y^{(T+1)}$, $T = 1, \dots, N-1$ ($y^{(1)}$ is always set to be the initialization token), in the Self-Attention layer of the Decoder we define $y^{(T)}_{target}$ as query and the sequence of targets $y^{(1)}_{target}, \dots, y^{(N)}_{target}$ as keys and values. Since with this architecture we want to mimic that $y^{(T+1)}$ depends on the previous outputs $y^{(1)}, \dots, y^{(T)}$, yet not on the future ones $y^{(T+2)}, \dots, y^{(N)}$, we have to annulate the influence of the targets $y^{(T+1)}_{target}, \dots, y^{(N)}_{target}$. In other words, we do not want the query to pay any Attention to them at all. We achieve this by applying a *mask*. This mask replaces all attention scores $u_i^{(T+1)}$ of (23) for any subsequent target $y^{(i)}_{target}$ with $i > T$ by $-\inf$, so that when applying $softmax$ in (24) their attention values $a_i^{(T+1)}$ will be equal to zero.

The vector caclulated by the Masked Self-Attention layer in the Decoder at time step $T+1$ embeds the last preceding target $y^{(T)}_{target}$ into the context of all antecedant targets $y^{(1)}_{target}, \dots, y^{(T)}_{target}$. This vector is used as query in the second Attention layer of the Decoder. Being a Cross-Attention layer, the keys and values here are the outputs $z^{(1)}, \dots, z^{(n)}$ of the Encoder. Its context therefore gets extended by the context given by the encoded input, which in turn is the embedding of the input sequence $x^{(1)}, \dots, x^{(n)}$ into its

own context. The keys and values therefore remain the same for any output creation step of the Decoder and are equivalent to the meaning of the sentence that shall be translated in tasks of language translation.

When testing the Transformer or when applying it in the real world, there will not be any target translations given anymore to be used as input for the Decoder. Therefore, it will use its own predictions, forcing it again into a recurrent structure. Since, however, the computation time is significantly more important during training than during testing or application, the Transformer Architecture holds a speed advantage inherent in its structure over any conventional RNN architecture, while also taking advantage of the attention mechanism to create Attention based contexts throughout its process.

Transformers have been implemented originally with the task of language translation in mind. When it comes to time series however, one advantage of the memory-mimicing properties of LSTMs becomes relevant: the ability to incorporate static, time-invariant meta-information about the instances of a sequence in its initial hidden state $h^{(0)}$ and cell state $c^{(0)}$. In fact, many state-of-the-art models like Google´s Temporal Fusion Transformer [48] or Amazon´s DeepAR [49] consist of a combination of the architecture of a Transformer and integrated LSTMs. For our task we will make use of the advantages of the different presented Network types from this section by using an LSTM to embed the static data of our Jobs and Machines, a Transformer-related architecture to put them into context and an Action-Pointer to output a decision of how to proceed the schedule based on this processed information. Therefore, in the next section we will see how to embed our problem enviroment so that we can apply such a Neural Network to it.

5 Problem Embedding

Job Scheduling Problems on parallel Machines are almost always NP-hard to solve and even approximation algorithms are difficult to find as soon as the objective functions becomes slightly more complex [1, p. 106]. In the real world, several factors often have to be considered to evaluate how good a schedule is. Moreover, it is desirable to have a polyvalent approach that can flexibly be applied to somewhat modified versions of the problem, too, without having to change the entire basic structure of the algorithm everytime some information gets added or substracted from the objective function. To find a solution more suitable to these circumstances, we will make use of the theoretical and mathematical background of Job Scheduling Problems and Deep Reinforcement Learning introduced in the previous sections so far, utilising the presented Network architectures from Natural Language Processing to implement the latter.

The goal is to create a Neural Network that, after being trained on some Job Scheduling Problems, can allocate a value to every possible action of assigning a Job to an available Machine or to turn the same off that correlates with the thereby inflicted scheduling costs. Schedules can then be created efficiently by repeatedly converting the current situation of the Job Scheduling Problem into a readable form for the Neural Network, computing its output and taking the action associated with the estimated optimal value whenver a Machine becomes free.

Another advantage is that when the objective function gets modified or more conditions get added to the Jobs or Machines, one might be able to train a new and suitable Neural Network by creating data according to these new requirements with just very little and intuitive changes to the architecture.

To create such action-values, we make use of the structure of Deep Reinforcement Learning. We therefore will represent any situation in which the need for a decision of which Job to apply to a free Machine emerges as a *state* s, so that we can apply the Bellman-Equation to estimate the Q-values from (7) and with that the corresponding scheduling costs for every action. Hence, every action will be associated with a value estimating the immediate costs of taking said action and transitioning to the associated successor state together with the future costs of continuing the schedule according to some target policy μ from there. This policy is derived from the Target Network by greedily choosing the action which it estimates to have the lowest costs. During the learning process these estimated costs should approximate the Q-values, so that μ converges to the optimal policy μ^*. To apply this approach we therefore need to embed the structure of Job Scheduling Problems

into a Q-learning environment. We will start by explaining how to create the states.

5.1 States and Actions

We recall that we have J Jobs that can be assigned to M Machines. Our conditions of (5) are that every Job j and every Machine m has a deadline d_j or δ_m and a weight w_j or ω_m respectively. All Machines are unrelated, so every Machine m has an individual processing time p_{jm} for every job j and the scheduling does neither allow for preemptiveness nor for idleness, since the latter would always be suboptimal compared to assigning the next Job immediately. Any situation in which a Machine becomes free and therefore needs an assignment will be transformed into a state. If more than one Machines become free at the same time, this leads to a separate state for each of them that get handled one after another. Deciding to not assign any Job to a Machine and to shut it down instead is always a feasible action in a state as long as there is at least one more operating Machine. Since we want to apply Reinforcment Learning, the states need to be markovian, so that the information about any current state does not depend on the previous ones. So in contrast to the Jobs, every Machine but one starts with an initial occupation, whose processing time r_m ranges from zero up to a predefined maximum value. This holds the additional advantage of making sequential states indistinguishable from initial ones and therefore also enabling us to embed smaller scheduling problems into bigger ones, because every state can be interpreted as initial, hence as its own complete scheduling environment, and every initial state could be embedded into the process of foregoing states of a larger scheduling environment. This will also allow us to flexibly address to an even more general set of Job Scheduling Problems.

To define a state, we wish to identify it with a mathematical representation that can be fed into a Neural Network. So for a given Job Scheduling Problem, let s denote an occuring state within a schedule at a moment in which a Machine m_s becomes unoccupied, $t^{(s)}$ the units of time that have passed until this point and $r^{(s)} \in \mathbb{R}^M$ the current remaining runtimes of all the Machines.
Let further $M^{(s)}$ be the number of Machines that have not been turned off yet and $J^{(s)}$ the number of Jobs that have yet to be assigned. Likewise to our comparative list scheduling algorithm 1, these remaining Jobs get sorted by Π_{m_s} as in the sorting algorithm of 2. Moreover, the Machines that are still operarting get ordered as well. They get sorted by the shorter remaining processing time $r_m^{(s)}$, consequently making the first Machine of the list the one that is free and shall get an assignment. If two Machines are

equal regarding this rule, analogous to Π_m they get sorted for their weights in descending order or, if these are equal as well, by the earlier deadline. We define this permutation as the mapping

$$\Pi : \{1, \ldots, M\} \mapsto \{1, \ldots, M\} \tag{33}$$

The subsequent indexing refers to these sorted lists of Jobs and Machines.

For every still operating Machine m we create a vector $s_m^{Machines} \in \mathbb{R}^3$ of dimension 3, consisting of the values of its remaining occupation time $r_m^{(s)}$, its weight ω_m and the remaining time until its deadline, i.e. its *earliness*

$$\epsilon_m^{(s)} := \max(0, \delta_m - t^{(s)})$$

For every remaining Job j we create a vector $s_j^{Jobs} \in \mathbb{R}^{M^{(s)}+2}$ with the processing times p_{jm} on each of the $M^{(s)}$ operating Machines m, as well as its weight w_j and its earliness, expressly the time until its deadline exceeds

$$e_j^{(s)} := \max(0, d_j - t^{(s)})$$

So if in state s we still have $M^{(s)}$ operating Machines and $J^{(s)}$ remaining Jobs, we receive $M^{(s)}$ vectors of dimension 3 for the Machines and $J^{(s)}$ vectors of dimension $M^{(s)} + 2$ for the Jobs. Define the matrices

$$\begin{aligned} [s^{Machines}] &:= [s_1^{Machines}, \ldots, s_{M^{(s)}}^{Machines}]^T && \in \mathbb{R}^{M^{(s)} \times 3} \\ [s^{Jobs}] &:= [s_1^{Jobs}, \ldots, s_{J^{(s)}}^{Jobs}]^T && \in \mathbb{R}^{J^{(s)} \times (M^{(s)}+2)} \end{aligned}$$

Any state s can now mathematically be identified as the 2-tupel

$$s := ([s^{Machines}], [s^{Jobs}])$$

An examplary illustration of how to extract the data from a state is given by the following figure 18:

p_{jm}	**Job 1**	**Job 2**	**Job 3**	**Job 4**	**Job 5**	$r_m^{(s)}$	$\epsilon_m^{(s)}$	ω_m
Machine 1	2	5	5	7	7	0	4	3
Machine 2	6	4	4	2	12	3	0	6
Machine 3	10	1	5	2	8	5	2	1
$e_j^{(s)}$	5	6	0	1	4			
w_j	9	6	3	2	2			

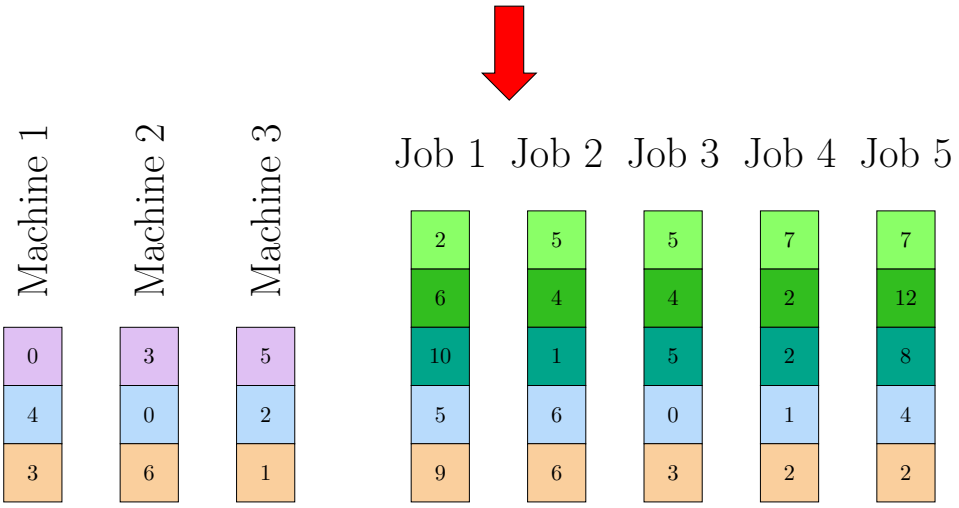


Figure 18: State to Data

If in s a Machine became free but there are no remaining Jobs anymore, this state is considered to be final. When such a final state is reached, all the Machines naturally get turned off as soon as they become free from their last occupation. Since no action can or has to be taken in a final state, we do not need to extract any data from it for our Neural Network. Hence, let in the following all regarded states be non-final.
On the other hand, the environment of the Job Scheduling Problem can be identified with its initial state s_1. Starting in this initial state s_1, there exists a discrete number of possible schedulings. The mathematical set of non-final states S is therefore derived by the corresponding 2-tuples of any moment in which a Machines becomes free and that requires a decision by following any of these schedulings.
The mathematical set of actions is then given by:

$$A := \{1, \dots, J+1\}$$

where action $a \in A$ stands for assigning Job $a = 1, \dots, J$ to the currently free Machine or corresponds to the act of turning it off if $a = J+1$. Let furthermore for any state $s \in S$ the subset $A(s)$ of A consist of all feasible

actions in s. An action $a = 1, \ldots, J$ is feasible in a state s if Job a has not been assigned in any previous state yet. The action $a = J + 1$ is feasible if there is still more than one Machine operating, i.e. if $M^{(s)} > 1$.
By choosing a feasible action $a \in A(s)$ in a state s we therefore transition to a successor state $s' \in S$, representing the next moment a Machine becomes free. Due to its new occupation, the post-decision runtime of Machine m_s in state s is given as

$$r_{m_s}^{(s,a)} := \begin{cases} p_{am_s} & \text{if a = 1,..., J} \in \text{A(s)} \\ 0 & \text{if a = J+1} \in \text{A(s)} \end{cases}$$

being either its processing time for the assigned Job a or 0 if it got shut down. The occupation of all other Machines did not change post-decision, so $r_m^{(s,a)} := r_m^{(s)}$ for $m \neq m_s$. Hence, by combining these values to the vector $r^{(s,a)} \in \mathbb{R}^M$, the time that will pass until the next moment a Machine becomes free (and therefore until s' occurs) is equal to the shortest post-decision occupation time of any of the working Machines:

$$t_\Delta^{(s,a)} := \min_m \{r_m^{(s,a)} \mid \text{Machine } m \text{ still works}\} \tag{34}$$

Since the processing times are deterministic, the successor state s' is as well and therefore given in form of the deterministic output

$$s' := tr(s, a)$$

of the *transition function* $tr : S \times A \mapsto S$. Let a_s be the row of $[s^{Jobs}]$ corresponding to Job a. To update the matrices $[s^{Machines}]$ and $[s^{Jobs}]$ to the 2-tupel that identifies s', we have to update the occupation times $r^{(s')}$ of the Machines as well as the earlinesses $e^{(s')}$ and $\epsilon^{(s')}$ to the time point $t^{(s')} = t^{(s)} + t_\Delta^{(s,a)}$ at which the next Machine becomes free in state s'. We also have to eliminate either the assigend Job a or respectively the Machine m_s in case it got turned off by taking action a in state s. The processing times as well as the weights remain unchanged. Hence, the transition function tr is defined by the following algorithm 3:

Algorithm 3: State Transition Function

Input : state s
action a
Output: successor state s'

for $m = 1, \ldots, M^{(s)}$ **do** // update Machine time values
  $s^{Machines}_{m,r_m} = r^{(s,a)}_m - t^{(s,a)}_\Delta$; // $r^{(s')}_m$
  $s^{Machines}_{m,\epsilon_m} = \max(0, \epsilon^{(s)}_m - t^{(s,a)}_\Delta)$; // $\epsilon^{(s')}_m$
for $j = 1, \ldots, J^{(s)}$ **do** // update Job time values
  $s^{Jobs}_{j,e_j} = \max(0, e^{(s)}_j - t^{(s,a)}_\Delta)$; // $e^{(s')}_j$
if $a = J + 1$ **then**
  delete row 1 of $[s^{Machines}]$; // Machine shut down
else
  delete row a_s of $[s^{Jobs}]$; // or Job assigned
apply Π to $[s^{Machines}]$; // sort Machines by $r^{(s,a)}$
identify $m_{s'}$ with $s^{Machines}_1$; // next free Machine
apply $\Pi_{m_{s'}}$ to $[s^{Jobs}]$; // sort Jobs by $p_{jm_{s'}}$
return $([s^{Machines}], [s^{Jobs}])$; // successor state s'

Note that we do not pass the times $t^{(s)}$ or $t^{(s')}$ as the earlinesses have received the equivalent recursive time-independent definition:

$$e^{(s')}_j = \max(0, e^{(s)}_j - t^{(s,a)}_\Delta)$$
$$\epsilon^{(s')}_m = \max(0, \epsilon^{(s)}_m - t^{(s,a)}_\Delta)$$

with their initial values $e^{(s_1)}_j = d_j$ and $\epsilon^{(s_1)}_m = \delta_m$ being equal to the respective deadlines.

5.2 Policies

To be able to associate a policy μ with a scheduling algorithm, we need to prove that every policy induces a feasible schedule as well as that every scheduling algorithm is representable as a policy for any given Job Scheduling Problem of our kind.

Theorem 3 *Let $\mu : S \mapsto A$ be a policy and all of its actions $\mu(s) \in A(s)$ be feasible for every state $s \in S$. For a Job Scheduling Problem represented by the initial state $s_1 \in S$, define the sequence of states*

$$s_{i+1} := tr(s_i, \mu(s_i)) \tag{35}$$

for $i = 1, \ldots, n-1$ with n being the index of a state s_n with $J^{(s_n)} = 1$ and

$\mu(s_n) = 1, \ldots, J$, *thus of the state where the last remaining Job* $\mu(s_n)$ *gets assigned.*
Let further i_j *denote the unique index of the state* s_{i_j} *in which Job* j *gets assigned on Machine* $m_j := m_{s_{i_j}}$ *for* $j = 1, \ldots, J$. *A feasible schedule is then given by the sequence:*

$$((t^{(s_{i_1})}, m_1), \ldots, (t^{(s_{i_J})}, m_J))$$

Proof. It is

$$\begin{aligned} st_j &= t^{(s_{i_j})} \\ p_{jm_j} &= r_{m_j}^{(s_{i_j}, j)} \end{aligned} \tag{36}$$

due to construction. Since per definition $C_j = st_j + p_{jm_j}$, it directly follows that:

$$C_j = t^{(s_{i_j})} + r_{m_j}^{(s_{i_j}, j)} \tag{37}$$

We can now proof all three feasibility requirements of (2), (3) and (4):

(2): Each Machine m becomes free the first time at r_m. If there are no Jobs remaining, it gets turned off naturally and $\zeta_m = r_m$.
Otherwise it results in a state s_i, $i = 1, \ldots, n$, with $t^{(s_i)} = r_m$.
If $\mu(s_i) = J + 1$, then $\zeta_m = r_m$.
If on the other hand $\mu(s_i) = 1, \ldots, J$, then $s_{i_j} = s_i$ with $m_j = m$ and the tuple $(t^{(s_{i_j})}, m_j)$ is the j-th element of the schedule.

(3): Let $m = 1, \ldots, M$. Then, for every $j_1 \in A_m = \{j \mid m_j = m\}$ there exists a state $s_{i_{j_1}}$, $i_{j_1} = 1, \ldots, n$, at time $t^{(s_{i_{j_1}})}$ with $\mu(s_{i_{j_1}}) = j_1$.
Thus from (37) follows that

$$C_{j_1} = t^{(s_{i_{j_1}})} + r_m^{(s_{i_{j_1}}, j_1)}$$

Hence, C_{j_1} denotes the next time at which an action is taken on Machine m

$$C_{j_1} = \min(\{t^{(s_{i_{j_2}})} \geq t^{(s_{i_{j_1}})} | j_2 \neq j_1 \in A_m\} \cup \{\zeta_m\})$$

and, due to (36), coincides with the earliest time the processing of another remaining Job j_2 could start on it, consequently implying that

$$\nexists j_2 \neq j_1 \in A_m : st_{j_1} \leq st_{j_2} < C_{j_1}$$

(4): Analogously, if there are no Jobs remaining at C_{j_1}, Machine m gets turned off naturally and $\zeta_m = C_{j_1}$.

Otherwise there exists a state s_i, $i = 1, \ldots, n$, with $t^{(s_i)} = C_{j_1}$.
If $\mu(s_i) = J + 1$, then $\zeta_m = C_{j_1}$.
If on the other hand $j_2 := \mu(s_i) = 1, \ldots, J$, then $s_{i_{j_2}} = s_i$ exists and $(t^{(s_{i_{j_2}})}, m)$ is the j_2-th element of the schedule, thus $j_2 \in A_m$ and

$$C_{j_1} = t^{(s_{i_{j_2}})} = st_{j_2}$$

□

Reversely, a scheduling algorithm is a mathematical mapping which receives as input the processing times, deadlines, weights and current occupation times of the remaing Jobs and Machines respectively. It is applied as soon as one Machine becomes free and shall induce a feasible schedule, so if it decides to assign a remaining Job j to Machine m at time t, its output implicitely or explicitely contains the information of the tuple (t, m) as well as j, since that will be its index in the scheduling-sequence. In case that there is at least one additional Machine on duty, it can also decide to turn Machine m off, which has to be unambiguously deducable by its output somehow deviating from the previous form.
Therefore, due to the construction of our Deep Reinforcement Learning Environment, its input can be described as a state $s \in S$ and its outputs by either $j = 1, \ldots, J$ in case of an assignment or by $J + 1$ in case of a Machine shut down.
Hence, every scheduling algorithm can be rewritten as a policy $\mu : S \mapsto A$ mapping exclusively to feasible actions $\mu(s) \in A(s)$ for every state $s \in S$.

5.3 Action Values

We now have to show that the Value function V_μ of a policy μ is equal to the costs of the objective function produced by the associated schedule. The objective function represents the accumulation of the costs over the entire duration of a schedule for a given Job Scheduling Problem. To assign immediate costs $c_s(a)$ to the action $a \in A(s)$ taken at any state $s \in S$, we therefore need to distribute this entirety of scheduling costs onto the state-action-pairs, so that the sum of immediate costs of an action sequence producing states equivalent to a schedule equals (1). Using property (37), we will now construct such a state-action-related implementation of the objective function by looking at the makespan related costs, the Job related costs and the Machine related costs successively.

For this purpose, let in the following for a given Job Scheduling Problem s_1 with J Jobs and M Machines $s_1, \ldots, s_n \in S$ be a sequence of states associated with a feasible schedule $(t^{(s_1)}, m_1), \ldots, t^{(s_J)}, m_J)$ with $m_i := m_{s_i}$. Let them be derived from a policy $\mu : S \mapsto A$ with feasible actions $a_i := \mu(s_i) \in A(s)$,

so $s_{i+1} = tr(s_i, a_i)$ with $a_n = 1, \ldots, J \in A(s_n)$ corresponding to the action of the last Job assignment, meaning that s_n is the last non-final state.

Makespan

We will first split C_{max} into a sum over the states by dividing it into the times that passes inbetween them. However, to do so we have to consider the time until the last Job finishes, not only until the last Job a_n is assigned. In the previous subsections we made the limitation for all states $s \in S$ to be non-final, including any successor state $tr(s, a)$, $a \in A(s)$, since due to the absence of a need to take a decision we do not have to extract data from final states for our Neural Network. Nevertheless, this means that we have to expand the definition of the time difference between two subsequent states (34) by the case that the successor state would be final. As mentioned, we do want to consider the time until the last Job finishes and consequently the last Machine gets shut, down which equals precisely its post-decision runtime. Therefore, we set:

$$t_\Delta^{(s_n,a_n)} := \max_m r_m^{(s_n,a_n)} \tag{38}$$

This leads us to the well-defined state-action-dependent definition of the makespan:

$$C_{max} = \sum_{i=1}^{n} t_\Delta^{(s_i,a_i)}$$

Weighted Tardiness of Jobs

For the part of the weighted tardiness of the Jobs, for every state s_i we look at how much of the time $t_\Delta^{(s_i,a_i)}$ that will pass until the successor state s_{i+1} is an exceedence to the deadline of any Job. Thus, for every remaining Job j in state s_i, we define the state-action-related Job tardiness as

$$T_j^{(s_i,a_i)} := \max(0, t_\Delta^{(s_i,a_i)} - e_j^{(s_i)}) \qquad j \notin \{a_1, \ldots, a_i\}$$

Aditionally, if a Job a_i gets assigned at the current state s_i, we take all the (possibly additional) exceedence to its deadline up to its completion time:

$$T_j^{(s_i,a_i)} := \max(0, p_{jm_{s_i}} - e_j^{(s_i)}) \qquad j = a_i$$

In case that Job j had been assigned at a former state already, all costs related to the exceedence of its deadline have already been considered in former states, hence

$$T_j^{(s_i,a_i)} := 0 \qquad\qquad j \in \{a_1, \ldots, a_{i-1}\}$$

Taking the sum over all state-action-pairs therefore results in

$$T_j = \sum_{i=1}^{n} T_j^{(s_i,a_i)}$$

Consequently, we get

$$\begin{aligned}\sum_{j=1}^{J} w_j T_j &= \sum_{j=1}^{J} w_j \sum_{i=1}^{n} T_j^{(s_i,a_i)} \\ &= \sum_{i=1}^{n} \sum_{j=1}^{J} w_j T_j^{(s_i,a_i)}\end{aligned}$$

Weighted Tardiness of Machines

With regards to the weighted tardiness of the Machines, in every state s_i we consider all the additional deadline exceedence of the currently free Machine m_i at once that emerges by assigning Job a_i to it:

$$\tau_{m_i}^{(s_i,a_i)} := \max(0, p_{a_i m_i} - \epsilon_{m_i}^{(s_i)}) \qquad\qquad a_i \neq J+1$$

If no Job gets assigned in s_i because $a_i = J+1$ denotes the action of turning Machine m_i off, no additional costs arise due to its deadline:

$$\tau_{m_i}^{(s_i,J+1)} := 0$$

Hence, all the other Machines do not produce any additional deadline related costs at this state either:

$$\tau_m^{(s_i,a_i)} := 0 \qquad\qquad m \neq m_i$$

Note that some initial deadline exceedence costs can occur due to the initial machine occupation, hence we need to consider

$$\tau_m^{(0)} := \max(0, r_m - \delta_m)$$

Therefore, we get

$$\tau_m = \tau_m^{(0)} + \sum_{i=1}^{n} \tau_m^{(s_i,a_i)}$$

and consequently the reformulation

$$\begin{aligned}\sum_{m=1}^{M} \omega_m \tau_m &= \sum_{m=1}^{M} \omega_m (\tau_m^{(0)} + \sum_{i=1}^{n} \tau_m^{(s_i,a_i)}) \\ &= \sum_{m=1}^{M} \omega_m \tau_m^{(0)} + \sum_{i=1}^{n} \sum_{m=1}^{M} \omega_m \tau_m^{(s_i,a_i)} \\ &= \tau^{(0)} + \sum_{i=1}^{n} \sum_{m=1}^{M} \omega_m \tau_m^{(s_i,a_i)}\end{aligned}$$

with $\tau^{(0)} := \sum_{m=1}^{M} \tau_m^{(0)}$ being the sum over all costs arising by initial occupations of Machines exceeding their deadlines.

Objective Function

Putting all of the above results together, we can rewrite the objective function of our associated schedule as sum over our sequence of state-action-pairs:

$$C_{max} + \sum_{j=1}^{J} w_j T_j + \sum_{m=1}^{M} \omega_m \tau_m \tag{39}$$

$$= \sum_{i=1}^{n} t_\Delta^{(s_i,a_i)} + \sum_{i=1}^{n} \sum_{j=1}^{J} w_j T_j^{(s_i,a_i)} + \sum_{s=1}^{n} \sum_{m=1}^{M} \omega_m \tau_m^{(s_i,a_i)} + \tau^{(0)} \tag{40}$$

$$= \sum_{i=1}^{n} [t_\Delta^{(s_i,a_i)} + \sum_{j=1}^{J} w_j T_j^{(s_i,a_i)} + \sum_{m=1}^{M} \omega_m \tau_m^{(s_i,a_i)}] + \tau^{(0)} \tag{41}$$

Thus, we can now define and embed immediate transition costs for any state $s \in S$ and feasible action $a \in A(s)$:

$$c_s(a) := t_\Delta^{(s,a)} + \sum_{j=1}^{J} w_j T_j^{(s,a)} + \sum_{m=1}^{M} \omega_m \tau_m^{(s,a)} \tag{42}$$

These costs do not depend on previous states, fulfilling the markovian requirements. Inserting their definition in (41) leads us to the Value function related expression of our objective function:

$$
\begin{aligned}
&C_{max} + \sum_{j=1}^{J} w_j T_j + \sum_{m=1}^{M} \omega_m \tau_m \\
&= \tau^{(0)} + \sum_{i=1}^{n} c_{s_i}(a_i) \\
&= \tau^{(0)} + V_\mu(s_1)
\end{aligned}
$$

Due to the the Machine deadline related initial costs $\tau^{(0)}$, the scheduling costs defined by the objective function (1) for a Job Scheduling Problem s_1 of following a policy μ do actually not equal its Value function $V_\mu(s_1)$. Since, however, $\tau^{(0)}$ is independent of the choosen policy μ, a policy that minimizes $V_\mu(s_1)$ produces an associated feasable schedule that minimzes the objective function of (1). So by setting the Machine deadlines $\delta_m := max(\delta_m, r_m)$ for $m = 1, \ldots, M$, we create a problem environment which is equivalent in its Q-values and therefore in terms of finding the optimal policy μ^* with regards to the scheduling costs. This guarantees that the Value function $V_\mu(s_1)$ of policy μ is equal to the objective function of its associated schedule:

$$
V_\mu(s_1) = C_{max} + \sum_{j=1}^{J} w_j T_j + \sum_{m=1}^{M} \omega_m \tau_m
$$

Moreover, since we constructed the state-action related tardiness of Machines $\tau_{m_s}^{(s,a)}$ depending on the non-negative earlinesses $\epsilon_m^{(s)} \geq 0$ of the Machines instead of their deadlines d_m, every state $s \in S$ can then be interpreted as initial state of its own Job Scheduling Problem with initial costs of zero. Consequently, our environment of Job Scheduling Problems is well-defined within the markovian structure.

5.4 Target Values

Since our states are markovian and therefore any state $s \in S$ can be defined to be initial, we will aliviate the notation for the rest of this section by using J and M instead of $J^{(s)}$ and $M^{(s)}$ and assume that the indices of the (remaining) Jobs and Machines are ordered accordingly to the sorting algorithms Π_{m_s} and Π used in the creation of the state related data in figure 18.

Being able to assign immediate costs $c_s(a)$ to every feasible action $a \in A(s)$ of any state $s \in S$, corresponding to assigning Job $a \neq J + 1$ to the free Machine m_s or to turning it off for $a = J + 1$, we can now calculate the accumulated future costs for any corresponding successor state $s' := tr(s, a)$,

given by the Value function $V_{\mu_\vartheta}(s')$ and arising by following the target policy μ_ϑ. Combining both, the transition and the future costs, we obtain the temporal difference targets from (10):

$$Q(s, a, \vartheta) := c_s(a) + \gamma V_{\mu_\vartheta}(s') \tag{43}$$

as an estimation of the Q-values of (7) which we want to incorporate into our *target values* $y_\vartheta^{(s)} \in \mathbb{R}^{J+1}$, so that our Neural Network can use them to learn how to deduct an optimal policy μ^* leading to the optimal schedule to any given Job Scheduling Problem. Since future costs are equally as important as immediate costs, we set $\gamma := 1$.

A simple approach to this would be to treat every state s as a multiclass classification problem, where the classes are associated with the feasible actions. The target vector would then be a one-hot-vector of dimension $J + 1$ with an entry for every possible Job assignment plus one for the option of deactivating the Machine in case that $M > 1$. The index of the single 1-value would then indicate the optimal action to take in that state. An intuitive loss function for that approach would therefore be the cross-entropy. However, we might loose some information by this implementation: even if not optimal, some actions can be better than others by producing smaller costs.
Thus, we decide to apply a regression instead and choose the mean-squared-error to be our loss function and the target vector to be the temporal difference target of (10), whose entries $y_\vartheta^{(s)}(a)$ consist of the estimations of the Q-values $Q(s, a, \vartheta)$ associated with every possible action $a = 1, \ldots, J + 1$. These are composed of the known immediate transition costs $c_s(a)$ and the future costs $V_{\mu_\vartheta}(s')$ estimating $V^*(s')$ from(6). Here, the target policy μ_ϑ is greedily derived from the estimations of the Q-values as in (43) by the Target Network with parameters ϑ:

$$\mu_\vartheta(s') = \arg \min_{a' \in A(s')} Q(s', a', \vartheta) \tag{44}$$

As we will see in section 7, due to our computationally advantageous construction it will actually be feasible to calculate the error for each action $a \in A(s)$ instead of having a probabilty distribution over them as behaviour policy π like in (9). This gives us the loss function

$$L_\vartheta(\Theta) := \sum_{a=1}^{J+1} [(y_\vartheta^{(s)}(a) - Q(s, a, \Theta))^2]$$

for a given state $s \in S$, where $Q(s, a, \Theta)$ is the estimation of the Q-values from our Q-Network with parameters Θ.

5.5 Normalization

To make the learning process faster and more stable, we have to normalize our data. We start by accordingly redefining the target values $y_\vartheta^{(s)}(a)$, which we have set to be the estimation of the Q-values from (7) by the Target Network given ϑ as its parameters.

Targets

One possible way to normalize these values would be to divide each of them by the highest occuring target entry

$$Q_\vartheta^{max} := \max_{a \in A(s)} Q(s, a, \vartheta)$$

associated with the action estimated to be most costly. This, however, would put an emphasis on this supposedly worst action, since every other action value would be scaled in its regard. Let $a_\vartheta \in A(s)$ be the index associated with the action predicted to be optimal by the target policy μ_ϑ:

$$Q_\vartheta^{min} := Q(s, a_\vartheta, \vartheta) = \min_{a \in A(s)} Q(s, a, \vartheta)$$

If then, for instance, Q_ϑ^{max} is 10 times as high as Q_ϑ^{min}, the Neural Network has to predict 0.1 as target value for entry a_ϑ. If, however, Q_ϑ^{max} is 100 times as costly, the Q-Network would need to predict 0.01. Hence, it would be necessary for it to precisely learn how bad the worst action is to correctly compare all the others to it.
Furthermore, if the second best action is twice as costly as the optimal one, its target value would be 0.02. Therefore, misspredicting the second best value $\min_{a \neq a_\vartheta} Q(s, a, \vartheta)$ to be the optimal one or vice versa would not have a big contribution to the regression error given by the loss function, leading the Q-Network to neglect to learn how to differentiate them precisely. If there is another bad action estimated to produce costs 80 times as high as the optimal action would, the Q-Network should predict 0.8. The difference between the normalized target values associated with these suboptimal actions, i.e. 0.8 to 1, would therefore contribute much more strongly, consequently forcing the Network to mainly focus on distinguishing how bad the worst actions are when compared to each other.
We, on the other hand, are more interested in gathering knowledge about the best options and do not really care to know which of the bad actions is worse by how much, knowing that they are far from optimal is enough for us. Predefining a maximum cost beforehand as a scaling factor for the target vectors of all states would yield the same problem, together with the additional limitation that we would somehow need to know that no action in any state is ever going to surpass said value.

To solve this dilemma, we decide to instead scale each of the costs by Q_ϑ^{min}, the supposedly smallest Q-value corresponding to the action a_ϑ, estimated to be optimal by the Target Network, and then inverting each of these resulting values:

$$\frac{Q(s, a_\vartheta, \vartheta)}{Q(s, a, \vartheta)} \in [0, 1]$$

To put even further emphasis on the comparision between good actions, we apply a $softmax$ to the normalized target values in the end. Hence, the normalized target value for taking action a in state s ranges between 0 and 1 and is given by:

$$y_\vartheta^{(s)}(a) := softmax(\frac{Q(s, a_\vartheta, \vartheta)}{Q(s, a, \vartheta)}) \in [0, 1] \tag{45}$$

This way, the Neural Network is specifically incentivised to learn how to correctly distinguish between the actions estimated to be the best, while neglecting bad actions the more the worse they are. The following figure 19 sums this process up:

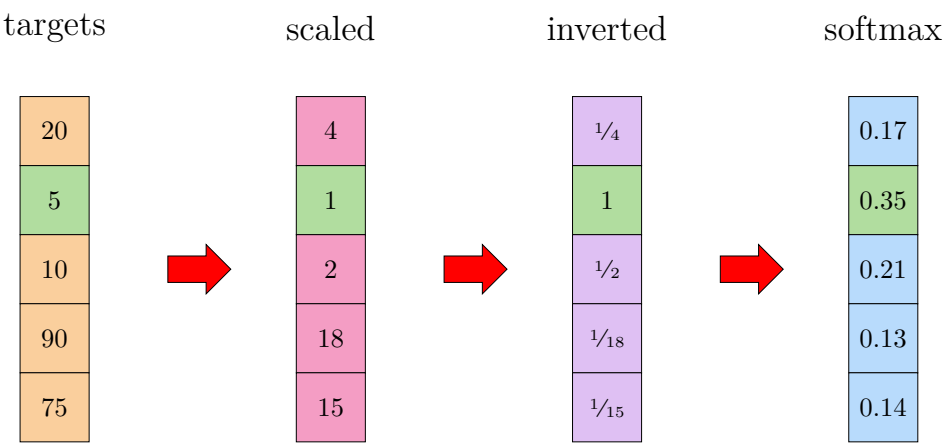


Figure 19: Normalization of Target Values

Note that the action a_ϑ corresponding to the Q-value estimated to be the smallest now indexes the greatest entry in the normalized target vector $y_\vartheta^{(s)}$ produced by our Target Network. Hence, after normalization we equivalently redefine our target policy μ_ϑ from (44) to:

$$
\begin{aligned}
\mu_\vartheta(s) &:= \arg\max_{a \in A(s)} y_\vartheta^{(s)}(a) \\
&= \arg\max_{a \in A(s)} softmax(\frac{Q(s, a_\vartheta, \vartheta)}{Q_(s, a, \vartheta)}) \\
&= \arg\max_{a \in A(s)} \frac{Q(s, a_\vartheta, \vartheta)}{Q_(s, a, \vartheta)} \\
&= a_\vartheta
\end{aligned}
$$

Inputs

For the input, we will need to scale two types of features: the weights as well as the time-related values. For the weights, we define a maximum weight w_{max} by which we divide all Job and Machine weights w_j and ω_m to scale them to a value between zero and one. For the time-related values, we will apply a dynamical scaling, so for every state $s \in S$ we will divide all processing times p_{jm}, earlinesses $e_j^{(s)}$, $\epsilon^{(s)}$ and machine occupations $r_m^{(s)}$ by the maximum time occuring in a remaining Job or an operating Machine in that state. So the factor by which we scale is given by:

$$
t_{max}^{(s)} = \max_{j,m} \max(p_{jm}, e_j, \epsilon_m, r_m^{(s)})
$$

The following figure 20 demonstrates how to normalize the values used for the input of figure 18 with a maximum weight set to $w_{max} = 10$ and a deduced maximum time of $t_{max}^{(s)} = 12$:

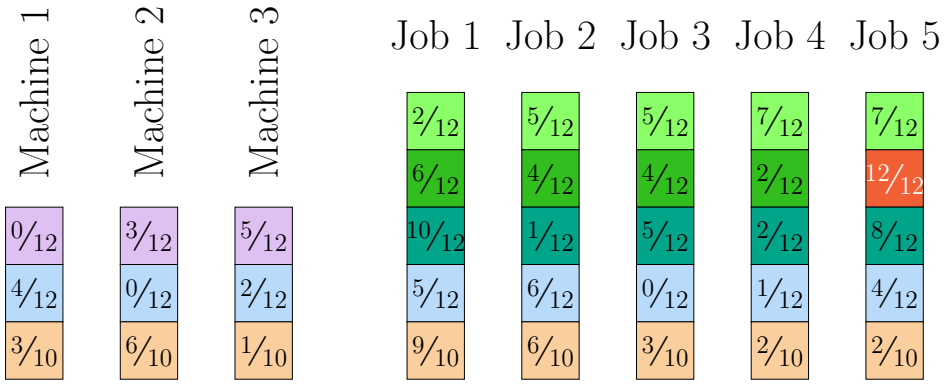


Figure 20: Normalization of Data

While for the static scaling of the weights w_j and ω_m the Q-Network learns the relation of the scaling through the data it receives since it remains the same for all instances, the dynamic scaling of the times changes for every

state. We therefore have to check now whether the Q-Network receives enough information to theoretically be able to predict the normalized target values for any given state. These correlate with the estimated Q-values which are equivalent to the policy costs of taking the respective feasible action and subsequently following the target policy from the successor state on.

So let $s_1, \ldots, s_n$ be the sequence of states for following an arbitrary feasible policy μ from some state $s_1 \in S$. Then, due to the definition of the transition costs of (42), the policy costs are given by

$$V_\mu(s_1) = \sum_{i=1}^{n} [t_\Delta^{(s_i,a_i)} + \sum_{j=1}^{J} w_j T_j^{(s_i,a_i)} + \sum_{m=1}^{M} \omega_m \tau_m^{(s_i,a_i)}]$$

Let now $\varphi^{(s_1)}$ be the mapping that divides every time-related value of the matrices of any state $s = ([s^{Machines}], [s^{Jobs}]) \in S$ by the factor $t_{max}^{(s_1)}$, resulting in $\hat{s} := \varphi^{(s_1)}(s)$. Since

$$r_m^{(\hat{s},a)} = \frac{r_m^{(s,a)}}{t_{max}^{(s_1)}}$$

it directly follows that

$$t_\Delta^{(\hat{s},a)} = \frac{t_\Delta^{(s,a)}}{t_{max}^{(s_1)}}$$

Due to the definition of the transition function tr in algorithm 3, this implies that

$$tr(\varphi^{(s_1)}(s), a) = \varphi^{(s_1)}(tr(s, a))$$

Consequently, for the sequence $\hat{s}_1, \ldots, \hat{s}_n$ of scaled states $\hat{s}_i := \varphi^{(s_1)}(s_i)$, it holds true that $\hat{s}_{i+1} := tr(\hat{s}_i, a_i)$ for every $i = 1, \ldots, n-1$. Therefore, by extending our policy with $\mu(\hat{s}_i) := \mu(s_i)$, we obtain the policy costs:

$$
\begin{aligned}
V_\mu(\hat{s}_1) &= \sum_{i=1}^{n}[t_\Delta^{(\hat{s}_i,a_i)} + \sum_{j=1}^{J} w_j T_j^{(\hat{s}_i,a_i)} + \sum_{m=1}^{M} \omega_m \tau_m^{(\hat{s}_i,a_i)}] \\
&= \sum_{i=1}^{n}[\frac{t_\Delta^{(s_i,a_i)}}{t_{max}^{(s_1)}} + \sum_{j=1}^{J} w_j \frac{T_j^{(s_i,a_i)}}{t_{max}^{(s_1)}} + \sum_{m=1}^{M} \omega_m \frac{\tau_m^{(s_i,a_i)}}{t_{max}^{(s_1)}}] \\
&= \frac{1}{t_{max}^{(s_1)}} \sum_{i=1}^{n}[t_\Delta^{(s_i,a_i)} + \sum_{j=1}^{J} w_j T_j^{(s_i,a_i)} + \sum_{m=1}^{M} \omega_m \tau_m^{(s_i,a_i)}] \\
&= \frac{1}{t_{max}^{(s_1)}} V_\mu(s_1)
\end{aligned}
$$

Since μ and $s_1 \in S$ were arbitrary, our Q-Network might therefore be able to predict

$$\frac{1}{t_{max}^{(s)}} Q(s,a,\vartheta) = Q(\hat{s},a,\vartheta)$$

for any state $s \in S$ with $\hat{s} := \varphi^{(s)}(s)$ and feasible action $a \in A(s) = A(\hat{s})$, but because there is no way for it to deduct $t_{max}^{(s)}$, neither from the data nor from the normalized input, it cannot learn the pre-normalized targets $Q(s,a,\vartheta)$. However, by defining its output $y_\Theta^{(s)}$ to predict the quotient of the costs $Q(\hat{s},a_\vartheta,\vartheta)$ and $Q(\hat{s},a,\vartheta)$ with an applied $softmax$ instead, it can estimate the normalized target values $y_\vartheta^{(s)}$ of (45):

$$
\begin{aligned}
& softmax(\frac{Q(\hat{s},a_\vartheta,a)}{Q(\hat{s},a,\vartheta)}) \\
=& softmax(\frac{\frac{1}{t_{max}} * Q(s,a_\vartheta,\vartheta)}{\frac{1}{t_{max}} * Q(s,a,\vartheta)}) \\
=& softmax(\frac{Q(s,a_\vartheta,\vartheta)}{Q(s,a,\vartheta)}) \\
=& y_\vartheta^{(s)}
\end{aligned}
\tag{46}
$$

Hence, normalizing the input together with the targets as constructed does not inherently inhibit the learning ability of the Q-Network. Therefore, the output $y_\Theta^{(s)}$ from (46) of our Q-Network serves as an estimation for

$$softmax(\frac{Q^*(s,\mu^*(s))}{Q^*(s,a)})$$

consequently enabling it to estimate the optimal action for any state $s \in S$:

$$
\begin{aligned}
&\arg \max_{a \in A(s)} softmax(\frac{Q^*(s, \mu^*(s))}{Q^*(s, a)}) \\
=&\arg \max_{a \in A(s)} \frac{Q^*(s, \mu^*(s))}{Q^*(s, a)} \\
=&\arg \min_{a \in A(s)} Q^*(s, a) \\
=&\mu^*(s)
\end{aligned}
$$

By doing so, we can then use these outputs $y_\Theta^{(s)}$ to induce the policy

$$\mu_\Theta(s) = \arg \max_{a \in A(s)} y_\Theta^{(s)}(a) \tag{47}$$

as our estimation of the optimal policy μ^*.

6 Network Architecture

Based on the architectures introduced in the section 4 about Natural Language Processing, we will now present our Neural Network whose structure is meant to counter the challenges arising by embedding Job Scheduling Problems into data for a Deep Reinforcement Learning environment. So the input will be a state, while the output will induce an action of which Job to assign to the currently free Machine or to turn the same off. The following is an overview of our model.

6.1 Overview

Our Neural Network receives the data representing a state $s \in S$ as input, consisting of the information about the $J^{(s)}$ Jobs and $M^{(s)}$ Machines which are ordered according to the permutation Π_{m_s} from the sorting algorithm 2 and Π from (33) respectively as explained in the previous section.

Its architecture is built of 3 successive parts, of which the first two heavily take advantage of parallelization:

1. Sequential Embedding
2. Transformer Encoder
3. Action-Pointer Decoder

The output will then be a vector $y \in \mathbb{R}^{J^{(s)}+1}$ corresponding to a probability distribution over the $J^{(s)}+1$ feasible actions with $y_{\Pi_{m_s}(j)}$ denoting the action of assigning Job j to Machine m_s for $\Pi_{m_s}(j) = 1, \dots, J^{(s)}$ and $y_{J^{(s)}+1}$ standing for the action of shutting the free Machine m_s of state s down.

In total, our Neural Network consists of 117.292 parameters, all trainable. The next subsections will disclose how they are distributed among the different parts of the model. In these, we will go through its architecture in detail and step by step, starting with how to preprocess the data of our states to bring them into the desired form. Since every state can be interpreted as initial state of its own Job Scheduling Problem, we will aliviate the notation throughout this section by assuming that $J^{(s)} = J$, $M^{(s)} = M$ and that the Jobs and Machines are already listed in the desired order, hence $j = \Pi_{m_s}(j)$ for $j = 1, \dots, J$ and $m = \Pi(m)$ for $m = 1, \dots, M$.

6.2 Preprocessing

We recall that the information about a state s is derived as in figure 18 and normalized as explained in the last section and illustrated in figure 20.

Thus, it is given in form of M vectors $s_1^{Machines}, \dots, s_M^{Machines} \in \mathbb{R}^3$ for the Machines, each consisting of their remaining occupation time $r_m^{(s)}$, their earliness $\epsilon_m^{(s)}$ and their weight ω_m, and J vectors $s_1^{Jobs}, \dots, s_J^{Jobs} \in \mathbb{R}^{M+2}$ for the Jobs, each consisting of 2 entries for their earliness $e_j^{(s)}$ and their weight w_j and the remaining M for the processing times p_{jm} on the respective Machine m.

We want to embed our data into a vector space from which we obtain one vector representation of fixed dimension for each Job. The state environment, i.e. the information about the Machines, should be geometrically integrated within this vector space. However, the number of Machines M and Jobs J varies for every state. So not only the amount of input vectors depends on the state, but also the dimension of these vectors for the Jobs. We will therefore use the architectures introduced in the section 4 about Natural Language Processing that are capable to process inputs of dynamic size to process our input as well.

First, we handle the variability of Machine numbers M. Split the information about job earliness $e_j^{(s)}$ and weight w_j from every vector s_j^{Jobs} of figure 18 apart from its processing times $p_{j1,}, \dots, p_{jM}$. Then incorporate the meta-data of the Machines by concatening the vector $s_m^{Machines}$ of each Machine m to the respective processing time p_{jm}, $m = 1, \dots, M$. Thus, the information about any Job j gets split into two parts:

- A vector $x_{urg}^{(j)} := [e_j^{(s)}, w_j]^T \in \mathbb{R}^2$, resembling the urgency of processing the respective Job j.
- A matrix $X_{res}^{(j)} := [x_1^{(j)}, \dots, x_M^{(j)}]^T \in \mathbb{R}^{M \times 4}$, consisting of the sequence of vectors $x_m^{(j)} := [p_{jm}, r_m^{(s)}, \epsilon_m^{(s)}, \omega_m]^T \in \mathbb{R}^4$ for $m = 1, \dots, M$. This sequence represents a time-series of resources in form of Machines to process said Job j that successively become available.

The following figure 21 demonstrates this process on an exemplary Job:

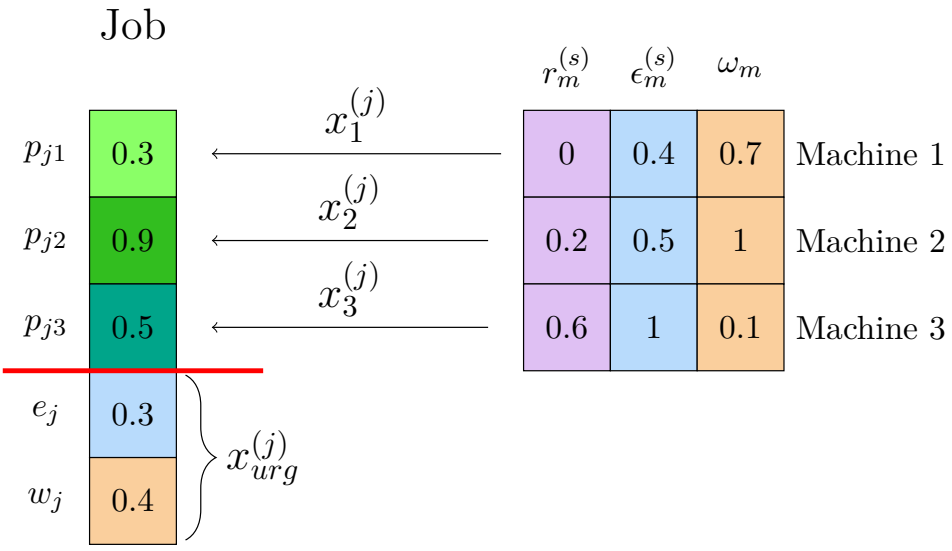


Figure 21: Data to Input

6.3 Input

By preprocessing every Job $j = 1, \ldots, J$ and Machine $m = 1, \ldots, M$ as explained above, the input for the Neural Network will consist of two elements of variable dimensions:

- A matrix $[x_{urg}^{(1)}, \ldots, x_{urg}^{(J)}]^T \in \mathbb{R}^{J \times 2}$ whose rows denote the sequence of vectors of urgencies of the Jobs.
- A cube $[X_{res}^{(1)}, \ldots, X_{res}^{(J)}]^T \in \mathbb{R}^{J \times M \times 4}$ standing for the sequence of resource related matrices of the Jobs. Each of these matrices has all the static Machine data immanent, defining the context of resources.

6.4 Sequential Embedding

The dimension $(J \times M \times 4)$ of the second element of our input is variable in two sizes, J and M. As anounced, we now want to embed our input so that its dimension does not depend on the variable M of number of Machines anymore. Therefore, we need to embed the resource conditions $X_{res}^{(j)} \in \mathbb{R}^{M \times 4}$ of any Job j into a fixed size vector. We do so by interpreting the corresponding sequence $x_1^{(j)}, \ldots, x_M^{(j)}$ as a time series, feeding each of these rows, illustrated in figure 21, to an LSTM of dimension 16. By doing this bidirectionally as in figure 9, we obtain an embedding $x_{res}^{(j)} \in \mathbb{R}^{32}$ of fixed dimension 32 for the resource environment of Job j. The sequential embedding process is demonstrated in the following figure 22:

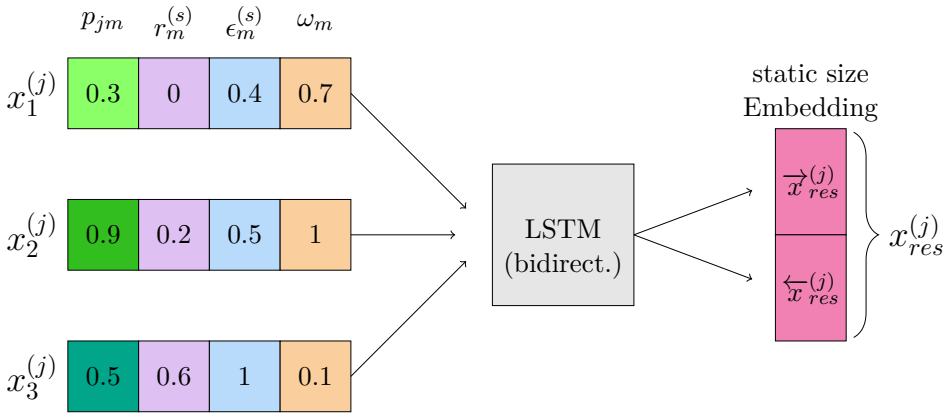


Figure 22: Sequential Embedding of a Job's Machine Resource Environment

The vector $x_{res}^{(j)}$ is then further processed by passing it through a Feed Forward layer FF_{res} as in (31) with the inner activation function ϕ_F being *Leaky ReLU* with constant gradient $\alpha = 0.1$, as it will be for every Feed Forward layer in our Neural Network, and both dimensions set to 16, hence producing:

$$\tilde{x}_{res}^{(j)} = FF_{res}(x_{res}^{(j)}) \in \mathbb{R}^{16}$$

We repeat this process for every Job $j = 1, \dots, J$ separately, so that we receive the sequence $\tilde{x}_{res}^{(1)}, \dots, \tilde{x}_{res}^{(J)} \in \mathbb{R}^{16}$. Alternatively, one could also feed the M Machine vectors to an RNN and use its output to initialize the cell and hidden state of this LSTM whose inputs would then consist only of the processing times of any Job.

The urgency of a Job is not defined only by its own deadline and weight, but always depends on the deadlines and weights of all the other Jobs as well. Therefore, we also wish to embed every $x_{urg}^{(j)}$, $j = 1, \dots, J$, by finding a vector representation of its relative urgency considering the entire sequence $x_{urg}^{(1)}, \dots, x_{urg}^{(J)}$.
First, for every Job j we offset its deadline and weight by passing $x_{urg}^{(j)}$ through a Feed Forward layer $FF_{urg}^{(1)}$ with dimensions equal to 4. To then give a sense of how important it would be to terminate that Job compared to the other ones, we need to know how much Attention we should pay to any of them. Therefore, we subsequently apply a Self-Attention layer with 2 heads and a key-dimension of 8. Afterwards, we run them each through another Feed Forward layer $FF_{urg}^{(2)}$ of dimensions 4, resulting in the sequence $\tilde{x}_{urg}^{(1)}, \dots, \tilde{x}_{urg}^{(J)} \in \mathbb{R}^4$ of relative Job urgencies as shown in the figure 23:

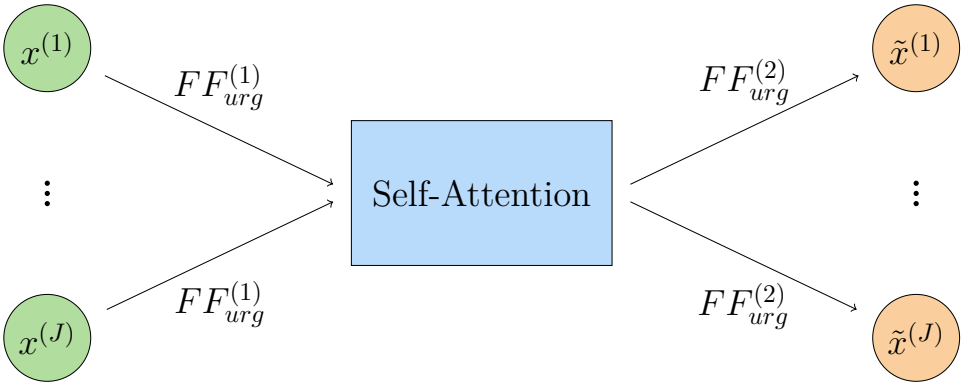


Figure 23: Embed Urgencies of Jobs

Now, to merge both of the embedded data information of every Job $j = 1, \ldots, J$, we concatenate $\tilde{x}^{(j)}_{res} \in \mathbb{R}^{16}$ and $\tilde{x}^{(j)}_{urg} \in \mathbb{R}^4$ to

$$\tilde{x}^{(j)} = concat(\tilde{x}^{(j)}_{res}, \tilde{x}^{(j)}_{urg}) \in \mathbb{R}^{20}$$

and then apply a Feed Forward layer FF_{emb} whose dimensions are both equal to 16:

$$x^{(j)}_{emb} = FF_{emb}(\tilde{x}^{(j)}) \in \mathbb{R}^{16}$$

The resulting sequence defines the embedded input

$$X_{emb} := [x^{(1)}_{emb}, \ldots, x^{(J)}_{emb}]^T \in \mathbb{R}^{J \times 16}$$

We have successfully eliminated the variable M from its dimension by embedding every Job j into a vector space. Its elements intrinsically represent the given set of Machines together with the Job's processing times $p_{j1}, \ldots, p_{jM}$ on them and the Jobs's contextual urgency relative to the other given Jobs. The following table 1 resumes the layers involved in this embedding process:

Layer	Parameters (total = 4476)	Output Dimension
Embedding Resources Bidirectional $LSTM$	2688	32 (2×16)
FF_{res}	800	16
$FF_{urg}^{(1)}$	32	4
Embedding Urgencies $Self\text{-}Attention$	308	4
$FF_{urg}^{(2)}$	40	4
Concatenate Inputs	0	20
FF_{emb}	608	16

Table 1: Embedding Layers

Note that each of these layers is run through by each of the J different Jobs in parallel, permitting for a significantly reduced computation time. Moreover, embedding inputs with an LSTM before processing them with a Transformer Encoder has shown itself benefical with regards to permance in some applications as in [50].

6.5 Transformer Encoder

The sequence $x_{emb}^{(1)}, \ldots, x_{emb}^{(J)}$ of embedded vector representations is now run through a Transformer Encoder as in figure 16, successively consisting of:

1. A Self-Attention layer with 4 heads and key dimension 32
2. A Residual Summation
3. A Feed Forward layer of dimensions 16
4. Another Residual Summation

We do, however, without a layer related normalization, since it will not prove itself necessary.

Every Job has already been embedded into the context of how important finishing it is compared to the other ones as well as into the given set of Machines and how long each of them would need to process it. However, its vector representation $x_{emb}^{(s)}$ contains no information about how long they would need to process any of these other Jobs. This information is added in the Self-Attention layer of the Encoder. The resulting output sequence of the Encoder

$$z^{(1)}, \ldots, z^{(J)} \in \mathbb{R}^{16}$$

is therefore an encoding of the corresponding vector representation $x_{emb}^{(j)}$ of each Job j into the context defined by the entire set of J embedded Jobs. Hence, $z^{(j)}$ is the encoding of Job j into the complete environment of the given state of our Job Scheduling Problem. The layers of the Encoder are summarized in the following table 2:

Layer	Parameters (total = 9136)	Output Dimension
Self-Attention Encoder	8592	16
Residual Connection 1	0	16
FF_{enc}	544	16
Residual Connection 2	0	16

Table 2: Encoder Layers

Again, each of these layers processes all of the J different Jobs in parallel. Next, we need to make a decision about which of them to assign or whether we should shut down the currently free Machine.

6.6 Action-Pointer Decoder

For the Decoder, we do not use a Transformer related architecture, since we only want one single output - a distribution over the feasible actions - annulating many of its otherwise immanent advantages. Moreover, the technique of Teacher Forcing for Deep Reinforcement Learning tasks has not been explored by the literature as much as for supervised tasks yet, with some papers discouraging its unmodified use as in [51].
Moreover, the desired shape of the output confronts us with another dynamic dimension problem as we aim to have precisely one entry for each of the J remaining Jobs, along with one for the option of deactivating the Machine. As a result, the dimensionality of the output requires to be variable, too. We therefore will use a Decoder based on an Action-Pointer, analogous to the one of figure 15.

The encoded sequence $z^{(1)}, \dots, z^{(J)}$ gets fed into a bidirectionally wrapped LSTM of dimension 2×32, which will be our bottom LSTM, producing the hidden states

$$h_B^{(1)}, \dots, h_B^{(J)} \in \mathbb{R}^{64}$$

These now do not represent the Jobs anymore but the corresponding action of assigning each of them to the currently free Machine. Nevertheless, we cannot point to them yet, since we need a further hidden state representing the action of turning the Machine off. An indication for this depends on

the entirety of all remaining Jobs, so to create such a vector with sufficient information, it has to be the result of the processing of the entire encoded sequence $z^{(1)}, \ldots, z^{(J)}$. This holds true for the final hidden state $h_B^{(J)}$, which we therefore pass through a Feed Forward layer FF_{shut} of dimensions 64. The purpose of this layer is to deduct a meaningful representation

$$h_B^{(J+1)} \in \mathbb{R}^{64}$$

for the action of deactivating the Machine, considering the whole context of the given state. An easier, yet less insightful alternative would be to append a zero-vector to the output sequence of the Encoder.

The final hidden state $h_B^{(J)} \in \mathbb{R}^{64}$ and the final cell state $c_B^{(J)} \in \mathbb{R}^{64}$ of our bottom LSTM consist each of the final forward and backward hidden state $\overrightarrow{h}_B^{(J)}, \overleftarrow{h}_B^{(J)} \in \mathbb{R}^{32}$ and $\overrightarrow{c}_B^{(J)}, \overleftarrow{c}_B^{(J)} \in \mathbb{R}^{32}$ respectively. Each of these 4 vectors is run through its own Feed Forward Layer of dimensions 32, resulting in the vectors of processed memory states:

- $\overrightarrow{h}_T^{(0)} = FF_{\overrightarrow{h}}(\overrightarrow{h}_B^{(J)}) \in \mathbb{R}^{32}$
- $\overleftarrow{h}_T^{(0)} = FF_{\overleftarrow{h}}(\overleftarrow{h}_B^{(J)}) \in \mathbb{R}^{32}$
- $\overrightarrow{c}_T^{(0)} = FF_{\overrightarrow{c}}(\overrightarrow{c}_B^{(J)}) \in \mathbb{R}^{32}$
- $\overleftarrow{c}_T^{(0)} = FF_{\overleftarrow{c}}(\overleftarrow{c}_B^{(J)}) \in \mathbb{R}^{32}$

Consequently, by using these processed memory states to initialize its forward and backward hidden and cell states, we stack a second bidirectional LSTM of dimension 2×32 onto it, being our top LSTM similiar to figure 10. The key difference is that its input will be the sequence of hidden states of the bottom LSTM expanded by $h_B^{(J+1)}$. This LSTM is our Action-Pointer, so its input sequence $h_B^{(1)}, \ldots, h_B^{(J)}, h_B^{(J+1)} \in \mathbb{R}^{64}$ will serve as keys. Its final hidden state $h_T^{(J+1)} \in \mathbb{R}^{64}$ gets passed through a final Feed Forward layer FF_q of dimensions 128 to create the query vector

$$q = FF_q(h_T^{(J+1)}) \in \mathbb{R}^{128}$$

The final output y of our Neural Network is then defined as

$$y = \hat{a} \in [0, 1]^{J+1}$$

and therefore equivalent to the attention distribution over the feasible $J + 1$ actions that we compute using the Attention mechanism of Pointer Networks from (26) and (26). Figure 24 resumes the architecture of the Decoder:

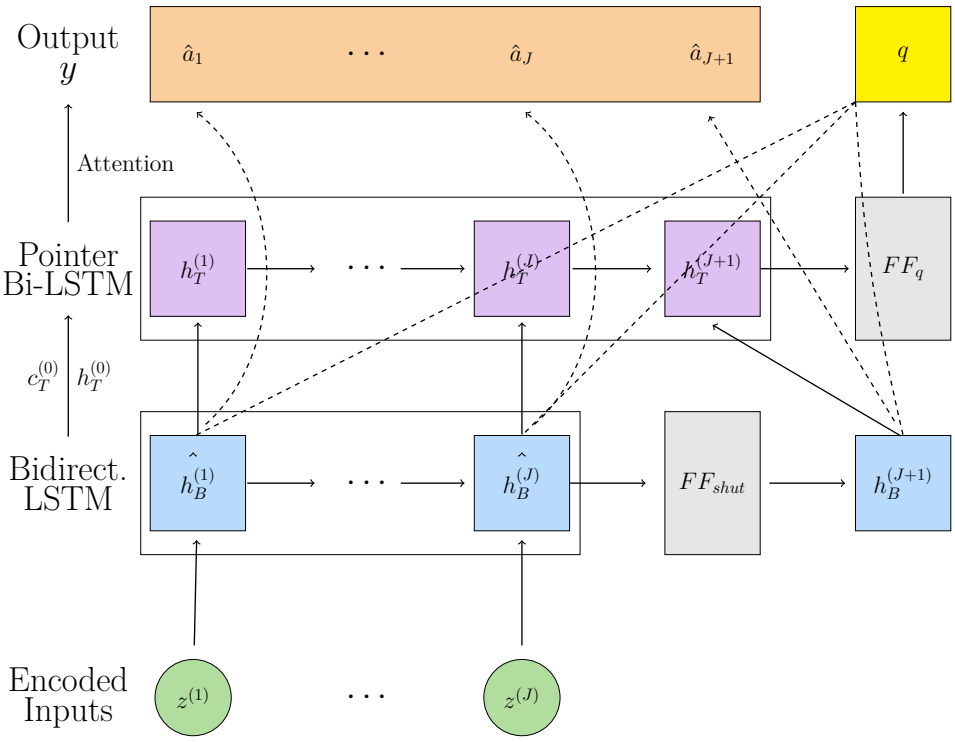


Figure 24: Action-Pointer Decoder

All layers of the Decoder are listed in the following table 3:

Layer	Parameters (total = 103680)	Output Dimension
Decoder Bottom Bidirectional $LSTM$	12544	$J \times 64(2 \times 32)$
FF_{shut}	8320	64
Concatenate $h_B^{(J+1)}$ to Sequence	0	$J + 1 \times 64$
$FF_{\overrightarrow{h}}$	2112	32
$FF_{\overleftarrow{h}}$	2112	32
$FF_{\overrightarrow{c}}$	2112	32
$FF_{\overleftarrow{c}}$	2112	32
Action-Pointer: Decoder Top Bidirectional $LSTM$	24832	$J + 1 \times 64(2 \times 32)$
FF_q	24832	128
Action-Pointer: *Cross-Attention*	24704	$J + 1$

Table 3: Decoder Layers

The Decoder consists of the most parameters by far. This is due to the fact that the Embedding part of the model as well as its Encoder processes each Job in parallel, applying the same parameters to each of them. The Decoder on the other hand combines the encoded vector representations of the Jobs, processing them sequentially in the bottom LSTM and the top LSTM.
Moreover, the inter-contextual information that the vector representation of each feasible action has to have immanent is successively augmented throughout the Neural Network, creating the need for higher dimensions in later layers, therefore increasing their required amount of parameters. So while the attention scores $\hat{a}_j$, $j = 1, \ldots, J+1$, of the output y are computed in parallel by the Action-Pointer, it still significantly contributes to the total number of parameters due to it being the highest layer.

7 Data Creation

Having implemented our Neural Network, we now want to train it to solve Job Scheduling Problems and subsequently test and evaluate its performance. To do so, we need data. The learning process will be separated into two stages. The first stage will be supervised, where we will create the data by computing the ground-true Q-values to every state and use them as targets after normalizing them accordingly to figure 19. In the second stage, we will then make use of the techniques of Deep Reinforcement Learning by using our supervisedly pre-trained Neural Network as Target Network to estimate the Q-values. These estimations will be the target values to create training data for scheduling problems with higher Job numbers. We then initialize a Q-Network with the parameters of the Target Network and train it on the augmented data set. Subsequently, the Target Network gets updated and we iteratively repeat this process until we reached the desired number of Jobs.

7.1 Supervised Job Scheduling Problems

To train our Neural Network, we first need to simulate Job Scheduling Problems as learning environments. For this, we have to define the environmental parameters of how many Jobs and Machines a problem shall consist of, in what value range their deadlines and weights as well as the processing times should be and what the longest initial occupation time for any Machine can be.
In this supervised learning process, the ground-true Q-values for every Job Scheduling Problem will be calculated by creating the entire state space S. Hence, if s_1 is the initial state of a Job Scheduling Problem, we compute the set of all successor states

$$\{tr(s_1, a) \mid a \in A(s)\}$$

For every element of this set, we compute the set of all its successors states again. We successively repeat this process for every non-final state that we created in the previous step. The conjunction of all these sets is then the entire state space S. Therefore, in this approach we compute all possible schedulings. Due to the structure of this approach being equivalent to a directed tree, we can identify every state as a node with the leaves corresponding to the final states. The feasible action space of these final states is the empty set, so by setting

$$\min_{a' \in \emptyset} Q(s', a') := 0 \tag{48}$$

and backtracking from their parents, we can recursively calculate

$$Q(s,a) := c_s(a) + \min_{a' \in A(s)} Q(tr(s,a), a') \tag{49}$$

for all states $s \in S$ in runtime $O(J*M)$. Since for the creation of this master thesis only very limited computational power is availalbe, we have to decide for a relatively small number of Jobs and Machines. By having access to better resources, this number could be increased significantly.
We decide for the following maximal values:

- **Jobs**: $J = 8$
- **Machines**: $M = 4$
- **weight**: $w_{max} = 10$
- **deadline**: $d_{max} = 30$
- **processing time**: $p_{max} \in [\frac{1}{3}d_{max}, \frac{3}{2}d_{max}]$
- **initial machine runtime**: $r_{max}^{(init)} \in [\frac{1}{3}p_{max}, p_{max}]$

We decided for a maximal deadline d_{max} of 30 to resemble an entire month represented by its days. For every Job Scheduling Problem, the maximal processing time and the maximal initial machine runtime are randomly drawn from the given interval. The interval for p_{max} is chosen in a way for the optimal costs to be dominated neither by the makespan nor by the deadlines. The maximal initial runtime is meant to loosely resemble the average machine occupation times in any subsequent state, making it interpretable as initial state of their own Job Scheduling Problem as well withouth diverging too much from the distribution of hyperparameters of the initial environment. Note that the maximal weight w_{max} also serves as scaling factor when normalizing the data. Therefore, we can now extract the data for our Neural Network from any given state $s \in S$ by constructing and normalizing the input as in figure 18 and 20 and analogously to (45) defining the normalized target values as

$$y_*^{(s)}(a) := softmax(\frac{Q^*(s,a^*)}{Q^*(s,a)}) \tag{50}$$

with $a^* := \mu^*(s) = \arg\min_{a \in A(s)} Q^*(s,a)$ being the true optimal action in state s.

To simulate Job Scheduling Problems compatible with the declared conditions, every Job $j = 1, \ldots, J$ and Machine $m = 1, \ldots, M$ gets randomly assigned processing times, deadlines, weights and initial runtimes according to these maximal values:

- $p_{jm} = 1, \ldots, p_{max}$
- $d_j = 0, \ldots, d_{max}$
- $\delta_m = 0, \ldots, d_{max}$
- $w_j = 1, \ldots, w_{max}$
- $\omega_m = 1, \ldots, w_{max}$
- $r_m^{(init)} = 0, \ldots, r_{max}^{(init)}$

If no machine got assigned 0 as initial runtime, choose one random Machine and reassign its initial runtime to 0.

Coherent to the stated specifications, we simulate 120.000 indepedent Job Scheduling Problems. We allocate them to 3 sets:

- **Training**: 100.000
- **Validation**: 10.000
- **Test**: 10.000

The Jobs of any state s will be sorted by Π_{m_s} for our Neural Network to receive at its input. However, this permutation might already induce a reasonable estimator of the optimal policy μ^* in form of the scheduling algorithm defined by always assigning the Job that is placed first. We aim to prevent our Neural Network from only concentrating on the majority of cases where one of the Jobs from the start of the ordered list will be optimal. Instead, we want to further encourage it to also focus on learning how to distuingish the optimal actions in supposedly less likely scenarios, where one of the Jobs with a higher runtime on the currently free Machine or deactivating the same might be the optimal action. In strive to boost its performance we will therefore balance our training data. Thus, for every Job Scheduling Problem of the training set, divide the state Space S into 18 subsets of the form

$$S_{\hat{J},\hat{M}} := \{s \in S \mid J^{(s)} = \hat{J} \wedge M^{(s)} = \hat{M}\} \tag{51}$$

associated with the respective pairs of remaining Job and Machine numbers. Ignore states with $J^{(s)} = 1, 2$ or $M^{(s)} = 1$, since these are so simple to solve that a straight forward computation of their Q-values makes more sense.
Next, divide each of these subsets of states into further $(\hat{J} + 1)$ sub-subsets, each of which is defined by the optimal action of all its states being equal to the respective $\hat{a}$ for $\hat{a} = 1, \ldots, J^{(s)}, J + 1$:

$$S_{\hat{J},\hat{M}}(\hat{a}) := \{s \in S_{\hat{J},\hat{M}} \mid \arg\min Q(s,a) = \hat{a}\}$$

Finally, for each of the 18 subsets $S_{\hat{J},\hat{M}}$, pick an action-index $\hat{a} = 1, \ldots, J+1$ at random and then uniformally sample over the immanent states of $S_{\hat{J},\hat{M}}(\hat{a})$. By repeating this for every Job Scheduling Problem, we receive a balanced training set of $100.000 * 18$ data points.

The validation and test set also get divided into 18 subsets of the form $S_{\hat{J},\hat{M}}$ each, defined by the amount of remaining Jobs $\hat{J} = 3, \ldots, 8$ and still working Machines $\hat{M} = 2, 3, 4$ of the corresponding states. However, we do not balance them for their optimal action, so for every state space S we just uniformally draw one random state from every subset, leaving us with a validation and test set of $10.000 * 18$ data points each.

7.2 Deeply Reinforced Job Scheduling Problems

To use our Neural Network on Job Scheduling Problems with a higher number J of Jobs, we need to train it on data from such environments. However, due to the computation complexity of the Q-values being in the order of $O(J * M)$, calculating the exact target values of (50) might exceed our computational resources. Thus, we will estimate them instead by applying the techniques of Deep Reinforcement Learning.

So far we have generated 100.000 Job Scheduling Problems, from each of which we have extracted one state for every combination of numbers of remaining Jobs from 3 to 8 and 2,3 or 4 operating Machines. With the associated data of these states we have trained our Neural Network, resulting in the parameters ϑ, to which we therefore initialize our Q-Network and Target Network. Our goal is now to continue training our Neural Network on data of successively increasing numbers of Jobs. Hence, anagolous to the supervised learning part of the previous subsection and starting with $J_0 := 8$, in every iteration step $i \in \mathbb{N}$ we simualte another set of 100.000 independent Job Scheduling Problems consisting of $J_i := J_{i-1} + 1$ Jobs and $M = 4$ Machines. From each we want to extract 3 states for the respective combinations $(J_i, 4)$, $(J_i, 3)$ and $(J_i, 2)$ of reamining Jobs and operating Machines. Naturally, if a Job Scheduling Problem is represented by the initial state s_{4M} and the induced state space S, the only possible states are:

$$\begin{aligned}
&(J_i, 4) \to s_{4M} \\
&(J_i, 3) \to s_{3M} := tr(s_{4M}, J_i + 1) \\
&(J_i, 2) \to s_{2M} := tr(s_{3M}, J_i + 1)
\end{aligned}$$

For each of these states $s \in \{s_{4M}, s_{3M}, s_{2M}\}$, the action space of feasible actions is the entire action space $A(s) = A = \{1, \dots, J_i + 1\}$. Thus, to transform these states to data that we can use to further train our Neural Network, we now need to estimate all the corresponding Q-values $Q^*(s, a)$. Per definition (7), these are the sum of the immediate costs $c_s(a)$ from (42) and the minimal future costs $V_{\mu^*}(s')$ of the Bellmann Equation (6), arising by following the optimal policy μ^* from the successor state $s' := tr(s, a)$ on.

For the case that the taken action $a = 1, \dots, J_i$ corresponds to a Job assignment, in the successor state s' only $J^{(s')} = J_i - 1 = J_{i-1}$ Jobs remain. Hence, we can use our Target Network with the parameters θ_{i-1} derived from the previous iteration step $i - 1$ to induce the policy $\mu_{\theta_{i-1}}$ that estimates μ^*. Consequently, $V_{\mu_{\theta_{i-1}}}(s')$ is a trained estimator of $V_{\mu^*}(s')$. Let $s_1, \dots, s_n$ be the sequence of states created by following $\mu_{\theta_{i-1}}$ as in (35), starting in $s_1 := s'$, and let ϑ be the paremters obtained from the supervised learning part. Since therefore μ_ϑ is the result of supervisedly and balancedly training our Neural Network on the ground-true Q-values of states with up to J_0 remaining Jobs, we want to use it as the estimator of the optimal policy μ^* for any state s_k, $k = 1, \dots, n$, with $J^{(s_k)} \leq J_0$. So to preserve the according parameters ϑ and consequently also further improve the stability of the learning process of the Q-Network, we decide to introduce a second set of parameters ϑ_{i-1} for the Target Network to use for any state with s_k with $J^{(s_k)} > J_0$. So while we keep ϑ static, ϑ_{i-1} will get updated in every iteration step. Therefore, we define

$$\mu_{\theta_{i-1}}(s_k) := \begin{cases} \mu_{\vartheta_{i-1}} & J^{(s_k)} > J_0 \\ \mu_{\vartheta,opt} & J^{(s_k)} \leq J_0 \end{cases} \tag{52}$$

with $\vartheta_0 := \vartheta$. The additive *opt* denotes that the actual optimal action $\mu^*(s_k)$ gets chosen by computing the ground-true Q-values $Q^*(s_k, a)$ for any state s_k with less than three Jobs remaining or only one Machine operating in case that also less than J_0 Jobs remain. This will be explained in greater detail in the next section. Consequently, we obtain the estimations

$$Q(s, a, \theta_{i-1}) = c_s(a) + V_{\mu_{\theta_{i-1}}}(s') \tag{53}$$

of the Q-values $Q^*(s, a)$ for every $a = 1, \dots, J_i$.

To estimate the Q-values $Q^*(s, J_i + 1)$, corresponding to the action $J_i + 1$ of turning the Machine m_s off, we additionally create the successor state $s_{1M} := tr(s_{2M}, J_i + 1)$. Shutting down the last operating Machine before terminating all Jobs is not a feasible action. Therefore, its space of feasible actions is given by $A(s_{1M}) = \{1, \dots, J_i\}$. By once again estimating all the

Q-values $Q^*(s_{1M}, a)$ of every job assignment $a = 1, \ldots, J_i$ as in (53), we can now recursively calculate the estimated values of:

1. $Q(s_{2M}, J+1, \theta_{i-1}) := c_{s_{2M}}(J_i + 1) + \min\limits_{a'=1,\ldots,J_i} Q(s_{1M}, a', \theta_{i-1})$
2. $Q(s_{3M}, J_i+1, \theta_{i-1}) := c_{s_{3M}}(J_i + 1) + \min\limits_{a'=1,\ldots,J_i} Q(s_{2M}, a', \theta_{i-1})$
3. $Q(s_{4M}, J_i+1, \theta_{i-1}) := c_{s_{4M}}(J_i + 1) + \min\limits_{a'=1,\ldots,J_i} Q(s_{3M}, a', \theta_{i-1})$

Finally, the estimated normalized target values are now exactly the ones of (45):

$$y^{(s)}_{\theta_{i-1}}(a) := softmax(\frac{Q(s, a_{\theta_{i-1}}, \theta_{i-1})}{Q(s, a, \theta_{i-1})}) \tag{54}$$

with $a_{\theta_{i-1}} := \arg\min\limits_{a=1,\ldots,J_i} Q(s, a, \theta_{i-1})$.
Note that due to our construction, for each of the states $s_{4M}, s_{3M}, s_{2M}, s_{1M}$ we are able to inexpensively compute (54) for every action

$$a \in A(s_{4M}) = A(s_{3M}) = A(s_{2M}) = \{1, \ldots, J_i, J_i + 1\}$$

and feasible action $a \in A(s_{1M}) = \{1, \ldots, J_i\}$, instead of having to apply a behaviour policy π which would only explore some of them.

These $100.000 * 3$ data points with estimated normalized target values can now be used to train the Q-Network analogously to the previous subsection, with the difference that it is already initialized with the parameters ϑ_{i-1}. The resulting set of parameters is our updated ϑ_i, as it has been illustrated in figure 2. Therefore, the policy μ_{θ_i} induced by our Target Network is updated as well and can then legitimely be applied to Job Scheduling Problems with up to $J_i = J_0 + i$ Jobs. This concludes one iteration step. Subsequently, we increase the number of Jobs J_i by one to $J_{i+1} = J_i + 1$ and repeat.

Theoretically, we could arbitrarily continue this process. One problem that we however might run into at later iteration steps is that when we simulate Job Scheduling Problems with significantly higher Job numbers than J_0 and then apply our target policy to create the sequence of successor states $s_1, \ldots, s_n$ to estimate the corresponding Q-values as we did above, the distribution of deadlines and machine occupation times might at some point deviate significantly for these states from the ones used to simulate Job Scheduling Problems in the supervised subsection. This can likely be managed by creating Job Scheduling Problems of the desired maximal number of Jobs J_{max} and taking the actions recommended by policy μ_{sched} of the

scheduling algorithm that we will compare our results to (or any other such decision rule) until reaching a state s_k where only $J^{(s_k)} = J_0$ Jobs are left. Then, we compute the ground-true Q-values of these states and train our Network in a supervised maneer on the resulting data instead of simulating Job Scheduling Problems that initially start with J_0 Jobs and therefore follow the distribution of deadlines and initial Machine-occupations defined at the beginning of this section.

8 Training and Testing

8.1 Supervised Training

To supervisedly train our Neural Network, we will use an *Adam*-Optimizer with a learning rate of 0.001 and the mean-squared-error as loss function. In one training epoch, we only consider one of the 18 subsets $S_{\hat{J},\hat{M}}$ of (51). In the next epoch, we then consider the next of these ($\hat{J}$,$\hat{M}$)-related subsets, so we need 18 epochs to see the entire training data set. Their ordering is random and after having completed an entire data loop, we shuffle their sequence again. In each of these epochs, we split the set into random batches of 128 data points. In total, we train for $30 * 18$ epochs.
Since we have computed the true Q-values, we can define a useful custom metric ΔQ, dividing the ground-true Q-value of the suggested action $a_\vartheta = \mu_\vartheta(s)$ from (47) of our Network by the one of the true optimal action $a^* = \mu^*(s)$ and substracting 1:

$$\Delta Q(s, a_\vartheta) := \frac{Q^*(s, a_\vartheta)}{Q^*(s, a^*)} - 1 = \frac{Q^*(s, a_\vartheta) - Q^*(s, a^*)}{Q^*(s, a^*)} \tag{55}$$

This metric denotes the relative deviation from the Q-value of the optimal action $\mu^*(s)$ in state s that arises by taking the action $\mu_\vartheta(s)$ proposed by our Neural Network instead. By printing its average over every data point, we obtain the results of the following table 4 for the training, validation and test data respectively after having finished the last epoch:

Jobs	Machines	training loss $[10^{-3}]$	validation loss $[10^{-3}]$	test loss $[10^{-3}]$	training ΔQ [%]	validation ΔQ [%]	test ΔQ [%]
3	2	0.27	1.43	1.44	0.45	0.14	0.15
	3	0.34	0.99	0.98	0.33	0.26	0.23
	4	1.06	1.54	1.58	0.36	0.46	0.45
4	2	1.58	0.79	0.79	0.63	0.16	0.13
	3	0.18	0.34	0.34	0.51	0.15	0.21
	4	1.38	0.52	0.51	0.24	0.31	0.32
5	2	0.26	0.54	0.54	0.58	0.17	0.17
	3	0.21	0.21	0.22	0.60	0.22	0.24
	4	0.24	0.24	0.25	0.53	0.31	0.31
6	2	0.36	0.41	0.41	0.28	0.20	0.21
	3	0.54	0.19	0.20	0.47	0.28	0.32
	4	0.21	0.17	0.17	0.45	0.32	0.35
7	2	0.71	0.33	0.33	0.31	0.20	0.19
	3	0.53	0.21	0.21	0.45	0.40	0.43
	4	0.42	0.18	0.18	0.45	0.41	0.40
8	2	0.21	0.31	0.31	0.34	0.32	0.32
	3	0.31	0.25	0.25	0.30	0.60	0.57
	4	0.18	0.21	0.22	0.45	0.63	0.63

Table 4: Data Results of Supervised Learning

All losses in this table have been multiplied by 10^3 for better readability. Note that they tend to be lower in some cases and the metric ΔQ is lower in general for the validation and test data compared to the training data. This is because only the latter has been balanced and is therefore harder to predict.
The values of the ΔQ-metric are less than 1% for every row of all three data sets. Hence, on average the action $\mu_\vartheta(s)$ produces costs that deviate less than 1% from the minimal costs arising by following the optimal policy μ^* for any state $s \in S$ of any of the state spaces S of the related Job Scheduling Problem.

8.2 Supervised Testing

Now, we want to see how well our Network compares to the scheduling algorithm of 1, whose induced policy will be denoted by μ_{sched}. We use two different versions of the Neural Network and the scheduling algorithm. One time we follow their policy for the entire scheduling, the other time we stop as soon as only two or less Jobs remain to be assigned or only one Machine is active:

$$\mu_{\vartheta,opt}(s) := \begin{cases} \mu_\vartheta(s) & (J^{(s)} \geq 3 \wedge M^{(s)} \geq 2) \vee J^{(s)} > 8 \\ \mu^*(s) & else \end{cases} \tag{56}$$

$\mu_{sched,opt}$ is defined analogously. We do this due to the fact that we have not trained our Neural Network for these cases, since they can be computed very inexpensively.
We create another 10.000 Job Scheduling Problems, each associated with an initial state s_1 in the same range of environmental values as stated in the beginning of the previous section, and calculate their optimal scheduling costs $V_{\mu^*}(s_1)$. We then divide the costs $V_\mu(s_1)$ created by any of the other policies μ by these optimal costs and subtract 1 to know how much additional relativ costs

$$\Delta V_\mu(s_1) := \frac{V_\mu(s_1)}{V_{\mu^*}(s_1)} - 1 = \frac{V_\mu(s_1) - V_{\mu^*(s_1)}}{V_{\mu^*(s_1)}} \tag{57}$$

we have to pay for following each of them. By taking their average over all 10.000 scheduling problems, we obtain the results of the following table 8.2:

Algorithm	Policy	ΔV_μ [%]
Neural Network with optimal end	$\mu_{\vartheta,opt}$	2.51
Neural Network	μ_ϑ	4.47
Scheduling Algorithm with optimal end	$\mu_{sched,opt}$	34.46
Scheduling Algorithm	μ_{sched}	45.33

Table 5: Policy Cost Results from Supervised Learning

We can see that our Neural Network massively outperforms the comparative scheduling algorithm, producing less than 10% of its additional costs and values very close to the optimal ones, with especially $\mu_{\vartheta,opt}$ seeming to be a great estimator of μ^*.

8.3 Deeply Reinforced Training

We now want to see if we can use the techniques of Deep Reinforcement Learning presented in the previous section to adapt our Neural Network to Job Scheduling Problems with greater numbers of Jobs J_i in every iteration $i \in \mathbb{N}$. Analogously to the previous subsection, we use an *Adam*-Optimizer with a learning rate of 0.001, the mean-squarred-error as loss function and only one of the 3 $(J_i, 2)$, $(J_i, 3)$, $(J_i, 4)$ combinations of numbers of remaining Jobs and Machines in every epoch. However, this time we do not shuffle them but loop through them in ascending order regarding their amount of active Machines $2, 3$ and 4. Again, in each of these epochs we split each set into random batches of 128 data points. At every iteration step, we initialize the Q-Network with parameters θ_{i-1} and loop 5 times through the newly es-

timated data of J_i Jobs, hence training it for $5 * 3$ epochs. Afterwards, from the second iteration on, we train it again on the entirety of so far estimated data of J_1 up to J_i Jobs. We therefore need $3 * i$ epochs for one data loop. We do 3 loops in the second iteration $i = 2$, 4 loops in the third $i = 3$ and 5 loops in the fourth $i = 4$, resulting in another $3i * (i + 1)$ training epochs for $i \geq 2$. After finishing the last epoch, we obtain the updated parameters θ_i, producing the training losses displayed in the following table 6:

Jobs	Machines	θ_1 loss $[10^{-3}]$	θ_2 loss $[10^{-3}]$	θ_3 loss $[10^{-3}]$	θ_4 loss $[10^{-3}]$	θ_1 ΔQ_1 [%]	θ_2 ΔQ_2 [%]	θ_3 ΔQ_3 [%]	θ_4 ΔQ_4 [%]
	2	1.65	1.42	1.59	1.65	0.36	0.32	0.33	0.34
9	3	1.27	1.01	1.08	1.26	0.68	0.64	0.66	0.72
	4	1.46	1.58	1.69	2.09	0.90	0.94	1.01	1.16
	2		1.21	1.29	1.31		0.31	0.31	0.30
10	3		0.87	0.84	0.86		0.67	0.68	0.69
	4		1.12	1.15	1.30		0.96	1.01	1.09
	2			1.13	1.12			0.32	0.30
11	3			0.75	0.68			0.72	0.70
	4			0.93	0.93			1.07	1.10
	2				0.98				0.28
12	3				0.58				0.68
	4				0.74				1.12

Table 6: Data Results of Deep Reinforcement Learning

Here, due to the true Q-values being unkwon, we used the estimation of the ΔQ-metric of (55):

$$\Delta Q_i := \frac{Q(s, a_{\theta_i}, \theta_i) - Q(s, a_{\theta_{i-1}}, \theta_{i-1})}{Q(s, a_{\theta_{i-1}}, \theta_{i-1})}$$

Similiar to the promising results from table 4, the values of the ΔQ_i-metric are at most sligthly above 1% for every row of every set of parameters θ_i, $i = 1, 2, 3, 4$. Hence, on average the action $\mu_{\vartheta_i}(s)$ produces estimated costs that deviate at most marginally more than 1% from the estimated minimal costs $Q(s, a_{\theta_{i-1}}, \theta_{i-1})$ for any state $s \in \{s_{4M}, s_{3M}, s_{2M}\}$ of any of the related Job Scheduling Problems.

8.4 Deeply Reinforced Testing

We now want to compare these augmented policies to $\mu_{\vartheta,opt}$, the one of our supervisedly trained Neural Network with parameters ϑ, to see if we were able to further improve its performance on higher numbers of Jobs. For this, we simulate 10.000 independent Job Scheduling Problems, each associated with an initial state s_{4M}, in the same range of environmental values as stated in the beginning of the previous section and with J_i Jobs and 4 Machines. We then calculate the average of the quotient $\Delta V^{(i)}(s)$ of the scheduling costs for following μ_{θ_i} and the scheduling costs arising by adhering to $\mu_{\vartheta,opt}$:

$$\Delta V^{(i)}(s_{4M}) := \frac{V_{\mu_{\theta_i}}(s_{4M})}{V_{\mu_{\vartheta,opt}(s_{4M})}}$$

Note that in both cases we use the actual optimal policy μ^* by computing the ground-true Q-values as soon as less than 3 Jobs remain or only one Machine continues operating while not more than 8 Jobs remain. This leads us to the quotients of the following table 8.4:

Iteration	Policy	J_i	$\Delta V^{(i)}$
1	μ_{θ_1}	9	1.02
2	μ_{θ_2}	10	1.03
3	μ_{θ_3}	11	1.04
4	μ_{θ_4}	12	1.04

Table 7: Comparing Policy Costs from Supervised Learning to DRL

Unfortunately, the scheduling costs produced by following the policy θ_i in iteration i of the DRL process is 2 to 4 percent higher than the one produced by $\mu_{\vartheta,opt}$. Therefore, we will continue using the set of parameters ϑ for our Neural Network and the corresponding policy $\mu_{\vartheta,opt}$, both obtained by supervised learning.

9 Results

We will now test how well our scheduling policy $\mu_{\vartheta,opt}$, derived from our Neural Network with parameters ϑ, compares to the heuristic policy $\mu_{sched,opt}$, defined in (56) and induced by the comparative scheduling algorithm of 1, with the addition of using μ^* as soon as less than 3 Jobs remain or only one Machine keeps operating while not more than 8 Jobs remain. We wish to test on a great variety of numbers of Jobs and Machines. For each such (J,M)-combination, we simualte 10.000 independent Job Scheduling Problems, each associated with an initial state s_1 in the same range of environmental values as stated in the beginning of the previous section and with J Jobs and M Machines. The entries of the following table 9 then denote the quotient

$$\frac{V_{\mu_{sched,opt}}(s_1) - V_{\mu_{\vartheta,opt}}(s_1)}{V_{\mu_{\vartheta,opt}}(s_1)}$$

of the additional relative costs in percent [%] arising from using the scheduling algorithm compared to applying our Neural Network with parameters ϑ:

J \ M	2	3	4	5	6	8	10	12	15
8	19.57	27.47	31.17	32.09	29.79	18.13	08.32	1.47	-06.52
9	21.11	25.38	32.39	33.79	32.44	20.22	9.38	-1.44	-6.42
10	20.09	25.62	32.26	35.43	34.43	23.02	11.09	00.29	-07.04
11	18.02	26.86	33.03	36.21	35.88	25.22	11.65	02.04	-06.57
12	18.83	26.33	32.43	36.53	37.09	26.36	12.61	02.90	-07.07
15	17.98	22.07	30.30	36.99	38.27	29.24	15.17	03.48	-07.62
20	15.14	21.29	26.90	34.23	38.14	31.53	17.09	04.87	-07.57
30	13.36	16.73	21.20	27.74	32.62	31.72	18.04	04.39	-08.51
50	10.65	12.52	14.94	18.56	23.40	26.13	16.36	04.13	-08.14
75	09.91	10.09	10.56	12.69	14.97	18.02	12.68	03.72	-07.24
100	09.94	08.96	07.97	07.59	08.45	11.15	09.10	03.27	-05.46

Table 8: Comparing Policy Costs from NN to Scheduling Algorithm

We see that our Neural Network significantly outperforms the comparative scheduling algorithm for up to 100 Jobs and 10 Machines, while the results seem to break even around 12 Machines. Increrasing the number of Machines seems to have a higher negative impact on the comparative performance of our Neural Network than to increase the number of Jobs. Interestingly, it does not hold the biggest advantage on Job Scheduling Problems with 8 Jobs and 4 Machines, even though this is the scenario it has primarily been trained on, but instead it appears to have the highest edge at the range of 8

to 20 Jobs and 4 to 6 Machines as well as 15 to 30 Jobs and 5 to 8 Machines. Nevertheless, the difference being slightly smaller for 2 and 3 Machines does not necessarily indicate that it performs worse there, but that the scheduling algorithm might just perform relatively better than under the mentioned circumstances.

In every constellation but two with up to 30 Jobs and between 3 and 8 Machines, the scheduling algorithm create costs that are at least 20% higher than the ones our Neural Network would incur, in most cases even over 30%. This especially becomes vast when relating it to the 2.51% of additional scheduling costs $\Delta V_{\mu_{\vartheta,opt}}$ (57) that following $\mu_{\vartheta,opt}$ produced when compared to optimal schedules for 8 Jobs and 4 Machines in table 8.2. We can reasonably assume that higher numbers of Jobs or Machines tendencially also lead to relatively higher mean additional scheduling costs. So while we do not know how much these grow for our Neural Network, we can deduce that any such relative increase that stems from augmenting the number of Jobs or Machines within these margins is on average 20% to 40% higher for $\mu_{sched,opt}$ than for $\mu_{\vartheta,opt}$ as well. This indicates that on this set of problems, even though it has been trained only on the specific environment of those starting with 8 Jobs and 4 Machines, our Neural Network generalizes better than the heuristic algorithm, while also starting from a much better baseline of 2.51% additional costs in contrast to the 34.46%. This becomes evident from the same table 8.2, too, when comparing the rise from $\Delta V_{\mu_{\vartheta,opt}}$ to ΔV_{μ_ϑ} and $\Delta V_{\mu_{sched,opt}}$ to $\Delta V_{\mu_{sched}}$ for not computing the optimal policy as soon as less than 3 Jobs remain or only 1 Machine continues operating while not more than 8 Jobs remain. The Neural Network has not been trained on these instances, yet its respective increase is less than 2%, whereas it is higher than 10% for the scheduling algorithm.

Based on these results, one can conclude that our Neural Network does perform and generalize outstandingly well, especially with regards to higher Job numbers, deeming it superior to the heuristic scheduling algorithm within the range of 100 Jobs and 10 Machines. They suggest that especially for Job Scheduling Problems with up to 30 Jobs and 8 Machines suited to our defined problem environment, the policy $\mu_{\vartheta,opt}$ induced by our Neural Network is an excellent estimator of the optimal policy μ^* and therefore of the least-cost producing schedule.

10 Conclusion and Future Work

We have embedded our specific type of Job Scheduling Problems defined in (5) into a Reinforcement Learning environment, associating every scheduling decision with a state. We have proven that we can identify any scheduling with a policy, so that our objective function (1) is equivalent to the policy costs. We were able to convert these states to normalized data without loosing any information. Our normalization of the time-related input values allows for the processing of states without predefined limitations for these values. Prospective studies might investigate giving a weight to the makespan C_{max} as well. Consequently, the weights of every state could be normalized by dividing them by the highest occuring weight in that state, yielding the same benefit of not having to predefine a maximum allowed weight implicit in the data set, therefore capacitating the Neural Network to be applied on an even broader range of Job Scheduling Problems without the need of retraining it first.

We have introduced Neural Network Architectures from Natural Language Processing to handle the challenges immanent in the data of varying numbers of Jobs and Machines. By interpreting the set of Jobs and Machines as an ordered time series and combining the architectures of LSTMs, Transformers and Attention based Action-Pointers to our needs, we constructed a Neural Network able to process an arbitrary amount of Jobs and Machines. Future work could explore the possibility of either adding an equivariant permutation layer as base, as presented in [52], or implementing the architecture of the permutation-invariant Set Transformer introduced in [53] to dispose the need of such an ordering.

We have derived the feasible policy $\mu_{\theta,opt}$ in (47) from the output of our Neural Network as an seemingly well performing estimator of the optimal policy μ^*, capable of proposing any feasible action. For any possible scheduling situation of any of our Job Scheduling Problems, it creates a schedule that estimates the optimal one.

Identifying scheduling costs with the structure of a directed tree enabled us to compute them for every possible schedule of 100.000 independent Job Scheduling Problems of 8 Jobs and 4 Machines. The resulting data gave us the capacity to supervisedly and efficiently train our Neural Network, using only little computational power. With access to more elaborate resources, future studies could extend this approach to decidedly higher numbers of Jobs and Machines. We then have tested the Neural Network on another 10.000 simulated Job Scheduling Problems of 8 Jobs and 4 Machines and compared it to the competetive heuristic algorithm of 1. The schedules in-

duced by our Neural Network only created 2.51% additional costs relative to the theoretical optimal schedules, while the scheduling algorithm produced relative additional costs of 34.46%, more than 10 times as much.

By initializing the Target Network to these learned parameters ϑ, we have introduced an approach that uses techniques from Deep Reinforcement Learning and tested the resulting Q-Network on Job Scheduling Problems with 9 to 12 Jobs and 4 Machines. Unfortunately, the performance of the Neural Network did not improve. This may be due to the fact that we have balanced the training data when training supervisedly, while for this extension of our approach we have not. Further research could therefore build on this method by balancing the training data for the Q-Network in every iteration step.

Finally, we have analyzed how well our Neural Network generalizes to Job Scheduling Problems of different numbers of Jobs and Machines. For each constellation, we have compared it to our heuristic algorithm for 100.000 Job Scheduling Problems ranging from 8 to 100 Jobs and 2 to 15 Machines. Our Neural Network was able to significantly outperform the scheduling algorithm for up to the entirety of 100 Jobs and 10 Machines, while breaking even at around 12 Machines. In general, it was also able to vastly outperform the scheduling algorithm in the margin of 3 to 8 Machines and up to 30 Jobs. Under these conditions, the costs incured by the latter were almost always at least 20% higher, in most cases even more than 30%. It seems viable to assume that the additionally incured costs relative to optimal schedule costs tend to increase when augmenting the numbers of Jobs or Machines. That would imply that our Neural Network also generalizes better within these ranges due to the average cost increments consequently being smaller. Further exploration of its performance under differing constraints would be helpful to fully understand its capability and potential as well as its weaknesses and limitations.

In light of these findings, our approach seems to be valuable and warrants further investigation. The results indicate that our created Neural Network may potentially achieve state-of-the-art results for Job Scheduling Problems that meet the required criteria, given that they are already extremly hard to solve with traditional approaches in an environment of about 5 machines and 30 Jobs, even for the case of scheduling on Uniform Machines [1, p. 143]. Future directions may include the attempt to take advantage of the flexibile nature of Neural Network Architectures, especially when compared to more monovalently predisposed and especialized approximation algorithms, as well as to analyze how well they generalize to Job Scheduling Problems with similiar, yet deviating constraints, such as differing distributions of

processing times, deadlines, weights and initial machine occupation times. It would especially be intriguing to examine how our Neural Network would perform on Stochastic Job Scheduling Problems, by feeding it the expected processing time on every respective Machine instead of a deterministic one.

11 References

[1] Michael L Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 2008.

[2] Vaibhav Kumar, Siddhant Bhambri, and Prashant Giridhar Shambharkar. "Multiple resource management and burst time prediction using deep reinforcement learning". In: *Eighth International Conference on Advances in Computing, Communication and Information Technology (CCIT)*. 2019, p. 8.

[3] Hongzi Mao et al. "Resource management with deep reinforcement learning". In: *Proceedings of the 15th ACM workshop on hot topics in networks*. 2016, pp. 50–56.

[4] Pierre Tassel, Martin Gebser, and Konstantin Schekotihin. "A reinforcement learning environment for job-shop scheduling". In: *arXiv preprint arXiv:2104.03760* (2021).

[5] David P Williamson and David B Shmoys. *The Design of Approximation Algorithms*. Cambridge university press, 2010.

[6] Diego de Oliveira Hitzges. *Dieguinho1612/Job-Scheduling-Deep-Reinforcement-Learning*. Version v1.0.0. Apr. 2023. DOI: 10.5281/zenodo.7872658. URL: https://github.com/Dieguinho1612/Job-Scheduling-Deep-Reinforcement-Learning.

[7] José Verschae and Andreas Wiese. "On the configuration-LP for scheduling on unrelated machines". In: *Journal of Scheduling* 17 (2014), pp. 371–383.

[8] Jan Karel Lenstra, David B Shmoys, and Éva Tardos. "Approximation algorithms for scheduling unrelated parallel machines". In: *Mathematical programming* 46 (1990), pp. 259–271.

[9] Hong Zhou, Zhengdao Li, and Xuejing Wu. "Scheduling unrelated parallel machine to minimize total weighted tardiness using ant colony optimization". In: *2007 IEEE international conference on automation and logistics*. IEEE. 2007, pp. 132–136.

[10] Tommi Jaakkola, Michael Jordan, and Satinder Singh. "Convergence of stochastic iterative dynamic programming algorithms". In: *Advances in neural information processing systems* 6 (1994), pp. 703–710.

[11] Tommi Jaakkola, Michael Jordan, and Satinder Singh. "On the convergence of stochastic iterative dynamic programming algorithms". In: *Neural Computation* (1993).

[12] Francisco S Melo. "Convergence of Q-learning: A simple proof". In: *Institute Of Systems and Robotics, Tech. Rep* (2001), pp. 1–4.

[13] Pieter Abbeel and John Schulman. “Deep reinforcement learning through policy optimization”. In: *Tutorial at Neural Information Processing Systems* (2016).

[14] Timothy P Lillicrap et al. “Continuous control with deep reinforcement learning”. In: *arXiv preprint arXiv:1509.02971* (2015).

[15] Artem Oppermann. *Deep Q-Learning*. [Online; accessed March 06, 2023]. 2021. URL: `https://artemoppermann.com/de/deep-q-learning/`.

[16] Guang Yang et al. “DHQN: a Stable Approach to Remove Target Network from Deep Q-learning Network”. In: *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2021, pp. 1474–1479.

[17] KR1442 Chowdhary and KR Chowdhary. “Natural language processing”. In: *Fundamentals of artificial intelligence* (2020), pp. 603–649.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[19] Tomas Mikolov et al. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[20] Diego de Oliveira Hitzges. “Worteinbettung für Machine Learning in der Sprache”. BA thesis. Technical University of Berlin, 2019.

[21] Maite Gimenez, Javier Palanca, and Vicent Botti. “Semantic-based padding in convolutional neural networks for improving the performance in natural language processing. A case of study in sentiment analysis”. In: *Neurocomputing* 378 (2020), pp. 315–323.

[22] Graham Neubig. “Neural machine translation and sequence-to-sequence models: A tutorial”. In: *arXiv preprint arXiv:1703.01619* (2017).

[23] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to sequence learning with neural networks”. In: *Advances in neural information processing systems* 27 (2014).

[24] Mikael Boden. “A guide to recurrent neural networks and backpropagation”. In: *the Dallas project* 2.2 (2002), pp. 1–10.

[25] Jeffrey L Elman. “Finding structure in time”. In: *Cognitive science* 14.2 (1990), pp. 179–211.

[26] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[27] Leila Arras et al. “Explaining recurrent neural network predictions in sentiment analysis”. In: *arXiv preprint arXiv:1706.07206* (2017).

[28] Zachary C Lipton, John Berkowitz, and Charles Elkan. “A critical review of recurrent neural networks for sequence learning”. In: *arXiv preprint arXiv:1506.00019* (2015).

[29] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. “Natural language processing: an introduction”. In: *Journal of the American Medical Informatics Association* 18.5 (2011), pp. 544–551.

[30] Abby A Goodrum. “Image information retrieval: An overview of current research”. In: *Informing Science* 3 (2000), p. 63.

[31] Wenhao Jiang et al. “Recurrent fusion network for image captioning”. In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 499–515.

[32] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[33] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. “Speech recognition with deep recurrent neural networks”. In: *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee. 2013, pp. 6645–6649.

[34] Razvan Pascanu et al. “How to construct deep recurrent neural networks”. In: *arXiv preprint arXiv:1312.6026* (2013).

[35] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training recurrent neural networks”. In: *International conference on machine learning*. Pmlr. 2013, pp. 1310–1318.

[36] Jingzhao Zhang et al. “Why gradient clipping accelerates training: A theoretical justification for adaptivity”. In: *arXiv preprint arXiv:1905.11881* (2019).

[37] Junyoung Chung et al. “Empirical evaluation of gated recurrent neural networks on sequence modeling”. In: *arXiv preprint arXiv:1412.3555* (2014).

[38] Ralf C Staudemeyer and Eric Rothstein Morris. “Understanding LSTM–a tutorial into long short-term memory recurrent neural networks”. In: *arXiv preprint arXiv:1909.09586* (2019).

[39] Gang Chen. “A gentle tutorial of recurrent neural network with error backpropagation”. In: *arXiv preprint arXiv:1610.02583* (2016).

[40] Hasim Sak, Andrew W Senior, and Françoise Beaufays. “Long short-term memory recurrent neural network architectures for large scale acoustic modeling”. In: (2014).

[41] Andrea Galassi, Marco Lippi, and Paolo Torroni. "Attention in natural language processing". In: *IEEE transactions on neural networks and learning systems* 32.10 (2020), pp. 4291–4308.

[42] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[43] Baosong Yang et al. "Context-aware self-attention networks". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 387–394.

[44] Ming Jiang and Shaoxiong Ji. "Cross-Modality Gated Attention Fusion for Multimodal Sentiment Analysis". In: *arXiv preprint arXiv:2208.11893* (2022).

[45] Yuan Yuan et al. "Using an attention-based LSTM encoder–decoder network for near real-time disturbance detection". In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 13 (2020), pp. 1819–1832.

[46] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. "Pointer networks". In: *Advances in neural information processing systems* 28 (2015).

[47] Jay Alammar. *The Illustrated Transformer*. [Online; accessed March 06, 2023]. 2018. URL: `https://jalammar.github.io/illustrated-transformer/`.

[48] Bryan Lim et al. "Temporal fusion transformers for interpretable multi-horizon time series forecasting". In: *International Journal of Forecasting* 37.4 (2021), pp. 1748–1764.

[49] David Salinas et al. "DeepAR: Probabilistic forecasting with autoregressive recurrent networks". In: *International Journal of Forecasting* 36.3 (2020), pp. 1181–1191.

[50] Albert Zeyer et al. "A comparison of transformer and lstm encoder decoder models for asr". In: *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE. 2019, pp. 8–15.

[51] Alex M Lamb et al. "Professor forcing: A new algorithm for training recurrent networks". In: *Advances in neural information processing systems* 29 (2016).

[52] Nicholas Guttenberg et al. "Permutation-equivariant neural networks applied to dynamics prediction". In: *arXiv preprint arXiv:1612.04530* (2016).

[53] Juho Lee et al. "Set transformer: A framework for attention-based permutation-invariant neural networks". In: *International conference on machine learning*. PMLR. 2019, pp. 3744–3753.