# TECHNICAL UNIVERSITY OF MUNICH

## TUM Data Innovation Lab

## Planning and Control using Model-Based Reinforcement Learning


| | |
|---|---|
| Authors | Pavel Czempin, Vincent Friedrich, Ruotong Liao, Jannik Nettelnstroth |
| Mentor(s) | M.Sc. Mathias Sundholm, M.Sc. Hamdi Belhassen PreciBake GmbH |
| Co-Mentor | PhD candidate Michael Rauchensteiner |
| Project Lead | Dr. Ricardo Acevedo Cabra (Department of Mathematics) |
| Supervisor | Prof. Dr. Massimo Fornasier (Department of Mathematics) |


Feb 2021

# Abstract

During the last years, there have been numerous breakthroughs with Reinforcement Learning agents playing complex games, the most famous being DeepMind's Alpha Go defeating one of the world's best Go players Lee Sedol. Managing the inventory in a bakery to perfectly meet the customer demand is also a very complex task, even though most customers take it for granted that they always get what they order. That is why we apply Reinforcement Learning techniques in order to assist the production process in a bakery. The goals are to maximize the number of sold products as well as their freshness while at the same time minimizing the number of products that go to waste. In this project we implement Model-Based Reinforcement Learning agents that use Monte Carlo Tree Search combined with a Hawkes Process as prediction model for the consumer behaviour. Our experiments evaluating how well a Hawkes process can capture consumer behaviour patterns in a real world scenario show promising results. In the end we assess the capability of different agents to adapt to previously unknown environments and discuss how well they can handle settings with an increased amount of possible products. While model-based approaches highly depend on the ability to accurately estimate future events, we show that Monte Carlo Tree Search combined with Hawkes predictions can outperform simpler agents. Our results indicate that adding Reinforcement Learning algorithms can further improve the performance of tree search, however this approach is limited by the capability of training the algorithms in accurate settings.

# Contents

# 1 Introduction

Reinforcement Learning (RL) techniques are expected to play a key role in the future development of Artificial Intelligence. Remarkable results in playing complex games that require strategic thinking such as Go attracted the attention of a broader audience and further aroused the enthusiasm for RL. In 2016, the RL-based computer program AlphaGo defeated one of the most successful professional Go players Lee Sedol by 4-1, which was considered a landmark achievement [1].
The core idea of RL is that an agent learns by trial and error to take best actions in an environment with regard to some predefined goal. Certainly, the application of RL is not limited to board games, but how exactly can a bakery benefit from such advanced techniques?
In a bakery, one has to decide at every time point of a day which amount and what products to produce. The goal is to maximize the sales, minimize the waste and to sell products as fresh as possible. Our aim in this project is to develop an RL-agent that decides in an optimal way when to put what amount of which product into the oven. One of the biggest challenges is that the customer orders are not known a priori. In concrete terms, this means that the same baking decisions in identical situations can lead to very different outcomes concerning sales, waste and the freshness of products. As a consequence, our approach is to equip the agent with a method to simulate potential customer behaviour in order to recognize which consequences certain decisions might have. In our simplified bakery setting all randomness stems from customer behaviour, while bakery processes are deterministic. For learning a model of the environment it is therefore sufficient to predict customer orders. RL combined with a learned model of the (stochastic) environment is referred to as Model-Based Reinforcement Learning [2].
In this project, we develop Model-Based RL agents that consist of two main components. Firstly, the consumer behaviour is modeled by a Hawkes process that allows the agents to sample possible future orders. Secondly, the agents resort to Monte Carlo Tree Search (MCTS) to decide on which action to take. The internal simulations of the agents performed in MCTS are based on the order samples generated by the Hawkes process.
In the following we show how we build our Model-Based RL agents and have a closer look at the underlying concepts. In later sections, we will evaluate the performance of the agents by presenting and interpreting the results of several experiments. Our work can be seen as a continuation of last year's TUM Data Innovation Lab project with PreciBake [3].

**Related Work** Inventory management is a promising application area for Reinforcement Learning with numerous prior work. For example, RL can adjust over a finite sales horizon to maximize expected revenue [4], although frequent changes on prices do not fit for bakery price settings. Others suggest that RL learns better when the age of the products is used in state representation. Moreover, demand variance and inter dependency of products are important for perishable inventory management [5]. Another group analysed that customer demand depends on a homogeneous Poisson process and reached a near optimal policy for perishable inventory management with deep Reinforcement Learning [6]. Using predictions of stochastic user behavior and considering the dynamics of a system with Model-Based Reinforcement Learning also shows great potential of better reward, as tested on photovoltaic production [7]. Even approximated predictions without prior

knowledge of the probability distributions of the demand can still contribute positively to inventory-optimization steps rather than solving them separately in the situation of the Newsvendor problem [8]. In the former project by the previous group in Precibake it is also pointed out that model free agents are sensitive to environment changes [3], which is why the goal of our project was to introduce planning via model-based RL. In our project we focused on implementing the model-based approach as used by AlphaGo [9], which combines Reinforcement Learning with Monte Carlo Tree Search. Further improvements in the form of AlphaZero [10] and MuZero [11] exist, however they are out of the scope of this project so we leave them for future work.

# 2 Bakery Setting

## 2.1 OpenAI Gym

To implement a framework for our experiments we use Open AI Gym, an open-source toolkit for Reinforcement Learning problems [12]. It has a library of pre-implemented example problems, called the registry, but also provides the possibility to add custom problems. The problems are modeled as so-called environments which share an interface composed of the following components:

- Attribute *action_space*: Describes all possible actions an agent can carry out. It is possible to sample actions from this space.
- Function *step()*: Takes an action from *action_space* as input, carries out this action and gives the following outputs:
  - *Observation*: Environment-specific object that represents what the agent can see from the environment. We use *observation* synonymously with *state*, as the observation in our case is always the complete state that is visible to the agent.
  - *Reward*: Value that describes the reward achieved by the action. The goal of the agent is to maximize the overall rewards.
  - *Done*: Boolean variable describing whether an episode (in our setting one day in the bakery) has terminated. If this is True, the agent cannot do actions in this environment anymore, until the environment is reset.
  - *Info*: A variable containing additional information used for debugging. The contents depend on the environment.
- Function *reset()*: Resets the environment to its initial state.

## 2.2 Bakery Environment

As already explained in section 2.1, it is possible to add custom environments to the OpenAI Gym registry. The environment used for this project is composed of three parts:

1. **Producer model**: Models an oven, which can hold up to 30 products of the same type (e.g. pretzels, bread). This is the only part of the environment the agent can interact with by deciding how many products of which type should be produced. While the oven is busy, no new actions can be carried out.

2. **Consumer model**: Defines when consumers enter the bakery and which products they order. In our setting, each order consists of only one product. When working without real-world data we use a Poisson distribution to generate customer orders in the environment. Further details are given in section 2.3.

3. **Inventory**: Models the inventory of the bakery, i.e. products which are ready for sale. When the producer model is done producing new products, they are added to the inventory. After a customer orders an available product, it will be removed from the inventory.

Figure 1 shows how these parts interact in the environment. Every time the function step() is called, the producer model carries out the chosen action and the consumer model simulates the customer behaviour of that time step. Feasible actions are tuples $(p, n)$ of a product type $p$ and a quantity $n$ with $0 \leq n \leq 30$.

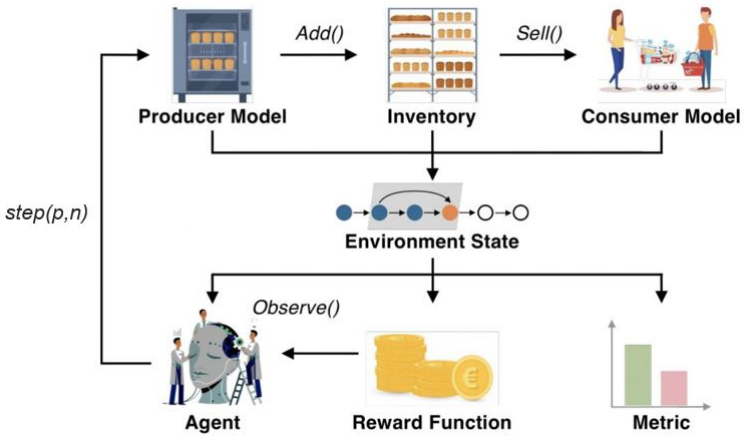



Figure 1: Graphical Representation of the PreciBake setting. The environment encapsulates item production, inventory and consumer demand. The agent tries to optimize the metric by maximizing the observed reward.

After the agent adds products to the oven, the oven is occupied and no new products can be produced until it finishes. Therefore, all agents that will be discussed in this paper skip time steps in which no action can be taken. The output *observation* of the function *step()* contains information about products currently in the oven and in the inventory as well as information about the orders during the last time step.
We divide a day into 100 time steps of equal length. By calling the function *step*(), the environment simulates one of these intervals. However, due to the skipping mentioned above, calling *step*() will advance the state forward by more than one time step for all actions that produce at least one product, as no meaningful action can be performed when the oven is full. Each type of product has a specific production time (how many time steps it takes to produce a batch of this product in the oven) and an expiration

time which describes how long a product of this type is considered to be fresh. It can still be sold after its expiration time but the freshness has an effect on the reward (see Section 2.4).

## 2.3 Consumer Models

In the course of this project we use two different models to generate consumer orders. The first approach is based on a generative model which samples the amount of orders per time step from a Poisson distribution. It is parameterized by a list of means for different intervals of the day of equal length. This means, the model can reflect the customer behaviour during different parts of the day to some degree, for instance by assigning an individual mean for the morning, noon, afternoon and evening period. For example the parameters $[15, 20, 10, 10]$ for one product type would mean that on average the model will generate 15 orders in the first quarter of the day, 20 orders in the second quarter and 10 orders in both the third and fourth quarter. A precise description on how the number of orders gets simulated can be found in the appendix.
The advantage of this model is that it is easy to parameterize and interpret. Additionally, it is a stochastic model and can generate an unlimited amount of training data.
In order to get more realistic consumer behaviour, our second approach was to use a dataset of anonymized real world sales data provided by PreciBake GmbH. For this we implemented a consumer model that simply outputs the number of observed real-world orders at each time step.

**Real Sales Data**

- customer data for 10 days in August and 15 days in December of the same year
- around 100 000 customer orders
- more than 500 different products

The main advantage of using this data is that it provides a very realistic environment for our agents to act in. The disadvantage is the fact that we are limited to 25 days.

## 2.4 Reward Metric

In order to asses the performance of different agents and give feedback to our learning agents, we designed a performance measure which will be referred to as our metric.
In our setting the agent does not receive immediate feedback for its actions and can only observe the metric at the end of an episode (i.e. the observed reward of the agent is 0 as long as the respective day has not finished yet).
The metric is composed of three different scores that correspond to the main goals in a bakery.

- *fulfilled-orders-ratio*: Corresponds to the goal of **maximizing sales**. High values mean that a high percentage of the orders are fulfilled. In our setting, an order can only be served immediately. Customers do not wait if their desired product is not in the inventory at the time they arrive.

$$m_s = \frac{\#\text{sold products}}{\#\text{ordered products}}$$

- *product-waste-ratio*: Corresponds to the goal of **minimizing waste**. High values mean that a low fraction of produced products goes to waste. Products are only wasted if they are still in the inventory or in the oven at the end of an episode.

$$m_w = 1 - \frac{\#\text{wasted products}}{\#\text{produced products}}$$

- *average freshness*: Corresponds to the goal of **maximizing the freshness of the sold products**. High values mean that a high percentage of the products were considered fresh at the time they were sold.

$$m_f = \frac{1}{\#\text{sold products}} \sum_{\text{sold product } i} \mathbb{1}[\text{age}_i \leq \text{expiration\_time}_i]$$

  Here $\mathbb{1}$ denotes the indicator function and $\text{age}_i$ is the number of time steps the product $i$ was in the inventory before being sold.

Since we want the agents to mainly focus on the first two goals and get some bonus points for complying with the third goal, we choose the overall metric as a weighted average of the three metrics:

$$m = \frac{4m_s + 4m_4 + m_f}{9}$$

The agent's objective is to maximize this quantity $m$. It would also be possible to assign different weights to the components of the metric depending on the exact objective of the bakery. All algorithms explained in this paper could adapt to any other metric but all results are based on this metric $m$.

# 3 Model-Based Reinforcement Learning

Model-based Reinforcement Learning is the key concept behind the implementations in this project. Reinforcement Learning is a technique that enables an agent to learn by trial and error to take optimal decisions in an environment. The feedback on the quality of an action in a specific situation is provided by a reward function like explained in Section 2.2. However, naïve tryouts during the learning process can be costly when directly applied in a real bakery. For example, unconventional decisions in the learning phase such as producing nothing at all could be fatal for the economic success of a bakery. Therefore, to avoid such cost, we equip the agent with an internal model of the customer behaviour. Consequently, the agent is able to internally simulate days in a bakery and to estimate the outcome of its actions. This summarizes the core idea of Model-Based RL which can

be figuratively described as enabling the agent to "think" before it acts.
More formally, Model-Based Reinforcement Learning can be defined as any RL approach that uses a known or learned model of the environment [13]. As shown in Figure 2, we use a prediction model to forecast the consumer behaviour which is based on a Hawkes intensity as we will further discuss in Section 4. We will name this model Hawkes prediction model. This Hawkes process is trained on orders that were previously generated by the real environment. Every time the agent can take an action, it applies Monte Carlo Tree Search in a simulated environment in order to decide on which action to take in the real world. Here, the most promising action based on the reward in the simulated environment will be chosen. Monte Carlo Tree Search will be explained in Section 6. In this simulated environment, the customer orders are samples from the learned Hawkes model.

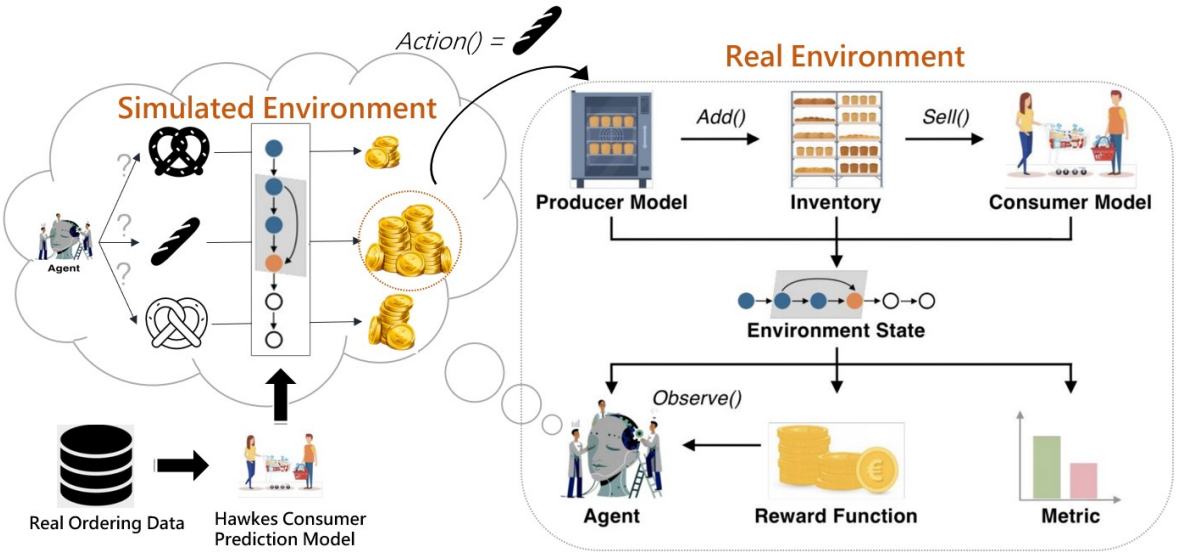



Figure 2: Graphical representation of Model-Based RL in PreciBake environment. The right-hand side is the real bakery environment. The left-hand side illustrates the simulated environment in which the agent performs simulations. The consumer orders in the simulated environment are generated by a Hawkes predition model whose parameters are trained on real ordering data beforehand. The agent uses Monte Carlo Tree Search in the simulated environment as planning method. Multiple different actions such as baking different amounts of pretzel or bread can be taken in the simulated world without influencing the real one. The action that leads to the highest average reward in the simulations will be carried out in the real world.

# 4 Hawkes Prediction Model

In order to plan their actions in an optimal way, our agents need a way to internally model what the environment might look like in the future. As explained in Section 2.2, the environment is mostly deterministic, except for the consumer model. In this section we will talk about how our agents predict this consumer behaviour.
In general, there is a trade-off between how much a model relies on bakery-specific information (for instance when peak times with an increased number of orders occur) and

how easily it can be transferred to another bakery or to a setting with different demand. The idea behind the application of the Hawkes model is to be as flexible as possible to adapt to different scenarios. In concrete terms, the model should be powerful enough to recognize whether the demand will increase or decrease based on the observed order history, without using a priori information when peak times usually occur. This is achieved by employing a Hawkes process. It only assumes that if one consumer orders a product, this will temporarily boost the probability of obtaining another order in the near future. Peak times are not explicitly defined.

## 4.1 Theory

To explain how a Hawkes process works, we first consider a setting with only one product to sell. At time $t$, we have already observed orders of this product at time steps $\mathcal{H}(t) = \{t_1, ..., t_n\}$ where $t_i < t$ for all $i \in \{1, ..., n\}$. Then, the conditional intensity function of a Hawkes process at time $t$ is defined as [14]:

$$\lambda\big(t|\mathcal{H}(t)\big) = \mu + \alpha \sum_{t_k \in \mathcal{H}(t)} \exp\big(-\omega \cdot (t - t_k)\big) \tag{1}$$

This conditional intensity has the interpretation as the expected infinitesimal rate at which orders are expected to occur around time $t$ given $\mathcal{H}(t)$[15].
It includes three parameters that can be learned: The base intensity $\mu \geq 0$, the excitement parameter $\alpha \geq 0$ and the decay parameter $\omega$. The effect of these parameters can be seen in Figure 3. Here, circles indicate the arrival of a customer. In the beginning, the intensity is equal to the base rate $\mu$, and every time a customer places an order it jumps up by $\alpha$. The speed of the exponential decay after an excitement is determined by $\omega$.

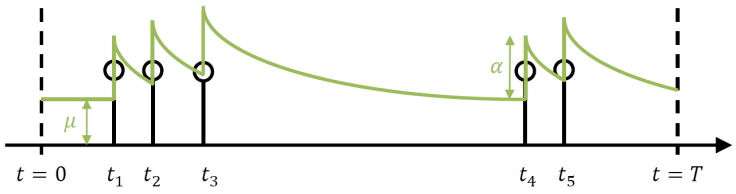



Figure 3: The intensity function of a Hawkes process [16]

This simple version of a Hawkes process uses independent intensities for each individual product, which only depend on orders of this particular type. However, the independence assumption might be violated in a real bakery setting. For instance, orders of croissants might increase the expected number of coffee orders in the next time step as they are often consumed during similar parts of the day.
To capture these possible relations, we decided to use an extension of the Hawkes process which also takes into account the inter-product dependencies. In this extension, the conditional intensity function for a product $i$ takes into account the order history $\mathcal{H}_j(t)$ of every product $j = 1, ..., N$ up to time $t$.

$$\lambda_i\big(t|\mathcal{H}_1(t), ..., \mathcal{H}_N(t)\big) = \mu_i + \sum_{j=1}^{N} \left( \alpha_{i,j} \sum_{t_k \in \mathcal{H}_j(t)} \exp(-\omega_{i,j} \cdot (t - t_k)) \right) \tag{2}$$

In this case there is a parameter $\mu_i \geq 0$ for every product $i$ as well as $\alpha_{i,j} \geq 0$ and $\omega_{i,j}$ for every pair of products $(i,j)$. Therefore the intensity of every product has an individual baseline and can jump by different amounts, depending on which product is being ordered. Note, this definition does not pose a constraint as it could restore the simple Hawkes model by setting all parameters $\alpha_{i,j}$ for product pairs $(i,j)$ with $i \neq j$ to zero.
For both versions of the Hawkes process, the parameters can be learned from observed orders via Maximum Likelihood Estimation. The Log-Likelihood of the parameters given observed order sequences $\mathcal{H} = \{\mathcal{H}_1, ..., \mathcal{H}_N\}$ is [17]:

$$\mathcal{LL}(\boldsymbol{\mu}, \boldsymbol{\alpha}, \boldsymbol{\omega} | \mathcal{H}) = \sum_{i=1,...,N} \left( \sum_{t_k \in \mathcal{H}_i} \log \lambda_i\big(t_k | \mathcal{H}(t_k)\big) - \int \lambda_i\big(t | \mathcal{H}(t)\big) \mathrm{d}t \right) \tag{3}$$

with $\boldsymbol{\mu} = \big(\mu_i\big)_{i=1,...,N}$ , $\boldsymbol{\alpha} = \big(\alpha_{i,j}\big)_{i,j=1,...,N}$ , $\boldsymbol{\omega} = \big(\omega_{i,j}\big)_{i,j=1,...,N}$ , $\mathcal{H}(t) = \{\mathcal{H}_1(t), ..., \mathcal{H}_N(t)\}$.

The integral can also be calculated analytically using the following formula (see Appendix for a derivation of this formula):

$$\int_a^b \lambda_i(t) \mathrm{d}t = \mu_i \cdot (b-a) - \sum_{j=1}^{n} \left( \alpha_{i,j} \sum_{t_k \in \mathcal{H}_j} \frac{e^{\omega_{i,j}(t_k - b)} - 1}{\omega_{i,j}} \right) \tag{4}$$

Even though we have an analytical formula to calculate the Log-Likelihood, there is no analytical solution to its maximization. Therefore we used (stochastic) Gradient Descent to numerically minimize the negative Log-Likelihood.
Fundamental for our agents is that they are able to sample customer orders based on the order history and our trained Hawkes parameters. Our approach is to sample the inter-event times between two consecutive orders. Since sampling these inter-event times from a complex intensity function like the Hawkes intensity is difficult, we use a method referred to as thinning [14]. This means, at time $t$ we sample candidate times $t'$ for the next order from a constant upper bound intensity $\mu_0 \geq \max_{t' \geq t} \sum_{k=1}^{N} \lambda_k(t' | \mathcal{H}(t))$. For each candidate event $t'$ we sample the corresponding product type from a categorical distribution where each type $1 \leq k \leq N$ is chosen with probability $\frac{\lambda_k(t' | \mathcal{H}(t))}{\mu_0}$ while the candidate event will be rejected with probability $1 - \sum_{k=1}^{N} \frac{\lambda_k(t' | \mathcal{H}(t))}{\mu_0}$.
The inter-event times for a constant intensity function $\mu_0$ can be sampled from an exponential distribution $\mathrm{Exp}(1/\mu_0)$ [14]. This means, after each event $t_i$, the next event $t_{i+1}$ is sampled in the following manner:

**Sampling via Thinning**

**Input**: time $t_i$, history $\mathcal{H}(t_i)$ and end of episode $t_{max}$
**Output**: tuple $(t_{i+1}, j)$ of next order time $t_{i+1}$ and product $j$ or *None*

1. **Set** $t = t_i$
2. **Choose** $\mu_0 \in \mathbb{R}^+$ such that $\mu_0 \geq \sum_{k=1}^{N} \lambda_k\big(t'|\mathcal{H}(t_i)\big) \quad \forall t' \geq t$
3. **Sample** inter-event time $\tau \sim \text{Exp}(1/\mu_0)$
4. **Set** $t = t + \tau$
5. **If** $t > t_{\text{max}}$:
   return *None*
6. **Sample** $j \sim \text{Categorical}\Big(\frac{\lambda_1(t|\mathcal{H}(t_i))}{\mu_0}, ..., \frac{\lambda_N(t|\mathcal{H}(t_i))}{\mu_0}, 1 - \frac{\sum_{k=1}^{N}\lambda_k(t|\mathcal{H}(t_i))}{\mu_0}\Big)$
   **If** $j \in \{1, ..., N\}$:
   return $(t, j)$
   **Else**:
   Go to 3.

Note that if the sampled next order occurs after the end of the episode $t_{max}$, the sampling method simply returns *None*. After one tuple was sampled, it gets added to the history and the procedure is executed again to obtain the next tuple based on the updated history. This process is repeated until the end of one episode where the algorithm returns *None*.

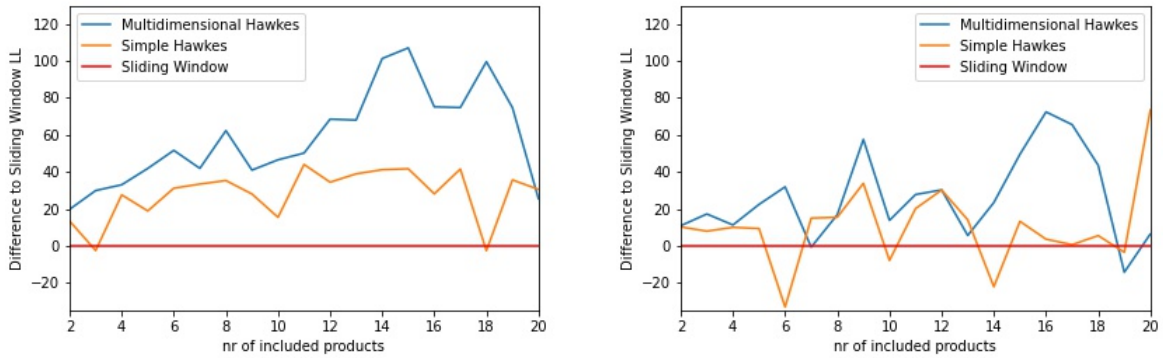
## 4.2 Hawkes Experiments on Real Data

### 4.2.1 Setup

In order to assess the capability of the Hawkes process to model consumer behaviour, we fit the simple version (1) and its extension (2) to the real sales data that was introduced in Section 2.3. The simple version of the Hawkes process will be abbreviated as SH and the Multidimensional Hawkes as MH. We train the two versions on the August data while the December data is used as a test set. The Log-Likelihood (3) serves as our performance measure for all experiments in this section. As the concrete value of the Log-Likelihood is difficult to interpret, we introduce a Sliding Window (SW) model as a baseline approach and consider the relative performance of the Hawkes processes compared to the simpler SW approach. In the SW model, the order intensity function $\lambda_{SW}\big(t|\mathcal{H}(t)\big)$ at time $t$ is simply the average number of orders per time step over the interval $[t - K, t[$. The scalar $K > 0$ is referred to as the window size and a detailed mathematical formulation of the intensity $\lambda_{SW}\big(t|\mathcal{H}(t)\big)$ can be found in the Appendix. For the experiments we choose a window size of $K = 10$.

### 4.2.2 Evaluation

In Figure 4 one can see an overview of the average performance of the Hawkes models compared to the baseline Sliding Window model.




(a) train set (b) test set

Figure 4: Comparison of Hawkes models and Sliding Window model on real sales data. Displayed is the average relative Log-Likelihood of order sequences compared to the baseline Sliding Window model on both the train and test set.

In both figures, the values on the x-axis refer to the number of included products.
In order to have the largest amount of data per number of included products at hand, the products with the highest number of orders on the whole data set were chosen. That means, if the number of included products in the figure equals four, we include the four most frequently ordered products in our analysis. The y-axis shows the average performance (Log-Likelihood) compared to the SW baseline model. The mathematical formulation can be found in the Appendix. A value larger than zero for the MH and SH model means that their average Log-Likelihood was higher in the corresponding experiment. In Figure 4a we consider the performance on the August train set while Figure 4b shows the evaluation on the December test set.
In Figure 4a we can observe that if we evaluate the MH model on the train data, it outperforms the simpler version of the Hawkes process SH and the baseline model SW. This is no surprise as the MH model is an extension of the SH model that can also take interdependencies between different products into account. Even if the orders of several products were independent of each other (that is, the assumption of the SH model would be correct), as mentioned in section 4.1 the MH model could recover the SH model.
There is also the tendency that the higher the number of included products gets, the more one can benefit from taking interdependencies into account as the gap between the MH model and the SH/SW models in Figure 4a increases.
For most numbers of included products the MH model also achieves a higher average Log-Likelihood on the test set , however the difference to the SH and SW model is smaller than on the train set. One reason for this observation might be that some of the learned interdependencies between several product pairs $(i, j)$ might not hold in the test period. In concrete terms, it could be the case that orders of product type $i$ may effect the frequency of orders of product type $j$ in August, while in December this dependency is not

present anymore as the consumer behaviour might be different in winter. Figure 4 aims to give an impression of the power of the MH approach as consumer model.
Next we will have a closer look at the intensity function.

### 4.2.3 Graphical Analysis

As an example, we consider the three most commonly sold products which we denote by 1,2 and 3. The parameters of the MH model obtained during the training period are the following:

$$\hat{\mu} = \begin{bmatrix} 0.109 \\ 0.074 \\ 0.359 \end{bmatrix} \qquad \hat{\alpha} = \begin{bmatrix} 1.006 & 0.000 & 0.045 \\ 0.000 & 0.895 & 0.015 \\ 0.415 & 0.101 & 0.362 \end{bmatrix} \qquad \hat{\omega} = \begin{bmatrix} 1.023 & 1.601 & 1.819 \\ 2.185 & 0.940 & 1.863 \\ 1.420 & 1.672 & 1.435 \end{bmatrix} \tag{5}$$

Figure 5: Learned Multidimensional Hawkes parameters on real sales data with three included products.

As a reminder, for $i, j \in \{1, 2, 3\}$ the quantity $\hat{\mu}_i$ corresponds to the base order rate of product $i$. The excitement parameter $\hat{\alpha}_{i,j}$ describes how much the intensity of product $i$ rises per order of product $j$ and $\hat{\omega}_{i,j}$ explains how fast the effect of an order of product $j$ on the intensity of product $i$ decays. The parameters allow to draw conclusions on the interdependecies between the different products. Due to the fact that $\hat{\alpha}_{1,2} = \hat{\alpha}_{2,1} = 0$ orders of product 1 do not influence the intensity of product 2 and vice versa in this learned model. While the intensity of product 1 almost entirely depends on the order history of product 1 (as $\hat{\alpha}_{1,1} \gg \max\{\hat{\alpha}_{1,2}, \hat{\alpha}_{1,3}\}$), the intensity of product 3 jumps higher per order of product 1 compared to an order of its own type ($\hat{\alpha}_{3,1} > \hat{\alpha}_{3,3}$). In Figure 6, we illustrate how the intensity function of each product gets affected by customer orders. For this, we pick a random day from the train set and consider at first only a small time interval.

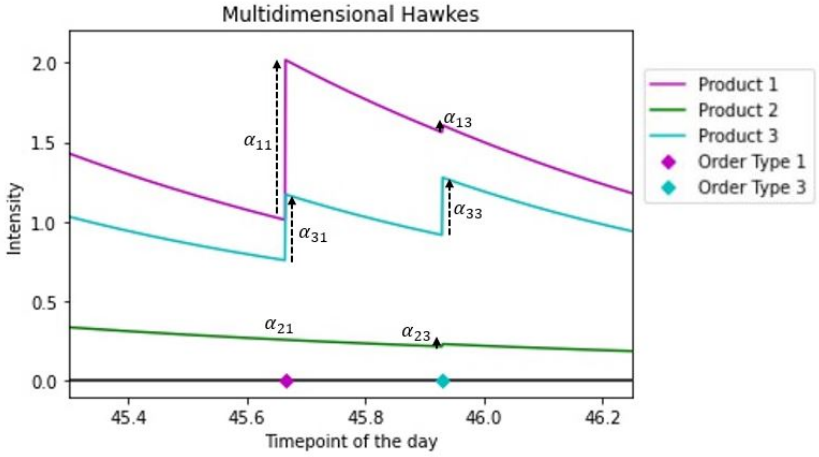
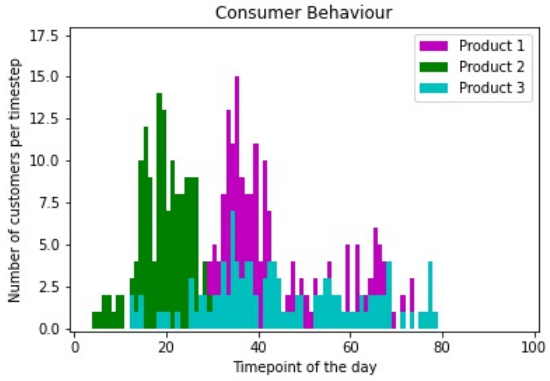
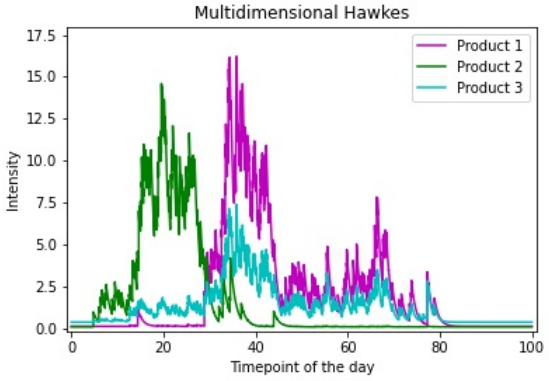



Figure 6: Intensity function of the Multidimensional Hawkes with trained parameters. Depicted is a small time interval of an example day from real sales data.

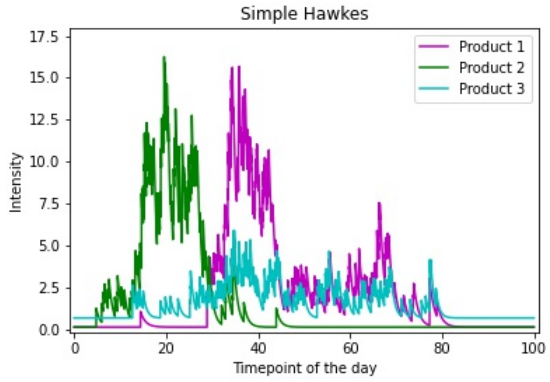
One can see that an order of product 3 has primarily an impact on its own intensity and only a very small effect on the intensity functions of product 1 and product 2. However, an order of product 1 leads to an increase in the intensity for both product 1 and product 3 while the intensity of product 2 remains unaffected.
Eventually, in Figure 7 we compare the histogram of real orders per product (Figure 7a) with the intensity function of the MH model and the SH model based on the learned parameters on the train set. On top, we further include the intensity of the SW baseline model. The depicted day is the same example day as used in Figure 6, but now the whole episode and not only a small interval is examined.



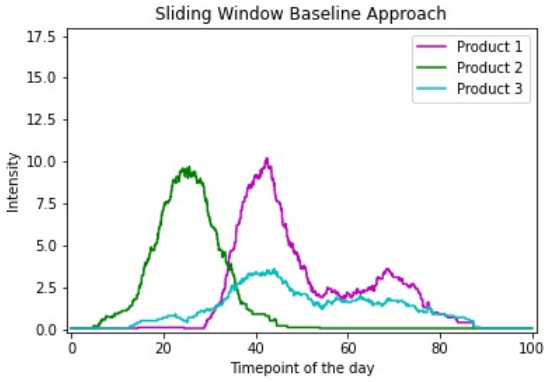

(a) histogram of real consumer behaviour




(b) intensity of Multidimensional Hawkes




(c) intensity of Simple Hawkes




(d) intensity of Sliding Window

Figure 7: Visualization of real customer orders on an example day and the corresponding intensity functions of the Multidimensional Hawkes model, Simple Hawkes model and Sliding Window model

Figure 7a shows that most orders of product 1 and product 3 appear at completely different parts of the day compared to orders of product 2. The time intervals with frequent orders of product 1 roughly coincide with peak order times of product 3. This behaviour accords with the observations we made when discussing the learned MH parameters from (5). Even though the MH intensity (Figure 7b) fails to perfectly match the real amount of orders for several time intervals of that specific day, the overall consumer behaviour seems to be mirrored by the intensity function. Remarkable is that by applying the optimal (learned) base rates, excitement parameters and decay parameters, peak times are

reflected as the intensity function is flexible enough to both rise and decay very sharply. The same holds for the intensity function of the Simple Hawkes process (Figure 7c). The Sliding Window intensity (Figure 7d) fails to reflect the real consumer behaviour especially at peak times as its intensity function is not flexible enough to rise sharply when the demand increases. Based on this simple graphical analysis of an example day, one cannot identify which version of the Hawkes model is more favourable, however our results presented in Figure 4 indicate that it is beneficial to select the MH model as internal prediction approach for our agents.

# 5 Monte Carlo Agent

As a baseline we implement a model-based agent that we call Monte Carlo (MC) agent. It uses Monte Carlo evaluation [18, 19], also called Monte Carlo search [20], to compare all possible actions at the current state. This agent evaluates these actions by performing them in the simulated environment and subsequently taking random actions for every step after, until the end of the episode. The orders in the simulation phase are sampled from a trained Hawkes model. The reward at the end of the day is used as the score for the initial action. By performing these simulations multiple times the obtained estimate of the reward gets more accurate. After all simulations are done, the agent greedily chooses the action with the highest estimated reward to act in the environment. We compare our more complex approaches to this baseline model.

# 6 Monte Carlo Tree Search

In this section we introduce the heuristic search algorithm Monte Carlo Tree Search (MCTS) [19] that plays the key role in the decision process of the upcoming agents. It enables the agent to plan ahead in a simulated environment first and then take the most promising action in the real world.

## 6.1 Theory

MCTS is a simulation based algorithm that aims to select the best action in the current state by building a search tree with each node assigning a value to a specific action. Based on these values MCTS uses a heuristic to expand towards the most promising actions. This way the tree is built more effectively than evaluating all possible actions in a brute-force way. MCTS has proven successful and has been used in Model-Based RL approaches with promising results [9, 10, 11, 19].
Our environment is not deterministic, which means that at any point in time the agent is uncertain about the exact future development of the environment. When building the tree, the state at the root of the tree is fixed, corresponding to the real-world state, but same action sequences can lead to different environment states. Therefore, in our case the values at the nodes represent the value of the sequence of actions leading to that node, or more precisely the estimated average value of all states that can be reached with the respective sequence of actions. Because we cannot calculate this value for all possible

states that an action sequence can possibly lead to, the values are based on Monte Carlo samples for that action sequence.

**Expanding the search tree** Figure 8a) shows a visual representation of an example Monte Carlo tree at time step $t = 30$. The root node, colored yellow, contains the information of the real-world state, at $t = 30$ in this example. All nodes contain the information of what action is needed to reach that node from the root and the current value of that node. This value is saved as to variables $x$ and $y$ (illustrated as $x/y$ in the figure). $x$ is the accumulated reward of all simulations performed at the node and from all descendants of that node. $y$ is the cumulative number of simulations of it and all descendant nodes. Actions are choices of how many products and which type to produce, represented by the icons with amounts. The time steps are not relevant to the agent, but were added in the visualization to show that the nodes do not have to correspond to a certain time step at a certain level in the tree. This is due to the fact that different product types have different production times.
Figure 8 illustrates the necessary steps in Monte Carlo Tree Search. Steps a), b), d) and e) are the common steps in standard Monte Carlo Tree Search, while c) is a step required due to the non-deterministic nature of our environment. Step e) illustrates updates after performing a step in the real-world based on a sufficiently expanded tree.

a) One central part of MCTS is selecting where to expand the tree in an order that is more efficient than brute-force. In the selection step the heuristic we use is to select the currently most promising node, which means the node with the highest cumulative reward divided by the number of simulations. This is node $A$ in the example figure.

b) In expansion the tree is expanded by adding a child node $B$ to the selected node. This node corresponds to an action that leads from the previous to the new node. This action is chosen randomly. In the case that the selected node already happens to be at the end of the day, it is not expanded and instead node $B$ will simply be set to the previously select node $A$.

c) In this step we sample a state that can be reached by performing the sequence of actions from the root $C$ to node $B$. The reason we sample this state starting from the root node is that it depends on the current information we have about our real-world state. This real-world state might have changed since we last simulated any of this node's ancestors. Also, the state at this node is not deterministic and can therefore only be approximated by a Monte Carlo sample. In order to sample this state we use our internal prediction model to estimate any potential orders from customers which affect the state.

d) Given this state the agent then performs a rollout of simulations. The rollouts simulate the environment until the end of the day using demand estimate from the internal model (in our case a Hawkes model) and a simulation policy, which determines what actions the agent performs in this simulation. In standard MCTS the simulation policy simply performs random actions chosen uniformly from the set of all possible actions. The cumulative reward at the end of the simulation divided

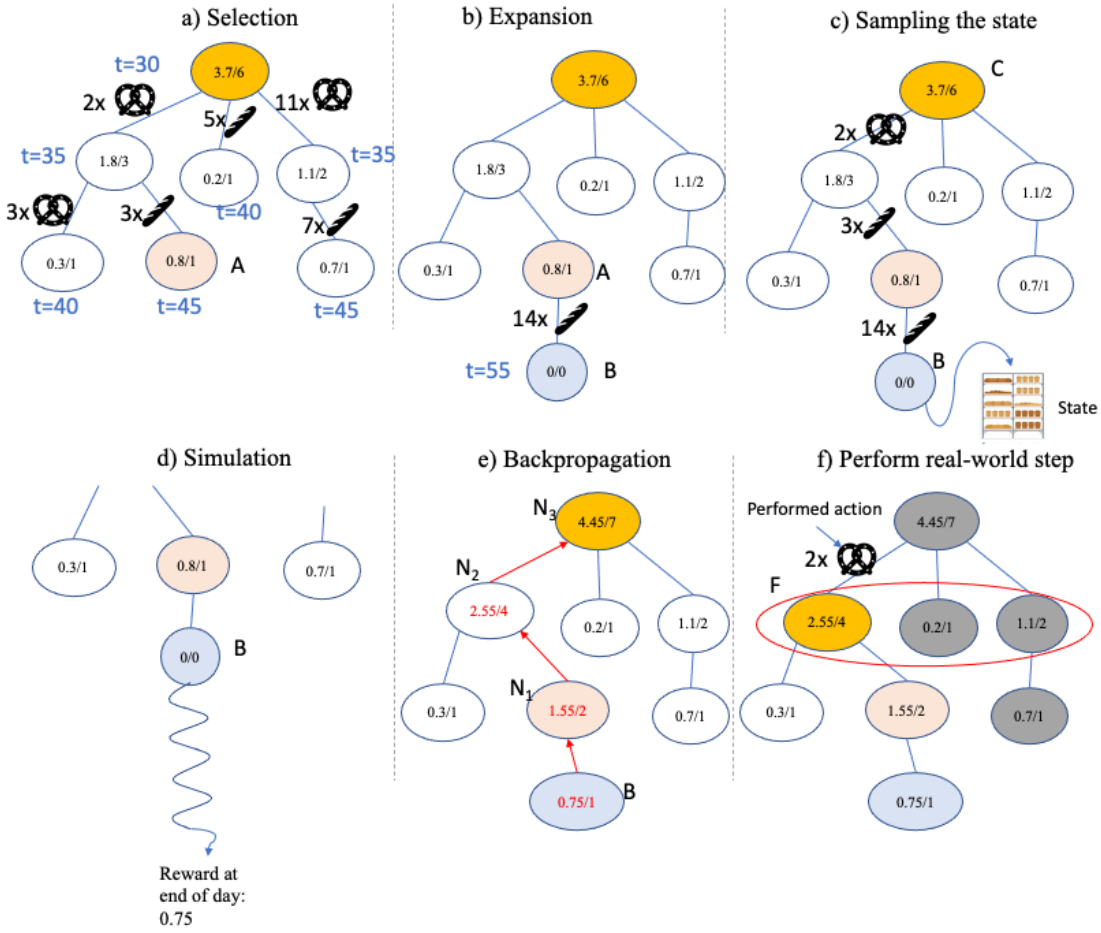



Figure 8: Steps of Monte Carlo Tree Search. Nodes save the estimated value of the actions leading to that node. The value consists of $x$ and $y$, where $x$ is the cumulative reward of simulations performed at that node and all successor nodes and $y$ is the total number of simulations performed at that node and all successor nodes. The estimated value of a node can be calculated by $x/y$.

by the number of simulations is saved as the value of the node $B$. In order to have more accurate estimates for the reward, multiple simulations can be executed for this node and the outcomes averaged.

e) The backpropagation step uses the results of the rollout to update information in all ancestors $N_i$ of the simulated node. The simulation numbers and accumulated rewards of the simulated node will be added as simulation for all nodes $N_i$. As a consequence, every $N_i$ will update its values using the number of new simulations and their accumulated rewards. Because we originally chose to expand from the current most promising node, this update causes the value at the most promising node to further approach a theoretically "correct" value.

Steps a) to e) are repeated as often as the computational budget allows, building up the search tree to become more accurate with more simulations. When the tree was expanded sufficiently it is used by the agent to choose which step to perform in the real world. As

shown in Figure 8f), the children of the current root node are compared, marked with a red circle in the figure. The one with the highest average value, marked with F in the figure, is selected as the new root node. The action belonging to that node is then performed in the real world. Now the state at F is changed to the observation in the real world. The nodes that cannot be reached anymore, marked in grey, are discarded and the agent continues, starting at step a) with the new root node F, further expanding the tree.

## 6.2 Improving Monte Carlo Tree Search

We implemented MCTS with the goal of outperforming the far simpler MC agent. Because we did not manage to find parameters that could do that with our first implementation, we experimented with different ways to improve the performance of MCTS in our bakery setting. In Section 6.2.1, 6.2.2 and 6.2.3 we present several modifications to the standard MCTS approach. Some but not all of them turned out to be successful.

**A note on the computational budget** In order to fairly compare the MC and MCTS agents we decided to do so under similar computational budget constraints. The individual simulations in both agents work the same. Then after all simulations were performed both types of agents choose actions in linear time with respect to the number of possible actions. Other computations should only have constant impact on the runtime and are negligible. Overall, computation times for both agents are therefore dominated by the number of simulations that are done for every step in the real world, i.e. for every decision an agent has to make. Therefore, we define the computational budget to be the fixed number of simulations each agent can perform per real-world time step. Figure 9 shows that in general apart from some minor irregularities a higher computational budget will increase the performance of the agent, as expected.

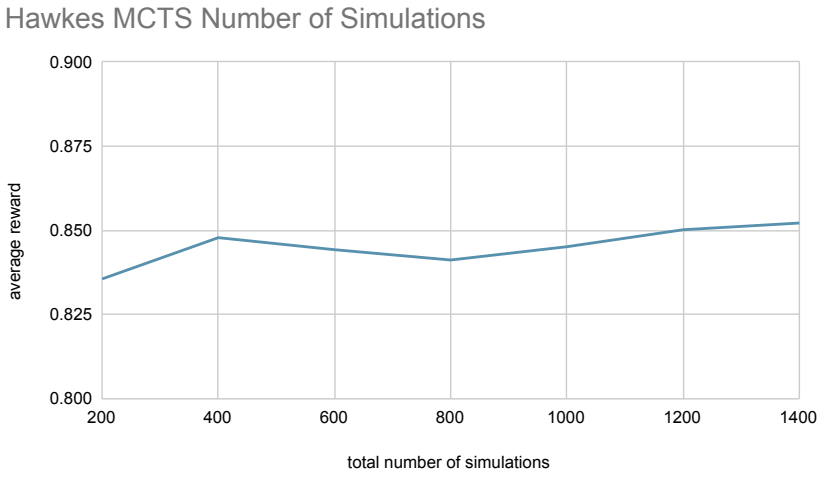



Figure 9: Average reward of the Hawkes MCTS agent for different numbers of simulations. Experiments were performed for simulation numbers from 200 to 1400, with intervals of 200. The resulting rewards were averaged over 40 episodes. In general, despite some irregularities a higher computational budget improves the performance.

### 6.2.1 Unsuccessful attempts to improve MCTS

The further away from the root a node is, the less accurate simulations of its state will be. Our first hypothesis was that these inaccuracies cause problems in the evaluation of our nodes and therefore the overall performance of our agent might deteriorate compared to the MC agent which only assigns values to states immediately following the current real-world state. As we perform steps in the real world, the root gets closer to this inaccurate node deep in the tree. However, the further we go in the real world, the further the value at that deep node most likely diverges from the real value. Therefore, our first approach was to rebuild the tree for every real-time step, essentially not remembering any past simulations that might have been done on less accurate data, because they were based on less information. This modification, however, did not improve the performance of our agent in our experiments. A similar approach based on the same idea was to discount the values of simulations based on how old the simulations themselves are. This discount would represent the unreliability of old node values. However, this also did not affect the performance of our agent.
Another method to balance between exploration and exploitation is forcing the root to try all possible actions, i.e. fully expanding the root, before the selection-expansion-simulation-backpropagation loop starts. However, this does not show great improvement under a reasonable computation budget.

### 6.2.2 Depth limit

Since Monte Carlo Tree Search is a biased searching method in which the tree tends to expand towards the most promising moves, it risks losing potential promising branches that have not been fully explored. Predictions far in the future from the current real state lose accuracy due to the inherent uncertainty of predictions of the future. The idea behind introducing a depth limit is to ensure that only a small fraction of the computational power is wasted on future time steps that might not result in accurate evaluations, no matter how often they are evaluated. The specific value of the depth limit is a balance between being able to plan far ahead in the future and not wasting resources on simulating states with high uncertainty.
In our implementation a depth limit of $n$ corresponds to only selecting nodes in the tree at a depth from 0 to $n$ in the selection phase of MCTS. The depth in this case is the relative distance to the current root, which corresponds to the current real-world state. Consequently, the current root has a depth of 0. Any new node that gets added into the tree in the expansion step can have a maximum relative depth of $n + 1$ (expanded nodes are direct children of the node that was determined to be the most promising node in the selection step). The depth limit only affects the maximum depth of the tree. The simulations in the simulation step are not affected and will still be performed until the end of the day, in order to provide a reward.
Empirically, our experiments showed that lower depth limits improve the performance of our MCTS agent. Figure 10 shows one such experiment. It can be seen that in this setting with 600 simulations per real time step there is not a large difference between depth limits between 0 and 3. However, starting from a depth limit of 4 the performance continually deteriorates. At depth limit 10 the performance is similar to not having any depth limit. While the tree can theoretically have a depth of up to 100, in practice most

trees will have a maximum depth around 10. The results also show that with lower depth limits MCTS manages to outperform the MC agent, as opposed to previous approaches without depth limit.

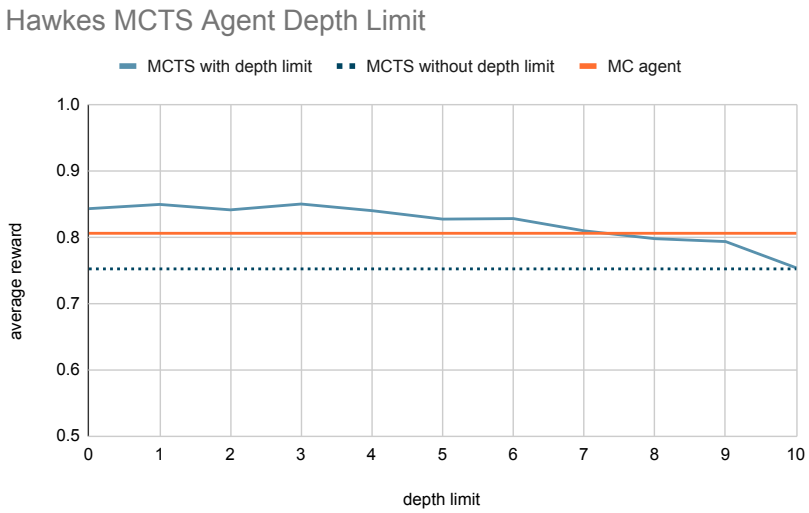



Figure 10: The blue solid line shows the average reward for different depth limits. The agent is a Hawkes MCTS agent with a computational budget of 600 simulations per time step. The reward was averaged over 40 episodes. The dotted line shows the average reward with no depth limit. The orange line shows the average reward achieved by the Monte Carlo agent without tree search. The experiments show that with lower depth limits MCTS can outperform the MC agent.

An MCTS agent with a depth limit of 0 behaves very similarly to the MC agent. The main difference is that the MC agent will loop over all possible actions in the current state and simulate the subsequent states, while the MCTS agent will use the heuristic to choose the most promising action. The results from Figure 10 indicate that applying this heuristic is already enough to improve the performance. Additionally, the fact that low depth limits larger than 0 perform similarly well suggests that the uncertainty in our predictions several actions into the future is not detrimental to the performance.

### 6.2.3 Learned simulation policy

MCTS lends itself to several modifications in which standard RL algorithms can be applied. Our approach was improving the simulation policy by replacing our previous policy, which simply performed random actions during simulation, with an agent that learns over time. Similar to one of the approaches used in [9] the idea is that an improved simulation policy makes the results of a simulation more accurate, therefore reducing the number of needed simulations and potentially improving the precision of the resulting score. Due to the large number of simulated experiences used for building the Monte Carlo tree, we decided to simply train the simulation policy on experience from the simulated environment. Consequently, this adds a training process which is required for our agent: Before the agent can be evaluated it needs to run for a number of training episodes in order to train the simulation policy.

For the RL policy we experimented on Deep Q Networks (DQN) [21] and Proximal Policy Optimization (PPO) [22]. Our empirical tests show no obvious difference and we decided that detailed hyperparameter-tuning and comparison of other algorithms is out of scope for this project. In PPO every policy gradient update step is limited, such that it does not change too far from the previous policy. We chose PPO because unlike a DQN policy, a PPO policy works with any combination of discrete and continuous observations and discrete and continuous action spaces.
In our original environment both the action space and the observation space are discrete: We can produce a discrete number of products from a discrete choice of product types; We can have a discrete number of products in our inventory and customers will purchase a discrete number of products. However, the real-world interpretation of actions and observations loses information when reduced to discrete vectors, where two different samples are completely independent. A time step with two orders of a product is more similar to a time step with three orders than to one with ten orders. The same holds for production decisions. By giving our action and observation spaces continuous vector shapes, we implicitly model this correlation. We chose to make the actions and observations continuous before passing them to the PPO algorithm and discretizing them when passing the output actions to the environment. For example, if the PPO algorithm chooses to produce 0.5 of a product that simply means 0.5 times the maximum number of products it can produce of that type. As we are using PPO, our simulation policy can work with these continuous spaces.
For the implementation of the RL algorithms we choose the library `stable-baselines3` [23]. It provides tested state-of-the-art implementations of many popular RL algorithms implemented in PyTorch, which we are familiar with.

# 7 Experiment Setup

**Environment configuration** We evaluate the agents in three different settings. Firstly, we evaluate the performance of our agents in a setting where the test consumer demand comes from the same distribution as the training demand. The second setting aims to evaluate how well the agents transfer to new environments. Here we train on the same distribution as in the first setting, however the test evaluations are performed in an environment where the demand has increased. This could be seen as a day where simply more customers decide to buy at the bakery than usual. The average number of products ordered are double the amount from the first setting. In the third setting we evaluate the agent on real-world data. Both training and testing is performed on customer orders provided by PreciBake from a real food establishment as described in Section 2.2. The training distribution comes from days in August while the tests are performed on data from December which includes Christmas day. Therefore, this setting also evaluates the robustness of the agent on changing external circumstances. All tests are performed on one environment with 2 different product types and one with 5 different product types. Table 1 in Section 8 illustrates the 6 different experiments.

**Hawkes prediction model** The Hawkes prediction model, as described in Section 4, is a priori trained independent from the other parts of the agents. Equipped with the

learned parameters, the agent can then act on train or test data. To provide comparable results we train the Hawkes model once for every environment setting (for the setting with double the amount of orders we do not need to train again, as we can use the trained model from the previous setting) and use the same parameters for all model-based agents.

**Threshold agent** We compare our agents with a rule-based baseline agent. This agent mimics the process a human baker would use to choose what to produce. The agent simply produces a fixed number of new products if the inventory for this product falls under a set threshold. This agent does not use any internal simulations and cannot be trained.

**Model-based agents** Choosing the computational budget for our agents, as it is described in Section 6.2, is a trade-off between agent performance and experiment duration. We choose a budget of 600 total simulations per real-world step.
We evaluate two agents that use MCTS. The first uses a random simulation policy and we will refer to it as the MCTS agent. The second one uses PPO as the simulation policy and we will refer to it as MCTS+PPO. Before evaluating the agent, MCTS+PPO is trained for 10 training days in the environment designated for training. During evaluation the PPO policy stays constant and does not learn further. Following our empirical experiments and discussion described in Section 6.2.2, we choose a depth limit of 2. This gives us a relatively low value with good performance while still allowing the tree to do some planning of the future.

# 8 Results & Discussion

| **Experiment** | **1a** | **1b** | **2a** | **2b** | **3a** | **3b** |
|---|---|---|---|---|---|---|
| **Training** | Poisson environment | | | | Real customer data *(August)* | |
| **Testing** | Identical to training | | Doubled product orders from training | | Real data *(December)* | |
| **Number of products** | 2 | 5 | 2 | 5 | 2 | 5 |

Table 1: Experiment configuration. Experiments 1 use the same distribution for training and testing, experiments 2 use a distribution with higher customer demand for testing, experiments 3 train and test on real data from two different months. Experiments a use a configuration with 2 different product types, experiments b with 5 different ones.

| **Experiment** | **1a** | **1b** | **2a** | **2b** | **3a** | **3b** |
|---|---|---|---|---|---|---|
| **Threshold** | 0.7672 | 0.5016 | 0.6876 | 0.5577 | 0.5527 | 0.4447 |
| **MC** | 0.8057 | 0.6842 | 0.7154 | 0.6323 | 0.6408 | 0.5008 |
| **MCTS** | 0.8353 | **0.7054** | **0.8755** | 0.6913 | **0.7618** | **0.6512** |
| **MCTS + PPO** | **0.8515** | 0.6946 | 0.8549 | **0.7070** | 0.7422 | 0.6360 |

Table 2: Results of experiments as described in Section 7. Specifics of experiment configuration can be seen in Table 1. Values are average rewards achieved evaluating in test setting. Values in bold represent the highest achieved average rewards in an experiment.

Table 2 shows our final experiments. The values are the average reward in evaluations of the four agents on the different settings. The result of the best performing agent for each experiment is marked as bold. Unsurprisingly, the threshold agent is outperformed by all other agents, as it only follows a very simple rule that cannot keep up with more sophisticated approaches. Furthermore, the Monte Carlo agent is outperformed by both the MCTS agent and MCTS+PPO. We explain this by the fact that MCTS is an extension of Monte Carlo search. If MCTS chose to only expand the root node, it would be very similar to Monte Carlo Search. Adding the heuristic of selecting useful states to simulate in MCTS as opposed to the brute-force equal simulation of all actions in the Monte Carlo agent, as well as the ability to do some planning of future actions apparently provides better results.
Comparing MCTS with and without an RL simulation policy shows that MCTS+PPO can outperform simple MCTS in the normal environment with two products and Experiment 2b, which has 5 products and different training and testing environments. However, MCTS+PPO performs worse in the other environments. The better results of MCTS+PPO in Experiment 1a and 2b show that improvement by using a smarter simulation policy is possible. A learned simulation policy can improve the accuracy of the

Monte Carlo samples allowing the agent to make more efficient use of its resources. For the same number of simulations, values in the tree might be more accurate. The lower performance of MCTS+PPO in experiments 2a, 3a and 3b could be related to the difference in training and test data. One would expect that the agent that has a simulation policy that learns over time is more sensitive to changes in the environment's distribution. The RL policy will learn to assume a distribution which is not true in the testing environment. The MCTS agent however does not contain any learned parameters except in the Hawkes predictions, and will therefore be less affected by a domain transfer. The difference between MCTS and MCTS+PPO in experiments 1b and 2a is relatively small, so is difficult to conclusively analyze.

In general it is difficult to compare rewards from different environments with each other. Small changes to the configuration can have large effects on the resulting rewards. Experiments 3a and 3b on the real data seem to generally be a more challenging environment. This does not seem to be the case in general for environment 2a and 2b, despite the difference in training and test distribution. It is possible that the higher amounts of orders in 2a and 2b counteract the difference in domain by allowing the agents the get more reward by making it easier to sell products without wasting them.

One advantage of the approach of having an agent that can learn, is that it is possible to train multiple agents and choose the best one. While all other agents' performance can only be improved by modifying the architecture or parameters, for example by increasing the computational budget, for the learning (PPO) agent different outcomes in the learning process can result in differently performing agents with the same parameters. Training multiple ones would allow to choose the best one in a given evaluation which would then be expected to perform better in general. Similarly, in our experiments we only evaluated agents that trained for 10 episodes. However, while MCTS agents cannot be trained, it is conceivable that training the MCTS+PPO agent for a longer time would allow it to improve further. Of course this additional training also takes more time, however as this can be done offline before the agent is deployed it does not affect online run-time.

# 9 Conclusion & Future Work

We have provided a way to improve agents in our baking environment with Model-Based Reinforcement Learning. While the non-determinism of the environment makes it difficult to use many model-based approaches out of the box, our results show that Model-Based RL is still feasible. The resulting agents are capable of planning by estimating the outcome of possible actions. Our agent lends itself to improvements in future work.
In the Hawkes model, we learn the optimal parameters considering the whole episode and do not distinguish between different parts of the day. This has the advantage that we do not have to provide any information about usual consumer behaviour for specific times. A possible extension would be to split the day into different parts with similar demand and learn parameters for each part individually.
Another approach would be to employ an LSTM to update the intensity function after each event. This would allow the model to capture even more complicated dependencies between product orders [24].
Additionally, our usage of a Hawkes model prioritizes transferability over long-term prediction accuracy. Using a prediction model that takes recorded past consumer behavior into greater account could improve agent performance in the main domain at the cost of transferability. This approach would also depend on the availability of past data.
There are also possible extensions associated with MCTS. An RL Policy used to decide the real action given a Monte Carlo Tree could make a smarter trade-off between exploration and exploitation than the current greedy method. Furthermore, improvements could be made in choosing which actions are expanded from a given node, which is currently random. Additionally, a regression network which provides a quick but not as accurate value for a given node could be used in conjunction with the rollouts similar to the approach taken in AlphaGo [9], which could reduce the number of needed simulations.
The approach used in the AlphaZero [10] algorithm expands on the previous two improvements in a way that seems promising for our setting. Firstly, it combines the policy used to make decisions in the real-world with the network that estimates the value of a node in the tree into a single neural network. Secondly, it forgoes simulations, only using the constantly learning neural network to estimate the value of the node. These improvements allow to expand the search tree more efficiently.
The non-determinism and branching factor of the tree make many model-based approaches difficult. Hierarchically structuring the world into nodes is inherently discrete, which discards some information we have about the similarity of actions and creates a large search space. Model-based approaches that do not rely on search trees, such as Imagination Augmented agents [25], or that use some kind of function approximation to represent the state could provide big improvements. Along similar lines is an approach used in the MuZero algorithm [11], an extension of AlphaZero. In MuZero learning the model is not separate from the MCTS, which by default expects an already learned model of the environment. Instead, neural networks are used to directly learn the dynamics of the environment in conjunction with the MCTS process. This approach could prove useful in solving inherent challenges with prediction of the environment.
Overall, Model-Based Reinforcement Learning seems promising for the bakery optimization problem and there are many extensions possible to improve on our existing work in the future.

# Bibliography


[1] DeepMind. *AlphaGo*. URL: https://deepmind.com/research/case-studies/alphago-the-story-so-far/ (visited on 02/03/2021).

[2] Michael Janner et al. *When to Trust Your Model: Model-Based Policy Optimization*. 2019. arXiv: 1906.08253 [cs.LG].

[3] Murat Karacabey et al. *Domain Transfer for Reinforcement Learning Agents*. 2020.

[4] Rupal Rana and Fernando S. Oliveira. "Dynamic pricing policies for interdependent perishable products or services using reinforcement learning". In: *Expert Systems with Applications* 42.1 (2015), pp. 426–436.

[5] Ahmet Kara and Ibrahim Dogan. "Reinforcement learning approaches for specifying ordering policies of perishable inventory systems". In: *Expert Systems with Applications* 91 (2018), pp. 150–158.

[6] Rui Wang et al. "Solving a Joint Pricing and Inventory Control Problem for Perishables via Deep Reinforcement Learning". In: *Complexity* 2021 (2021), p. 6643131.

[7] Ana Soares et al. "Using reinforcement learning for maximizing residential self-consumption â" Results from a field test". In: *Energy and Buildings* 207 (2020), p. 109608.

[8] Afshin Oroojlooyjadid, Lawrence V. Snyder, and Martin Takác. "Applying Deep Learning to the Newsvendor Problem". In: *CoRR* abs/1607.02177 (2016). arXiv: 1607.02177.

[9] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". en. In: *Nature* 529.7587 (Jan. 2016), pp. 484–489.

[10] David Silver et al. "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play". In: *Science* 362.6419 (Dec. 7, 2018), pp. 1140–1144. pmid: 30523106.

[11] Julian Schrittwieser et al. "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model". In: *Nature* 588.7839 (7839 Dec. 2020), pp. 604–609.

[12] OpenAI. *OpenAI Gym documentation*. URL: https://gym.openai.com/docs/ (visited on 02/01/2021).

[13] Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. *Model-based Reinforcement Learning: A Survey*. 2020. arXiv: 2006.16712 [cs.LG].

[14] Abir De, Utkarsh Upadhyay, and Manuel Gomez-Rodriguez. *Temporal Point Processes*. Lecture Script (Saarland University). Winter 2018-19.

[15] Wolfram MathWorld. *Conditional Intensity Function*. URL: https://mathworld.wolfram.com/ConditionalIntensityFunction.html (visited on 02/05/2021).

[16] Stephan Günnemann. *Machine Learning for Graphs and Sequential Data (IN2323)*. Lecture Script (Technical University of Munich). delivered 06/17/2020.

[17] Mehrdad Farajtabar et al. *A Continuous-time Mutually-Exciting Point Process Framework for Prioritizing Events in Social Media*. 2015. arXiv: 1511.04145 [cs.SI].

[18] B. Abramson. "Expected-Outcome: A General Model of Static Evaluation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12.2 (Feb. 1990), pp. 182–193.

[19] Guillaume Chaslot et al. "Progressive Strategies for Monte-Carlo Tree Search". In: *New Mathematics and Natural Computation* 04 (Nov. 1, 2008), pp. 343–357.

[20] Gerald Tesauro and Gregory Galperin. "On-Line Policy Improvement Using Monte-Carlo Search". In: *Advances in Neural Information Processing Systems*. Ed. by M. C. Mozer, M. Jordan, and T. Petsche. Vol. 9. MIT Press, 1997, pp. 1068–1074.

[21] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: `1312.5602 [cs.LG]`.

[22] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *ArXiv* abs /1707.06347 (2017).

[23] Antonin Raffin et al. *Stable Baselines3*. `https://github.com/DLR-RM/stable-baselines3`. 2019.

[24] Hongyuan Mei and Jason Eisner. *The Neural Hawkes Process: A Neurally Self-Modulating Multivariate Point Process*. 2017. arXiv: `1612.09328 [cs.LG]`.

[25] Sébastien Racanière et al. "Imagination-Augmented Agents for Deep Reinforcement Learning". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017, pp. 5690–5701.

# Appendix

## Sampling Algorithm of the Poisson Model

**Poisson Consumer Model Algorithm**

**Input:** List of means $\boldsymbol{\mu}$, time step $t \leq 100$.
**Output:** Number of orders for time step $t$.

1. **Set** $l = \text{length}(\boldsymbol{\mu})$
2. **Identify** which entry $\mu_i$ of $\boldsymbol{\mu}$ corresponds to the part of the day that contains $t$.
3. **Sample** $x \sim \text{Pois}\left(\frac{\mu \cdot 100}{l}\right)$
4. **Return** $x$

Remark: In our implementation, one day consists of 100 time steps.

## Calculation of Hawkes integral

For given order sequences $\mathcal{H}_1, ..., \mathcal{H}_N$ for the products $1, ..., N$, the integral of the intensity of product $i$ is:

$$
\begin{aligned}
\int_{\mathrm{a}}^{\mathrm{b}} \lambda_i(t)\,\mathrm{d}t &= \int_{\mathrm{a}}^{\mathrm{b}} \mu_i + \sum_{j=1}^{N} \left( \alpha_{i,j} \sum_{t_k \in \mathcal{H}_j(t)} \exp\left(-\omega_{i,j} \cdot (t - t_k)\right) \right) \mathrm{d}t \\
&= \int_{\mathrm{a}}^{\mathrm{b}} \mu_i\,\mathrm{d}t + \sum_{j=1}^{N} \left( \alpha_{i,j} \int_{\mathrm{a}}^{\mathrm{b}} \sum_{t_k \in \mathcal{H}_j(t)} \exp\left(-\omega_{i,j} \cdot (t - t_k)\right) \mathrm{d}t \right) \\
&= \mu_i \cdot (\mathrm{b} - \mathrm{a}) + \sum_{j=1}^{N} \left( \alpha_{i,j} \int_{\mathrm{a}}^{\mathrm{b}} \sum_{t_k \in \mathcal{H}_j} \exp\left(-\omega_{i,j} \cdot (t - t_k)\right) \cdot \mathbb{1}(t_k < t)\,\mathrm{d}t \right) \\
&= \mu_i \cdot (\mathrm{b} - \mathrm{a}) + \sum_{j=1}^{N} \left( \alpha_{i,j} \sum_{t_k \in \mathcal{H}_j} \int_{\mathrm{a}}^{\mathrm{b}} \exp\left(-\omega_{i,j} \cdot (t - t_k)\right) \cdot \mathbb{1}(t_k < t)\,\mathrm{d}t \right) \\
&= \mu_i \cdot (\mathrm{b} - \mathrm{a}) + \sum_{j=1}^{N} \left( \alpha_{i,j} \sum_{t_k \in \mathcal{H}_j} \int_{t_k}^{\mathrm{b}} \exp\left(-\omega_{i,j} \cdot (t - t_k)\right) \mathrm{d}t \right) \\
&= \mu_i \cdot (\mathrm{b} - \mathrm{a}) - \sum_{j=1}^{N} \left( \alpha_{i,j} \sum_{t_k \in \mathcal{H}_j} \frac{e^{\omega_{i,j}(t_k - \mathrm{b})} - 1}{\omega_{i,j}} \right)
\end{aligned}
$$

Here, $\mathcal{H}_j(t)$ denotes all orders in sequence $\mathcal{H}_j$ that have occurred before time $t$ and $\mathbb{1}$ denotes the indicator function.

## Baseline Sliding Window Model

In the SW model, the order intensity function of product $j$, namely $\lambda_{SW,j}\big(t|\mathcal{H}(t)\big)$ at time $t$ is simply the average number of orders per time step over the interval $[t-K, t[$ with $K > 0$. For some product $j$ and time $t > 0$ $\mathcal{H}_j(t) = \{t_1, ..., t_n\}$ denotes the set of past orders of product $j$ and the Sliding Window intensity reads as

$$\lambda_{SW,j}\big(t|\mathcal{H}_j(t)\big) = \left(\frac{1}{K}\sum_{t_l \in \mathcal{H}_j(t)} \mathbb{1}[t - t_l \leq K]\right)\mathbb{1}[t > K] + \frac{|\mathcal{H}_j(t)|}{t}\mathbb{1}[t \leq K] \tag{6}$$

where $\mathbb{1}$ denotes the indicator function and $|\mathcal{H}_j(t)|$ the cardinality of set $\mathcal{H}_j(t)$.

## Average relative Log-Likelihood compared to SW model

Denote as $LL^{MH}_{k,d}$ the Log-Likelihood of the order sequence for day $d$ with $k$ included products for the trained MH model (and analogously $LL^{SH}_{k,d}$ for the SH model), then with $D$ denoting a set of days we define by

$$\hat{LL}^{MH}_{k,D} := \frac{1}{|D|}\sum_{d\in D} LL^{MH}_{k,d} - \frac{1}{|D|}\sum_{d\in D} LL^{SW}_{k,d} \tag{7}$$

$$\hat{LL}^{SH}_{k,D} := \frac{1}{|D|}\sum_{d\in D} LL^{SH}_{k,d} - \frac{1}{|D|}\sum_{d\in D} LL^{SW}_{k,d} \tag{8}$$

the average relative Log-Likelihood compared to the SW model.