DATA INNOVATION LAB

TECHNISCHE UNIVERSITÄT MÜNCHEN

Project Report

Domain Transfer for Reinforcement Learning Agents

Authors:	Murat Karacabey, Ahmad Qasim, Abdul Moeed, Sebastian Oehme
Mentors:	M.Sc. M. Sundholm, M.Sc. H. Belhassan
Co-Mentor:	M.Sc. Michael Rauchensteiner
Project Lead:	Dr. Ricardo Acevedo Cabra
Supervisor:	Prof. Dr. Massimo Fornasier
Submission Date:	07.02.2020

Contents

1 Introduction **1**

2 Theory **2**
2.1 Sequential Decision Making 2
2.2 Domain Transfer 5

3 Reinforcement Learning Environment **8**
3.1 OpenAI Gym 8
3.2 Precibake Environment 9

4 Consumer Model **11**
4.1 Public Data 11
4.2 Synthetic Data 12

5 Agents **14**
5.1 Baseline 14
5.2 Deep Deterministic Policy Gradient (DDPG) 14
5.3 Constraint Optimization with Demand Prediction 16

6 Results & Discussion **21**
6.1 Training 21
6.2 Metric Benchmark 24

7 Conclusion **26**

Bibliography **28**

1 Introduction

Often it is taken for granted, but managing inventory to meet demand is quite complex. Many consumers might have already experienced it when standing in front of empty shelves in a supermarket or having seen piles of fresh bread rolls in a bakery just before closing time. Besides the obvious interest for a business in the food industry to meet the consumer demand and maximize sales for profits, it has to balance between a product's freshness and potential waiting time for consumers, while becoming more sustainable through minimizing waste.

To address this inventory management problem, Reinforcement learning (RL) agents seem a feasible approach, as they have recently shown impressive performance in learning complex tasks such as controlling robots [KBP13] and playing video games [ER98; Mni+13]. However, training such agents for real systems is complicated in practice. Since, in order to learn, the agent needs to act within the environment. Further, most RL agents are quite data hungry and might require a lot of training episodes before performing on par with humans on a given task. Training an agent in the real physical environment is therefore most often neither safe or feasible. That is why, most RL agents are trained and tested in simulated environments before they get deployed in the real world. Nonetheless, there is no guarantee that an agent trained in the simulation will perform well in the real physical domain. Since the dynamics of the simulation deviate from the dynamics of the physical world, agents will most likely under-perform or fail completely once deployed in the new domain. An active area of research is to generalize RL agents trained in randomized, simulated environments [LK01; Tob+17] to also perform well when deployed into a new environment.

The goal of this project is to develop a method to train RL agents that can perform in a real food inventory control problem even if they were only trained on simulated data. We analyze the related work and theoretical background in Chapter 2. In Chapter 3, we introduce an environment to model inventory control for the food industry. In Chapter 4 and 5, we explain our approach for modeling consumer demand and our agents respectively. Subsequently, we evaluated the results discuss them in Chapter 6. Finally, we draw our conclusions and point out future work in Chapter 7.

2 Theory

In this chapter we first provide an overview of the well-developed and approved aspects of reinforcement learning by the research community. Afterwards, we augment our analysis with references to ongoing research in domain transfer and how it can be utilized towards creating RL agents that can be applied in the real world.

2.1 Sequential Decision Making

Inventory control is one of many examples where decisions are made in stages. These kinds of situations are referred to as Sequential Decision Making (SDM) [BSW89]. Here an agent aims to achieve a predefined goal and identify the decision rule by interacting with its environment, shown in Figure 2.1. As a consequence of its interactions, the agent receives observations or feedback from the environment, then adapts to make appropriate next actions to it.

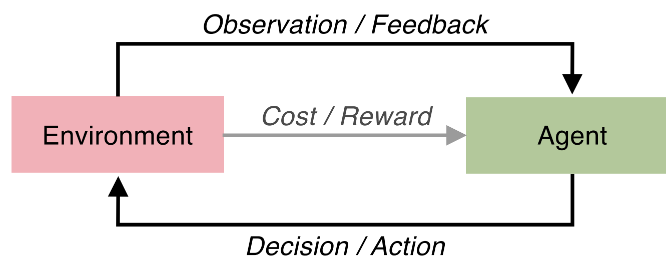


Figure 2.1: Sequential interactions between agent and its environment.

At each decision epoch, a decision is to be made, according to an appropriate evaluation of the sequence of states and actions generated by the agent (called trajectory). This is done by evaluating trajectories only locally in time with the current state x_k, action u_k and following state x_{k+1}. Such a local evaluation is known as the cost-to-go function g_k or the reward function r_k in the RL literature [S+98]. Commonly, the reward function is assumed to be bounded, i.e., $|g_k| < 1$.

The agent's objective is to minimize a certain cost (or to maximize a reward) in accompanying situations. A key idea here is that the agent's actions must balance the

present costs with future expected costs. RL and Dynamic Programming (DP) [Bel66] techniques capture this trade off.

When there is no noise or disturbance in the system, the problem is simply deterministic. However, most systems of interest are almost always uncertain. A successive state is then statistically dependent on the current state and a chosen action. We can either describe SDM as a discrete-time dynamic system

$$x_{k+1} = f_k(x_k, u_k, w_k) \text{ with } k = 0, 1, ..., N-1, \tag{2.1}$$

where

k indexes discrete time

x_k is the state of the system at time k

u_k is the control to be selected at time k

w_k denotes the uncertainty

N is the finite horizon

f_k is function that fully describes the systems

or with a probabilistic transition model as a Markov Decision Process (MDP) $\{S, A, p, g, T\}$, where

S is a finite set of states

A is a finite set of actions

$p(x_{k+1}|x_k, u_k)$ is the transition probability

$g(x_k, u_k, x_{k+1})$ is the cost function defined on state transition (x_k, u_k, x_{k+1})

N is the finite horizon.

For a discrete-time dynamic system the problem is an optimization of the expected cost

$$\mathbb{E}_{w_k}\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k)\right]. \tag{2.2}$$

The goal of MDP is to find an optimal decision rule or policy π^* for arbitrary initial state s that minimizes the so called Value function V. V is defined as the expectation of evaluations of trajectories following sequence of actions $\mu := \{u_0, ..., u_{N-1}\}$ given s:

$$V(s, \mu) := \mathbb{E}_{p_h(s,\mu)}\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, x_{k+1}) \middle| x_0 = s\right] \tag{2.3}$$

where a trajectory starting with s becomes a random variable with its probability density function:

$$p_h(s,\mu) = \prod_{k=0}^{T} p\left(x_{k+1}|x_k,u_k\right)\Big|_{x_0=s} \tag{2.4}$$

Putting this in terms of a policy which maps the state-dependent feature space to its admissible action space the Value function can be reformulated[1] to

$$V_\pi(s) := \mathbb{E}_{\substack{p(w_k)\\ \pi_k(u_k|x_k)}} \left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k,u_k,w_k) \middle| x_0 = s \right] \tag{2.5}$$

Therefore, the optimal Value function is

$$\begin{aligned} V^*(s) &= \min_{\mu_k} \mathbb{E}_{p_h(s,\mu)} \left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k,u_k,x_{k+1}) \middle| x_0 = s \right] \\ &= \min_{\pi} V_\pi(s). \end{aligned} \tag{2.6}$$

Although RL and DP both share the same working principles they have different naming conventions. Furthermore, the key difference between classic DP and classic RL is that the first assumes full knowledge of the MDP but is of limited utility because of great computational expense when the problem faces many states and actions (Curse of Dimensionality [Bel66]).

The fundamental concept for DP that deals with solving SDM problems is known as the "Principle of Optimality" [Bel66]. The main idea behind it is that, for solving SDM problems an optimal action that minimizes the value function at the current state $V_k(x_k)$ will simultaneously minimize the Value function that involve all subsequent states $V_{k+1}(x_{k+1})$. In practice the DP algorithm is used to compute an optimal policy by backward recursion. An optimal policy can be constructed in step-wise, first constructing an optimal policy for the "tail sub-problem" involving the last state and then continuously extending the "tail sub-problem" until an optimal policy

$$\pi_k^*(x_k) \in \operatorname*{argmin}_{u_k} \mathbb{E}_{p(w_k)} \left[g_k(x_k,u_k,w_k) + V_{\pi_{k+1}^*}(x_{k+1}) \right] \tag{2.7}$$

is constructed for the entire problem. Hence, leads to the optimal value function

$$\begin{aligned} V_k^*(x_k) &= \min_{\mu_k} \mathbb{E}_{p(w_k)} \left[g_N(x_N) + \sum_{t=k}^{N-1} g_t(x_t,u_t,w_t) \right] \\ &= \min_{u_k} \mathbb{E}_{p(w_k)} \left[g_k(x_k,u_k,w_k) + V_{k+1}^*(x_{k+1}) \right]. \end{aligned} \tag{2.8}$$

[1] For further details we refer to the lecture notes of Approximate Dynamic Programming and Reinforcement Learning from Dr. Hao Shen, TUM Department of Electrical and Computer Engineering.

So far we have described SDM for finite horizon problems. It it also possible to extend this to infinite horizon problems, e.g., a case without a terminal state. However, this is outside of the scope of this project.

As mentioned earlier a key difference in RL is the lack of a model. This causes a fundamental trade-off. The agent seeks to find the right balance between experimenting within the environment to gain more knowledge on the dynamics (exploration) and exploiting actions that are already known to have a high chance of reward in certain situations. Many RL agents makes use of incremental updates to the Value function with Temporal-difference (TD) learning [SB87]. TD methods can update (←) the Value function at every step, e.g., in the simple method TD(0):

$$V(x_k) \leftarrow V(x_k) + \alpha[g(x_k, u_k, x_{k+1}) + \gamma V(x_{k+1} - V(x_k)] \tag{2.9}$$

where

α is the learning rate,

$g(x_k, u_k, x_{k+1})$ is the observed cost and

γ is a discount factor.

The learning rate and discount factor influence how much the new observations are accounted for and how much information from future states are propagate to the current state respectively. Through sampling TD-methods are able to find a optimal policy, too. This is convenient as it is possible for a agent to learn a certain task even if the underlying model is unknown. As one can imagine sampling might be a tedious, time-consuming task. Therefore, it's mostly done in with the help of a simulator. In the simulator we can design the environment and set up constrains for the agent to start learning for mastering a given task. Afterwards, we are able transfer the learnings to the real world, e.g. a robot performing some task inside a warehouse.

2.2 Domain Transfer

As the title of the project suggests, the core problem to be solved is domain transfer – the ability of a reinforcement agent to generalize to unseen environments. Domain transfer in RL is a special case of the more general problem of "transfer learning" in machine learning. Transfer learning is defined as the "improvement of learning in a new task through the transfer of knowledge from a related task that has already been learned" [TS10].

In the RL setting, a number of approaches have been proposed to achieve domain transfer. For example, for image-based tasks such as game playing using pixels, an

image-to-image translation scheme using generative adversarial networks (GANs) has been employed [GG18]. A more sophisticated approach uses hierarchical reinforcement learning to learn sub-goals that are common in the source and target tasks [SOL18].

For this project, we want to generalize the RL agent's learning from simulations to the real world. A technique that deals specifically with this problem is "Sim2Real". The key idea is to train the RL agent in different configurations/parameters of the environment on each episode, in order to make it robust to unforeseen environments. This is known as "dynamics randomization" – randomly sampling parameters that define the dynamics of the environment. Formally, the objective is to maximize the expected reward over a distribution of environment dynamics:

$$\mathbb{E}_{\mu \sim \rho_\mu} \left[\mathbb{E}_{\tau \sim p(\tau|\pi,\mu)} [\sum_{k=0}^{N-1} g(x_t, u_t)] \right] \tag{2.10}$$

where μ is the set of parameters that can parameterize the dynamics of the environment, ρ_μ is the distribution of dynamics and $p(\tau|\pi, \mu)$ is the likelihood of a trajectory $\tau = (x_0, u_0, x_1, ..., u_{N-1}, x_N)$ given a certain policy π and parameters μ.

(a) Cartpole environment with default parameters.

(b) Cartpole with randomized/altered parameters i.e. pole height

Figure 2.2: Example of modified environment dynamics in OpenAI's Gym Cartpole.

A simple example of dynamics randomization is given in Figures 2.2a & 2.2b. 'Cartpole', an environment implemented in the OpenAI Gym package (more details in chapter 3), is often used as a benchmark for RL algorithms. Here, the task is to balance the pole by moving the cart to the left or right. Let us suppose that we want to train an agent that can balance the pole, not just with the given height, but on different pole heights. RL algorithms such as Q-learning can be used to learn pole balancing on the standard environment (Figure 2.2a), but they fail to generalize when the pole height is changed (Figure 2.2b). This is where dynamics randomization is useful – we can use the same algorithm (Q-learning), only this time, on each episode we must sample a

different pole height to learn on. Given that the agent has seen a wide variety of pole heights during training, it should be able to generalize to a pole which it has not seen during training.

The next Chapter provides an overview of our environment, and how it relates to dynamics randomization.

3 Reinforcement Learning Environment

As previously mentioned, RL agents are generally not employed right away in the real world. Rather, they are first trained in a virtual environment that models the real world. This means that the fidelity of the environment can have a major impact on the agent's performance in the real world. As such, various RL software frameworks e.g. OpenAI Gym [1], Google's Dopamine [2], Keras-RL [3] etc. have been introduced, which help the developers by providing out-of-the-box environments and tools for core RL algorithms. For this project, we use OpenAI's Gym package to build our RL environment.

3.1 OpenAI Gym

Gym is a toolkit provided by OpenAI, and made available to the developers as a Python library. The `gym` package is a collection of test problems called environments. `gym` environments share a common interface which makes it easier to write algorithms which do not have to be modified for different environments. The common interface is modeled after the classic "agent-environment loop" depicted in Figure 2.1. The main components of the common interface are:

- `step()` takes the action as input and marks a single time-step in the environment.
- The output object of the step function, which includes:

  observation – environment specific object which represents the state,

  reward – is received as a result of the action,

  done – a Boolean representing whether the agent has reached the terminal state of the environment or not, and

  info – a debugging object used for diagnostics.
- Action – which is carried out by the RL agent given the output object.
- Observation and action space: Environment dependent objects which represent the valid values for environment state and actions respectively.

[1] https://gym.openai.com/
[2] https://opensource.google/projects/dopamine
[3] https://keras-rl.readthedocs.io/en/latest/

3.2 Precibake Environment

gym also provides an interface to create custom environments – PreciBake used this to create the inventory control environment for this project. As explained in Section 3.1, given the environment state i.e. observation and reward of the last time-step, the agent generates the action which should be carried out during a particular time-step. As an example for inventory control we are considering a bakery environment in this project. Here, the action is comprised of the information about which type of product has to be baked and also its quantity. In the real-world, ovens are commonly used to bake one type of product in batches. Therefore, the agent's action is limited to only produce one type of product and its amount at a given time-step. As shown in Figure 3.1, the PreciBake environment state is composed of the following components:

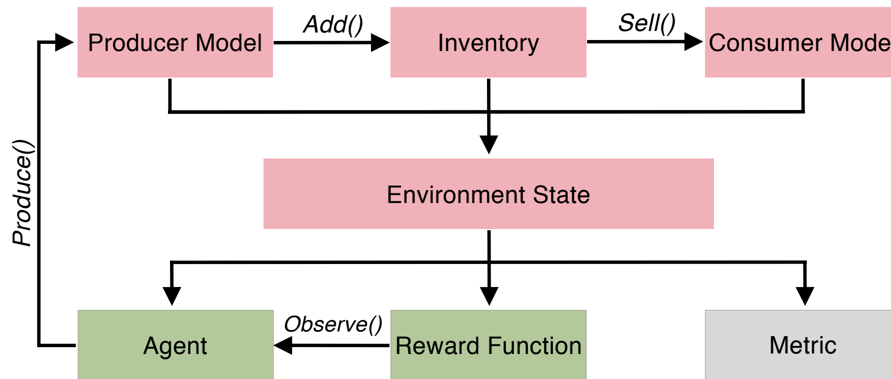


Figure 3.1: The PreciBake environment represented as a graph.

- **Producer model** This component simulates the bakery oven. The input to the producer model is the action which is to be performed during the current time-step. Each product has a production time. After receiving the action, the producer model adds the amount of the respective product specified by the agent to the inventory, after the production time of the specified product has elapsed. While, the production model waits for the production time to elapse, it does not process any actions from the agent.

- **Inventory** represents the bakery's product display. The Producer model adds produced items to the inventory. They stay in the inventory until they are sold to a consumer.

- **Consumer model** models the consumer behavior. In the real world, this can be influenced by multiple factors e.g. freshness of the items in the inventory, time of

the day, the day being a holiday/weekend etc. The consumer model has to be a realistic representation, which takes at least some of these factors into account. When an order is made by the consumer model, the order is either satisfied by the inventory at the moment or the delivery, or it is removed from the queue in the next time-step. However, it is noteworthy that such missed orders affect the metrics negatively (we describe this in detail below).

The reward function can be used as a representation of how well the agent is doing within the environment. However, the reward function can be agent specific i.e. different agents can have different reward functions. Hence, we use two consistent metrics in order to able to compare them with each other. As shown in Figure 3.1, these metrics can be derived from the last state of the environment. The metrics used in the PreciBake environment are given hereby. The sales ratio s_{ratio}, which measures that how good the agent is at serving incoming orders from the consumers. A higher sales ratio means that higher number of orders were fulfilled. An agent with a sales ratio of one would have completed all the orders that it received from the consumers.

$$s_{ratio} = \frac{o_f}{o_f + o_u} \tag{3.1}$$

where,
o_f = number of fulfilled orders
o_u = number of unfulfilled orders
The products ratio p_{ratio}, which measures the average age of all products, within the inventory. A higher products ratio means that, the products within the inventory were more old or less fresh, when they were sold to the consumer.

$$p_{ratio} = \frac{\sum_{i=0}^{N} a_i}{N} \tag{3.2}$$

where,
a_i = age of all products at time-step i in the inventory
N = number of time-steps
Hence, the goal of an agent is to maximize the s_{ratio} while minimizing the p_{ratio}.
As mentioned earlier in this section, all orders are generated by the consumer model and it is one of the more critical parts of the PreciBake environment. The next section provides more details about the consumer model.

4 Consumer Model

As stated in the previous chapter, a major part of the project pertains to successfully modeling consumer behavior. An ideal consumer model would have enough stochasticity to mirror demand of the real world, while also showcase enough predictability for our algorithms to detect certain patterns. Initially, public data was used to model consumer behavior. However, as the project progressed the limitations of such a model became apparent. We thus proceeded to develop a parameterized consumer model that would better reflect the demands of the project. A parametrized model would enable generation of whatever synthetic demand we would like to have, making testing much easier.

Mathematically, the goal is to generate a demand vector given a certain time step:

$$demand_t = f(t) \tag{4.1}$$

where $demand_t =< \tilde{d_t^a}, \tilde{d_t^b} ..., \tilde{d_t^n} >$ for products $P = \{a, b, ..., n\}$.

4.1 Public Data

A good place to start looking into consumer behavior is actual food order data. Fortunately, there are a number of online data sources aggregating consumer orders. One such data source is Kaggle – a popular data science competitive platform. It hosts a multitude of data sets as part of its numerous contests. *Takeaway Food Orders* [Kag] is a publicly available data set on the platform, with food order data for more than 30,000 orders. It contains information such as ordered items, time-stamp and quantity for each order. A number of models were created to fit the data set, given as follows.

Average Consumer The most basic of all consumer models. Here, we simply take an average of the quantities ordered at the current time step for each product.

$$\tilde{d_t^p} = mean(d_t^p) \quad \forall p \in P \tag{4.2}$$

The most apparent shortcoming of this model is its lack of randomness. The same demand would be generated at a given time step each time.

Nearest Neighbor Consumer A slightly more realistic model compared to the previous attempt, the nearest neighbor consumer looks at the closest time-stamp in the data set to the current time step:

$$t^*(t) = \begin{cases} t & \text{when } \exists\ demand_t \\ t' & \text{when } \nexists\ demand_t \wedge argmin_{t'}(t - t') \end{cases} \tag{4.3}$$

This gives us a list of orders that were made at this time on different days. It then selects an order from this list in a uniform random manner.

$$\tilde{d_t^p} = uniform_random(d_{t^*}^p) \quad \forall p \in P \tag{4.4}$$

While this model is an improvement over the average consumer as it is stochastic, it has the limitation that it can only generate orders that were already made in the data set.

Variational Auto Encoder (VAE) Consumer Another approach used was to train a generative model (in our case a VAE) that might have the capability of learning the dependencies between the orders in a given amount of time e.g. day, since it has proven its ability to learn the underlying structures in a given image [Doe16] with some mild assumptions in the latent space. After some experiments, we realized that our public data is highly sparse in terms of providing usable information for our problem. Therefore, we decided to use a different approach, which we explain in the next section.

4.2 Synthetic Data

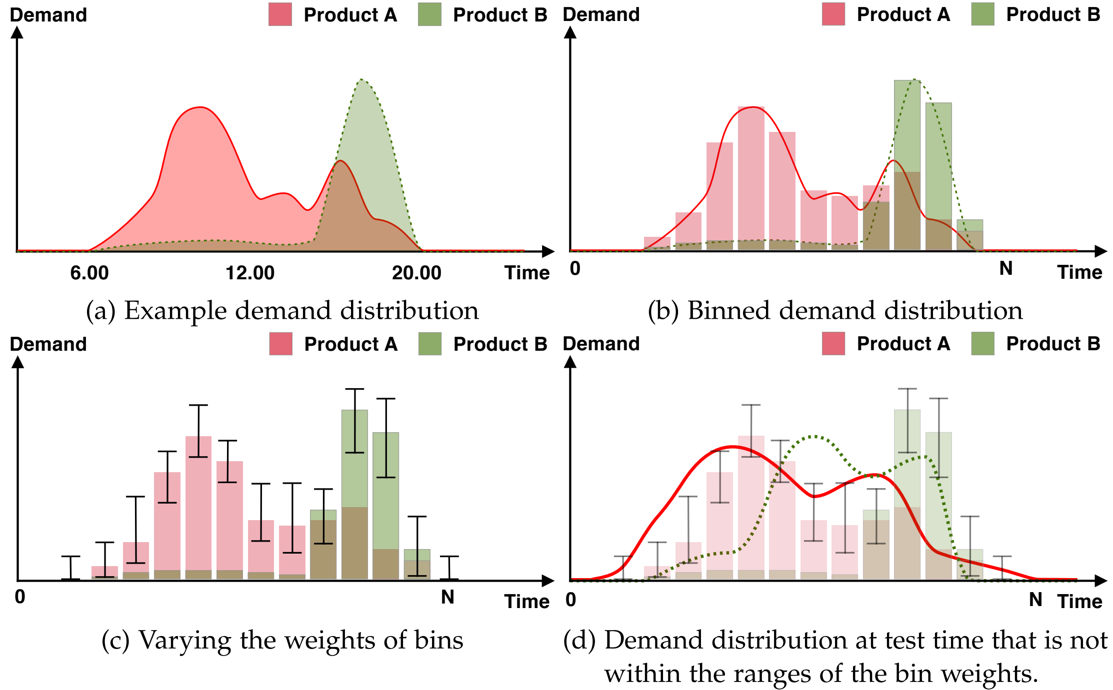
Poisson Consumer Considering the desired characteristics of a consumer model for this project, the Poisson distribution is a valuable tool in our arsenal. We developed a simple-yet-effective generative model, based on sampling from a Poisson distribution. It is also simple to interpret and adaptable to our learning algorithms. Furthermore, we consider it to be useful for domain randomization as it is easily parameterizable.

$$P(k \text{ orders in a bin }) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{4.5}$$

Our Poisson consumer model takes a list of means (λs in Equation 4.5) and generates a Poisson distribution with each of them. After assigning those Poisson distributions to the created bins of a certain time interval, it can be used to sample a demand given the current time-step.

The parameterization of the means in such a way makes the model applicable for domain randomization purposes. The Poisson distribution provides stochasticity and can be used for randomization, even when the weights are not changed.

By giving a range of mean demand for each bin, as in depicted in Figure 4.1c, we are able to generate a variety of customer demands trends for training. We want to use this to generalize our models and make them robust against unseen demand distributions; even extreme values outside the training ranges, as depicted in Figure 4.1d.



(a) Example demand distribution

(b) Binned demand distribution

(c) Varying the weights of bins

(d) Demand distribution at test time that is not within the ranges of the bin weights.

Figure 4.1: Bars indicate the means (λs in the Equation 4.5) for the Poisson distribution of the related bin. a) Shows an example demand distribution of 2 products throughout the day. b) An example of creating the bins out of the given demands to use as parameters for the Poisson model. c) A visualization of variating the weights of the Poisson distributions of each time interval to have randomized samples. d) Plots of the test demand data and its divergence from the Poisson models average values.

One other use-case for our binned Poisson distribution approach (one might consider it as a time-wise mixture of Poissons), is the ability to plug-in demand data at hand. This allows us to generate similar but random samples using the resulting bin weights, as shown Figure 4.1a and Figure 4.1b. This versatile approach can also be utilized for other demand problems where data is at hand. After providing an overview on our approaches for modeling consumer demand in this chapter, we focus on our implementations for the agent in the next chapter.

5 Agents

In our literature review, we saw how DP and RL can be used to engage SDM problems. In this chapter, we will describe adapted version of different agents for solving the real-world problem of inventory management for the food industry.

5.1 Baseline

To assess the performance of our proposed solutions, we must first determine a benchmark or baseline. As the project deals with the real-world problem of inventory management in a baking environment, we implemented a baseline that best reflects how a real bakery replenishes its inventory. The most common approach is to monitor the amount of each product in the inventory, and restock when it falls below a certain threshold. The choice of the threshold is arbitrary; we choose a threshold equaling to difference between the maximum amount the inventory can hold and maximum amount the oven can produce. For instance, if the inventory can only hold a maximum of 10 pieces and the oven can produce 5 pieces at most, the threshold (point at which the product must be restocked) is set to 5. This approach makes baseline agent a competitive benchmark in terms of s_{ratio} which will be introduced later. The algorithm for the baseline agent is given as follows:

for $p \in products$ **do**
  if $inv_p \leq inv_{max} - prod_{max}$ **then**
    $action \leftarrow prod_{max}$
  end if
end for

As can be observed, the chosen *action* also depends on the maximum capacity of the inventory. In case of multiple products falling below $inv_{max} - prod_{max}$ at the same time step, the agent simply picks the first product that satisfies the condition.

5.2 Deep Deterministic Policy Gradient (DDPG)

According to Equation 2.5, the policy state-value function $V_\pi(s)$ can be found given the state $s \in S$. However, the policy state-value function can also be re-written as a

function of a state $s \in S$ and an action $a \in A$. Equation 2.5 can be restated as, the policy action-value function Q:

$$Q_\pi(s,a) := \mathbb{E}_{\substack{p(w_k) \\ \pi_k(u_k|x_k)}} \left[g_N(x_N, u_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k) \middle| x_0 = s, u_0 = a \right] \tag{5.1}$$

where $g(x_k, u_k, x_{k+1})$ is the observed cost. Let us denote this as c. Hence, the optimal action-value function in conjunction to optimal state-value function is:

$$Q^*(s,a) = \min_\pi Q_\pi(s,a) \tag{5.2}$$

For a infinite horizon, the optimal value function $Q^*(s,a)$ satisfies the following equality for each pair of state $s \in S$ and an action $a \in A$:

$$Q^*(s,a) = \mathbb{E}_{x_k+1 \in S} \left[g(x_k, u_k, x_{k+1}) + min_{u_k+1 \in A} Q^*(x_{k+1}, u_{k+1}) \right] \tag{5.3}$$

In equation 5.2, if the action set is infinite e.g. a continuous action space, then it is not possible to find the value of $min_{u_k+1 \in A} Q^*(x_{k+1}, u_{k+1})$. We can approximate the value function by using an efficient, gradient based learning rule for policy $\pi(s)$.
Suppose that, the approximator is a neural network $Q_\phi(x,a)$ with parameters ϕ. We can state the mean-squared error (MSE) equation which tells us how close $Q_\phi(x_k, a_k)$ comes to satisfying the equation , as:

$$L(\phi) = \mathbb{E}_{(x_k, u_k, x_{k+1}, c)} \left[\left(Q_\phi(x_k, a_k) - (c + (1-d) \min_{u_{k+1}} Q^*_\phi(x_{k+1}, u_{k+1})) \right)^2 \right] \tag{5.4}$$

where

ϕ are the approximation parameters

d is a Boolean variable which specifies whether μ_{k+1} is the terminal state

Deep Deterministic Policy Gradient (DDPG) [Lil+15] is based on minimizing this MSE. In equation 5.4, it is known that the approximator is trying to make the Q-function closer to the target, i.e. $(1-d) \min_\phi Q^*_{k+1}(x_{k+1}, u_{k+1})$, but this is a problem because the target is dependent on the same parameters ϕ which we are trying to train. The solution to this problem is to calculate the target $(1-d) \min_\phi Q^*_{k+1}(x_{k+1}, u_{k+1})$, using a second network which is called the target network. This network is a time-delayed version of the main network which is being trained. The parameters of the target network will be referred to as ϕ_{targ}. After each epoch, the parameters of the main network are copied over to the target network.

While training the neural network, in order to make the training process more stable, an experience replay buffer is used, which is a set of (x_k, u_k, x_{k+1}, c) transitions. The size of this set is a parameter which has to be tuned because if the size of experience replay buffer is too small then the neural network will start to overfit itself on more recent experiences while if the size is too large then, that will impact the training speed of the neural network. The neural network is trained on batches of transitions which are sampled from the experience replay buffer.

To put it all together, DDPG uses the target network to tune the parameters of the approximator and minimize the mean squared error equation:

$$L(\phi) = \mathbb{E}_{(x_k,u_k,x_{k+1},c)}\left[(Q_\phi(x_k, a_k) - (c + (1-d)\min_{u_{k+1}} Q^*_{\phi_{targ}}(x_{k+1}, u_{k+1})))^2\right] \tag{5.5}$$

Our project specifications require both discrete actions (which product to bake) and continuous actions (quantity to bake). While DDPG is the state-of-the-art RL algorithm for continuous learning tasks, it is not designed to solve tasks for discrete actions spaces. The performance of the algorithm was thus not excellent. We subsequently decided to develop our own agent using the classic dynamic programming approach combined with a consumer demand prediction component.

5.3 Constraint Optimization with Demand Prediction

After the previous approach to solving the inventory management problem with model-free RL, we now describe an agent that combines a predictive model for the stochastic consumer demand with closed-loop optimization through dynamic programming. However, we replace the stochastic problem with a deterministic problem. Which is known as Certainty Equivalent Control (CEC) [Ber95]. At each stage k, the future uncertain demands are fixed to the predicted value. This allows for a on-line implementation, where the agent uses the first element in the optimal control sequence as the action for the current stage.

Demand prediction w/ Autoregressive Model Autoregressive models are widely used for time-series prediction, for instance in temperature and economic forecasting. The model assumes that the value of a variable X at time t is a linear combination of its previous p values and a stochastic term modeled as noise:

$$X_t = c + \sum_{i=1}^{p} \varphi_i X_{t-i} + \epsilon_t \tag{5.6}$$

where p is the length of the window looking into the past, φ is the set of parameters, ϵ_t is white noise and c is a constant. We use Python's `statsmodels` package implementation of the autoregressive model, which uses conditional maximum likelihood to learn the parameters. Our method is to train an AR model on a few episodes (days). To predict demand at test time, we first gather data for the next n steps, and calculate the average difference of this data with our prediction of the next n steps. This difference is then added to our prediction of whole trajectory. This process is repeated for each step, so that the model keeps improving its predictions given more data for each episode.

The advantage of an autoregressive model is its ability to adjust the demand prediction on-the-fly without requiring re-training. This is extremely important for robust demand prediction; we want to be able to adjust our predictions if a particular day is more or less busy than average.

Dynamic Programming and Certainty Equivalent Control After receiving a prediction of the demand for the remaining horizon, the production schedule needs to be optimized accordingly to minimize the cost. For this we extend the basic formulation from Bertsekas of a model of optimal control of a discrete-time dynamic system over a finite number of stages to account for our complex environment [Ber95]. Bertsekas describes the inventory control problem of ordering a quantity a single item (and getting it immediately delivered) at each stage to meet the stochastic demand (modeled as independent random variables), while minimizing the accompanying expected cost. For our real-world problem we need to account for multiple products with a time-shift in between production and delivery, constraining the production capacity. Therefore, we model the change in stock with an augmented state
$\tilde{x}_k^i = (x_k^i, d_{k-p}, ..., d_{k-1})$ according to the discrete-time equation

$$\tilde{x}_{k+1}^i = \tilde{x}_k^i + d_k^i - w_k^i, \tag{5.7}$$

where

k indexes discrete step

i denotes the product type

p is the maximal production time of all products.

$\tilde{x}_k^i$ is the stock per product type available at the beginning of the k th stage in combination with the information about previous deliveries, indicating the agent's ability to produce products.

d_k^i quantity delivered per product type at the beginning of the k th stage,

w_k^i demand per product type during the k th stage with given probability distribution, where $w_0^1, w_0^2, ..., w_{N-1}^i$ are independent random variables.

The cost incurred in stage k consists of the following components:

1. A cost $r(\tilde{x}_k^i)$ representing a penalty for positive stock $\tilde{x}_k^i$ (holding cost for excess inventory).
2. The cost $o(\tilde{x}_k^i)$ penalizing negative stock $\tilde{x}_k^i$ (shortage cost for unfilled demand).
3. The production cost $c^i d_k$, where c is product specific cost per delivered unit. The agent is limited to produce only one type of product per stage and can't produce anything else when the production is ongoing.
4. Terminal cost $R(x_N^i)$ for being left with inventory x_n^i at the end of N stages.

Thus, the cost for a initial stock x_0^i over N periods which we want to minimize with policy π, mapping stock x_k into delivery u_k is

$$V_\pi(x_0^i) = \mathbb{E}_{p(w_k^i)}\left[R(\tilde{x}_N^i) + \sum_{k=p}^{N-1}\sum_i (o(\tilde{x}_k^i) + r(\tilde{x}_k^i) + c^i u_k))\right] \tag{5.8}$$

In our algorithm we account for the total cost from holding and shortage costs through the sum over all product types $\sum_i \left[r(\tilde{x}_k^i) + o(\max(0, w_k^i - d_k^i - \tilde{x}_k^i))\right]$. Therefore, algorithm takes the form

$$\begin{aligned} V_k(\tilde{x}_k^i) = \min_{\substack{0<d_k<m-x_k \\ d_k=0,\dots,q}} \mathbb{E}_{p(w_k^i)}\Bigg[cd_k + \sum_i \left[r(\tilde{x}_k^i) + o(\max(0, w_k^i - d_k^i - \tilde{x}_k^i))\right] + \\ + V_{k+1}(\max(0, \tilde{x}_k^i + d_k^i - w_k^i))\Bigg]. \end{aligned} \tag{5.9}$$

For the on-line implementation with CEC we fix the w_i, $i \leq k$, at the predicted demand $\bar{w}_i$. Therefore, we solve the deterministic problem:

$$\begin{aligned} \min_{\substack{0<d_k<m-x_k \\ d_k=0,\dots,q}} \Bigg[cd_k + \sum_i \left[r(\tilde{x}_k^i) + o(\max(0, \bar{w}_k^i - d_k^i - \tilde{x}_k^i))\right] + \\ + V_{k+1}(\max(0, \tilde{x}_k^i + d_k^i - \bar{w}_k^i))\Bigg] \end{aligned} \tag{5.10}$$

where the stock $\tilde{x}_k^i$ is known, and $\tilde{x}_{k+1}^i = \tilde{x}_k^i + d_k^i - \bar{w}_k^i$. For each k our model predicts the consumer demand for the rest of the episode. The agents calculates the optimal

delivery sequence $\bar{\mu}_k^*(x_k)$ through backward recursion. It then chooses the optimal the first production decision for the current state by following $\bar{\mu}_k^*(x_k)$. This process is repeated for the entire horizon[1].

Example To visualize the combination of demand prediction with CEC we created a simple example in Figure 5.1. We consider two products over four stages. Further, the inventory capacity is limited to two items, the production is limited to a maximum of two items and has a production time of two stages. In order to minimize the inventory it begins in the last stage (depicted in blue). Here, the agent needs to meet the predicted demand for that stage. With the CEC the agent calculates that it would be optimal to deliver two *productsA* (still in blue). It is then extended to the whole example. For the given initial inventory of one item per product type, the agent identified an optimal policy by backwards recursion: deliver one *productA* in stage two and two *productsA* in the last stage.

After describing all relevant components for our project. We will continue with the evaluation and discussion of our results in the next chapter.

[1]Note, that we limited the maximal inventory capacity m=10 and the maximal production capacity q=5, to reduce computational complexity.

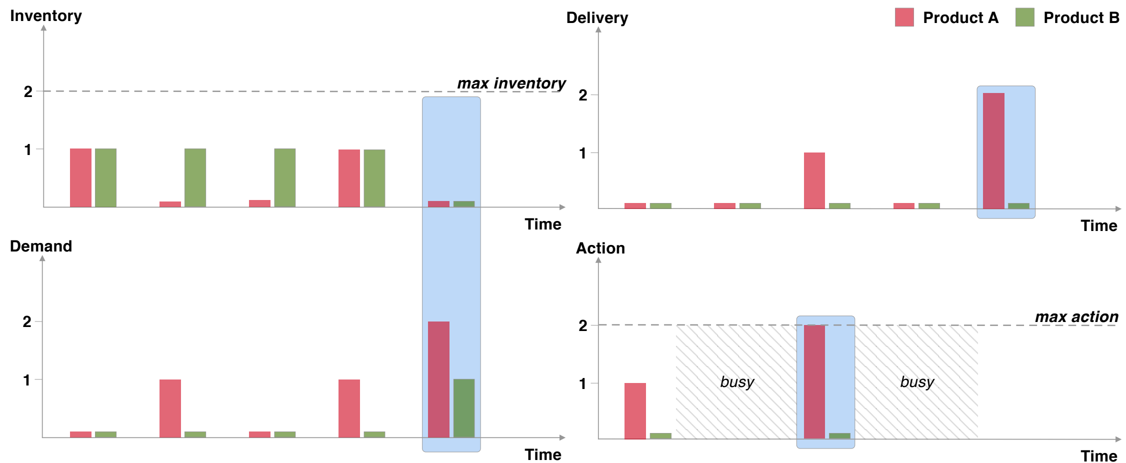


Figure 5.1: Example for serving the predicted demand with optimizing the production schedule that minimizes cost. We have extended the canonical inventory management example from Bertsekas [Ber95], by including more than 1 products and also introducing the producing time that can occupy the system for more than 1 intervals. In the top-right, we see the delivery that is optimal in case of the demand on the bottom-left and the underlying actions to take at the bottom right. Taking into account the agent's choice of starting inventory on top-left, it achieves a complete satisfaction of the demands. The optimality can be verified by checking if the delivery and the inventory together is able to satisfy the orders in each time-steps.

6 Results & Discussion

In this chapter we will evaluate the aforementioned agents. We want to identify how they perform on a set of unseen test cases, after having been trained in simulation on randomized demand data. However, we will first discuss the training procedure for the agents.

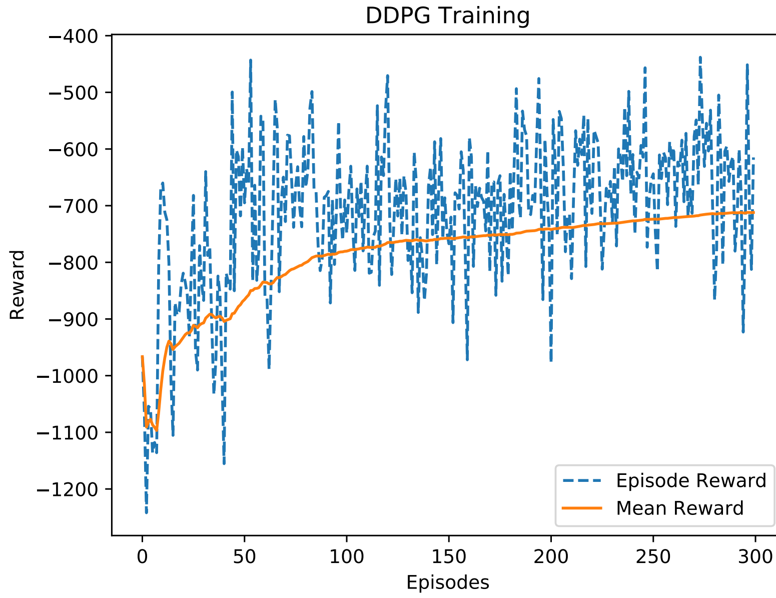
6.1 Training

DDPG The DDPG agent does not use the metrics described in Section 3.2 as reward signals, due to the metrics being available only at the end of an episode (rather than at each time step, making the signal sparse). Instead, we use:

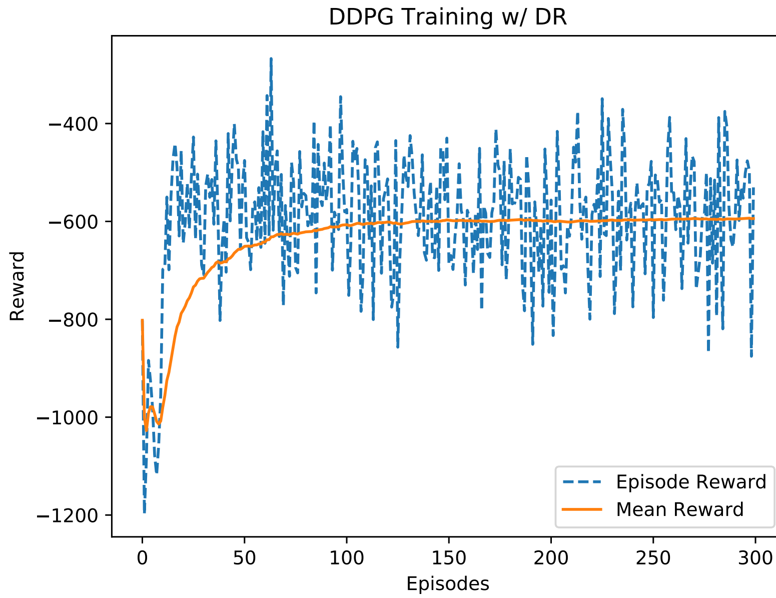
$$reward = -c_m, \tag{6.1}$$

where c_m is the number of missed customers (number of orders not fulfilled). The agent is trained with dynamics randomization, which means sampling from a different Poisson consumer demand model on each episode. The results for training convergence are shown in Figures 6.1a and 6.1b.

Autoregressive demand prediction For the agent described in Section 5.3, the training is required only on the demand prediction problem. To train the autoregressive model for demand prediction, data for 10 days was generated. The data was sampled from the same Poisson distribution; simulating a normal day of consumer demand. To validate that the autoregressive model adjusts its predictions to the demand on a day different from the training distribution, we sample data from a different Poisson distribution. Figure 6.2a shows the mean of demand of a product on a given time-step for both distributions. Note that the test data is a scaled-up version of the training data – this can be interpreted as the bakery having a busier day than usual. Figure 6.2b illustrates how the autoregressive model adjusts its predictions when the demand becomes greater than it expects. The dotted curve is the demand it predicts on a normal day, as it has been trained on only this distribution.

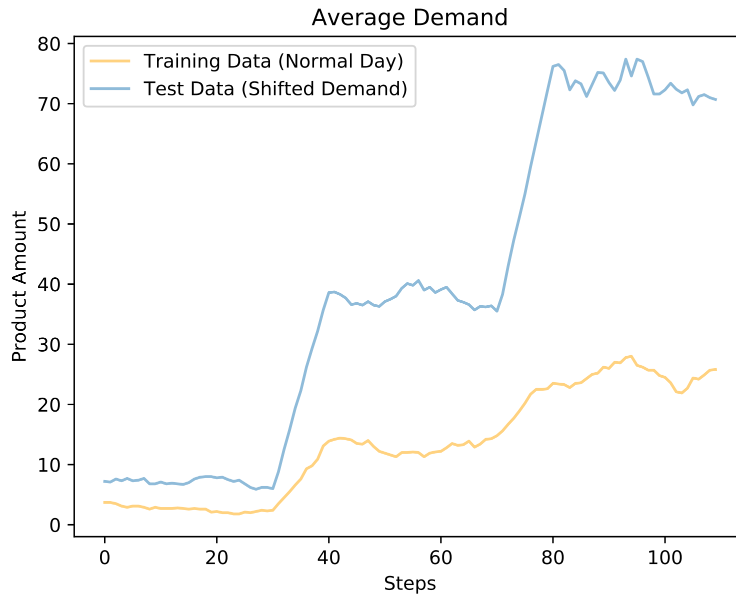


(a) Standard DDPG agent training

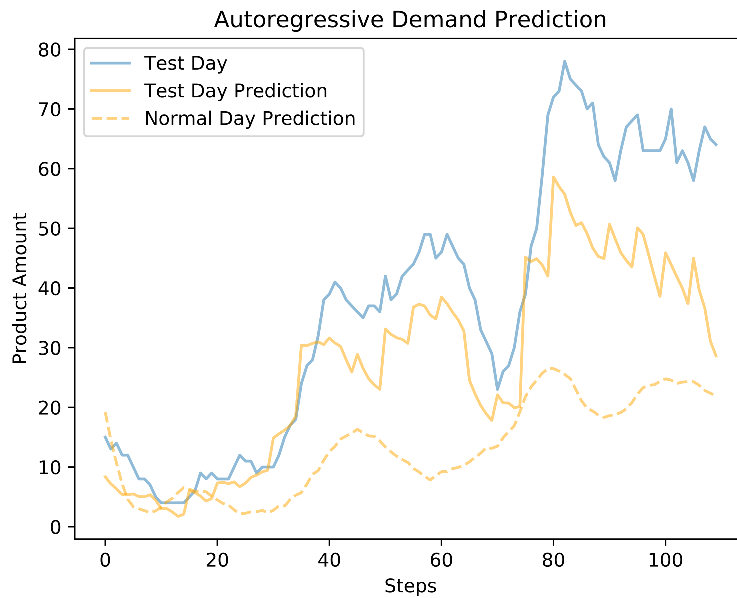


(b) DDPG with dynamics randomization

Figure 6.1: DDPG agent training with and without dynamics randomization (different consumer demand distribution per episode). Plots show per episode reward (dotted line) and rolling mean reward.



(a) Poisson distributions for training and testing. The test distribution is a scaled-up version of the training distribution.



(b) Trained AR model adjusts its normal prediction (dotted line) when the given (test) day shows higher demand.

Figure 6.2: Training the autoregressive model for demand prediction

6.2 Metric Benchmark

To test our agents against the baseline, four scenarios are considered, each corresponding to a different environment:

D_{in} – Consumer demand within trained distribution

D_{out^1} – 50% increase in consumer demand

D_{out^2} – 50% decrease in consumer demand

D_{out^3} – Consumer demand sampled from [-50%,+50%] of trained dist.

As the subscripts suggest, the first environment uses the same Poisson distribution for demand as the agents saw in training. The rest of the three environments are out-of-distribution: D_{out^1} can be interpreted as a busier day or location compared to a normal bakery, D_{out^2} as a less busier or remote location with less demand and D_{out^3} as a more chaotic bakery where the demand is uncertain and thus has a variance. The metrics used for testing correspond to the same as described in Section 3.2. A higher score for s_{ratio} means better performance, as it corresponds to more sales. Conversely, a higher score for p_{ratio} means a worse performance due to it is corresponding average waiting times(inverse of the freshness in the baking settings) of the products.

Agents	s_{ratio}				p_{ratio}			
	D_{in}	D_{out^1}	D_{out^2}	D_{out^3}	D_{in}	D_{out^1}	D_{out^2}	D_{out^3}
Baseline	0.5503	0.4121	0.8188	0.5867	5.3800	4.0300	10.169	6.4300
DDPG	0.4412	0.3316	0.7343	0.4222	2.5700	1.5399	6.5000	2.1500
DDPG w/ DR	0.2512	0.1902	0.4055	0.2597	1.6750	0.9650	1.9200	1.2600
DP w/ AR	0.5649	0.4013	0.8934	0.5736	3.2736	1.6421	13.000	3.6578
DP w/ Oracle	0.6716	0.4880	0.9521	0.6347	3.5250	2.7700	2.4500	3.6650

Table 6.1: Results for agents tested on various environment dynamics. Agents include DDPG with and without dynamics randomization (DR), DP agent with oracle (upper bound) and with autoregressive predictor.

The results are summarized in Table 6.1. Besides our agent combining DP with the the autoregressive model, we also evaluate the DP with an oracle demand model where we already know the exact consumer demand. Therefore, its prediction represents a upper-bound for our metric. It should be noted that each result is an average of 10 experiments.

It can be observed that the standard DDPG agent struggles with satisfying both metrics simultaneously. DDPG with dynamics randomization has lower scores on s_{ratio}

but better scores on p_{ratio}. DP with AR is the better than both DDPG variants when it comes to balancing both metrics in all but one out-of-distribution environments – it doesn't fall too much behind the best scores of others on s_{ratio}, even occasionally surpassing every other agent apart from the upper bound, all the while maintaining a lower p_{ratio}. DP with oracle is expectedly better than DP with AR – it shows us the best achievable performance and how much improvement we can expect with a better demand prediction model.

7 Conclusion

In this project, we have presented a successful approach for training RL agents that can perform in a new, realistic inventory control domain even if they were only trained on simulated demand data. We started by applying the theory of Dynamic Programming and Reinforcement Learning to our problem. We then got familiar with the PreciBake environment behaviour and tested dynamics randomization on other environments. Furthermore, we implemented different agents with respect to our inventory control environment. Parallel to that we focused on creating a realistic consumer model. For this we tried to find a suitable public data set that we could augment with new synthetic data. However, we did not further pursue this direction due to lack of a suitable data set. After having tried out multiple approaches, we finally achieved our goal by

- creating a parametric consumer demand model that generates intervals of synthetic data from Poission distributions with certain mean values and
- then training our adjusted agents, i.e. a DDPG agent and DP with demand prediction by an autoregressive model, on the synthetic demand data.

Our results show that the DP algorithm combined with an autoregressive (AR) demand prediction provides satisfactory solutions compared to the baseline and DDPG agents.

Furthermore, we observe that there is room for improvement on the demand prediction side; the DP agent with oracle demand offers a performance increase over its AR counterpart. Other methods exist that could perform competitively such as Long-Short-Term-Memory (LSTM) [HS97]. Such learning-based methods have been used more recently for time-series prediction [Kar+17] [Mal+15] to a successful degree. A potentially interesting future extension of the project would be to compare autoregressive prediction with an LSTM-based approach. Moreover, recent work has demonstrated that combining LSTM and autogressive models can yield state-of-the-art results in time-series prediction tasks [Lai+17], which could be another possible future direction.

Another interesting direction would be to extend the DP agent with function approximation to embrace the cases where computational cost might be too expensive for the exact DP approach presented in this work. Since we have simplified our problem

by constraining the inventory and the delivery sizes, we didn't need to delegate such approximation functions within this project. However, one might require to have an approximation for, e.g., the Value function, when the state space gets too large e.g. too many products and/or high inventory and delivery spaces.

Last but not least, it would be of interest to test the whole system with real data. On the one hand we could test how well parametric consumer model can be applied to existing data. On the other hand, we could retrain and test our agents to fulfill real demand by controlling an actual bakery.

Bibliography

[Bel66] R. Bellman. "Dynamic programming." In: *Science* 153.3731 (1966), pp. 34–37.

[Ber95] D. P. Bertsekas. *Dynamic programming and optimal control*. Vol. 1. 2. Athena scientific Belmont, MA, 1995.

[BSW89] A. G. Barto, R. S. Sutton, and C. Watkins. *Learning and sequential decision making*. University of Massachusetts Amherst, MA, 1989.

[Doe16] C. Doersch. "Tutorial on variational autoencoders." In: *arXiv preprint arXiv: 1606.05908* (2016).

[ER98] I. Erev and A. E. Roth. "Predicting how people play games: Reinforcement learning in experimental games with unique, mixed strategy equilibria." In: *American economic review* (1998), pp. 848–881.

[GG18] S. Gamrian and Y. Goldberg. "Transfer Learning for Related Reinforcement Learning Tasks via Image-to-Image Translation." In: *CoRR* abs/1806.07377 (2018). arXiv: `1806.07377`.

[HS97] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory." In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: `10.1162/neco.1997.9.8.1735`.

[Kag] Kaggle. *Takeaway Food Orders*. kaggle.com. Accessed: 2020-02-03.

[Kar+17] F. Karim, S. Majumdar, H. Darabi, and S. Chen. "LSTM fully convolutional networks for time series classification." In: *IEEE access* 6 (2017), pp. 1662–1669.

[KBP13] J. Kober, J. A. Bagnell, and J. Peters. "Reinforcement learning in robotics: A survey." In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1238–1274.

[Lai+17] G. Lai, W. Chang, Y. Yang, and H. Liu. "Modeling Long- and Short-Term Temporal Patterns with Deep Neural Networks." In: *CoRR* abs/1703.07015 (2017). arXiv: `1703.07015`.

[Lil+15] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. *Continuous control with deep reinforcement learning*. 2015. arXiv: `1509.02971 [cs.LG]`.

[LK01] S. M. LaValle and J. J. Kuffner Jr. “Randomized kinodynamic planning.” In: *The international journal of robotics research* 20.5 (2001), pp. 378–400.

[Mal+15] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal. “Long short term memory networks for anomaly detection in time series.” In: *Proceedings*. Vol. 89. Presses universitaires de Louvain. 2015.

[Mni+13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. “Playing atari with deep reinforcement learning.” In: *arXiv preprint arXiv:1312.5602* (2013).

[S+98] R. S. Sutton, A. G. Barto, et al. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge, 1998.

[SB87] R. S. Sutton and A. G. Barto. “A temporal-difference model of classical conditioning.” In: *Proceedings of the ninth annual conference of the cognitive science society*. Seattle, WA. 1987, pp. 355–378.

[SOL18] S. Sohn, J. Oh, and H. Lee. “Multitask Reinforcement Learning for Zero-shot Generalization with Subtask Dependencies.” In: *CoRR* abs/1807.07665 (2018). arXiv: 1807.07665.

[Tob+17] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. “Domain randomization for transferring deep neural networks from simulation to the real world.” In: *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2017, pp. 23–30.

[TS10] L. Torrey and J. Shavlik. “Transfer learning.” In: *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI Global, 2010, pp. 242–264.