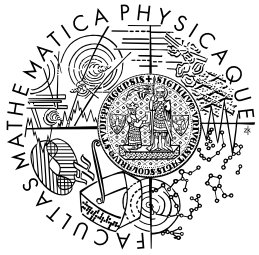


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Jakub Vondráček

Optimal Choice of Scenario Tree using Reinforcement Learning

Department of Probability and Mathematical Statistics

Supervisor of the master thesis: doc. RNDr. Ing. Miloš Kopa, Ph.D.

Study programme: Probability, mathematical statistics and econometrics

Study branch: MPSP

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I am infinitely grateful to my supervisor doc. RNDr. Ing. Miloš Kopa, Ph.D. for his expertise, patience and all the time and effort spent supervising my thesis and for providing the necessary GAMS license.

I am also deeply indebted to my consultant RNDr. Karel Kozmík for many great ideas, suggestions and always being ready to help whenever I needed guidance.

I dedicate this thesis to my grandmother and my mother, as without them, I would be nowhere near as far along in life as I am today. Thank you.

Title: Optimal Choice of Scenario Tree using Reinforcement Learning

Author: Bc. Jakub Vondráček

Department: Department of Probability and Mathematical Statistics

Supervisor: doc. RNDr. Ing. Miloš Kopa, Ph.D., Department of Probability and Mathematical Statistics

Abstract: This thesis deals with multistage stochastic programs and explores the dependence of the obtained objective value on the chosen structure of the scenario tree. In particular, the scenario trees are built using the moment matching method, a multistage mean-CVaR model is formulated and a reinforcement learning agent is trained on a set of historical financial data to choose the best scenario tree structure for the mean-CVaR model. For this purpose, we implemented a custom reinforcement learning environment. Further an inclusion of a penalty term in the reward obtained by the agent is proposed to avoid scenario trees that are too complex. The reinforcement learning agent is then evaluated against an agent that chooses the scenario tree structure at random and outperforms the random agent. Further the structure of scenario trees chosen by the reinforcement learning agent is analyzed.

Keywords: Stochastic optimization, Multistage problem, Reinforcement learning

Contents

Introduction

Stochastic programming is a branch of mathematical optimization that allows to account for uncertain parameters when solving mathematical programs, which led to the widespread adoption of stochastic programming in fields such as finance, transportation, scheduling and telecommunications (Shapiro et al. [35], Ruszczynski and Shapiro [31]).

This makes it a very powerful tool, which however comes at a significant computational cost. Due to the fact that the random parameters may follow a continuous distribution, approximating such distributions by a discrete set of scenarios is necessary to even be able to formulate the model and also to be able to solve it in finite time. Even more demanding are so called multistage programs, which allow multiple decision periods. To be able to solve multistage programs, the scenarios approximating the continuous distributions in every stage are arranged in a scenario tree. The structure of this tree is very important for the obtained solution, as a tree that is very simple may not approximate the underlying distribution correctly, while a tree that is too complex suffers from extensive computational costs.

This is the main idea of this thesis – to discover whether it is possible to predict a scenario tree structure that is optimal with regard to the objective function and also potentially with regard to the complexity of the scenario tree. To solve this problem, we propose an experiment to train a reinforcement learning agent (Sutton and Barto [38]) using the solutions of a mean-CVaR model (Salahi et al. [32], Rockafellar and Uryasev [29], Rockafellar and Uryasev [30]) calculated using scenario trees generated from historical financial data.

Chapters 1, 2 and 3 provide the necessary theory for Multistage stochastic programming, the mean-CVaR model and Reinforcement learning respectively. The mean CVaR model formulated in Section 2.2.1, while certainly not novel, is of our own design. The main contribution of this thesis is Chapter 4, the pinnacle of this thesis, where we implement the experiment described above and analyse the results. To the best of our knowledge, such an experiment has not been proposed in the literature. Also a notable contribution is the compilation and standardization of notation for several machine learning algorithms in Chapter 3 from multiple different sources.

Chapter 1

Stochastic programming

In this chapter, we give an introduction to the theory of stochastic programming with particular focus on multistage linear programs. Most of the content in this chapter is based on the material covered in Ruszczynski and Shapiro [31, Chapter 1], Shapiro et al. [35, Chapters 1-3] and Dupačová et al. [9, Part 2] if not specified otherwise.

1.1 Basic definitions

Definition 1.1.1. Mathematical program in $\mathbb{R}^n$ (Dupačová et al. [9, p. 107])
Let $p, m, n \in \mathbb{N}$. A mathematical program in $\mathbb{R}^n$ is defined as

$$\min\{f(\mathbf{x}), \mathbf{x} \in \mathbf{M}\},$$

where $\mathbf{M} \subset \mathbb{R}^n$ and $f : \mathbf{M} \to \mathbb{R}$. The function f is called the objective function and the set $\mathbf{M}$ is called the set of feasible solutions. This set is usually defined by constraints as follows:

$$\mathbf{M} = \{\mathbf{x} \in \mathbb{R}^n : h_j(\mathbf{x}) = 0,\ j = 1, \dots, p,\ g_k(\mathbf{x}) \leq 0, k = 1, \dots, m\},$$

where h_j and g_k are real functions.

If all functions in Definition 1.1.1 are linear, we call the problem a *Linear program*. Furthermore, if any of the functions mentioned in Definition 1.1.1 depend on parameters, we call the problem a *Parametric program*. If any of the parameters are random variables, we call the problem a *Stochastic program*.

However, this definition of a Stochastic program is not well formulated. Consider the Definition 1.1.1 and let Ω be a non-empty set, $\mathcal{F}$ be a σ-algebra on Ω, $\omega \in \Omega$ and $\mathcal{P}$ be a probability measure on $(\Omega, \mathcal{F})$, leading to a probability space $(\Omega, \mathcal{F}, \mathcal{P})$. In the context of a Stochastic program, the function f does not depend on $\mathbf{x}$ only, but also on the realisation of ω. This would lead to a nonsensical definition, as for different realisations of ω, the optimal value may be different. The standard way to handle this problem is to consider minimisation of expected value of the function f:

$$\min\{\mathbb{E}\left[f(\mathbf{x}, \omega)\right], \mathbf{x} \in \mathbf{M}\},$$

where $\mathbb{E}$ is the expected value operator defined with respect to the probability measure $\mathcal{P}$.

1.2 Multistage stochastic programming

In the most basic form, the stochastic programming paradigm allows to make an optimal decision with regard to the expectation only for one decision period. This is a considerable limitation, which can be overcome by extending the notion of a *Stochastic program* to a *Multistage stochastic program*.

1.2.1 Notation and general idea

This section is heavily inspired by Ruszczynski and Shapiro [31, Section 3.3.]. Following the notation established there, consider the following sequence of events

$$\begin{gathered}\text{Observation } \xi_1 \\ \downarrow \\ \text{Decision } x_1 \\ \downarrow \\ \text{Observation } \xi_2 \\ \downarrow \\ \text{Decision } x_2 \\ \downarrow \\ \text{Observation } \xi_3 \\ \downarrow \\ \vdots \\ \downarrow \\ \text{Observation } \xi_T \\ \downarrow \\ \text{Decision } x_T,\end{gathered}$$

where T is the number of decision stages, $x = (x_1, \ldots, x_T)$ is called the decision process (x_1 is a non-random vector of variables), $\xi = (\xi_1, \xi_2, \ldots, \xi_T)$ is a stochastic data process (ξ_1 is assumed to be known before making the decision x_1 and can thus be considered deterministic). The decision process x represents the decisions made at each stage (i.e. for a portfolio optimization problem, x_t may be a random vector of proportions of some assets in a portfolio in stage t) and ξ_t is a random vector representing the data process in stage t (i.e. it may be a vector of yearly asset returns). Furthermore, the probability distribution of ξ is assumed to be known. Usually, one more observation from the stochastic data process, denoted as ξ_{T+1}, is considered to happen after the last decision x_T.

1.2.2 Nonanticipativity

Both processes x and ξ are random and thus depend on the realised $\omega \in \Omega$. In order for the program to be well defined, the decision process x must not take into account future observations of either ξ or decisions x, but only the past and present. This is formalised by the so called nonanticipativity constraints, which assure that the x_t at time t may depend only on $(x_1, \ldots, x_{t-1})$ and $(\xi_1, ..., \xi_t)$.

Definition 1.2.1. Nonanticipativity constraints (Ruszczynski and Shapiro [31, Ch. 1, Section 3.3., P. 35])
The decision process x is termed nonanticipative if

$$x_t = \mathbb{E}\left[x_t | \xi_1, \ldots, \xi_t\right], t = 1, \ldots, T,$$

or equivalently, if $\mathcal{F}_t$ is the σ-algebra generated by $(\xi_1, \ldots, \xi_t)$, then x_t must be measurable with respect to $\mathcal{F}_t$, where $\mathcal{F}_1 \subset \mathcal{F}_2 \subset \cdots \subset \mathcal{F}_T \subset \mathcal{F}$.

This is in contrast to general deterministic (multiperiod) programs, where all data are taken into account. Of course, this definition is not very practical from an implementation standpoint. We provide explicit formulation of nonanticipavity constraints in Section 2.2.1.

1.2.3 General multistage optimization problem formulation

Considering the single period optimization in each stage recursively (in the reverse order, i.e. from the last stage to the first stage) leads to the following nested problem formulation, see Definition 1.2.2. This formulation is very similar to problems found in dynamic programming.

Definition 1.2.2. Multistage nested problem formulation (Shapiro et al. [35, Section 3.1.1.])
Let $\xi = (\xi_1, \ldots, \xi_T)$ be the data process as defined in the previous sections, $x = (x_1, \ldots, x_t)$ the decision vector, n_t, $d_t \in \mathbb{N}$ be the dimensions of x_t and ξ_t respectively, $f_t : \mathbb{R}^{n_t} \times \mathbb{R}^{d_t} \to \mathbb{R}$ be continuous functions and $\mathcal{X}_t$ be the sets of constraints at each stage $t = 1, \ldots, T$. The general multistage optimization problem is then formulated as follows:

$$\min_{x_1 \in \mathcal{X}_1} f_1(x_1) + \mathbb{E}\left[\inf_{x_2 \in \mathcal{X}_2(x_1, \xi_2)} f_2(x_2, \xi_2) + \mathbb{E}\left[\cdots + \mathbb{E}\left[\inf_{x_T \in \mathcal{X}_T(x_{T-1}, \xi_T)} f_T(x_T, \xi_T)\right]\right]\right], \quad (1.1)$$

where the multifunctions $\mathcal{X}_t(x_{t-1}, \xi_t)$ are random, since they dependend on ξ_t and x_{t-1}, $t = 2, \ldots, T$.

1.2.4 Linear programming formulation

If the functions f_t and constraints defining the sets $\mathcal{X}_t$, $t = 1, \dots, T$ are linear in Equation 1.1, then we say that the problem is a *Multistage linear programming problem*, see Definition 1.2.3.

Definition 1.2.3. Multistage nested linear problem formulation (Ruszczynski and Shapiro [31, Ch. 1, p. 25])
Considering the notation in Definition 1.2.2, let

$$\begin{aligned} f_1 &= c_1^T x_1, \\ \mathcal{X}_1 &= \{x : A_{1,1}x_1 = b_1, x \geq 0\}, \\ f_t &= {c_t}^T x_t, \\ \mathcal{X}_t &= \{x : A_{t,t-1}x_{t-1} + A_{t,t}x_t = b_t\,, x \geq 0\}, \end{aligned}$$

$t = 2, \dots, T$, *where* $A_{t,t}, A_{t,t-1}$ *are matrices and* b_t, c_t *are vectors of consistent dimensions with* x_t *and* x_{t-1}. *Assuming that all infima are attained and all expectations exist, rewriting Equation 1.1, we obtain:*

$$\min_{\substack{A_{1,1}x_1 = b_1 \\ x_1 \geq 0}} c_1^T x_1 + \mathbb{E}\left[\min_{\substack{A_{2,1}x_1 + A_{2,2}x_2 = b_2 \\ x_2 \geq 0}} {c_2}^T x_2 + \mathbb{E}\left[\cdots + \mathbb{E}\left[\min_{\substack{A_{T,T-1}x_{T-1} + A_{T,T}x_T = b_T \\ x_T \geq 0}} {c_T}^T x_T\right]\right]\right], \tag{1.2}$$

where the quantities $c_t, A_{t,t-1}, A_{t,t}, b_t$ *are random, or according to notation established earlier,* $\xi_t = (c_t, A_{t,t-1}, A_{t,t}, b_t)$, $t = 2, \dots, T$, *while* $c_1, A_{1,1}, b_1$ *are known and deterministic.*

If we view the decision process x_t (at time t) as a function of the part of the data process ξ known up to time t, i.e. $x_t = x_t(\xi_1, \dots, \xi_t)$, we can properly rewrite the multistage linear programming problem (Equation 1.2) in the same general form as other linear programming problems, see Definition 1.2.4.

Definition 1.2.4. Multistage linear problem formulation
(Ruszczynski and Shapiro [31, Ch. 1, p. 22])

$$\begin{aligned} &\min_{x_1 \geq 0, x_2, \dots, x_T \geq 0} \quad \mathbb{E}\left[c_1^T x_1 + \cdots + c_T^T x_T\right] && && (1.3) \\ s.t.\ & A_{1,1}x_1 && = b_1 \\ & A_{2,1}x_1 + A_{2,2}x_2 && = b_2 \\ & A_{3,2}x_2 + A_{3,3}x_3 && = b_3 \\ & && \vdots \\ & && A_{T,T-1}x_{T-1} + A_{T,T}x_T = b_T, \\ & && x_t \geq 0, t = 1, \dots, T, \end{aligned}$$

where the nonanticipativity constraints are included implicitly.

Remark. The decision process x is also called *policy* in the literature. Particularly Ruszczynski and Shapiro [31, p. 95, Definition 29] define that x is called an *implementable policy* if x_t is a function of the part of the data process known up to time t, i.e. $(\xi_1, \dots, \xi_t)$. Furthermore, if the implementable policy satisfies all constraints, it is called a *feasible policy*.

All aforementioned stochastic programming problems contain random variables that may follow general distributions, which, in particular, may be continuous. The literature suggests that while theoretically formulating such multistage stochastic programs is possible, solving such programs is not feasible. Particularly Pflug [27] mentions that this is "*due to the fact that the decisions are functions, making the problem a functional optimization problem, which cannot be numerically solved as it is*". To get around this problem, the continuous distributions can be approximated using discrete distributions. We need to keep in mind that the resulting formulation is only approximate and solving it thus provides an estimate of the true objective value (and true decision variables).

Formulation using scenario trees

To obtain a solvable program, we must obtain a suitable approximation of the underlying continuous distribution. This approximation may be a discrete distribution with a finite number of atoms (scenarios) $\xi^i, i = 1, \ldots, S$ where S is the number of scenarios. The scenarios are usually organised in tree form as shown in Figures 1.1 and 1.2. The following description of scenario tree formulation is heavily inspired by Dupačová et al. [8, Section 2].

Consider a multistage program with T stages. In each of these stages, we need to find a discretization of the underlying distribution, particularly the marginal distribution in the first stage and the conditional distributions in the following stages. The number of atoms in each of these stages need not be the same. If the number of children is the same for each node in a given stage and this property holds for all stages, the scenario tree is termed *balanced*, see Figure 1.1.

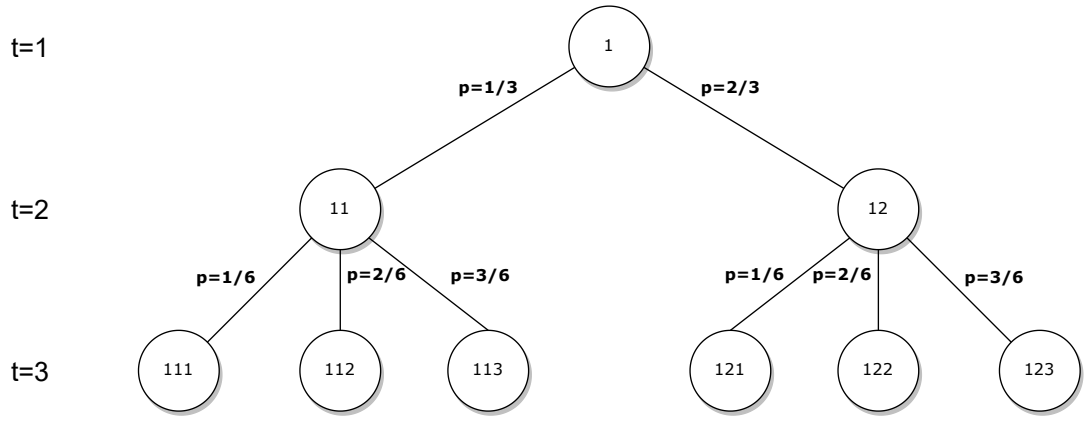


Figure 1.1: Balanced scenario tree.

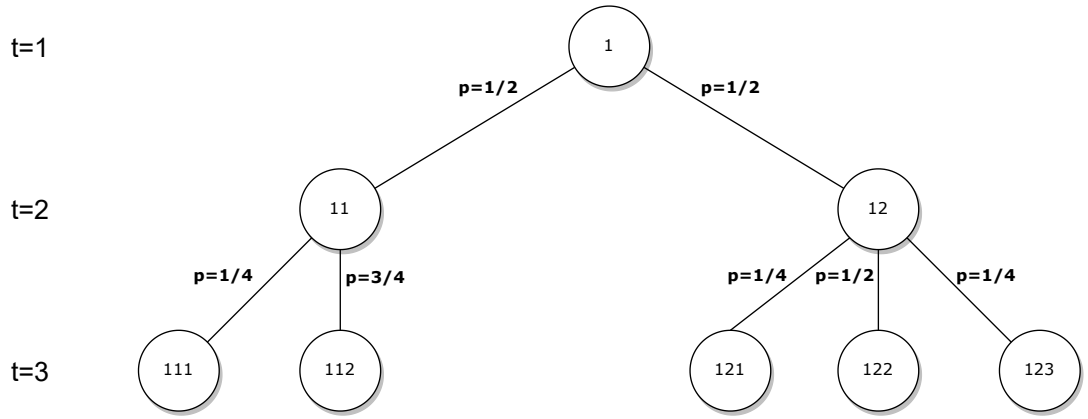


Figure 1.2: Unbalanced scenario tree.

Define *scenario at stage* t as $\xi_{[:t]} = (\xi_1, \ldots, \xi_t)$. Let $S_t(\xi_{[:t-1]})$ be the finite support of the conditional probability distribution of ξ_t, $t = 3, \ldots, T$ and denote $S_2(\xi_1)$ the finite support of the marginal probability distribution of ξ_2. Naturally, we only want to consider scenarios in the set

$$\mathcal{S} = \{\xi | \xi_t \in S_t(\xi_{[:t-1]}), t = 2, \ldots, T\}.$$

Let us now define the probabilities associated with each scenario. Let $P(\xi^i_{[:T]})$ be the probability of scenario i at the very last stage, i.e.

$$P(\xi^i_{[:T]}) = p_i = P(\xi^i_2) \prod_{t=3}^{T} P(\xi^i_t | \xi_{[:t-1]}), i = 1, \ldots, S, \tag{1.4}$$

where S (which earlier represented the number of scenarios of a given distribution) now represents the number of scenarios in the very last stage, i.e. the number of leaves of the scenario tree), $P(\xi^i_2)$ is the unconditional marginal probability on S_2 in the first stage and $P(\xi^i_t | \xi_{[:t-1]})$ are the conditional probabilities on $S_t(\xi_{[:t-1]})$ in the following stages of scenario i. The probabilities $P(\xi^i_2)$ and $P(\xi^i_t | \xi_{[:t-1]})$ are termed *arc probabilities* and their products

$$P(\xi^i_{[:t]}) = P(\xi^i_2) \prod_{\tau=3}^{t} P(\xi^i_\tau | \xi_{[:\tau-1]}), i = 1, \ldots, S, t = 3, \ldots, T, \tag{1.5}$$

are called *path probabilities*. Obviously, the path probability of scenario i at time T coincides with p_i.

We define two special cases of a scenario tree:

1. Interstage independent tree

   - If $P(\xi^i_t | \xi_{[:t-1]}) = P(\xi^i_t)$ (the conditional probabilities are equal to the marginal probabilities), then the tree is termed *interstage independent*.

2. Fan of scenarios

   - If $S_2(\xi_1)$ and $S_t(\xi_{[:t-1]}), t = 3, \ldots, T$ are singletons, the tree collapses into a *fan of scenarios*, see Figure 1.4.

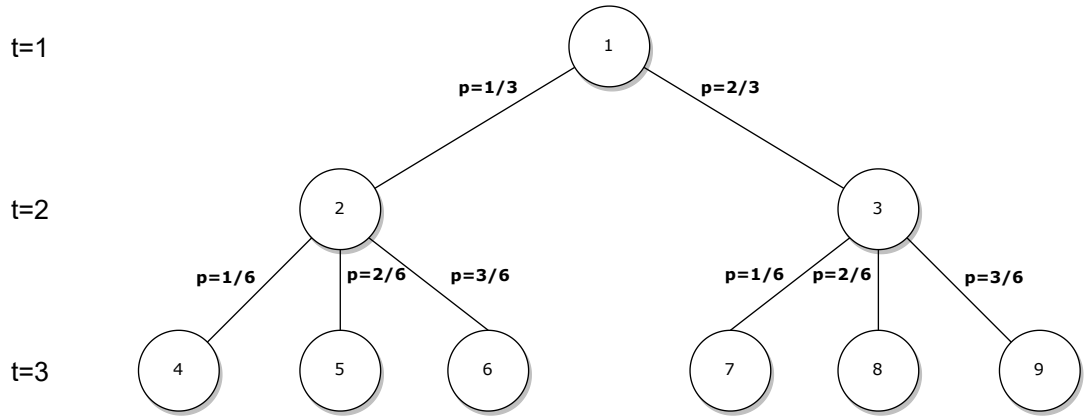


Figure 1.3: Balanced scenario tree illustration for how sets $\mathcal{K}_t$ are constructed.

Having defined the necessary notation for the scenario tree structure, we can now formulate a multistage linear program using a scenario tree. Let $\mathcal{K}_t, t = 1, \ldots, T$ be disjoint sets of indices enumerating nodes in each stage of the scenario tree. This is best illustrated using Figure 1.3, where $\mathcal{K}_1 = \{1\}$, $\mathcal{K}_2 = \{2, 3\}$ and $\mathcal{K}_3 = \{4, 5, 6, 7, 8, 9\}$. This leads to Definition 1.2.5.

Definition 1.2.5. Arborescent form of a scenario based linear program (Dupačová et al. [8, p. 3])

$$\min c_1^T x_1 + \sum_{k_2=2}^{K_2} p_{k_2} c_{k_2}^T x_{k_2} + \sum_{k_3=k_2+1}^{K_3} p_{k_3} c_{k_3}^T x_{k_3} + \cdots + \sum_{k_T=k_{T-1}+1}^{K_T} p_{k_T} c_{k_T}^T x_{k_T} \tag{1.6}$$

$$\begin{aligned} s.t.\ A_{1,1}x_1 &= b_1 \\ A_{k_2,1}x_1 + A_{k_2,k_2}x_{k_2} &= b_{k_2}, k_2 \in \mathcal{K}_2 \\ A_{k_3,a(k_3)}x_{a(k_3)} + A_{k_3,k_3}x_{k_3} &= b_{k_3}, k_3 \in \mathcal{K}_3 \\ &\vdots \\ A_{k_T,a(k_T)}x_{a(k_T)} + A_{k_T,k_T}x_{k_T} &= b_{k_T}, k_T \in \mathcal{K}_T, \end{aligned}$$

$$x_{k_t} \geq 0, k_t \in \mathcal{K}_t,\ t = 1, \ldots, T,$$

where $K_1 = 1$, $\mathcal{K}_t = \{K_{t-1} + 1, \ldots, K_t\}$, $t = 2, \ldots, T$, $a(k_t)$ denotes the predecessor of k_t and $\mathcal{K}_t$, $t = 1, \ldots, T$ are disjoint sets of indices enumerating nodes in stage t of the scenario tree.

In definition 1.2.5, the nonanticipativity constraints are again included implicitly. With the explicit inclusion of nonanticipativity constraints and by taking the expectation over all scenarios, we arrive at a formulation that is more suitable for practical use, see Definition 1.2.6. This is the formulation that we will use in the rest of this text as basis for formulating and solving scenario based multistage linear programs.

Definition 1.2.6. Scenario splitted form of a scenario based linear program (Defourny et al. [7])

$$
\begin{aligned}
&\min \sum_{i=1}^{S} p_i \left[(c_1^T)x_1^i + (c_2^i)^T x_2^i + \cdots + (c_T^i)^T x_T^i\right] \\
&s.t.\ A_{1,1}x_1^i = b_1 \\
&\quad A_{2,1}^i x_1^i + A_{2,2}^i x_2^i = b_2^i \\
&\quad\quad A_{3,2}^i x_2^i + A_{3,3}^i x_3^i = b_3^i \\
&\quad\quad\quad \vdots \\
&\quad\quad\quad A_{T,T-1}^i x_{T-1}^i + A_{T,T}^i x_T^i = b_T^i, \\
&i = 1, \ldots, S \\
&x_t^i \geq 0, i = 1, \ldots, S,\ t = 1, \ldots, T, \\
&x_t^j = x_t^k, \text{ if } \xi_{[:t]}^j = \xi_{[:t]}^k\ j, k \in \{1, \ldots, S\}, t = 1, \ldots, T,
\end{aligned} \tag{1.7}
$$

where p_i is the probability of scenario i as defined in Equation 1.4, $x_t^i, A_{t,t-1}^i, A_{t,t}^i, b_t^i$ are respectively the decision variables, matrices and vector of constraints in scenario i, $t = 1, \ldots, T$. Equation 1.7 represents the nonanticipativity constraints and obviously, the nonanticipativity constraints hold for $t = 1$, since ξ_1 is deterministic and common to all scenarios.

1.2.5 Methods for generation of scenario trees

There is a plethora of literature on the methods for scenario tree generation. In this section, we aim to present a summary of the basic methods, of which only the moment matching method is relevant for the purposes of this text in later chapters (due to reasons explained in Section 4.7) and is therefore treated in much more detail.

Prior to generating a scenario tree, we need to decide the structure of the tree, particularly the number of stages and the branching structure of the tree (i.e. how many descendants should each node have). This is in general not an easy task, which we will come back to in Section 1.2.6.

Moment matching

Høyland and Wallace [15] presented a novel method of generating a scenario tree from historical data that matches some statistical properties of the data, particularly by minimising the distance between moments and correlations calculated from historical data and the moments and correlations calculated from the scenario tree. They formulated a nonlinear optimization program to solve this task. In practical use, this method can be very time demanding and this led to development of a heuristic method for generating scenario trees using moment matching in Høyland et al. [16]. Calfa et al. [5] further improved on top of the non-heuristic method and proposed a way to also include empirical cumulative distribution function matching in the program. We will follow the notation established in Calfa et al. [5] and describe the optimization program used for generating a simple one stage scenario tree that consists only of a root and N child nodes using the moment matching method (without the cumulative distribution function matching).

Remark. For the purposes of this thesis, we can restrict the theory only to this simple case, as we will be working with stagewise independent balanced scenario trees and thus use the moment matching method sequentially in each stage. Further details on how the moment matching method is used are described in Section 4.5.1. For the more complex case, where the whole tree is generated at once (which is in principle the same as presented here, only with more complex notation), see Høyland and Wallace [15].

Definition 1.2.7. Moment matching (Calfa et al. [5, p. 9])
Let the scenario tree consist of only one stage, i.e. a root and N children. Let I be the set of assets, K be the number of moments to be matched,
$\mathcal{M} = \{1, \ldots, K\}$, *$M_{i,k}$ the k-th sample moment calculated from historical data of i-th asset, $C_{i,i'}$ be the sample correlation calculated from historical data between i-th and i'-th assets. Then the moment matching problem can be formulated as follows:*

$$\min_{\substack{p_j, x_{i,j}, \\ j \in \{1,\ldots,N\}, i \in I}} \sum_{i \in I} \sum_{k \in \mathcal{M}} (m_{i,k} - M_{i,k})^2 + \sum_{(i,i') \in I, i < i'} (c_{i,i'} - C_{i,i'})^2$$

$$s.t. \sum_{j=1}^{N} p_j = 1$$

$$m_{i,1} = \sum_{j=1}^{N} p_j x_{i,j}, i \in I$$

$$m_{i,k} = \sum_{j=1}^{N} p_j (x_{i,j} - m_{i,1})^k, i \in I, k > 1$$

$$c_{i,i'} = \sum_{j=1}^{N} (x_{i,j} - m_{i,1})(x_{i',j} - m_{i',1}) p_j, i, i' \in I, i < i'$$

$$x_{i,j}^L \leq x_{i,j} \leq x_{i,j}^U, i \in I, j = 1, \ldots, N,$$
$$0 \leq p_j \leq 1, j = 1, \ldots, N,$$

where $m_{i,k}$ are the moments and $c_{i,i'}$ are the correlations between asset i and i' calculated from the tree, p_j are the probabilities of each child and $x_{i,j}^L$ and $x_{i,j}^U$ are the lower and upper bounds on the decision variable $x_{i,j}$.

Simulation and reduction

For the sake of completeness of this text, another common approach to generate a scenario tree is to use a parametric model for the underlying process, simulate multiple trajectories of such process and then use a reduction procedure to obtain a valid tree.

Many reduction methods can be considered, examples include clustering of similar trajectories together, nodal and scenario extraction methods and distance based methods, which find a smaller tree that is in some sense close to the original (based on a given metric). For an exhaustive summary, we recommend Vitali et al. [41].

To demonstrate the procedure, we give an overview of the clustering method.

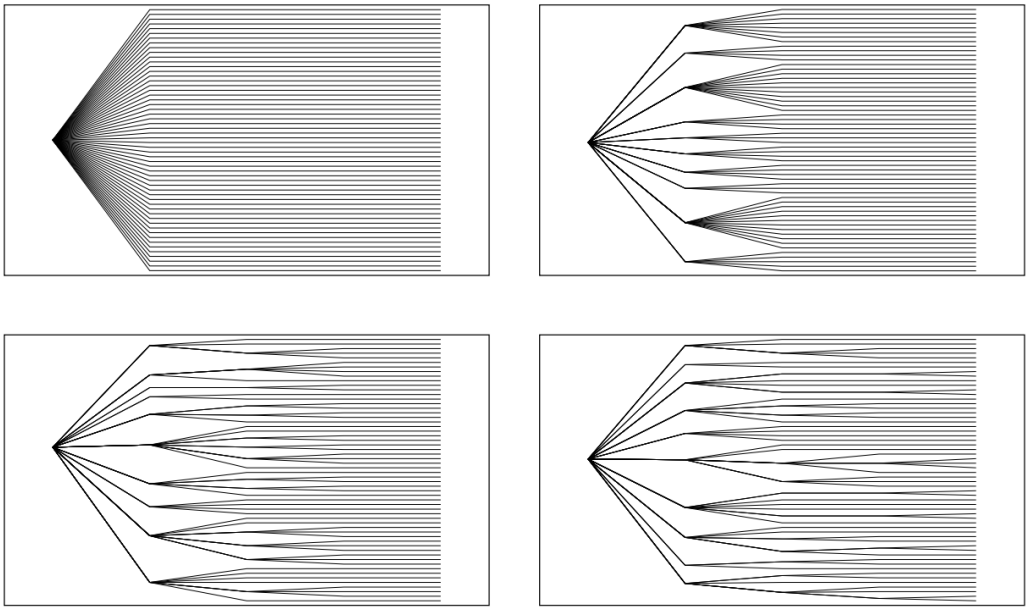


Figure 1.4: Illustration of a clustering procedure applied to a fan of scenarios which generates an unbalanced scenario tree. Image sourced from Heitsch and Roemisch [13, Figure 5].

Clustering

After a sufficient number of paths generated from a parametric model are obtained, we are faced with a fan of scenarios (see the top left illustration in Figure 1.4.). To obtain a valid scenario tree, we need to cluster similar paths together in each stage while preserving the tree structure. This is done by applying a clustering algorithm that partitions the given set of observations in the first stage into several groups (and then applying the algorithm again on each group recursively). An algorithm that may be useful for this task is the k-means algorithm, originally developed in MacQueen [20]. A precise description of the process can be found in Šutiene et al. [42, Section 3].

1.2.6 Curse of dimensionality

The curse of dimensionality is a phenomenon where increasing the complexity (dimension) of a problem leads to much higher (often exponentially higher) computational and memory demands, many times so large that the problem becomes unsolvable. A particularly relevant example is dynamic programming, where all states of a system are (usually recursively) explored to obtain the optimal solution. This means that increasing the dimensionality of the problem exponentially increases the number of states, leading to an unsolvable problem.

The scenario tree approach to stochastic optimization suffers from the curse of dimensionality as well, since the number of scenarios grows with the number of decision stages and the number of descendants in each stage. Many methods have been developed to reduce the number of scenarios (as explained in the previous section). However, many of these methods take the already generated fan of scenarios as input and return a tree with predefined branching structure. The

branching structure is usually chosen ad hoc or by an expert opinion. This may not be the optimal approach, as a different tree structure (different discretization of the randomness) may yield better results in terms of the objective function. Particularly choosing a tree with too many stages and many children in each stage leads to a computationally intractable problem.

This is the main question we explore in this thesis – what is the dependence of the objective function on the tree structure and can we find a scenario tree structure that is optimal with regard to the objective function and potentially also with regard to the size of the scenario tree?

Chapter 2

Risk measures

When investing, the investor has to make a decision whether to trade certainty for potentially higher profit in the future. If he would invest in a risk free asset, then some small positive return is guaranteed. Investing in a risky asset can yield significantly higher returns, but of course, there is no free lunch. The potentially higher returns are compensated for by the fact that the investor may incur loss. To quantify the degree of riskiness of such assets, several risk measures have been introduced over time. In the following, we will follow mainly Leippold [19] and Cornuejols and Tütüncü [6, p. 275-278] if not specified otherwise.

Definition 2.0.1. Measure of risk
Let $\mathcal{V}$ be the set of real random variables. A risk measure ρ maps the random variable to a real value, i.e. $\rho : \mathcal{V} \to \mathbb{R}$.

Let $\mathcal{R}$ be a random variable representing returns of a portfolio. In the last century, one of the most popular risk measures was the variance of returns of an asset, used famously in the Nobel Prize winning model in Markowitz [21], where a porfolio selection model was formulated that maximised the expected return while minimising the variance in returns. To be precise, variance in returns is defined as

$$\mathbb{E}(\mathcal{R} - \mathbb{E}\mathcal{R})^2.$$

Although at the time the achievement of the Markowitz model was groundbreaking, the use of variance as a risk measure has been a subject of debate. The problem is that variance is symmetric and does not take into account the tails of the distribution of $\mathcal{R}$. To handle the symmetry problem, another risk measure was proposed, the semivariance defined as

$$\mathbb{E}(max(\mathcal{L} - \mathbb{E}\mathcal{L}, 0))^2 = \mathbb{E}(max(-\mathcal{R} + \mathbb{E}\mathcal{R}, 0))^2,$$

where $\mathcal{L} = -\mathcal{R}$ is the loss random variable.

Let us now introduce the notion of a *coherent measure of risk*, which was developed in Artzner et al. [1, Defintion 2.4.] and aims to provide a set of properties that a "nice" risk measure should satistfy.

Definition 2.0.2. Coherent measure of risk
Let $\mathcal{V}$ be the set of real random variables. We say that a risk measure $\rho : \mathcal{V} \to \mathbb{R}$ is coherent if it satisfies the following properties:

1. *Monotony:* $X, Y \in \mathcal{V}, X(\omega) \leq Y(\omega) \, \forall \omega \in \Omega \implies \rho(X) \geq \rho(Y)$.
2. *Subadditivity:* $X, Y, X + Y \in \mathcal{V} \implies \rho(X + Y) \leq \rho(X) + \rho(Y)$.
3. *Positive homogeneity:* $X \in \mathcal{V}, h \geq 0, hX \in \mathcal{V} \implies \rho(hX) = h\rho(X)$.
4. *Translation equivariance:* $X \in \mathcal{V}, a \in \mathbb{R} \implies \rho(X + a) = \rho(X) - a$.

The properties have a quite nice interpretation. The monotony property implies that a portfolio Y that is more favourable in all possible scenarios should have smaller risk compared to the less favourable portfolio X. The subadditivity property pertains to the notion of diversification, as it says that if we combine two portfolios X and Y, the resulting combined portfolio should not be riskier than the portfolios separately. The positive homogenity property implies that lowering/increasing the size of the portfolio by the factor $h \geq 0$, the risk lowers/increases proportionally to h. Last is the property of translation equivariance, which says that increasing the value of portfolio by risk free a, the risk is decreased by a. Unfortunately, neither variance nor semivariance are coherent risk measures.

Let us now introduce another risk measure, the *value at risk*. Using the notation developed in Cornuejols and Tütüncü [6], let $f(x, \Upsilon)$ be a loss function of a vector x which may be considered as a portfolio and a random vector Υ which represents the unknown returns or other random aspects influencing the distribution of loss and denote the loss random variable $\mathcal{L}(x, \Upsilon) = f(x, \Upsilon)$. We assume that the probability distribution of Υ is known and for simplicity that Υ has a probability density $p(y)$. The cumulative distribution function of $\mathcal{L}(x, \Upsilon)$ is then defined as

$$\Psi(x, \upsilon) = \int_{f(x,y) \leq \upsilon} p(y) \, \mathrm{d}y.$$

Definition 2.0.3. Value at risk (Cornuejols and Tütüncü [6, p. 275])
Let α be the chosen confidence level and let $\mathcal{L}(x, \Upsilon)$ have the meaning of loss distribution as defined above with the cumulative distribution function $\Psi(x, \upsilon)$. Then the value at risk $VaR_\alpha(x)$ is defined as

$$VaR_\alpha(x) = q_\alpha(x)$$

where $q_\alpha(x) = \min\{\upsilon \in \mathbb{R} : \Psi(x, \upsilon) \geq \alpha\}$ is the lower α quantile of distribution of $\mathcal{L}(x, \Upsilon)$. α is usually chosen as 0.95 or 0.99.

Value at risk also comes with several advantages and disadvantages. While it is simple to understand and globally accepted by regulators, it is not coherent in general (the subadditivity requirement is not fulfilled), it doesn't quantify the losses exceeding $VaR_\alpha(x)$ and it is not convex (this makes it hard to optimize a portfolio with regard to value at risk).

Due to the aforementioned disadvantages of value at risk, another risk measure was considered. Considering the expected loss exceeding the $VaR_\alpha(x)$ level leads to the notion of *Conditional value at risk*. For an illustration, see Figure 2.1.

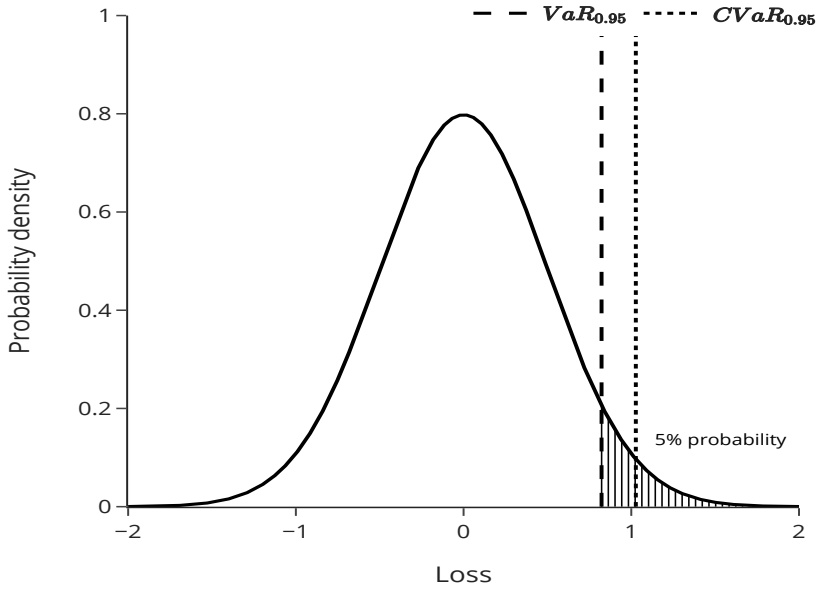


Figure 2.1: Illustration of $VaR_{0.95}$ and $CVaR_{0.95}$. The crosshatching indicates that the indicated area under the curve represents 5% probability.

Definition 2.0.4. Conditional value at risk (Rockafellar and Uryasev [30, Definition 3])
Let α be the chosen confidence level and let $\mathcal{L}(x, \Upsilon)$ have the meaning of loss distribution as defined above and assume that $\mathbb{E}(|\mathcal{L}(x, \Upsilon)|) < \infty$. Then the conditional value at risk or expected shortfall $CVaR_\alpha(x)$ is defined as

$$CVaR_\alpha(x) = \mathbb{E}_{\Psi_\alpha}\left[\mathcal{L}(x, \Upsilon)\right], \tag{2.1}$$

where

$$\Psi_\alpha(x, \upsilon) = \begin{cases} 0, & \upsilon < VaR_\alpha(x), \\ \frac{\Psi(x,\upsilon)-\alpha}{1-\alpha}, & \upsilon \geq VaR_\alpha(x), \end{cases}$$

where $\mathbb{E}_{\Psi_\alpha}$ refers to the fact that the expectation is calculated with regard to the distribution function Ψ_α.

The definition of CVaR may seem a little convoluted. This is due to the fact that the definition is generalised to handle both discrete and continuous distributions. We will be using mainly the discrete case, which is why we present the general version.

Compared to value at risk, conditional value at risk is a coherent risk measure (for the proof, see McNeil et al. [23, Example 2.26.]), but since value at risk is present in its definition, optimizing a portfolio with regard to conditional value at risk according to this definition suffers from many of the same problems as optimizing a portfolio with regard to value at risk. Therefore, a new method has been developed in Rockafellar and Uryasev [29] that allows optimization of conditional value at risk without computing value at risk using linear programming.

2.1 Minimising CVaR using scenarios

In this section, we closely follow the exposition provided in Cornuejols and Tütüncü [6, p. 275-278]. Let

$$F_\alpha(x,\gamma) = \gamma + \frac{1}{1-\alpha}\int_{f(x,y)\geq\gamma}(f(x,y)-\gamma)p(y)\,\mathrm{d}y, \tag{2.2}$$

where $\gamma \in \mathbb{R}$. The function $F_\alpha(x,\gamma)$ has three important properties:

Lemma 1. Properties of $F_\alpha(x,\gamma)$ (Cornuejols and Tütüncü [6, p. 276])
The following three properties hold for $F_\alpha(x,\gamma)$:

1. *It is a convex function of γ.*
2. $VaR_\alpha(x) = \underset{\gamma}{\operatorname{argmin}}\, F_\alpha(x,\gamma)$.
3. $CVaR_\alpha(x) = \min_\gamma F_\alpha(x,\gamma)$.

Proof. For the proof, see Rockafellar and Uryasev [29, Theorems 1 and 2] for the continuous case or Rockafellar and Uryasev [30, Theorems 10, 14 and Corrolary 11] for the general case. □

2.1.1 CVaR formulation

If we want to choose a portfolio x that minimises $CVaR_\alpha(x)$, we can now do so by minimising $F_\alpha(x,\gamma)$ over $x \in \mathcal{X}$ and $\gamma \in \mathbb{R}$ (where $\mathcal{X}$ is some set of portfolios) thanks to the third property in Lemma 1. Of course, Equation 2.2 is not particularly suitable for numerical computations. In practice, as was explained in detail in Chapter 1, the distribution of Υ is approximated using scenarios γ_s with associated probabilities $p_s, s = 1,\ldots,S$.

We can then calculate an approximation of $F_\alpha(x,\gamma)$ as

$$\hat{F}_\alpha(x,\gamma) = \gamma + \frac{1}{1-\alpha}\sum_{s=1}^{S} p_s \max(f(x,\gamma_s)-\gamma, 0). \tag{2.3}$$

We have arrived at the optimization problem

$$\min_{x\in\mathcal{X},\gamma} \hat{F}_\alpha(x,\gamma) = \min_{x\in\mathcal{X},\gamma} \gamma + \frac{1}{1-\alpha}\sum_{s=1}^{S} p_s \max(f(x,\gamma_s)-\gamma, 0). \tag{2.4}$$

A trick can be used to turn Equation 2.4 into a linear programming problem. If we create new variables $z_s \geq 0$ such that $z_s \geq f(x,\gamma_s)-\gamma$, we can write:

$$\min_{x\in\mathcal{X},z_s\geq 0,\gamma} \quad \gamma + \frac{1}{1-\alpha}\sum_{s=1}^{S} p_s z_s, \tag{2.5}$$

$$s.t. \qquad z_s \geq f(x,\gamma_s)-\gamma, s = 1,\ldots,S, \tag{2.6}$$

which is a linear programming problem.

2.1.2 Mean-CVaR formulation

In this section, we present a more precise formulation for practical use adopted from Salahi et al. [32], particularly when we want to choose a portfolio that minimises the conditional value at risk and also allows for controlling the minimum expected return or setting the degree of risk aversion.

Formulation with prescribed minimal expected return

Let $x = (x_1, \ldots, x_n)$ be a vector denoting the weights of each of n assets in a portfolio and consider that $\mu = (\mu_1, \ldots, \mu_n)$ is a random vector representing the returns of the assets. Consider S scenarios, each with probability p_s and let $r_s = (r_{1,s}, \ldots, r_{n,s})$ be the particular realisation of μ in scenario s and let r_0 be the minimal required expected return. For simplicity, we do not allow short selling (condition $x_i \geq 0, i = 1, \ldots, n$). Then we can write

$$\min_{x_i \geq 0, z_s \geq 0, \gamma} \quad \gamma + \frac{1}{(1-\alpha)} \sum_{s=1}^{S} p_s z_s, \tag{2.7}$$

$$\begin{aligned} s.t. \qquad & z_s \geq -\sum_{i=1}^{n} x_i r_{i,s} - \gamma, s = 1, \ldots, S, \\ & \sum_{i=1}^{n} x_i \bar{R}_i \geq r_0, \qquad (2.8) \\ & \sum_{i=1}^{n} x_i = 1, \end{aligned}$$

where $\bar{R}_i = \sum_{s=1}^{S} p_s r_{i,s}$, which is still a linear programming problem. Equation 2.8 assures the prescribed minimal expected return.

Formulation using risk aversion

Another equivalent formulation might be useful when the decision maker does not require a prescribed minimal expected return explicitly, but rather wants to set his risk aversion expectations. This can be achieved by introducing a risk aversion parameter $\lambda \geq 0$ and writing

$$\min_{x_i \geq 0, z_s \geq 0, \gamma} \quad -\sum_{i=1}^{n} x_i \bar{R}_i + \lambda \left(\gamma + \frac{1}{(1-\alpha)} \sum_{s=1}^{S} p_s z_s \right), \tag{2.9}$$

$$\begin{aligned} s.t. \qquad & z_s \geq -\sum_{i=1}^{n} x_i r_{i,s} - \gamma, s = 1, \ldots, S, \\ & \sum_{i=1}^{n} x_i = 1. \end{aligned}$$

2.2 Minimising CVaR using scenarios in the multistage setting

In the multistage setting, the problem is a bit more complicated. Since the returns now do not occur at one single time but rather it is a sequence of returns, the notion of a risk measure must be extended accordingly. For the purposes of this thesis, we focus on the *end of horizon CVaR*, for more advanced topics such as *Nested CVaR model* or *Sum of CVaR model*, we refer the reader to the summary in Kozmík [18, Section 1.4.].

2.2.1 End of horizon CVaR

Definition 2.2.1. End of horizon *CVaR*.
Consider Definition 2.0.4. If we consider the CVaR calculated from the last stage (at the end of the investment horizon), we call it end of horizon CVaR.

The definition of *end of horizon CVaR* is very similar to the definition of regular *CVaR*, with the small difference that the multistage formulation now allows the decision maker to reallocate funds during the investment period (in each stage).

End of horizon CVaR - scenario formulation

We now extend Formulations 2.7 and 2.9 to the multistage case. Consider we want to optimise a portfolio consisting of n stocks over T stages, consider a scenario tree with S leaves and denote sets $\mathbb{S} = \{1, \dots, S\}$ and $\mathbb{I} = \{1, \dots, n\}$. The problem 2.7 can then be reformulated as Equation 2.10 by introducing variables $w_{t,s}$ which represent the wealth in scenario s at time t and tot_s which is the final wealth in scenario s.

$$
\begin{aligned}
\min\ & \gamma + \frac{1}{(1-\alpha)} \sum_{s=1}^{S} p_s z_s, && (2.10)\\
s.t.\quad & z_s \geq -tot_s - \gamma, \forall s \in \mathbb{S},\\
& \sum_{s=1}^{S} p_s tot_s \geq r_0,\\
& w_{1,s} = 1, \forall s \in \mathbb{S},\\
& w_{t,s} = \sum_{i=1}^{n} x_{i,t,s}, \forall s \in \mathbb{S}, \forall t \in \{1, \dots, T-1\}, && (2.11)\\
& w_{t+1,s} = \sum_{i=1}^{n} r_{i,t,s} x_{i,t,s}, \forall s \in \mathbb{S}, \forall t \in \{1, \dots, T-1\}, && (2.12)\\
& tot_s = w_{T,s}, \forall s \in \mathbb{S},\\
& z_s \geq 0, \forall s \in \mathbb{S},\\
& x_{i,t,s} \geq 0, \forall s \in \mathbb{S}, \forall i \in \mathbb{I}, \forall t \in \{1, \dots, T-1\},\\
& \gamma \in \mathbb{R},\\
& + \text{nonanticipativity constraints},
\end{aligned}
$$

The initial wealth $w_{1,s}$ is set to 1 and in each stage, the wealth increases by the returns obtained in the previous stage (Equation 2.12) and is distributed again (Equation 2.11, at the end of the investment horizon we do not need to distribute the wealth into assets again). $r_{i,t,s}$ is the return obtained from stock i at stage t in scenario s (we consider returns indexed by t to occur at the end of stage t (after the portfolio allocations $x_{i,t,s}$ are set), so that Equation 2.12 makes sense). The returns $r_{i,t,s}$ are calculated as

$$r_{i,t,s} = \frac{price_{i,t+1,s}}{price_{i,t,s}},$$

where $price_{i,t,s}$ is the price of asset i at the beginning of stage t in scenario s. This justifies Equation 2.12.

Remark. The cautious reader would expect the equation for total scenario return $tot_s = w_{T,s}$ to read $tot_s = w_{T,s}/w_{1,s}$. This would make the program nonlinear and we overcome this problem by using the assumption $w_{1,s} = 1$.

Remark. We do not include the nonanticipativity constraints in the above formulation, since they must be specified explicitly according to the structure of the scenario tree. To illustrate the explicit formulation of nonanticipativity constraints, consider Figure 1.1. In the context of the above problem, there are 6 scenarios $(111, 112, 113, 121, 122, 123)$. For this illustration, let $\mathbf{x}_{t,s}$ be the vector of allocations to each asset at time t and in scenario s. The nonanticipativity constraints then are:

$$\mathbf{x}_{t1,111} = \mathbf{x}_{t1,112} = \mathbf{x}_{t1,113} = \mathbf{x}_{t1,121} = \mathbf{x}_{t1,122} = \mathbf{x}_{t1,123}$$
$$\mathbf{x}_{t2,111} = \mathbf{x}_{t2,112} = \mathbf{x}_{t2,113}, \mathbf{x}_{t2,121} = \mathbf{x}_{t2,122} = \mathbf{x}_{t2,123}.$$

Similarly, the problem 2.9 can then be reformulated as Equation 2.13:

$$\begin{aligned}
\min \quad & -\sum_{s=1}^{S} p_s tot_s + \lambda \left(\gamma + \frac{1}{(1-\alpha)} \sum_{s=1}^{S} p_s z_s \right), && (2.13)\\
s.t. \quad & z_s \geq -tot_s - \gamma, \forall s \in \mathbb{S},\\
& w_{1,s} = 1, \forall s \in \mathbb{S},\\
& w_{t,s} = \sum_{i=1}^{n} x_{i,t,s}, \forall s \in \mathbb{S}, \forall t \in \{1, \ldots, T-1\},\\
& w_{t+1,s} = \sum_{i=1}^{n} r_{i,t,s} x_{i,t,s}, \forall s \in \mathbb{S}, \forall t \in \{1, \ldots, T-1\},\\
& tot_s = w_{T,s}, \forall s \in \mathbb{S},\\
& z_s \geq 0, \forall s \in \mathbb{S},\\
& x_{i,t,s} \geq 0, \forall s \in \mathbb{S}, \forall i \in \mathbb{I}, \forall t \in \{1, \ldots, T-1\},\\
& \gamma \in \mathbb{R},\\
& + \text{nonanticipativity constraints},
\end{aligned}$$

where $\lambda \geq 0$ is a risk aversion parameter and the nonanticipativity constraints would again need to be provided explicitly according to the structure of the scenario tree.

Chapter 3

Reinforcement learning

Reinforcement learning is a machine learning paradigm inspired by the natural learning process of humans – learning by interacting with an environment. All actions we take in our daily lives are in some way punished or rewarded.

For an example, consider touching a hot stove. An immediate negative reward (pain) is received and one learns quickly not to do it again. On the other hand, eating something sweet usually produces a feeling of pleasure (positive reward) and that makes us want to eat more sweets.

Reinforcement learning methods work in pretty much the same way: an agent is placed in an artificial environment and based on the actions it takes, it receives rewards (positive or negative) and learns to perform the actions that yield the most positive rewards. This is in constrast to the other machine learning paradigms (supervised and unsupervised learning), where no environment exists and the model is taught by minimising some kind of loss over a given dataset.

In the recent years, machine learning has seen a large surge in activity due to rising computational power and this has not avoided the field of reinforcement learning. Many large institutions and corporations have built teams that specialise in reinforcement learning and have produced groundbreaking results in many disciplines, ranging from beating the best player in the world in the game of Go (see Silver et al. [37]), solving the protein folding problem (see Jumper et al. [17]), beating some of the best teams in Dota 2 (see Berner et al. [3]) or most recently, finding a faster matrix multiplication algorithm than current state of the art (see Fawzi et al. [10]).

In this chapter, we aim to provide the necessary exposition of reinforcement learning methods used in the computational part of this thesis. We mainly follow Sutton and Barto [38].

3.1 Basic definitions

As we mentioned, the agent operates in an environment. The agent is aware of the environment and based on the current state of the environment chooses an action. The action is usually chosen as the action that maximises the expected cumulative reward (this may not always be the case, as choosing a less optimal action might be beneficial, we will touch on this in Section 3.2). After the agent performs an action, the state of the environment changes and the agent receives a reward. This happens in a sequence of discrete timesteps until a terminal state

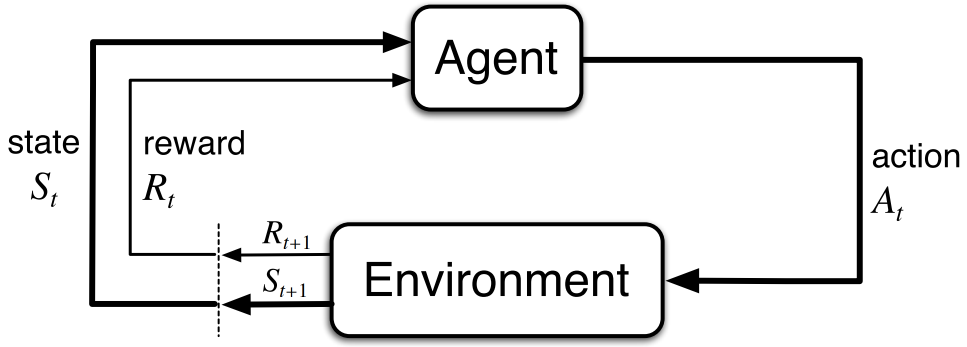


Figure 3.1: Illustration of the agent environment interaction. Agent is in state S_t and received reward R_t for the last chosen action A_{t-1}, then A_t is chosen by the agent and new state S_{t+1} and reward R_{t+1} are obtained. Image sourced from Sutton and Barto [38, Figure 3.1.].

of the environment is reached (i.e., if the last chosen action led to winning/losing in the game of chess). One iteration of solving an environment from the initial to the terminal state is called an *episode*. This is best illustrated by the chart that can be found in almost all reinforcement learning books, see Figure 3.1.

We now introduce the notion of a *Markov decision process*, which is a formalisation of the agent environment interaction discussed above. The following is heavily inspired by Sutton and Barto [38, Chapter 3]

Definition 3.1.1. Markov decision process (Zelinka [46, Definition 2.1.2.], Sutton and Barto [38, Chapter 3])
A Markov decision process is a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$*, where*

1. $\mathcal{S}$ *is the set of all possible states,*
2. $\mathcal{A}$ *is the set of all possible actions,*
3. $\mathcal{P}(s_{t+1}|s_t, a_t) = P(S_{t+1} = s_{t+1}|S_t = s_t, A_t = a_t)$ *is the probability that choosing action* a_t *in state* s_t *yields state* s_{t+1}*,*
4. $\mathcal{R}(s_t, a_t) = R_{t+1}(S_t = s_t, A_t = a_t)$ *is the reward received by choosing action* a_t *in state* s_t*,*
5. $\gamma \in [0, 1)$ *is the discount factor,*

where S_t, A_t and R_t are random variables representing the state, action and reward at timestep t. The name Markov decision process is not a coincidence and is related to the Markovian property. Notice that the state transition probabilities depend only on previous state and chosen action and not on the preceeding history (Zelinka [46, p.15]). This means that the state must contain all information and no information is carried by the previously visited states and taken actions. Of course, this is a simplification and does not hold in real life (for example, the history of moves can hold information about the strength of an opponent in the game of chess), but it suffices to model even complex phenomena and allows for precise mathematical treatment.

The agent chooses an action based on the current state to maximise the expected cumulative reward. A function that maps states to probabilities of actions is termed *policy*, see Definition 3.1.2.

Definition 3.1.2. Policy (Sutton and Barto [38, Section 3.5])
Let $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Then the policy is defined as

$$\pi(a|s) = P(A_t = a|S_t = s).$$

Another fundamental concept is the value function, see Definition 3.1.3.

Definition 3.1.3. Value function (Sutton and Barto [38, Section 3.5])
Let π be a policy and $s \in \mathcal{S}$. Then we define the value function $v_\pi(s)$ as

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right],$$

where the subscript π in $\mathbb{E}_\pi$ refers to the fact that the agent acts according to policy π. The value function assigns each state the expected cumulative discounted reward – the reward that the agent may expect to gain from state s_t into the future. The discount factor γ weights the future rewards by how far into the future they may be attained.

Another related notion is the *action value function*, see Definition 3.1.4.

Definition 3.1.4. Action value function (Sutton and Barto [38, Section 3.5])
Let π be a policy and $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Then we define the action value function $q_\pi(s, a)$ as

$$q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right].$$

The action value function is much the same as value function, but it maps the actions taken in a state to the expected cumulative discounted reward rather than just states. The action value function allows the agent to not always take the immediate most rewarding (greedy) action, but rather optimise the reward while taking into account the possible following states and actions.

The value function and action value functions must be learned by exploring the environment. Unfortunately, the plethora of theory about estimation of these functions (e.g. using the Bellman equations) is out of scope of this thesis. We refer the interested reader to Sutton and Barto [38, Section 3.5.] and the references therein.

Another related notion, which builds on the definitions given above is the *advantage function*, see Defintion 3.1.5.

Definition 3.1.5. Advantage function (Mnih et al. [25, Section 3])
Let π be a policy and $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Denote the value function as $v_\pi(s)$ and the action value function as $q_\pi(s, a)$. Then we define the advantage function as

$$A_\pi(s, a) = q_\pi(s, a) - v_\pi(s).$$

The advantage represents the gain that is obtained by taking action a in state s compared to following policy π.

Our aim is to obtain a policy that maximises the expected cumulative discounted reward. We thus define the *optimal policy*, *optimal value function* and *optimal action value function*, see Definition 3.1.6.

Definition 3.1.6. Optimal policy, value function and action value function (Sutton and Barto [38, Section 3.6])
Let $R(\pi)$ be the expected cumulative discounted reward obtained by following policy π. Then we define the optimal policy as

$$\pi^* = \underset{\pi}{argmax}\, R(\pi).$$

Similarly, the optimal value function and optimal action value function are given by:

$$v^*(s) = \max_{\pi} v_\pi(s)$$

and

$$q^*(s,a) = \max_{\pi} q_\pi(s,a).$$

respectively for $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

3.2 Exploration vs exploitation

The optimal policy, value function and action value function must be learned by interacting with the environment. The agent now faces a dillema – either to maximise his known reward (act greedily, but potentially get stuck with a policy that is not optimal) or explore the environment and update the policy in order to get the (globally) optimal policy. This exploration-exploitation tradeoff is always present with reinforcement learning and many approaches for dealing with it exist. An example are the ϵ-greedy methods, where the agent acts greedily $(1-\epsilon)\,\%$ of the time and performs a random action $\epsilon\,\%$ of the time. ϵ is usually set to a value close to 1 at the beginning of training and decreased over time, the final threshold at which ϵ is kept constant is usually somewhere in $[0, 0.1]$. For more information, we refer the reader to Sutton and Barto [38, Section 2.7.].

3.3 Algorithm classes

In this section, we aim to present a basic summary of current reinforcement learning methods with particular focus on policy-gradient methods.

3.3.1 Model free vs. model based algorithms

The most fundamental dividing line between reinforcement learning algorithms is whether the agent is given a model of the environment which allows the agent to take into account future states before they are experienced. The model is represented by the state transition function $\mathcal{P}$ as defined in Definition 3.1.1. Having the model in hand obviously helps the agent learn tremendously, but having such a model in practice is quite rare, thus the model free methods are being used much more extensively. We focus on model free algorithms in this text.

3.3.2 Model free methods

In comparison to model based methods, the model free methods learn by trial and error. Examples of these methods are e.g. Monte Carlo Sampling, SARSA, Q-learning, Actor critic, Proximal policy optimization (PPO) and Trust region policy optimization (TRPO). These methods can be divided into two groups – value based methods and policy based methods. In this thesis, we focus particularly on the Proximal policy optimization method (as it will be used in the computational part of this thesis for reasons explained later), but we also give an introduction to Q learning, as it helps with understanding of how deep neural networks are used in the field of reinforcement learning, the Actor critic architecture and the Trust region policy optimization. We assume that the reader is familiar with basics of deep learning (such as architecture of neural networks and basic algorithms for training them such as stochastic gradient descent), a great introduction can be found in Goodfellow et al. [12, Part 2].

Value based methods

Q learning

In this section, we follow Sutton and Barto [38, Section 6.5.]. The most famous example of a value based method is Q learning. Let $q^*(s,a)$ denote the optimal action value function as defined in Definition 3.1.6 and let $Q(s,a)$ be an estimate of $q^*(s,a), s \in \mathcal{S}, a \in \mathcal{A}$. If the sets $\mathcal{S}$ and $\mathcal{A}$ are finite, then the values of $Q(s,a)$ can be represented by a table and updated according to the updating rule presented in Equation 3.1.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \varphi \left[r_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1},a) - Q(s_t,a_t) \right], \tag{3.1}$$

where φ is the learning rate, γ is the discount factor, s_t, a_t are the current state and current chosen action respectively, s_{t+1} is the next state following action a_t, r_{t+1} is the reward obtained by choosing action a_t and the subscript t is added to emphasize the transition between current and next step. The whole algorithm can be summarised as follows:

Algorithm 1 Vanilla Q-learning (Sutton and Barto [38, p. 131])

Input: Choose the learning rate $\varphi \in (0,1]$ and exploration parameter $\epsilon > 0$, number of episodes, initialise $Q(s,a)$ randomly, $s \in \{s \text{ for } s \text{ in } \mathcal{S} \text{ if } s \text{ is not terminal}\}$ and $Q(s,a) = 0$ for s terminal, $a \in \mathcal{A}$ (terminal state of the environment means that the episode is finished).

for episode in {1,...,number of episodes} **do**
  $s_t \leftarrow s_0$ Reset environment state
  while s_t is not terminal **do**
    $a_t \leftarrow$ Choose action a_t using an epsilon-greedy policy according to $Q(s_t,\cdot)$.
    $s_{t+1}, r_{t+1} \leftarrow$ act on action a_t, obtain new state s_{t+1} and reward r_{t+1}
    Update $Q(s_t,a_t)$ using Equation 3.1
    $s_t \leftarrow s_{t+1}$
  end while
end for

It has been shown that under the assumption that all state-action pairs continue to be updated during training, then Q converges to q^* almost surely, see Sutton and Barto [38, p. 131]. This variant of Q-learning has an obvious problem – it depends on a table for keeping the values of the Q function and thus does not generalise to complex state spaces (such as infinite ones). To tackle this problem, a neural network has been introduced in place of the $Q(s, a)$ table as a function approximator, which we can then write as $Q(s, a; \theta)$, where θ are the weights of the neural network. This approach has been popularised by Mnih et al. [24], where they used Q-learning along with a neural network as a function approximator (and many other particular improvements such as experience replay, that are unfortunately out of the scope of this thesis) to achieve superhuman performance on several Atari 2600 games. The reader interested in deep Q-learning can find the algorithm in Mnih et al. [24, Algorithm 1].

Policy approximation methods

The theory of value based methods assumed that we estimate the value function and action value function and then based on them somehow choose the policy (such as taking the action with maximum value). However, another approach is possible – modelling the policy explicitly. In this section, we mostly follow Sutton and Barto [38, Chapter 13] and Schulman et al. [34, Section 2].

Policy approximation methods assume that the policy $\pi(a|s; \theta), a \in \mathcal{A}, s \in \mathcal{S}$ is dependent on parameters $\theta \in \mathbb{R}^d$ where $d \in \mathbb{N}$. In general, the aim of these methods is to maximise some kind of performance measure $J(\theta)$. The performance measure J is chosen such that the gradient $\nabla_\theta J$ exists and by estimating the gradient as $\widehat{\nabla_{\theta_\tau} J(\theta_\tau)}$, the policy can be optimised using stochastic gradient ascent (where the subscript τ implies that this estimate is computed in the τ-th update in the stochastic gradient ascent process). In practice, $\widehat{\nabla_{\theta_\tau} J(\theta_\tau)}$ is estimated by averaging a batch of samples, we will denote this as $\mathbb{E}_b \widehat{\nabla_{\theta_\tau} J(\theta_\tau)}$ where the symbol $\mathbb{E}_b$ refers to the average of a batch of samples. We can then write the very general update rule

$$\theta_{\tau+1} = \theta_\tau + \varphi \mathbb{E}_b \widehat{\nabla_{\theta_\tau} J(\theta_\tau)},$$

where φ is the learning rate. Due to the aforementioned updating rule, these methods are also often called *policy gradient methods*.

Remark. When using policy approximation methods, we do not need to randomly sample actions $\epsilon\%$ of the time to ensure exploration. All that is needed is to ensure that the policy does not become deterministic. This can be achieved by ensuring that $\pi(a|s; \theta) \in (0, 1)$, see Sutton and Barto [38, Section 13.1].

A significant theoretical advantage compared to value based methods is that "*with continuous policy parameterization the action probabilities change smoothly as a function of the learned parameter, whereas in ϵ-greedy selection the action probabilities may change dramatically for an arbitrarily small change in the estimated action values, if that change results in a different action having the maximal value. Largely because of this stronger convergence guarantees are available for policy-gradient methods than for action-value methods*" Sutton and Barto [38, Section 13.2]. We refer the reader to the Policy gradient theorem located therein.

There exist also hybrid methods between policy gradient methods and value based methods, where the policy, value function and action value function are all learned – such methods are called *actor critic* methods.

Advantage actor critic

In this section, we present the theory behind the (asynchronous) advantage actor critic (A3C) algorithm as developed in Mnih et al. [25], as it shows well the structure of the neural net that is used in the PPO algorithm.

The advantage actor critic is a policy approximation algorithm that learns not only the policy but also the action value function. Let us first decipher the name of the algorithm. Advantage refers to the advantage function defined in Definition 3.1.5, actor refers to the learned policy approximation and critic refers to the learned action value function approximation (both the actor and the critic are neural networks that are used as function approximators). In practice, the actor and critic networks share some parameters (they can be thought of as a single neural net with diverging structure, such as a first shared layer and then diverging such that the second layer is not shared between the actor and the critic).

Denote the weights of the actor as θ and weights of the critic as θ_v, thus the estimated policy can be written as $\pi(a|s;\theta)$, $a \in \mathcal{A}, s \in \mathcal{S}$. The performance measure J that A2C maximises can be written as

$$\log(\pi(a|s;\theta))\widehat{A}(s,a;\theta,\theta_v),$$

and the gradient of the performance measure can be written as

$$\mathbb{E}_b \nabla_{\theta'} \log(\pi(a_b|s_b;\theta'))\widehat{A}(s_b,a_b;\theta,\theta_v),$$

where the gradient is taken only with respect to the actor variables affecting the policy (the advantage can be thought of as constant with regard to the differentiation) and where we add the subscript b to a_b and s_b to indicate that they belong to a batch of samples and where $\widehat{A}(s_b,a_b;\theta,\theta_v)$ is the estimated advantage function (for details on how it can be estimated, see Mnih et al. [25, Section 4]). Note that this update only updates the policy and not the value function. A different updating scheme is used for the parameters of the critic, where a quadratic error between the estimated value function and observed rewards is minimised, the details can be found in the original paper Mnih et al. [25, Algorithm S3].

In the original paper, they made use of asynchronous updates to supposedly improve training stability. It was later shown in the paper Wu et al. [43] that the asynchronicity provides no added benefit in performance.

3.3.3 Trust region policy optimization

The trust region policy optimization method, developed in Schulman et al. [33], is a special case of policy gradient methods, as it introduces a special constraint on the policy parameters, such that the change in the policy is not too large at each step. This is done by imposing the constraint that the Kullback-Leibler Divergence between the two policies is not too large. The Kullback-Leibler Divergence is defined in Definition 3.3.1.

Definition 3.3.1. Kullback-Leibler Divergence
Let P and Q be discrete random variables with the same support $\mathcal{S}$. Let $P(x)$ and $Q(x)$ denote the probability distribution functions of P and Q, $x \in \mathcal{S}$. Then the Kullback-Leibler Divergence, denoted $D_{KL}(P, Q)$ is calculated is

$$D_{KL}(P, Q) = \mathbb{E}_P \log(\frac{P}{Q}) = \sum_{x \in \mathcal{S}} P(x) \log(\frac{P(x)}{Q(x)}),$$

where the subscript P in $\mathbb{E}_P$ denotes that the expectation is taken with respect to the probability distribution of the random variable P.

D_{KL} quantifies the difference between discrete probability distributions. More details can be found in Shlens [36].

In practice, $D_{KL}(\pi_A, \pi_B)$ between two policies π_A and π_B is bounded from above using some parameter δ. At each step, the optimization problem that is being solved to update the policy is given in Equation 3.2.

$$\begin{aligned} &\max_{\theta_{\tau+1}} \mathbb{E}_b \frac{\pi(a_b, s_b|\theta_{\tau+1})}{\pi(a_b, s_b|\theta_\tau)} \widehat{A}(a_b, s_b) \\ &s.t. \qquad \mathbb{E}_b D_{KL}(\pi(\cdot, s_b|\theta_\tau), \pi(\cdot, s_b|\theta_{\tau+1})) \leq \delta, \end{aligned} \tag{3.2}$$

where $\theta_{\tau+1}$ are the new parameters of the policy after the update and $\widehat{A}(a_b, s_b)$ is an estimate of the advantage function, where again the subscript b is added to imply that a_b and s_b belong to a batch of samples. The objective function that is maximised here is a local approximation of a quantity that *"represents the expected return of another policy $\pi(\cdot, \cdot|\theta_{\tau+1})$ in terms of the advantage over $\pi(\cdot, \cdot|\theta_\tau)$"*, see Schulman et al. [33, Equation 1]. For details (such as how $\widehat{A}(a_b, s_b)$ is calculated[1]), see Schulman et al. [33, Section 2-4].

[1] which we do not show, as it would require developing needlessly complex notation and it is not particularly relevant for our purposes

3.3.4 Proximal policy optimization

The Proximal policy optimization algorithm (PPO) was developed in Schulman et al. [34] by combining a neural network used for estimation of action value function and the policy approximation with the trust region idea (limiting the magnitude of change in KL diveregence) used in TRPO, we follow their exposition. Consider Equation 3.3.

$$L_{CLIP}(\theta'_\tau, \theta_\tau) = \min(r(\theta'_\tau, \theta_\tau)\widehat{A}(a,s),\ \text{clip}(r(\theta'_\tau, \theta_\tau), 1-\delta, 1+\delta)\widehat{A}(a,s))), \tag{3.3}$$

where $r(\theta'_\tau, \theta_\tau) = \frac{\pi(a,s|\theta'_\tau)}{\pi(a,s|\theta_\tau)}$, δ is a hyperparameter and

$$\text{clip}(a, 1-\delta, 1+\delta) = \min(1+\delta, \max(a, 1-\delta)), a \in R.$$

The first term in Equation 3.3 is the same as in the objective function of the TRPO method, see Equation 3.2. The second term in the minimum clips the ratio $r(\theta'_\tau, \theta_\tau)$, which has the effect of limiting the change of the policy so that the change is not too large (controlled by δ). The minimum in Equation 3.3 is then taken to get a lower bound on the same objective as used in TRPO.

$L_{CLIP}(\theta'_\tau, \theta_\tau)$ is then combined with a value function error term $L_{VF}(\theta'_\tau)$ and potentially also an entropy term (which is omitted in this exposition), giving rise to the following performance measure L as given in 3.4.

$$L(\theta'_\tau, \theta_\tau) = L_{CLIP}(\theta'_\tau, \theta_\tau) - c_1 L_{VF}(\theta'_\tau), \tag{3.4}$$

where c_1 is a hyperparameter and $L_{VF}(\theta'_\tau) = (v_{\theta'_\tau}(s) - v^{target})^2$, where v^{target} is the observed cumulative return of the state s obtained during training. This performance measure is then maximised using stochastic gradient ascent (again, the stochastic gradient ascent update is performed using $\mathbb{E}_b \nabla_{\theta'_\tau} L(\theta'_\tau, \theta_\tau)$, which is the average over a batch of samples). Note that while the parameters θ'_τ are shared for the policy approximation and the value function approximation $v_{\theta'_\tau}$, a similar shared neural net architecture as was used in the actor critic framework can be used here such that only some of the parameters are shared (such as a common first layer). Particularly, A2C is a special case of PPO as was shown in Huang et al. [14].

Empirically, PPO performs better than A2C and TRPO and is more sample efficient (requires less timesteps to reach a given level of performance). Particularly, in a blogpost, OpenAI said that *"it has become the default reinforcement learning algorithm they use due to its ease of use and good performance"*, see OpenAI [26].

Chapter 4

Optimal scenario tree selection

In this chapter we propose an experiment to find out if it is possible to predict the optimal scenario tree structure with regard to the objective function and also propose a way to control for the complexity of the scenario tree using reinforcement learning. For this purpose, we implemented the moment matching method for generation of scenario trees, the multistage mean-CVaR model and a reinforcement learning agent.

4.1 Methods

The whole implementation was programmed in Python (Van Rossum and Drake [39, Version 3.11]), mathematical optimization problems were implemented in GAMS (GAMS Development Corporation [11, version 40.3.0]) using the Python API. Data were sourced from Yahoo Finance [44] using the yfinance [45, version 0.1.74] package. The reinforcement agent was implemented using the *Stable baselines 3* package (Raffin et al. [28, version 1.6.2]) and the environment was implemented using the *gym* package (Brockman et al. [4, version 0.21.0]).

4.2 Data

For our experiments, we used data obtained from Yahoo Finance [44] using the yfinance [45, version 0.1.74] package. We downloaded historical weekly asset price data from 1.1.2000 to 31.12.2019 for 49 financial stocks given in Table A.1. We consider two indexing sets with regard to time, set *train* (from 1.1.2000 to 31.12.2009) and set *test* (from 1.1.2010 to 31.12.2019) and also consider two sets of assets, set A (see Table A.2) and set B (see Table A.3). This yielded us four distinct sets:

- (*train*, A), denoted TrA,
- (*test*, A), denoted TeA,
- (*train*, B), denoted TrB,
- (*test*, B). denoted TeB,

and we write the set that contains all these sets as $\kappa = \{TrA, TeA, TrB, TeB\}$. We trained the agent on the set TrA and evaluate its performance on all four sets, to evaluate whether the agent is able to:

1. learn something from the training data (performance on TrA),
2. generalise to unseen assets in the same period (performance on TrB),
3. generalise to the training assets in the future (performance on TeA),
4. generalise to unseen assets in the future (performance on TeB).

Remark. We are mainly interested in the performance of the agent on sets TrA and potentially also TrB. If the agent was able to generalise across time, this would be a breaktrough finding. Such generalisation is however unlikely due to the possibility that the distribution of the data may be different in the *test* time period.

We set the investment horizon to 2.5 years and from each set in κ, we needed to obtain data for the moment matching method in such a way that the scenario trees were constructed with the same investment horizon independent of the number of stages. This was achieved by splitting the investment horizon into equisized parts based on the number of stages of the tree. Particularly, for a scenario tree with a given number of stages (denoted as T), we split the investment period into T equisized periods (we denote the length of these periods as $\mathcal{L}$), each corresponding to the given stage.

We then used the whole available 10 years of data to estimate the distribution of returns in a period of length $\mathcal{L}$ by splitting the 10 years of data into equisized parts of length $\mathcal{L}$, calculating simple returns (this yielded $4T$ observations), and calculating the first four sample moments and correlations between each of the assets. These moments and correlations were then used as input for the moment matching method, where the obtained moments and correlations were used for generating each stage of the scenario tree.

Remark. We considered only stagewise independent scenario trees. This is in line with the data preparation we used above, as financial returns are generally considered independent when taken over a period of time that is longer than a few days. The shortest period we used is about 6 months (which corresponds to an investment period of 2.5 years with 5 stages), which is much longer than a few days.

4.3 Environment

For the purposes of training the reinforcement agent to choose the best scenario tree structure, we adapted the well known GridWorld environment to represent iterative stage by stage building of the scenario tree.

Remark. The GridWorld environment is a well known introductory environment for training reinforcement learning agents. It consists of a n by n grid, $n \in \mathbb{N}$, where the agent starts on a given tile (a position on the grid, i.e. for example the bottom left corner) and must reach a target tile and upon reaching the tile, it receives a reward. The agent can perform 4 actions – move up, move down, move

left and move right. None of the tiles apart from the target tile return rewards. An illustration of such an environment is given in Figure 4.1.

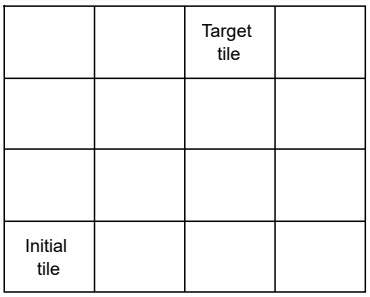


Figure 4.1: Illustration of a 4 by 4 GridWorld environment. The agent starts at the Initial tile and must reach the Target tile to receive a reward.

We designed and implemented a custom GridWorld environment, which we call *Tree Building Environment*.

4.3.1 Tree Building Environment

Tree Building Environment is an adaptation of the GridWorld environment to allow the agent to build scenario trees based on given predictors, which consist of return data of the assets that are used to build the scenario tree. The state is given as a 8 by 8 tree building grid (which starts filled with zeros only in the initial state) and the set of 9 predictors.

At the beginning of each episode, between 7 and 10 assets are randomly chosen (with uniform probabilities for the number of assets) from the provided data (a set from κ) and the set of predictors is obtained at the beginning of each episode using the return data for the chosen assets for the whole time period and is constant throughout the episode.

Each row in the tree building grid represents a part of the tree building process. The first row corresponds to the chosen number of stages in the tree. The following rows each represent the chosen branching (number of descendants of each node) in each stage, from the first to last.

In each state, the agent can take any action in the set $\{3, 4, 5, 6, 7\}$, which we from now on refer to as *action set*. In the initial state, we allow the agent to perform only actions $\{3, 4, 5\}$ from the action set to choose the depth of tree and upon performing any of these actions, 1 is placed at the corresponding position in the first row. If the agent chooses action 6 or 7, it is forced to perform action 5[1].

[1]We do not allow the number of stages to be 6 or 7, as the tree is then considerably more complex and building it based on historical data and solving the resulting mean-CVaR model takes too much time for it to be practical for our experiments. The action 5 is forced for the reinforcement algorithm to be able to associate action 6 and 7 with the move to a state where action 5 was chosen.

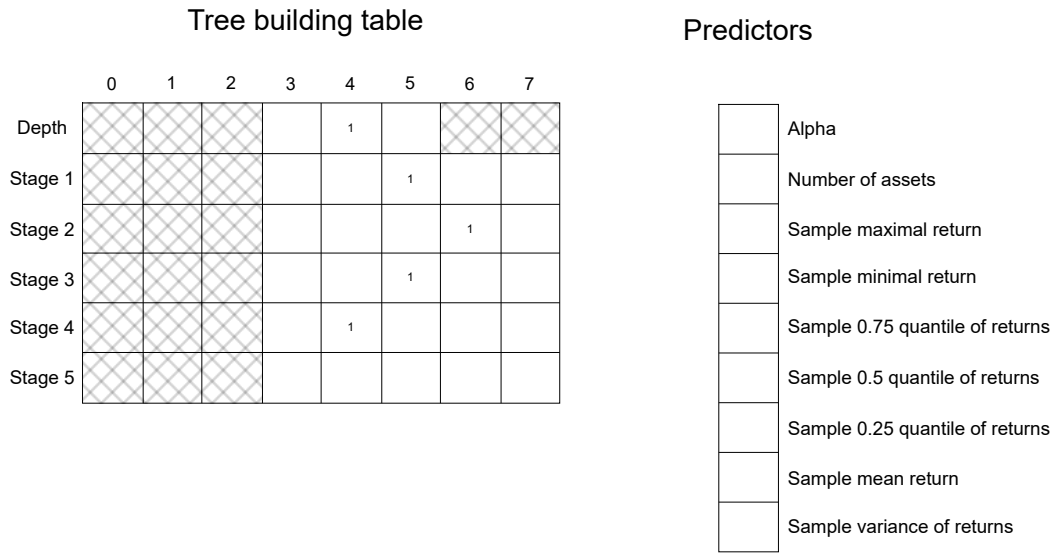


Figure 4.2: Tree Building Environment. State illustration when a 4 stage tree is generated with 5 children in the first stage, 6 in the second stage, 5 in the third stage and 4 in the fourth stage. Crosshatching represents invalid actions. Predictors are represented as an empty array, in reality they are populated with numerical data on the whole period returns, see Section 4.3.2.

In the following states, where the agent chooses the branching (the number of descendants) in each stage, the agent can perform any action in the action set and again upon performing each action, 1 is placed at the position of the chosen action in the next row (if the action is valid, see below).

To obtain reasonable trees, we had to constrain the size of the tree the agent can take (the maximum number of scenarios). We always require that the tree must have at least 100 scenarios and at most 1200. In each state, we check if it is possible to perform the chosen action such that a final tree that remains within these limits is possible. If it is not possible, the action is considered invalid and the agent is forced to take the maximum[2] valid action (where valid action refers to any action for which building a tree that stays in the given limits is possible). In case the chosen action was invalid, 1 is placed at the position of the forced maximum valid action.

When the agent has taken as many actions as the chosen depth of tree, the episode ends, the mean-CVaR problem is solved using the given scenario tree structure and the obtained reward is returned (see Section 4.4). An illustration of the environment can be found in Figure 4.2.

4.3.2 Predictors

At the beginning of each episode, between 7 and 10 assets are randomly chosen (with uniform probabilities for the number of assets) from the provided data and simple returns are calculated for the whole time range. α is randomly chosen with uniform probabilities from the set $\{0.8, 0.85, 0.9, 0.95\}$ and the following predictors are provided to the agent:

[2]we use the maximum valid action, rather than a random action, so that the agent is able to associate that an invalid action results in taking a large branching in the given stage

1. α (parameter of mean-CVaR model),
2. Number of sampled assets,
3. Sample maximum return,
4. Sample minimum return,
5. Sample 0.75 quantile of returns,
6. Sample 0.5 quantile of returns,
7. Sample 0.25 quantile of returns,
8. Sample mean of returns,
9. Sample variance of returns.

The predictors are then constant throught the entire episode.

4.4 Rewards

A reward is returned after every action the agents takes, which is 0 for every action in states that are not terminal and at the end of the episode, in the terminal state, a reward is returned based on the solved mean-CVaR problem using the scenario tree structure that is specified by the terminal state.

Remark. Note that the fact that the reward is sparse (reward 0 is returned when not in a terminal state) is actually quite a common occurence in reinforcement learning. Reinforcement learning is, after all, designed exactly to handle such problems.

We need to specify the reward in such a way that a higher reward corresponds to a more favourable value of the objective function. We need to consider that the mean-CVaR model, as formulated in Equation 2.13, is formulated using the distribution of loss. This means that obtaining a smaller objective value from the mean-CVaR model is beneficial. Denoting the obtained objective value from solving the problem in Equation 2.13 as ς, we have to write the reward given to the agent as

$$r_{terminal} = -\varsigma(s_{terminal}),$$

where we add the subscript *terminal* to emphasize that this reward is calculated at the terminal state of the episode and that the reward depends on the terminal state (which represents the scenario tree structure). The objective of the agent is then to maximise $r_{terminal}$.

4.4.1 Penalty

As was mentioned in Section 1.2.6, choosing a scenario tree that has too many stages and too many descendants in each stage leads to a computationally intractable problem. On the other hand, choosing a scenario tree that is too small in terms of number of stages and with too few descendants in each stage may

lead to a very rough approximation of the underlying continuous distributions, leading to results with high variability.

To ameliorate these problems, we propose to include a penalty term in the reward $r_{terminal}$ which penalizes such scenario trees. Particularly, we propose that the penalty be dependent on the number of scenarios in the tree in the last stage, i.e. the number of leaves in the tree. It is not straightforward to represent the complexity of a tree due to the multidimensional structure, but we consider that using the number of leaves provides a good enough proxy for the complexity of the scenario tree, while being simple to implement.

Denoting the number of leaves in the tree as Ψ and the penalization function as $\delta(\Psi)$, we thus propose that the reward $r_{terminal}$ be penalized as follows

$$r_{terminal} = -\varsigma(s_{terminal}) - \delta(\Psi(s_{terminal})),$$

where the dependence of Ψ on $s_{terminal}$ stresses the fact that $\Psi(s_{terminal})$ is calculated from the scenario tree structure represented in $s_{terminal}$.

To penalize the scenario tree that is too complex, we propose a linear penalization δ_1

$$\delta_1(\Psi) = c\frac{\Psi - \Psi_{min}}{\Psi_{max} - \Psi_{min}},$$

where c is a chosen coefficient (which must be chosen based on the magnitude of values obtained as solutions from solving the mean-CVaR problem) and Ψ_{max} and Ψ_{min} are the maximum and minimum allowed number of leaves in the scenario tree respectively.

Of course, this is just one possible penalization function out of infinite possibilities. The shape of the penalty function can be adjusted based on the problem at hand (and the parameter c has to be adjusted as well).

In this section, we proposed only a penalization function that penalizes trees that are too complex. It might make sense to penalise also trees that are too simple, but we do not use such a penalisation in this thesis, as we already have a lower bound set on the number of leaves in the tree in the environment (the lower bound is 100 scenarios).

4.5 Implementation

4.5.1 Moment matching

We used the moment matching method in the form given in Definition 1.2.7 sequentially on each stage using the first four sample moments and correlations which were estimated as explained in Section 4.2 with one small adjustment.

When looking at the generated scenarios for larger numbers of descendants, we noticed that usually, only 3 scenarios with positive probabilities were generated and the rest had almost zero probability. This would fundamentally change the properties of scenario trees that we want to explore (dependence of objective function on tree size), since then we might think we are using a large tree, which in reality is much smaller due to the scenarios with zero probability.

To counteract this effect, we added the constraint $p_j \geq 0.03$ to the implementation of Definition 1.2.7, which solved the problem. With the notation developed

in Definition 1.2.7, the model now reads

$$\min_{\substack{p_j, x_{i,j}, \\ j\in\{1,\dots,N\}, i\in I}} \sum_{i\in I}\sum_{k\in\mathcal{M}} (m_{i,k} - M_{i,k})^2 + \sum_{(i,i')\in I, i<i'} (c_{i,i'} - C_{i,i'})^2$$

$$\begin{aligned}
s.t. \sum_{j=1}^{N} p_j &= 1 \\
m_{i,1} &= \sum_{j=1}^{N} p_j x_{i,j}, i \in I \\
m_{i,k} &= \sum_{j=1}^{N} p_j (x_{i,j} - m_{i,1})^k, i \in I, k > 1 \\
c_{i,i'} &= \sum_{j=1}^{N} (x_{i,j} - m_{i,1})(x_{i',j} - m_{i',1}) p_j, i, i' \in I, i < i'
\end{aligned}$$

$$\begin{aligned}
0 \leq x_{i,j}, i \in I, j = 1, \dots, N, \\
\frac{3}{100} \leq p_j \leq 1, j = 1, \dots, N.
\end{aligned}$$

This means that we are generating stagewise independent balanced scenario trees, where the probabilities of each child node may vary, but are at least 0.03. This may lead to the fact that we are not able to account for scenarios with very small probability, which is a limitation, as financial data distributions are generally heavy tailed. However, for the purposes of this thesis, this assumption is not too restrictive.

4.5.2 Mean-CVaR model

The mean-CVaR model was implemented exactly as given in Equation 2.13 using the scenario tree generated from the moment matching method, where we used the risk aversion parameter $\lambda = 0.3$.

4.5.3 Reinforcement agent

We chose to use a tried and tested implementation of state of the art algorithms in the *Stable Baselines 3* library (Raffin et al. [28]). We experimented with multiple architectures and algorithms implemented therein, particularly A2C and PPO, while eventually settling on using PPO in the results given in Section 4.6. Here we share our experience with training the reinforcement agent.

We first experimented with the algorithms using a toy environment (Tree Building Environment with synthetic predictors and rewards) to obtain some semblance of how long it takes to obtain a reward better than random guessing. We experimented with several neural net architectures and found out that PPO usually outperformed A2C (converged much faster) with the same neural net architecture.

We also experimented with neural net architectures and found that even for very simple tasks (such as learning a different action based on the value of a single predictor p), a very nontrivial number of neurons in the hidden layers is required for the model to be able to solve the environment. Particularly, for a deterministic

toy example where p was randomly sampled from the set $\{0.1, 0.2\}$ and based on the given p the best number of stages to take was 3 if $p = 0.1$ and 5 if $p = 0.2$, the reinforcement agent didn't learn anything within hudreds of thousands of timesteps, unless we used an architecture with at least two hidden layers of 128 and 64 neurons respectively, where the second layer is separate for the actor and the critic (which estimate the policy and the action value). Furthermore, we used ReLu activations between each layer.

Due to the results obtained from the synthetic toy environment, we decided to use a very similar architecture as given above, where the only change is that we use 256 and 128 neurons in the hidden layers instead of 128 and 64, as the task we are trying to solve is much more difficult and stochastic. Unfortunately, due to the computational difficulties presented in Section 4.7, we couldn't experiment with multiple neural network architectures by doing hyperparameter optimization on the number of neurons in each layer. The final neural net architecture that we use is visualised in Figure 4.3.

With regard to the hyperparameters, we used the learning rate $\varphi = 0.001$, discount factor $\gamma = 1$ (which is used for the calculation of $\widehat{A}(a, s)$, see Schulman et al. [34, Section 5]) so that the agent is not penalised for using trees with more stages and performed 192 timesteps per update of the parameters of the neural network. Otherwise, we used the default hyperparameters, particularly $c_1 = 0.5$, where c_1 is the value function coefficient given in Equation 3.4).

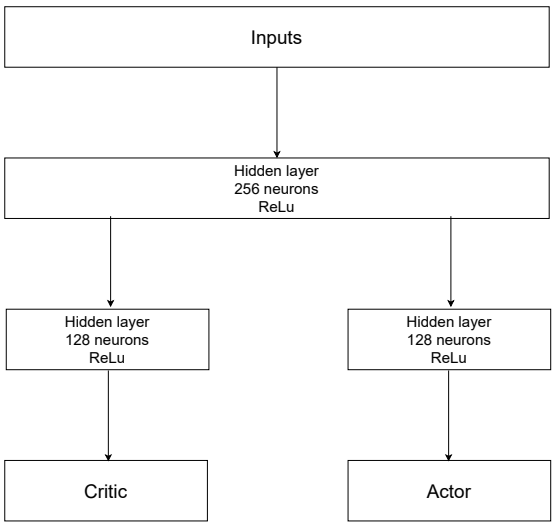


Figure 4.3: Illustration of the neural network architecture used for training the PPO agent.

4.6 Experimental results

In this section, we present the results from evaluating the reinforcement agent. We trained two agents, one without any penalty term and the other with inclusion of a penalty term to penalize the complexity of the scenario trees. Exploratory analysis of scenario trees generated for fixed asset sets is given in Section 4.6.1, while the results from training agents without penalty and with penalty are given in Sections 4.6.2 and 4.6.3 respectively.

4.6.1 Exploratory analysis

We chose 3 asset sets from set TrA and generated 250 scenario trees for each asset set by randomly choosing the number of stages and branching in each stage using Tree Building Environment with fixed $\alpha = 0.9$. The specification of which stocks belong to which set can be found in Table A.4. Boxplots of the obtained rewards can be seen in Figure 4.4. What is particularly interesting is that for a fixed set of assets, there are stark differences between the obtained rewards based on the chosen number of stages, while the variance between trees that have the same number of stages is smaller[3]. Furthermore, the best number of stages for one set of assets is not the best for all sets of assets. This is a particularly interesting result, as it shows that choosing the largest number of stages is not always the best approach with regard to obtaining the best value of the objective function.

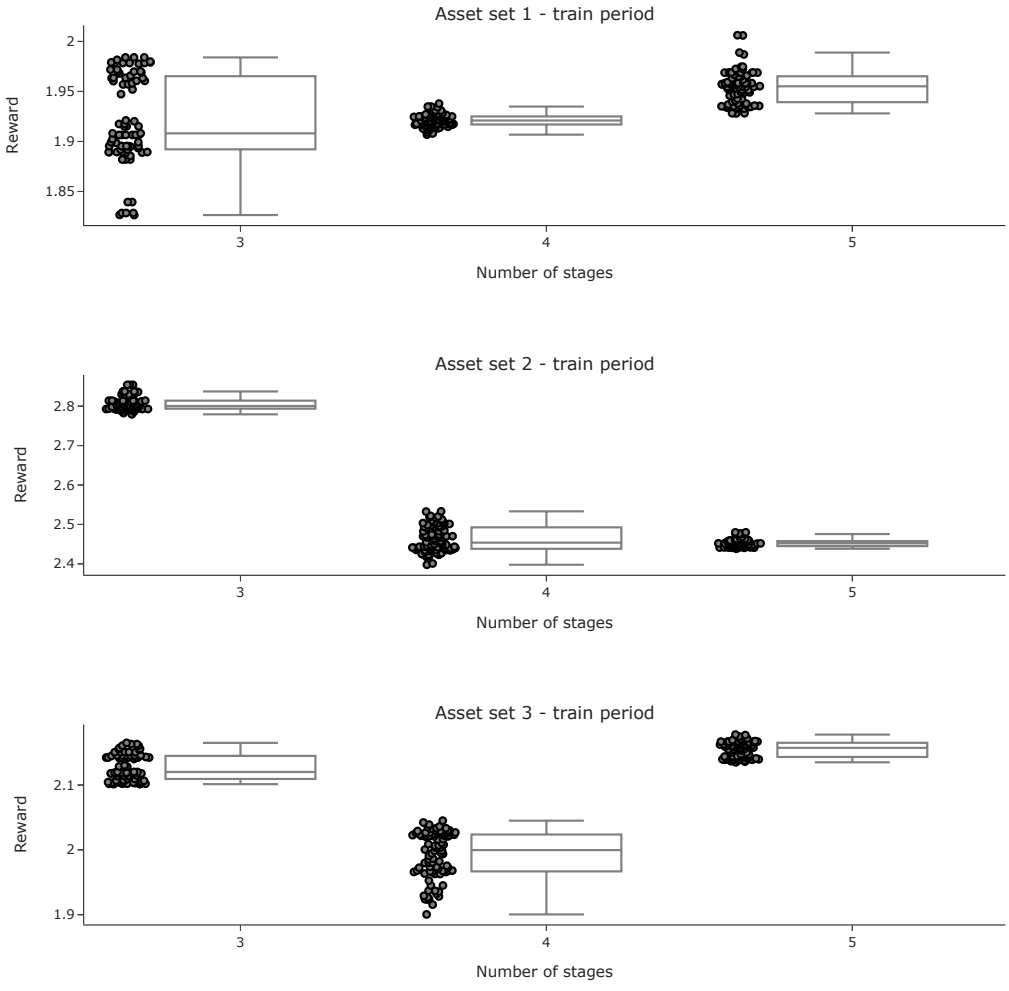


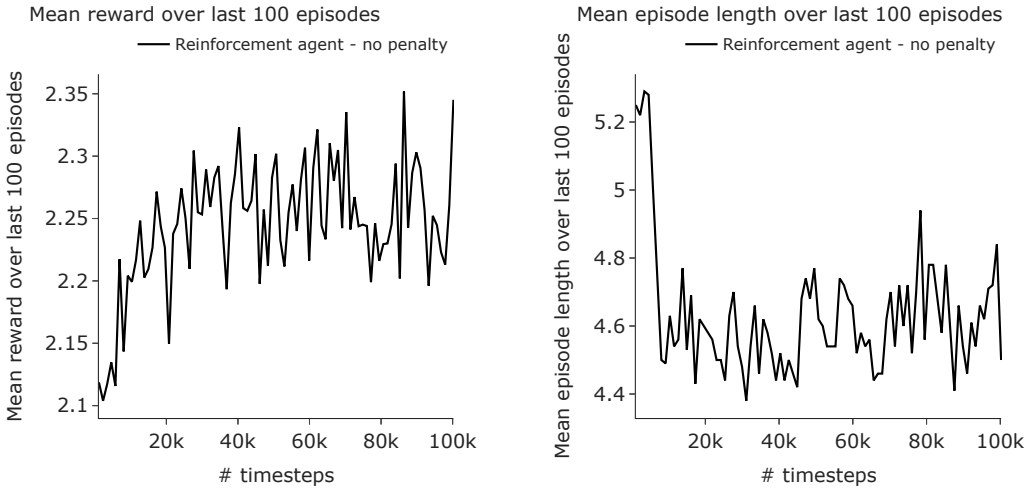
Figure 4.4: Boxplots of rewards obtained from 250 scenario trees, each for a fixed asset set obtained from set TrA. Notice the large differences based on number of stages, while the variance within stages is not that significant.

[3]we note that in the top graph of Figure 4.4, the variance for 3 stage trees is rather large

4.6.2 No penalty

The main aim in this and the following section is to compare the trained reinforcement agent to an agent that chooses the tree structure at random, evaluate the performance in contrast to the random agent and also investigate the structure of scenario trees chosen by the reinforcement agent. For this purpose, we ran the reinforcement agent and a random agent in the Tree Building Environment each for 500 episodes[4] for each of the sets in κ. We used the same random seed for initialising the environment for the reinforcement agent and the random agent, which means that both agents have been evaluated on the same samples from the given set in κ.

Figures 4.5a and 4.5b show the training curves (mean reward and mean episode length) over the last 100 episodes for the reinforcement agent with no penalty. We trained the agent for only 100k timesteps due to computational constraints. Notice that in Figure 4.5a, after about 30k timesteps the agent reached the maximum mean reward and didn't improve much from there. On the other hand, in Figure 4.5b, notice that at the start of training (within 20k timesteps) the mean length of the episode dropped quite significantly, which means that the agent started to use more 3 stage trees.



(a) Reward training curve. Mean reward over last 100 episodes.

(b) Mean training episode length over last 100 episodes. The episode length equals number of stages in the tree + 1, since the agent has to choose also the number of stages.

Figure 4.5: Training curves for reinforcement agent with no penalty. Timestep corresponds to a single state (acting on a single action) of the environment. An episode consists of multiple timesteps (particularly 4 for 3 stage trees, 5 for 4 stage trees and 6 for 5 stage trees).

[4]which means that the reinforcement agent and the random agent both generated exactly 500 trees

Dataset	Reinforcement agent mean reward	Random agent mean reward	Relative improvement of Reinforcement agent over Random agent
TrA	2.2214	2.1043	5.5647%
TrB	2.3350	1.8356	27.2063%
TeA	2.0346	2.0285	0.3007%
TeB	2.0573	2.0499	0.3609%

Table 4.1: Mean rewards of the reinforcement agent and random agent on every set in κ evaluated over 500 scenario trees. Last column is the relative mean performance of the reinforcement agent compared to the random agent, calculated as e.g. $2.2214/2.1043 \approx 1.055647 \rightarrow 5.5647\%$ in the first row (set TrA).

Table 4.1 shows the absolute performance (mean reward over 500 evaluation episodes) of the reinforcement agent and the random agent and also shows the relative performance (relative improvement in the mean reward) of the reinforcement agent compared to the random agent. Notice, that the reinforcement agent outperformed the random agent on datasets TrA and TrB. This shows that it is possible to train an agent to choose a scenario tree structure that is better than the random agent. Furthermore, notice that the outperformance is quite significant on sets TrA and TrB, while we refrain from making any claims about outperformance in the cases of TeA and TeB (at least we can say that the agent performs about the same as random guessing and doesn't underperform).

It is not surprising that the reinforcement agent has outperformed the random agent on set TrA, since this is the set that it was trained on and it has seen the distribution of the underlying data. On the other hand, the sizable outperformance on TrB is quite surprising, since it seems that the agent has learned a policy that fit very well to the distribution of assets in TrB. Figure 4.6 shows boxplots of the achieved rewards on set TrB.

The fact that the performance of the reinforcement agent is comparable to the random agent on sets TeA and TeB can be explained by the fact that the distribution of the underlying data has shifted. This is not surprising, as the sets TrA and TrB contain data from the years 2000–2009, while the sets TeA and TeB contain data for the period 2010–2019. As is well known, there were several crises affecting the markets (and financial stocks in particular) in the 2000–2009 period, while the 2010–2019 period contained an unprecedented bull run. We thus conclude that the agent we trained does not generalise into the future.

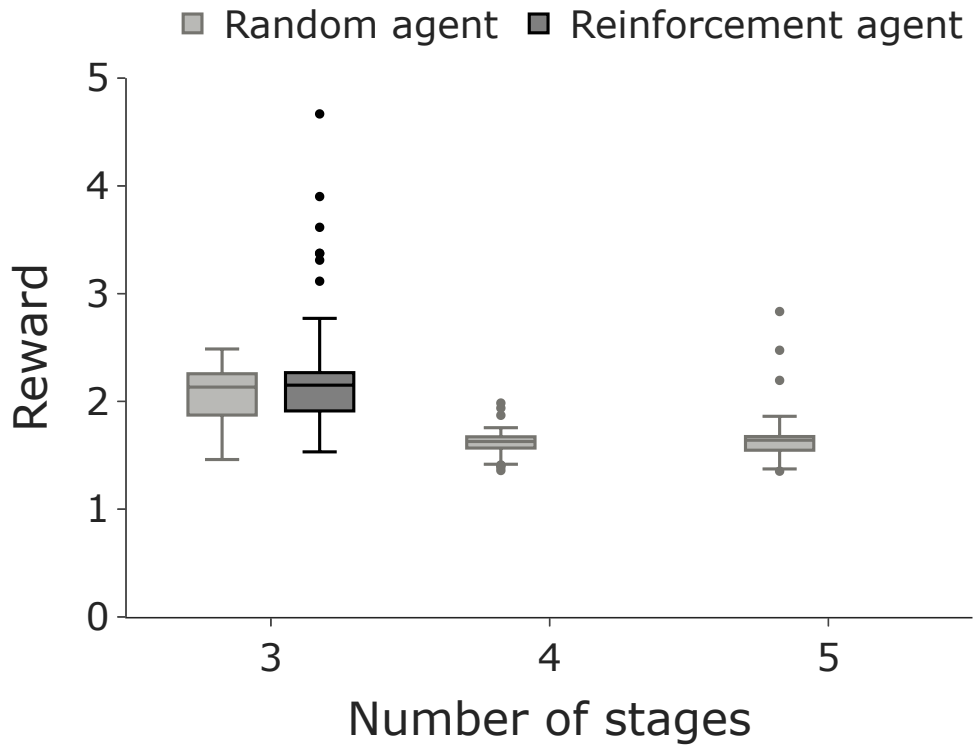


Figure 4.6: Boxplots of achieved rewards by number of stages in the scenario tree for reinforcement and random agent without penalty evaluated on TrB. Note the reinforcement agent chose only 3 stage trees. This figure implies that the reinforcement agent indeed chose the three stage trees correctly to maximise the reward.

Structure of predicted scenario trees

In this section, we analyse the structure of the scenario trees chosen by the reinforcement agent. We must be particularly careful here, as the construction of the environment as given in Section 4.3.1 constrains the built trees in such a way that forces a particular structure on the chosen branching in each successive stage. Particularly, since we require that the tree must have at least 100 leaves in the last stage, a 3 stage tree, where the first chosen branching was low (e.g. 3) will be forced to take a large branching in the next stages to actually end up with at least 100 scenarios at the end. The same holds for trees with 5 stages, where the choice of a large branching in the first and second stages will lead to forcing low branching in the last few stages to stay below 1200 scenarios.

We must thus interpret the results given here in comparison to the random agent, which shows the mean branching (mean number of descendants) given the constraints discussed above. Consider Figure 4.7. First thing to notice is that the reinforcement agent chooses trees with 3 stages much more frequently than trees with 5 stages and trees with 4 stages are chosen only very infrequently.

A peculiar thing is visible in the very bottom plot of Figure 4.7, which shows the mean branching of 5 stage trees chosen by the reinforcement agent and the random agent. It is immediately noticeable that the reinforcement agent has a particularly strong propensity to choose a large branching in the first stage

and then choose a lower/higher branching alternatively in the successive stages. We must stress that this interpretation is considered with regard to the mean branching chosen by the random agent, as the generally decreasing manner is caused by the constraints on the size of the trees as discussed above.

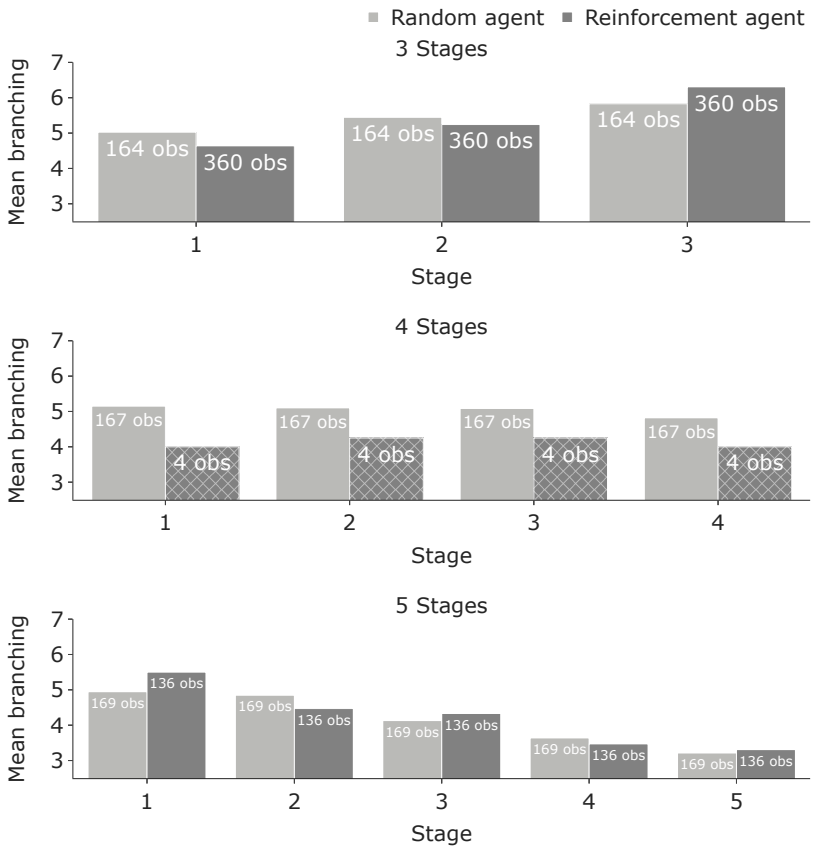


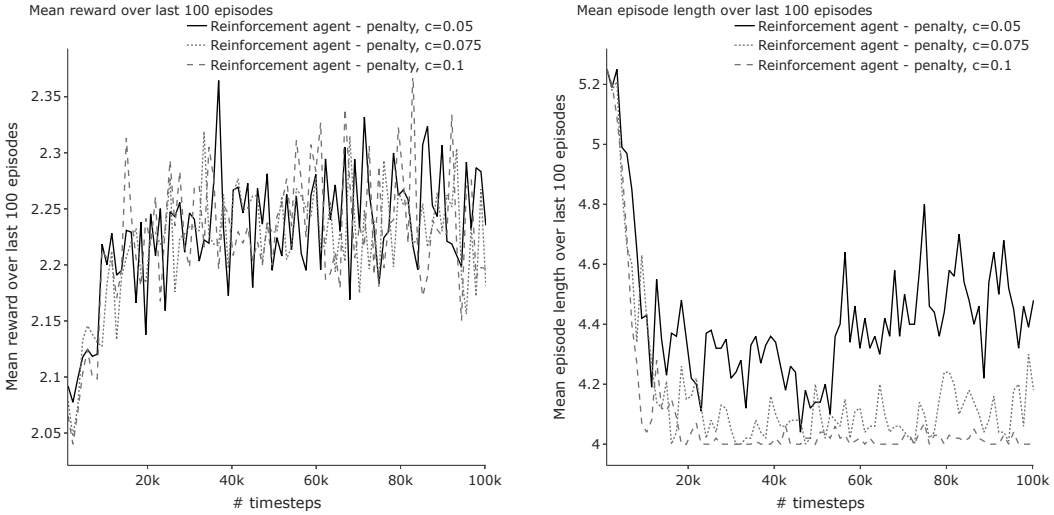
Figure 4.7: Mean branching (mean number of descendants) in each stage by number of stages on dataset TrA. Crosshatching indicates that the number of observations is less than 10.

It is interesting that for the 3 stage trees, the structure is radically different. The agent chooses a smaller branching in the first two stages and then a larger branching in the last stage. We refrain from interpreting the branching of 4 stage trees due to the low number of observations.

In practice, the trees are usually built with as many stages as possible and with as large branching as possible in the first stage, which then decreases in the successive stages. This however is in disagreement with our results. First of all, we have shown in Figure 4.4 that choosing the largest number of stages is not always the best option. Furthermore, while the agent used a similar structure (large initial branching) as the structure that is used in practice for the 5 stage scenario trees, for the 3 stage scenario trees, the structure was different. We thus conclude that there is room for improvement in the way that the structure of scenario trees is usually chosen in practice and using a model to choose the structure is preferable for the purpose of maximising the objective value.

4.6.3 Penalty

In this section, we show the results from training the reinforcement agent with a penalty based on the number of scenarios in the last stage of the tree (the number of leaves). We trained three agents with the penalty coefficients $c = 0.05$, $c = 0.075$ and $c = 0.1$ (see Section 4.4.1 for how the penalty is applied). The training curves are given in Figures 4.8a and 4.8b.



(a) Reward training curve. Mean reward over last 100 episodes.

(b) Mean training episode length over last 100 episodes. The length equals number of stages in the tree + 1, since the agent has to choose also the number of stages.

Figure 4.8: Training curves for reinforcement agents trained with penalty coefficients $c = 0.05$, $c = 0.075$ and $c = 0.1$. Timestep corresponds to a single state (acting on a single action) of the environment. An episode consists of multiple timesteps (particularly 4 for 3 stage trees, 5 for 4 stage trees and 6 for 5 stage trees).

It is immediately noticeable that the penalty coefficient has a significant effect on the mean training episode length (Figure 4.8b). A large penalty makes the agent consider 3 stage trees almost exclusively ($c = 0.1$), while with $c = 0.075$ the mean episode length is still low but allows also for larger trees compared to the $c = 0.1$ case. The mean episode length in the case $c = 0.05$ is then only a little bit lower compared to the no penalty case presented in the previous section. Note that in Figure 4.8a, the effect of the penalty on the achieved mean reward is not particularly noticeable, as the lines are almost indistinguishable[5]. We further discuss only the performance of agents trained with $c = 0.05$ and $c = 0.075$, as the case $c = 0.1$ is rather degenerate and was included mainly for the purpose of demonstrating that choosing a penalty coefficient that is too large leads to a rather degenerate behaviour.

Tables 4.2 and 4.3 report the achieved performance of the two agents trained with $c = 0.05$ and $c = 0.075$ respectively. Note that we report both the performance achieved when no penalty is considered in the environment (labeled "No

[5]if we disregard the variance

penalty") and the performance achieved when the penalty is taken into account (labeled "Penalty"). Note that the reinforcement agents have been trained with the penalty coefficient included, the "No penalty" case corresponds to the agents trained with a given penalty being evaluated in an environment where no penalty is considered so that the results are comparable with Table 4.1.

	No penalty			Penalty $c = 0.05$		
Set	RFA	RA	RFA/RA	RFA	RA	RFA/RA
TrA	2.2227	2.1043	5.6266%	2.2105	2.0823	6.1567%
TrB	2.3171	1.8356	26.2312%	2.3160	1.8135	27.7089%
TeA	2.0580	2.0285	1.4543%	2.0229	2.0064	0.8224%
TeB	2.0880	2.0499	1.8586%	2.0502	2.0453	0.2396%

Table 4.2: Results of evaluating reinforcement agent (RFA) trained with penalty coefficient $c = 0.05$ on 500 scenario trees on each set in κ against the random agent (RA). Two sets of results are given, No penalty – agent trained with penalty included evaluated with penalty not included in the reward. Penalty – agent trained with penalty included evaluated with penalty included in the reward. The relative improvement of the reinforcement agent over the random agent RFA/RA is reported similarly as in Table 4.1.

	No penalty			Penalty $c = 0.075$		
Set	RFA	RA	RFA/RA	RFA	RA	RFA/RA
TrA	2.2101	2.1043	5.0278%	2.2034	2.0713	6.3776%
TrB	2.1476	1.8356	16.9971%	2.1390	1.8093	18.2225%
TeA	2.0549	2.0285	1.3015%	2.0166	1.9954	1.0624%
TeB	2.0860	2.0499	1.7610%	2.0473	2.0198	1.3615%

Table 4.3: Results of evaluating reinforcement agent (RFA) trained with penalty coefficient $c = 0.075$ on 500 scenario trees on each set in κ against the random agent (RA). Two sets of results are given, No penalty – agent trained with penalty included evaluated with penalty not included in the reward. Penalty – agent trained with penalty included evaluated with penalty included in the reward. The relative improvement of the reinforcement agent over the random agent RFA/RA is reported similarly as in Table 4.1.

The results are quite similar as in Table 4.1. It is not surprising that the relative improvement is larger on sets TrA and TrB for the case of evaluating the agent with penalty compared to no penalty – the agent learns to use particular tree structures to incur lower penalty, while the random agent does not take the penalty into account. The fact that the same does not hold for sets TeA and TeB can again be explained by distribution shift of the underlying data and therefore choosing a particular tree structure might not lead to the same benefits as on sets TrA and TrB.

Structure of predicted scenario trees

In this section we analyse the structure of the scenario trees chosen by the reinforcement agents trained with penalty coefficients $c = 0.05$ and $c = 0.075$. We stress that the same care must be taken when interpreting the tree structure as in the case with no penalty as explained in the previous section.

Consider Figure 4.9, which shows the mean branching in each stage on TrA obtained by evaluation on 500 trees for reinforcement agent trained with $c = 0.05$. Note that in comparison to Figure 4.7, the agent trained with penalty chose a very similar number of 3 stage trees and 5 stage trees, while not using any 4 stage trees. What is more interesting is the structure of the 5 stage trees – inclusion of the penalty coefficient made the agent use exactly the scenario tree structure that is usually used in practice – a large branching (a large number of descendants) in the first and second stages and then a smaller branching (small number of descendants) in the following stages.

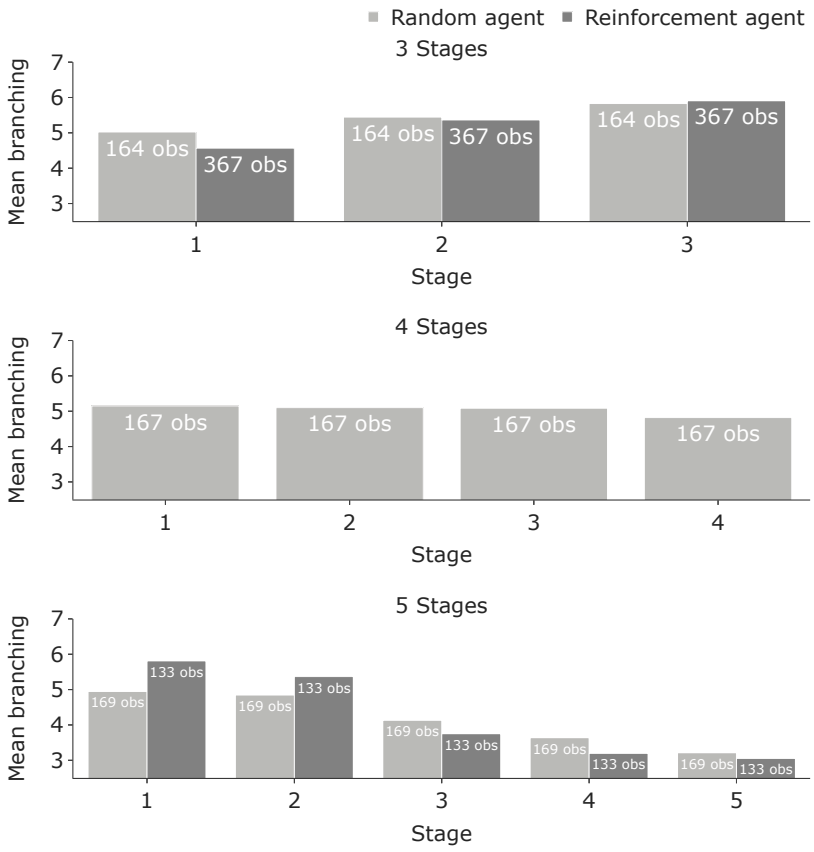


Figure 4.9: Mean branching (mean number of descendants) in each stage by number of stages on dataset TrA for reinforcement agent trained with $c = 0.05$.

Figure 4.10 shows the mean branching in each stage on TrA for the reinforcement agent trained with $c = 0.075$. Note that in comparison to Figure 4.9, the agent chose 5 stage trees much less frequently and also the structure of the 5 stage trees has shifted due to the larger penalty incurred for choosing larger trees. It is interesting to note that the agent still chooses a very large branching in the first stage for the 5 stage trees.

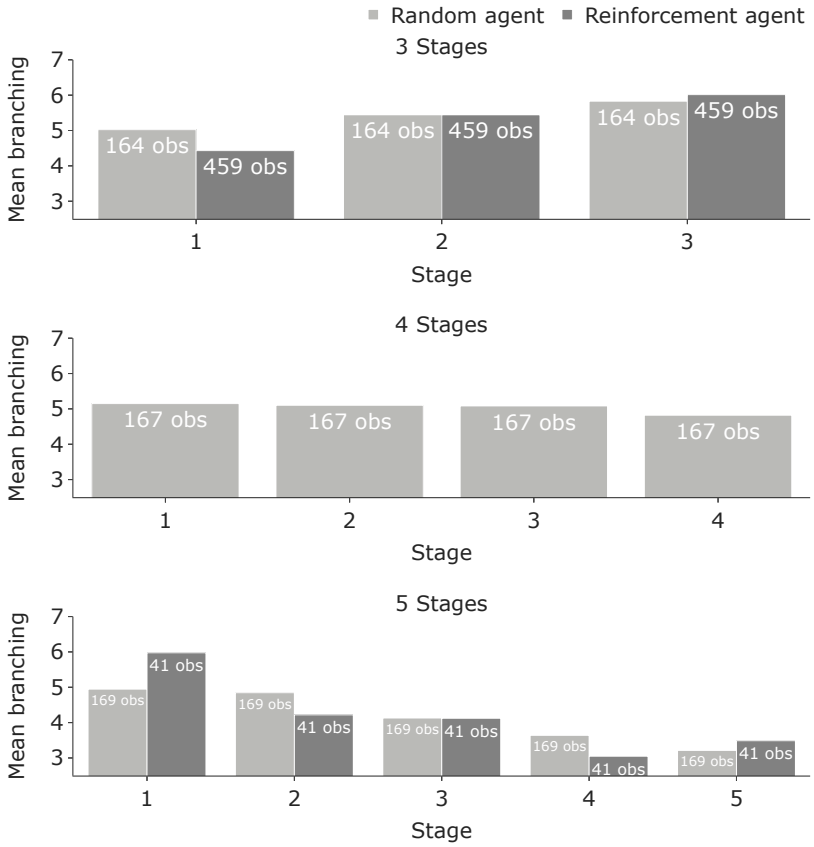


Figure 4.10: Mean branching (mean number of descendants) in each stage by number of stages on dataset TrA for reinforcement agent trained with $c = 0.075$.

4.7 Computational difficulties

The writing of this thesis has been plagued by computational difficulties from the start and we wish to share our experience with implementing all parts of this thesis here.

First of all, the moment matching method, which seems quite easy to implement, required a significant amount of experimentation and trying different optimization frameworks and solvers to actually obtain a working implementation. We have tried several python packages with open source solvers (most notably *SciPy* (Virtanen et al. [40]) , *Mystic* (McKerns et al. [22]) and *Gekko* (Beal et al. [2]) and the IPOPT solver) and none of them produced a suitable result despite correct implementation due to low strength of the open source solvers. We finally settled on using GAMS with the CONOPT solver which worked out quite nicely. This however required us to connect our Python code to GAMS using the GAMS Python API, which meant that we could not use a compute cluster due to licensing limitations.

Since we already had the dependence on GAMS, we also implemented the mean-CVaR model in GAMS using the CPLEX solver. While the implementation itself in GAMS was not terribly difficult, bending all data in the correct way and formulating the nonanticipativity constraints correctly took significant effort.

Lastly, the reinforcement learning part. This part was plagued by slow training and therefore a significant amount of time was spent on training the models, since the dependence on GAMS didn't allow training the reinforcement agent on a compute cluster with hundreds of cores, but we were rather constrained to a personal computer with 6 cores. This was a significant limitation, since training reinforcement agents is usually very computationally intensive (e.g. the state of the art models mentioned in the beginning of Chapter 3 were usually trained for months on hundreds of machines). We must note that training each agent took over a day of runtime on a personal computer.

Conclusion

In this thesis, we explored the dependence of multistage scenario models on the chosen scenario tree structure.

We explored the dependence of the objective value on the scenario tree structure and we have shown that the structure of scenario trees that is usually used in practice may not always be the best. We found out that there are significant differences in the obtained rewards when using a different number of stages, while the differences between different scenario trees with a fixed number of stages are not that staggering.

Further we proposed an experiment to explore the dependence of the value of the objective function of the mean-CVaR model on the structure of a scenario tree that was built using the moment matching method from historical data using reinforcement learning. Further, we trained several reinforcement agents and evaluated their performance, finding that it is possible to train such an agent to aid in the task of choosing a scenario tree structure and we have shown that for 3 and 5 stage trees, the structure of the scenario trees chosen by the agent is rather different compared to the structure usually used in practice. We further explored the effect of including a penalty for choosing a scenario tree structure that is too complex.

This thesis could be extended in several ways. Due to computational limitations, we had to constrain ourselves only to a small set of trees. It would be interesting to extend the experiment such that more stages and more branchings are allowed. Moreover, other scenario tree building methods than moment matching could be explored. It would be also very interesting to train the agent for a lot longer than we did, as we cannot be sure that a significantly longer training would not lead to better results. All of these extensions would however come at a significant computational cost with the current implementation.

Bibliography

[1] P Artzner, Fy Delbaen, E Jean-Marc, and D Heath. Coherent measures of risk. *Mathematical Finance*, 9:203 – 228, 07 1999. doi: 10.1111/1467-9965.00068.

[2] L Beal, D Hill, R Martin, and J Hedengren. Gekko optimization suite. *Processes*, 6(8):106, 2018. doi: 10.3390/pr6080106.

[3] C Berner, G Brockman, B Chan, V Cheung, P Debiak, C Dennison, D Farhi, Q Fischer, S Hashme, C Hesse, R Józefowicz, S Gray, C Olsson, J W Pachocki, M Petrov, H Pondé de Oliveira Pinto, J Raiman, T Salimans, J Schlatter, J Schneider, S Sidor, I Sutskever, J Tang, F Wolski, and S Zhang. Dota 2 with large scale deep reinforcement learning. *ArXiv*, abs/1912.06680, 2019.

[4] G Brockman, V Cheung, L Pettersson, J Schneider, J Schulman, J Tang, and W Zaremba. Openai gym, 2016. URL `https://github.com/openai/gym`.

[5] B.A. Calfa, A. Agarwal, I.E. Grossmann, and J.M. Wassick. Data-driven multi-stage scenario tree generation via statistical property and distribution matching. *Computers and Chemical Engineering*, 68:7–23, 2014. ISSN 0098-1354. doi: https://doi.org/10.1016/j.compchemeng.2014.04.012. URL `https://www.sciencedirect.com/science/article/pii/S009813541400129X`.

[6] G Cornuejols and R Tütüncü. *Optimization Methods in Finance*. Cambridge University Press, 2006. doi: 10.1017/CBO9780511753886.

[7] B Defourny, D Ernst, and L Wehenkel. Multistage stochastic programming: A scenario tree based approach to planning under uncertainty. *LE, Sucar, EF, Morales, and J., Hoey (Eds.), Decision Theory Models for Applications in Artificial Intelligence: Concepts and Solutions. Hershey, Pennsylvania, USA: Information Science Publishing*, 01 2011. doi: 10.4018/978-1-60960-165-2.ch006.

[8] J Dupačová, G Consigli, and S Wallace. Scenarios for multistage stochastic programs. *Annals of Operations Research*, 100:25–53, 12 2000. doi: 10.1023/A%3A1019206915174.

[9] J Dupačová, J Hurt, and J Štěpán. *Stochastic Modeling in Economics and Finance*. 01 2003. ISBN 1-4020-0840-6. doi: 10.1007/b101992.

[10] A Fawzi, M Balog, A Huang, T Hubert, B Romera-Paredes, M Barekatain, A Novikov, F Ruiz, J Schrittwieser, G Swirszcz, D Silver, D Hassabis, and P Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610:47–53, 10 2022.

[11] GAMS Development Corporation. General Algebraic Model System (GAMS), 2017. https://www.gams.com.

[12] I Goodfellow, Y Bengio, and A Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[13] H Heitsch and W Roemisch. Generation of multivariate scenario trees to model stochasticity in power management. pages 1 – 7, 07 2005. doi: 10.1109/PTC.2005.4524696.

[14] S Huang, A Kanervisto, A Raffin, W Wang, S Ontañón, and R F J Dossa. A2c is a special case of ppo, 2022. URL `https://arxiv.org/abs/2205.09123`.

[15] K Høyland and S Wallace. Generating scenario trees for multistage decision problems. *Management Science*, 47:295–307, 02 2001. doi: 10.1287/mnsc.47.2.295.9834.

[16] K Høyland, M Kaut, and S Wallace. A heuristic for moment-matching scenario generation. *Computational Optimization and Applications*, 24:169–185, 02 2003. doi: 10.1023/A:1021853807313.

[17] J Jumper, R Evans, A Pritzel, T Green, M Figurnov, O Ronneberger, K Tunyasuvunakool, R Bates, A Žídek, A Potapenko, A Bridgland, C Meyer, S Kohl, A Ballard, A Cowie, B Romera-Paredes, S Nikolov, R Jain, J Adler, and D Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 596:1–11, 08 2021. doi: 10.1038/s41586-021-03819-2.

[18] V Kozmík. *Multi-Stage Stochastic Programming with CVaR: Modeling, Algorithms and Robustness.* PhD thesis, Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra pravděpodobnosti a matematické statistiky, 2015. URL `https://dspace.cuni.cz/handle/20.500.11956/67018`.

[19] M Leippold. Value-at-risk and other risk measures. *SSRN Electronic Journal*, 01 2015. doi: 10.2139/ssrn.2579256.

[20] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[21] H Markowitz. Portfolio selection*. *The Journal of Finance*, 7(1):77–91, 1952. doi: https://doi.org/10.1111/j.1540-6261.1952.tb01525.x. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1540-6261.1952.tb01525.x`.

[22] M M McKerns, L Strand, T Sullivan, A Fang, and M A G Aivazis. Building a framework for predictive science, 2012. URL https://arxiv.org/abs/1202.1056.

[23] A.J. McNeil, R. Frey, and P. Embrechts. *Quantitative Risk Management: Concepts, Techniques and Tools - Revised Edition*. Princeton Series in Finance. Princeton University Press, 2015. ISBN 9780691166278. URL https://books.google.cz/books?id=REQLogEACAAJ.

[24] V Mnih, K Kavukcuoglu, D Silver, A Graves, I Antonoglou, D Wierstra, and M Riedmiller. Playing atari with deep reinforcement learning. 12 2013.

[25] V Mnih, A Badia, M Mirza, A Graves, T Lillicrap, T Harley, D Silver, and K Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 02 2016.

[26] OpenAI. https://openai.com/blog/openai-baselines-ppo/.

[27] G Pflug. Scenario tree generation for multiperiod financial optimization by optimal discretization. *Mathematical Programming*, 89:251–271, 01 2001. doi: 10.1007/PL00011398.

[28] A Raffin, A Hill, A Gleave, A Kanervisto, M Ernestus, and N Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html. https://stable-baselines3.readthedocs.io/.

[29] R T Rockafellar and S Uryasev. Optimization of conditional value-at risk. *Journal of Risk*, 3:21–41, 2000.

[30] R T Rockafellar and S Uryasev. Conditional value-at-risk for general loss distributions. *Journal of Banking and Finance*, 26(7):1443–1471, 2002. ISSN 0378-4266. doi: https://doi.org/10.1016/S0378-4266(02)00271-6. URL https://www.sciencedirect.com/science/article/pii/S0378426602002716.

[31] A Ruszczynski and A Shapiro. *Stochastic programming (handbooks in operations research and management science)*. Elsevier, 2003.

[32] M Salahi, F Mehrdoust, and F Piri. Cvar robust mean-cvar portfolio optimization. *ISRN Applied Mathematics*, 2013, 01 2013. doi: 10.1155/2013/570950.

[33] J Schulman, S Levine, P Moritz, M I Jordan, and P Abbeel. Trust region policy optimization, 2015. URL https://arxiv.org/abs/1502.05477.

[34] J Schulman, F Wolski, P Dhariwal, A Radford, and O Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

[35] A Shapiro, D Dentcheva, and A Ruszczyński. *Lectures on Stochastic Programming*. Society for Industrial and Applied Mathematics, 2009. doi: 10.1137/1.9780898718751. URL https://epubs.siam.org/doi/abs/10.1137/1.9780898718751.

[36] J Shlens. Notes on kullback-leibler divergence and likelihood. *CoRR*, abs/1404.2000, 2014. URL http://arxiv.org/abs/1404.2000.

[37] D Silver, A Huang, C J Maddison, A Guez, L Sifre, G van den Driessche, J Schrittwieser, I Antonoglou, V Panneershelvam, M Lanctot, S Dieleman, D Grewe, J Nham, N Kalchbrenner, I Sutskever, T Lillicrap, M Leach, K Kavukcuoglu, T Graepel, and D Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[38] R S Sutton and A G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[39] G Van Rossum and F L Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

[40] P Virtanen, R Gommers, T E Oliphant, M Haberland, T Reddy, D Cournapeau, E Burovski, P Peterson, W Weckesser, J Bright, S J van der Walt, M Brett, J Wilson, K J Millman, N Mayorov, A R J Nelson, E Jones, R Kern, E Larson, C J Carey, İ Polat, Y Feng, E W Moore, J VanderPlas, D Laxalde, J Perktold, R Cimrman, I Henriksen, E A Quintero, C R Harris, A M Archibald, A H Ribeiro, F Pedregosa, P van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

[41] S Vitali, M Horejšová, M Kopa, and V Moriggia. Evaluation of scenario reduction algorithms with nested distance. *Computational Management Science*, 06 2020. doi: 10.1007/s10287-020-00375-4.

[42] K Šutiene, D Makackas, and H Pranevičius. Multistage k-means clustering for scenario tree construction. *Informatica*, 21(1):123–138, jan 2010. ISSN 0868-4952.

[43] Y Wu, E Mansimov, S Liao, R B Grosse, and J Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *CoRR*, abs/1708.05144, 2017. URL http://arxiv.org/abs/1708.05144.

[44] Yahoo Finance. https://finance.yahoo.com.

[45] yfinance. https://github.com/ranaroussi/yfinance, 2022.

[46] M Zelinka. Using reinforcement learning to learn how to play text-based games. Master's thesis, Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra teoretické informatiky a matematické logiky, 2017. URL https://dspace.cuni.cz/handle/20.500.11956/90588.

Appendix

A.1 Definitions of asset sets

ACGL	AFL	AIG	AJG	ALL	AON	AXP
BAC	BEN	BK	BLK	BRO	C	CB
CINF	CMA	COF	FDS	FITB	GL	GS
HBAN	HIG	IVZ	JPM	KEY	L	LNC
MCO	MMC	MS	MTB	NTRS	PGR	PNC
RE	RF	RJF	SCHW	SIVB	SPGI	STT
TFC	TROW	TRV	USB	WFC	WRB	ZION

Table A.1: Stock tickers used in Chapter 4.

ALL	BK	ACGL	FDS	ZION	TROW
STT	RJF	SPGI	AON	LNC	USB
AXP	AIG	AJG	BLK	RE	SIVB
NTRS	CB	WRB	BRO	L	IVZ
BAC	GS	WFC	MTB	MMC	BEN

Table A.2: Stock tickers in set A.

HBAN	COF	JPM	AFL
PGR	CMA	RF	TFC
CINF	MCO	MS	C
HIG	FITB	TRV	GL
SCHW	PNC	KEY	

Table A.3: Stock tickers in set B.

Asset set 1	ALL	BK	ACGL	FDS	ZION	TROW	STT	RJF	SPGI	AON
Asset set 2	LNC	USB	AXP	AIG	AJG	BLK	RE	SIVB	NTRS	CB
Asset set 3	WRB	BRO	L	IVZ	BAC	GS	WFC	MTB	MMC	BEN

Table A.4: Specification of which stock tickers belong to which asset set.

A.2 Electronic attachment

The Python source code used to obtain the results in Chapter 4 is included in the electronic attachment.