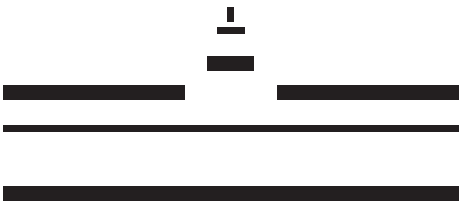

Westfälische
Wilhelms-Universität
Münster

Efficient Probabilistic Inference with Orthogonal Tensor Train Decomposition

Bachelor Thesis

submitted by:
Frederik Schittny

Matriculation number: 515495
Study programme: BSc. Computer Science

First reviewer:
JProf. Dr. Tanya Braun

Second reviewer:
Sagad Hamid

Münster, 8th September 2023

Abstract

Hidden Markov models are used to represent a system with random variables and model the system's behaviour through the transitions between the system's possible states. The probabilities determining the system's behaviour are divided into transition and emission probabilities. In recent years, the use of tensors, often called multidimensional arrays, has increased in the field of machine learning due to the advantages of high expressive power and efficient algorithms. Using a single tensor to represent a hidden Markov model's emission probabilities, however, leads to a memory demand that is exponential in the model's size. Finding a way to represent a hidden Markov model using tensors without an exponential memory demand can combine the advantages of hidden Markov models with the advantages of tensor representations. This thesis investigates a new representation of hidden Markov models, which uses an internal, orthogonal tensor decomposition on the emission probabilities. The representation lowers the memory demand of the emission probabilities in comparison to storing them in a high-rank tensor, at the cost of introducing additional computational effort when working with the emission probabilities. The goal of the thesis is to find efficient inference algorithms on this representation, by using the representation's structure and properties. Two different parallelization approaches for the forward algorithm are covered, the first of which parallelises the incorporation of evidence, while the second uses parallelised tensor train contraction on the decomposed emission tensor train. The algorithms' efficiency is verified by recording runtime data on an implementation for models with a variety of parameters. The recorded runtime data identifies the parallelised evidence incorporation as the better of the two approaches, due to it having lower runtimes, better scalability, and applying to a wider variety of models. It is shown, that both approaches can only compensate for the additional runtime introduced by the decomposition of the emission probabilities. The considerations made on learning internally decomposed hidden Markov models present the problem, that learning algorithms like the Baum-Welch algorithm cannot be transferred to the new representation by simply converting their operations to tensor train operations. Both the inference and the learning approaches that are discussed are unable to use the representation's structure and special properties to achieve higher efficiency than existing representations. This inability is mainly attributed to the lack of logical and structural meaning of the internal decomposition. Overall, the new representation through internal decomposition is found to be inferior to existing representations, due to the lack of a logical meaning in the decomposition's structure.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Probabilistic graphical models (PGMs) and other machine learning models are often represented as graphs with their data being stored in matrices, arrays, and other well-known data structures. In recent years, the usage of tensors, often called multidimensional arrays, in machine learning applications has been increasing. This trend started after tensors were carried over from other scientific fields like physics, where tensors and tensor networks are used to represent quantum systems and their states as well as quantum algorithms and their efficient and concise implementation (Biamonte and Bergholm, 2017).

In machine learning, tensors and tensor networks are used for everything from signal processing and data modification (Rabanser et al., 2017) over the representation of complete machine learning models with tensor networks (Robeva and Seigal, 2018) to learning models through tensor decomposition (Anandkumar et al., 2014; Khouja et al., 2022). All of these advances are based on and profit from the algorithms, libraries, and optimisations that have been developed over the years for the usage of tensors in multiple disciplines. These include specialised hardware (Jouppi et al., 2017) as well as software libraries (Fishman et al., 2022a; Roberts et al., 2019). This widespread support for tensors makes it possible to use tensors interdisciplinarily for storing data efficiently using tensor decomposition, a technique that mainly reduces the memory complexity of tensors, and processing the data stored in tensors with special operations like tensor contraction. The widespread usage of tensors in different disciplines and sciences did not just drive the development of tensor-focused and universally applicable techniques and methods but also uncovered similarities and parallels between previously unrelated subjects, like the duality of some graphical machine learning models and quantum systems (Glasser et al., 2019).

Due to the infrastructure and optimisations built around tensors, the usage of tensors for representing PGMs allows for the application of existing tensor accelerations and optimisations to PGMs. This usage of tensor-specific optimisations applies to the memory efficiency of the representation as well as to the algorithms used with PGMs, like learning and inference algorithms, which can be executed directly on the tensor representation. In addition to more efficient representations and algorithms, the usage of tensors could potentially enable the development of new techniques only realizable with tensors. One approach is to represent a full PGM with a tensor network, which is addressed in some of the related work mentioned above. Another approach is to represent individual parts of a model with separate tensors. In contrast to the approach of representing a full PGM with a tensor network, the approach of representing each part individually provides the opportunity to adjust each representation to the specific properties and functions of one

part of the PGM. The approach of using individual tensors for each part of a PGM, however, comes with the problem that a tensor's memory demand increases exponentially with the size of the PGM's part it represents, which can make it impossible to store this kind of representation for large and complex PGMs.

This thesis deals with investigating an alternative representation of PGMs through tensor networks, which uses an internal tensor decomposition for individual parts of a PGM. An internal decomposition of individual PGM components into tensor networks preserves the flexibility of an individual representation while being able to use existing tensor-specific optimisations and also solving the problem of exponential memory demand. One disadvantage of the internal decomposition is the introduction of additional runtime when accessing values from the PGM that are stored in a tensor train. The focus of the thesis is on finding efficient inference algorithms on internally decomposed PGMs, which can make use of the structure and properties of the decomposition. More specifically, the thesis presents a practical case study on internally decomposed hidden Markov models (HMMs), a type of PGM. The HMM's emission probabilities, which are one part of the HMM, are represented by an orthogonally decomposed tensor train, for which multiple approaches to efficient inference algorithms are implemented and verified by recording runtime data. Besides the examination of inference algorithms, the thesis also investigates the possibilities of learning such internally decomposed models.

The runtime data recorded in the case study identifies the parallelisation of evidence incorporation as the most efficient and universally applicable approach to efficient inference techniques on internally decomposed models. Because inference algorithms frequently access the values stored in the decomposed tensor trains, they have the disadvantage of being highly impacted by the additional runtime for accessing values, while also deconstructing the tensor trains special structure on each access of a value. It is shown that due to this behaviour, optimised inference algorithms can only compensate for the additional runtime introduced by the internal decomposition. The easiest way of transferring any algorithm for HMMs to use the new representation efficiently is to transfer each operation of the algorithm's implementation into a corresponding tensor train operation. However, the considerations made on learning algorithms for internally decomposed models demonstrate that learning algorithms cannot be adapted to the new representation by transferring each operation. The problems with using the structure and property of the decomposition in inference and learning algorithms are mainly attributed to the lack of logical and structural meaning of the new representation.

The thesis starts by giving an introduction to the most essential aspects of tensors, tensor networks, and tensor decompositions on the one hand and PGMs, more specifically HMMs, as well as inference and learning algorithms on the other hand. How these two different foundations are combined to form a new representation of HMMs through the internal decomposition of emission probabilities will be explained in the chapter on algorithm approaches, which also describes the different parallelisation approaches for inference algorithms and the theoretical considerations on adapting learning algorithms

to the new representation. The most important design decisions for the implementation used to verify the approaches for efficient inference will be explained, followed by the presentation of the runtime data acquired with said implementation. The last chapters are used to interpret the numerical results and analyse the usability of the new representation overall, based on both the recorded runtime data and the theoretical findings regarding inference and learning algorithms.

2 Preliminaries

The thesis is based on the two separate subject areas of tensor networks and dynamic Bayesian networks. How these two different fields are combined will be explained in Section 3.1. The third cornerstone are the different inference algorithms used with hidden Markov models.

2.1 Tensor Networks

Tensor networks have originally been used in physics, to describe quantum systems and their behaviour (Biamonte and Bergholm, 2017). After parallels between tensor networks and models from machine learning and artificial intelligence (AI) contexts were discovered, as described in Section 2.2.2, usage of tensors and tensor networks increased in those domains, which previously only used tensors for data storage. The following sections give an overview of tensors in general, their combinations into basic tensor networks, and the algorithms for constructing tensor networks through decomposition.

2.1.1 Tensors

A tensor is a mathematical, multilinear object associated with a vector space of fixed dimension (Hillar and Lim, 2013, p.2-3). In computer science, tensors are often interpreted as multidimensional arrays, used to store information (Glasser et al., 2020, p.2). The two main features characterising a tensor are its rank and dimensions. To avoid the confusion often arising from the different definitions used for the terms rank, dimension, and order, these two features will be described with a uniform definition and any quoted sources will be adapted for the terminology to stay consistent.

The rank of a tensor denotes the minimal set of vectors $\{\mathbf{u}_1, ..., \mathbf{u}_r\}$, typically from $\mathbb{R}^{n_i}, n_i \in \mathbb{N}$, which can be combined with the tensor product to form the tensor $\mathcal{T}$ (Halaseh et al., 2022, p.2-3):

$$\mathcal{T} = \mathbf{u}_1 \otimes ... \otimes \mathbf{u}_r \in \mathbb{R}^{n_1 \times ... \times n_r}. \tag{2.1}$$

The tensor product $\otimes$ is a bilinear map operator for tensors $\mathcal{A} \in \mathbb{R}^{n_1 \times ... \times n_a}$ and $\mathcal{B} \in \mathbb{R}^{n_{a+1} \times ... \times n_{a+b}}$, which maps the two vector spaces associated with $\mathcal{A}$ and $\mathcal{B}$ to the vector space associated with the tensor $\mathcal{C} = \mathcal{A} \otimes \mathcal{B} \in \mathbb{R}^{n_1 \times ... \times n_a \times n_{a+1} \times ... \times n_{a+b}}$. The rank also describes the number of indices a tensor has. Indices are used to denote every element in the tensor. Any combination $i_1, ..., i_r \in \mathscr{D}$ of given values for the indices denotes an

element $\mathcal{T}_{i_1,...,i_r}$ within the tensor (Halaseh et al., 2022, p.2):

$$\mathcal{T}_{i_1,...,i_r} = (\mathbf{u}_1)_{i_1} \cdot ... \cdot (\mathbf{u}_r)_{i_r} \in \mathbb{R}, \tag{2.2}$$

with $(\mathbf{u}_1)_{i_1}$ denoting the i_1-th element within the vector $\mathbf{u}_1$. The dimension $d = |\mathscr{D}|$ describes the range of values $\mathscr{D} = \{v_1^i, ..., v_d^i\} \subset \mathbb{R}$ each index can take on. By this definition, each index can have a separate dimension. The lowest four tensor ranks all have individual names and geometric equivalents. A rank-0 tensor is typically referred to as a scalar or a single value. A rank-1 tensor is a vector, the length of which is the dimension of the tensor's single rank. A rank-2 tensor is a matrix, with the two dimensions being the number of columns and rows and a rank-3 tensor is the geometric equivalent of a cuboid (Halaseh et al., 2022, p.2). Tensors of rank 4 and higher lack classic geometric equivalents. To represent tensors and tensor operations of any rank consistently, the tensor diagram notation can be used (Bradley, 2019). In this notation, a tensor is depicted as a shape. Its indices are shown as lines, which can be labelled with the names of the indices and their dimension for clarity. A depiction of tensors of different ranks in the tensor diagram notation can be seen in Figure 2.1. To distinguish between tensors of different ranks in the following sections, scalars will always be written as normal lowercase variables like n, vectors will be written as bold lowercase variables like $\mathbf{u}$, matrices will be written with bold uppercase letters like $\mathbf{U}$, and tensors of higher or unknown rank will be written as calligraphic letters like $\mathcal{T}$. Furthermore, sets will be written with script-style uppercase letters like $\mathscr{R}$.

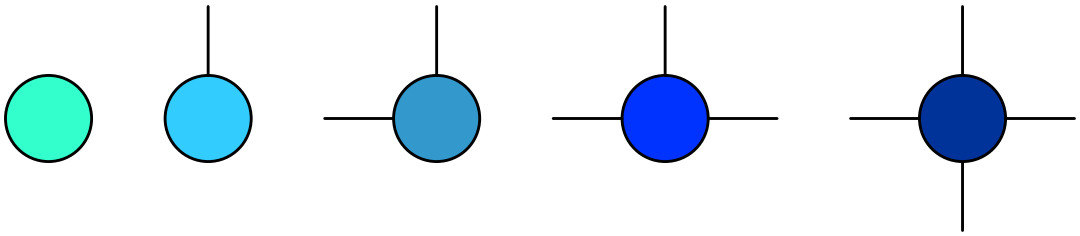

Figure 2.1: Tensors of ranks zero to four represented in the tensor diagram notation. Index names or dimensions can be added at the end of the lines for clarity depending on the context.

Tensors are used to represent physical quantities in physics and engineering (Kolecki, 2002) and have use cases in the description of quantum physics. In computer science, tensors are mostly used for data storage as an alternative to arrays and matrices, especially when data has a natural mapping to indices, with a high amount of required fields. One example is the storage of data analysed by neural networks and comparable models in machine learning applications (Patidar, 2019). This common usage is partially due to the low access times of the tensor elements. The memory complexity of a tensor is $\mathcal{O}(r^d)$ (Stoudenmire, 2018), meaning it is exponential in its rank r, with the maximum dimension of the tensor being d. These properties make tensors an attractive option for storing data with low access times but at the cost of high memory demand.

There are different operations defined on tensors, with the most trivial ones being multiplication and addition of every field with a constant. Operations between two or more tensors include element-wise operations like element-wise addition or multiplication (Porat, 2014, p.15-16). These operations require the tensors to be of equal ranks and dimensions. One less trivial but essential operation is the so-called tensor contraction. A contraction combines two tensors $\mathcal{S}$ of rank o and $\mathcal{T}$ of rank p, which have a shared index $l_i = m_j = k$ of dimension d_k, into a tensor $\mathcal{R}$ of rank $q = o + p - 2$. The contraction is carried out by first multiplying elements with matching index values for k, then summing out the index k by adding up all entries combined over the values of k (Halaseh et al., 2022, p.3):

$$\mathcal{R}_{l_1,\ldots,l_p,m_1,\ldots,m_o} = \sum_{k=1}^{d_k} \mathcal{T}_{l_1,\ldots,l_{i-1},k,l_{i+1},\ldots,l_p} \cdot \mathcal{S}_{m_1,\ldots,m_{j-1},k,m_{j+1},\ldots,m_o}. \tag{2.3}$$

In the tensor diagram notation, a contraction of two tensors is shown by connecting the shapes of the two tensors with a shared index (Bradley, 2019), as shown in Figure 2.2.

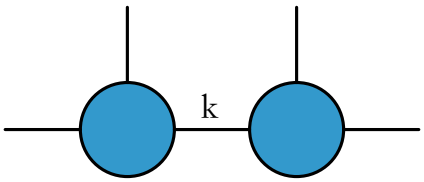

Figure 2.2: Tensor diagram notation of two rank-3 tensors which are contracted over the shared index k.

A simple example of the tensor contraction is the contraction of two rank-2 tensors over one shared index. Given the rank-2 tensors $\mathbf{A}_{jk}$ and $\mathbf{B}_{kl}$, the contraction $\mathbf{C}_{jl}$ over the shared index k with dimension d_k is described by

$$\mathbf{C}_{jl} = \sum_{k=1}^{d_k} \mathbf{A}_{jk} \cdot \mathbf{B}_{kl}, \tag{2.4}$$

according to Equation (2.3). This formula is identical to the standard matrix multiplication, which highlights the fact that tensor operations, like the tensor contraction, can be interpreted as higher-rank matrix operations, which is often done in computer science contexts.

2.1.2 Tensor Trains

Tensor networks are constructed by contracting multiple tensors over shared indices. The most basic form is the tensor train, also called matrix product state. A tensor train consists of rank-3 tensors which are contracted in a row, with two adjacent tensors sharing one index (Halaseh et al., 2022, p.3-4). The tensor diagram notation of a tensor

train can be seen in Figure 2.3. The individual tensors of a tensor train are referred to as carriages and the indices over which the tensors are contracted are called bond indices, while the non-contracted indices are called free indices.

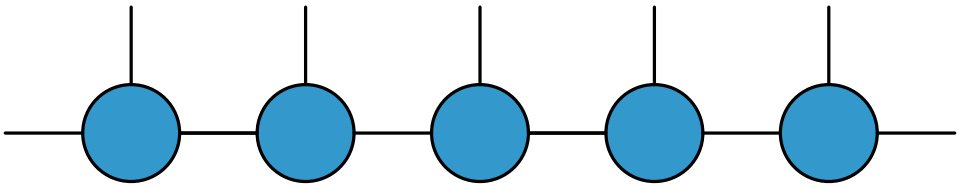

Figure 2.3: A tensor train with five carriages, seven free indices and four bond indices.

One of the most important use cases of tensor trains is to encode data from high-rank tensors in a memory-efficient way without any information loss. A tensor train can encode a high-rank tensor of rank r by preserving the indices of the original tensor as the free indices of the tensor train and using the bond indices to encode the elements of the original tensor in a smaller number of elements in the contracted carriages (Halaseh et al., 2022, p.3-4). The basis of the encoding is the factorisation of the tensor elements. Factors that appear in multiple elements can be stored once, while the association between the factors and the elements from the original tensor are encoded in the bond indices. Reusing factors lowers the memory complexity from being exponential in the rank r, as described in Section 2.1.1, to being linear in the rank r and quadratic in the dimension of the bond indices b, with the memory complexity of a tensor train being $\mathcal{O}(r \cdot d \cdot b^2)$. Because the bond indices are used to encode the appearance of factors in multiple tensor elements, the dimension of the bond indices is essential for the expressiveness of the tensor train (Stoudenmire, 2018). For the full information from the original tensor to be preserved, the dimension of the bond indices b in the tensor train has to be at least $b_{min} = d^{r/2}$, with d being the maximum dimension of the original tensor (Stoudenmire, 2018). The tensor diagram notation of a high-rank tensor encoded as a tensor train can be seen in Figure 2.4.

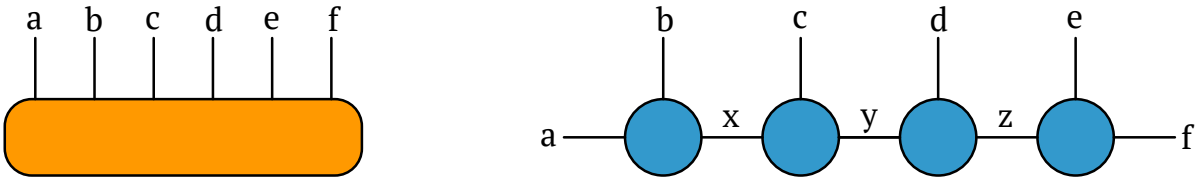

Figure 2.4: A rank-6 tensor (left) and its tensor train representation (right). The indices (a-f) of the original tensor become the free indices of the tensor train. The bond indices (x-z) are added, to encode the full information of the original tensor.

Due to the memory-efficient encoding, an element from the original tensor cannot simply be accessed when encoded in a tensor train. To retrieve an element, the values for the indices corresponding to that element have to be absorbed, meaning all elements that

do not match those values of the free indices in the carriage are deleted. This removal of the free indices results in a reduction of the rank of every carriage, transforming the tensor train into a row of contracted matrices with one vector at each end. By repeatedly multiplying the vectors from the ends with the contracted matrices and summing out the result, the structure can be reduced to a single value, which is the element from the original tensor corresponding to the absorbed index values. In comparison to the near instantaneous access of elements possible with the original tensor, this operation has a runtime of $\mathcal{O}(r \cdot b^2)$ (Stoudenmire, 2018). The properties of tensor trains make them an attractive option for avoiding the exponential memory complexity of high-rank tensors, at the cost of making each access of a tensor element an operation linear in the rank of the tensor.

Some of the operations that are defined on tensors can also be carried out on the respective tensor train representations. These operations include element-wise operations like element-wise addition or element-wise multiplication, but also more complex operations like the Kornecker product or the inner product (Lee and Cichocki, 2018, p.25-33). Being able to carry out operations without the need to recover the high-rank tensors from the tensor train representations means, that tensor trains offer advantages not only for efficient storage but also for algorithms working with high-rank tensors.

2.1.3 Tensor Decomposition

The process of constructing a tensor train representation for a high-rank tensor is called decomposition. The decomposition of an arbitrary tensor into a tensor train is NP-hard (Hillar and Lim, 2013, p.26-27) and many algorithms only find non-optimal decompositions (Halaseh et al., 2022, p.1). Certain properties of tensors can make it easier to find a tensor train representation. One example are orthogonally decomposable (odeco) tensors. A tensor $\mathcal{T}$ of rank r is orthogonally decomposable if it can be constructed from sets of r mutually orthogonal vectors $\{\mathbf{a}_i\} \subset \mathbb{R}^{n_a}, \{\mathbf{b}_i\} \subset \mathbb{R}^{n_b}, \{\mathbf{c}_i\} \subset \mathbb{R}^{n_c}, i \in [1, r] \subset \mathbb{N}$, by adding up the tensor products of these vectors (Halaseh et al., 2022, p.4):

$$\mathcal{T} = \sum_{i=1}^{r} \lambda_i \cdot \mathbf{a}_i \otimes \mathbf{b}_i \otimes \mathbf{c}_i, \tag{2.5}$$

with $\lambda_i \in \mathbb{R}$ being an arbitrary factor. The tensor is symmetric, if the multiplied vectors are identical, meaning $\mathbf{a}_i = \mathbf{b}_i = \mathbf{c}_i$. A tensor train is odeco or symmetric if all of its carriages are odeco or symmetric. The paper "Orthogonal Decomposition of Tensor Trains" (Halaseh et al., 2022) describes multiple algorithms with polynomial runtimes, to find odeco and symmetric tensor train representations. The algorithms work by finding the vectors $\mathbf{a}_i$ and coefficients λ_i for each carriage of the tensor train for any tensors of rank 4 or higher, which have such a representation. The paper also describes an algorithm for finding a decomposition of symmetric but non-orthogonal tensors through the use of a whitening process before the decomposition, which effectively finds an orthogonal equivalent to the decomposed tensor before decomposing it. The approach using whitening

is however only applicable to rank-4 tensors and the whitening process is not guaranteed to find an orthogonal equivalent for every non-orthogonal tensor. Not all tensors are odeco. Odeco tensors are limited in their complexity with a maximum complexity of $\mathcal{O}(d^{r-2})$, while an arbitrary tensor of rank r and maximum dimension d can have a complexity of up to $\mathcal{O}(d^{r-1})$ (Halaseh et al., 2022, p.24). There is currently no algorithm for testing whether or not a tensor is odeco, meaning the only way to verify the odeco property is attempting to find a decomposition (Halaseh et al., 2022, p.4). Finding efficient tensor decompositions is an intensely investigated topic, however, to this date, there is no algorithm for finding a decomposition for a tensor of arbitrary complexity in sub-exponential time.

2.2 Probabilistic Modelling

HMMs find applications in a variety of fields including industrial and consumer applications, speech recognition, and information extraction (Russel and Norvig, 2021, p.76, p.1571). These models contain knowledge about a given system and can be used to answer questions about the system they represent. The structure and function of PGMs in general and HMMs in particular will be described in the following sections.

2.2.1 Probabilistic Graphical Models

PGMs describe real-world environments with a set of $n \in \mathbb{N}$ random variables $\mathscr{R} = \{R_1, ..., R_n\}$, each defining a feature of the system with a given dimension of $o \in \mathbb{N}$ possible values $\mathscr{D}_i = \{r_1, ..., r_o\}$ (Koller and Friedman, 2009, p.20). Each possible combination of the variables' values defines a state of the system. All possible states are stored in the so-called full joint distribution P_R, which contains the probability of each state. Table 2.1 shows the full joint distribution of an exemplary system, which is described by three random variables and aims at modelling the presence of a neighbour in a suburban environment. The variable `Home` encodes whether the neighbour is currently in his house, the variable `Lights` encodes whether the lights are on or off, and the variable `Car` encodes whether the neighbour's car is parked in the driveway.

The full joint distribution has a memory complexity of $\mathcal{O}(r^n)$, where r is the maximum dimension of any variable in $\mathscr{R}$ (Koller and Friedman, 2009, p.21-22). This exponential memory complexity makes it impractical to store the full joint distribution for systems with many variables and high dimensions. PGMs represent a full joint distribution in a compact, graphical form by exploiting conditional independencies between the variables. The incorporation of independencies makes it possible to represent the full joint distribution as a graph with a reduced memory complexity (Koller and Friedman, 2009, p.3-5). The most common PGMs include Markov networks, which are based on undirected graphs and Bayesian networks, which are based on directed graphs and are described in more detail in the following section.

Table 2.1: Full joint distribution of a system described by the three Boolean variables `Home`, `Lights` and `Car`. The full joint distribution gives the probability for each possible state of the system.

`Home`	`Lights`	`Car`	P
false	off	gone	0.51
false	off	there	0.09
false	on	gone	0.17
false	on	there	0.03
true	off	gone	0.09
true	off	there	0.03
true	on	gone	0.06
true	on	there	0.02

2.2.2 Bayesian Networks

A Bayes network represents a full joint distribution in the form of an acyclic, directed graph $B = (\mathscr{V}, \mathscr{E})$, with $\mathscr{V}$ being the nodes and $\mathscr{E}$ being the edges of the graph. Each node $v \in \mathscr{V}$ of the graph represents one variable from the full joint distribution, while the edges $\mathscr{E}$ encode the conditional independencies between the variables (Koller and Friedman, 2009, p.45-54). Two random variables R_1 and R_2 are called independent if the probability of two states of the variables occurring together is the same as the product of the two individual probabilities (Koller and Friedman, 2009, p.23):

$$P(R_1, R_2) = P(R_1) \cdot P(R_2). \tag{2.6}$$

This connection means, that the two random variables do not influence each other and are notated as $R_1 \perp R_2$. Bayesian networks do not just encode independencies but also conditional independencies between variables. The variables R_1 and R_2 are called conditionally independent given a set of over variables $\mathscr{R}'$, if the probability of two variables combined is the same as the probability of the individual variables, provided that the state of the variables in $\mathscr{R}'$ is known (Koller and Friedman, 2009, p.24):

$$P(R_1, R_2 \mid \mathscr{R}') = P(R_1 \mid \mathscr{R}') \cdot P(R_2 \mid \mathscr{R}'). \tag{2.7}$$

This means, that the two variables have the common influencing factors $\mathscr{R}'$. If the variables in $\mathscr{R}'$ already have values assigned, the variables R_1 and R_2 are independent, which is denoted as $R_1 \perp R_2 \mid \mathscr{R}'$. The structure of a Bayesian network encodes the conditional independencies in the form of the rule, that each variable in the model is independent of its non-descendants given the variables from its parent nodes (Koller and Friedman, 2009, p.56-57).

The knowledge about conditional independencies between the variables of a PGM has to either come from context or has to be figured out by explicitly testing groups of

variables for conditional independencies. Given the conditional independencies between the variables in a full joint distribution, the distribution can be decomposed into several conditional probability distributions (CPDs). The CPD for each variable is then stored in the node of the graph corresponding to the variable for that CPD. All CPDs combined contain the full information of the original full joint distribution, which can be recovered by multiplying all CPDs together (Koller and Friedman, 2009, p.52-54). The decomposition of a full joint distribution into CPDs has the advantage of reducing the memory complexity from being exponential in the number of variables n, as described in Section 2.2.1, to being just linear in the number of variables, as a Bayesian network has a memory complexity of $\mathcal{O}(n \cdot r^m)$, with m being the maximum number of parents any node has in the graph (Russel and Norvig, 2021, p.904). Therefore the effectiveness of the reduction of memory demand is mainly dependent on the degree of independence between the variables in the model.

Building on the example from Section 2.2.1, it can be assumed that `Lights`$\perp$`Car` | `Home`, meaning given the information whether the neighbour is home or not, the status of the lights and the presence of the car do not influence each other. With this information, the full joint distribution from Table 2.1 can be decomposed into three CPDs which can be seen in the resulting Bayesian network from Figure 2.5.

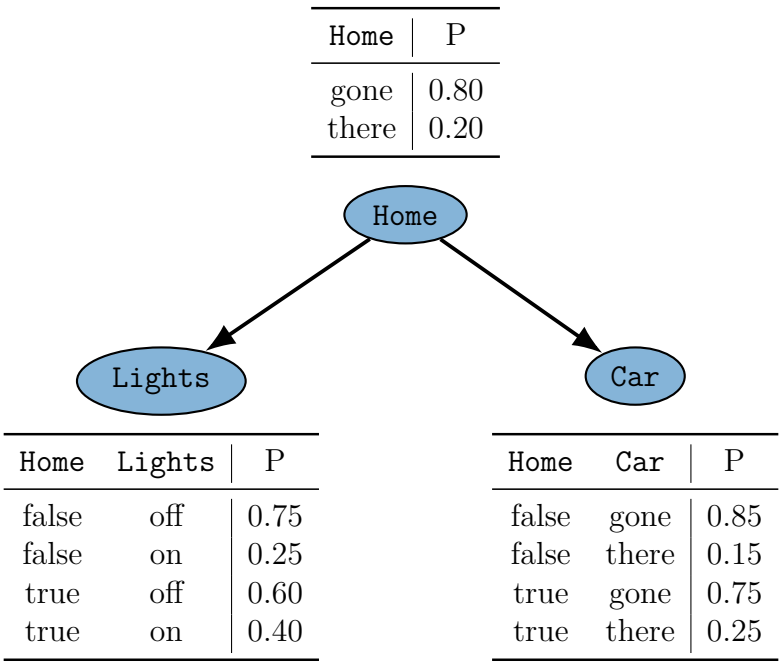

Figure 2.5: Bayesian network corresponding to the full joint distribution from Table 2.1. The variables `Lights` and `Car` have the common parent `Home`, which encodes the conditional independence `Lights`$\perp$`Car` | `Home`.

The parallel between the decomposition of full joint distributions into CPDs and the

decomposition of high-rank tensors into tensor trains is not a coincidence but a logical duality. Both decompositions reduce the memory complexity from being exponential in the number of indices or variables to being linear in those parameters. This duality between tensor networks and different kinds of PGMs is the main reason tensor networks are increasingly used in machine learning and AI contexts (Glasser et al., 2019, p.2-6). The logical equivalent of contracting a tensor network to retrieve one or several elements from the original tensor are the inference algorithms, which can be used to retrieve one or several probabilities from the full joint distribution, as described in Section 2.3.

2.2.3 Hidden Markov Models

HMMs are a subclass of so-called dynamic Bayesian networks, which are a type of PGM used to describe the development of a system over time, with each state being defined by an allocation of specific values to the random variables (Koller and Friedman, 2009, p.202–204, p.208–209). HMMs typically have only two variables. The hidden variable cannot be observed, meaning its value is always unknown. The state of the observable variable can be observed in each time step (Russel and Norvig, 2021, p.885). In a temporal version of the exemplary system modelling the neighbour's presence, the variable `Home` is a hidden variable, because it cannot be directly observed whether or not the neighbour is home. The variable `Lights` is an observable variable because it can be seen from outside the house whether or not the lights are turned on or off. The resulting HMM for the exemplary system can be seen in Figure 2.6.

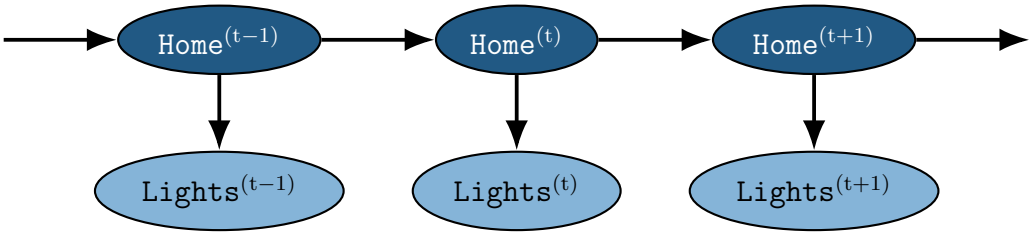

Figure 2.6: HMM with the observable variable `Lights` and the hidden variable `Home`. The exclusive dependence of the observable on the hidden variable is encoded in the directed edge between them. The Markov assumption is encoded in the edges between the time steps.

HMMs make two important assumptions about the dependencies of their variables. The first assumption is that the observable variable is only influenced by the hidden variable (Rabiner and Juang, 1986, p.7). In real systems, this assumption is often an approximation. In the example, the state of the variable `Lights` will not be exclusively influenced by the presence of the neighbour, but might also be influenced by power outages. However, approximations of the exclusive dependence between the observable and the hidden variable can still lead to HMMs producing an accurate representation of the system at hand. The second assumption made by HMMs is the so-called Markov assumption. The more general Markov-k assumption states, that the hidden variable in each time

step only depends on the states of the hidden variable in the $k \in \mathbb{N}$ time steps before. HMMs typically use the Markov-1 assumption, which is often called Markov assumption for short, meaning the hidden state in the next time step exclusively depends on the current state of the hidden variable (Russel and Norvig, 2021, p.867). Both dependency assumptions can also be seen in Figure 2.6 in the form of the directed edges of the HMM, as described in Section 2.2.2.

$\texttt{Home}^{(t)}$	$\texttt{Home}^{(t+1)}$	**A**
false	false	0.70
false	true	0.30
true	false	0.10
true	true	0.90

Home	Lights	**B**
false	off	0.75
false	on	0.25
true	off	0.60
true	on	0.40

$\texttt{Home}^{(t)}$	$\boldsymbol{\pi}$
false	0.55
true	0.45

Figure 2.7: Transition matrix **A**, emission matrix **B** and initial state vector $\boldsymbol{\pi}$ of the HMM.

HMMs are typically represented by the tuple $M = (\mathbf{A}, \mathbf{B}, \boldsymbol{\pi})$ of two matrices and one vector. The first matrix is the so-called transition matrix **A**. It contains the probabilities of all possible transitions between two hidden states when advancing from one time step to the next. The second matrix is the so-called emission matrix **B**. It contains the probabilities of observing a certain state of the observable variable, given the current state of the hidden variable. The vector $\boldsymbol{\pi}$ contains the initial probabilities of the hidden states (Rabiner and Juang, 1986, p.7). Together these three components fully describe the behaviour of the system encoded in the HMM and can be used to answer questions on the system as described in Section 2.3. The transition and emission matrices as well as the initial state vector of the exemplary model can be seen in Figure 2.7.

As it becomes evident in the exemplary system, the limitation to two variables in HMMs can lead to information about a system being left out of the model. The example HMM only includes `Lights` as the observable variable and leaves out the `Car` variable. This information can be included by considering a multivariate HMM. Multivariate HMMs can have more than one observable variable (Visser and Speekenbrink, 2022, p.201). Adding the variable `Car` to the HMM, as shown in Figure 2.8, does not change the transition matrix but does result in the emission probabilities either having to be represented by one matrix per variable or by a higher rank tensor. This high-rank representation of emission probabilities is the basis for the internal tensor train decomposition which is discussed in Section 3.1.

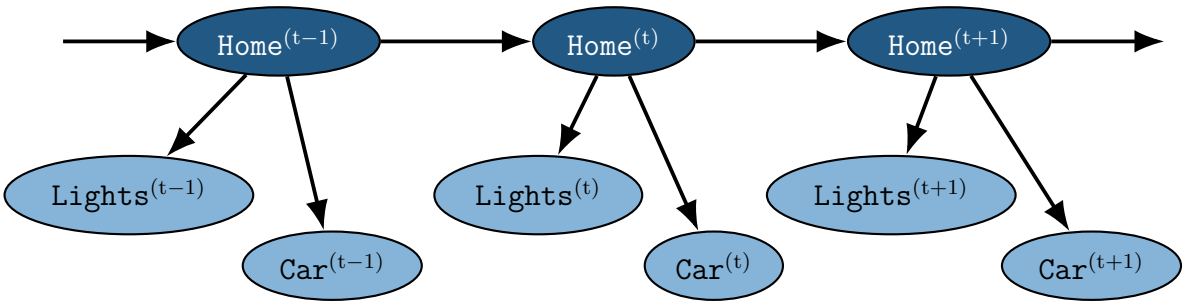

Figure 2.8: Multivariate HMM with two observable variables, which share the hidden variable as the parent node in the graph.

2.3 Inference Algorithms

The goal of inference algorithms is to answer queries on a system encoded in a PGM. These algorithms often incorporate observations made on the model called evidence (Koller and Friedman, 2009, p.5). In the case of an HMM, the evidence consists of the observed values of the observable variables at a series of time steps. There are different types of inference queries. So-called maximum a posteriori (MAP) queries aim at finding the most likely state of a system while optionally given a partial observation of the system. Most probable explanation (MPE) queries find the most likely states of all variables that are not set with evidence for a partially observed system (Russel and Norvig, 2021, p.872-873).

Specifically for HMMs, a common query type is finding the probability of a hidden state at a given time based on an evidence sequence, which can be computed using the forward algorithm described in Section 2.3.1. Other typical query types for HMMs are calculating the probability of observing a certain evidence sequence, predicting the most likely future states of a system given an evidence sequence up to the current time step, and computing the most likely sequence of hidden states to a sequence of observed states (Rabiner and Juang, 1986, p.8). The last query type is the classical MPE query for HMMs. While it is typically desirable to calculate the exact answer to an inference query, there are algorithms to compute approximate inference, which can be used when the calculation of the exact answer would take too long (Russel and Norvig, 2021, p.822). This might be the case when using PGMs with many variables of high dimension, very long evidence sequences or when working in time-sensitive applications.

2.3.1 Forward Algorithm

The forward algorithm is used to calculate the belief states of the hidden variable given an evidence sequence. A believe state $P(h^{(t)} \mid o^{(0:t)})$ is the probability of the hidden state $h^{(t)} \in \mathscr{H}$ at a time step t given an evidence sequence of observations $o^{(0:T)} \in \mathscr{O}$ of the states of the observable variable from the start of the modelled time up to the time step T (Koller and Friedman, 2009, p.652). Here $\mathscr{H}$ is the set of all hidden states and

$\mathcal{O}$ is the set of all observable states. It is important to understand, that the forward algorithm cannot be used on its own to calculate the most likely sequence of hidden states by simply choosing $\underset{h^{(t)}}{\text{argmax}}\, P(h^{(t)} \mid o^{(0:T)})$ for every time step (Koller and Friedman, 2009, p.652-653). The forward algorithm only incorporates the evidence made up to the time step t in a so-called forward pass to calculate the belief states and does not incorporate evidence made after the timestep. Algorithms like the Viterbi algorithm (Russel and Norvig, 2021, p.881-883) do incorporate all given evidence, and can be used to compute the MPE sequence of hidden states for a given evidence sequence.

Using the example HMM from Figure 2.6, the forward algorithm could be used to compute the probability of the neighbour being home at six o'clock in the morning, given a sequence of observations of the neighbour's lights from midnight until six o'clock. The query would be written as $P(true^{(6)} \mid \texttt{Lights}^{(0:6)})$. The evidence would be the observation of the variable `Lights` and could look something like this: $\texttt{Lights}^{(0:6)} = (off,\ off,\ off,\ off,\ on,\ on)$. Because the forward algorithm only incorporates evidence made before the requested time step, the queries are typically asked for the current time step T, which is the last time step from the evidence sequence. However, any time step t within the sequence can be the basis for a query.

The forward algorithm uses so called forward messages $\alpha_h^{(t)} = P(h^{(t)}, o^{(0:T)})$, which encode the joint probability of the hidden state h associated with the forward message and the evidence sequence $o^{(0:T)}$ up to the timestep T (Rabiner and Juang, 1986, p.9). It is important to recognise the difference between the joint probability $P(h^{(t)}, o^{(0:T)})$ of a hidden state and an evidence sequence and the conditional probability $P(h^{(t)} \mid o^{(0:T)})$ of a state given an evidence sequence. The conditional probability is the answer to the query and can be computed from the joint probability by normalising it with the sum over the forward messages for all possible hidden states in a time step (Koller and Friedman, 2009, p.18):

$$P(h^{(t)} \mid o^{(0:T)}) = \frac{P(h^{(t)}, o^{(0:T)})}{P(o^{(0:T)})} = \frac{\alpha_h^{(t)}}{\alpha^{(t)}} \quad \text{with} \quad \alpha^{(t)} = \sum_h \alpha_h^{(t)}. \tag{2.8}$$

Due to the conditional independencies in an HMM, which are described in Section 2.2.3, the forward message $\alpha_h^{(t)}$ can be calculated from the forward message of the prior time step $\alpha_h^{(t-1)}$ and values from the emission and transition matrix using the formula (Rabiner and Juang, 1986, p.9):

$$\begin{aligned} \alpha_h^{(t)} &= \mathbf{B}_{o^{(t)}}^{h^{(t)}} \sum_{h^{(t-1)}} \mathbf{A}_{h^{(t-1)}}^{h^{(t)}} \alpha_{h^{(t-1)}}^{(t-1)} \\ \alpha_h^{(0)} &= \boldsymbol{\pi}_h. \end{aligned} \tag{2.9}$$

This calculation uses all transition values from $\mathbf{A}$ but only one value from the emission matrix $\mathbf{B}$. Which value from $\mathbf{B}$ is used depends on the observed evidence $o^{(t)}$. The access

of individual emission probabilities depending on the observed evidence is a characteristic shared between all inference algorithms for HMMs. Besides the forward algorithm, these include the forward-backward algorithm (Rabiner and Juang, 1986, p.9) for MAP queries and the Viterbi algorithm (Russel and Norvig, 2021, p.881-883) for MPE queries. Because of this similar incorporation of emission probabilities, the forward algorithm can be used as a representative for all inference algorithms on HMMs when it comes to working with emission probabilities.

Due to the nature of Equation (2.9), the forward algorithm is typically implemented recursively. The base case of the recursion is the forward message from the first time step, which is equal to the field from the vector with the initial probabilities $\boldsymbol{\pi}$ corresponding to the hidden state h: $\alpha_h^{(0)} = \boldsymbol{\pi}_h$. The formulas described above lead to the forward algorithm being implemented according to the pseudocode from Algorithm 2.1.

Algorithm 2.1 Forward Algorithm

Input
$\mathbf{A}, \mathbf{B}, \boldsymbol{\pi}$
$o^{(0:T)}$

function ForwardMessage(TimeStep t, HiddenState h)
  if $t \neq 0$ **then**
    $\alpha_h^{(t)} = \mathbf{B}_{o^{(t)}}^{h^{(t)}} \sum_{h^{(t-1)}} \mathbf{A}_{h^{(t-1)}}^{h^{(t)}}$ **ForwardMessage**$(t-1,\ h^{(t-1)})$
    return $\alpha_h^{(t)}$
  else
    return $\boldsymbol{\pi}_h$

$\alpha^{(T)} = 0$
for all $h \in \mathscr{H}$ **do**
  $\alpha_h^{(T)} =$ **ForwardMessage**$(T,\ h)$
  $\alpha^{(T)} = \alpha^{(T)} + \alpha_h^{(T)}$
return $\alpha_h^{(T)} / \alpha^{(T)}$

Because the forward algorithm makes use of the independence assumptions in HMMs, it does not have to consider all possible sequences of hidden states but can work along the length of the evidence sequence in time. This usage of independencies reduces the algorithm's runtime from being exponential in the length of the evidence sequence T to being just linear in T with a runtime of $\mathcal{O}(T \cdot |\mathscr{H}|^2)$ (Rabiner and Juang, 1986, p.9). The runtime results from the algorithm having to factor in $T \cdot |\mathscr{H}|$ emission probabilities for the recursive calculation of the forward message associated with each hidden state and needing to do so for all $|\mathscr{H}|$ hidden states to be able to normalise the forward messages $\alpha_h^{(T)}$ at time T with the normalisation factor $\alpha^{(T)}$. If the algorithm would not use the independence between the variables, it would have to consider a combination

of length $|\mathscr{H}|$ for all T timesteps resulting in $|\mathscr{H}|^T$ considered states and therefor a runtime exponential in T (Rabiner and Juang, 1986, p.9).

The forward algorithm does not change significantly when using it on a multivariate HMM. Because the number of hidden states does not change when adding observable variables to a multivariate HMM, the loop over all hidden states that is performed during the calculation of each forward message is unaffected by the addition of observable variables. The main change to the algorithm is the denotation of emission probabilities. Every evidence in a sequence is now a tuple of values instead of a single scalar. Therefore the emission probabilities used in the calculation of the forward messages are denoted with more than two indices, however, the number of accessed emission probabilities and the runtime of the forward algorithm do not change.

2.3.2 Baum-Welch Algorithm

The Baum-Welch algorithm is not an inference algorithm, but an algorithm for learning the parameters of an HMM, like the entries in $\mathbf{A}$, $\mathbf{B}$ and $\boldsymbol{\pi}$. However, it is considered here alongside the forward algorithm, because learning algorithms and inference algorithms together cover all of the most important algorithms used with PGMs. Besides that, the Baum-Welch algorithm shares a lot of basic approaches with inference algorithms by making use of forward and backward passes, as can be seen by the formulas described below.

The Baum-Welch algorithm is an expectation-maximisation algorithm, which is used to adjust the parameters in an HMM, based on a given set of training data. The goal is to match the HMM to the system, from which the training data was drawn, as closely as possible. In the expectation step of the algorithm, the expected number of transitions and emissions is calculated based on a given evidence sequence, using a forward-backward procedure identical to the one found in inference algorithms like the forward-backward algorithm (Rabiner and Juang, 1986, p.10-11). The calculation of the forward messages is done according to the definition given in Equation (2.9). The corresponding backward messages $\beta_h^{(t)} = P(o^{(t+1:T)} \mid h^{(t)})$ encode the probability of making the observation sequence $o^{(t+1:T)}$ with the knowledge of the hidden state being $h^{(t)}$ at time t. The calculation of backward messages is also done recursively, but starting at $t = T$ and going backwards in time towards the beginning of the evidence sequence $o^{(0:T)}$ (Rabiner and Juang, 1986, p.9):

$$\begin{aligned} \beta_h^{(t)} &= \sum_{h^{(t+1)}} \mathbf{A}_{h^{(t+1)}}^{h^{(t)}} \mathbf{B}_{o^{(t)}}^{h^{(t+1)}} \beta_{h^{(t+1)}}^{(t+1)} \\ \beta_h^{(T)} &= 1. \end{aligned} \tag{2.10}$$

Just like with the forward pass, the runtime complexity of calculating the forward messages along an evidence sequence with T time steps is $\mathcal{O}(T \cdot |\mathscr{H}|^2)$. The runtime is

identical to the one of the forward algorithm in Section 2.3.1 because the backward pass carries out analogous operations, with the main difference being that it moves along the evidence sequence from timestep T to 1 instead of from 1 to T like the forward algorithm. Based on the forward and backward messages, the probability of the system being in the hidden state h at the time step t given the observation sequence $o^{(0:T)}$ can be defined as $\gamma_h^{(t)} = P(h^{(t)} \mid o^{(0:T)})$, which can be calculated as follows (Rabiner and Juang, 1986, p.10):

$$\gamma_h^{(t)} = \frac{\alpha_h^{(t)} \beta_h^{(t)}}{\sum_h \alpha_h^{(t)} \beta_h^{(t)}}. \tag{2.11}$$

The difference between the conditional probability $\gamma_h^{(t)}$ and the conditional probability described in Equation (2.8) for the forward algorithm is, that $\gamma_h^{(t)}$ is based on the full observation sequence, as it incorporates the recursively defined backward message $\beta_h^{(t)}$. The conditional probability of the model transitioning from the hidden state h^t at time t to the hidden state h^{t+1} at time $t+1$ is $\xi_h^{(t)} = P(h^t, h^{t+1} \mid o^{(0:T)})$, with the given observation sequence $o^{(0:T)}$. The formula for calculating this probability is (Rabiner and Juang, 1986, p.11):

$$\xi_h^{(t)} = \frac{\alpha_h^{(t)} \mathbf{A}_{h^{(t+1)}}^{h^{(t)}} \mathbf{B}_{o^{(t+1)}}^{h^{(t+1)}} \beta_h^{(t+1)}}{\sum_h \alpha_h^{(t)} \beta_h^{(t)}}. \tag{2.12}$$

With these components, all parameters of the model $M = (\mathbf{A}, \mathbf{B}, \boldsymbol{\pi})$ can be updated (Rabiner and Juang, 1986, p.11):

$$\begin{aligned} \boldsymbol{\pi}_{\boldsymbol{h}}^{*} &= \gamma_h^{(1)} \\ \mathbf{A}^{*h^{(t)}}_{\ h^{(t+1)}} &= \frac{\sum_{t=0}^{T} \xi_h^{(t)}}{\sum_{t=0}^{T} \gamma_h^{(t)}} \\ \mathbf{B}^{*h}_{\ o} &= \frac{\sum_{t=0}^{T} 1_{o^{(t)}=o}\ \gamma_h^{(t)}}{\sum_{t=0}^{T} \gamma_h^{(t)}}, \end{aligned} \tag{2.13}$$

where $1_{o^{(t)}=o}$ is an indicator function which is $1_{o^{(t)}=o} = 1$ if $o^{(t)} = o$ and $1_{o^{(t)}=o} = 0$ otherwise. Equation (2.13) shows, that the updated values for the transition matrix $\mathbf{A}$ and the emission matrix $\mathbf{B}$ are built by counting the number of expected emissions or transitions and normalising them with the total number of emissions or transitions to obtain the corresponding probabilities. Therefore the Baum-Welch algorithm updates the HMM parameters to maximise the probability of observing the given training data (Rabiner and Juang, 1986, p.11).

Just like with the forward algorithm, the Baum-Welch algorithm does not change significantly when being used on multivariate HMMs. As described in Section 2.3.1, the main change is that evidence and emission probabilities are now denoted with tuples of values instead of one value each for the hidden and observable variables. Because all loops in the calculations of the forward and backward messages as well as the normalisation

are performed over all possible hidden states or all time steps, neither the process nor the runtime of the Baum-Welch algorithm change when adding observable variables to a multivariate HMM.

3 Algorithm Approaches

This chapter presents the fundamental idea of the new representation of HMMs using internal decomposition and gives an overview of the approaches to efficient learning and inference algorithms. The first part of the chapter explains how the two fields of HMMs and tensor trains are combined in the new representation. The second part deals with two parallelisation approaches at inference algorithms. Because efficient probabilistic inference is the main topic of this thesis, the inference algorithms are described on a lower abstraction level as the more conceptual new representation. Concrete examples for both the new representation and the parallelisation of inference algorithms can be found in the implementation that is described in Chapter 4. In addition to the new representation and efficient inference, this chapter also discusses approaches to learning internally decomposed models. The learning algorithms are discussed on a higher abstraction level than the inference algorithms because they are not the main focus of this thesis, however, learning is considered because the benefits of efficient inference algorithms are only applicable to the real world if the corresponding models can be learned efficiently.

3.1 Representing HMMs with Tensor Trains

As described in Section 2.2.3, adding additional observable variables to a multivariate HMM creates multiple emission matrices within the model. Besides the representation as individual emission matrices, these can be interpreted as a tensor of rank $r = n_O + 1$, with n_O being the number of observable variables in the multivariate HMM. The additional rank results from the inclusion of the single hidden variable. For multivariate HMMs with a low number of total variables and low dimensions, the exponential memory complexity of this tensor can be neglected. However, there are situations in which the number of variables and their dimensions have a size which leads to the emission tensor having a memory demand that is too high for it to be stored or worked with efficiently.

An example of a model with high memory demand can be constructed by expanding on the exemplary system from the previous chapters. The model used in Chapter 2 only has three variables, each having a dimension of 2. Therefore the emission probabilities can be represented by a rank-3 tensor with a manageable memory demand. However, the model could easily be extended with additional variables. Examples could be `Bins`, stating whether the bins are out in the street, `Weekend` for whether or not the current time step is on the weekend, `Lawn` for modelling the length of grass in the neighbour's lawn and `Curtains` describing the state of the blinds and curtains. With the additional

variables, the system now has a total of seven variables. Considering that variables like `Lawn`, `Lights`, and `Curtains` could have high dimensions, this easily comprehensible model leads to an emission tensor with a very high memory demand. Especially problematic can be the discretisation of variables that describe continuous properties like length or temperature and can lead to dimensions well beyond 100 different possible values if high accuracy is necessary to describe the modelled system.

As described in Section 2.1.2, one possibility for dealing with the high memory demand of a high-rank tensor is decomposing the tensor into a tensor network. In this context, the emission tensor is decomposed into an emission tensor train with one free index for each variable in the HMM. The decomposition can make the representation of the emission probabilities possible even for high-complexity HMMs. The tensor train representation of the emission probabilities in the exemplary system would have seven free indices for the seven variables and would therefore consist of five carriages, each with a rank of 3. Assuming that all indices have an identical dimension of d, the memory complexity is reduced from $\mathcal{O}(7^d)$ to $\mathcal{O}(5 \cdot 3 \cdot d^3)$. The memory complexity of the tensor train representation results from the five carriages, each with rank 3 and identical dimension d, according to the memory complexity described in Section 2.1.2. Assuming a dimension of 10 for each variable, the number of stored elements goes from about $2.8 \cdot 10^8$ down to just 15000 elements. However, due to the decomposition into a tensor train, the access of individual probabilities from the emission tensor is now an operation with a runtime linear in the number of variables, as pointed out in Section 2.1.2. These high access times increase the runtime of algorithms that rely on the frequent access of emission probabilities, like the forward algorithm described in Section 2.3.1. In addition to the higher runtimes of inference algorithms, the decomposition of the emission tensor requires an additional computational effort. The runtime of the decomposition can be kept low by using an odeco and symmetric tensor decomposition, described in Section 2.1.3. One advantage of this new approach to representing HMMs with tensor trains is the opportunity to use the structure of the tensor train for more efficient inference algorithms through parallelisation. The different parallelisation approaches to using this special structure for lower runtimes of various algorithms on HMMs are described in the following section.

It is important to emphasise the difference between this usage of tensor trains for representing the emission probabilities of an HMM and the duality of PGMs and tensor networks. When representing a complete PGM with a tensor network, the decomposition contains all the data needed to represent the PGM. In contrast to that, the decomposition described above only offers an alternative representation of the emission probabilities in an HMM. It uses the memory reduction associated with tensor decomposition to make a complete multivariate HMM storable, by only decomposing the emission probabilities, which are by far the part with the highest memory demand, when representing an HMM with high-rank tensors. Due to the focus on memory reduction, the decomposition lacks the logical meaning that is inherent to the duality of tensor networks and PGMs and also introduces additional runtime when accessing emission probabilities from the tensor train representation.

3.2 Parallelising Inference Algorithms

The internal tensor train representation allows for two different kinds of parallelisation of inference algorithms. Both of these approaches aim at reducing the runtime complexity, which was increased by the tensor train decomposition. Compared to the near instantaneous access with the representation of the emission probabilities as a high-rank tensor, the access of emission probabilities becomes a task with a runtime linear in the number of variables with the tensor train representation, as described in Section 3.1. The additional runtime could make it unattractive or even impossible to work with complex models in a tensor train representation, even though the tensor train decomposition solves the issue of memory demand. The parallelisation approaches reduce the inference runtime, to make it viable to work with complex models while using a tensor train representation.

3.2.1 Parallel Evidence Incorporation

Inference algorithms like the forward algorithm are often used to answer queries on already completed and fully available evidence sequences. The exception to this are online applications, which incorporate observations as they are made. Which emission probabilities are utilised in the computation is dependent on the evidence, as pointed out in Section 2.3.1. Without any modifications to the procedure of an inference algorithm, the direct access of an emission probability from a high-rank tensor is replaced with the time-intensive operation of calculating the emission probability from the emission tensor train.

If the full evidence sequence is available at the start of the algorithm's execution, the emission probabilities that will be needed during the execution can be predicted based on the evidence. The knowledge about the accessed emission probabilities allows the calculation of all required emission probabilities beforehand and eliminates the need to interrupt the inference algorithm every time an emission probability is accessed. Just moving the calculations necessary to access the emission probabilities ahead of the inference calculations, however, does not decrease the total runtime. A runtime reduction can be achieved by splitting up the evidence sequence and calculating the emission probabilities for each part of the total sequence in a separate thread. Each time step in the sequence typically corresponds to one emission probability accessed by an inference algorithm, meaning that a split of the evidence sequence into parts with an equal number of time steps leads to an even distribution of computational effort among the threads.

Pseudocode for the pre-calculation of the emission values can be seen in Algorithm 3.1. The code shows how n_{thr_evi} threads work simultaneously by each calculating one section of the emission values accessed by the forward algorithm. The emission probability for each time step t is calculated from the tensor train representation $\mathbf{B}_{TT}$ of the emission probabilities and the given evidence $o^{(t)}$. Each thread fills one part of an array for all T emission values, which can be used afterwards in the forward algorithm. The only change

to the forward algorithm from Algorithm 2.1 is that the emission probability needed for the calculation of the forward message corresponding to a time step t is now accessed from the t-th element of the pre-calculated array, instead of from the representation of the emission probabilities directly.

Algorithm 3.1 Emission Probability Pre-Calculation

Input
$\mathbf{B}_{TT}$, $o^{(0:T)}$, $emission[T]$, n_{thr_evi}

1: **function** GetEmissionValue(Evidence $o^{(t)}$)
2: $\quad\mathbf{B}_{TT}$.**absorbEvidence**($o^{(t)}$)
3: $\quad e^{(t)} = \mathbf{B}_{TT}$.**contract**()
4: $\quad$**return** $e^{(t)}$

5: **function** CalculateEmissionValues(StartTime t_{start}, Length l)
6: $\quad$**for** $t_{start} \leq t < t_{start} + l$ **do**
7: $\quad\quad e^{(t)} =$ **GetEmissionValue**($o^{(t)}$)
8: $\quad\quad emission[t] = e^{(t)}$

9: ThreadPool $threads[n_{thr\_evi}]$
10: $l = T$ / n_{thr_evi}
11: **for** $0 < i < n_{thr\_evi}$ **do**
12: $\quad threads$.**addThread**(**CalculateEmissionValues**, $0 + i \cdot l$, l)
13: $threads$.**waitForAll()**

The access of an emission probability from the emission tensor train has a runtime of $\mathcal{O}(|\mathscr{V}| \cdot b^2)$, with $\mathscr{V} = \mathscr{H} \cup \mathscr{O}$ being the set of all variables from the HMM. The additional runtime introduced by the new representation is determined by the runtime of computing one emission probability per observation in an evidence sequence of length T and is therefore $\mathcal{O}(T \cdot |\mathscr{V}| \cdot b^2)$. When neglecting the overhead of a multi-threaded setup, this runtime can be reduced linearly in the number of threads to:

$$\mathcal{O}(\frac{T}{n_{thr\_evi}} \cdot |\mathscr{V}| \cdot b^2). \tag{3.1}$$

This linear reduction in runtime for inference algorithms can significantly reduce the added computational effort introduced by the decomposition of the emission tensor into a tensor train. However, even when neglecting any potential overhead, the reduction is limited by $n_{thr\_evi} \leq T$, since each thread needs to work on at least one observation. Due to this limitation, the approach of parallel evidence incorporation on its own can only reduce the additional runtime, but cannot make up for all of it. In addition to that, real-world limitations like runtime overhead from the creation of threads will lead to the optimal number of threads n_{thr_opt} for any realistic system to comply with $1 < n_{thr\_opt} < T$.

3.2.2 Parallel Tensor Train Contraction

To obtain an emission probability from the tensor train representation, the evidence corresponding to that emission probability has to be absorbed into the tensor train, followed by the multiplication of the resulting chain of contracted matrices. Absorbing evidence is done by deleting all elements from a carriage, which do not correspond with the given value for the variable associated with the carriage's free index. The evidence absorption removes all free indices from the tensor train and creates a series of matrices, contracted over the bond indices, with one vector at each end. The tensor diagram of a tensor train with absorbed evidence can be seen in Figure 3.1. This structure can then be contracted from the edges to obtain the emission probability corresponding with the observation and given value for the hidden variable.

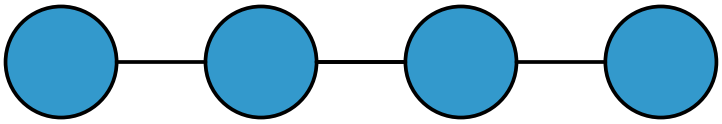

Figure 3.1: Emission tensor train with absorbed evidence. The free indices have been removed, reducing the rank of the carriages.

The structure of a tensor train provides the opportunity to parallelise the process of obtaining an emission probability corresponding to a given set of values for the variables of an HMM. The emission tensor train can be split up along its length with each section, containing at least one carriage, being assigned to a separate thread. Each thread can then absorb the evidence into the carriages assigned to it and then multiply together all carriages from the assigned section of the tensor train. The contraction over the bond indices, which are split between two threads, needs to be calculated from the interim results that are returned by the threads.

The process of absorbing and contracting a tensor train in two parallel threads is illustrated in Figure 3.2. It is also presented as pseudo-code in Algorithm 3.2. Both the graphical representation and the pseudo-code show how each thread first absorbs the evidence in one section of the tensor train. After that, both visualisations display that each thread contracts some of the tensor train's carriages, while the remaining intermediate results have to be contracted sequentially after all threads are finished.

This technique of parallel contraction can be applied to any tensor network. The method described above is the most basic form of parallel contraction that is applicable to tensor trains and has the number of threads as its only parameter. When working with more complex tensor networks, the task of finding an efficient or even optimal way of parallelising the contraction in regards to runtime and memory complexity of interim results becomes a far more challenging problem (Dudek et al., 2020).

Similar to the parallelisation described in the previous section, this approach can reduce

Algorithm 3.2 Parallel Contraction

Input
$\mathbf{B}_{TT}$, $o^{(0:T)}$, n_{thr_con}

1: **function** ContractParallel(Evidence $o^{(t)}$, StartCarriage c_{start}, Length l)
2: **for** $c_{start} < c < c_{start} + l$ **do**
3: $\mathbf{B}_{TT}[c]$.**absorbEvidence**($o^{(t)}$)
4: $\mathbf{B}_{TT}[c_{start}$ **to** $c_{start} + l]$.**contract**()

5: ThreadPool $threads[n_{thr\_con}]$
6: $l = T$ / n_{thr_con}
7: **for** $0 < i < n_{thr\_con}$ **do**
8: $threads$.**addThread**(**ContractParallel**, $0 + i \cdot l$, l)
9: $threads$.**waitForAll()**
10: $\mathbf{B}_{TT}$.**contract**()

the runtime of retrieving a single emission probability from an emission tensor train from a complexity of $\mathcal{O}(|\mathscr{V}| \cdot b^2)$ to a complexity linearly reduced by the number of threads n_{thr_con}:

$$\mathcal{O}(\frac{1}{n_{thr\_con}} \cdot |\mathscr{V}| \cdot b^2). \tag{3.2}$$

However, while the number of time steps T in an evidence sequence might easily reach into the hundreds, with systems making observations at high frequency or over a long time, the number of variables in the model, and therefore the number of carriages in the emission tensor train, will likely be much lower: $|\mathscr{V}| \ll T$. The limited tensor train length limits the maximum number of threads for the parallel contraction to be much lower than the maximum number of threads for the parallel incorporation of evidence described in Section 3.2.1. In addition to that, a low number of carriages increases the impact of an uneven distribution of the overall workload. If the carriages cannot be divided between the threads evenly, the runtime added by having to wait for the completion of the threads with a higher number of carriages to process might significantly lower the theoretical speed-up.

Because the approaches of parallelising the incorporation of evidence and parallelising the contraction of the emission tensor train aim at different parts of the inference calculation, they do not influence each other. The independence of the two approaches means, that a combination of both approaches has a maximum expected runtime complexity of

$$\mathcal{O}(\frac{T}{n_{thr\_evi} \cdot n_{thr\_con}} \cdot |\mathscr{V}| \cdot b^2) \tag{3.3}$$

for the calculation of all emission probabilities needed for an evidence sequence of length T, with one emission probability calculated per time step. This runtime complexity is

based on the assumption that all threads can work in parallel, which might not be the case for real-world systems with limited resources.

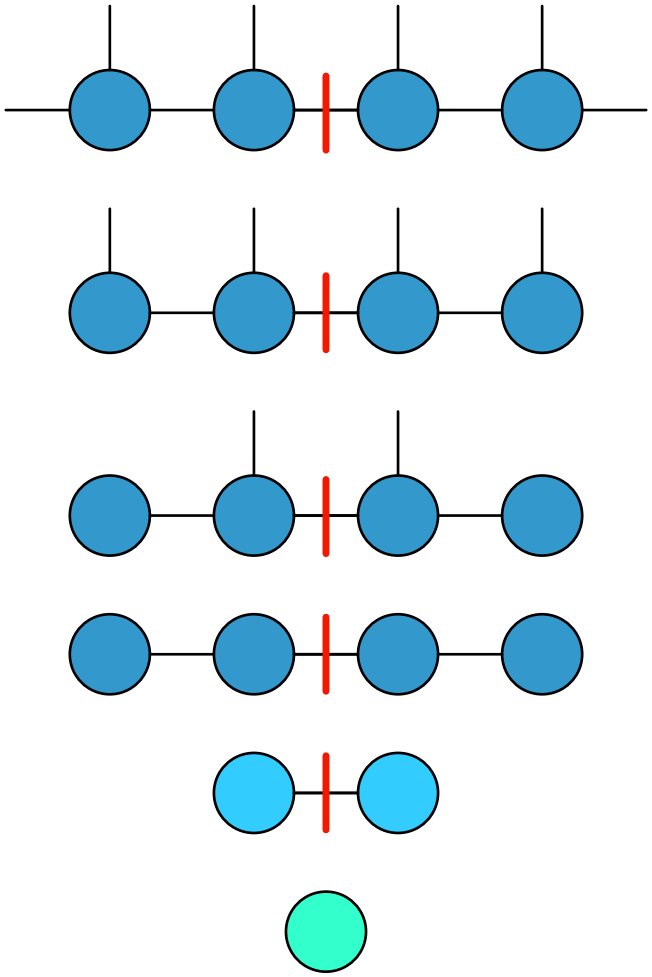

Figure 3.2: Process of parallel emission calculation with two threads. The steps are shown from top to bottom with the red line indicating the separation between the threads. The evidence is absorbed (steps 1-4), followed by the contraction of two carriages in each thread (step 5). The final emission probability is calculated by contracting the results from both threads (step 6).

3.3 Learning Decomposed HMMs

The main goal of internally decomposing the emission probabilities of HMMs is to reduce the memory demand compared to a high-rank tensor representation and offer an alternative to the representation with individual emission matrices. Besides efficient inference discussed in the previous sections, any new representation has to offer efficient learning algorithms. Without a way of learning an HMM, that offers the same advantages concerning runtime and memory efficiency as the inference based on the resulting

representation, no model can be produced to perform inference. Just like the forward algorithm is used as a representative for inference algorithms in the previous sections, the Baum-Welch algorithm is used as an example of learning algorithms in the following section.

3.3.1 Baum-Welch Algorithm with Tensor Train Operations

As described in Section 2.3.2, the Baum-Welch algorithm works by updating each emission probability. The naive approach to implementing the Baum-Welch algorithm on a tensor train representation is to reassemble the high-rank emission tensor from the emission tensor train, update the individual emission probabilities with Equation (2.13), and then decompose the tensor back into a tensor train representation as described in Section 2.1.3. While this naive approach is a way of learning a model that has the emission probabilities represented by a tensor train, it directly contradicts the goal of reducing the memory demand of the emission data, since the full high-rank tensor has to be stored temporarily. Besides the high memory complexity of the intermediate results, the contraction of the full emission tensor train to build the original tensor and the following decomposition result in additional runtime on top of the calculations that have to be performed for the update of each emission probability.

A better approach would be to implement the Baum-Welch algorithm using the tensor train operations described in Section 2.1.2, which would make it possible to update the emission probabilities without having to reconstruct and re-decompose the full emission tensor. To do so, a formula $\mathbf{B}^* = f(\mathbf{A}, \mathbf{B}, \boldsymbol{\pi}, \{o^{(0:T)}\})$ for the calculation of the new emission tensor train from the old parameters of the HMM and the given set of training evidence sequences would have to be found. To find the dependence of the new emission tensor train $\mathbf{B}^*$ on the old model parameters, the formulas of the Baum-Welch algorithm described in Section 2.3.2 have to be transferred to matrix operations, which can then be implemented as tensor train operations.

Starting with the formula for updating the individual emission probabilities:

$$\mathbf{B}^{*h}_{\ o} = \frac{\sum_{t=0}^{T} 1_{o^{(t)}=o}\ \gamma_h^{(t)}}{\sum_{t=0}^{T} \gamma_h^{(t)}}, \tag{3.4}$$

the indices h and o can be removed to convert the formula from calculating individual matrix elements to updating the complete matrix using matrix operations. Removing the index h converts the scalar $\gamma_h^{(t)}$ to a vector $\boldsymbol{\gamma}^{(t)} = (\gamma_1^{(t)}, \gamma_2^{(t)}, ..., \gamma_{n_h}^{(t)})$, with $n_h = |\mathscr{H}|$ being the number of hidden states. Removing the index o leads to the value $1_{o^{(t)}=o}$ from the indicator function becoming a vector $\mathbf{1}_{o^{(t)}} = (1_{o^{(t)}=1}, 1_{o^{(t)}=2}, ..., 1_{o^{(t)}=n_o})$, with $n_o = |\mathscr{O}|$ being the number of observable states. In the resulting formula for the matrix $\mathbf{B}^*$ instead of the individual elements $\mathbf{B}^{*h}_{\ o}$, the multiplication $1_{o^{(t)}=o}\ \gamma_h^{(t)}$ is replaced by

the tensor product between $\mathbf{1}_{o^{(t)}}$ and $\boldsymbol{\gamma}^{(t)}$:

$$\mathbf{1}_{o^{(t)}} \otimes \boldsymbol{\gamma}^{(t)} = \begin{pmatrix} 1_{o^{(t)}=1}\gamma_1^{(t)} & 1_{o^{(t)}=1}\gamma_2^{(t)} & \cdots & 1_{o^{(t)}=1}\gamma_{n_h}^{(t)} \\ 1_{o^{(t)}=2}\gamma_1^{(t)} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 1_{o^{(t)}=n_o}\gamma_1^{(t)} & 1_{o^{(t)}=n_o}\gamma_2^{(t)} & \cdots & 1_{o^{(t)}=n_o}\gamma_{n_h}^{(t)} \end{pmatrix}. \tag{3.5}$$

In the following, the formula for $\boldsymbol{\gamma}^{(t)}$ could be converted to matrix operations and then inserted into Equation (3.5), which would result in a formula for $\mathbf{B}^*$ describing the relation to the old model parameters and training data. However, there is already a problem with expressing Equation (3.5) with tensor train operations. The matrix shown in Equation (3.5) depends on the values of $\mathbf{1}_{o^{(t)}}$, which is by definition dependent on the evidence sequences $\{o^{(0:T)}\}$ that are given as training data. If the Baum-Welch algorithm was restructured to use tensor train operations, all involved mathematical structures would have to be represented as tensor trains. Since at least some of the structures would have to incorporate $\mathbf{1}_{o^{(t)}}$, they would depend on the training data. This dependency means that a fixed tensor train representation cannot be found. Building a structure like the matrix in Equation (3.5) and decomposing it would require storing a tensor of the rank and dimensions of the emission tensor itself and would therefore contradict the intention of representing the emission probabilities as a tensor train in the same way as the naive approach described in the previous section.

This problem only occurs when trying to find a fixed formula for updating all emission probabilities simultaneously, which is the equivalent of implementing the Baum-Welch algorithm using tensor train operations. When updating each probability individually, the values for $1_{o^{(t)}=o}$ can be calculated for each training sequence as it is done in Equation (3.4). When trying to find a uniform formula $\mathbf{B}^* = f(\mathbf{A}, \mathbf{B}, \boldsymbol{\pi}, \{o^{(0:T)}\})$, a tensor train decomposition of $\mathbf{1}_{o^{(t)}}$ has to be known, which can only be acquired by performing a decomposition with each update step, since the values of $\mathbf{1}_{o^{(t)}}$, and therefore the corresponding tensor train representation, depend on the evidence sequence that is Incorporated in the update process. Due to the dependency of the structures involved in the update process on the evidence, using the structure of the tensor train representation in the learning process is not as trivial as implementing each operation of the Baum-Welch algorithm using tensor train operations.

An alternative implementation of the Baum-Welch algorithm on an emission tensor train could be using the formulas described in Section 2.3.2, by calculating the emission probabilities from the tensor train as described for the inference algorithms in Section 3.2. Just like with the inference algorithms, this approach would add the runtime of acquiring the emission probabilities from the tensor train to the runtime of the Baum-Welch algorithm itself, which could be partially compensated by the same parallelisation approaches described in Section 3.2.1 and Section 3.2.2. The main problem would however be, that the updated emission probabilities would have to be inserted into the tensor

train one by one to prevent the construction of a new high-rank emission tensor, like in the naive approach. Since each element from a tensor contributes in highly complex ways to many elements of the corresponding tensor train, it is not possible to change the values of the original tensor after the decomposition. One way of updating elements after the decomposition would be to construct an update tensor train, which could be combined with the emission tensor train with a tensor train operation like the element-wise multiplication. However, acquiring such an update tensor train would once again require the decomposition, and therefore temporary storage, of a tensor of the same size as the original emission tensor. The size of the temporary update tensor does not depend on the number of updated emission probabilities and would therefore once again contradict the goal of making the representation of the emission probabilities more memory efficient by representing them as a tensor train.

3.3.2 Symmetric Property of Learned Tensor Trains

Besides the problems with adapting the Baum-Welch algorithm to the structure of a decomposed emission tensor mentioned in the previous section, learning algorithms can also interfere with the odeco symmetric property of the decomposition described in Section 2.1.3. This interference can be illustrated using a simple example, considering a symmetric emission matrix like:

$$\begin{pmatrix} 0.1 & 0.5 & 0.4 \\ 0.5 & 0.2 & 0.3 \\ 0.4 & 0.3 & 0.3 \end{pmatrix}.$$

Given an evidence sequence as training data, the updated emission probabilities could result in a new emission matrix that is still symmetric, for example by only changing values on the diagonal. However, such a behaviour is not guaranteed by learning algorithms. Depending on the training data, the symmetry of the emission matrix could easily be destroyed, even by changing a single emission probability:

$$\begin{pmatrix} 0.1 & 0.5 & 0.4 \\ 0.1 & 0.2 & 0.3 \\ 0.4 & 0.3 & 0.3 \end{pmatrix}.$$

While the odeco property can be omitted in certain scenarios by using a whitening process and still finding an orthogonal decomposition (Halaseh et al., 2022, p.6-7), the symmetry of the decomposed tensor is critical to every method of orthogonal decomposition. The fact that learning algorithms can alter the symmetric property of the emission probabilities and therefore the ability to find a tensor train representation using orthogonal decomposition means that the orthogonal decomposability has to be retested by attempting an orthogonal decomposition. This retesting could only be avoided reliably by performing the updates directly on the tensor train representation. When working with the tensor train representation, the emission tensor train could still lose its odeco symmetric property, but this might not become a problem if those properties were only used for finding an initial decomposition. Constraining learning algorithms to preserve

the symmetry of the tensors could be done by first determining the tensor element which has to be updated according to the Baum-Welch algorithm. Instead of updating just this element, all elements that are denoted by a permutation of the updated element's indices have to be changed in the same way. In the example above not only the element with the indices $(2, 1)$ but also the element with the indices $(1, 2)$ would have to be updated:

$$\begin{pmatrix} 0.1 & 0.1 & 0.4 \\ 0.1 & 0.2 & 0.3 \\ 0.4 & 0.3 & 0.3 \end{pmatrix}.$$

This example immediately shows, that this constraint would lead to the change of unrelated probabilities and would therefore influence how well the model is adapted to the training evidence. If the tensor is a symmetric carriage in an emission tensor train, the effect on unrelated probabilities would be even bigger, because the elements of the carriage contribute to many emission probabilities from the represented model. The more probabilities are altered which are not supposed to be updated by the Baum-Welch algorithm, the further the model can potentially deviate from the training data it is supposed to be adjusted to. Updating unrelated emission probabilities could even lead to an evidence sequence being less likely to originate from the learned model after the update, which is the opposite of the Baum-Welch algorithm's purpose.

3.4 Interim Conlusion

The first part of this chapter demonstrates that it is possible to use internal tensor train decomposition for the representation of HMMs. The new representation reduces the memory demand of an HMM compared to a representation using a high-rank tensor to store the model's emission probabilities. The main downside of the new representation is the additional runtime necessary to access emission probabilities. The additional runtime increases the runtime complexity of algorithms like inference algorithms, which frequently access emission probabilities.

The parallelisation approaches to inference algorithms presented in the second part of this chapter have the potential to reduce the additional runtime introduced by the new representation. Because efficient inference is the main focus of this thesis, the real-world benefit of the parallelisation approaches is tested by recording runtime data. In the following, Chapter 4 describes the implementation and Chapter 5 presents the acquired runtime results.

The discussion of options for learning shows that learning algorithms like the Baum-Welch algorithm cannot be adjusted to the new representation as easily as inference algorithms. This problem is taken into consideration together with the findings from the runtime measurements in the discussion in Chapter 6.

4 Implementation

To be able to evaluate the real-world advantages of the parallelisation approaches described in Section 3.2, the methods are implemented using C++. Due to the similarity between all inference algorithms on HMMs in terms of working with emission probabilities, the forward algorithm is used as a representative for all inference algorithms in the implementation.

4.1 Model Generation and Storage

Instead of decomposing tensors into odeco and symmetric tensor train representations, the tensor trains are generated by going along the definition of an odeco and symmetric tensor in Equation (2.5). Generating data makes it possible to create emission probabilities for performance testing that are guaranteed to fulfil the odeco symmetric definition and eliminate the need to discard tensors, for which an odeco and symmetric tensor train representation cannot be found. An orthogonal matrix $\mathbf{Q}$ is produced by performing a householder rank-revealing QR decomposition of a randomly generated matrix, using the Eigen linear algebra library (Guennebaud et al., 2010). A QR decomposition factorises a matrix $\mathbf{A}$ into a product $\mathbf{A} = \mathbf{Q} \cdot \mathbf{R}$ of an upper triangular matrix $\mathbf{R}$ and an orthonormal matrix $\mathbf{Q}$. The column vectors from the orthonormal matrix $\mathbf{Q}$ are then used as the set of mutually orthogonal vectors $\mathbf{a}_i$ to form a rank-3 tensor as a carriage with the dimensions specified for the generated model. This means all bond dimensions are identical to the dimensions of the free indices for simplicity. All tensor representations and tensor operations are using the ITensor software library for tensor network calculations (Fishman et al., 2022a). According to the rank r specified for the final model, $r-2$ individual carriages are generated and contracted over shared indices to form an emission tensor train. ITensors MPS class is used for the representation of tensor trains as it is compatible with the utilised tensors and provides a brought range of methods for modifying and working with the emission tensor train. Each element in the carriages is replaced by its absolute value to shift the value range from the values initially randomly sampled from [-1, 1], to a value range of [0, 1]. Therefore the values have the correct value range, however, because they are randomly generated and do not add up to 1 for all elements under an index of the hidden variable, they are not actual probabilities. As the forward algorithm does not rely on working with actual probabilities and the performance of the algorithm is unlikely to be influenced as long as the processed values are in the correct value range, this does not impact the runtime measurements discussed in Chapter 5.

To obtain the high-rank tensor represented by the generated emission tensor train, all carriages are contracted over the bond indices. Both the full emission tensor and the emission tensor train are stored in the `HMM` wrapper class. Besides the two different representations of the emission probabilities, this class contains the dimension and number of both hidden and observable variables. To reduce memory usage, the high-rank tensor representation is only calculated from the emission tensor train if it is explicitly needed for calculations. In addition to that, the expected memory demand of the high-rank tensor representation is calculated in advance of a model's generation and the high-rank representation is only generated if it does not exceed the memory limit of the machine at hand. This limitation is supposed to prevent the usage of swap memory during the runtime testing described in Chapter 5, as swap usage would drastically vary the memory access times and therefore the access time of emission tensor elements.

4.2 Forward Algorithm Implementation

The implementation is structured into three parts. The `generate_symmetric` script contains methods for generating arbitrary odeco symmetric tensors and tensor trains as well as methods for the contraction of tensor trains. The `hmm` script is used to generate HMM representations using the methods from `generate_symmetric`. `hmm` also contains methods for the generation of observations, the prediction of models memory demand, the absorption of evidence into emission tensor trains, and the method for acquiring an emission probability from a tensor train representation with different parallelisation parameters. The script `forward_algorithm` uses the methods in `hmm` to generate models and evidence sequences of various ranks and dimensions and contains several implementations of the forward algorithm. The script is also used to measure the runtime of the forward algorithm with different parameters, which is the basis for the data analysed in Chapter 5.

The basis of the implemented forward algorithm is the recursive algorithm described in Section 2.3.1. The implementation calculates the probability of all hidden states at the end of a given evidence sequence for a given model. The results for the computed forward messages are normalised after each recursion step. This normalisation prevents the forward messages from converging to 0 when calculating long evidence sequences, which is caused by the pseudo-probability emission probabilities described above. The way the algorithm works can be influenced by two parameters. The first parameter specifies the representation of the emission probabilities which is to be used, meaning either a high-rank tensor or an odeco symmetric tensor train. The second parameter specifies which parallelisation options described in Section 3.2 are applied. If the parallelisation option `no_parallel` is given, the forward algorithm either takes the emission probabilities needed for the calculations directly from the high-rank tensor or interrupts the algorithm to compute the needed emission probability from the tensor train representation. Using the emission tensor representation is only possible in the `no_parallel` mode.

When using the tensor train representation of the emission probabilities, the parallelisation options `parallel_evidence` and `parallel_contraction` can be chosen, which coincide with the approaches described in Section 3.2.1 and Section 3.2.2. With the option `parallel_evidence`, the emission probabilities needed for the calculations of the forward algorithms are pre-calculated in the specified number of threads. The forward algorithm receives an array with all necessary emission probabilities and can access them directly without any interruption. With the option `parallel_contraction`, the forward algorithm is interrupted when an emission probability is needed, but the calculation of the emission probability is performed in two parallel threads as described in Section 3.2.2. In contrast to the adjustable thread count with the `parallel_evidence` options, the thread count is fixed to two parallel threads with the `parallel_contraction` option, because the length of the tensor train does not exceed eight carriages in the performed runtime tests. The additional option `both_parallel` combines both approaches by pre-calculating the necessary emission probabilities with two parallel threads for the computation done on each probability.

To ensure the correctness of the different implementations of the forward algorithm, a test method can calculate the probabilities of all hidden states for all possible combinations of emission representation and parallelisation options, using a single model and evidence sequence for all settings. The consistency of the different implementations is then tested by comparing the results for all settings and making sure there are no significant deviations between the calculated probabilities.

5 Evaluation

To determine the real-world efficacy of the parallelisation approaches described in Section 3.2.1 and Section 3.2.2, runtime data is recorded using the implementation of the forward algorithm from Chapter 4. First, the setup and test methodology are described in Section 5.1. The following sections each deal with one test, comparing different implementations of the forward algorithm with each other. The runtimes presented in Section 5.2 establish a baseline for the performance of the forward algorithm on an HMM with a high-rank emission tensor. The test described in Section 5.3 aims at determining the additional runtime of the non-optimised implementation of the forward algorithm when using an emission tensor train instead of an emission tensor. Section 5.4 and Section 5.5 compare the runtimes of the optimisations described in Section 3.2 with the non-optimised forward algorithm on both an emission tensor and an emission tensor train. The last test, described in Section 5.6, investigates the combination of both parallelisation approaches by comparing it against the individual parallelisations.

5.1 Test Setup

The tests are run on a 2020 Mac mini with an M1 processor, featuring four high-performance and four high-efficiency cores and 16 GB of RAM. The memory limit for the models used in testing is 12 GB to prevent swap usage. Only 12 GB instead of the full 16 GB are used due to the consideration of memory use by background processes. All models used in testing are generated according to the description in Chapter 4. When models exceed 75% of the memory limit, the program pauses for one second after the deallocation of the model's memory to prevent swap usage when immediately generating a new model.

To record runtime data for a wide range of HMMs, the rank of the emission tensor varies between 4 and 10 while the uniform dimension varies between 2 and 100. All possible combinations of rank and dimension are used for each test, as long as the resulting model does not exceed the memory limit. If not clearly stated otherwise, all runs use an evidence sequence with a length of 400 time steps. For each combination of rank and dimension, each test is run ten times. For each run, a new model and evidence sequence are generated according to the model parameters, to prevent a high dependency of the measured runtime data on a small number of HMMs chosen by hand. The measured runtime only includes the execution of the forward algorithm, not the creation or deallocation of the model data. The ten runtimes per setup are averaged using the arithmetic mean. The uncertainties given for the runtime data originate from the devi-

ation of measured runtimes from the arithmetic mean and are therefore uncertainties of measurement and not deviations of the calculated probabilities from the correct results. The uncertainty of the arithmetic mean as well as all uncertainties propagated during data analysis are calculated using the Gaussian uncertainty propagation according to the "Guide to the expression of uncertainty in measurement" (Taylor and Kuyatt, 1995).

5.2 Sequential High Rank Emission Tensor

The results of the runtime measurement of the forward algorithm with HMMs using a high-rank emission tensor can be seen in Figure 5.1. The diagram shows a colour map of the runtime with a field for each combination of rank and dimension of the HMMs emission tensor.

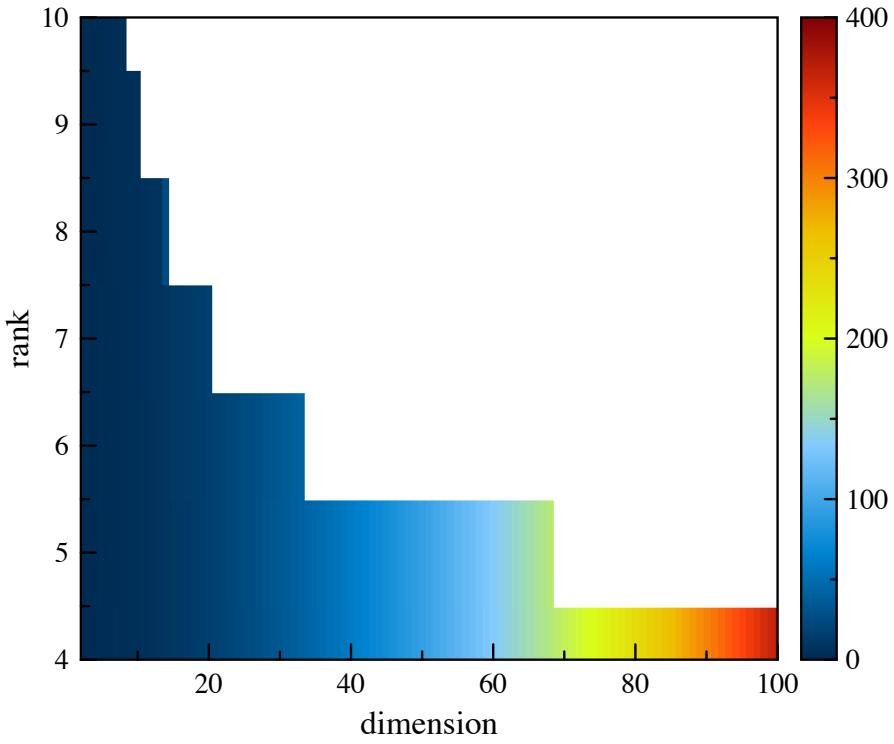

Figure 5.1: Runtime (in ms) of the forward algorithm on an HMM with an emission tensor. The tensor's rank is identical to the total number of variables, the dimension is identical across all variables.

This kind of visualisation gives an overview of all measured runtimes for the sequential implementation of the forward algorithm, with the emission probabilities being represented as a high-rank tensor. It can be seen that only about a third of all possible combinations of rank and dimension have been executed. All other combinations, which

are displayed white in the graphic, exceed the memory limit of 12 GB and are therefore not executed.

It also becomes apparent from Figure 5.1, that the runtime is only dependent on the dimension and is therefore identical for all ranks with a fixed dimension. This is expected because the forward algorithm has a runtime of $\mathcal{O}(T \cdot |\mathscr{H}|^2)$, as described in Section 2.3.1. In the implementation, the length of the evidence sequence T is fixed at 400 time steps, which leaves only the quadratic dependency of the runtime from the number of hidden states $|\mathscr{H}|$, which is the dimension of the emission tensor.

The maximum relative runtime uncertainty of all measurements is 65.4%, however, the mean relative uncertainty is only 2.7%. Higher relative uncertainties occur mostly in the runs with low dimensions and therefore very low runtimes of less than 10 ms. The higher uncertainty rates are most likely resulting from influences outside of the implementation of the forward algorithm like CPU scheduling or variance in memory access times. The full visualisation of all relative uncertainties from the measurements shown in Figure 5.1 can be seen in Figure A.1.

5.3 Sequential Emission Tensor Train

Figure 5.2 shows the results of the runtime measurements of the sequential forward algorithm on an HMM with an emission tensor train instead of an emission tensor. Due to the much lower memory demand, all combinations of dimension and rank are within the memory limits. In contrast to the implementation using an emission tensor, the algorithm can therefore be executed on all model variants.

The visualisation also shows that the runtime is not only dependent on the dimension, as it is when using an emission tensor, but also depends on the rank r, meaning the number of variables in the model. This additional dependency is due to the added runtime complexity resulting from the tensor train contraction, which is necessary to acquire the emission probabilities. As the contraction has a runtime of $\mathcal{O}(r \cdot b^2)$, a dependency on the rank r is added, while the dependency on the dimension, which is identical to the bond dimension $b = d$ in the implementation, is increased in its exponent. This added complexity results in much higher runtimes, which can be seen in Figure 5.3, which visualises the relative runtime difference between the implementation using an emission tensor from Figure 5.1 and the implementation using an emission tensor train from Figure 5.2. It shows that for the maximum dimensions that were executed using an emission tensor, the tensor train implementation takes about 80 times longer for one execution of the forward algorithm. This is with otherwise identical parameters of dimension, rank, and length of the evidence sequence. How much longer the implementation using an emission tensor train took depends on the rank as well as on the dimension of the emission tensor, once again due to the runtime complexity of the tensor train contraction.

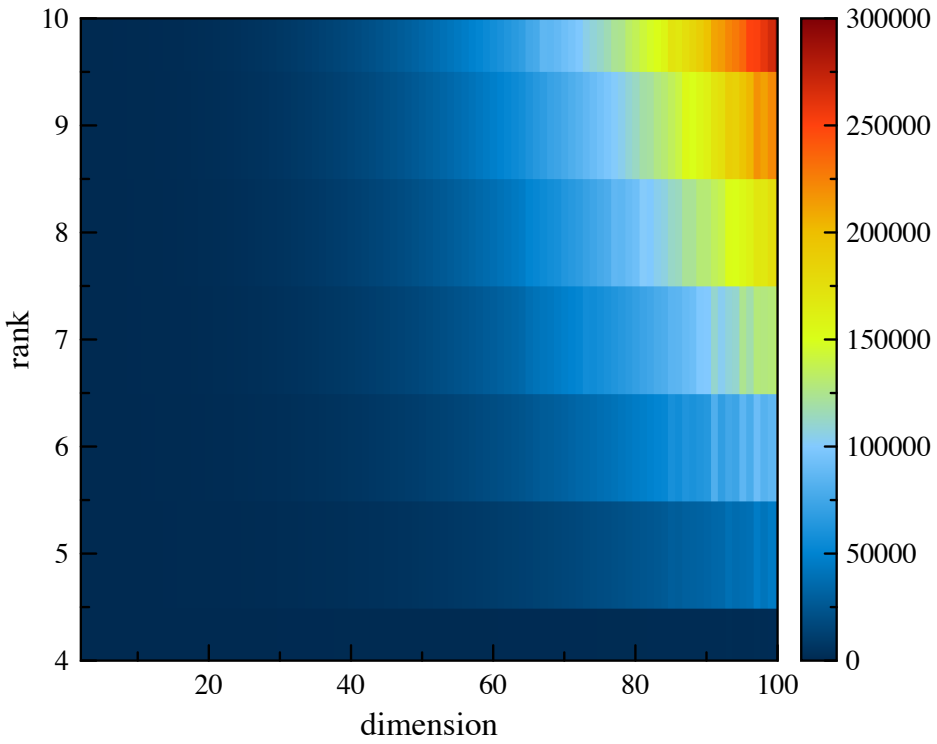

Figure 5.2: Runtime (in ms) of the sequential forward algorithm on an HMM with an emission tensor train.

Since the runtime of the actual forward algorithm is identical for both implementations, the runtime difference results solely from the tensor train contraction. Because of the dependency of the contraction's runtime on rank and dimension, the relative difference between the implementation gets bigger both with increasing rank and with increasing dimension, which can be seen in Figure 5.3. An exception from this increasing runtime difference is the runtime of models with an emission tensor of rank 4. As can be seen in the graphic, the runtime difference stays almost constant for all dimensions, with the implementation using a tensor train taking at most about 12 times longer than the implementation using an emission tensor. This behaviour is not expected from the runtimes of the tensor train contraction or the forward algorithm, since the runtimes should not differ explicitly for tensors of rank 4. Besides a bug in the implementation, this behaviour could result from an optimisation in the ITensor library, however, no such optimisation could be found in the ITensor documentation (Fishman et al., 2022b).

The mean relative uncertainty of the runtimes in Figure 5.2 is 1.2%, making the runtime data consistent overall. In contrast to the relative uncertainties in Figure A.1, the relative uncertainties for the runtimes in Figure 5.2 have more outliers in the high dimensions across all ranks, as can be seen in Figure A.2. Due to the high runtimes of 100 s and more per execution of the forward algorithm, these higher relative uncertainties cannot

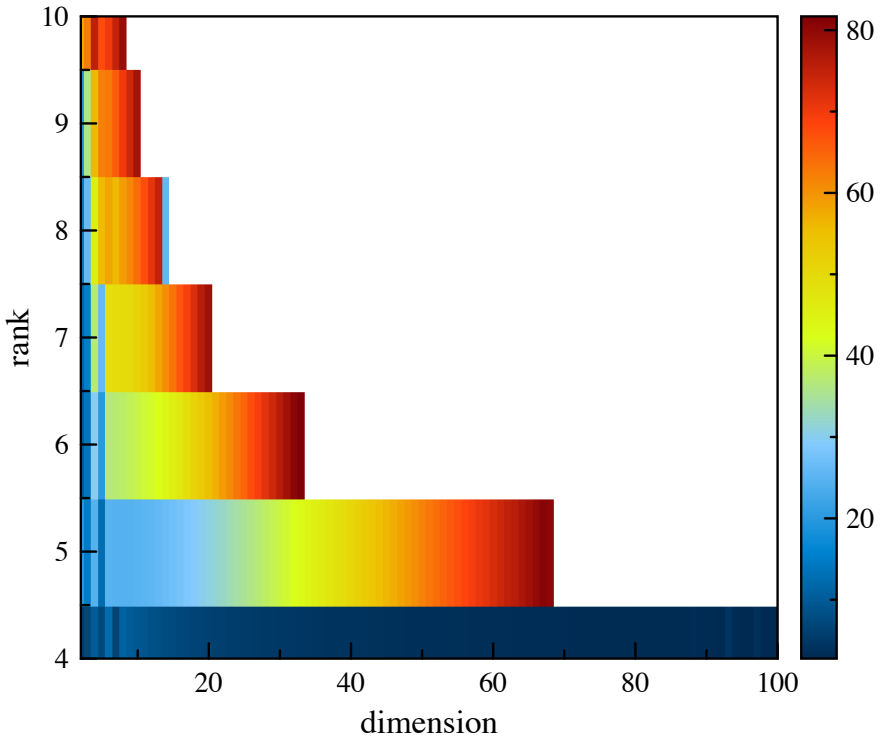

Figure 5.3: Relative runtime difference between the sequential forward algorithm running on an emission tensor and running on the corresponding emission tensor train.

be easily explained by outside influences causing small variations in runtime, since the relative effect on such high runtimes would be minimal. The behaviour is not consistent for all runs and only occurs for some, seemingly random, high dimensions at all ranks. An explanation could once again be implementation details, either in the implementation of the forward algorithm or in the implementation of the ITensor library.

5.4 Parallel Evidence Incorporation

The approach of parallelised evidence incorporation described in Section 3.2.1 is tested with 2, 4, 6, and 8 parallel threads. The relative runtime differences for the run with $n_{thr\_evi} = 2$ can be seen in Figure 5.4. The uncertainty of the relative runtime difference mostly follows the uncertainty of the runtime recorded for the unparallelised implementation using an emission tensor train, which can be seen by comparing Figure A.2 with Figure A.3.

The visualisation in Figure 5.4 shows, that the relative runtime difference between an unoptimised implementation using an emission tensor train and an implementation using an emission tensor train with parallelised evidence incorporation is almost identical

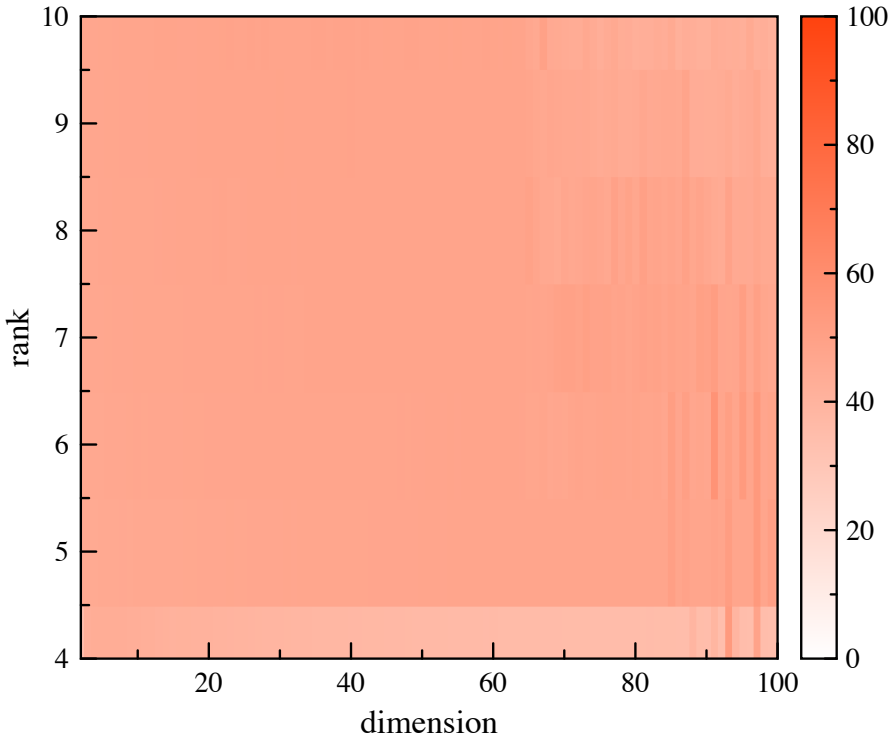

Figure 5.4: Relative runtime difference (in %) between the unoptimised forward algorithm running on an emission tensor train and running on an emission tensor train with two parallelised threads.

for all combinations of rank and dimension. This uniform relative runtime difference is expected since the approach divides the evidence sequence along the time axis and the runtime difference is therefore independent of the model's parameters.

The theoretical maximum improvement of the parallelisation is reciprocal in the number of threads as described by Equation (3.1). The mean relative runtime differences for all runs can be seen in Table 5.1. Because the runtimes for models of rank 4 differ in the same way described in the previous section, two separate mean relative runtime differences are shown for each run, one averaging over all ranks and the other only averaging over ranks 5 to 10, to exclude the effects of the unexpected behaviour in rank 4. The data shows that for $n_{thr\_evi} = 2$ the mean relative runtime difference is lower than the theoretical maximum of 50% by being about $(46.04 \pm 0.09)\%$ faster than the non-parallelised implementation on average, as can be seen in Figure 5.4. The mean relative runtime differences of the runs with a higher number of threads increase further, up to an improvement of $(78.30 \pm 0.03)\%$ lower runtimes when using $n_{thr\_evi} = 8$ threads. However, the relative runtime differences are always lower than the theoretical maximum. That the theoretical maximum is not reached is expected because the approach does not parallelise the calculation of the forward messages, which is part of the

measured runtimes. The parallelisation also introduces some overhead for the creation and termination of threads. Overhead and the runtime needed for calculating the forward messages can explain a constant difference between the measured runtimes and the theoretical maximum and are the reason for the theoretical maximum being unreachable in real-world implementations.

Table 5.1: Mean relative runtime difference (in %) between an unoptimised implementation using an emission tensor train and an implementation using parallel evidence incorporation. Percentages for different numbers of threads for the parallelisation of evidence calculations and the theoretical limit of relative runtime difference as a comparison.

	$n_{thr\_evi} = 2$	$n_{thr\_evi} = 4$	$n_{thr\_evi} = 6$	$n_{thr\_evi} = 8$
all ranks	46.04 ± 0.09	69.37 ± 0.06	72.89 ± 0.05	75.89 ± 0.05
ignoring rank 4	47.37 ± 0.07	71.19 ± 0.04	75.11 ± 0.04	78.30 ± 0.03
theoretical max	50.00	75.00	83.33	87.50

However, the data in Table 5.1 shows a growing difference between the theoretical maximum and the measured relative runtime differences with an increasing number of threads. This behaviour is expected because the higher number of threads for parallelising an evidence sequence of constant length leads to fewer evidence values being calculated in each thread. In the executions of the forward algorithm shown in Table 5.1, the evidence sequence had a fixed length of $T = 400$. Therefore each of the $n_{thr\_evi} = 2$ threads in the first run is working on 200 evidence values, while each thread for $n_{thr\_evi} = 8$ only works on 50 evidence values, which increases the effect of the overhead on the overall runtime measured. Besides this increasing influence of overhead, the architecture of the M1 processor used for the runtime testing could increase the effect due to the dissimilar cores. The processor has four high-performance and four high-efficiency cores, which coincides with the observation that the runtime improvement scales better for the usage of up to $n_{thr\_evi} = 4$ and worse for the usage of more threads.

A comparison between the parallelised evidence incorporation and the sequential implementation of the forward algorithm on an emission tensor can be seen in Figure 5.5. The relative runtime difference is calculated using the run with $n_{thr\_evi} = 8$, since it has the best overall runtime improvements from Table 5.1.

Comparing the data from Figure 5.5 with the similar comparison between the implementation using an emission tensor and the one using an unparallelised emission tensor train from Figure 5.3, it can be seen that the maximum runtime for the parallelised implementation is about 17 times longer compared to using the full emission tensor. This is a significant reduction from the approximately 80 times longer runtimes of the unparallelised implementation compared to the usage of the emission tensor. However, the same incline of relative runtime difference with growing rank and dimension can be

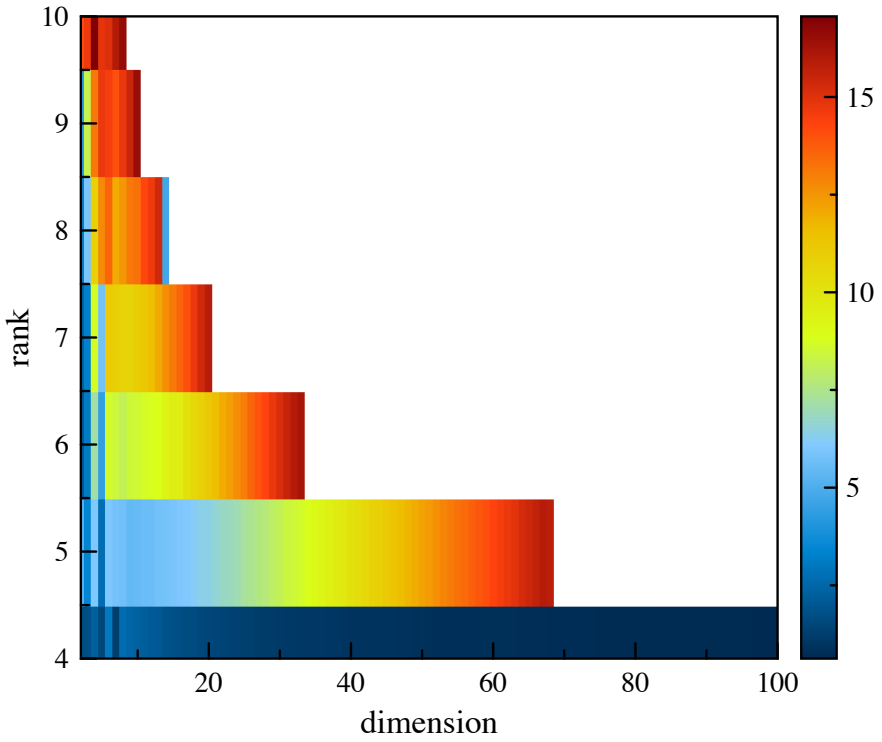

Figure 5.5: Relative runtime difference between the forward algorithm running on an emission tensor and running on the corresponding emission tensor train with parallelised evidence incorporation in $n_{thr\_evi} = 8$ parallel threads.

seen in Figure 5.5 that can be observed in Figure 5.3. This incline results from the parallelised evidence incorporation creating a runtime improvement that is determined by the number of threads used and does not increase with higher dimensions and ranks like the runtime itself. Therefore the parallelisation has a diminishing relative return when used on a model with fixed parameters and running on hardware with real-world limits, which does not allow for keeping up the number of threads used with the size of the model growing.

5.5 Parallel Tensor Train Contraction

Because only emission tensors of up to rank $r = 10$ are used in the runtime measurements, the parallelised tensor train contraction is implemented with a fixed number of $n_{thr\_con} = 2$ threads. The runtimes measured for all combinations of ranks and dimensions are visualised in Figure 5.6.

It can be seen that the runtimes for ranks 5 and 6, 7 and 8, as well as 9 and 10 are nearly identical for the same dimensions. This similarity is a result of the division of

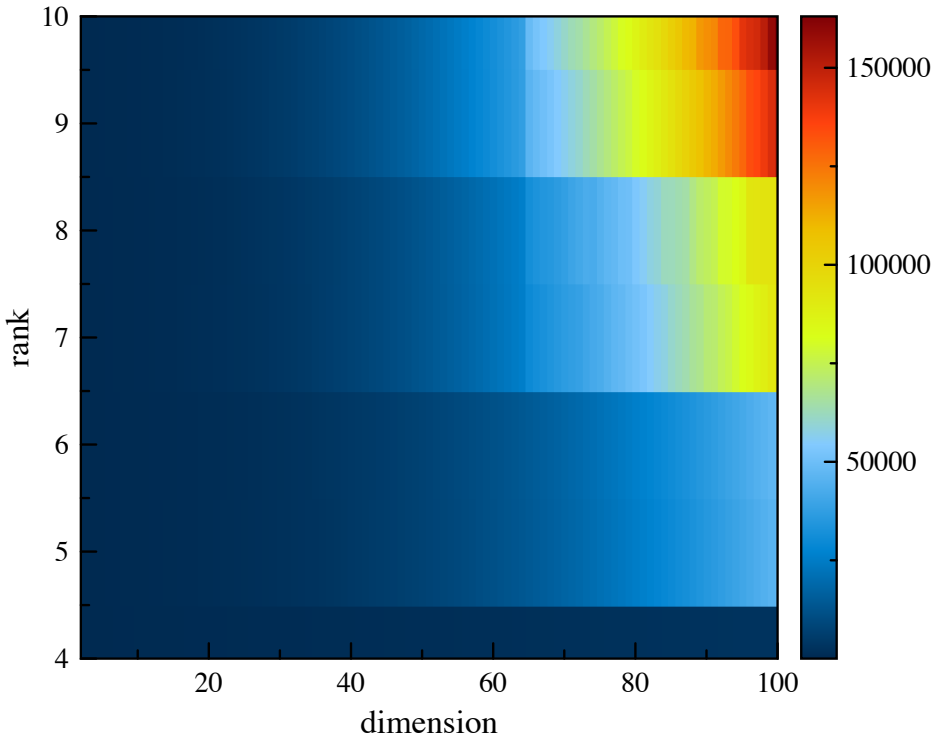

Figure 5.6: Runtime (in ms) of the forward algorithm performing parallelised contraction with 2 threads on an HMM with an emission tensor train.

the emission tensor train into two parallel threads. For uneven ranks like $r = 7$, the carriages of the train cannot be split up evenly, which leads to the threads containing a different number of carriages. For $r = 7$, two carriages are processed in one and the remaining three carriages are processed in the other thread. Assuming that both threads take about equal runtime per carriage, the thread that only contains two threads finishes first, however, an emission probability can only be calculated after both threads have terminated, meaning the total runtime is always determined by the thread containing the most carriages. This dependence on the thread with the most carriages is the reason for ranks 7 and 8 having the same runtime since in both executions the threads with the most carriages contain three carriages.

The identical runtimes for emission tensor trains with different ranks lead to the executions on tensor trains with an even rank having a higher relative runtime difference to the unparallelised algorithm than the corresponding tensor trains of uneven rank, which can be seen in Figure 5.7. The graphic shows the relative runtime difference in percent as a temperature-style colour map, similar to Figure 5.4. Red values indicate a percentage runtime improvement over the non-parallelised implementation, white values indicate a similar runtime, and blue values indicate a longer runtime. The mean relative runtime differences per rank are listed in Table 5.2. The relative differences are averaged over all

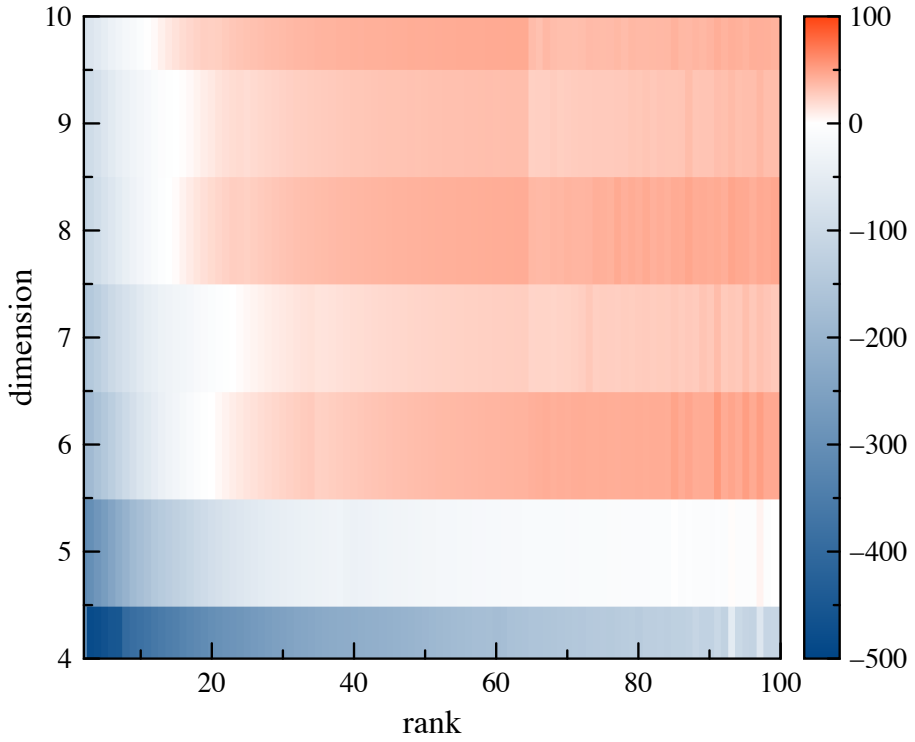

Figure 5.7: Relative runtime difference (in %) between the unoptimised forward algorithm on an HMM using an emission tensor train and using an emission tensor train with parallelised contraction in $n_{thr\_con} = 2$ threads.

dimensions as well as over dimensions 70 to 80. The second mean is an approximation of the maximum possible relative runtime difference. The dimensions 70 to 80 are chosen instead of the dimensions 90 to 100, to minimise the relative uncertainty of the values, which was much higher for the dimensions 90 to 100 due to a higher density of outlier uncertainties in the high dimensions, which are also observed with the measurements analysed in Section 5.3 and Section 5.4. The mean uncertainty of all relative runtime differences, however, was low at 1.53%.

It can be observed, that the runtimes for ranks 4 and 5 are higher than the runtimes of the forward algorithm running on a non-parallelised tensor train. This higher runtime is expected due to the low number of carriages with these ranks. For a rank of $r = 4$, the tensor train has only two carriages, which means that one carriage is processed per thread. While the evidence absorption is now done in parallel, both threads have to return the one carriage they processed before they can be contracted, which means that no part of the contraction is carried out in parallel due to the low number of carriages. The overhead created through the creation and termination of the threads adds more runtime than is saved by carrying out the evidence absorption in parallel.

Table 5.2: Mean relative runtime difference (in %) between an unoptimised implementation using an emission tensor train and an implementation using an emission tensor train with parallelised tensor train contraction in two threads. Runtime differences are given for averaging over all dimensions and only over dimensions 70 to 80 for all tested ranks.

	all dimensions	dimension 70 to 80
$r = 4$	-216.48 ± 456.60	-143.02 ± 0.13
$r = 5$	-53.15 ± 14.65	-11.07 ± 0.02
$r = 6$	14.48 ± 5.48	43.20 ± 0.02
$r = 7$	5.03 ± 7.25	25.84 ± 15.02
$r = 8$	25.85 ± 2.97	41.46 ± 10.27
$r = 9$	18.12 ± 2.51	27.97 ± 2.32
$r = 10$	29.33 ± 2.55	35.71 ± 1.81

For a rank of $r = 6$, the emission tensor train has a total of four carriages, which means that two carriages can be contracted in each thread. This parallelisation of parts of the contraction leads to a runtime improvement relative to the non-parallelised implementation for ranks 6 and higher, as can be seen in Table 5.2. The influence of the overhead is higher for lower total runtimes and the relative runtime difference is therefore lower for lower dimensions within the same rank. For most ranks the relative runtime difference is only positive for dimensions of $d \gtrsim 25$, which leads to low mean relative runtime differences when averaging over all dimensions within a rank. The mean relative runtime differences are higher when the mean is only calculated for high dimensions, like dimensions 70 to 80 in Table 5.2.

According to Equation (3.2), the theoretical maximum of the relative runtime difference is 50% for $n_{thr\_con} = 2$. Table 5.2 shows, that even for higher ranks the parallelised tensor train contraction is further away from this theoretical maximum than the parallelised evidence incorporation from Table 5.1. The uneven ranks do not exceed 30% shorter runtimes due to the influence of the uneven split of the tensor train described above. The even ranks have a maximum relative runtime difference of $(43.20 \pm 0.02)\%$, which is lower than the maximum of $(47.37 \pm 0.07)\%$ from the parallel evidence incorporation with $n_{thr\_evi} = 2$ threads. The scalability of the parallelisation of the tensor contraction could not be investigated, since the impact of overhead is already high for two threads with a maximum rank of $r = 10$ and would be further increased by a higher number of threads.

5.6 Combined Parallelisations

As pointed out in Section 3.2.2, the two parallelisation attempts can be combined because they focus on parallelising different aspects of the inference calculations. In the

runtime measurements, a fixed number of $n_{thr\_con} = 2$ is combined with 2, 4, 6 and 8 threads for n_{thr_evi}.

Table 5.3: Mean relative runtime difference (in %) between an implementation using an unparallelised emission tensor train and an implementation using an emission tensor train with both parallelisation options enabled. Runtime differences are averaged over all dimensions for different ranks and number of threads in evidence parallelisation.

	$n_{thr\_evi} = 2$	$n_{thr\_evi} = 4$	$n_{thr\_evi} = 6$	$n_{thr\_evi} = 8$
$r = 4$	-89.87 ± 222.38	-56.66 ± 230.79	-67.71 ± 389.10	-72.78 ± 579.83
$r = 5$	15.70 ± 9.12	47.87 ± 13.80	48.41 ± 32.01	49.62 ± 50.96
$r = 6$	52.29 ± 4.37	60.38 ± 9.40	61.39 ± 16.87	60.62 ± 14.36
$r = 7$	47.60 ± 3.32	62.09 ± 7.94	64.96 ± 6.16	65.07 ± 7.78
$r = 8$	59.29 ± 1.78	66.40 ± 4.08	68.14 ± 3.12	68.23 ± 3.37
$r = 9$	54.36 ± 1.79	67.11 ± 2.48	69.69 ± 2.71	69.51 ± 1.84
$r = 10$	59.14 ± 1.65	68.71 ± 2.30	70.98 ± 1.18	70.85 ± 1.00

The relative runtime difference for $n_{thr\_con} = n_{thr\_evi} = 2$ can be seen in Figure 5.8. The visualisation shows a combination of the characteristics from Figure 5.4 and Figure 5.7. The relative runtime differences are more uniform than in Figure 5.7 but still depend on the rank. The positive relative runtime differences start at rank $r = 5$ instead of $r = 6$ and at a dimension of $d \gtrsim 15$ instead of $d \gtrsim 25$.

The mean relative runtime differences for all combinations of n_{thr_con} and n_{thr_evi} are listed in Table 5.3, averaged per rank. For a rank of $r > 4$, the relative runtime differences for the combination of $n_{thr\_con} = n_{thr\_evi} = 2$ are higher than the implementation which is just using the parallelized evidence incorporation with $n_{thr\_evi} = 2$. For all other combinations of n_{thr_con} and n_{thr_evi}, the implementation that uses just the parallel evidence incorporation produces higher relative runtime differences with the same number of threads n_{thr_evi}.

In addition to that, the total number of threads used for the combined parallelisation is $n_{thr\_con} \cdot n_{thr\_evi}$, which means a better comparison for the combined approach $n_{thr\_con} = n_{thr\_evi} = 2$ is the parallelised evidence incorporation with $n_{thr\_evi} = 2 \cdot 2 = 4$ threads, which produces higher relative runtime results than any combination of the combined parallelisation. Therefore the approach using just the parallelised evidence incorporation should be preferred over the combined approach for any number of threads. Even for high ranks, for which the parallelised tensor train contraction results in high relative runtime differences, it is advantageous to use all available threads for parallelising the evidence incorporation. The almost identical relative runtime differences between the combined approaches using $n_{thr\_evi} = 6$ and $n_{thr\_evi} = 8$ threads can be explained by hardware limitations. The combined approach with $n_{thr\_evi} = 4$ uses up to 8 total

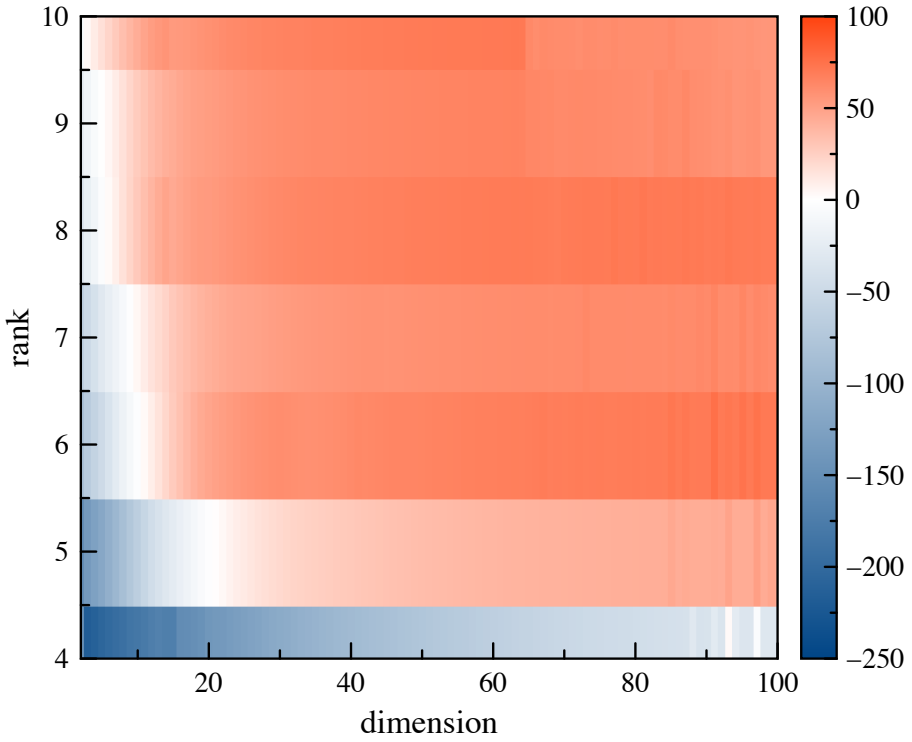

Figure 5.8: Relative runtime difference (in %) between the unoptimised forward algorithm on an HMM using an emission tensor train and using an emission tensor train with both parallelisation options with $n_{thr\_evi} = 2$ and $n_{thr\_con} = 2$.

threads, which means all combinations with $n_{thr\_evi} > 4$ create more threads than the system has CPU cores, forcing some sequential execution of threads. This sequential execution only adds overhead and cannot lower the runtime any further.

6 Discussion

The runtime measurements presented in Chapter 5 suggest that the parallelised evidence incorporation from Section 3.2.1 is the overall best approach for efficient inference on decomposed emission probabilities. The scalability with higher numbers of threads does not reach the theoretical maximum due to hardware limitations and the limited length of the evidence sequence, but the scalability is better than with the parallelised contraction or the combined parallelisations. In addition to that, the parallel evidence incorporation has a constant runtime improvement for all combinations of ranks and dimensions, while there is only a runtime improvement for certain ranks and dimensions with the parallelised tensor train contraction. Most importantly, the parallelised evidence incorporation results in higher runtime improvements, compared to the non-parallelised approach, than the parallelised contraction or the combined approach, even when using the same total number of threads. As pointed out in Section 5.6, all available threads should therefore be used to parallelise the evidence incorporation to obtain minimal runtimes.

It is important to notice that the conclusion above is based on the types of models that are used in the runtime measurements. A brought spectrum of combinations of dimensions and ranks is used, but for very specific types of models, the parallelised evidence incorporation might not be the best approach. Edge cases like models with many variables of high dimension and especially short evidence sequences might profit more from parallelisation of the tensor train contraction or the combined approach. Conversely, the parallelised evidence incorporation works best with long evidence sequences but has the advantage of working the same for models of any rank and dimension, as shown in Section 5.4.

All investigated approaches aim at reducing the additional runtime introduced by the tensor train decomposition. Therefore, the best possible outcome would be to match the runtimes of the implementation using an emission tensor. Even the parallelised evidence incorporation only reduces the runtime linearly, which means the runtime difference between the parallelised and the non-parallelised implementation gets bigger with increasing rank and dimension as shown in Section 5.4. Because neither the parallelised evidence incorporation nor the parallelised tensor train contraction or the combined approach can break the “curse of dimensionality”, the approaches can only keep the runtime on a tractable level for lower ranks and dimensions. The main reason for the parallelised inference not being able to keep the runtime on a consistent level is the fact that the inference algorithms can not make use of the tensor train structure or the odeco properties of the decomposition properly. As described in Section 3.2.2, the immediate

evidence absorption necessary for inference algorithms leaves limited options for working with the resulting structure of contracted matrices.

The considerations of using learning algorithms to make more efficient use of the emission tensor train structure in Section 3.3 show that it is not possible to adapt learning algorithms to the tensor train representation by converting each operation in the formulas into a tensor train operation, which means that learning would have to be done on a high-rank emission tensor. Due to the high memory demand of the tensor representation, the approach of internal decomposition is limited to niche use cases unless a more efficient learning approach is found. An example of such a use case would be the deployment of a model on devices with low computing power like smartphones. The model could be built, learned, and decomposed on a high-performance computer, and then deployed to the operating units in the decomposed representation. Even these use cases are however limited by the exponential memory growth of the emission tensor with increasing rank, which limits the approach to relatively small models because the exponential growth in memory demand cannot be compensated by increasing hardware capabilities.

The main cause for the structural challenges described above is the lack of a logical meaning of the orthogonal tensor train decomposition. Existing inference and learning algorithms, like the forward algorithm or the Baum-Welch algorithm, gain their efficiency by using the logical foundations of HMMs like the independence of variables and the Markov assumption, which are encoded in the structure of the model. There is no known equivalent for this logical meaning in the odeco decomposition of emission probabilities at this time. The lack of logical or structural meaning prevents the new approaches from using the odeco properties or the structure of the tensor train representation in the same way as existing algorithms do and makes the lower memory demand the only advantage of the representation through internal tensor train decomposition. Additionally, it is unclear how many real-world models have emission probabilities that could be represented through an odeco tensor train decomposition, because the number of tensors that can be represented this way is limited, as pointed out in Section 2.1.3. Therefore the main benefit of the odeco decomposition in comparison to other tensor train decompositions is the polynomial runtime, because the odeco properties are not used in the investigated approaches.

7 Conclusion

Two approaches to parallelising inference algorithms for making inference calculations on HMMs with an emission tensor train representation more efficient have been found. The parallelisation of evidence incorporation as well as using parallelised tensor train contraction on the emission representation. To be able to compare the real-world performance of the two separate approaches as well as the combination of both approaches to the theoretical performance, the forward algorithm was implemented on models using an internal tensor train representation of their emission probabilities. The implementation was used to gather runtime data on the different approaches for a variety of parameters like the model's rank and dimension as well as the number of threads used in the parallelisation. The runtimes of the parallelised implementations were compared to each other as well as to the runtimes of non-parallelised implementations to determine which parallelisation resulted in the best runtimes in real implementations.

The runtime measurements showed that the parallelised evidence incorporation performed best for the models used in the testing process, while the parallelised tensor train contraction might be advantageous for more specific models. Approaches for efficient inference on models with internal, orthogonal tensor train decomposition were therefore successfully conceived, implemented, and reviewed for their performance. The theoretical considerations on inference algorithms on tensor train decompositions showed, however, that parallelisation approaches cannot make use of the emission tensor train's structure efficiently, which limits their efficiency to compensate for the additional runtime introduced by the decomposition.

The odeco properties of the decomposition, which were supposed to be examined on their usability in inference calculations, could not be used. All implementations of the forward algorithm as well as the considerations on learning algorithms only relied on the general structure of tensor trains and could therefore be used with any other form of tensor train decomposition. This was attributed to the fact that there was no logical meaning found in the odeco tensor train decomposition in regards to the model's properties or structure. Additionally, it was shown that the odeco properties of a decomposed emission tensor train are not preserved when learning additional data with the decomposed model. It was therefore concluded, that the odeco decomposition has no advantage over other forms of tensor train decompositions in this use case other than its polynomial runtime.

Most parts of the implementation used to obtain the runtime measurements of the forward algorithm can be reused for further investigation on the internal emission de-

composition of HMMs. These foremost include the generation and representation of HMMs with decomposed emission probabilities as well as the methods used to process and modify those models. In future work, the implementation could be expanded to further test and improve the approaches described above. Additional tests could include the impact of a caching approach for often or recently used emission probabilities as well as additional tests of the already implemented algorithms on data from genuine HMMs used in real-world implementations. Testing on existing HMM datasets would require a transfer of the algorithms for orthogonal tensor train decomposition from the existing implementation in Python and MATLAB (Halaseh et al., 2020) to an implementation using C++. Testing the algorithms in the real world instead of on generated datasets would make it possible to determine the real-world performance of the approaches and could also provide an approximation of how many real-world HMMs have emission probabilities that can be decomposed using an odeco tensor train decomposition. Finding out how common the odeco properties are amongst datasets used for inference could definitively answer the question of whether the odeco tensor train decomposition applies to this use case. Another approach to making the conclusions from Chapter 6 more robust, would be to compare the runtime of the different parallelisation approaches directly to the runtime of a representation using multiple emission matrices. This comparison was not done as part of the conducted runtime tests because the element access times of a high-rank emission tensor and several separate emission matrices were assumed to be identical. Since the implementation aimed at evaluating the real-world impact of the parallelisation approaches, it would be consequential to verify this assumption.

Besides further investigations on the internal decomposition of HMMs, future work could include expanding the approach of internal decomposition to other PGMs. The main structural problem described in Chapter 6 was the lack of logical and structural meaning of the odeco decomposition. One of the main properties of HMMs and tensor trains in comparison to PGMs and tensor networks in general is their low complexity. Increasing the complexity of the model from an HMM to a more general PGM and considering other forms of internal decomposition than the odeco tensor train decomposition could reveal a different kind of internal decomposition into a tensor network, which combines the advantage of a low memory demand with the provision of a logical connection to the PGMs structure, which could make it possible to implement inference and learning algorithms more efficiently. Finding an alternative internally decomposed representation and basing algorithms on its structure and logic might enable the representation to offer an alternative to the common representation of PGMs while being able to utilise existing libraries, acceleration techniques, and specialised hardware that was already developed for other tensor-based applications.

A Appendix

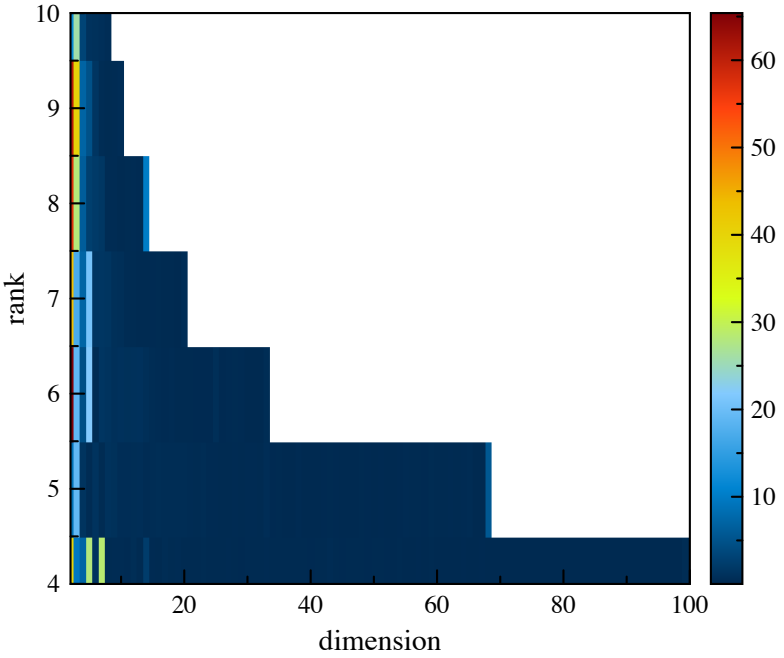

Figure A.1: Relative runtime uncertainty (in %) of a sequential implementation of the forward algorithm using an emission tensor.

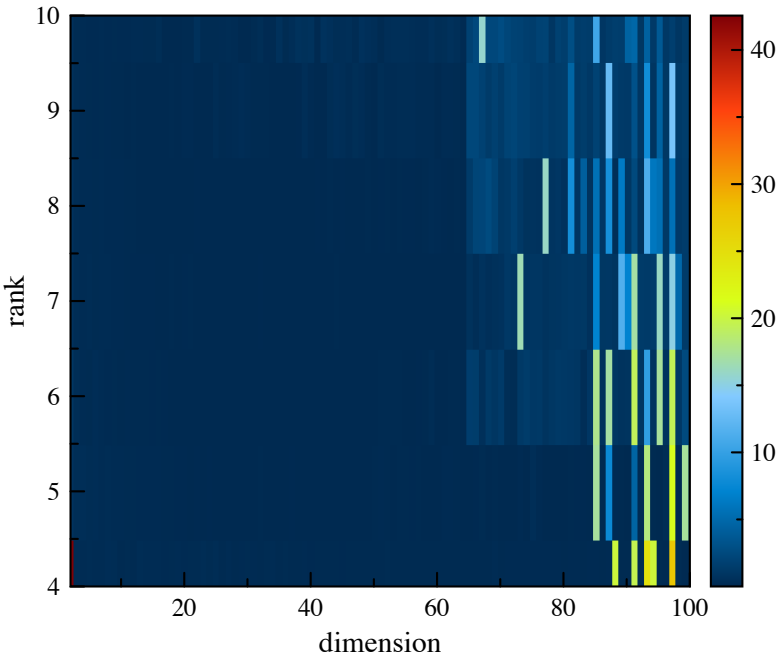

Figure A.2: Relative runtime uncertainty (in %) of a sequential implementation of the forward algorithm using an emission tensor train.

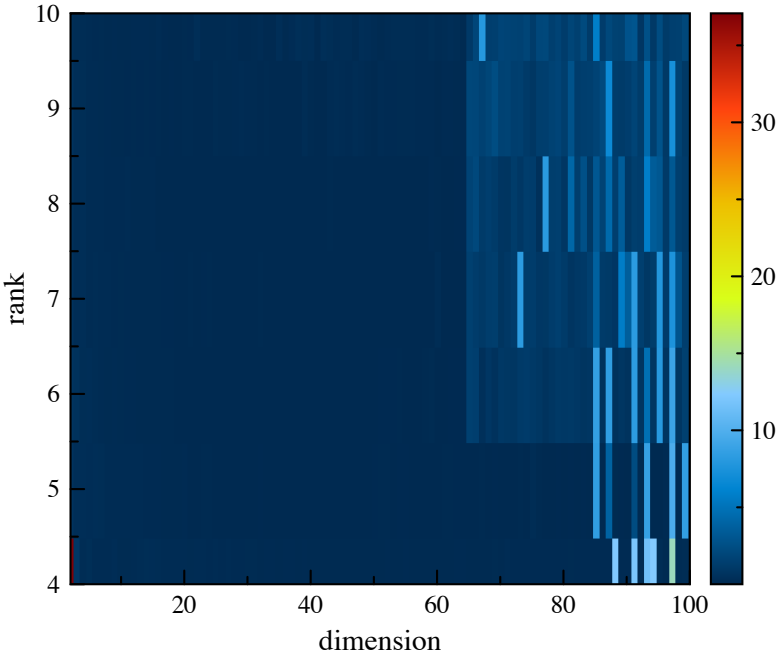

Figure A.3: Uncertainty (in %) of the relative runtime differences between a sequential implementation of the forward algorithm using an emission tensor train and an implementation with parallelised evidence incorporation in 2 threads.

Bibliography

Jacob Biamonte and Ville Bergholm. Tensor networks in a nutshell, 2017. URL https://arxiv.org/abs/1708.00006. visited on 23/08/2023.

Stephan Rabanser, Oleksandr Shchur, and Stephan Günnemann. Introduction to tensor decompositions and their applications in machine learning, 2017. URL https://arxiv.org/abs/1711.10781. visited on 28/08/2023.

Elina Robeva and Anna Seigal. Duality of graphical models and tensor networks. *Information and Inference: A Journal of the IMA*, 8(2):273–288, 06 2018. ISSN 2049-8772. URL https://doi.org/10.1093/imaiai/iay009. visited on 07/09/2023.

Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M Kakade, and Matus Telgarsky. Tensor decompositions for learning latent variable models. *Journal of machine learning research*, 15:2773–2832, 2014.

Rima Khouja, Pierre-Alexandre Mattei, and Bernard Mourrain. Tensor decomposition for learning gaussian mixtures from moments. *Journal of Symbolic Computation*, 113: 193–210, nov 2022. URL https://doi.org/10.1016/j.jsc.2022.04.002. visited on 07/09/2023.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. The itensor software library for tensor network calculations. *SciPost Phys. Codebases*, page 4, 2022a. URL https://doi.org/10.21468/SciPostPhysCodeb.4. visited on 09/08/2023.

Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. Tensornetwork: A library for physics and machine learning, 2019. URL https://arxiv.org/abs/1905.01330. visited on 28/08/2023.

Ivan Glasser, Ryan Sweke, Nicola Pancotti, Jens Eisert, and Ignacio Cirac. Expressive power of tensor-network factorizations for probabilistic modeling. *Advances in neural information processing systems*, 32, 2019.

Christopher J Hillar and Lek-Heng Lim. Most tensor problems are np-hard. *Journal of the ACM (JACM)*, 60(6):1–39, 2013.

Ivan Glasser, Nicola Pancotti, and J Ignacio Cirac. From probabilistic graphical models to generalized tensor networks for supervised learning. *IEEE Access*, 8:68169–68182, 2020.

Karim Halaseh, Tommi Muller, and Elina Robeva. Orthogonal decomposition of tensor trains. *Linear and Multilinear Algebra*, 70(21):6609–6639, 2022.

Tai-Danae Bradley. math3ma, 2019. URL https://www.math3ma.com/blog/matrices-as-tensor-network-diagrams. visited on 23/08/2023.

Joseph C. Kolecki. An introduction to tensors for students of physics and engineering. Technical report, NASA, 2002. URL https://www.grc.nasa.gov/www/k-12/Numbers/Math/documents/Tensors_TM2002211716.pdf. visited on 11/08/2023.

Paras Patidar. Tensors — representation of data in neural networks, 2019. URL https://medium.com/mlait/tensors-representation-of-data-in-neural-networks-bbe8a711b93b. visited on 11/08/2023.

Miles Stoudenmire. Tensor network, 2018. URL https://tensornetwork.org/mps/. visited on 17/08/2023.

Boaz Porat. A gentle introduction to tensors, 2014. URL https://www.ese.wustl.edu/~nehorai/Porat_A_Gentle_Introduction_to_Tensors_2014.pdf. visited on 25/08/2023.

Namgil Lee and Andrzej Cichocki. Fundamental tensor operations for large-scale data analysis in tensor train formats. *Multidimensional Systems and Signal Processing*, 29:921–960, 2018. URL https://doi.org/10.1007/s11045-017-0481-0. visited on 27/08/2023.

Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, fourth edition edition, 2021.

Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

L. Rabiner and B. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, 3(1):4–16, 1986. URL https://doi.org/10.1109/MASSP.1986.1165342. visited on 07/09/2023.

Ingmar Visser and Maarten Speekenbrink. *Mixture and Hidden Markov Models with R*. Springer Cham, 2022.

Jeffrey M. Dudek, Leonardo Dueñas-Osorio, and Moshe Y. Vardi. Efficient contraction of large tensor networks for weighted model counting through graph decompositions, 2020. URL https://arxiv.org/abs/1908.04381. visited on 28/08/2023.

Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*, 2010. URL http://eigen.tuxfamily.org. visited on 07/08/2023.

Barry Taylor and C Kuyatt. Guide to the expression of uncertainty in measurement, 1995. URL https://www.bipm.org/documents/20126/2071204/JCGM_GUM_6_2020.pdf/d4e77d99-3870-0908-ff37-c1b6a230a337. visited on 23/08/2023.

Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. *ITensor C++ documentation*, 2022b. URL https://itensor.org/docs.cgi?vers=cppv3&page=classes. visited on 23/08/2023.

Karim Halaseh, Tommi Muller, and Elina Robeva. *Tensor Network Decompositions - Implementation*, 2020. URL https://github.com/karimhalaseh/Tensor-Network-Decompositions. visited on 27/08/2023.