Universidad Complutense de Madrid
Facultad de Ciencias Matemáticas



Tesis Doctoral

Privacy via tensor network methods

Privacidad mediante métodos de redes tensoriales

José Ramón Pareja Monturiol

Directores:

David Pérez García
Alejandro Pozas Kerstjens

2025

Universidad Complutense de Madrid

Facultad de Ciencias Matemáticas

Programa de Doctorado en Investigación Matemática

Instituto de Ciencias Matemáticas



Tesis Doctoral

Privacy via tensor network methods

Privacidad mediante métodos de redes tensoriales

Memoria para optar al grado de doctor presentada por

José Ramón Pareja Monturiol

Directores:

David Pérez García

Alejandro Pozas Kerstjens

2025

UNIVERSIDAD COMPLUTENSE MADRID

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR

D./Dña. José Ramón Pareja Monturiol,
estudiante en el Programa de Doctorado en Investigación Matemática,
de la Facultad de Ciencias Matemáticas de la Universidad Complutense de Madrid, como autor/a de la tesis presentada para la obtención del título de Doctor y titulada:

Privacy via tensor network methods

Privacidad mediante métodos de redes tensoriales

y dirigida por: David Pérez García y Alejandro Pozas Kerstjens

DECLARO QUE:

La tesis es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, de acuerdo con el ordenamiento jurídico vigente, en particular, la Ley de Propiedad Intelectual (R.D. legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, modificado por la Ley 2/2019, de 1 de marzo, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), en particular, las disposiciones referidas al derecho de cita.

Del mismo modo, asumo frente a la Universidad cualquier responsabilidad que pudiera derivarse de la autoría o falta de originalidad del contenido de la tesis presentada de conformidad con el ordenamiento jurídico vigente.

En Madrid, a 25 de junio de 2025



Fdo.: _____

Esta DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD debe ser insertada en la primera página de la tesis presentada para la obtención del título de Doctor.

Para Alba,
sin cuyo apoyo,
constante e incondicional,
no podría haber llegado hasta aquí.

Agradecimientos

"There is nothing like looking, if you want to find something.
You certainly usually find something, if you look,
but it is not always quite the something you were after."

— J.R.R. Tolkien, *The Hobbit*

Y así, concluye este largo viaje de 4 años, plasmando en unas pocas páginas, los resultados—y tan sólo los resultados—obtenidos en este tiempo. Pero como podréis imaginar, realizar un doctorado conlleva muchas más cosas que no quedan reflejadas en una tesis, y que merecen al menos este breve espacio. Estos últimos 4 años han sido para mí una constante montaña rusa emocional. Por supuesto, ha sido una etapa de continuo aprendizaje, en la que he adquirido, creo, las habilidades y valores necesarios para hacer buena ciencia. Pero por el camino he aprendido otras cosas de las que nadie me había avisado, o que al menos no habría entendido antes de empezar el doctorado. Estar meses, o años, persiguiendo una misma idea, repitiendo la misma tarea una y otra, y otra, y otra vez, para que luego no salgan los resultados esperados y haya que volver a repetir o replantear todo... mientras ni siquiera tienes del todo claro lo que estás haciendo, si es interesante, si tiene aplicaciones, o si a alguien le importa... y mientras le dedicas a esto más horas diarias de las recomendables, a cambio de cobrar casi el salario mínimo, lo que te deja sin tiempo ni dinero para dedicar a otras actividades que te despejen... Esto es, sin duda, lo más difícil que he hecho nunca. Y supongo que este es el mayor aprendizaje que puedo sacar del doctorado, como imagino será para la mayoría. El aprender que la repetición, el pegarse contra un muro todos los días, es una componente intrínseca de la investigación, pero que luego da los frutos más satisfactorios. El aprender a estar orgulloso de mi trabajo, a bajarle el volumen a mis propias autocríticas, y esuchar un poco más los halagos de los que me apoyan. El aprender que las cosas requieren su tiempo, que hay que dejar respirar a las ideas, y que tengo que evitar obsesionarme tanto. En definitiva, aprender a disfrutar del arduo camino, y a que no pasa nada por desviarse de vez en cuando de éste, y sentarse en un banco para apreciar lo bonito de alrededor.

Pero que no parezca esto un lamento. Los caminos más difíciles son siempre los más bonitos y los que más recompensas traen. Y aunque es verdad que en estos años he pasado por oscuros

valles, también he coronado las cimas más brillantes. Gracias al doctorado he tenido la oportunidad de visitar algunos de los más relevantes centros de investigación del mundo, donde he conocido a personas increíblemente talentosas e interesantes. He sentido, en alguna ocasión, el momento *¡Eureka!*, que te invade de una intensa y duradera euforia al descubrir algo importante que llevaba atascando la investigación durante meses. He disfrutado enormemente aprendiendo de la belleza, simple y a la vez compleja, de las tensor networks, desconocidas anteriormente para mí. Y he disfrutado, puede que de lo que más, de las incansables horas discutiendo frente a una pizarra. Puede que este trabajo se sienta solitario en ocasiones, pasando muchas horas solo, mirando una pantalla, pero si algo tiene de bonito, son esos momentos, desgranando un problema a golpe de tiza junto con algún colaborador, compañero o amigo. Y es por estos momentos que aprecio y agradezco mi etapa en el doctorado, pero sobre todo he de agradecer a las personas que sostenían la otra tiza.

En primer lugar, como no podía ser de otra manera, quiero agradecer a mis directores, David y Alex, con quienes compartí esas primeras pizarras. Recuerdo cuando estaba acabando el grado y pensé en pedir una beca de introducción a la investigación para el máster, con la idea de abrir la puerta a, quizás, hacer un doctorado después. No tenía del todo claro qué quería investigar, pero lo que sí tenía claro era que, si tenía que elegir a alguien de la facultad bajo cuya supervisión trabajar los siguientes años, tenía que ser David Pérez García. Primero, y más importante, porque habiéndole conocido en el grado, sabía que era la mejor persona de la facultad, alguien cercano, amable y humilde, que ya me animaba al atacar esos problemas más difíciles, o exploratorios, de Cálculo Integral. Y segundo porque, además, sabía que investigaba cosas interesantes de cuántica, lo cual era un plus. No conseguí la beca, pero aún así David confió en mí y me contrató durante el máster, para después ofrecerse a dirigirme la tesis. Después de 5 años trabajando con él, no puedo más que dar las gracias, ante todo por darme la oportunidad de comenzar este camino y por brindarme tantas otras oportunidades durante el trayecto, pero también por acompañarme enseñándome a cómo ser científico y a qué debo aspirar, por enseñarme a tener el ojo atento para mirar todo con detalle, y por labrar un ambiente en el que sentirse cómodo al hacer las preguntas aparentemente más estúpidas, evitando que el ego quede nunca por encima de la duda o el rigor.

De Alex no tenía referencia previa, le conocí al comenzar a trabajar con David durante el máster, pero qué suerte tuve de que en esa lotería me tocase otra gran persona con la que trabajar. Aunque podría hacer extensibles a Alex las mismas palabras que tenía para David, y lo haría sin equivocarme, esto no haría justicia al papel tan fundamental que ha acarreado durante estos años. Gracias, Alex, por llevar esa voz cercana, la de, incluso, un amigo, que ha estado siempre disponible, siempre ahí para resolver mis dudas—no sólo las técnicas, que habrán sido las que menos. Gracias por enseñarme a hacer ciencia, a escribir artículos, a programar, a redactar informes y cartas educadas, gracias por enseñarme los mecanismos internos de la ciencia, por hablar sin tapujos, por ser comprensivo y un apoyo, por repetirme una y otra vez el valor de mi trabajo, y gracias por enseñarme a llevar la vida de un científico, no solo en lo laboral, sino también en lo personal.

Por otro lado, tengo que dar las gracias a todos aquellos que han estado presentes a lo largo de este camino, sin los cuales habría sido más difícil terminarlo y, sin duda, menos divertido. Gracias a mis compañeros de doctorado, a Pablo H., Quino, Fer, Laura y Juan Carlos, por hacer los días más amenos, enseñarme nuevas formas de jugar a los dardos y cuidar de mi escritorio en mi ausencia. Gracias, en especial, a mi hermano de doctorado, Pablo, por tantos momentos y, sobre todo, penas compartidas. Y gracias, por supuesto, a Alberto, Sofyan, Gianni, Angelo

y demás miembros de mathQI, por el rápido acogimiento, la cercanía y el buen ambiente que se respira en el grupo. No me olvido de dar las gracias a todas las personas increíbles que he conocido en mis viajes, con las que he vivido varios de los mejores días de esta etapa, en especial a Sergio, Ema y Paula. Many thanks as well to Roger Melko for welcoming me during my stay in Waterloo; without a doubt, one of the most enriching experiences I've had, in which I felt supported and included from the very beginning. I'm also deeply grateful to the wonderful friends I made there, Sehmi, Nishan, Schuyler, Mohammad, Estelle, Peam, and the rest, for making me feel like one of the group from day one, for the inspiring atmosphere you create, and for the great times skating, climbing, and hiking together.

Por último, quiero dar las gracias a mi familia, que aunque no tengan del todo claro lo que he estado haciendo estos últimos 4 años, han sido una pieza fundamental para llegar hasta aquí. Gracias a mis padres, por haberme dado todas las oportunidades y herramientas necesarias para estar donde estoy hoy, y por apoyarme siempre, pero sobre todo por educarme en el esfuerzo, por enseñarme a no pedir, y conseguir por mí mismo lo que me proponga, que es la herramienta más valiosa. A mis hermanas y sobrinas, por ser ese núcleo cálido al que volver, donde puedo olvidar si los experimentos se han roto o la demostración no sale, y solo tengo que estar atento de lo realmente importante, jugar. Gracias, en especial, a Ele, por ser un pilar, por entenderme. Y gracias, por supuesto, a la persona más importante de mi vida, Alba, sin la que no habría podido llegar al final del camino, o al menos no de la forma en que lo he hecho. Perdón, por hacerte lididar con todos los aspectos negativos de mi trabajo, los momentos de mayor ansiedad, demasiado comunes, las semanas de no levantarme del escritorio, y los momentos donde la creciente obsesión me iba sumiendo en un pozo. Gracias, por estar ahí siempre para sacarme de esa oscuridad, por apoyarme de forma constante e incondicional, por dedicarme tanto de tu tiempo, por escucharme siempre, por saber lo que necesitaba en cada momento y, sobre todo, por hacer todo ello con una sonrisa y siempre con el mismo cariño. Gracias, en definitiva, por recorrer todo este camino a mi lado, mientras ibas sembrando flores para hacerlo más bonito.

Esta tesis, junto con los trabajos que incluye y las actividades de formación y divulgación realizadas durante el doctorado, ha sido posible gracias al apoyo económico de diversas entidades públicas. En particular, mi doctorado ha sido financiado por el Ministerio de Ciencia e Innovación mediante un contrato predoctoral para la formación de doctores (CEX2019-000904-S-20-4), asociado al proyecto Severo Ochoa (CEX2019-000904-S) del ICMAT (CSIC), y financiado por MCIN/AEI/10.13039/501100011033. Además, la mayor parte de las actividades y desplazamientos que he realizado han contado con financiación de los proyectos PID2020-113523GB-I00, PID2023-146758NB-I00 y CEX-2023-001347-S del mismo Ministerio, así como del proyecto QUITEMAD (ref.TEC-2024/COM-84-QUITEMAD-CM) de la Comunidad de Madrid y del proyecto FEI-EU-22-06 de la Universidad Complutense de Madrid.

Contents

Agradecimientos v

Contents viii

List of Figures xi

List of Tables xii

List of Abbreviations xiii

Abstract xv

Resumen xvii

List of Publications xix

1 Introduction 1
- 1.1 The emergence and impact of neural networks . . . 1
- 1.2 Motivating limitations of neural networks . . . 2
- 1.3 Solutions from tensor network methods . . . 4
  - 1.3.1 Software package for tensor networks . . . 5
  - 1.3.2 Learning via low-rank decompositions . . . 5
- 1.4 Organization of the thesis . . . 6

2 Preliminaries 7
- 2.1 Notations . . . 7
- 2.2 Fundamentals of machine learning . . . 8
  - 2.2.1 Supervised learning . . . 9
  - 2.2.2 Unsupervised learning and density estimation . . . 10
  - 2.2.3 Illustrative examples . . . 11
- 2.3 Privacy in machine learning . . . 13
  - 2.3.1 Privacy attacks . . . 14

2.3.2 Defense mechanisms . . . 15
2.4 Introduction to tensor networks . . . 16
2.4.1 Tensor trains . . . 18
2.4.2 Representation of continuous functions . . . 21
2.4.3 Canonical forms . . . 22
2.5 Matrix factorizations and tensor train decompositions . . . 23
2.5.1 Singular value decomposition . . . 23
2.5.2 Sketching techniques . . . 24
2.5.3 Cross interpolation . . . 26
2.5.4 A unified framework . . . 27

3 Tensor networks for machine learning 29
3.1 Supervised learning . . . 29
3.1.1 Tensor train models as SVMs . . . 30
3.1.2 Embedding functions . . . 30
3.1.3 Multi-label classification . . . 31
3.2 Density estimation . . . 32
3.2.1 Direct sampling from tensor trains . . . 34
3.3 Training algorithms . . . 35
3.3.1 Direct gradient descent . . . 35
3.3.2 DMRG-based optimization . . . 36

4 Tensorization of neural networks 39
4.1 Motivation . . . 39
4.2 Description of the algorithm . . . 40
4.2.1 Main idea . . . 40
4.2.2 Details of subroutines . . . 42
4.2.3 Continuous case . . . 45
4.2.4 Complexity . . . 48
4.3 Relation with TT-CI . . . 48
4.4 Performance . . . 49
4.4.1 Non-densities . . . 50
4.4.2 Neural networks . . . 52
4.4.3 Dependence on hyperparameters . . . 55
4.5 Conclusions and future directions . . . 56

5 Privacy via tensor networks 59
5.1 Motivation . . . 59
5.2 White-box privacy vulnerability . . . 60
5.3 Protection via reparameterization invariance . . . 62
5.4 Univocal canonical form . . . 63
5.5 Experiments . . . 67
5.5.1 Medical data . . . 67
5.5.2 Audio data . . . 71
5.6 Conclusions and future directions . . . 77

6 Further applications 79

6.1 Interpretability from tensor trains . . . 80
6.1.1 AKLT Model . . . 81
6.1.2 Tensorization . . . 82
6.1.3 Order parameter . . . 83
6.2 Initialization mechanism . . . 85
6.3 Compression of neural networks . . . 88
6.4 Conclusions and future directions . . . 90

7 TensorKrowch **91**
7.1 Motivation . . . 91
7.2 Installation and requirements . . . 92
7.3 Basic usage . . . 93
7.4 The building blocks . . . 95
7.4.1 Types of nodes . . . 96
7.5 Operations between nodes . . . 97
7.5.1 Trace and reset . . . 100
7.6 Building and training TN models . . . 101
7.6.1 Model training . . . 104
7.7 Time and memory optimizations . . . 104
7.7.1 Operations . . . 105
7.7.2 Tracing . . . 105
7.7.3 Memory Modes . . . 105
7.8 Additional package information . . . 106
7.8.1 Documentation for TensorKrowch . . . 106
7.8.2 Contribution guidelines . . . 107
7.9 Conclusions and future directions . . . 108

Bibliography **111**

List of Figures

4.1 Performance of TT-RSS applied to random TTs and Slater functions when varying the number of pivots N . . . 51
4.2 Performance of TT-RSS applied to Bars and Stripes and MNIST neural network classifiers when varying the number of pivots N. . . . 54
4.3 Distribution of accuracies of tensorized models when varying the embedding dimension d, the rank r and the number of pivots N. . . . 56

5.1 Illustration of the proposed white-box privacy vulnerability. . . . 61
5.2 Average accuracies of NN and TT models predicting COVID-19 outcomes across datasets with varying imbalance q. . . . 69
5.3 Average attack accuracies for predicting the majority value of the irrelevant feature in a medical dataset, for NN models and TTs under different reparameterizations. . . . 70
5.4 Accuracy of the original NN models, TT models from TT-RSS, and TT models after re-training, evaluated on a balanced test set across English/Canadian women/men, for all percentages q of English speakers. . . . 73
5.5 Difference between pivot and test accuracy for TT models across varying proportions of English speakers. . . . 75
5.6 Accuracy of logistic regression attacks classifying models by majority accent in their training data, for black-box and white-box scenarios. . . . 76

6.1 Fidelity between $|\Psi\rangle$ and $|\widehat{\Psi}\rangle$ obtained via TT-RSS, estimated order parameter $16\mathcal{E}_{10}$, and TT-RSS runtime. . . . 85
6.2 Training accuracies of TT models for binary classification with different embeddings and initialization methods. . . . 87
6.3 Comparison of memory and time efficiency for models tensorized via different methods, using various choices of ranks r. . . . 89

7.1 Comparison of the maximum memory usage for a single contraction of the built-in `MPSLayer` class, with and without tracing, using different bond dimensions. . . . 106
7.2 Comparison of running times for a single contraction of the built-in `MPSLayer` class, using different bond dimensions. . . . 107

List of Tables

5.1 Pearson correlation coefficients between the features of the medical dataset used for training the NN and TT models. 68

6.1 Correspondence between elements g_α of the symmetry group $\mathbb{Z}_2 \times \mathbb{Z}_2$, elements u_α of a unitary representation, and elements V_α of a projective representation. 83

List of Abbreviations

AI	Artificial Intelligence
AKLT	Affleck–Kennedy–Lieb–Tasaki
ALS	Alternating Least Squares
BR	Born Rule
CDE	Core Determining Equation
CI	Cross Interpolation
CNN	Convolutional Neural Network
DMRG	Density Matrix Renormalization Group
DNN	Deep Neural Network
DP	Differential Privacy
ERM	Empirical Risk Minimization
FFT	Fast Fourier Transform
GAN	Generative Adversarial Network
GMM	Gaussian Mixture Model
GPU	Graphics Processing Unit
KKT	Karush–Kuhn–Tucker
LLM	Large Language Model
MCMC	Markov Chain Monte Carlo
MERA	Multi-scale Entanglement Renormalization Ansatz

ML	Machine Learning
MLE	Maximum Likelihood Estimation
MLP	Multi-Layer Perceptron
MPO	Matrix Product Operator
MPS	Matrix Product State
NN	Neural Network
NNQS	Neural Network Quantum State
OBC	Open Boundary Condition
OC	Orthogonality Center
PBC	Periodic Boundary Condition
PCA	Principal Component Analysis
PEPS	Projected Entangled Pair State
RBM	Restricted Boltzmann Machine
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SVD	Singular Value Decomposition
SVM	Support Vector Machine
SPT	Symmetry-Protected Topological
TI	Translationally Invariant
TN	Tensor Network
TNN	Tensor Neural Network
TR	Tensor Ring
TT	Tensor Train
TT-CI	Tensor Train via Cross Interpolation
TT-RS	Tensor Train via Recursive Sketching
TT-RSS	Tensor Train via Recursive Sketching from Samples
TTM	Tensor Train Matrix
VAE	Variational Autoencoder

Abstract

Although the foundational ideas behind Neural Networks (NNs) date back to the 1950s, their widespread application has emerged only in recent years. This surge has been driven by their flexibility, scalability, and the rapid development of specialized hardware, allowing these models to achieve state-of-the-art performance across diverse domains such as computer vision, natural language processing, and scientific modeling. However, NNs present several concerning limitations—particularly in terms of privacy, interpretability, and efficiency—that have become increasingly evident with the advent of generative models trained on vast internet-sourced datasets. These shortcomings raise ethical and practical concerns, including the leakage of sensitive data and limited transparency regarding the internal mechanisms of these models, which also restricts their applicability.

A central motivation of this thesis is the problem of privacy. Specifically, we identify a novel vulnerability: when training NNs via gradient descent, information from the training dataset can imprint identifiable patterns in the model parameters. To address this, we propose a solution based on Tensor Network (TN) models, leveraging their reparameterization flexibility. In particular, we introduce a holomorphic canonical form for one-dimensional TNs that provably eliminates all latent patterns from the parameters. To demonstrate both the vulnerability and our proposed solution, we perform privacy attacks on models trained with a medical dataset, showing that TNs offer effective protection against such leakage.

Tensor networks, originally developed in condensed matter physics, are low-rank factorizations of high-dimensional tensors that support both analytical and practical analysis through their efficient representations and algebraic structure. Among the various TN architectures, we focus on one-dimensional networks, known as Matrix Product States (MPS) or Tensor Trains (TT), which have recently gained attention as versatile models for machine learning tasks.

However, although several computational tools exist for performing calculations with TNs, there remains a lack of specialized, user-friendly software tailored for machine learning applications—particularly tools that facilitate rapid exploration of diverse architectures, optimization methods, and use cases. To address this limitation, a key contribution of this thesis is the development of an open-source software package, TensorKrowch, that integrates TN models within one of the most widely adopted machine learning frameworks, PyTorch. This package provides all the necessary tools to build and train any TN, leveraging built-in PyTorch features such as

automatic differentiation, GPU acceleration, and the ability to construct complex models from modular TN and NN components.

Alternatively to training via gradient descent using automatic differentiation, TNs can be learned through low-rank tensor decompositions. Building on this idea, we introduce a tensorization algorithm that converts pretrained NNs into TT form, thereby enabling the benefits of TNs as a post-processing step. This algorithm, called Tensor Train via Recursive Sketching from Samples (TT-RSS), combines sketching and cross interpolation techniques to produce an efficient decomposition that requires only black-box access to the NN and a representative sample set defining the domain of interest. As such, it is particularly well-suited for machine learning models, where the domain is naturally defined by the training data.

Applying this tensorization, we extend our privacy experiments to more realistic, high-dimensional scenarios, including NNs trained to classify voice samples by gender. We demonstrate that the parameters of standard NNs can reveal whether the training data included speakers with Canadian or English accents, an inference not possible with their TN-based counterparts.

Beyond privacy, we explore further applications of TNs through TT-RSS. In terms of interpretability, we show that TT-RSS can be used to estimate topological phases of matter, which are readily accessible via the TT representation. Moreover, we demonstrate that this tensorization serves as an efficient initialization method for optimizing TTs in general settings. For model compression, our algorithm achieves a superior trade-off between memory and time complexity compared to conventional layer-wise tensorizations of NNs.

Resumen

Aunque las ideas fundamentales que dieron origen a las redes neuronales, o Neural Networks (NNs), se remontan a la década de 1950, su uso generalizado no ha llegado hasta los últimos años. Este auge ha sido posible gracias a su flexibilidad y escalabilidad, así como al rápido desarrollo de hardware especializado, lo que ha permitido que estos modelos alcancen resultados punteros en ámbitos tan diversos como la visión por ordenador, el procesamiento del lenguaje natural o la modelización científica. No obstante, las NNs presentan varias limitaciones preocupantes—especialmente en términos de privacidad, interpretabilidad y eficiencia—que han cobrado mayor relevancia con la aparición de modelos generativos entrenados con enormes volúmenes de datos obtenidos de internet. Estas carencias plantean retos tanto éticos como prácticos, como la posible filtración de información sensible o la falta de transparencia en su funcionamiento interno, lo cual también restringe su aplicación en determinados contextos.

Una de las principales motivaciones de esta tesis es la cuestión de la privacidad. En concreto, identificamos una nueva vulnerabilidad: al entrenar NNs mediante descenso por gradiente, puede suceder que la información contenida en los datos de entrenamiento quede reflejada en los parámetros del modelo de forma identificable. Como solución, proponemos el uso de modelos basados en redes tensoriales, o Tensor Networks (TNs), aprovechando su capacidad de reparametrización. En particular, introducimos una forma canónica holomorfa para TNs unidimensionales, que permite eliminar de manera demostrable cualquier patrón latente en los parámetros. Para ilustrar tanto la vulnerabilidad como la eficacia de nuestra propuesta, realizamos ataques de privacidad sobre modelos entrenados con un conjunto de datos médicos, y mostramos que las TNs ofrecen una defensa eficaz frente a este tipo de fugas.

Las redes tensoriales, originarias de la física de la materia condensada, son factorizaciones de bajo rango de tensores de alta dimensión que permiten tanto un análisis teórico como práctico gracias a su estructura algebraica y representación eficiente. De entre las distintas arquitecturas de TNs, nos centramos en las redes unidimensionales, conocidas como Matrix Product States (MPS) o Tensor Trains (TT), que recientemente han ganado interés como modelos versátiles para tareas de aprendizaje automático.

Sin embargo, aunque existen varias herramientas computacionales para realizar cálculos con TNs, sigue habiendo una carencia de software especializado y accesible, diseñado específicamente para aplicaciones de aprendizaje automático—especialmente herramientas que faciliten la exploración rápida de diversas arquitecturas, métodos de optimización y casos de uso. Para cubrir

esta necesidad, una contribución clave de esta tesis es el desarrollo de un paquete de software libre, TensorKrowch, que integra modelos basados en TNs en PyTorch, uno de los entornos más utilizados en aprendizaje automático. Este paquete ofrece todas las herramientas necesarias para construir y entrenar cualquier TN, aprovechando funcionalidades propias de PyTorch como la diferenciación automática, la aceleración mediante GPU y la construcción modular de modelos combinando TNs y NNs.

Como alternativa al entrenamiento mediante descenso por gradiente usando diferenciación automática, los modelos de TNs también pueden aprenderse a través de descomposiciones tensoriales de bajo rango. Siguiendo esta idea, introducimos un algoritmo de tensorización que permite transformar NNs ya entrenadas en forma de TT, lo que permite beneficiarse de las ventajas de las TNs como paso posterior al entrenamiento. Este algoritmo, llamado Tensor Train via Recursive Sketching from Samples (TT-RSS), combina técnicas de sketching y cross interpolation para obtener una descomposición eficiente que solo requiere acceso de caja negra a la NN y un conjunto de muestras representativas que definan la región de interés. Por ello, resulta especialmente adecuado para modelos de aprendizaje automático, donde el dominio relevante viene naturalmente definido por los datos de entrenamiento.

Gracias a esta tensorización, ampliamos nuestros experimentos de privacidad a escenarios más realistas y de mayor dimensión, como NNs entrenadas para clasificar muestras de voz según el género. Mostramos que es posible averiguar si los datos de entrenamiento incluían hablantes con acento canadiense o británico a partir de los parámetros de una NN convencional, mientras que esto no es posible con sus análogos en forma the TN.

Más allá del ámbito de la privacidad, exploramos otras aplicaciones de las TNs mediante TT-RSS. En lo que respecta a la interpretabilidad, mostramos que TT-RSS puede utilizarse para estimar fases topológicas de la materia, que resultan fácilmente accesibles desde la representación en forma de TT. Además, demostramos que esta tensorización es útil como método eficiente de inicialización para la optimización de TTs en contextos más generales. Por último, en el ámbito de la compresión de modelos, nuestro algoritmo consigue una mejor relación entre memoria y tiempo de cómputo que las convencionales tensorizaciones capa a capa de NNs.

List of Publications

The main results of this thesis have led to two published papers—one of which corresponds to an open-source software package—and a pre-print:

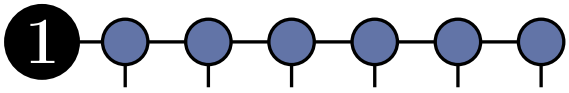
- A. Pozas-Kerstjens, S. Hernández-Santana, J. R. Pareja Monturiol, M. Castrillón López, G. Scarpa, C. E. González-Guillén, and D. Pérez-García.
  Privacy-preserving machine learning with tensor networks.
  Quantum, 8 (2024), p. 1425, arXiv:2202.12319.

- J. R. Pareja Monturiol, D. Pérez-García, and A. Pozas-Kerstjens.
  TensorKrowch: Smooth integration of tensor networks in machine learning.
  Quantum, 8 (2024), p. 1364, arXiv:2306.08595.

  `https://github.com/joserapa98/tensorkrowch`

- J. R. Pareja Monturiol, A. Pozas-Kerstjens, and D. Pérez-García.
  Tensorization of neural networks for improved privacy and interpretability.
  arXiv pre-print (2025), arXiv:2501.06300.

Introduction



This thesis revolves around the intersection of two main themes: the privacy risks inherent in current machine learning practices and the potential of tensor networks as alternative models that address some of the fundamental limitations of neural networks. The following provides a brief historical overview of neural networks and their associated challenges, alongside an introduction to tensor network methods and the advantages motivating their adoption. This context serves to introduce the topics explored in this thesis, presenting its contributions in the chronological order in which the research was conducted, as opposed to the thematic organization of the subsequent chapters.

1.1 The emergence and impact of neural networks

On October 8th, 2024, the Nobel Prize in Physics was awarded to John Hopfield and Geoffrey Hinton, as the official citation stated:

> *"for foundational discoveries and inventions that enable machine learning with artificial neural networks".*

This decision came as a surprise to part of the community, marking a paradigm shift in the recognition of Artificial Intelligence (AI) as a field with profound implications for physics, rather than awarding more traditional areas within the discipline.

John Hopfield, originally trained as a physicist, bridged the disciplines of neuroscience and statistical mechanics by introducing the Hopfield network in 1982 [1], one of the first Machine Learning (ML) models to employ energy-based principles for describing systems that converge to stable states representing stored patterns. Geoffrey Hinton, often referred to as the "godfather of AI", developed the Boltzmann machine, a stochastic network capable of learning probability distributions [2]. He later introduced layered learning schemes—most notably, the training of deep belief networks via stacked Restricted Boltzmann Machines (RBMs) [3–5]—which provided the first practical method for training deep architectures.

These foundational concepts catalyzed the development of increasingly complex models, broadly categorized under the term Neural Networks (NNs). Over time, specialized architectures emerged to address different data modalities, including Convolutional Neural Networks (CNNs) for image data [6] and Recurrent Neural Networks (RNNs) for sequential data such as text [7]. Despite their advantageous expressive power, early networks faced challenges in optimization and scalability.

A major breakthrough occurred in 2012 with the introduction of AlexNet [8], which demonstrated that Deep Neural Networks (DNNs) could be trained effectively. This success was enabled by combining deep CNNs with Rectified Linear Unit (ReLU) activations [9], and training via backpropagation [7], on Graphics Processing Units (GPUs) for the first time. This milestone marked a turning point in machine learning. Coupled with rapid advances in hardware, the field experienced significant progress, as reflected in the evolution from AlexNet to more advanced architectures such as VGGNet [10], GoogleNet [11], and ResNet [12].

In 2017, the introduction of the transformer model [13] marked another paradigm shift. Transformers enabled the modeling of long-range dependencies and overcame the memory limitations of RNNs. Moreover, their highly parallelizable design facilitated the scalability of models to billions of parameters. These advances paved the way for the development of large-scale models trained on massive corpora of text or images. Among them, Large Language Models (LLMs), such as BERT [14], GPT [15] and their successors [16–18], have demonstrated remarkable capabilities in language understanding, generation, and reasoning, with far-reaching implications for science, education, and society at large.

As a consequence of these developments, the foundational ideas introduced by Hopfield and Hinton—rooted in both neuroscience and physical principles—have profoundly influenced these and other fields, underpinning some of the most significant scientific advances in recent decades. These include the discovery of the Higgs boson [19, 20], the analysis of gravitational lensing and cosmic microwave background data for inferring large-scale cosmic structures [21], and the accelerated discovery of materials with tailored properties [22].

In quantum many-body physics, RBMs and more general energy-based models have proven particularly successful, with their expressive power enabling the approximation of complex quantum systems within exponentially large Hilbert spaces, thereby mitigating the *curse of dimensionality*. This capability has facilitated applications such as quantum state tomography [23–26], ground-state wave function approximation [27–31], and phase classification of quantum systems [32–34]. A range of architectures has been employed in these contexts, including RBMs [27, 29], CNNs [25], RNNs [30], and transformer-based models [31], inspired by the progress in language modeling.

Another landmark achievement in AI was AlphaFold 2 [35], a transformer-based model released in 2021, which effectively solved the decades-old protein folding problem, predicting three-dimensional structures from amino acid sequences with unprecedented accuracy. Notably, this accomplishment earned Demis Hassabis and John Jumper the Nobel Prize in Chemistry also in 2024.

1.2 Motivating limitations of neural networks

Despite their successes in scientific development, NNs have seen limited adoption in industrial applications due to several inherent issues. One of the main criticisms is that NNs function as *black boxes*, offering no direct means of understanding their inner workings. This lack of

interpretability is especially problematic in fields such as medicine or finance, where it is not only important to obtain accurate predictions, but also to understand which factors influence the outcome and how. Simpler models like logistic regression and decision trees are inherently interpretable but less expressive, limiting their applicability to complex tasks. In contrast, the versatility of NNs makes them more powerful tools, but interpreting their output typically requires *post hoc* techniques to explain their decisions [36, 37]. To address this limitation and strike a better balance between expressiveness and interpretability, some researchers aim to design NN architectures and optimization methods that are inherently interpretable [38, 39].

Another significant concern with NNs is privacy, particularly with the advent of generative models trained on vast datasets collected from the internet [40, 41]. Although privacy concerns surrounding the use of personal data predate the widespread use of NNs, it has now become increasingly evident that much of the data available online is treated as public and, in particular, is absorbed—and in some cases memorized—by large AI models trained on internet-scale data. This has raised questions about how companies manage such data and whether their models retain sensitive information, leading to growing demands for individuals to have the right to opt out of their data being used for model training. These concerns extend beyond privacy to include intellectual property and copyright issues. However, determining whether a model has memorized sensitive data remains a major challenge, in part due to the lack of interpretability in these complex systems. As a result, in the absence of more transparent and controllable models, such issues can only be addressed heuristically by imposing behavioral constraints on these black-box models. For instance, some techniques aim to reduce the risk of generating private data by incorporating reinforcement learning with human feedback during training [42]. Nevertheless, recent studies have shown that such defenses can be circumvented, allowing the extraction of training data and thereby compromising privacy [43, 44].

A more rigorous approach to ensuring privacy in machine learning is Differential Privacy (DP) [45], a formal framework that requires the output distribution of a randomized algorithm to be nearly indistinguishable whether or not a particular data point is included in the input dataset. This definition protects individual privacy and enables the design of algorithms with provable privacy guarantees [46]. DP mechanisms typically work by injecting noise into the computation, with variance proportional to the function's *sensitivity*—i.e., the maximum change in output resulting from modifying a single data point [47]. A key drawback of DP is its impact on accuracy: stronger privacy guarantees require more noise, which can degrade model performance.

The state-of-the-art algorithm for preserving DP in NNs is DP-SGD [48], a variant of Stochastic Gradient Descent (SGD) that adds noise at each training step to ensure privacy during optimization. However, this approach makes the final model highly sensitive to the amount of noise needed to maintain a meaningful level of privacy, limiting its effectiveness to small NN architectures.

Moreover, since DP is intended to protect individual data points, DP mechanisms are generally vulnerable to attacks that aim to infer broader properties of the training data distribution. This vulnerability motivated the first contribution of this thesis:

> A. Pozas-Kerstjens, S. Hernández-Santana, J. R. Pareja Monturiol, M. Castrillón López, G. Scarpa, C. E. González-Guillén, and D. Pérez-García.
> *Privacy-preserving machine learning with tensor networks*.
> *Quantum*, 8 (2024), p. 1425, arXiv:2202.12319.

In this work, we demonstrate that gradient-based optimization can cause data leakage, as infor-

mation from the training dataset can create identifiable patterns within the NN parameters. We also proposed Tensor Network (TN) models as a provable solution to such attacks, introducing a novel holomorphic canonical form that eliminates these patterns through reparameterization. To validate both the vulnerability and our proposed solution, we conduct privacy attacks on models trained with a medical dataset, showing that TNs offer effective protection against this form of leakage.

1.3 Solutions from tensor network methods

Tensor networks provide efficient representations of high-dimensional tensors with low-rank structure. Originating in condensed matter theory, they were developed as tractable representations of quantum many-body states, facilitating the study of entanglement, symmetries, and phase transitions [49–51]. Due to their practical efficiency, TNs have become a fundamental tool to simulate large quantum systems [52–57].

Recently, TNs have been proposed as quantum-inspired ML models for broader tasks. Initial studies in this area [58, 59] implemented 1D models known as Matrix Product States (MPS) [60] or Tensor Trains (TTs) [61]. Subsequent research introduced the use of more complex TNs, such as Trees [62, 63], Multi-scale Entanglement Renormalization Ansatz (MERA) [64] or Projected Entangled Pair States (PEPS) [65], for both supervised and unsupervised learning tasks.

Typically, TN models are trained through gradient descent methods, similarly to NNs. However, one drawback of these algorithms is the number of hyperparameters that must be adjusted for effective optimization. TNs present additional challenges, as it is necessary to determine how to fix or update the inner ranks of the tensors composing the network. Additionally, the application of a TN involves the contraction of multiple tensors, making computations highly sensitive to small perturbations, which can cause rapid explosion or vanishing of outputs and gradients. This sensitivity complicates the design of effective initialization methods. Although some initialization strategies have been proposed [66, 67], they are often not generalizable across different network structures or data embeddings.

Despite these issues, TN models offer certain advantages over NNs. For instance, TNs have been shown to outperform other deep and classical algorithms on tabular data in anomaly detection tasks [68]. Furthermore, from their successful application in representing complex quantum systems, TNs are known to be effective in modeling probability distributions, especially when the network layout aligns with the data structure (e.g., using 2D PEPS for image modeling [65]). This structural alignment enables TNs to access critical information directly, such as the correlations within the data (i.e., the entanglement structure when studying quantum systems), or efficient computation of marginal and conditional distributions. This interpretability has been highlighted in recent studies [69, 70]. The capacity to model entire distributions and compute conditional probabilities has been leveraged to create TN models capable of both classification and generation, suggesting potential robustness against adversarial attacks [71, 72]. Regarding the previously mentioned data leakages that arise as a side effect of gradient descent training, TNs have been proposed as a solution due to the explicit characterization of their gauge freedom. This property allows for the definition of new sets of parameters without altering the behavior of the black-box, thereby eliminating potential patterns introduced by training data [73].

Another key aspect that has drawn attention to TNs is their efficiency. This feature has already been utilized to compress pre-trained NNs by converting each linear layer in the model into a TN using low-rank tensor decompositions [74–78]. A particularly successful application

has been the compression of LLMs with billions of parameters, reducing parameter counts by up to 90%, and decreasing training and inference times by 50% and 25%, respectively [79–82].

1.3.1 Software package for tensor networks

There are several reasons to believe that the integration of TNs into deep learning pipelines can enhance the capabilities of current models. Several packages exist [83–87] that allow to use some concrete TN architectures as ML models, or that use gradient-based methods to optimize TN ansätze for quantum many-body calculations. However, from the point of view of machine learning, there is still a need for extensive research to determine, for instance, the situations in which the properties of TNs can be maximally leveraged, which are the most effective training methods, initializations, and optimal architectures—including, importantly, architectures beyond those used by physicists. Consequently, there is a demand for user-friendly tools that enable rapid experimentation in this field.

To address this demand, part of this thesis is dedicated to the development of an open-source software package, TensorKrowch, which integrates TN utilities into PyTorch [88], one of the most widely used machine learning frameworks. This effort led to a second published paper detailing the components of the package:

> J. R. Pareja Monturiol, D. Pérez-García, and A. Pozas-Kerstjens.
> *TensorKrowch: Smooth integration of tensor networks in machine learning.*
> *Quantum*, 8 (2024), p. 1364, arXiv:2306.08595.
> `https://github.com/joserapa98/tensorkrowch`

The package enables users to construct arbitrary TN models using the familiar language and capabilities of PyTorch. At its foundation, the package introduces a set of basic components—`Nodes` and `Edges`—that serve as the building blocks for constructing `TensorNetwork` models. These integrate seamlessly with other PyTorch modules, thus benefiting from GPU acceleration, automatic differentiation, compilation to XLA, and the modular composition of multi-layer networks. Additionally, TensorKrowch includes implementations of common TNs such as MPS, MPOs, Trees, and PEPS; tools for tensorizing layers in pre-trained NNs; and methods developed within the context of quantum information theory that enable interpreting the inner workings of the models [62, 69, 70], such as the computation of reduced density matrices and entanglement entropies.

1.3.2 Learning via low-rank decompositions

While TensorKrowch provides flexibility for implementing static TNs inspired by physical models, one of its primary objectives is to facilitate training TNs via gradient descent for ML tasks. However, as discussed earlier, gradient-based optimization presents some difficulties, since it requires relying on heuristics to find proper hyperparameters, initializations, and model architectures. An alternative is to construct TN models directly using low-rank decompositions. For instance, using basic notions from linear algebra such as the Singular Value Decomposition (SVD), one can derive a TN representation of an empirical distribution, a pre-trained model, or more general functions. This approach bypasses many of the choices inherent in gradient-based training, and offers benefits in terms of privacy (in the sense defined in Ref. [73]), interpretability, efficiency, and robustness to other threats, including adversarial attacks.

Motivated by this, we aim to employ low-rank tensor decompositions to build TT models only from black-box access to a pre-trained NN, thereby enhancing interpretability and privacy in the reconstructed model. To this end, we introduce an adaptation of the sketching approach used in the Tensor Train via Recursive Sketching (TT-RS) algorithm [89], which also incorporates principles from Tensor Train via Cross Interpolation (TT-CI) [90, 91]. This algorithm, Tensor Train via Recursive Sketching from Samples (TT-RSS), is better suited for decomposing NNs (those modeling probability distributions) when provided with a set of training samples that define the subregion of interest within the model's domain. Using this method, we achieve promising results in approximating image (Bars and Stripes, MNIST [92]) and speech (CommonVoice [93]) classification models, as well as in approximating more general functions beyond probability distributions. Additionally, we explore how this approach enables leveraging the benefits of TNs by obtaining TT representations of NN models. In particular, we address previously mentioned concerns, including privacy and interpretability, and discuss further applications such as TT initialization for gradient-descent training and model compression. A detailed presentation of TT-RSS and the associated experiments is provided in the following pre-print, which constitutes the final contribution of this thesis:

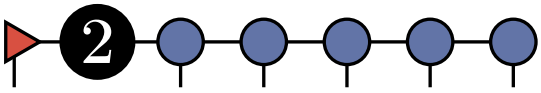
J. R. Pareja Monturiol, A. Pozas-Kerstjens, and D. Pérez-García.
Tensorization of neural networks for improved privacy and interpretability.
arXiv pre-print (2025), arXiv:2501.06300.

1.4 Organization of the thesis

The remainder of the thesis is structured as follows. Chapter 2 presents the foundations of this work, including a broad introduction to machine learning, the concept of privacy—defining differential privacy and outlining common attacks on ML models—the basics of TNs along with the graphical notation used throughout the thesis, and key concepts from matrix factorization relevant to TT decompositions. Chapter 3, also serving as background, focuses on the use of TNs in ML.

From Chapter 4 to Chapter 7, we present the core contributions of this thesis. For clarity, the results are organized not chronologically—as in this chapter—but thematically: we first introduce methods for learning TTs, then explore their applications, and finally describe the TensorKrowch software package. Specifically, Chapter 4 introduces TT-RSS and demonstrates its application to the tensorization of NNs and more general functions. Chapter 5 presents the privacy vulnerability identified in our first paper, provides a proof of the robustness of TT models against such attacks through the introduction of a holomorphic canonical form, and illustrates this with two case studies: one involving a small medical dataset and gradient-based training of TTs, and another, more realistic, case involving learning TT representations of voice classifiers via TT-RSS. Chapter 6 explores additional applications of TT-RSS, including: (i) interpretability in a physics-inspired setting, by estimating the topological phase of the ground state of the Affleck–Kennedy–Lieb–Tasaki (AKLT) model—commonly referred to as the AKLT state—from the TT representation produced by TT-RSS; (ii) initialization, demonstrating the use of TT-RSS for pre-training prior to gradient-based optimization of TTs; and (iii) compression, comparing single-TT tensorization via TT-RSS with layer-wise tensorization of linear layers in NNs.

Preliminaries



This chapter introduces the core concepts and technical tools that underpin the rest of this thesis. It begins with an overview of fundamental ideas in machine learning, followed by a discussion on privacy, which includes common attack strategies against machine learning models and corresponding defense mechanisms, with particular emphasis on the formal definition of differential privacy. The chapter then covers the basics of tensor networks, introducing the graphical notation used throughout the thesis and key structural properties such as canonical forms in tensor train models. It concludes with essential notions from matrix factorizations that are particularly relevant to tensor train decompositions.

2.1 Notations

Throughout this thesis we are interested in ML tasks, which can be expressed as problems where one tries to approximate an underlying, unknown function f, typically an n-dimensional real-valued continuous function $f : \mathcal{X}_1 \times \cdots \times \mathcal{X}_n \to \mathbb{R}$, with $\mathcal{X}_1, \ldots, \mathcal{X}_n \subset \mathbb{R}$. In most cases of interest, the goal will be to approximate densities p, restricting to the case where f is non-negative and normalized. To denote the approximations given by the models, we will use $\hat{f}$ or $\hat{p}$.

Discrete functions are denoted as $T : [d_1] \times \cdots \times [d_n] \to \mathbb{R}$, where $[d]$ denotes the set $\{1, \ldots, d\}$ for any $d \in \mathbb{N}$. We generally treat T as an *order-n tensor*, interpreting its evaluations as indexing tensor elements. For instance, a $m \times n$ matrix $(A_{ij})_{i\in[m],j\in[n]}$ is expressed as a function $A : [m] \times [n] \to \mathbb{R}$, with elements accessed via $A(i, j)$. Thus, tensors and functions are treated interchangeably, and the same notations are used for both continuous and discrete cases.

To contract tensors, we adopt the Einstein convention, which assumes that a summation occurs along the common indices of the tensors. For instance, for matrices A and B of sizes $m \times n$ and $n \times k$, respectively, their product—contracting the second index of A with the first

index of B—is expressed as:

$$A(i,j)B(j,k) := \sum_{j=1}^{n} A(i,j)B(j,k). \tag{2.1.1}$$

We use bold-face letters such as $\mathbf{x}$ or $\mathbf{j}$ to denote sets of samples or multi-indices. For instance, a dataset is represented as $\mathbf{x} = \{(\mathbf{x}_1^i, \ldots, \mathbf{x}_n^i) : \mathbf{x}_k^i \in \mathcal{X}_k\}_{i\in[N]}$, where subscripts index variables and superscripts index samples. The set of possible values for the k-th variable in $\mathbf{x}$ is represented as $\mathbf{x}_k = \{\mathbf{x}_k^i\}_{i\in[N]}$. For integers $a \leqslant b$, the notation $a\!:\!b$ represents the set $\{a, a+1, \ldots, b\}$, which is useful for describing subsets of variables. For example, $\mathbf{x}_{a:b} = \{(\mathbf{x}_a^i, \ldots, \mathbf{x}_b^i)\}_{i\in[N]}$. We use $\mathbf{x}^i = (\mathbf{x}_1^i, \ldots, \mathbf{x}_n^i)$ to select a single point, and subscript notation may also be applied to specify subsets of variables.

Although all elements in a set are assumed to be distinct, some may share the same values for certain variables. As a result, the sets $\mathbf{x}_k$ and $\mathbf{x}_{a:b}$ may contain fewer than N unique elements. To account for this, we extend the subscript notation to specify the number of unique elements in each set. Specifically, we denote that there are N_k unique elements in $\mathbf{x}_k$ and $N_{a:b}$ unique elements in $\mathbf{x}_{a:b}$.

This notation is further extended to group sets of variables when calling a function. For instance, for a function $f(x_1, \ldots, x_n)$, we may group its variables as $f((x_1, \ldots, x_k), (x_{k+1}, \ldots, x_n))$, or more concisely, as $f(x_{1:k}, x_{k+1:n})$, where $x_{1:k} \in \mathcal{X}_1 \times \cdots \times \mathcal{X}_k$ and $x_{k+1:n} \in \mathcal{X}_{k+1} \times \cdots \times \mathcal{X}_n$. This notation naturally extends to higher-order tensors by grouping variables into multiple sets, as in $f(x_{1:k-1}, x_k, x_{k+1:n})$. When clear from context, such transformations may not be made explicit, and expressions like $f(x_{1:k-1}, x_k, x_{k+1:n})$ and $f(x_{1:k}, x_{k+1:n})$ will be used interchangeably.

Function evaluations on particular samples are denoted by $f(\mathbf{x}^i)$, while for the full dataset we write $f(\mathbf{x})$ to denote the vector $(f(\mathbf{x}^i))_{i\in[N]}$. Generally, notations such as $f(\mathbf{x}_{1:k-1}, \mathbf{x}_k, \mathbf{x}_{k+1:n})$ refer to the tensor obtained by restricting f to the corresponding sets of variables. Specifically, we may access the elements of this tensor by defining an auxiliary function $\bar{f} : [N_{1:k-1}] \times [N_k] \times [N_{k+1:n}] \to \mathbb{R}$ such that

$$\bar{f}(\alpha, \beta, \gamma) := f(\mathbf{x}_{1:k-1}^{\alpha}, \mathbf{x}_k^{\beta}, \mathbf{x}_{k+1:n}^{\gamma}). \tag{2.1.2}$$

2.2 Fundamentals of machine learning

One of the most fundamental challenges across diverse fields is pattern recognition. When the underlying laws or equations governing a process are known, such systems can be studied with greater precision and control. However, in most realistic scenarios, these laws and equations remain unknown. It is therefore crucial to develop techniques capable of recognizing and, more importantly, modeling complex patterns to enable subsequent analysis.

Rather than relying on hard-coded, explicit instructions to capture a system's behavior, a more effective approach employs statistical algorithms that learn patterns from observed data. In particular, a machine learning system models a function $\hat{f}$ that approximates an underlying function f from a given set of N observed samples $\{\mathbf{x}^i\}_{i\in[N]} \subset \mathcal{X}$ by minimizing a loss function ℓ [94, 95]. In practice, the minimization is performed over a restricted hypothesis class $\mathcal{F}$ to control model complexity and prevent overfitting. Many models are thus interpreted as parameterized families $\hat{f}_\theta \in \mathcal{F}$, with learning corresponding to optimization over the parameter space. Since we typically consider parameterized models $\hat{f}_\theta$, we omit explicit reference to the parameters θ unless necessary.

This optimization or learning process is commonly referred to as *training*, and the observed samples constitute the *training dataset*. The ultimate objective is to obtain a model that *generalizes* well, i.e., performs accurately on previously unseen data.

This rather general description reflects the broad scope of ML, which encompasses a wide range of applications. More specifically, ML tasks are typically categorized into three main branches: supervised, unsupervised, and reinforcement learning. Throughout this thesis, we focus on supervised and unsupervised learning and thus omit reinforcement learning from further discussion.

2.2.1 Supervised learning

In supervised learning, the training dataset comprises not only input features $\mathbf{x} \subset \mathcal{X}$ but also corresponding target variables $\mathbf{y} \subset \mathcal{Y}$, yielding N input-output pairs $\{(\mathbf{x}^i, \mathbf{y}^i)\}_{i\in[N]}$, which are assumed to be independently drawn from an unknown distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$. Assuming the existence of a function $f : \mathcal{X} \to \mathcal{Y}$ that governs the relationship between inputs $x \in \mathcal{X}$ and targets $y \in \mathcal{Y}$, via $y = f(x)$, the objective is to learn an approximation $\hat{f}$ by minimizing a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_{\geqslant 0}$ that quantifies the discrepancy between predicted outputs $\hat{f}(x)$ and true targets y.

The ideal goal is to minimize the *expected risk*:

$$\mathcal{R}(\hat{f}) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(\hat{f}(x), y)]. \tag{2.2.1}$$

However, since the distribution $\mathcal{D}$ is unknown and only a finite dataset $(\mathbf{x}, \mathbf{y})$ is available, one typically minimizes the *empirical risk* instead:

$$\widehat{\mathcal{R}}(\hat{f}) = \frac{1}{N}\sum_{i=1}^{N} \ell(\hat{f}(\mathbf{x}^i), \mathbf{y}^i). \tag{2.2.2}$$

This approach is known as Empirical Risk Minimization (ERM).

The choice of loss function ℓ and the interpretation of the model output $\hat{f}(x)$ depend critically on the nature of the output space $\mathcal{Y}$. When $y \in \mathcal{Y}$ takes continuous values, such as real numbers or vectors in $\mathbb{R}^m$, the problem is referred to as *regression*. In this setting, the loss function typically measures geometric distances, with a common choice being the squared ℓ_2 error:

$$\ell(\hat{f}(x), y) = \frac{1}{2}\|\hat{f}(x) - y\|_2^2, \tag{2.2.3}$$

which penalizes deviations between predictions and true values quadratically, often yielding closed-form solutions and convex optimization landscapes.

In contrast, when the target variable y is discrete, taking values in a finite label set, the task is referred to as *classification*. The goal is to assign each input x to one of several distinct categories, such as identifying the digit in a handwritten image or detecting sentiment in a text passage. The simplest instance is binary classification, where $y \in \{0, 1\}$. A naive approach might involve learning a function $\hat{f} : \mathcal{X} \to \mathbb{R}$ and assigning x to class 1 if $\hat{f}(x) > a$ for some threshold a, and to class 0 otherwise. However, this hard-decision scheme is limited, especially in settings with inherent uncertainty or ambiguity.

To address this, many classification models adopt a probabilistic interpretation of the output: instead of returning a deterministic label, the model produces a score $\hat{f}(x) \in [0, 1]$ interpreted

as the estimated probability $\mathrm{P}(y = 1 \mid x)$ that the input belongs to class 1. In cases where $\hat{f}(x)$ is interpreted as a conditional probability distribution, we may denote it by $\hat{p}(\cdot \mid x)$.

In binary classification, a commonly used loss function under this probabilistic view is the binary *cross-entropy*, given by:

$$\ell(\hat{p}(1 \mid x), y) = -\left[y \log \hat{p}(1 \mid x) + (1 - y) \log(1 - \hat{p}(1 \mid x))\right], \tag{2.2.4}$$

which corresponds to the negative log-likelihood of a Bernoulli distribution that is parametrized by $\hat{p}(1 \mid x)$. In multi-label classification, where $y \in [C]$ with $C > 2$ the total number of classes, the model outputs a vector of class probabilities, that is, $\hat{p}(\cdot \mid x) \in [0, 1]^C$, and the cross-entropy loss generalizes to:

$$\ell(\hat{p}(\cdot \mid x), y) = -\log \hat{p}(y \mid x). \tag{2.2.5}$$

Minimizing the average cross-entropy over the dataset is equivalent to performing Maximum Likelihood Estimation (MLE) of the parameters θ, assuming the data is sampled from the true distribution $\mathcal{D}$.

In NNs, for instance, the final layer in a classification model often uses a *softmax* function to convert the output logits into a probability distribution:

$$\hat{p}(y \mid x) = \mathrm{softmax}(z)_y = \frac{\exp(z_y)}{\sum_{k=1}^{C} \exp(z_k)}, \tag{2.2.6}$$

where $z = (z_1, \ldots, z_C)$ are the unnormalized outputs (logits) computed by the network.

2.2.2 Unsupervised learning and density estimation

In unsupervised learning, one only has access to unlabeled inputs $\{\mathbf{x}^i\}_{i \in [N]} \subset \mathcal{X}$ drawn independently from an unknown probability distribution $\mathcal{D}$ over the input space $\mathcal{X}$. The goal is to capture the underlying density $p(x)$ to uncover meaningful structure, patterns, or representations from the data itself.

A central objective in unsupervised learning is to approximate the density $p(x)$ with a model $\hat{p}_\theta(x)$, a task known as *density estimation*. Classical approaches to density estimation rely on strong parametric assumptions. For instance, Gaussian Mixture Models (GMMs) represent $p(x)$ as a convex combination of multivariate Gaussian components, each characterized by its own mean and covariance. While effective in certain scenarios, the expressiveness of such models is limited by their assumptions.

In recent years, *deep generative models* have emerged as powerful tools for flexible density estimation, capable of capturing highly nonlinear and multimodal distributions. These include Variational Autoencoders (VAEs) [96], Generative Adversarial Networks (GANs) [97], normalizing flows [98, 99], and diffusion models [100, 101]. Since the primary application of such models is generative modeling, they typically emphasize efficient or direct sampling. This contrasts with models like RBMs, which allow straightforward evaluation of the density function but rely on costly Markov Chain Monte Carlo (MCMC) methods for sampling. Notably, some models, such as GANs, do not provide direct access to the density function at all and serve solely as sampling mechanisms.

A central focus of this thesis is the modeling of functions, and in particular densities, using TT representations. These models offer advantages in both efficient density evaluation and sample generation, as will be elaborated in Chapter 3. Furthermore, a key benefit is that density

estimation can be performed directly using tensorization techniques, as the one presented in Chapter 4.

Beyond generative modeling, unsupervised learning techniques are widely used for clustering (e.g., k-means [102], spectral clustering [103]), manifold learning (e.g., t-SNE [104], UMAP [105]), and self-supervised learning [106, 107], where the model generates auxiliary labels or tasks to guide the learning process. Despite lacking explicit supervision, unsupervised methods are increasingly central in settings with limited labeled data or where uncovering latent structure is essential.

2.2.3 Illustrative examples

In the following, we present a couple of classic learning tasks, namely polynomial curve fitting and classification with Support Vector Machines (SVMs), which will aid in understanding the TT models discussed in Chapter 3. Additionally, we introduce the basic formulation of common NN models that will appear throughout the thesis.

Polynomial curve fitting

In this simple regression problem, the observed data points $(\mathbf{x}, \mathbf{y}) \in \mathbb{R}^2$ are assumed to be generated by a polynomial function, $f(x) = w_0 + w_1 x + \cdots + w_n x^n$, where the superscripts here denote powers of x. The objective is to learn this polynomial, given that the exact coefficients w and the degree n are unknown, and the samples may be noisy. Accordingly, the model is another polynomial,

$$\hat{f}_{\hat{w}}(x) = \hat{w}_0 + \hat{w}_1 x + \cdots + \hat{w}_m x^m, \tag{2.2.7}$$

which, denoting $x^{0\dots m} = [1, x, \dots, x^m]^\top$ as the vector of powers of x, can be rewritten as

$$\hat{f}_{\hat{w}}(x) = \hat{w}^\top x^{0\dots m}. \tag{2.2.8}$$

This formulation highlights that $\hat{f}_{(\cdot)}(x)$, viewed as a function of the coefficients, is linear in them.

Leaving aside the choice of an appropriate degree m to ensure good generalization and avoid overfitting, which is part of the *model selection* process, suitable coefficients $\hat{w}$ can be obtained by minimizing the squared ℓ_2 loss. Since the error function is quadratic in $\hat{w}$, the minimization yields a unique solution, denoted by $w^\star$, which can be computed in closed form. The resulting polynomial is represented by the function $\hat{f}_{w^\star}(x)$.

Support vector machines

Support vector machines are among the foundational models in machine learning, providing a simple yet powerful illustration of a principle that underlies many ML models, including complex NNs. Originally introduced by Vapnik and collaborators [108], SVMs aim to find a decision boundary that maximizes the margin between classes, a strategy that often leads to better generalization.

Given a dataset $\{(\mathbf{x}^i, \mathbf{y}^i)\}_{i\in[N]}$, with $\mathbf{x}^i \in \mathbb{R}^n$ and $\mathbf{y}^i \in \{-1, +1\}$, the objective is to find a hyperplane $w^\top x + b = 0$ that separates the two classes while maximizing the margin. In the linearly separable case, this leads to the hard-margin SVM formulation:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}\|w\|^2 \\ \text{subject to} \quad & \mathbf{y}^i(w^\top \mathbf{x}^i + b) \geqslant 1 \quad \forall i \in [N]. \end{aligned} \tag{2.2.9}$$

When exact separation is not feasible, the soft-margin SVM introduces slack variables $\xi^i \geqslant 0$ and a regularization parameter $C > 0$ to balance margin maximization and misclassification.

While the primal formulation provides a direct expression of the optimization objective, the dual formulation more explicitly reveals how to construct the optimal hyperplane. The dual problem is typically written as:

$$\begin{aligned} \max_{\alpha \in \mathbb{R}^N} \quad & \sum_{i=1}^{N} \alpha^i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha^i \alpha^j \mathbf{y}^i \mathbf{y}^j (\mathbf{x}^i)^\top \mathbf{x}^j \\ \text{subject to} \quad & \alpha^i \geqslant 0 \quad \forall i \in [N], \quad \sum_{i=1}^{N} \alpha^i \mathbf{y}^i = 0, \end{aligned} \tag{2.2.10}$$

whose solution has the form

$$w^\star = \sum_{i=1}^{N} (\alpha^\star)^i \mathbf{y}^i \mathbf{x}^i, \tag{2.2.11}$$

where $\alpha^\star$ denotes the optimal Lagrange multipliers. This shows that the optimal weight vector lies in the span of the training data. Only those samples for which $(\alpha^\star)^i > 0$ contribute to $w^\star$; these are known as *support vectors*.

However, this corresponds to a linear model, which is often insufficient for more complex tasks. To model nonlinear decision boundaries, input data can be implicitly mapped to a higher-dimensional space $\mathcal{H}$ via a *feature map* or *embedding* $\phi : \mathbb{R}^n \to \mathcal{H}$. The dual formulation then becomes:

$$\max_{\alpha \in \mathbb{R}^N} \sum_{i=1}^{N} \alpha^i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha^i \alpha^j \mathbf{y}^i \mathbf{y}^j \langle \phi(\mathbf{x}^i), \phi(\mathbf{x}^j) \rangle, \tag{2.2.12}$$

where the scalar product in $\mathcal{H}$ is typically replaced by a *kernel function* $K(x, x') = \langle \phi(x), \phi(x') \rangle$. This enables the use of expressive feature spaces without computing $\phi(x)$ explicitly, a technique known as the *kernel trick*.

This formulation makes SVMs one of the most general models, at least from a theoretical standpoint. The key idea is to find expressive embeddings ϕ that lift the data to a space where linear separation becomes possible. This perspective extends even to NN models, where all layers except the final one can be interpreted as a parameterized, learnable, and nonlinear feature map, with the final layer acting as a linear classifier. In SVMs, efficiency is preserved via the kernel trick, which avoids explicit computation of feature maps and instead computes inner products in the embedded space. In contrast, models such as TTs compute these feature maps explicitly. In such cases, it is the TT structure that ensures the efficiency of the model, as will be discussed in Chapter 3.

Neural networks

Inspired by the structure and function of the human brain, neural networks are computational models designed to learn complex relationships between inputs and outputs. Their origins trace back to the perceptron model introduced by Rosenblatt in 1958 [109], with significant developments occurring in the 1980s through Multi-Layer Perceptrons (MLPs) and the backpropagation algorithm [7]. The resurgence of deep learning in the mid-2000s was largely enabled by unsupervised pretraining techniques introduced by Hinton *et al.* [4], leading to the current era of deep learning [95].

A standard feedforward NN with L layers defines a function $\hat{f}_\theta : \mathbb{R}^n \to \mathbb{R}^m$ as a composition of affine transformations and nonlinear activations:

$$\begin{gathered} h^{(0)} = x, \quad h^{(l)} = \sigma(W^{(l)} h^{(l-1)} + b^{(l)}), \quad l \in [L], \\ \hat{f}_\theta(x) = h^{(L)}, \end{gathered} \tag{2.2.13}$$

where $W^{(l)}$ and $b^{(l)}$ are learnable weights and biases, and σ is a nonlinear activation function.

Historically, sigmoidal functions such as the logistic $\sigma(z) = 1/(1 + e^{-z})$ and hyperbolic tangent $\sigma(z) = (e^z - e^{-z})/(e^z + e^{-z})$ were common, but often suffered from vanishing gradients. The ReLU [9], defined as $\sigma(z) = \max\{0, z\}$, is now standard due to its simplicity and favorable optimization properties. Variants such as leaky ReLU [110], ELU [111], and GELU [112] have also been proposed to enhance learning stability.

Training is typically performed via gradient-based optimization methods such as SGD. Given a parameter vector $\theta = [W^{(1)}, b^{(1)}, \ldots, W^{(L)}, b^{(L)}]$, updates follow

$$\theta^{(t+1)} = \theta^{(t)} - \eta \, \nabla_\theta \widehat{\mathcal{R}}(\hat{f}), \tag{2.2.14}$$

where $\eta > 0$ is the *learning rate*. In practice, gradients are computed over mini-batches to reduce variance and improve convergence. More advanced algorithms build on SGD by incorporating momentum, or adaptive gradient scaling. Among the most widely used is the Adam optimizer [113], which leverages moving averages of both gradients and squared gradients.

The above describes the basic *fully connected neural network*, where each neuron is connected to all neurons in the subsequent layer, represented by dense weight matrices $W^{(l)}$. However, much of the power of NNs lies in their flexibility to define architectures tailored to the specific structure of each problem. For instance, CNNs, which can be viewed as a special case of fully connected networks, exploit local stationarity and translational invariance by applying small learnable filters that convolve across the input space. This induces a stronger implicit bias, which helps reduce computational costs and enables faster convergence. Moreover, the expressive capacity of NNs is supported by theoretical results showing that, under mild assumptions, they are *universal function approximators* [114], capable of approximating any continuous function on a compact domain to arbitrary precision given sufficient width.

However, this architectural complexity often comes at the cost of interpretability, rendering NNs effectively black-boxes, with limited transparency into their internal mechanisms. This lack of interpretability impacts several critical concerns in the ML community, including adversarial robustness, fairness, and privacy—an issue that lies at the core of this thesis.

2.3 Privacy in machine learning

Preserving the privacy of users whose data is utilized in data analysis or machine learning processes is centered on safeguarding personal information that is sensitive or unintended for public exposure. Unlike security, which focuses on preventing unauthorized access to data, privacy aims to prevent the inference of personal information not only from access to the data but also from the processes where the data was employed.

Similarly to fields such as cryptography, the study of privacy also involves developing the malicious attacks that are able to recover sensitive data by exploiting all accessible information about a model or its training process.

2.3.1 Privacy attacks

Depending on the assumptions made and the information available about the model, various attacks have been defined for both *black-box* (access limited to input-output pairs) and *white-box* (access to the model's parameters and structure) settings. Some of the most notable attacks include:

Model inversion

Originally introduced by Fredrikson *et al.* in the context of pharmacogenetics [115], this attack seeks to infer statistical properties of individual training samples. In deep learning settings, for instance with an image classifier, the adversary searches the input space for an image that maximizes the model's confidence in a given class. This can be framed as an optimization problem solved via gradient descent. When the model is overfitted, the optimized image may closely resemble an actual training sample. This attack operates under a black-box setting, requiring only query access to the model.

Property inference

In contrast to model inversion, the attack proposed by Ateniese *et al.* [116] targets global properties of the training data rather than individual samples. The adversary trains multiple shadow models with and without a given property P in the training data, using the same architecture as the target model. A meta-classifier is then trained on the parameters of these shadow models to predict the presence of property P. When provided with the parameters of the target model, the meta-classifier can infer whether property P holds in its training data. This is a white-box attack, as it relies on access to the model's internal parameters.

Membership inference

Proposed by Shokri *et al.* [117], this attack determines whether a specific data point was used in training the target model. The adversary trains multiple shadow models with and without the target sample, then uses their outputs to train a meta-classifier that learns to distinguish between members and non-members of the training set. Once trained, the meta-classifier is used to infer the membership status of the target sample with respect to the model. Like model classification, this attack assumes white-box access.

Reconstruction

Reconstruction attacks aim to recover actual training samples, typically assuming white-box access. In Ref. [118], the adversary is given the model parameters and the full training set excluding one sample. By training multiple shadow models—each omitting a different sample—and using different initializations or hyperparameters, a meta-classifier is trained to reconstruct the missing data point.

Other strategies, such as those in Refs. [119, 120], frame reconstruction as a system of equations derived from the Karush–Kuhn–Tucker (KKT) conditions, which characterize optima of models trained via SGD with cross-entropy loss. By identifying training sets that satisfy these conditions given the known parameters, the adversary can reconstruct multiple training samples with high fidelity in realistic scenarios.

The types of attacks considered in this thesis may be categorized under *property inference*. However, the shadow model-based attack methodology employed—along with the proposed defense—can be readily extended to other white-box scenarios, such as *membership inference* and *reconstruction*, which similarly aim to extract information from model parameters. The attacks developed here are grounded in the observation that gradient-based training drives model parameters toward distinct regimes when trained on different datasets, an effect that becomes more pronounced when the datasets differ in global properties or overall distribution. This topic will be explored in detail in Chapter 5.

2.3.2 Defense mechanisms

Given the diversity of privacy-related attacks, a variety of concepts and defense mechanisms have been developed to address specific scenarios. Among these, differential privacy [45] stands out as a widely adopted framework due to its rigorous approach to assessing privacy risks and designing mitigation strategies. Differential privacy measures the likelihood that an attacker can determine whether a specific user's data was included in a statistical process. Specifically,

Definition 2.3.1. A randomized algorithm $\mathcal{A}$ is (ε, δ)-*differentially private* (abbreviated (ε, δ)-DP), with $\varepsilon, \delta \geqslant 0$, if for all measurable subsets $\mathcal{S} \subseteq \text{Range}(\mathcal{A})$ and for all datasets $\mathbf{x}, \mathbf{x}'$ such that $d_H(\mathbf{x}, \mathbf{x}') \leqslant 1$, where $d_H(\cdot)$ denotes the Hamming distance, it holds that

$$\mathrm{P}[\mathcal{A}(\mathbf{x}) \in \mathcal{S}] \leqslant \exp(\varepsilon)\mathrm{P}[\mathcal{A}(\mathbf{x}') \in \mathcal{S}] + \delta, \tag{2.3.1}$$

where the probability space is over the randomness of the mechanism $\mathcal{A}$. If $\delta = 0$, we say that $\mathcal{A}$ is ε-*differentially private* (abbreviated ε-DP).

When $\delta = 0$, Eq. (2.3.1) can be rewritten as:

$$\log\left(\frac{\mathrm{P}[\mathcal{A}(\mathbf{x}) \in \mathcal{S}]}{\mathrm{P}[\mathcal{A}(\mathbf{x}') \in \mathcal{S}]}\right) \leqslant \varepsilon. \tag{2.3.2}$$

This metric, akin to the Kullback-Leibler divergence, quantifies privacy protection through the parameter ε. It enables the design of random mechanisms by introducing a precisely calibrated amount of noise to achieve a desired privacy level ε, based on the *sensitivity* of the function being protected [46, 47].

Several methods have been proposed to train DNNs with DP. These include adding noise to input data [121] or to gradients during each step of gradient descent [48]. The latter, known as DP-SGD, has gained prominence as it ensures DP at the training level, making it independent of data acquisition or model deployment processes. However, DP-SGD is computationally expensive and negatively impacts model performance, including accuracy and fairness [122]. To address these challenges, Ref. [123] adapted DP-SGD to a training scheme involving low-rank updates to weight matrices, as in Ref. [124]. Additionally, pruning techniques have been proposed as an alternative to achieve DP without significantly affecting the training process [125, 126].

Nevertheless, the practical application of DP faces certain limitations. For instance, while the framework provides theoretical bounds on privacy loss, its practical implications are often unclear, requiring reliance on experimental evaluation [127]. Furthermore, by its definition, DP is primarily designed to protect against membership inference attacks and does not inherently protect against property inference attacks.

Due to these practical limitations, DP is not always a viable solution for preserving model performance while ensuring privacy. One of the main contributions of this thesis is to demonstrate that, thanks to the well-defined characterization of the gauge freedom in TN models, white-box information can be obfuscated. This effectively reduces the privacy threat to what can be inferred solely from the model's outputs. In Chapter 5, we present a series of experiments showing that, in practice, attacks on TN models perform similarly regardless of whether the adversary has white-box or black-box access.

Additional strategies have been explored to enhance privacy in production-level models while minimizing the impact on accuracy. One example is training models with reinforcement learning with human feedback to reduce the memorization of sensitive information [42]. However, studies have shown that such approaches remain vulnerable to exploitation, raising significant privacy concerns [43, 44].

2.4 Introduction to tensor networks

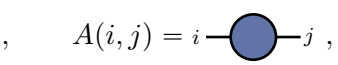
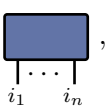
In this section, we introduce the concept of a tensor network and the basic properties that are relevant in the context of machine learning. For a more in-depth analysis, we refer the reader to [49–51, 60].

Tensor networks are low-rank decompositions of high-dimensional tensors. They were originally developed in condensed matter physics for their ability to represent quantum many-body states through efficient and interpretable descriptions, supporting both analytical and practical studies. More broadly, the efficient representations they offer enable the compression of tensors and functions in various settings, leading to applications in signal processing [128], high-dimensional integral estimation [129], and, as in this work, machine learning [58, 59].

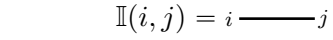
At their core, TNs consist of two main components: the *tensors* and the *contraction* operation. Tensors are an extension of vectors and matrices to higher dimensions. They can be visualized as collections of indexed numbers arranged in multi-dimensional arrays. In general, an order-n tensor with dimensions $d_1 \times \cdots \times d_n$ is an element of the tensor product vector space $\bigotimes_{k\in[n]} \mathbb{C}^{d_k} \simeq \mathbb{C}^{\times_{k\in[n]} d_k}$. Its elements are denoted by $T_{i_1\cdots i_n}$, where each $i_k \in [d_k]$. As noted earlier, we typically treat tensors as discrete functions $T : [d_1] \times \cdots \times [d_n] \to \mathbb{C}$, accessing elements via $T(i_1, \ldots, i_n)$.

Although the methods presented in this thesis and the general formulation of TNs apply to complex tensors, the applications of interest in ML generally involve real-valued functions. Accordingly, unless explicitly stated otherwise, we will typically consider real tensors $T : [d_1] \times \cdots \times [d_n] \to \mathbb{R}$.

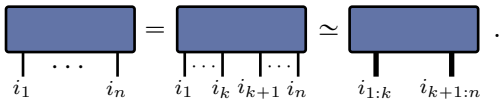
The literature on TNs also presents a useful and practical graphical notation, where tensors are represented as the nodes of a graph with edges corresponding to their indices. In this notation, vectors, matrices and arbitrary tensors are depicted as follows:

$v(i) =$  , $A(i,j) =$  , $T(i_1,\ldots,i_n) =$  , (2.4.1)

and the identity matrix is simply represented by a line:

$\mathbb{I}(i,j) =$  (2.4.2)

In certain contexts, the direction of the edges in a tensor, such as pointing upwards or downwards, is used to represent different properties. In our setting, however, we impose no such restrictions and allow edges to be freely moved and reordered. A common transformation in this framework is to *group* or *ungroup* sets of indices for convenience, rearranging the entries of a multi-dimensional array to represent a vector, a matrix, or a higher-order tensor. When indices of a tensor are grouped into two sets, as in $T(i_{1:k}, i_{k+1:n})$, the result is referred to as the k-th *unfolding matrix* or *matricization* of the tensor, depicted as



(2.4.3)

This representational flexibility highlights that tensors are fundamentally linear maps between tensor product spaces, analogous to how matrices represent linear maps between vector spaces. It thus provides a natural framework for using matrix factorization techniques as subroutines for decomposing higher-order tensors.

The contraction of indices (within the same tensor or between different tensors) generalizes the scalar product of vectors—that is, the summation of products over corresponding elements along the specified axes. For vectors u and v, the scalar product is written as

$$u(i)v(i) = w, \tag{2.4.4}$$

while for matrices A and B, their product is expressed as

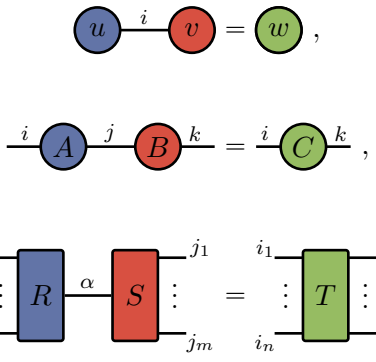
$$A(i,j)B(j,k) = C(i,k), \tag{2.4.5}$$

and, more generally, the contraction of tensors R and S along a common index α is given by

$$R(i_1,\ldots,\alpha,\ldots,i_n)\,S(j_1,\ldots,\alpha,\ldots,j_m) = T(i_1,\ldots,i_n,j_1,\ldots,j_m), \tag{2.4.6}$$

where T is an order-$(n+m-2)$ tensor. In all cases, we adopt the Einstein summation convention, where summation over repeated indices is implied and therefore omitted.

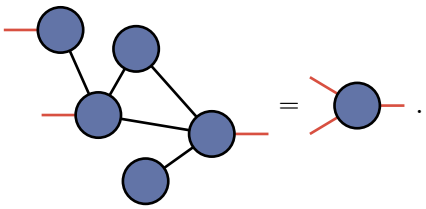
In the graphical notation, these contractions are represented by connecting the corresponding edges of the tensors:



(2.4.7)

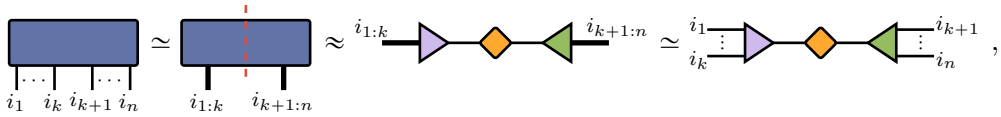
Contraction enables connecting nodes to form graphs, with the only requirement being that the nodes must have the same dimensions along the connected axes. By contracting all the

connected edges in the graph, one can then contract a whole TN to obtain a single tensor that preserves the remaining dangling edges:



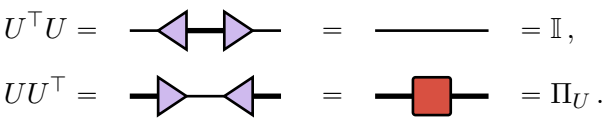
(2.4.8)

Conversely, one can construct a TN by decomposing tensors using various established methods [130]. As previously noted, a common approach involves reshaping tensors into matrices by grouping and ungrouping indices, followed by applying matrix factorizations such as the SVD. This decomposition provides the optimal low-rank approximation when singular values are truncated, as elaborated in Section 2.5. In graphical notation, the decomposition $T = USV^\top$ is represented as



(2.4.9)

where triangular nodes denote isometries, satisfying the properties



(2.4.10)

Nevertheless, although such factorizations can significantly reduce the number of elements needed to represent high-order tensors, evaluating their practical efficiency also requires considering the computational complexity of contracting the TN. In this thesis, we focus on tree-like TNs, which are efficient to contract. However, it is important to note that contracting TNs with closed loops, such as PEPS, is known to be #P-hard [131], and practical implementations must therefore rely on approximate schemes. More generally, finding optimal contraction paths in arbitrary graphs is also #P-hard [132].

2.4.1 Tensor trains

The most prominent TNs, due to their simplicity and efficiency, are the one-dimensional models known as matrix product states [60] or tensor trains [61]. The former term, used in quantum many-body physics, refers to how the entries of the tensor representing a quantum state are obtained as products of matrices. In classical tensor notation, a general quantum many-body state can be written as

$$|\Psi\rangle = \sum_{i_1,\ldots,i_n=0}^{d-1} T_{i_1,\ldots,i_n} |i_1 \cdots i_n\rangle , \tag{2.4.11}$$

where $\{|i_1 \cdots i_n\rangle\}_{i_k=0}^{d-1}$ denotes the canonical basis of the tensor product space, expressed in bra-ket notation, as is customary in the quantum physics literature. An MPS representation of this state imposes the structure

$$T_{i_1,\ldots,i_n} = \mathrm{Tr}[A_1^{i_1} A_2^{i_2} \cdots A_n^{i_n}], \tag{2.4.12}$$

where each $A_k^{i_k}$ is an $r_{k-1} \times r_k$ matrix for every $i_k \in [d]$. This assumes that the dimensions r_k of the matrices are compatible with their neighbors to allow matrix multiplication, although these dimensions need not be uniform across all k. Similarly, while we fix all indices i_k to range over $[d]$, they may in general belong to different sets $i_k \in [d_k]$. A particularly relevant case arises when all dimensions d_k and r_k are equal for all k, corresponding to Translationally Invariant (TI) states, where all tensors A_k coincide.

This formulation, involving a product of matrices followed by a trace, corresponds to a state with Periodic Boundary Conditions (PBC). Conversely, a state with Open Boundary Conditions (OBC) can be represented as

$$T_{i_1,\ldots,i_n} = A_1^{i_1} A_2^{i_2} \cdots A_n^{i_n}, \tag{2.4.13}$$

where $A_1^{i_1}$ are $1 \times r_1$ matrices and $A_n^{i_n}$ are $r_{n-1} \times 1$ matrices, which can be viewed as row and column vectors, respectively.

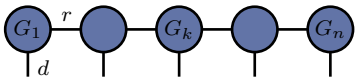
Throughout this thesis, we adopt the TT formulation, as it is standard in numerical analysis and numerical linear algebra communities. Accordingly, we focus on representing functions in TT form in general settings, rather than exclusively considering quantum states, using the associated notation. Specifically, we define the TT format as follows:

Definition 2.4.1. A function $f : [d_1] \times \cdots \times [d_n] \to \mathbb{R}$ admits a tensor train representation with ranks $r_1, \ldots, r_{n-1}$ if there exist functions $G_1 : [d_1] \times [r_1] \to \mathbb{R}$, $G_k : [r_{k-1}] \times [d_k] \times [r_k] \to \mathbb{R}$ for all $k \in \{2, \ldots, n-1\}$, and $G_n : [r_{n-1}] \times [d_n] \to \mathbb{R}$, referred to as *cores*, such that

$$f(i_1, \ldots, i_n) = G_1(i_1, \alpha_1) G_2(\alpha_1, i_2, \alpha_2) \cdots G_{n-1}(\alpha_{n-2}, i_{n-1}, \alpha_{n-1}) G_n(\alpha_{n-1}, i_n) \tag{2.4.14}$$

for all $(i_1, \ldots, i_n) \in [d_1] \times \cdots \times [d_n]$. Note that the right-hand side of the expression above has, per Einstein's convention, implicit sums over the indices $\{\alpha_1, \ldots, \alpha_{n-1}\}$.

Graphically, TTs are represented as



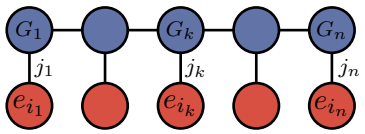
, (2.4.15)

where subscripts for the *ranks* r and *input dimensions* d—also referred to as *bond* and *physical* dimensions, respectively, in the physics community—are omitted when assumed equal across all sites. When r is polynomial in d, this representation achieves an exponential reduction in the number of elements required to represent f, from $O(d^n)$ to $O(ndr^2)$.

This structure also enables efficient computation. For instance, if f is expressed in the tensor product basis $\bigotimes_{k\in[n]}\{e_1, \ldots, e_d\}$, where $\{e_1, \ldots, e_d\}$ denotes the canonical basis of $\mathbb{R}^d$, the elements of f can be accessed via

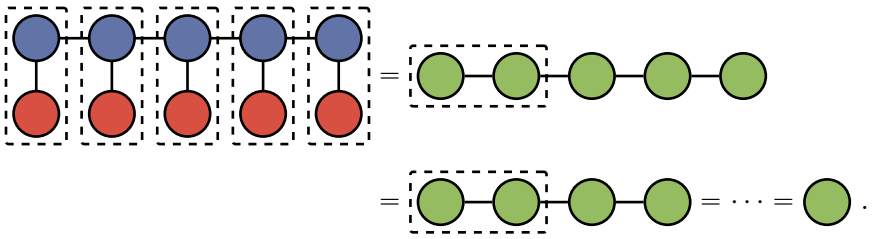
$$f(i_1, \ldots, i_n) = f(j_1, \ldots, j_n) e_{i_1}(j_1) \cdots e_{i_n}(j_n), \tag{2.4.16}$$

or equivalently,



(2.4.17)

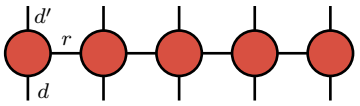
which can be efficiently contracted in $O(nr^2(d+r))$ operations via the following steps:



(2.4.18)

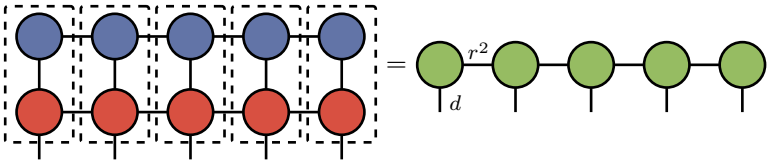
Note that the contractions inside the boxes in the first term of Eq. (2.4.18) involve distinct tensors and can therefore be executed in parallel, offering additional computational savings in practice.

In a similar fashion to how one defines the TT representation of a function f, which can be seen as an element of the exponentially large vector space $\bigotimes_{k\in[n]} \mathbb{R}^{d_k}$, one can define TT representations of matrices, or linear maps between vector spaces $\bigotimes_{k\in[n]} \mathbb{R}^{d_k}$ and $\bigotimes_{k\in[n]} \mathbb{R}^{d'_k}$. This model is known as Matrix Product Operator (MPO) [133] or Tensor Train Matrix (TTM) [61], and is represented as



(2.4.19)

Correspondingly, a TT vector can be transformed via a TTM by contracting each TT edge with the corresponding *input* edge of the TTM:



(2.4.20)

This contraction requires only $O(nd^2r^4)$ operations, assuming uniform TT and TTM ranks r and dimensions d. The result of this contraction is another TT vector with ranks r^2.

A central question addressed in this thesis is how to obtain a TT representation of a desired function. From a machine learning perspective, this can be framed as an optimization problem, solvable via gradient descent schemes. Notably, a major advantage of TTs is that, at least in low-dimensional regimes, the best low-rank TT approximation of a function can be efficiently obtained through iterative application of SVD. In contrast, other low-rank factorizations such as the CP decomposition are known to be NP-hard to compute exactly [130, 134, 135]. A further comparison can be made with the Tensor Ring (TR) decomposition [136], the PBC counterpart of TTs, for which minimal ranks cannot be obtained via iterative SVDs [137]. In Chapter 4, we introduce a *tensorization* mechanism that enables TT approximation in high-dimensional settings, as commonly encountered in ML applications.

2.4.2 Representation of continuous functions

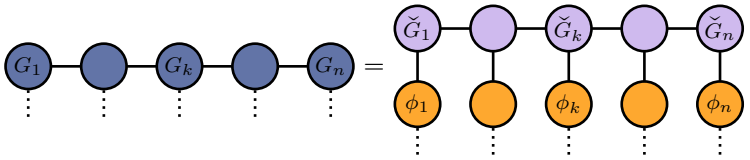
Although the TT format is originally defined for discrete functions, it naturally extends to the continuous case, where $f : \mathcal{X}_1 \times \cdots \times \mathcal{X}_n \to \mathbb{R}$ with $\mathcal{X}_1, \ldots, \mathcal{X}_n \subset \mathbb{R}$; typically, $\mathcal{X}_k = [u_k, v_k]$, with $u_k, v_k \in \mathbb{R}$, for all $k \in [n]$. In this setting, it is assumed that f can be expressed as an expansion in a basis formed by the tensor product of one-dimensional vector-valued functions $\phi_k(x) = (\phi_k^{i_k}(x))_{i_k \in [d_k]}$, referred to as the *embedding functions*. Thus, f is represented as

$$f(x_1, \ldots, x_n) \approx \check{f}(i_1, \ldots, i_n)\phi_1(i_1, x_1) \cdots \phi_n(i_n, x_n), \tag{2.4.21}$$

where $\phi_k(i_k, x_k) = \phi_k^{i_k}(x_k)$, and $\check{f} : [d_1] \times \cdots \times [d_n] \to \mathbb{R}$ is the tensor of coefficients. The TT representation of $\check{f}$ is therefore defined by cores $\check{G}_1, \ldots, \check{G}_n$ satisfying

$$G_k(\alpha_{k-1}, x_k, \alpha_k) = \check{G}_k(\alpha_{k-1}, i_k, \alpha_k)\phi_k(i_k, x_k), \tag{2.4.22}$$

where the cores $G_k : [r_{k-1}] \times \mathcal{X}_k \times [r_k] \to \mathbb{R}$ are also interpreted as continuous functions, leading to the functional TT representation of f. In this context, the input dimensions d_k of each core $\check{G}_k$ are referred to as the *embedding dimensions*. This functional form is depicted as



, (2.4.23)

where the dotted edges indicate continuous axes.

Notably, this continuous version naturally reduces to the discrete case by adopting the embedding functions $\phi_k^{i_k}(x_k) = \delta_{i_k x_k}$ for $i_k, x_k \in [d_k]$, with $\delta_{i_k x_k}$ denoting the Kronecker delta. Thus, ϕ maps the elements of $[d_k]$ to the canonical basis vectors of $\mathbb{R}^{d_k}$, $\{e_1, \ldots, e_{d_k}\}$. We refer to this embedding as the *trivial embedding*.

Besides this functional approach, another common method for representing continuous functions in TT form is by discretizing their domain, a representation commonly referred to as *quantics-TT* [138, 139]. Specifically, let f be an m-dimensional continuous function with input domains $\mathcal{X}_k = [u_k, v_k]$, with $u_k, v_k \in \mathbb{R}$, for all $k \in [m]$. We discretize each variable x_k into b^l steps, where b is the base and l the level of precision. This yields a discrete version of f, denoted $f^{b,l} : (\{0, \ldots, b-1\}^l)^m \to \mathbb{R}$, defined by

$$f^{b,l}(x_{1,1}, \ldots, x_{1,l}, \ldots, x_{m,1}, \ldots, x_{m,l}) := f(x_1, \ldots, x_m), \tag{2.4.24}$$

where

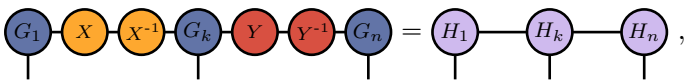
$$x_k = (v_k - u_k)\left(\sum_{i=1}^{l} x_{k,i} b^{-i} + y_k\right) + u_k, \tag{2.4.25}$$

and $y_k \in [0, b^{-l}]$ accounts for the residual approximation error introduced by discretization.

Although the domain is $\{0, \ldots, b-1\}$ rather than $[b]$, we may shift it by one unit, defining $\bar{f}^{b,l} : ([b]^l)^m \to \mathbb{R}$ as $\bar{f}^{b,l}(x) := f^{b,l}(x - \mathbf{1})$, where $\mathbf{1}$ denotes the all-ones vector. For simplicity, this distinction is typically omitted. In this setting, we take $d = b$ as the input dimension and $n = ml$ as the total number of variables.

2.4.3 Canonical forms

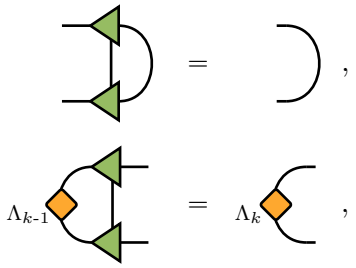
Another central aspect that characterizes TNs is the freedom in choosing the tensors that compose the network while representing the same overall tensor. This is evident by introducing invertible matrices and their inverses along all connected edges, thereby redefining the tensors of the TN as



(2.4.26)

where both sets of cores G_k and H_k result in the same TT. Indeed, as established by the Fundamental Theorem of MPS [60], this is the only transformation that allows for the reparameterization of a TT model.

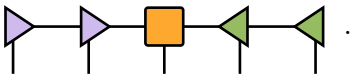
This representational freedom, commonly known as gauge freedom, opens the door to fixing specific parameterizations that are beneficial for various purposes, referred to as *canonical forms*. In this regard, various strategies exist to fix the gauge in a TT. One of the most common, as described in Ref. [60], imposes the following conditions on each core G_k:



(2.4.27)

where $\Lambda_0 = \Lambda_n = 1$, and Λ_k for $k \in [n-1]$ are $r_k \times r_k$ diagonal, positive, full-rank matrices. Additionally, if the TT is normalized (i.e., represents a unit vector), then $\mathrm{Tr}[\Lambda_k] = 1$.

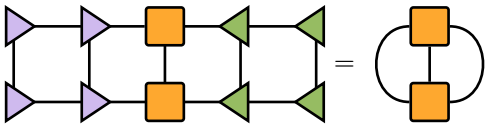
This canonical form can be obtained by successively applying the SVD from right to left along the chain, yielding a representation in which all cores except G_1 are isometries. This is typically referred to as the *right canonical form*. Analogously, one can define the *left canonical form* by reversing the direction, leaving G_n as the sole non-isometric tensor. Finally, if a central core G_k is the only non-isometric tensor, with the surrounding cores on both sides satisfying the canonical conditions, the structure is referred to as a *mixed canonical form*, illustrated as



(2.4.28)

In this setting, the non-isometric tensor, depicted as a yellow square node in the graphical representation, is commonly referred to as the Orthogonality Center (OC).

This canonical structure is particularly useful for simplifying the analysis of TN properties, as most of the representational freedom is fixed. It also facilitates computation, since contractions of isometric tensors with their transposes can be readily omitted. For example, the squared norm of the TT reduces to



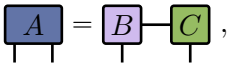
(2.4.29)

In addition to this canonical form, others have been proposed, such as the Vidal form [140] and the minimal canonical form [141]. One of the main contributions of this thesis is the introduction of a new holomorphic canonical form, derived directly from the entries of the tensor, as will be detailed in Chapter 5. Unlike SVD-based forms, which retain an $\mathrm{O}(1)^{\times r}$ freedom on each bond, this representation is unique. Building on this idea, Ref. [142] introduced other canonical forms within a more general framework.

2.5 Matrix factorizations and tensor train decompositions

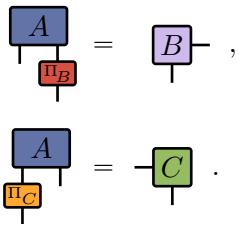
As introduced in the previous section, one of the main goals of this thesis is to learn TN models via low-rank decompositions, aiming to overcome the limitations of gradient descent optimization algorithms. Due to the one-dimensional structure of TTs, a natural approach to decompose tensors in this form is through the sequential application of matrix factorizations.

In particular, we seek a decomposition of a rank-r matrix $A \in \mathbb{R}^{m\times n}$ as the product of two matrices such that



(2.5.1)

with B an $m \times r$ matrix and C an $r \times n$ matrix. In this representation, the columns of B span the column space of A, and the rows of C span the row space of A. When such a decomposition exists, there must also be projections Π_B and Π_C that yield the B and C components of A. That is,



(2.5.2)

In the following, we present techniques used to decompose matrices in this way—namely singular value decomposition, sketching, and cross interpolation—and introduce their applicability to TT decompositions.

2.5.1 Singular value decomposition

The singular value decomposition is arguably the most important matrix factorization in numerical linear algebra. Its theoretical roots can be traced to the work of Eugenio Beltrami and Camille Jordan in the 1870s, while its modern formulation and computational techniques were primarily developed in the 20th century [143]. The SVD decomposes any matrix $A \in \mathbb{R}^{m\times n}$ into the product

$$A = U\Sigma V^\top = \quad , \tag{2.5.3}$$



where $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ are orthogonal matrices, and $\Sigma \in \mathbb{R}^{m\times n}$ is diagonal with non-negative entries $\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant 0$ called the *singular values* of A. Although this decomposition yields $B = U$ and $C = \Sigma V^\top$ implicitly, the corresponding projections can be constructed as $\Pi_B = V\Sigma^{-1}$ and $\Pi_C = U^\top$.

Geometrically, the columns of V, known as the *right singular vectors*, identify a set of orthogonal directions in the input space along which A acts most distinctly, while the columns of U, the *left singular vectors*, describe the corresponding output directions. The magnitudes of the singular values indicate the strength of each direction's contribution to the image of A. Thus, the SVD provides a decomposition into orthogonal modes representing the most significant linear combinations of rows and columns, with respect to how A maps one space to another.

This offers a natural mechanism for low-rank matrix approximation. Specifically, truncating the decomposition to the r largest singular values yields the best rank-r approximation of A in both the spectral and Frobenius norms:

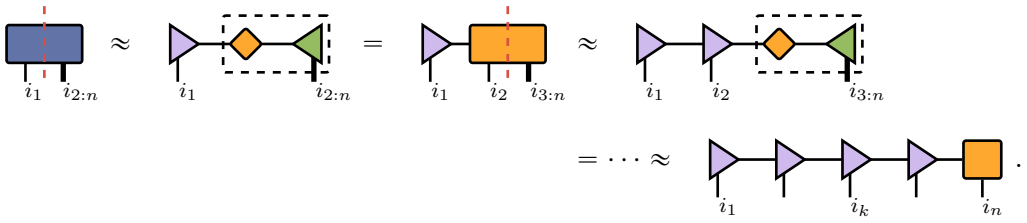
Theorem 2.5.1 (Eckart–Young–Mirsky [144])**.** *Let $A_r = U\Sigma_r V^\top$ be the rank-r truncated SVD of A, where Σ_r contains only the first r singular values on the diagonal and zeros elsewhere. Then*

$$\min_{\operatorname{rank}(B)\leqslant r} \|A - B\|_{\mathrm{op}} = \|A - A_r\|_{\mathrm{op}} = \sigma_{r+1}, \tag{2.5.4}$$

and

$$\|A - A_r\|_F = \left(\sum_{k>r} \sigma_k^2\right)^{1/2}. \tag{2.5.5}$$

When the order n of a given tensor is small, it is common to apply the (truncated) SVD iteratively to decompose the tensor into TT form, as



(2.5.6)

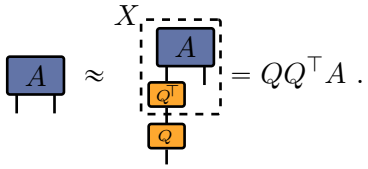
However, this method becomes infeasible for high-order tensors, as the corresponding matricizations have dimensions $d^k \times d^{n-k}$, and computing the SVD becomes exponentially expensive. Specifically, computing the full SVD of an $m \times n$ matrix requires $O(mn\min(m,n))$ operations, while a truncated SVD retaining r singular values can be computed in $O(mnr)$. Consequently, for an order-n tensor with input dimension d and rank r, the overall cost scales as $O(d^n r)$. To mitigate this exponential complexity, it is necessary to first reduce the dimensionality of the matrix before applying the SVD, which can be achieved through sketching techniques.

2.5.2 Sketching techniques

A common approach to reduce dimensionality prior to SVD is to use random projections to efficiently capture the dominant subspace of the matrix. Given an $m\times n$ matrix A of rank r, with $r \ll \min(m,n)$, the goal is to construct a projection that returns m' rows spanning the original

row space of A. Various strategies can be employed, but the most general involves random Gaussian (or orthogonal) projections, often referred to as *sketches*. This idea is partly inspired by the Johnson–Lindenstrauss lemma [145], which states that a set of n points in a high-dimensional Euclidean space can be embedded into an m'-dimensional subspace, where m' depends logarithmically on n, while approximately preserving pairwise distances. Sketching techniques based on this principle have been used to accelerate numerous linear algebra routines [146–148].

For instance, the randomized SVD (rSVD), developed by Halko, Martinsson, and Tropp [146], seeks a projection $Q \in \mathbb{R}^{m\times m'}$, where $m' = r + s$ and s is the *oversampling parameter*, such that



(2.5.7)

The smaller matrix $X = Q^\top A$ is then factored using standard SVD:

$$X = \tilde{U}\Sigma V^\top, \tag{2.5.8}$$

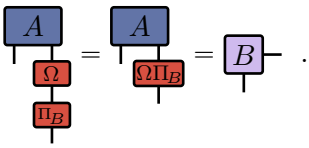
and the final approximation is

$$A \approx U\Sigma V^\top, \tag{2.5.9}$$

with $U = Q\tilde{U}$.

To find Q, one first sketches A with a random Gaussian matrix $\Omega \in \mathbb{R}^{n\times(r+s)}$, yielding $Y = A\Omega$. With high probability, $Y \in \mathbb{R}^{m\times(r+s)}$ spans the column space of A up to an error controlled by the tail singular values. An orthonormal basis of the columns of Y, obtained via QR decomposition, gives Q.

In general, sketching A requires only multiplying it by Ω. To capture the column space of A, one can compute the SVD of $A\Omega = U(\Sigma V^\top) = BC$ and then project again using the corresponding Π_B, i.e.,



(2.5.10)

This sketching strategy has been extended to decompose tensors into TT form. Shi *et al.* [149] proposed a novel method where each TT core is computed by solving an equation formulated using the column spaces of subsequent matricizations of the tensor, which are derived from random sketches. However, projecting an $m \times n$ matrix A using $\Omega \in \mathbb{R}^{n\times n'}$ requires $O(mnn')$ operations, which also becomes infeasible for high-order tensors. To reduce storage requirements, random projections can be constructed via the Khatri–Rao product [150], though this does not alleviate time complexity.

Building on the scheme proposed in [149], Hur *et al.* [89] introduced a method for approximating an empirical distribution $\hat{p}_{\mathrm{E}}$ in TT format to recover the underlying density p. Their algorithm, Tensor Train via Recursive Sketching (TT-RS), is efficient when the sample size N is not too large, since random projections scale linearly with N. However, accurately approximating p requires N to grow exponentially with dimension n, limiting this approach in

high-dimensional settings. Notably, TT-RS can be applied directly to continuous functions, avoiding the need for explicit quantization by utilizing continuous embeddings. This framework has been further extended, for example, by incorporating kernel density estimation to smooth the empirical distribution, as shown in [151], and by adapting the approach to hierarchical tensor structures [152, 153].

2.5.3 Cross interpolation

While the SVD implicitly identifies vectors spanning the column and row spaces, Cross Interpolation (CI) [154], also known as pseudoskeleton [155] or CUR decomposition [156], reconstructs a matrix using r selected columns and r selected rows. Assuming a matrix A has rank r, there exist r linearly independent columns and rows such that the full matrix can be recovered from their linear combinations. Specifically, for a matrix $A \in \mathbb{R}^{m\times n}$, given index sets $\mathbf{i} \subset [m]$ for rows and $\mathbf{j} \subset [n]$ for columns, A can be decomposed as

$$A = A([m], \mathbf{j})A(\mathbf{i}, \mathbf{j})^{-1}A(\mathbf{i}, [n]), \tag{2.5.11}$$

where $A([m], \mathbf{j})$ and $A(\mathbf{i}, [n])$ are the submatrices formed by the selected columns and rows, respectively, and $A(\mathbf{i}, \mathbf{j})$ is the intersection submatrix.

While this can be exact for rank-r matrices, the approximation error for higher-rank matrices is typically larger compared to SVD. The optimality of SVD stems from its selection of orthogonal input directions where A acts most distinctly. Similarly, CI is more accurate when $A(\mathbf{i}, \mathbf{j})$ has maximal volume, i.e., is as close to orthogonal as possible. This is formalized in the following:

Theorem 2.5.2 (Goreinov–Tyrtyshnikov–Zamarashkin [155])**.** *Let A have rank r, and let $A(\mathbf{i}, \mathbf{j})$ be a submatrix of maximal volume. Then*

$$\|A - A([m], \mathbf{j})A(\mathbf{i}, \mathbf{j})^{-1}A(\mathbf{i}, [n])\|_2 \leqslant (r + 1)\sigma_{r+1}. \tag{2.5.12}$$

Although this decomposition introduces greater approximation error, its primary advantage lies in its efficiency. If suitable sets of rows and columns are available such that the intersection matrix has large volume, the decomposition involves only selecting the corresponding submatrices—an operation that can be performed in constant time—and computing the (pseudo)inverse of $A(\mathbf{i}, \mathbf{j})$, which requires $O(r^3)$ operations. In practice, an additional step is typically included to identify appropriate index sets $\mathbf{i}$ and $\mathbf{j}$ using the `maxvol` algorithm. As finding the maximum-volume submatrix is an NP-hard combinatorial problem, greedy algorithms have been developed to approximate the solution efficiently while maintaining practical performance [157].

Building on this idea, it was shown that the iterative application of CI can decompose a high-order tensor into TT format [90, 91], in a manner analogous to the SVD-based decomposition in Eq. (2.5.6). This approach is commonly referred to as Tensor Train via Cross Interpolation (TT-CI). However, as with the SVD, the exponential growth in the size of tensor matricizations with increasing order renders this direct approach infeasible in high-dimensional settings. To address this limitation, a parallel version of TT-CI tailored for high-dimensional scenarios was introduced in [129].

This method begins with an initial guess for the sets of rows and columns—referred to as *pivots* or *interpolation sets*—for all tensor matricizations, and constructs a preliminary TT

approximation. Specifically, for an order-n tensor $T \in (\mathbb{R}^d)^{\otimes n}$, given a set of N multi-indices $\{\mathbf{i}^\alpha\}_{\alpha\in[N]} \subset [d]^n$, the TT cores are defined as:

$$\begin{aligned} G_1(i_1, \alpha_1) &= T(i_1, \mathbf{i}^{\alpha_1}_{2:n}), \\ G_k(\alpha_{k-1}, i_k, \alpha_k) &= T(\mathbf{i}^{\beta_{k-1}}_{1:k-1}, \mathbf{i}^{\alpha_{k-1}}_{k:n})^{-1} T(\mathbf{i}^{\beta_{k-1}}_{1:k-1}, i_k, \mathbf{i}^{\alpha_k}_{k+1:n}), \quad k \in \{2, \dots, n-1\}, \\ G_n(\alpha_{n-1}, i_n) &= T(\mathbf{i}^{\beta_{n-1}}_{1:n-1}, \mathbf{i}^{\alpha_{n-1}}_{n})^{-1} T(\mathbf{i}^{\beta_{n-1}}_{1:n-1}, i_n), \end{aligned} \tag{2.5.13}$$

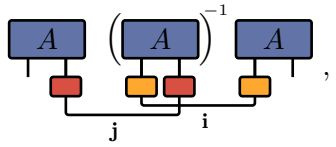
yielding a TT format with all ranks equal to N. This approximation is then refined through iterative sweeps along the TT chain, optimizing the pivots at each split. Additionally, the method can incorporate rank-revealing decompositions, such as the SVD, as subroutines to reduce the initial rank N to a more accurate rank r, achieving a desired approximation accuracy.

Although originally developed for tensor decomposition, TT-CI can also be applied to continuous functions by quantizing them [139]. Promising results using this approach were demonstrated in [142], where TT-CI was employed across a broad range of tasks. This study highlighted the critical role of selecting an effective initial set of pivots, as poor choices may hinder exploration of the function's entire domain. Further developments include extensions to Tree TNs for hierarchical data representation [158].

2.5.4 A unified framework

When comparing the preceding methods, it becomes apparent that CI can also be viewed as a sketching method, where the sketches are orthogonal projections onto the spaces spanned by the selected columns and rows, rather than random projections. In particular,

$$\begin{aligned} A(i,j) &= A(i, \mathbf{j}^\beta) A(\mathbf{i}^\alpha, \mathbf{j}^\beta)^{-1} A(\mathbf{i}^\alpha, j) \\ &= \Big(A(i,j')\mathbb{I}(j', \mathbf{j}^\beta)\Big)\Big(\mathbb{I}(\mathbf{i}^\alpha, k) A(k,l) \mathbb{I}(l, \mathbf{j}^\beta)\Big)^{-1}\Big(\mathbb{I}(\mathbf{i}^\alpha, i') A(i', j)\Big) \\ &= \end{aligned} \tag{2.5.14}$$



where $\mathbb{I}$ denotes the identity matrix, and $\mathbb{I}(j', \mathbf{j}^\beta)$ and $\mathbb{I}(\mathbf{i}^\alpha, i')$ can be interpreted as orthogonal projection sketches. If we seek a decomposition $A = BC$, we can identify

$$\begin{aligned} \Pi_B(j', \beta) &= \mathbb{I}(j', \mathbf{j}^\beta)\,, \\ \Pi_C(\beta, i') &= \Big(\mathbb{I}(\mathbf{i}^\alpha, k) A(k,l) \mathbb{I}(l, \mathbf{j}^\beta)\Big)^{-1} \mathbb{I}(\mathbf{i}^\alpha, i')\,. \end{aligned} \tag{2.5.15}$$

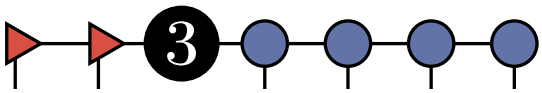
This approach can be extended to incorporate more general sketches by replacing the orthogonal projections with random projections. Specifically, we can define *left sketches* $S \in \mathbb{R}^{r\times m}$ and *right sketches* $T \in \mathbb{R}^{n\times r'}$, such that

$$A = (AT)(SAT)^\dagger (SA)\,, \tag{2.5.16}$$

where r and r' are not necessarily equal, and $(SAT)^\dagger$ denotes the Moore–Penrose pseudoinverse. This framework generalizes all the previously discussed techniques: the choice of sketches governs both the accuracy of the approximation and the computational cost. As one progresses

from orthogonal projections onto specific columns and rows, to general random projections, and ultimately to optimally designed projections such as those provided by the SVD, the approximation error decreases while the computational cost increases. This principle, when applied iteratively to the matricizations of a high-order tensor, underpins the tensorization mechanism presented in Chapter 4.

Tensor networks for machine learning



This chapter sets the foundations for the use of TNs as ML models, highlighting their specific characteristics. It begins by introducing the earliest works in the area, which extended the application of TT models beyond physics to broader ML tasks, particularly in supervised learning. We then discuss how TN models can be naturally interpreted as density functions, enabling density estimation. The chapter concludes with an overview of two common optimization strategies within this framework.

The advantages of TNs over NNs are not examined in depth in this chapter; instead, these topics are addressed in greater detail in later chapters, as they comprise core contributions of the thesis. The learning of TTs via tensorization, as an alternative to SGD, is covered in Chapter 4, while the beneficial applications of TNs over NNs are explored throughout Chapter 5 and Chapter 6. Finally, we emphasize again that a key outcome of this thesis has been the development of a Python package for integrating TN models into standard ML workflows. This package, named TensorKrowch, has supported the implementation of the methods and experiments described in both this and the following chapters. Its core components are detailed in Chapter 7.

3.1 Supervised learning

The first applications of TN models to general ML tasks appeared in 2016 with the works of Stoudenmire and Schwab [58], and Novikov *et al.* [59]. Both papers introduced TT models for supervised learning, offering complementary perspectives on the same problem. Ref. [58] adopted a quantum-inspired viewpoint, interpreting the TT as a normalized quantum state contracted with another normalized state vector (the embedded input), and using this overlap, or *fidelity*, as the classification score. Optimization was performed using a physics-inspired approach, namely Density Matrix Renormalization Group (DMRG) [159, 160]. In contrast, Ref. [59] adopted a more conventional view of TTs as low-rank decompositions of high-order tensors, used to effi-

ciently represent the coefficients of high-dimensional polynomials, with optimization performed via standard SGD.

3.1.1 Tensor train models as SVMs

Generally, a TN model can be interpreted as a linear mapping operating in an exponentially large space. This leads to a natural interpretation of TN models as SVMs, defined by the function

$$\hat{f}_{\text{TN}}(x) = \langle w, \Phi(x) \rangle, \tag{3.1.1}$$

where $\Phi : \mathbb{R}^n \to (\mathbb{R}^d)^{\otimes n}$ is a continuous embedding into an exponentially large space, and $w \in (\mathbb{R}^d)^{\otimes n}$ is the tensor of coefficients. This defines a binary classifier by separating points on either side of the decision boundary $\hat{f}_{\text{TN}}(x) = a$, for some threshold $a > 0$.

As discussed in Section 2.2.3, the high computational cost of applying the embedding to all data points and computing scalar products in the embedded space is typically avoided via the kernel trick, which only requires access to a kernel function $K(x, x')$ that computes these products implicitly. In contrast, our setting applies the embedding function explicitly to each data point, yet maintains efficiency by employing low-rank representations for both the embedding and the coefficient tensor w. While more general TN forms could be used, it is customary in this context to adopt embeddings that are tensor products of one-dimensional functions, such as

$$\Phi(x) = \phi_1(x_1) \otimes \cdots \otimes \phi_n(x_n), \tag{3.1.2}$$

where each $\phi_k(x)$ is a vector $(\phi_k^{i_k}(x))_{i_k \in [d_k]}$ whose elements form a set of linearly independent functions. Correspondingly, the coefficient tensor is given a low-rank structure to reduce storage and computational costs. The most common choice is the TT representation, due to its favorable properties, as discussed in Section 2.4.1. Accordingly, Eq. (3.1.1) can be expressed as in Eq. (2.4.21):

$$\hat{f}_{\text{TT}}(x) = \langle w, \Phi(x) \rangle = \check{f}(i_1, \ldots, i_n)\phi_1(i_1, x_1) \cdots \phi_n(i_n, x_n), \tag{3.1.3}$$

where w is identified with the tensor $\check{f}$, represented in TT format with cores $\breve{G}_1, \ldots, \breve{G}_n$. The embedding functions are defined as $\phi_k(i_k, x_k) = \phi_k^{i_k}(x_k)$, and typically the same set of functions $\{\phi^{i_k}(x)\}_{i_k \in [d]}$ and embedding dimension d are used uniformly across all features x_k in the TT chain.

The expressivity of the model depends on the choice of embedding functions and the TT ranks r. In the limit where the embedding space is infinite-dimensional—spanned by a basis $\{\phi_k^{i_k}(x)\}_{i_k}$—and the TT ranks grow exponentially with n, the model in Eq. (3.1.3) can represent any function. As the ranks are reduced and only a finite subset of basis functions is considered, constraints are introduced that limit the model's expressive power. Consequently, when using small values of r and d, the choice of embedding functions becomes critical, as different embeddings can exhibit significantly different expressive capacities.

3.1.2 Embedding functions

A common choice for the embedding, introduced by Stoudenmire and Schwab [58], is

$$\phi(x) = \left[\cos\left(\frac{\pi}{2}x\right), \sin\left(\frac{\pi}{2}x\right)\right], \quad x \in [0, 1], \tag{3.1.4}$$

which we refer to as the *unit embedding*. This embedding has the notable property that $\phi(x)$ is always a unit vector, implying that $\Phi(x)$ is a normalized tensor. Consequently, $\hat{f}_{\mathrm{TT}}$ can be interpreted as the overlap between normalized states $\Phi(x)$ and $\check{f}$, provided the latter is also a normalized TT.

In contrast, the perspective described in Ref. [59] is realized by choosing the embedding

$$\phi(x) = \left[1, x, x^2, \ldots, x^{d-1}\right], \tag{3.1.5}$$

referred to as the *polynomial embedding*. This choice results in a tensor product of basis functions that spans all possible monomials in the variables $x_1, \ldots, x_n$ up to degree $d-1$ in each variable:

$$\Phi(x) = \left(x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}\right)_{i_1,\ldots,i_n=0}^{d-1}. \tag{3.1.6}$$

This illustrates how certain ML models can be generalized as SVMs. In particular, polynomial curve fitting, as discussed in Section 2.2.3, can be formulated in this way, enabling its application in high-dimensional settings.

Although the unit and polynomial embeddings are among the most widely used, the approach is general and allows for any choice of embedding functions, with some being better suited to specific scenarios, data distributions, or domains. Commonly used function bases include the Fourier basis, as well as Legendre, Laguerre, and Hermite polynomials [161]. Furthermore, the framework can be extended by designing more flexible embedding functions. For example, Ref. [162] proposes using NNs with trainable parameters as embeddings.

3.1.3 Multi-label classification

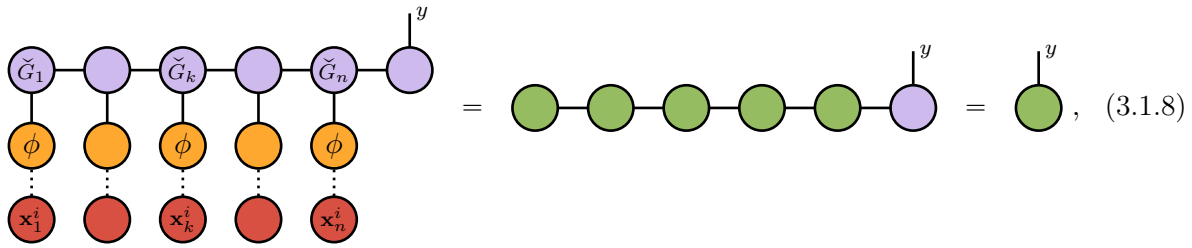
According to the previous formulation, we have defined a scalar-valued function $\hat{f}_{\mathrm{TT}}$ that returns a single score used for binary classification—or regression. However, in many practical scenarios, the task involves multi-label classification, where the target variable is $y \in [C]$ with $C > 2$ denoting the number of classes. One possible strategy in this setting is the *one-versus-all* approach, training C independent classifiers, each responsible for distinguishing instances of one class from the rest. A more natural and commonly used alternative, especially in NNs, is to define a vector-valued function that outputs a score for each class.

To handle this, we extend the TT model by incorporating an additional variable $y \in [C]$, yielding

$$\hat{f}_{\mathrm{TT}}(x)_y = \check{f}(i_1, \ldots, i_n, y)\phi(i_1, x_1) \cdots \phi(i_n, x_n), \tag{3.1.7}$$

where $\check{f}(i_1, \ldots, i_n, y)$ is an order-$(n+1)$ TT, with an extra core assigned to the output variable. While the output variable is placed in the final position here, it can in general be inserted at any position within the TT, a design choice that can significantly influence model performance.

Graphically, such a TT model, when contracted with a specific input sample $\mathbf{x}^i$, is represented as



(3.1.8)

thereby producing a vector of class scores. In this representation, we denote by



(3.1.9)

the evaluation of ϕ on $\mathbf{x}_k^i$, where the red node can be identified with the Dirac delta distribution δ, satisfying

$$\int_{\mathcal{X}_k} \phi^{i_k}(x_k)\delta(x_k - \mathbf{x}_k^i)dx_k = \phi^{i_k}(\mathbf{x}_k^i)\,. \tag{3.1.10}$$

Remark 3.1.1. We establish the above notation to maintain consistency with the graphical representation of the discrete case, interpreting dotted, connected edges as *continuous contractions*, where sums are replaced by integrals. In general, such contractions, as in Eq. (3.2.7), are well-defined, since the embeddings are typically polynomial or sinusoidal functions. The definition of $\phi^{i_k}(\mathbf{x}^{i_k})$ in Eq. (3.1.10), however, is less straightforward, but it can be made rigorous through the theory of distributions. In any case, it should be noted that the delta distribution is introduced solely to maintain consistent notation, and its use throughout this and the following chapters should be understood as evaluating functions at the corresponding samples.

As loss function, one may optimize the quadratic loss

$$\ell(\hat{f}_{\text{TT}}(x), y) = \frac{1}{2}\|\hat{f}_{\text{TT}}(x) - e_y\|_2^2, \tag{3.1.11}$$

where e_y denotes the y-th canonical basis vector, having a 1 at position y and 0 elsewhere. Beyond this type of loss, where the TT model's output is unrestricted, one can apply a softmax transformation to convert the output logits vector into a conditional probability distribution $\hat{p}_{\text{TT}}(\cdot|x)$, enabling the use of other loss functions such as cross-entropy. However, when the goal is to represent probabilities directly using TT models, a different strategy is typically employed, inspired by the use of TTs in representing quantum systems. This approach will be discussed in more detail in the next section.

3.2 Density estimation

When representing a quantum state as in Eq. (2.4.11), in a basis $\{|i_1 \cdots i_n\rangle\}_{i_k=0}^{d_k-1}$, the complex coefficients of the tensor correspond to the amplitudes of the state, whose squared moduli yield the probabilities of observing the system in each basis state upon measurement. This formulation provides a natural perspective for representing probability distributions using TT models. Specifically, one can define a distribution $\hat{p}_{\text{TT}}$ as

$$\hat{p}_{\text{TT}}(x) = |\hat{f}_{\text{TT}}(x)|^2\,, \tag{3.2.1}$$

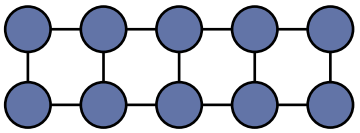
a principle known as the Born Rule (BR). The only requirement for $\hat{f}_{\text{TT}}$ to define a valid probability distribution is that it must be normalized.

When $\hat{f}_{\text{TT}}$ represents a discrete function, and the input variables $x_k \in [d_k]$ are mapped to the canonical basis vectors $e_{x_k} \in \mathbb{R}^{d_k}$, the model is equivalent to a quantum state representation,

where $|\hat{f}_{\text{TT}}(x)|^2$ gives the probability of observing each input configuration $(x_1, \ldots, x_n)$. In this setting, normalizing the distribution requires dividing $\hat{f}_{\text{TT}}$ by the partition function

$$Z = \sum_{\forall k\, x_k \in [d_k]} |\hat{f}_{\text{TT}}(x_1, \ldots, x_n)|^2 = \hat{f}_{\text{TT}}(x_1, \ldots, x_n)\hat{f}_{\text{TT}}(x_1, \ldots, x_n), \tag{3.2.2}$$

which corresponds to computing the squared norm of the TT, graphically represented as



. (3.2.3)

More generally, when the input variables are transformed via discrete or continuous embeddings ϕ into a non-orthonormal set of vectors or functions, the partition function takes the form

$$Z = \check{f}(i_1, \ldots, i_n)\phi(i_1, x_1) \cdots \phi(i_n, x_n)\phi(j_1, x_1) \cdots \phi(j_n, x_n)\check{f}(j_1, \ldots, j_n), \tag{3.2.4}$$

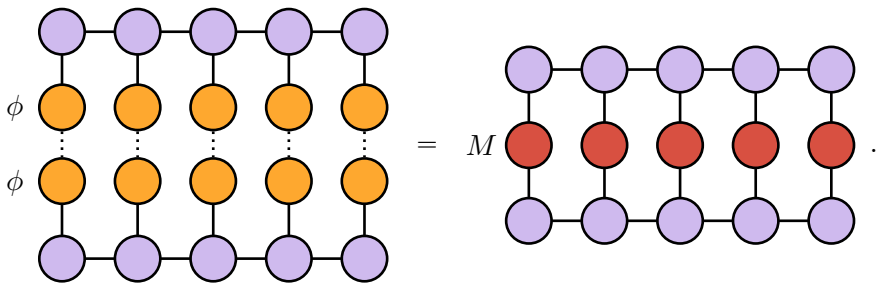
which can be simplified to

$$Z = \check{f}(i_1, \ldots, i_n)M(i_1, j_1) \cdots M(i_n, j_n)\check{f}(j_1, \ldots, j_n), \tag{3.2.5}$$

by introducing the *overlap matrices* $M \in \mathbb{R}^{d \times d}$, defined as

$$M(i, j) = \int_{x \in \mathcal{X}} \phi(i, x)\phi(j, x)\, dx\,. \tag{3.2.6}$$

Graphically,



. (3.2.7)

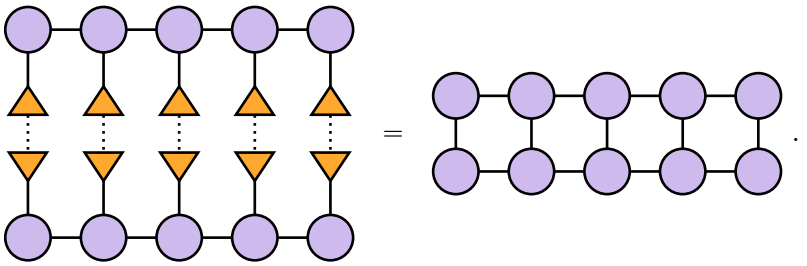
However, as explained in Ref. [161], any such transformation can be converted into an isometric one through a gauge transformation that modifies both the embedding and the TT representation. Specifically, there exists a matrix X such that $XMX = \mathbb{I}$. By redefining the embeddings as

$$\tilde{\phi}(i, x) = X(i, j)\phi(j, x)\,, \tag{3.2.8}$$

and the TT cores as

$$\tilde{G}_k(\alpha_{k-1}, i_k, \alpha_k) = \check{G}_k(\alpha_{k-1}, j_k, \alpha_k)X^{-1}(j_k, i_k)\,, \tag{3.2.9}$$

one obtains an isometric embedding satisfying



(3.2.10)

Therefore, embeddings can generally be taken to be isometric, eliminating the need to explicitly include the matrices M in proofs or computations.

This formulation of TTs for representing distributions via the BR has been widely adopted, with applications to image [163] and text [164] datasets. Moreover, the framework has been extended to other types of TNs, such as Trees [63] and PEPS [65].

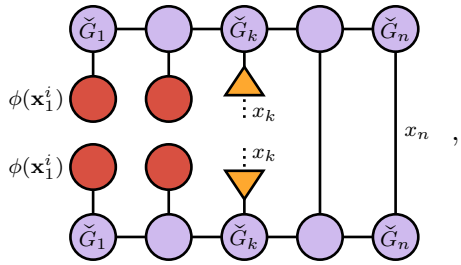
3.2.1 Direct sampling from tensor trains

A key advantage of this representation is its ability to compute marginal and conditional distributions efficiently, a feature that is fundamental for enabling direct sampling. In particular, sampling from a density p can be simplified by considering the factorization

$$p(x_1, \ldots, x_n) = \prod_{k=1}^{n} p(x_k \mid x_{1:k-1}), \tag{3.2.11}$$

where we define $p(x_1 \mid x_{1:0}) := p(x_1)$. Thus, if the conditional densities $p(x_k \mid x_{1:k-1})$ can be computed efficiently, it becomes straightforward to sequentially draw each feature $\mathbf{x}_k^i$ of the i-th sample, conditioned on the previously drawn features $\mathbf{x}_{1:k-1}^i$.

The TT representation enables this computation via



(3.2.12)

which can be performed efficiently. This allows one to draw $\mathbf{x}_k^i \sim \hat{p}_{\mathrm{TT}}(x_k \mid \mathbf{x}_{1:k-1}^i)$. Moreover, this procedure can be extended to draw N samples simultaneously with a computational cost that scales linearly. This scheme has also been generalized to higher-dimensional TNs, such as PEPS, with further adaptations to mitigate the additional computational cost inherent in cyclic TNs [165].

3.3 Training algorithms

Apart from the direct tensorization approach detailed in Chapter 4, which may not be applicable in all scenarios, a general method for learning TT models—common in many ML contexts—is via gradient descent. As with NNs, this approach enables computation of gradients with respect to all TT cores for iterative updates. However, TT models also support alternative schemes that leverage the specific properties of TNs, notably one based on the DMRG algorithm.

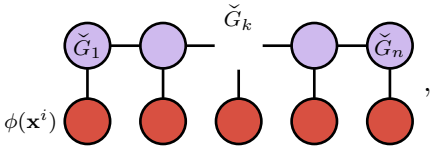
3.3.1 Direct gradient descent

The direct approach applies the chain rule to propagate the gradient of the loss function to each TT core, allowing all cores to be updated independently at each step. Specifically, for a loss function ℓ, performing ERM over possible TT representations requires computing the gradients

$$\frac{\partial \widehat{\mathcal{R}}(\hat{f}_{\mathrm{TT}})}{\partial \breve{G}_k} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \ell(\hat{f}_{\mathrm{TT}}(\mathbf{x}^i), \mathbf{y}^i)}{\partial \hat{f}_{\mathrm{TT}}(\mathbf{x}^i)} \frac{\partial \hat{f}_{\mathrm{TT}}(\mathbf{x}^i)}{\partial \breve{G}_k}, \tag{3.3.1}$$

where, by linearity, the derivative of $\hat{f}_{\mathrm{TT}}(\mathbf{x}^i)$ with respect to each core is given by

$$\frac{\partial \hat{f}_{\mathrm{TT}}(\mathbf{x}^i)}{\partial \breve{G}_k} =$$



, (3.3.2)

that is, the contraction of the remaining TN, excluding core $\breve{G}_k$, resulting in a tensor of the same order as $\breve{G}_k$. Each core is then updated via

$$\breve{G}_k^{(t+1)} = \breve{G}_k^{(t)} - \eta \frac{\partial \widehat{\mathcal{R}}(\hat{f}_{\mathrm{TT}})}{\partial \breve{G}_k}, \tag{3.3.3}$$

where η denotes the learning rate. As with NNs, more sophisticated optimizers such as Adam can be readily employed. Furthermore, following standard SGD practice, gradients are typically averaged over small batches rather than the full dataset at each step.

This approach offers the key advantage of fast convergence when paired with suitable initialization and an appropriate embedding. However, it presents two main limitations. First, the TN structure must be fixed in advance, meaning the TT ranks remain static throughout training. Although the TT can be re-canonicalized in intermediate steps via SVDs, potentially reducing ranks, this process cannot adapt dynamically to the data. The initial ranks therefore impose an upper bound that SVDs cannot later exceed. Second, direct SGD allows the cores to drift into undesirable regimes, which can degrade model performance, destabilize training, and hinder convergence. For instance, a common strategy to maintain stability is to keep the TT in canonical form, which is typically lost when the cores are updated.

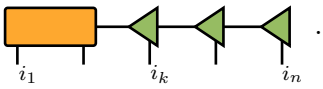
The first issue has been addressed in various ways, such as by pre-processing the data distribution or introducing intermediate steps that enable adaptive learning of the TN structure. These include methods that attempt to discover arbitrary graph structures [166, 167], reorder variables within a fixed graph [168, 169], or learn appropriate ranks for a fixed graph and variable

ordering [170]. The second issue can be partially mitigated by periodically re-canonicalizing the TT or by employing Riemannian optimization, which projects gradients to remain within the *canonical form manifold*. Nonetheless, in certain scenarios, it may be advantageous to adopt an alternative optimization strategy that inherently addresses these limitations, as introduced in the next section.

3.3.2 DMRG-based optimization

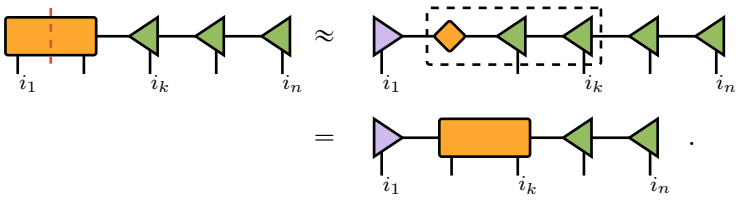
The Density Matrix Renormalization Group [159] is a variational technique originally developed in condensed matter physics to compute ground states of one-dimensional quantum systems. Although it was not initially formulated in terms of TTs, the realization that DMRG implicitly operates within the TT ansatz provided a natural and powerful framework for both theoretical analysis and practical implementation [160]. This insight has inspired optimization strategies in ML, where DMRG-like algorithms are employed to train TT models through iterative sweeps. These sweeps update one or two tensors at a time while keeping the others fixed, thereby enabling efficient learning with adaptively adjusted ranks [58]. In the numerical linear algebra community, a related approach is known as Alternating Least Squares (ALS) [171].

Specifically, DMRG begins by placing the TT in right canonical form. Instead of using a single core as the OC, the OC typically consists of a block of two or more contracted tensors, such as



(3.3.4)

Then, SGD is applied for one or more steps, treating only the OC as the trainable core and updating it accordingly. The updated core is subsequently split via SVD, and the block is moved to the right as



(3.3.5)

This new block is again updated using SGD and split, moving the OC sequentially along the chain to the rightmost site and then back to the left, performing complete sweeps.

Notably, since only the OC is updated at each step, the TT remains in canonical form throughout the process, promoting numerical stability. If normalization is also desired, it suffices to normalize the OC, which is computationally inexpensive. Moreover, because the TT is linear with respect to the trainable core, minimizing convex loss functions such as the quadratic loss (3.1.11) becomes a convex problem over this core. This guarantees a descent direction at each step, though not necessarily global optimality for the full TT. Nonetheless, as demonstrated in condensed matter applications, this method often yields highly accurate solutions.

One limitation, however, is that while convergence in terms of sweep count is often fast, the total number of updates is typically larger than in direct SGD methods, which can be a drawback in very high-dimensional settings.

On the other hand, since the OC aggregates more than one core, it allows the optimization to explore regimes where this tensor acquires higher rank. This rank is later revealed when the tensor is split and shifted to the next position. Thus, this mechanism provides a natural approach for adaptively learning the TT ranks along the chain. In fact, it is common to initialize the TT with all ranks set to one—that is, as a product state—and let the ranks grow adaptively during training.

Tensorization of neural networks



As discussed in previous chapters, one advantage of TNs is that they can be derived through low-rank tensor decompositions, thereby overcoming certain limitations of gradient descent optimization. This chapter introduces one such decomposition method aimed at tensorizing pretrained NNs—that is, converting them into TN models—to enhance privacy and interpretability by leveraging the inherent properties of TNs. The algorithm and its performance analysis are presented as described in the latest preprint of this thesis [172].

4.1 Motivation

Our primary objective is to decompose NN models, which approximate probability distributions, into a TT. Specifically, we consider a non-negative, n-dimensional function $\hat{p}_{\text{NN}}$ (the neural network) that approximates an underlying density p, and aim to construct a new approximation of p, $\hat{p}_{\text{TT}}$ (in TT form), using a limited number of evaluations of $\hat{p}_{\text{NN}}$. To achieve this, we build on sketching and CI techniques—such as those discussed in Section 2.5—which have shown promising results in decomposing high-dimensional tensors into TT form.

The approach proposed by Hur *et al.* [89] emphasizes not finding the most accurate approximation to $\hat{p}_{\text{E}}$ (the empirical distribution), but rather constructing a different approximation of p in TT form. In our work, we follow this scheme, where the initial approximation $\hat{p}_{\text{NN}}$ is provided by a NN, rather than by an empirical distribution. Consequently, we favor TT-RS over TT-CI for three primary reasons: (i) random sketches help reduce the variance of the approximation $\hat{p}$, (ii) SVD can be performed in each step without requiring modifications to the algorithm, and (iii) the method's applicability to continuous functions without increasing the number of tensors in the TT is crucial for maintaining efficiency. This is particularly important as we aim to avoid creating TT models with significantly more parameters than the original NNs, which could occur if we discretize the network to very high precision. Although such continuous adaptation is also feasible in TT-CI, TT-RS provides a more favorable approach. It allows, in a single

sweep, to obtain the discrete tensors corresponding to the specific continuous embeddings, while simultaneously performing SVD to determine the appropriate ranks.

However, as discussed earlier, TT-RS with random sketches is generally inefficient, especially in the case we consider where the approximation $\hat{p}_{\text{NN}}$ is not an empirical distribution. To address this, we incorporate the core idea of CI into TT-RS: first selecting a set of columns and rows to reconstruct the entire tensor. Unlike CI, where a rank-r matrix is reconstructed from r columns and r rows with a submatrix of maximum volume at their intersection, we select a larger number $N > r$ of columns and rows, which may not be optimal in the sense of maximum volume. A random projection is then applied to this subspace to reduce variance, followed by SVD to determine the proper rank r.

In our case studies, we assume that the selected N columns and rows correspond to points in the function's domain where the function exhibits significant behavior. For instance, when decomposing NNs, these points can be elements of the training dataset or samples drawn from the same distribution. Since these points are the ones the model was trained on, they represent the subregion of interest that the TT must capture, avoiding flat or noisy regions of the density. In scenarios where training points are unavailable, Monte Carlo techniques could be used to sample points with high probability. It should be noted that the selection of these points is heuristic, and we do not prove its optimality in this work.

4.2 Description of the algorithm

In this section we provide a detailed description of our proposed algorithm, Tensor Train via Recursive Sketching from Samples, or TT-RSS, presented in a similar form to the presentation of TT-RS in Ref. [89]. We first consider the simplified case of a discrete function $f : [d_1] \times \cdots \times [d_n] \to \mathbb{R}$ and assume it admits a TT representation with ranks $r_1, \ldots, r_{n-1}$. Additionally, we are given a set of N pivots, $\mathbf{x} = \{(\mathbf{x}_1^i, \ldots, \mathbf{x}_n^i) : \mathbf{x}_k^i \in [d_k]\}_{i\in[N]}$, which, as stated previously, could represent training points from the model or points defining a region of interest within the domain of the function f.

4.2.1 Main idea

The algorithm is based on the observation that f can be treated as a low-rank matrix. This means that there exists a decomposition of each unfolding matrix of f of the form

$$f(x_{1:k}, x_{k+1:n}) = \Phi_k(x_{1:k}, \alpha_k)\Psi_k(\alpha_k, x_{k+1:n}), \tag{4.2.1}$$

where $\alpha_k \in [r_k]$, and Φ_k and Ψ_k are matrices whose column and row spaces, respectively, span the column and row spaces of f. Since we assume that f admits a TT representation, one possible decomposition of its k-th unfolding matrix is obtained by contracting the first k and the last $n - k$ cores, respectively:

$$\begin{aligned} \Phi_k(x_{1:k}, \alpha_k) &:= G_1(x_1, \alpha_1)G_2(\alpha_1, x_2, \alpha_2)\cdots G_k(\alpha_{k-1}, x_k, \alpha_k), \\ \Psi_k(\alpha_k, x_{k+1:n}) &:= G_{k+1}(\alpha_k, x_{k+1}, \alpha_{k+1})\cdots G_{n-1}(\alpha_{n-2}, x_{n-1}, \alpha_{n-1})G_n(\alpha_{n-1}, x_n), \end{aligned} \tag{4.2.2}$$

for all $k \in [n-1]$. Therefore, if we can efficiently obtain a set of vectors spanning the column space of the unfolding matrices of f, we can define the following series of equations:

$$\Phi_k(x_{1:k}, \alpha_k) = \Phi_k(x_{1:k-1}, \alpha_{k-1})G_k(\alpha_{k-1}, x_k, \alpha_k), \tag{4.2.3}$$

from which each core can be determined. This result is formalized in the following proposition, as proved in Ref. [89].

Proposition 4.2.1. *Let $f : [d_1] \times \cdots \times [d_n] \to \mathbb{R}$ a discrete function that admits a TT representation with ranks $r_1, \ldots, r_{n-1}$, and define tensors $\Phi_k : [d_1] \times \cdots \times [d_k] \times [r_k] \to \mathbb{R}$ such that the column space of $\Phi_k(x_{1:k}, \alpha_k)$ is the same as the column space of the k-th unfolding matrix of f. Then, for the unknowns $G_1 : [d_1] \times [r_1] \to \mathbb{R}$, $G_k : [r_{k-1}] \times [d_k] \times [r_k] \to \mathbb{R}$ for all $k \in \{2, \ldots, n-1\}$, and $G_n : [r_{n-1}] \times [d_n] \to \mathbb{R}$, we consider the following series of equations:*

$$\begin{aligned} G_1(x_1, \alpha_1) &= \Phi_1(x_1, \alpha_1), \\ \Phi_{k-1}(x_{1:k-1}, \alpha_{k-1}) G_k(\alpha_{k-1}, x_k, \alpha_k) &= \Phi_k(x_{1:k-1}, x_k, \alpha_k), \ k \in \{2, \ldots, n-1\}, \\ \Phi_{n-1}(x_{1:n-1}, \alpha_{n-1}) G_n(\alpha_{n-1}, x_n) &= f(x_{1:n-1}, x_n), \end{aligned} \tag{4.2.4}$$

which we refer to as the Core Determining Equations (CDEs). Each of these equations has a unique solution, and the solutions $G_1, \ldots, G_n$ satisfy

$$f(x_1, \ldots, x_n) = G_1(x_1, \alpha_1) G_2(\alpha_1, x_2, \alpha_2) \cdots G_{n-1}(\alpha_{n-2}, x_{n-1}, \alpha_{n-1}) G_n(\alpha_{n-1}, x_n). \tag{4.2.5}$$

Remark 4.2.1. When defining the CDEs (4.2.4), one possible approach would be to use the first $k-1$ cores obtained on the left-hand side to solve for the k-th core. However, introducing the matrices Φ_{k-1} and Φ_k on both sides of the equation helps mitigate the accumulation of errors that could arise from reusing the previously computed solutions $G_1, \ldots, G_{k-1}$.

To implement this idea efficiently, we need to define two types of sketch functions, that serve distinct purposes. First, similarly to rSVD [146], we define random projections $T_{k+1} : [d_{k+1}] \times \cdots \times [d_n] \times [N_{k+1:n}] \to \mathbb{R}$ for $k \in [n-1]$, referred to as *right sketches*. These projections will be used to generate random vectors in the range of the k-th unfolding matrix of f. Secondly, it should be noted that the CDEs (4.2.4) are matrix equations with a number of coefficients that grows exponentially with the dimension n, making the system of equations overdetermined and computationally infeasible to solve. To reduce the number of equations, we define, for $k \in \{2, \ldots, n\}$, the *left sketches* $S_{k-1} : [N_{1:k-1}] \times [d_1] \times \cdots \times [d_{k-1}] \to \mathbb{R}$. Applying these sketches to both sides of Eq. (4.2.4), we obtain

$$S_{k-1}(\beta_{k-1}, x_{1:k-1}) \Phi_{k-1}(x_{1:k-1}, \alpha_{k-1}) G_k(\alpha_{k-1}, x_k, \alpha_k) = S_{k-1}(\beta_{k-1}, x_{1:k-1}) \Phi_k(x_{1:k}, \alpha_k), \tag{4.2.6}$$

with $k \in \{2, \ldots, n-1\}$.

As will be detailed later in this section, the left sketch functions must have a recursive definition to incorporate the SVD into the algorithm for determining the actual rank r_k. Specifically, the sketch S_k is constructed as a composition of the previous sketch, S_{k-1}, and an auxiliary function $s_k : [N_{1:k}] \times [N_{1:k-1}] \times [d_k] \to \mathbb{R}$:

$$S_k(\beta_k, x_{1:k}) := S_{k-1}(\beta_{k-1}, x_{1:k-1}) s_k(\beta_k, \beta_{k-1}, x_k) \tag{4.2.7}$$

for $k \in \{2, \ldots, n-1\}$, and $S_1 := s_1$ with $s_1 : [N_1] \times [d_1] \to \mathbb{R}$.

Remark 4.2.2. Since $N_{a:b}$ denotes the number of unique elements in $\mathbf{x}_{a:b}$, it is clear that it is upper bounded by the total amount of possible values in this set. Specifically, $N_{a:b} \leqslant d_a \cdots d_b$.

In contrast to the TT-RS approach [89], which employs random sketches, we propose the use of orthogonal projection sketches, as introduced in Section 2.5.4. Specifically, we define sketches S_{k-1} and T_{k+1} such that applying them to f corresponds to evaluating the function on the subsets of variables $\mathbf{x}_{1:k-1}$ and $\mathbf{x}_{k+1:n}$, respectively, thereby constructing the tensor $f(\mathbf{x}_{1:k-1}, [d_k], \mathbf{x}_{k+1:n})$. This approach offers efficiency for any kind of function, provided that some additional information about the function's domain is available, which is typically the case in most scenarios. Moreover, this extra knowledge allows us to overcome some of the limitations of the `maxvol` algorithm used in TT-CI. In particular, it ensures that the algorithm explores the relevant subregion of the domain, thus avoiding ergodicity issues that may arise in sparse tensors, functions with discrete symmetries, or multivariate functions with very narrow peaks [142].

The complete TT-RSS decomposition is outlined in Algorithm 4.1, where it is divided into a series of subroutines, following a structure similar to how the TT-RS algorithm is presented in Ref. [89]. These subroutines are described in detail in the following section.

Algorithm 4.1 Tensor Train via Recursive Sketching from Samples

Require: Discrete function $f : [d_1] \times \cdots \times [d_n] \to \mathbb{R}$.
Require: Sketch samples $\mathbf{x} = \{(\mathbf{x}_1^i, \ldots, \mathbf{x}_n^i) : \mathbf{x}_k^i \in [d_k]\}_{i \in [N]}$.
Require: Target ranks $r_1, \ldots, r_{n-1}$.
$s_1, \ldots, s_{n-1}, T_2, \ldots, T_n \leftarrow$ SketchBuilding$(\mathbf{x})$.
$\widetilde{\Phi}_1, \ldots, \widetilde{\Phi}_d \leftarrow$ Sketching$(f, s_1, \ldots, s_{n-1}, T_2, \ldots, T_n)$.
$B_1, \ldots, B_n \leftarrow$ Trimming$(\widetilde{\Phi}_1, \ldots, \widetilde{\Phi}_n, r_1, \ldots, r_{n-1})$.
$A_1, \ldots, A_{n-1} \leftarrow$ SystemForming$(B_1, \ldots, B_{n-1}, s_1, \ldots, s_{n-1})$.
Solve via least-squares for the unknowns $G_1 : [d_1] \times [r_1] \to \mathbb{R}$, $G_k : [r_{k-1}] \times [d_k] \times [r_k] \to \mathbb{R}$ for all $k \in \{2, \ldots, n-1\}$, and $G_n : [r_{n-1}] \times [d_n] \to \mathbb{R}$:

$$\begin{aligned} G_1(x_1, \alpha_1) &= B_1(x_1, \alpha_1), \\ A_{k-1}(\beta_{k-1}, \alpha_{k-1}) G_k(\alpha_{k-1}, x_k, \alpha_k) &= B_k(\beta_{k-1}, x_k, \alpha_k), \quad k \in \{2, \ldots, n-1\}, \\ A_{n-1}(\beta_{n-1}, \alpha_{n-1}) G_n(\alpha_{n-1}, x_n) &= B_n(\beta_{n-1}, x_n). \end{aligned}$$

return $G_1, \ldots, G_n$

4.2.2 Details of subroutines

For clarity in the description of the entire algorithm, we divide it into the following subroutines: SketchBuilding, Sketching, Trimming, SystemForming, and Solving.

SketchBuilding

The first step of the algorithm involves applying the sketch functions T_{k+1} and S_{k-1} to f, forming the coefficient matrices $\widetilde{\Phi}_k$. To ensure efficiency in the sketching process, we define preliminary sketches $P_{a:b} : [N_{a:b}] \times [d_a] \times \cdots \times [d_b] \to \mathbb{R}$, which will be used to construct the left and right sketches, as:

$$P_{a:b}(i, x_{a:b}) := \mathbb{I}_{a:b}(\mathbf{x}_{a:b}^i, x_{a:b}), \tag{4.2.8}$$

for $i \in [N_{a:b}]$, where $\mathbb{I}_{a:b} \in \mathbb{R}^{d_a \cdots d_b}$ denotes the identity matrix. This is, $P_{a:b}$ represents the orthogonal projection onto the subspace determined by the variables x_a to x_b of the pivots, $\mathbf{x}_{a:b}$.

Similarly, we denote $P^{\top}_{a:b} : [d_a] \times \cdots \times [d_b] \times [N_{a:b}] \to \mathbb{R}$ as the function whose matricization is the transpose of the matricization of $P_{a:b}$, such that:

$$P^{\top}_{a:b}(x_{a:b}, i) = P_{a:b}(i, x_{a:b}). \tag{4.2.9}$$

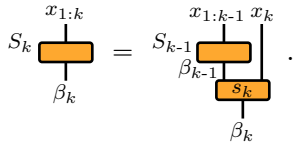
From these projections, we can define the left and right sketches as follows:

$$\begin{aligned} S_k(\beta_k, x_{1:k}) &:= P_{1:k}(\beta_k, x_{1:k}) = \mathbb{I}_{1:k}(\mathbf{x}^{\beta_k}_{1:k}, x_{1:k}), \\ T_k(x_{k:n}, \gamma_{k-1}) &:= P^{\top}_{k:n}(x_{k:n}, \eta_{k-1}) U_k(\eta_{k-1}, \gamma_{k-1}) = \mathbb{I}_{k:n}(x_{k:n}, \mathbf{x}^{\eta_{k-1}}_{k:n}) U_k(\eta_{k-1}, \gamma_{k-1}), \end{aligned} \tag{4.2.10}$$

where $U_k \in \mathrm{O}(N_{k:n})$ is a random orthogonal—or, in the case of complex coefficients, unitary—matrix sampled from the Haar measure as described in Ref. [173]. Note that with this definition, the sketches S_k admit a recursive formulation if we define

$$s_k(\beta_k, \beta_{k-1}, x_k) := \begin{cases} 1, & \text{if } \beta_k = \beta_{k-1} \text{ and } x_k = \mathbf{x}^{\beta_k}_k \\ 0, & \text{otherwise} \end{cases} \tag{4.2.11}$$

for $k \in \{2, \ldots, n-1\}$. In pictorial form, sketches S_k are decomposed as follows:



(4.2.12)

Remark 4.2.3. Choosing U_k as an orthogonal random matrix aims to construct the sketches T_k as Johnson-Lindenstrauss embeddings. Specifically, instead of applying the whole U_k, one can retain only a subset of its columns, yielding a projection to a lower-dimensional space, which helps reduce the computational cost of subsequent steps—such as the SVD in TRIMMING. Furthermore, since the orthogonal projections $P^{\top}_{a:b}$ are deterministic matrices that restrict the action of f to the subspace defined by $\mathbf{x}_{a:b}$, the isometry U_k introduces randomness while approximately preserving pairwise distances. No additional isometry is introduced in the left sketches S_k, as they are solely intended to reduce the number of equations, and introducing an isometry would be irrelevant for this purpose.

SKETCHING

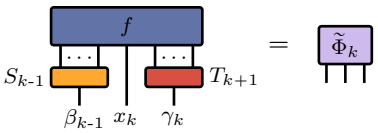
Applying these sketches entails first constructing the tensors $\bar{\Phi}_k : [N_{1:k-1}] \times [d_k] \times [N_{k+1:n}] \to \mathbb{R}$ as

$$\begin{aligned} \bar{\Phi}_k &:= P_{1:k-1}([N_{1:k-1}], x_{1:k-1}) f(x_{1:k-1}, [d_k], x_{k+1:n}) P^{\top}_{k+1:n}(x_{k+1:n}, [N_{k+1:n}]) \\ &= \mathbb{I}_{1:k-1}(\mathbf{x}_{1:k-1}, x_{1:k-1}) f(x_{1:k-1}, [d_k], x_{k+1:n}) \mathbb{I}_{k+1:n}(x_{k+1:n}, \mathbf{x}_{k+1:n}) \\ &= f(\mathbf{x}_{1:k-1}, [d_k], \mathbf{x}_{k+1:n}), \quad k \in \{2, \ldots, n-1\}, \end{aligned} \tag{4.2.13}$$

this is, evaluating the function f on all the points in the set $\mathbf{x}_{1:k-1} \times [d_k] \times \mathbf{x}_{k+1:n}$. From $\bar{\Phi}_k$, we form the tensors $\widetilde{\Phi}_k$ by multiplying with U_{k+1} on the right:

$$\begin{aligned} \widetilde{\Phi}_k(\beta_{k-1}, x_k, \gamma_k) &:= \bar{\Phi}_k(\beta_{k-1}, x_k, \eta_k) U_{k+1}(\eta_k, \gamma_k) \\ &= S_{k-1}(\beta_{k-1}, x_{1:k-1}) f(x_{1:k-1}, x_k, x_{k+1:n}) T_{k+1}(x_{k+1:n}, \gamma_k), \end{aligned} \tag{4.2.14}$$

which is graphically depicted as



(4.2.15)

Finally, let $\bar{\Phi}_1 := f([d_1], \mathbf{x}_{2:n})$ and $\bar{\Phi}_n := f(\mathbf{x}_{1:n-1}, [d_n])$, thereby defining the corresponding tensors $\widetilde{\Phi}_1$ and $\widetilde{\Phi}_n$.

The tensors $\widetilde{\Phi}_k$ serve the same role as the tensors Φ_k on the right-hand side of Eq. (4.2.4), with the distinction that the indices β_{k-1} and γ_k range over $[N_{1:k-1}]$ and $[N_{k+1:n}]$, respectively, instead of $[r_{k-1}]$ and $[r_k]$. To form the tensors on the left-hand side of Eq. (4.2.4), one would typically contract f with S_{k-1} and T_k. However, as will be detailed in the subsequent subroutines, this additional computation can be avoided by reusing the previously computed $\widetilde{\Phi}_{k-1}$ and taking advantage of the recursive definition of S_{k-1}.

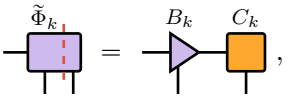
Remark 4.2.4. As commented earlier, in the general description of TT-RS provided in Ref. [89], left and right sketches are described as random projections, and only the construction of efficient sketches for Markov chains is discussed. However, applying random projections in a space of size $\prod_{k=1}^{n} d_k$ is generally infeasible. Our adaptation, TT-RSS, addresses this challenge by introducing the projections $P_{a:b}$, whose practical computation only requires evaluating f on the selected pivots. This approach reduces the complexity of creating tensors $\bar{\Phi}_k$ to $O(N_{1:k-1} d_k N_{k+1:n})$.

Trimming

To find the proper ranks r_k, we perform SVD on the matrices $\widetilde{\Phi}_k((\beta_{k-1}, x_k), \gamma_k)$ to find functions $B_k : [N_{1:k-1}] \times [d_k] \times [r_k] \to \mathbb{R}$ and $C_k : [r_k] \times [N_{k+1:n}] \to \mathbb{R}$ satisfying

$$\widetilde{\Phi}_k(\beta_{k-1}, x_k, \gamma_k) = B_k(\beta_{k-1}, x_k, \alpha_k) C_k(\alpha_k, \gamma_k), \tag{4.2.16}$$

or equivalently,



(4.2.17)

for all $k \in \{2, \ldots, n-1\}$. For $k = 1$ we obtain $\widetilde{\Phi}_1(x_1, \gamma_1) = B_1(x_1, \alpha_1) C_1(\alpha_1, \gamma_1)$ by performing SVD on $\widetilde{\Phi}_1(x_1, \gamma_1)$, and finally, we set $B_n(\beta_{n-1}, x_n) := \widetilde{\Phi}_n(\beta_{n-1}, x_n)$.

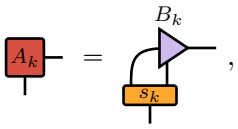
Remark 4.2.5. The SVD is required only if $N_{k+1:n} > r_k$; otherwise, we set $B_k := \widetilde{\Phi}_k$. However, it is standard and recommended to choose the number of pivots N such that $N_{k+1:n} > r_k$ for all $k \in [n-1]$, in order to effectively capture the range of all the unfolding matrices of f. This is particularly important when the selected pivots are not guaranteed to provide the full range of the matrix, as they serve merely as a heuristic.

SystemForming

To form the coefficient matrices on the left-hand side of the CDEs (4.2.4), we construct the following matrices A_k from B_k and s_k:

$$A_k(\beta_k, \alpha_k) := s_k(\beta_k, \beta_{k-1}, x_k) B_k(\beta_{k-1}, x_k, \alpha_k), \quad k \in \{2, \ldots, n-1\}, \tag{4.2.18}$$

or graphically,



(4.2.19)

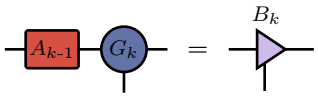
This highlights the necessity of determining the sketches S_k recursively, as it enables the construction of A_k from the tensor B_k, which already has the appropriate rank r_k. While one could argue that recursiveness might be avoided if an explicit formulation of B_k from $\widetilde{\Phi}_k$ were provided, such an approach would require prior knowledge of the rank or first performing SVD to subsequently identify an explicit projection that yields the left singular vectors. In the proposed algorithm, left sketches are inherently described recursively by definition, and thus the non-recursive case is disregarded.

Solving

Finally, the CDEs (4.2.4) can be reformulated in terms of the tensors A_{k-1} and B_k as follows:

$$\begin{aligned} G_1(x_1, \alpha_1) &= B_1(x_1, \alpha_1), \\ A_{k-1}(\beta_{k-1}, \alpha_{k-1}) G_k(\alpha_{k-1}, x_k, \alpha_k) &= B_k(\beta_{k-1}, x_k, \alpha_k), \quad k \in \{2, \dots, n-1\}, \\ A_{n-1}(\beta_{n-1}, \alpha_{n-1}) G_n(\alpha_{n-1}, x_n) &= B_n(\beta_{n-1}, x_n). \end{aligned} \tag{4.2.20}$$

In the general case, the equations above are depicted as



(4.2.21)

From this reformulation, the cores $G_1, \dots, G_n$ can be determined via least-squares.

Remark 4.2.6. In high-dimensional scenarios, it is crucial to have a functional approximation $\hat{p}$ of p, rather than merely an empirical distribution $\hat{p}_{\text{E}}$. If only N samples, polynomial in n, are available, typically there will be no observations in the off-diagonal regions of the sets $\mathbf{x}_{1:k} \times \mathbf{x}_{k+1:n}$, especially for values of k sufficiently far from 1 and n. Therefore, when sketching $\hat{p}_{\text{E}}$ as described in this section, the resulting matrices $\widetilde{\Phi}_k$ will essentially be submatrices of the identity multiplied by random matrices U_{k+1}. This results in a series of CDEs (4.2.4) where both the coefficient and bias matrices, A_{k-1} and B_k, are randomly generated, yielding erroneous results. In contrast, with a polynomial number of samples, we could first train a simple ML model and use it as the approximation $\hat{p}$ for the decomposition. This approach would enable us to fill in the off-diagonal elements of the matrices $\widetilde{\Phi}_k$, leading to significantly better results.

4.2.3 Continuous case

We adapt the described algorithm to decompose continuous functions, the most common application scenario. Following Ref. [89], we generalize their approach by relaxing the requirement of an orthonormal function basis to represent f. Instead, we allow any linearly independent set of embedding functions as a basis and aim to identify discrete cores that approximate the function effectively within this framework.

When f is a continuous function, the main argument remains nearly the same, with just a few necessary adaptations. Let $f : \mathcal{X}_1 \times \cdots \times \mathcal{X}_n \to \mathbb{R}$, where $\mathcal{X}_1, \ldots, \mathcal{X}_n \subset \mathbb{R}$, with $\mathcal{X}_k = [u, v]$ for some $u, v \in \mathbb{R}$ and all $k \in [n]$, as is typically the case. As discussed in Section 2.4.2, f can be represented in TT format by assuming it can be approximated by an expansion in a basis formed by the tensor product of continuous embedding functions, thereby decomposing f as

$$f(x_1, \ldots, x_n) \approx \check{f}(i_1, \ldots, i_n)\phi_1(i_1, x_1) \cdots \phi_n(i_n, x_n). \tag{4.2.22}$$

The goal is to obtain cores $\check{G}_1, \ldots, \check{G}_n$ that provide the TT representation of $\check{f}$.

To achieve this, the tensors $\widetilde{\Phi}_k$ are first constructed from evaluations of f, similarly to Eq. (4.2.13), and the following equations are imposed:

$$\widetilde{\Phi}_k(\beta_{k-1}, x_k, \gamma_k) = \check{\Phi}_k(\beta_{k-1}, i_k, \gamma_k)\phi_k(i_k, x_k), \tag{4.2.23}$$

from which the tensors $\check{\Phi}_k$ can be obtained. Using these, one can construct the tensors B_k and A_k, and solve the CDEs (4.2.4).

To determine the functions $\bar{\Phi}_k$, one must define continuous versions of the left and right sketches. This requires defining projections $P_{a:b}$ as the Dirac delta distribution δ, which satisfies

$$\int_{\mathcal{X}_a \times \cdots \times \mathcal{X}_b} f(x_{a:b})\delta(x_{a:b} - \mathbf{x}^i_{a:b})dx_{a:b} = f(\mathbf{x}^i_{a:b}). \tag{4.2.24}$$

The projections $P_{a:b}$ are thus given by

$$P_{a:b}(i, x_{a:b}) := \delta(x_{a:b} - \mathbf{x}^i_{a:b}), \tag{4.2.25}$$

and their application to f yields

$$\bar{\Phi}_k([N_{1:k-1}], x_k, [N_{k+1:n}]) := f(\mathbf{x}_{1:k-1}, x_k, \mathbf{x}_{k+1:n}), \quad k \in \{2, \ldots, n-1\}. \tag{4.2.26}$$

As in the finite case, functions $\widetilde{\Phi}_k$ are then constructed by multiplying $\bar{\Phi}_k$ on the right by U_{k+1}.

To solve Eq. (4.2.23), if the embedding functions $\{\phi_k^{i_k}\}_{i_k \in [d_k]}$ form an orthonormal basis, tensors $\check{\Phi}_k$ can be computed as

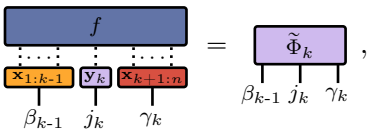
$$\check{\Phi}_k(\beta_{k-1}, j_k, \gamma_k) = \int_{\mathcal{X}_k} \widetilde{\Phi}_k(\beta_{k-1}, x_k, \gamma_k)\phi_k^{j_k}(x_k)dx_k \tag{4.2.27}$$

for all $j_k \in [d_k]$. However, in practice, common embeddings do not form an orthonormal basis, in which case Eq. (4.2.23) is solved via least-squares. This involves evaluating $\widetilde{\Phi}_k$ and $\{\phi_k^{i_k}\}_{i_k \in [d_k]}$ at a set of points $\mathbf{y}_k$ to impose the matrix equation

$$\widetilde{\Phi}_k(\beta_{k-1}, \mathbf{y}_k, \gamma_k) = \check{\Phi}_k(\beta_{k-1}, i_k, \gamma_k)\phi_k(i_k, \mathbf{y}_k). \tag{4.2.28}$$

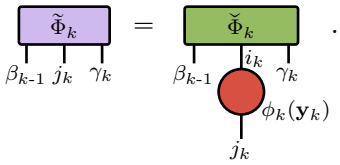
For well-posedness, $\mathbf{y}_k$ must contain at least d_k elements. If the focus is primarily on the subregion defined by the pivots, $\mathbf{y}_k$ can be chosen as $\mathbf{x}_k$. More generally, a set of equidistant points within the domain, such as $\mathbf{y}_k = \{u, u + h, u + 2h, \ldots, v\}$, is used, where it is assumed that h is a suitable step size to ensure a good approximate solution while maintaining computational efficiency. We will refer to this additional subroutine as FITTING, which is executed after SKETCHING.

Graphically, the continuous SKETCHING step is depicted as



(4.2.29)

where the nodes labeled $\mathbf{x}_{1:k-1}$, $\mathbf{y}_k$, and $\mathbf{x}_{k+1:n}$ correspond to the respective projections or evaluations of f at those points. Accordingly, the FITTING step is represented as



(4.2.30)

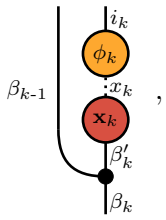
Performing SVD on $\breve{\Phi}_k$ yields tensors $\breve{B}_k$, and tensors $\breve{A}_k$ are constructed as in Eq. (4.2.18). However, a subtlety must be addressed when defining the auxiliary functions s_k. To provide a recursive definition of the left sketches, one can proceed similarly to the discrete case and define functions $s_k : [N_{1:k}] \times [N_{1:k-1}] \times \mathcal{X}_k \to \mathbb{R}$ as

$$s_k(\beta_k, \beta_{k-1}, x_k) := \begin{cases} \delta(x_k - \mathbf{x}_k^{\beta_k}), & \text{if } \beta_k = \beta_{k-1}, \\ 0, & \text{otherwise.} \end{cases} \tag{4.2.31}$$

This definition aligns with the discrete formulation of B_k used to construct A_k. However, since Eq. (4.2.23) effectively "removes the embedding", to appropriately compute the evaluations of f the embedding must be reintroduced. Thus, we need auxiliary functions $\check{s}_k : [N_{1:k}] \times [N_{1:k-1}] \times [d_k] \to \mathbb{R}$, defined as

$$\check{s}_k(\beta_k, \beta_{k-1}, i_k) := \begin{cases} \phi_k(i_k, x_k)\delta(x_k - \mathbf{x}_k^{\beta_k}), & \text{if } \beta_k = \beta_{k-1}, \\ 0, & \text{otherwise.} \end{cases} \tag{4.2.32}$$

Pictorially, functions $\check{s}_k$ are constructed as



(4.2.33)

where the red node represents the δ distribution, i.e., evaluation at the samples $\mathbf{x}_k^{\beta'_k}$, and the small black dot denotes a *copy tensor*, which corresponds to $\delta_{\beta_{k-1}\beta_k}\delta_{\beta_k\beta'_k}$.

Finally, $\breve{A}_k$ is constructed as

$$\breve{A}_k(\beta_k, \alpha_k) := \check{s}_k(\beta_k, \beta_{k-1}, i_k)\breve{B}_k(\beta_{k-1}, i_k, \alpha_k). \tag{4.2.34}$$

The tensors for the boundary cases $k = 1$ and $k = n$ are constructed analogously, and the CDEs (4.2.4) are solved to obtain cores $\breve{G}_1, \ldots, \breve{G}_n$.

The continuous version of TT-RSS is implemented in the TensorKrowch package [174].

4.2.4 Complexity

Let $d = \max_{k\in[n]} d_k$. Assuming that the complexity of evaluating f is constant and the cardinality of $\mathbf{y}_k$ is a multiple of d for all $k \in [n]$, the SKETCHING step requires $O(N^2dn)$ evaluations. Here, as per the notation used earlier, $N = \max_{1\leqslant a\leqslant b\leqslant n} N_{a:b}$. This complexity includes the SKETCHBUILDING step since forming sketches is implicitly achieved through the evaluations of f on the selected partitions of pivots. Solving the least-squares problems in FITTING (4.2.23), by multiplying by the pseudoinverse of the coefficient matrices, can be completed in $O((d+N^2)d^2n)$ time. The SVD in TRIMMING is performed in $O(N^3dn)$ time. Denoting $r = \max_{k\in[n-1]} r_k$, the matrix multiplications in SYSTEMFORMING to construct matrices A_k involve $O(N^2drn)$ operations. Finally, SOLVING the CDEs (4.2.4) requires $O(Nr^2(d+1)n)$ time. Altogether, the overall complexity is

$$O\left(N^2dn + (d+N^2)d^2n + N^3dn + N^2drn + Nr^2(d+1)n\right), \tag{4.2.35}$$

which is linear in the number of variables n. If $N \gg d$, as is typically the case, the dominant term will be N^3dn, making the complexity cubic in N.

4.3 Relation with TT-CI

With the choice of sketches described in this section, the close relation of our approach with TT-CI [129] becomes evident. As stated at the beginning of the section, we assume that the unfolding matrices of f admit a decomposition as in Eq. (4.2.1). Using the skeleton decomposition [155], this can be expressed as

$$f(x_{1:k}, x_{k+1:n}) = f(x_{1:k}, \mathbf{x}_{k+1:n}) f(\mathbf{x}_{1:k}, \mathbf{x}_{k+1:n})^{-1} f(\mathbf{x}_{1:k}, x_{k+1:n}). \tag{4.3.1}$$

Remark 4.3.1. In our setting, $\mathbf{x}_{a:b}$ denotes the subset formed by the variables x_a to x_b of the fixed set of pivots $\mathbf{x}$, and thus the number of unique elements $N_{a:b}$ could vary for each interval. In TT-CI, these partial pivots $\mathbf{x}_{a:b}$ are determined independently by a combinatorial optimization procedure, allowing us to fix $N_{a:b} = N$ for all intervals $a:b$.

As in our approach, the CI technique can be applied iteratively on the left and right factors of the right-hand side of Eq. (4.3.1) to obtain a TT representation similar to that described in Eq. (2.5.13). Consequently, the cores can be computed as follows:

$$\begin{aligned} G_1 &= f([d_1], \mathbf{x}_{2:n}), \\ G_k &= f(\mathbf{x}_{1:k-1}, \mathbf{x}_{k:n})^{-1} f(\mathbf{x}_{1:k-1}, [d_k], \mathbf{x}_{k+1:n}), \quad k \in \{2, \dots, n-1\}, \\ G_n &= f(\mathbf{x}_{1:n-1}, \mathbf{x}_n)^{-1} f(\mathbf{x}_{1:n-1}, [d_n]), \end{aligned} \tag{4.3.2}$$

which closely resembles Eq. (4.2.20), with the only difference being that these equations are solved directly by taking inverses. This underscores how our approach builds upon the ideas inherent in TT-CI, and how this connection can guide the design of better heuristics for selecting pivots.

Although we do not formally establish properties that the selected pivots must satisfy to yield good results, our proposal of choosing as pivots points from the training set, combined with the defined left and right sketches, aligns with key results from Refs. [90, 91, 129, 142].

The maximum-volume principle [157] states that interpolation sets yielding the best approximation are those for which the intersection matrix has maximum volume. In our case studies,

these interpolation sets are formed from points with high probability. Consequently, the matrices $p(\mathbf{x}_{1:k}, \mathbf{x}_{k+1:n})$ tend to have large diagonal values and smaller off-diagonal values. For instance, consider a distribution over a set of images: points $(\mathbf{x}^i_{1:k}, \mathbf{x}^j_{k+1:n})$, with $i \neq j$, are similar to $\mathbf{x}^i$ when k is close to n, and more similar to $\mathbf{x}^j$ when k is close to 1. These off-diagonal points will still have relatively high probability. However, most images will be noisy mixtures of $\mathbf{x}^i$ and $\mathbf{x}^j$, and thus will have low probabilities. Additionally, using random matrices U_k with orthonormal rows helps preserve this high volume.

Moreover, Ref. [90] demonstrated that Eq. (4.3.2) yields a TT representation of f if the interpolation sets $\mathbf{x}_{a:b}$ are nested, i.e., if they satisfy $\mathbf{x}_{1:k} \subset \mathbf{x}_{1:k-1} \times [d_k]$ and $\mathbf{x}_{k:n} \subset [d_k] \times \mathbf{x}_{k+1:n}$. This condition is inherently satisfied in our approach due to the construction of the interpolation sets from fixed pivots.

The TT-RSS algorithm proposed in this paper, while closely resembling TT-CI, differs fundamentally in its purpose: instead of seeking an accurate approximation of a model $\hat{p}$, TT-RSS aims to construct a TT representation of the underlying distribution p. This distinction is essential, as tensorizing intricate models may result in a TT model that, while differing significantly from the original model $\hat{p}$ due to potential mismatches in expressive capacity, still provides a good approximation of p. TT-RSS achieves this efficiently through a single sweep, performing SVDs to reveal ranks and employing random projections U_k to reduce variance. In contrast, TT-CI aims to accurately approximate $\hat{p}$ through iterative optimization with multiple sweeps to refine interpolation sets, with its rank-revealing adaptation further increasing computational cost via two-site sweeps [129].

Furthermore, as previously discussed, the `maxvol` step in TT-CI may suffer from ergodicity issues [142], failing to identify good representatives of the function's support in cases of high sparsity, narrow peaks, or discrete symmetries—situations commonly encountered in high-dimensional ML tasks. This limitation renders the TT-CI approach insufficient for the problems we aim to address. On the other hand, even when results from TT-RSS are unsatisfactory, rather than iterating with `maxvol` to refine interpolation sets, we can optimize all cores using gradient descent methods, starting from the TT-RSS result, to find better approximations of p. This approach motivates a general initialization method for TTs that aligns with the training set distribution and selected embedding, as we detail in Section 6.2.

4.4 Performance

The proposed algorithm, TT-RSS, can be applied to decompose probability distributions p, as well as more general functions, since the methods described do not rely on the non-negativity or normalization of p at any step. In this section, we present a series of experiments where TT-RSS is applied to various types of functions. The results demonstrate that our algorithm achieves promising outcomes across all tested scenarios.

Before detailing each example, we must define some elements that are used in the practical implementation of the algorithm. First, to determine the ranks via SVD, we introduce two parameters that bound the number of singular vectors retained. On one hand, we set an upper bound r for all the ranks r_k, where $k \in [n-1]$. To account for situations where the actual rank r_k is smaller, we define a parameter $c \in [0, 1]$, referred to as the *cumulative percentage*. Using this parameter, we define $\tilde{r}_k$ as the smallest integer satisfying:

$$\frac{\sum_{i=1}^{\tilde{r}_k} \sigma_i}{\sum_j \sigma_j} \geqslant c, \tag{4.4.1}$$

where σ_i are the singular values. The ranks are then determined as $r_k = \min(\tilde{r}_k, r)$.

To evaluate the accuracy of the results, we compute the relative error of the outputs obtained from the TT approximation provided by TT-RSS, compared to the exact outputs of the functions. Specifically, if $\mathbf{s}$ is a set of M points and we consider $f(\mathbf{s})$ as a vector in $\mathbb{R}^M$, the relative error is computed as:

$$R(\mathbf{s}) := \frac{\left\| f(\mathbf{s}) - \hat{f}_{\text{TT}}(\mathbf{s}) \right\|_2}{\|f(\mathbf{s})\|_2 + \varepsilon}, \tag{4.4.2}$$

where ε is a small value added to avoid division by zero. The set $\mathbf{s}$ will usually be referred to as *test samples*, even when the functions to be tensorized are not densities. Relative errors will also be evaluated at the pivots, enabling a comparison between the errors at the points used for tensorizing the functions and those at previously unseen points.

All experiments presented in this work were conducted on an Intel Xeon CPU E5-2620 v4 with 256GB of RAM and an NVIDIA GeForce RTX 3090, using the implementation of TT-RSS available in the TensorKrowch Python package [174]. The code for the experiments is publicly available at [175].

4.4.1 Non-densities

We begin by tensorizing functions that are not densities. In the two study cases presented here, we consider discrete functions where all input dimensions are equal, i.e., $d_k = d$ for all $k \in [n]$. Consequently, these functions take the form $f : [d]^n \to \mathbb{R}$. However, the algorithm employed in all scenarios is the continuous version of TT-RSS. Notably, this continuous version naturally reduces to the discrete case by adopting the trivial embedding functions $\phi^j(i) = \delta_{ij}$ for $i, j \in [d]$. With this setup, the matrix $\phi(i, j)$ appearing on the right-hand side of Eq. (4.2.23) becomes the identity matrix, and the equation is automatically satisfied.

Random TT

The first scenario we consider involves a function with an exact TT representation. Specifically, we define functions $f^n_{\text{TT}} : [d]^n \to \mathbb{R}$ as TTs, where $n \in \{100, 200, 500\}$ represents, as before, the length of the chain, or the number of variables. In all cases, we fix $d = 2$ and set a constant rank $\bar{r} = 10$. To randomly initialize the cores while avoiding numerical instability (e.g., exploding or vanishing outputs), we construct tensors G_k as stacks of orthogonal matrices. For each $x_k \in [2]$, the matrix $[G_k(\alpha_{k-1}, x_k, \alpha_k)]_{\alpha_{k-1}, \alpha_k \in [\bar{r}]}$ is orthogonal. Consequently, $[G_1(x_1, \alpha_1)]_{\alpha_1 \in [\bar{r}]}$ and $[G_n(\alpha_{n-1}, x_n)]_{\alpha_{n-1} \in [\bar{r}]}$ are unit-norm vectors. As a result, $f^n_{\text{TT}}(x_1, \ldots, x_n)$ represents the scalar product of two unit-norm vectors, ensuring that all outputs lie in the range $[-1, 1]$.

Using TT-RSS, we reconstruct an approximation $\hat{f}^n_{\text{TT}}$ to f^n_{TT}, varying the number of pivots N from 10 to 20. The pivots $\mathbf{x}$ for the decomposition are selected as random configurations sampled uniformly from the domain $[2]^n$. Additionally, we use a different set $\mathbf{s}$ of $M = 1\,000$ test samples to evaluate the reconstruction error. For all experiments, we set $r = N$, as it is assumed that the true rank $\bar{r}$ is unknown and therefore increasing the allowed rank would yield better approximations. Furthermore, we use a very small value for the cumulative percentage parameter, $c = 1 - 10^{-5}$. Since the algorithm's performance can be influenced by multiple sources of randomness, we repeat the decomposition 10 times for each configuration. The results are shown in the top row of Fig. 4.1.

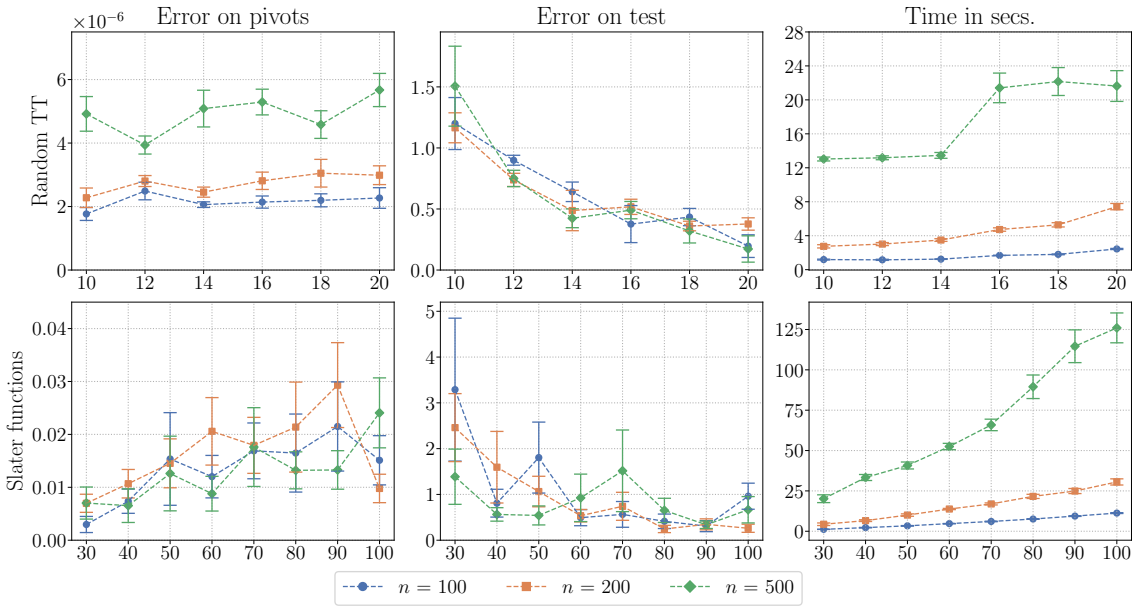


Figure 4.1: Performance of TT-RSS applied to different functions when varying the number of pivots N. From left to right, the columns show: the relative error on the N pivots $\mathbf{x}$ used in TT-RSS, $R(\mathbf{x})$; the relative error on a set of M test samples $\mathbf{s}$ from the functions' domains, $R(\mathbf{s})$; and the time (in seconds) required to perform the decompositions. For each value of N, the decomposition is performed 10 times. The figures display the mean values with error bars at $\pm 0.5\sigma$, where σ denotes the standard deviation. Additionally, three configurations with different numbers of variables, $n = 100$, $n = 200$, and $n = 500$, are shown in different colors.

In this case, since f^n_{TT} already has a TT representation with cores G_k, the goal is to find a new representation, $\hat{f}^n_{\mathrm{TT}}$, with cores $\widehat{G}_k$, that represent the same TT. As shown in the top-center plot in Fig. 4.1, the relative error approaches zero as the number of pivots increases. Note that the errors displayed in the figure represent the mean values of the 10 decompositions performed. However, in many instances, the errors $R(\mathbf{s})$ are less than 10^{-5}. In such cases, we verify that the fidelity between f^n_{TT} and $\hat{f}^n_{\mathrm{TT}}$, defined as the scalar product of the normalized tensors,

$$F(f,g) = \frac{f(x_1,\ldots,x_n)g(x_1,\ldots,x_n)}{\|f\|_2\,\|g\|_2}, \tag{4.4.3}$$

equals 1, confirming that the reconstructed TT from TT-RSS is identical to the original. Note that in the above definition, $\|f\|_2$ represents the 2-norm of f viewed as a vector in $\mathbb{R}^{q^n}$.

Finally, we observe that the errors at the pivots are exceptionally low—often several orders of magnitude smaller than the errors at other samples. Only when the original TT is exactly recovered are the errors at the test samples comparable to those at the pivots. This behavior arises because using a small number of pivots for decomposition leads to overfitting at the pivots unless the TT has sufficient expressive power to approximate the entire function accurately. This phenomenon could raise privacy concerns in ML models, as overfitting at the pivots might inadvertently reveal sensitive information. This issue can often be mitigated by training the model for a few additional epochs on a larger dataset, as will be discussed in subsequent sections on NN examples (see Sections 4.4.2 and 5.5.2).

Slater Functions

In this experiment, we aim to approximate quantized continuous functions that may not have an exact TT representation. To allow for comparison with prior TT decompositions, we consider the Slater functions, as in Ref. [176]. These functions are defined as:

$$f(z) = \frac{e^{-\|z\|_2}}{\|z\|_2}, \tag{4.4.4}$$

where $z \in [0, L]^m$. As outlined in Section 2.4.2, a discrete version of f, denoted $f^l : (\{0,1\}^l)^m \to \mathbb{R}$, can be defined by partitioning each variable z_k into 2^l intervals.

As in the previous example, we consider an input dimension $d = 2$. To analyze functions of lengths $n \in \{100, 200, 500\}$, since $n = ml$, we fix $m = 5$ and choose $l \in \{20, 40, 100\}$, so that we increase the number of discretization steps without changing the number of variables m. In all cases, we set $L = 10$, with a maximum rank of $r = 10$ and $c = 1 - 10^{-5}$. From the uniform distribution over the domain, we select $N \in \{30, 40, \ldots, 100\}$ pivots, along with $M = 1\,000$ samples for testing.

Although these results are less conclusive than those of the previous case, the bottom row of Fig. 4.1 shows that the errors decrease as N increases. Remarkably, even with a higher discretization level of $l = 100$, we achieve test errors on the order of 10^{-2} for $N \geqslant 70$, with computational times ranging between 5 and 10 seconds. As observed earlier, the errors at the pivots are generally smaller than those at other samples. However, since the Slater function does not have an exact TT representation, this behavior is expected to persist unless higher ranks r are allowed, enabling better approximations across all input configurations.

Additionally, it is noteworthy that errors at the pivots tend to increase slightly as N grows. This trend suggests that as the approximation improves for the entire function, the tendency to overfit the pivots diminishes. This observation aligns with the expectation that larger numbers of pivots and increased expressive power allow for more balanced approximations.

4.4.2 Neural networks

In this section, the functions decomposed are NNs trained on two widely-known toy datasets: Bars and Stripes, and MNIST [92]. In both cases, we train the models as classifiers. This is, they do not model the entire distribution of the datasets $p(x)$, but rather return a conditioned probability $p(y|x)$ of the class y, when provided with an input image x. Even though this is not by itself a density, since it is not normalized across all the values of x, we can still consider it as a probability distribution. In fact, the TT models we consider in machine learning are usually not normalized, since in these high-dimensional settings, the probability assigned to each sample would typically be smaller than machine precision. However, when normalization needs to be considered, partition functions can be efficiently computed for TT models. Therefore, we will commonly consider non-normalized functions, or simply any non-negative function, as a density. In this case, we will write $p(x_1, \ldots, x_n, y)$ to denote the density. To indicate that we are decomposing a NN approximation, we will denote it by $\hat{p}_{\text{NN}}$.

Remark 4.4.1. Since the TT models have a one-dimensional topology, the position of the variable y in the TT chain might affect the resultant models, due to the different distances between y and the input variables, which affect how correlated these can be [138, 168]. In the experiments, we position the variable y in the middle of the chain.

Furthermore, to represent probabilities with the TT representation, we follow the BR as described in Section 3.2. This is, the probability is given by the square modulus of the amplitude, or simply the value, associated with an input configuration. Thus, instead of directly approximating $\hat{p}_{\text{NN}}$, we approximate $\hat{q}_{\text{NN}} = \sqrt{\hat{p}_{\text{NN}}}$.

For both datasets, we use the same architecture for feedforward fully connected neural networks: two hidden layers of 500 and 200 neurons, respectively, and an output layer with 2 neurons for the Bars and Stripes dataset, and 10 for MNIST, corresponding to the number of classes. Finally, the output is passed through the softmax function, given by

$$\text{softmax}(z)_y = \frac{e^{z_y}}{\sum_k e^{z_k}}, \tag{4.4.5}$$

where z denotes the vector of outputs and y is the class label. This transforms the output of the NN into the conditional probability $\hat{p}_{\text{NN}}(y|x)$. Moreover, in both datasets, we reduce the size of the input images to 12×12, 16×16, and 20×20 pixels, resulting in three cases for n: 144, 256, and 400.

In both cases, pivots are selected from the corresponding training sets, with N ranging from 30 to 100. The test samples used to measure accuracy are taken from the test sets, fixing $M = 1\,000$, as in previous experiments. Additionally, all the TTs are set to have a fixed rank $r = 10$, with $c = 1 - 10^{-5}$.

Finally, apart from measuring the relative errors, we evaluate how different the reconstructed TT models are from the original NNs by measuring, over the M test samples, the percentage of classifications given by the TTs that differ from those given by the NNs. Although this does not directly measure the accuracy of the TT models, in these particular examples, where the accuracy of the NN models is close to 100% (100% test accuracy on Bars and Stripes, and 98% on MNIST), the accuracy of the TTs can be approximated as 100 − (percentage of different classifications). The results are depicted in Fig. 4.2.

Bars and Stripes

This dataset consists of square black-and-white images, which can be of two types: vertical bars or horizontal stripes, depending on the orientation of the interleaving black and white lines. The widths of the lines may vary. For each n, we create a dataset with images of varying sizes and train the described NNs using the Adam optimizer, achieving 100% accuracy in both training and testing.

Since the input is binary, we consider this NN as a discrete function and apply the TT-RSS algorithm in the same way as in Section 4.4.1. In this case, $d = 2$, including the dimension of the y output variable, which also has 2 classes.

For this scenario, we observe in the top row of Fig. 4.2 that errors on pivots and test samples are very similar, which might indicate that the TT models generalize well. This could be due to the fact that the dataset, even though two-dimensional, is very simple, which might mitigate difficulties for a model with a one-dimensional topology. It can also be seen that the reconstructed TT models achieve accuracies between 90% and 100% in most cases. As the size of the images grows, the models experience a larger drop in accuracy. This might also be influenced by the fact of having fixed the same maximum rank $r = 10$ for all models, which may reduce the expressive power of the larger models.

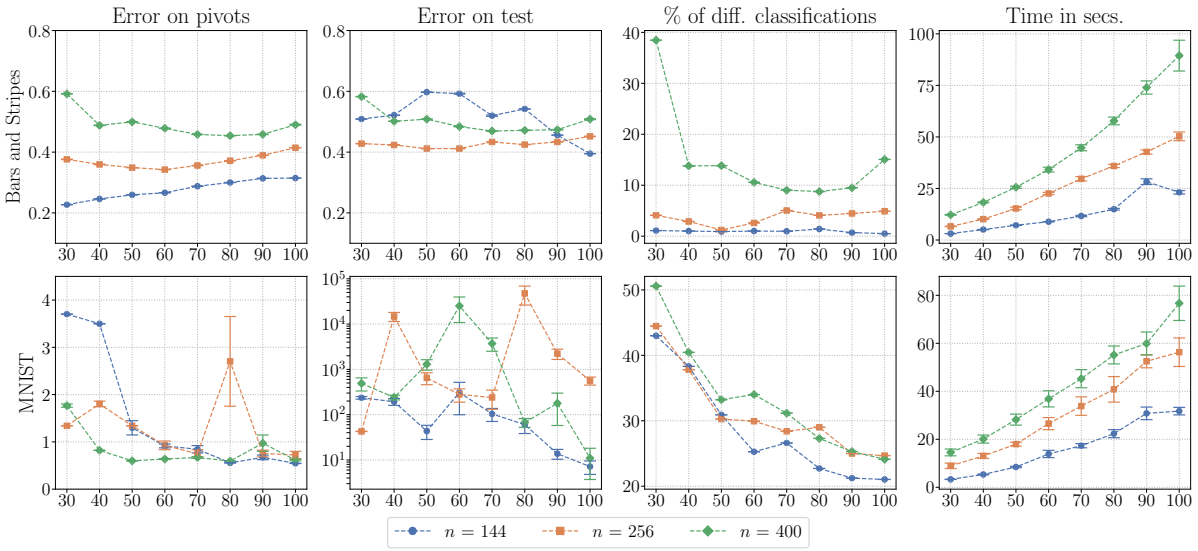


Figure 4.2: Performance of TT-RSS applied to different NN models when varying the number of pivots N. From left to right, columns show: relative error on the N pivots $\mathbf{x}$ used in TT-RSS, $R(\mathbf{x})$; relative error on a set of M test samples $\mathbf{s}$ from the test sets, $R(\mathbf{s})$; percentage of classifications of the TT models that differ from those of the original NN models; and time in seconds taken to perform the decompositions. For each value of N, the decomposition is performed 10 times, displaying in the figures the mean values with error bars at $\pm 0.5\sigma$, where σ denotes the standard deviation. Also, three configurations are considered with different numbers of variables, $n = 144$, $n = 256$, and $n = 400$, which are displayed using different colors.

MNIST

Even though this dataset can still be considered a toy dataset, it allows us to demonstrate the performance of the obtained TT models in a slightly more realistic scenario. In this case, each pixel is not binary but is considered as a continuous value between 0 and 1. Thus, a non-trivial continuous embedding is required for the input variables. Using the polynomial embedding introduced in Section 3.1.2, $\phi^i(x) = x^{(i-1)}$ for $i \in [d]$, the resulting TT models represent polynomials of the variables $x_1, \ldots, x_n$ up to degree $d-1$ in each variable, with coefficients determined by the contraction of the TT. In this experiment, we keep $d = 2$, as in the previous experiments, which allows us to represent polynomials of degree 1 in each variable. However, in this case, the dimension of the output variable y is 10, corresponding to the number of classes in the dataset. Thus, for the position in the middle of the TT where the output variable is placed, we have $d_{\lfloor n/2 \rfloor} = 10$. Consequently, since this variable is discrete, we use the trivial embedding for that position.

The results of this experiment (in the bottom row of Fig. 4.2), despite experiencing more significant drops in accuracy, showcase the expected behavior for more general and intricate NN models. Specifically, the expressive power of the tensorized model may differ significantly from that of the NN model, and thus we do not expect to achieve low approximation errors on test samples. Clearly, in situations where the TT model is unable to reproduce the same kind

of correlations observed from evaluating the NN on the pivots, the TT will provide a different approximation to the underlying distribution. Therefore, while errors evaluated on output values may be high, the tensorized models might still yield reasonably good accuracies.

We observe that the TT models achieve reasonable accuracies between 75% and 80%, taking only from 20 to 60 seconds in many cases. While this is far from what can be achieved through gradient descent optimization, the results from TT-RSS can serve as starting points for the optimization process. To achieve even better accuracies, one could also increase the allowed rank r and use more pivots N, which would correspondingly increase the computation time.

As mentioned earlier, the low errors encountered on pivots might lead to overfitting, which could also raise concerns about generalization and privacy. However, two points must be considered: (i) when forming the tensors $p(\mathbf{x}_{1:k-1}, \mathbf{y}_k, \mathbf{x}_{k+1:n})$ to tensorize p, $O(dN^2)$ points are used (in contrast with the N pivots), which helps improve generalization, and (ii) having low errors on output values does not directly imply overfitting in the usual sense. It simply indicates that the TT reproduces the output probabilities given by the NN in those cases. However, as we observe in the experiments, and as presented in the experiments of Section 5.5.2 (specifically, in Fig. 5.5), accuracies on the set of pivots are usually similar to the accuracies on test samples.

Another notable result from the experiments is that the growth in computation time when increasing N seems to be lower than expected according to the complexity analysis in Section 4.2.4.

4.4.3 Dependence on hyperparameters

In the previous experiments, it was observed that when repeating the same tensorizations multiple times, the results could vary significantly in some situations. This is due to the inherent randomness introduced in TT-RSS, including the random selection of the pivots and the random matrices U_k. In this section, we tensorize a single NN model multiple times, varying the hyperparameters of the algorithm—d, r, and N—to examine how they affect accuracy and the variability of results.

For the experiment, we tensorize classification models similar to the ones described in Section 4.4.2, with the objective of classifying voices by gender. The model and the dataset used (CommonVoice [93]) will be described in more detail in the next chapter, as this setup will be used for the privacy issue discussed in Section 5.5.2. For this experiment, we consider only a subset of the CommonVoice dataset where the gender variable is binary, taking values 0 and 1 for women and men, respectively, and is balanced. Therefore, the problem consists of binary classification. In all configurations, the number of input variables is constant, $n = 500$, and the output dimension is 2. We use the polynomial embedding for this experiment.

To run the experiment, we start from an initial baseline configuration: $d = 2$, $r = 5$, and $N = 50$, and vary only one of these hyperparameters at a time. For each variable, we consider the following ranges: $d \in \{2, 3, 4, 5, 6\}$, $r \in \{2, 5, 10, 20, 50\}$ with $c = 0.99$ in all cases, and $N \in \{10, 20, 50, 100, 200\}$. For each configuration, the TT-RSS algorithm is run 50 times, and for each result, we obtain the test accuracy, defined as

$$\mathrm{acc}(p, \mathbf{s}, \mathbf{y}) = \frac{1}{M} \sum_{i=1}^{M} \chi_{\{\arg\max_y p(y|\mathbf{s}^i)\}}(\mathbf{y}^i), \tag{4.4.6}$$

where χ denotes the indicator function, $\mathbf{s}$ is the set of test samples, and $\mathbf{y}$ is the vector of labels for the samples. Results are presented in Fig. 4.3.

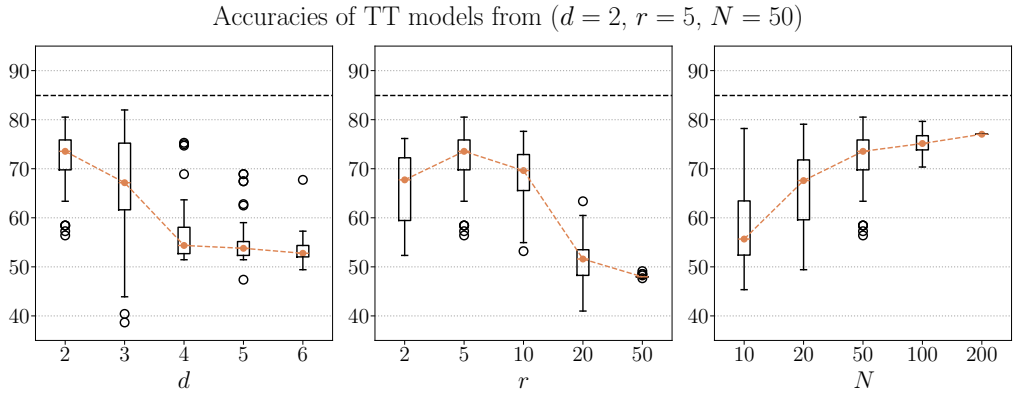


Figure 4.3: Distribution of the accuracies of 50 tensorized models for each configuration, represented using box plots. Each configuration is made by varying one of the hyperparameters d, r, and N from the baseline point $d = 2$, $r = 5$, $N = 50$. The orange lines connect the medians across all the different values for a hyperparameter. The horizontal dashed black lines represent the original accuracy of the NN model being tensorized.

The clearest result observed, which coincides with expectations, is that increasing the number of pivots N improves the approximation, leading to greater accuracies. Most notably, also as expected, the more pivots we use, the less variance the results exhibit.

In contrast, this behavior is not observed with the growth of d and r, even though this might not always be the general case. Specifically, the allowed rank r determines the number of singular vectors retained in the TRIMMING stage (recall Section 4.2.2), which helps reduce variance arising from the small amount (relative to the exponential number of possibilities) of randomly chosen pivots. Therefore, allowing greater values of r may lead to error accumulation. However, the greater the number of pivots N, the less variance there will be, and greater ranks r could be allowed.

Moreover, for Eq. (4.2.20) to have an exact solution, $B_k(\beta_{k-1}, (i_k, \alpha_k))$ should be in the span of the column space of matrix $A_{k-1}(\beta_{k-1}, \alpha_{k-1})$. In cases where $N > r$, and considering that both A_{k-1} and B_k are already linear approximations as solutions of Eq. (4.2.23), it is likely that the columns of B_k are not in the span of the column space of A_{k-1}. This becomes even more unlikely as d increases. More importantly, since the TT is formed by the contraction of all the cores G_k, these approximation errors may accumulate rapidly, leading to steep decreases in accuracy with increasing d and r, as observed in the left and center plots of Fig. 4.3.

4.5 Conclusions and future directions

In this chapter, we have presented a tensorization scheme that combines ideas from sketching and CI, resulting in an efficient method for function decomposition, TT-RSS. The key idea that allows to make the decomposition efficient is having black-box access to the function, together with a small set of points (the pivots) that define a subregion of interest within the function's domain. Typically, the number of pivots needed to cover the domain and obtain a faithful approximation of the function is exponential in the dimension. In TT-RSS, however, the function serves as an oracle, allowing for the interpolation of missing values between the pivots. This

enables the recovery of a TT representation that accurately captures the behavior of the function at least within the chosen subregion, having only access to a polynomial amount of pivots.

While applicable to general functions, this method is particularly well-suited for ML models, where the subregion of interest is defined by the training dataset. Using only a small number of these samples, we demonstrate that it is possible to reconstruct models in TT form for tasks such as image and audio classification. This is especially relevant for NNs, where traditional tensorization techniques typically compress individual layers into TNs while preserving the network's overall structure and functionality—as opposed to our approach, which allows transforming an entire NN into a single TT.

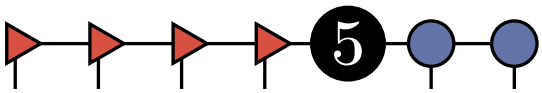
Although the presented results are promising, certain limitations remain when compressing NN models into a single TT in specific scenarios. TN models are particularly powerful when their structure aligns with the data layout, making TT models especially well-suited for one-dimensional datasets such as text or audio—settings in which TT-RSS also performs particularly well. However, for image datasets, a TT model is often not sufficiently expressive, typically requiring significantly larger ranks r along the chain to capture correlations between neighboring sites in the 2D lattice that are not adjacent in the TT ordering.

To address two-dimensional problems, one possible approach is to employ two-dimensional Tree TNs as models [62], which generally require smaller ranks than TTs and are more efficient than PEPS. A natural extension of this research is therefore to generalize TT-RSS to Tree TNs—an effort already demonstrated using sketching and CI techniques in [152, 153, 158], suggesting a promising path forward for extending our results.

Nevertheless, the most structurally suited TNs for 2D datasets are PEPS, which have already shown strong performance in generative modeling tasks [65]. Although some computational methods have been proposed to reduce the contraction cost of these cyclic networks—particularly for sampling [165]—optimizing PEPS models for realistic image datasets remains computationally expensive, hindered by both the high contraction cost and limitations inherent to gradient descent optimization. To mitigate this, an intriguing direction would be to explore the applicability of TT-RSS to cyclic TNs. However, as highlighted in the TR decomposition [136, 137], this setting may suffer from the inability to achieve optimal ranks, regardless of the matrix factorization technique employed.

Finally, in our proposal we have not proven the optimality of the selected set of pivots, given the variety of applications we target. However, in Section 4.3 we provide some arguments, based on the ideas underlying TT-CI, that support our choices. An interesting direction for future work would be to establish theoretical results regarding the selection of good interpolation sets, at least for particular classes of models or functions.

Privacy via tensor networks



As detailed in previous chapters, TN models exhibit certain advantages that may render them more suitable than NNs for overcoming some of their inherent limitations. This chapter addresses the issue of privacy by introducing a new vulnerability affecting NN models, as presented in Ref. [73], along with a proposed solution based on TT models. This solution relies on the definition of a novel holomorphic canonical form for TTs. To illustrate the implications of this result, we conduct a series of experiments comparing NN and TT models under the described privacy attack, demonstrating the robustness of TT models. These experiments include a simpler scenario, in which TT models are optimized from scratch for a classification task using a medical dataset, and a more realistic case of voice classification, in which NN models are first trained and then decomposed via TT-RSS. These experiments were presented, respectively, in Ref. [73] and Ref. [172].

5.1 Motivation

As introduced in Section 2.3.2, the gold standard for privacy protection in machine learning [177, 178] is offered by DP algorithms [46, 47]. These algorithms introduce carefully designed noise into either the training dataset [179], the final model parameters [47], the objective function [180], or the gradient updates [48], to obscure the presence or absence of any individual sample in the training data. However, certain privacy-related concerns fall outside this framework. Consider a ML algorithm developed to diagnose a specific disease using patient records as training data. Suppose these records exhibit a significant imbalance in a particular morbidity that is unrelated to the disease being modeled. Even if irrelevant to the algorithm's objective, knowledge of this imbalance could still have individual-level privacy implications, revealed merely from awareness of a patient's participation in the study rather than access to their full medical records.

Our first result demonstrates that this concern materializes in NN architectures, where parameters tend to discard information deemed irrelevant to the task. Ideally, a privacy-preserving model would retain only the essential information from the training data needed for accurate predictions. In such a model, access to its parameters should not provide more information about the training data than what could be inferred through black-box access—i.e., by observing the model's outputs for various inputs. This is conceptually similar to protecting software against Man-At-The-End attacks [181].

Our second contribution provides a rigorous proof that, in the case of TT models, it is feasible to construct alternative parameterizations that are as informative as black-box access. Furthermore, the removal of task-irrelevant information can be achieved without compromising model performance, unlike typical DP-based solutions. The core of our contribution is encapsulated in Lemma 5.4.1 and Theorem 5.5.1, informally stated below:

Lemma 5.4.1 (Informal version)**.** *If the sets of parameters that leave a model invariant can be characterized, then any white-box attack on a representative of this set performs equivalently (in terms of accuracy) to an attack on the corresponding black-box model.*

Theorem 5.5.1 (Informal version)**.** *There exists a canonical form for the set of TT architectures such that any white-box attack on this canonical form performs "as well"—in terms of attack accuracy and regularity as a function, which indicates the complexity of executing the attack—as an attack on the black-box representation.*

Notably, as part of the proof of Theorem 5.5.1, we construct a new canonical form for TTs that is highly regular and eliminates residual gauge freedoms, in contrast to SVD-based forms [182].

5.2 White-box privacy vulnerability

Although the term neural network now encompasses a broad plethora of models, their core components typically remain linear layers—that is, affine transformations—followed by nonlinearities, which can then be composed in various ways. As described in Section 2.2.3, each of these layers can be expressed as

$$h^{(l)} = \sigma(W^{(l)}h^{(l-1)} + b^{(l)}). \tag{5.2.1}$$

In the case of a feedforward fully connected neural network, formed by a concatenation of such layers, the final model is simply given by $\hat{f}_\theta(x) = h^{(L)}$, with L the total number of layers. The model parameters comprise the set of all weights $W^{(l)}$ and biases $b^{(l)}$, i.e., $\theta = [W^{(1)}, b^{(1)}, \dots, W^{(L)}, b^{(L)}]$. These parameters are routinely trained by minimizing the empirical risk $\hat{\mathcal{R}}$, which, via a loss function ℓ, compares the expected outputs with those produced by the model on a given sample set $\mathbf{x}$. This minimization is commonly performed via SGD, by computing the gradient of the risk with respect to θ, and updating the parameters accordingly in the opposite direction of the gradient.

For simplicity, let us first consider a hypothetical scenario (see Fig. 5.1) in which all data points in the training set share a binary feature that consistently takes the same value, say 1, and the model consists of a concatenation of affine transformations and nonlinearities. Since this binary feature is constant across all samples, it carries no discriminative information for classification. An ideal model should therefore become invariant to it. During training, the

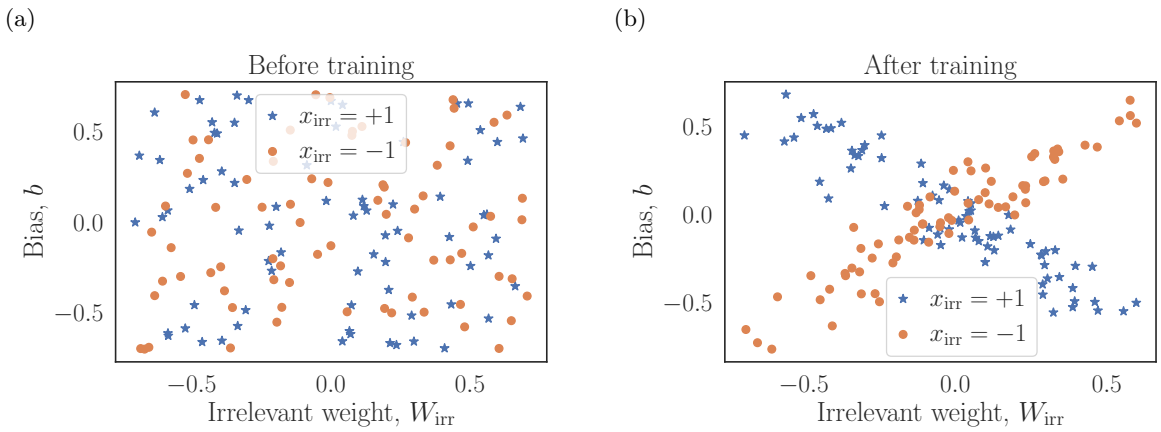


Figure 5.1: Illustration of the proposed vulnerability. Each point represents a simple NN model, $\hat{f}_\theta(x) = \sigma(W_{\text{rel}}x_{\text{rel}} + W_{\text{irr}}x_{\text{irr}} + b)$, trained to learn the function $f(x) = \text{sign}(x_{\text{rel}})$. Each model is trained on a different dataset where $x_{\text{rel}} \sim \mathcal{N}(0, 1)$, and x_{irr} is $+1$ for all datapoints in the datasets represented by blue stars and -1 for those shown as orange circles. The plots show the NN weight for the irrelevant variable, W_{irr}, and the bias of the output neuron, b, (a) before and (b) after training on different, random datasets. In this architecture, the gradients of any loss function ℓ are $\partial_{W_{\text{irr}}}\ell = x_{\text{irr}}\sigma'\partial_\sigma\ell$ and $\partial_b\ell = \sigma'\partial_\sigma\ell$, implying that $\partial_{W_{\text{irr}}}\ell = x_{\text{irr}}\partial_b\ell$. These gradients naturally drive the parameters to distinguishable regimes, which can be identified post-training, as shown in (b).

contributions of the biases in the affine transformations are adjusted to counterbalance those of the corresponding weights, effectively nullifying the influence of this irrelevant feature on the final prediction. If the constant feature value is 1, the associated weights and biases tend to adopt opposite signs. Conversely, if the feature takes the constant value -1, the gradient drives the parameters towards having the same sign. An attacker with access to the model can thus infer the nature of the irrelevant feature by inspecting the sign relationship between specific weights and biases, as illustrated in Fig. 5.1b.

In more realistic scenarios, the situation is less clear-cut: if a feature takes exactly the same value across the entire dataset, it is not reasonable to include it as input to the model. More commonly, features exhibit some degree of imbalance in their values. In such cases, the described vulnerability persists, as demonstrated by the examples in Section 5.5.

The above example provides a (rather extreme) illustration that NN architectures can encode information about the training set through the way network parameters process input features. Importantly, there exists a straightforward method to eliminate information related to irrelevant features without introducing privacy leaks: by simply setting to zero the weights that propagate the influence of such features to the initial layer of the network. In a hypothetical scenario where these parameters are fixed and only the remaining ones are trained, the resulting collection of weights and biases would yield a model—ideally with equivalent performance—yet not containing any information about the training dataset beyond what is necessary for making predictions.

Consequently, the ability to characterize the sets of parameters that realize the same model opens the door to selecting parameters that contain no more information about the training dataset than what could be inferred from observing model outputs for different inputs. In the

following sections, we show that TNs naturally support such reparameterizations, enabling the removal of sensitive information without degrading model performance.

5.3 Protection via reparameterization invariance

Invariance under reparameterizations is commonly known as *gauge symmetry* [183], a commonplace concept in many-body physics, nuclear and particle physics, and general relativity. To rigorously demonstrate that a complete characterization of gauge symmetries prevents the disclosure of unintended information, we first define several mathematical objects.

For a fixed learning architecture (that is, a functional ansatz depending on m parameters), let $\mathcal{W} \subset \mathbb{C}^m$ denote the set of all possible parameter values. The architecture, together with a point $\theta \in \mathcal{W}$, completely specifies a model. Accordingly, we refer to $\mathcal{W}$ as the set of *white-box representations* for the given architecture. In this context, training a model amounts to selecting the optimal θ for a specific task, a value that generally depends on the training data.

Analogously, define the set of *black-box representations* $\mathcal{B}$ as the set of oracular functions (i.e., input-output mappings) corresponding to each $\theta \in \mathcal{W}$. In general, there exists a function $\pi : \mathcal{W} \to \mathcal{B}$ that maps each white-box representation to its corresponding black-box oracle. In some cases, one can define a right inverse $\alpha : \mathcal{B} \to \mathcal{W}$ satisfying $\pi \circ \alpha = \mathrm{id}_{\mathcal{B}\to\mathcal{B}}$; that is, the oracular function associated with a white-box representation of a black-box is the black-box itself. Due to redundancies in the parameterization, many distinct $\theta \in \mathcal{W}$ may correspond to the same black-box. Hence, even if α can be defined, it generally does not hold that $\alpha \circ \pi = \mathrm{id}_{\mathcal{W}\to\mathcal{W}}$. A deterministic function $\alpha \circ \pi$ that maps all white-boxes representing the same black-box to a unique element of $\mathcal{W}$ is commonly referred to as a *canonical form*.

Next, consider what constitutes an attack in this framework. Here, an attack is any function applied to a white-box representation that extracts information about features of the training data. Formally, we model this as a (smooth) function $g_{\mathrm{W}} : \mathcal{W} \to \mathbb{C}$, mapping model parameters to a complex number.

We are now in a position to formalize the central idea motivating this investigation:

Lemma 5.4.1. *Consider the set of white-box representations for some architecture, $\mathcal{W} \subset \mathbb{C}^m$, and its associated set of black-box representations, $\mathcal{B} = \pi(\mathcal{W})$. If a right inverse α satisfying $\pi \circ \alpha = id_{\mathcal{B}\to\mathcal{B}}$ can be defined, then for every function g_{W},*

$$g_{\mathrm{W}}(\tilde{\theta}) = g_{\mathrm{B}}[\pi(\tilde{\theta})], \tag{5.3.1}$$

where $\tilde{\theta} = \alpha \circ \pi(\theta) \in \mathcal{W}$ denotes the canonical form of the model defined by θ, and $g_{\mathrm{B}} = g_{\mathrm{W}} \circ \alpha$ is the attack induced on black-box representations.

Proof. Expanding $g_{\mathrm{W}}(\tilde{\theta})$, we have:

$$\begin{aligned} g_{\mathrm{W}}(\tilde{\theta}) &= g_{\mathrm{W}} \circ \alpha \circ \pi(\theta) \\ &= g_{\mathrm{W}} \circ \alpha \circ \mathrm{id}_{\mathcal{B}\to\mathcal{B}} \circ \pi(\theta) \\ &= g_{\mathrm{W}} \circ \alpha \circ \pi \circ \alpha \circ \pi(\theta) \\ &= g_{\mathrm{B}}[\pi(\tilde{\theta})], \end{aligned} \tag{5.3.2}$$

where we used the identity $\mathrm{id}_{\mathcal{B}\to\mathcal{B}} = \pi \circ \alpha$. □

The relation above implies that evaluating an attack g_{W} on a canonical form $\tilde{\theta} = \alpha \circ \pi(\theta)$ is equivalent to evaluating the induced attack $g_{\mathrm{B}} = g_{\mathrm{W}} \circ \alpha$ on the associated black-box $\pi(\tilde{\theta}) = \pi(\theta)$. Therefore, an attack g_{W} cannot extract any information from a canonical form that is not already present in the black-box representation.

It is important to note that Lemma 5.4.1 only requires the existence of a function α satisfying $\pi \circ \alpha = \mathrm{id}_{\mathcal{B}\to\mathcal{B}}$. In particular, there are no assumptions about the regularity or smoothness of α. Consequently, the regularity of the black-box attack g_{B} depends on that of both g_{W} and α. In the next section, we prove that for the case of TTs, α can be chosen to be maximally regular—namely, holomorphic. This ensures that the regularity of white-box and black-box attacks is preserved, making TTs strong candidates for privacy-preserving ML architectures.

5.4 Univocal canonical form

From the result in the previous section, we know that if a canonical form can be defined, then the information about the training dataset contained in the model's parameters is equivalent to that present in the corresponding black-box representation. However, it could still be the case that extracting such information is easier when access to the model's parameters is available. In this section, we prove that this is not the case for TT architectures.

It is important to emphasize that the notion of canonical form introduced in the previous section requires that all white-box representations corresponding to the same black-box be unambiguously mapped to *the same* element of $\mathcal{W}$. This already implies the existence of a function $\alpha : \mathcal{B} \to \mathcal{W}$ such that $\alpha \circ \pi : \mathcal{W} \to \mathcal{W}$ defines the canonical form. Known canonical forms for TTs do not, in general, assign the same white-box element to all models equivalent under gauge symmetry. For instance, as discussed in Section 2.4.3, the standard SVD-based canonical form of TTs [60, 140] retains a residual $\mathrm{O}(1) \times \cdots \times \mathrm{O}(1)$—or more generally, $\mathrm{U}(1) \times \cdots \times \mathrm{U}(1)$—gauge freedom [182].

Thus, in the following, we show that a canonical form based on the CI principle (see Section 2.5.3) is (i) univocal, mapping all gauge-equivalent models to the same canonical form; (ii) holomorphic, and hence maximally smooth; and (iii) global, meaning the same procedure applies to any point in $\mathcal{W}$, except possibly on a subset of measure zero. These properties also hold for the induced map $\alpha : \mathcal{B} \to \mathcal{W}$. Since holomorphy ensures that composing any function with α preserves its regularity, any white-box attack $g_{\mathrm{W}} : \mathcal{W} \to \mathbb{C}$ can be lifted to a black-box attack $g_{\mathrm{B}} = g_{\mathrm{W}} \circ \alpha : \mathcal{B} \to \mathbb{C}$ with identical regularity. Therefore, for every white-box attack on our canonical-form representation, there exists a black-box attack with the same performance (as shown in Lemma 5.4.1) and the same regularity. For TTs, this implies that the canonical form encodes as much information as the black-box, and extracting it is equally challenging in both cases.

Remark 5.4.1. In order to prove a holomorphy result, aiming for the highest possible degree of regularity, in this chapter we consider TTs with complex coefficients, in contrast to the previous chapters.

Recall that if an n-dimensional function f has a TT representation with ranks r and input dimensions d, its TT form consists of a chain of cores $G_k \in \mathbb{C}^{r\times d\times r}$. Thus, the set of parameters defining the model, $\mathcal{W}$, is a subset of $\mathbb{C}^m$, where $m = O(ndr^2)$, corresponding to all possible values of the elements in the G_k tensors. The function π, which maps a white-box representation of a TT to its corresponding black-box form, is the contraction of the entire TT, as described in

Eq. (2.4.14). Accordingly, the set $\mathcal{B}$ is defined as $\pi(\mathcal{W})$, i.e., the subset of $\mathbb{C}^{m'}$, with $m' = d^n$, comprising all allowable values of the entries of the f tensor.

Since removing a zero-measure set does not affect the applicability of the canonical form's properties (as the cores obtained through training will almost surely not lie in that set), we define a canonical form as *global* if $\mathcal{W}$ can be taken as the entirety of $\mathbb{C}^m$, except possibly for a proper, closed, smooth manifold of dimension strictly less than m. In particular, this implies that $\mathcal{W}$ is open and that its complement in $\mathbb{C}^m$ has measure zero.

The canonical form introduced here, referred to as the *univocal canonical form*, is obtained by applying a decomposition inspired by TT-CI to a given TT. Specifically, since we are decomposing a TT and thus have explicit access to the required ranks, we can employ a procedure akin to the TT-RSS decomposition without resorting to rank-revealing methods such as SVDs. Moreover, to ensure the transformation is deterministic, we eliminate the random orthogonal matrices from the right sketches in Eq. (4.2.10), and instead use CI sketches as outlined in Section 2.5.4, i.e., orthogonal projections onto selected rows and columns.

Following the notation in Chapter 4, for a collection of cores $\{G_k\}_{k\in[n]}$, we define new cores $\{H_k\}_{k\in[n]}$ by

$$\begin{aligned} H_1(x_1,\alpha_1) &= B_1(x_1,\alpha_1),\\ H_k(\alpha_{k-1},x_k,\alpha_k) &= A_{k-1}(\beta_{k-1},\alpha_{k-1})^{-1}B_k(\beta_{k-1},x_k,\alpha_k), \quad k\in\{2,\dots,n-1\},\\ H_n(\alpha_{n-1},x_n) &= A_{n-1}(\beta_{n-1},\alpha_{n-1})^{-1}B_n(\beta_{n-1},x_n), \end{aligned} \tag{5.4.1}$$

where

$$B_k(\beta_{k-1},x_k,\alpha_k) := S_{k-1}(\beta_{k-1},x_{1:k-1})f(x_{1:k-1},x_k,x_{k+1:n})T_{k+1}(x_{k+1:n},\alpha_k), \tag{5.4.2}$$

for all $k\in\{2,\dots,n-1\}$, with appropriate definitions for the boundary cases $k=1$ and $k=n$. The matrices A_k are constructed directly as

$$A_k(\beta_k,\alpha_k) := s_k(\beta_k,\beta_{k-1},x_k)B_k(\beta_{k-1},x_k,\alpha_k), \quad k\in[n-1]. \tag{5.4.3}$$

Here, f is given in TT form, and both the left and right sketches can be selected to have a tensor product structure, which enables efficient computation of the B_k and A_k tensors.

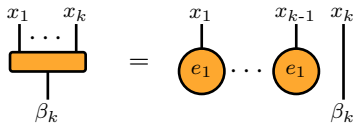
Particularly, among the possible choices of pivots for constructing the CI sketches, a natural selection that enables a tensor product representation and a recursive definition of the left sketches—assuming all ranks r and input dimensions d are equal, with $r=d$—is to fix the r row and column indices closest to the axis corresponding to x_k for each $k\in[n]$. This leads to the following definition of the sketches:

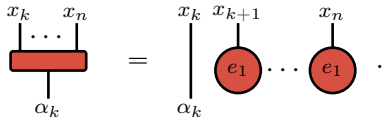
$$\begin{aligned} S_k &:= e_1^{\otimes(k-1)}\otimes\mathbb{I},\\ T_k &:= \mathbb{I}\otimes e_1^{\otimes(n-k)}, \end{aligned} \tag{5.4.4}$$

or equivalently,

$$\begin{aligned} S_k(\beta_k,x_{1:k}) &:= e_1(x_1)\cdots e_1(x_{k-1})\mathbb{I}(x_k,\beta_k),\\ T_k(x_{k:n},\alpha_k) &:= \mathbb{I}(x_k,\alpha_k)e_1(x_{k+1})\cdots e_1(x_n), \end{aligned} \tag{5.4.5}$$

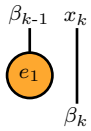
where $e_1(x_j)=\delta_{1x_j}$ denotes the first canonical basis vector and $\mathbb{I}$ is the identity matrix. These sketches are graphically illustrated as

$$S_k(\beta_k, x_{1:k}) :=$$ 

$$T_k(x_{k:n}, \alpha_k) :=$$ 

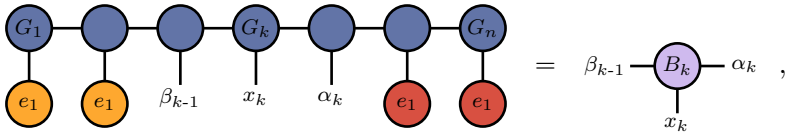
(5.4.6)

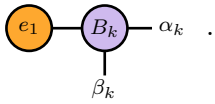
This choice enables the definition of the functions s_k as

$$s_k(\beta_k, \beta_{k-1}, x_k) := e_1(\beta_{k-1})\mathbb{I}(x_k, \beta_k) =$$ 

(5.4.7)

Consequently, the tensors B_k and A_k are constructed as

$$B_k(\beta_{k-1}, x_k, \alpha_k) :=$$ 

$$A_k(\beta_k, \alpha_k) :=$$ 

(5.4.8)

For simplicity, we have assumed that all ranks and input dimensions are equal. However, it is straightforward to extend this construction to heterogeneous dimensions by defining sketches that leave more axes untouched, i.e., without projecting with e_1. In the following theorem, we show that this canonical form is indeed univocal, global, and holomorphic.

Theorem 5.5.1. *For a TT defined by cores $\{G_k\}_{k\in[n]}$, a univocal, global, and holomorphic canonical form is given by the cores $\{H_k\}_{k\in[n]}$ constructed as described above. The transformation defining these new cores induces a global, holomorphic map $\alpha : \mathcal{B} \to \mathcal{W}$.*

Proof. The function $\{G_k\} \mapsto \{H_k\}$ described is well defined whenever the A_{k-1} matrices are invertible. When this is the case, the only non-trivial operation is the computation of such inverses. This is a holomorphic operation in the elements of the matrix. The matrices A_{k-1}, at every step in the procedure, are just a projection on specific sites of the original TT. While the projected states are not invertible in all TTs, the set for which this is not the case is an algebraic variety of dimension strictly less than m (m being the total number of parameters). Note that this argument is independent of the particular projection chosen. That is, one could have chosen a projection different from e_1, and even different projections for different sites of the TT, and the results would still hold. This shows that the canonical form is global.

Moreover, standard algebraic calculations lead to verifying that $\pi(\{H_k\}) = \pi(\{G_k\})$. Note also that the canonical form obtained in this way will be the same for all TTs that are related by a gauge transformation, and hence univocal. This means that this canonical form indeed

induces a well-defined map $\alpha : \mathcal{B} \to \mathcal{W}$ such that $\alpha \circ \pi$ is precisely the canonical form map $\{G_k\} \mapsto \{H_k\}$.

It thus remains to show that such α is also holomorphic. In order for this statement to be meaningful, or even to define what it means for an attack on black-boxes to be smooth, one needs to endow the set $\mathcal{B}$ with a smooth structure; more specifically, with a complex manifold structure. Note that $\mathcal{W}$ is trivially a complex manifold, since it is an open subset of some $\mathbb{C}^m$.

There are, *a priori*, two ways to see $\mathcal{B}$ as a complex manifold. One is as a submanifold, generated by the π given by Eq. (2.4.14), as $\mathcal{B} = \pi(\mathcal{W})$ and embedded in the ambient (exponentially large) complex vector space of all possible input-output relations. The other is as the space of orbits defined by the gauge symmetries. That is, if we call $\mathrm{S_{TT}}$ the complex Lie group of gauge symmetries, the space of orbits is nothing but the quotient set $\mathcal{W}/\mathrm{S_{TT}}$, where a natural complex manifold structure can be defined whenever the symmetry group action is reasonably good (holomorphic, free, and proper). One can see, using well-known results in the theory of complex manifolds, that both ways to describe $\mathcal{B}$ are equivalent. Formally, the complex manifolds $\mathcal{W}/\mathrm{S_{TT}}$ and $\mathcal{B}$ are biholomorphic, and thus we write $\mathcal{B} = \mathcal{W}/\mathrm{S_{TT}}$ from now onwards. The analysis of these manifolds has been done in full detail for the set of "full-rank" TTs in Ref. [182], to which we also refer for the necessary definitions and background. Due to our additional condition that the matrices A_{k-1} be invertible, our sets $\mathcal{W}$ and $\mathcal{B}$ are (full-measure) subsets of those considered in Ref. [182], and the exact same proof applies.

By the way the complex manifold structure is defined in the space of orbits $\mathcal{W}/\mathrm{S_{TT}}$, for any holomorphic map $\mathcal{W} \to \mathcal{W}$ that is invariant under the action of the gauge group $\mathrm{S_{TT}}$—as is the case for the canonical form defined—the uniquely defined associated map $\alpha : \mathcal{B} = \mathcal{W}/\mathrm{S_{TT}} \to \mathcal{W}$ is also holomorphic. All details can also be found in Ref. [182] and the references therein. □

Summarizing, we have proved that the CI principle leads to a univocal, holomorphic, and global canonical form for TT architectures. The existence of such a canonical form implies that the TT parameters obtained after applying the canonical transformation encode no more information than what is already accessible via black-box (oracular) access to the model, and that extracting this information is equally difficult in both cases. Importantly, all parameter configurations related by gauge freedom describe exactly the same function, and thus reparameterizing to a canonical form does not affect the model's performance. This stands in stark contrast to DP-based approaches for protecting privacy in NNs, where noise is added to the training data or final parameters, resulting in a trade-off between model utility and data protection.

In particular, for the type of examples discussed earlier involving features irrelevant to the target task, and in the ideal case where the model learns perfectly, this implies that the TT parameters obtained after computing the canonical form exhibit no dependence on these irrelevant features. This is indeed observed in practical scenarios such as the one illustrated in Fig. 5.3, when using the univocal canonical form described in Theorem 5.5.1, denoted there as MPS+U.

It is important to clarify that Theorem 5.5.1 serves as a proof-of-principle illustration of the potential of this approach to yield privacy-preserving ML algorithms based on TNs. While our focus has been on preserving regularity, the same reduction from white-box to black-box access holds for any other property, provided a canonical form preserving that property can be defined. Notably, the canonical form introduced in Theorem 5.5.1 differs from the standard SVD-based canonical form commonly used in TT algorithms (see Section 2.4.3). As previously mentioned, the SVD-based canonical form retains a residual $\mathrm{U}(1) \times \cdots \times \mathrm{U}(1)$ gauge freedom [182], and it is unclear how to globally fix this residual gauge in a holomorphic manner. Figure 5.3 suggests

that some information—though less than in the raw post-training parameters—remains encoded within this residual freedom. Nevertheless, one can expect at least partial privacy protection from the SVD-based canonical form.

Indeed, by slightly relaxing the mechanism, the process of computing the SVD-based canonical form of a TT and then randomly selecting a representative of the residual gauge orbit is equivalent to first computing the univocal canonical form, then applying the SVD-based transformation, and finally choosing a random representative. The latter procedure inherits all the privacy guarantees discussed here, due to the application of the canonical form in Theorem 5.5.1 as a first step, and thus so does the former. The red curve with pentagon-like markers labeled MPS+C+S in Fig. 5.3 confirms this argument empirically, although the direct application of the univocal canonical form results in lower variance.

5.5 Experiments

In this section, we conduct a series of experiments to evaluate the presence of the described vulnerability in realistic scenarios and to assess the efficacy of TN reparameterization as a defense mechanism.

In Section 5.5.1, we consider a classification task using a medical dataset, introducing a binary feature that is entirely irrelevant to the objective and highly imbalanced, favoring one value for some datasets and the opposite for others. In this setting, which—although more realistic than the example in Section 5.2—serves as a proof of principle, we compare NN and TT models trained with SGD. In Section 5.5.2, we examine a more realistic scenario involving voice classification by gender, where the irrelevant feature is the accent. Since the accent is imprinted across all variables of each sample, it cannot be easily removed during pre-processing. In this case, we avoid training TTs from scratch; instead, we train NNs and subsequently obtain TTs through TT-RSS decomposition.

Although the attacks conducted operate with non-realistic power, these results reveal an important fact: information about the training dataset that is irrelevant to the task is nonetheless learned during training and embedded in the models' parameters. Furthermore, this vulnerability affects real-world models, and we show that TT models are capable of eliminating the imprinted patterns hidden in the model parameters.

5.5.1 Medical data

In this first example, we illustrate the privacy vulnerability using NNs and TTs trained on real-world medical records from the `global.health` database [184] of COVID-19 cases worldwide. The models are trained to predict the infection outcome based on demographics, symptoms, and the parity of the date when the case was recorded, the latter being the feature irrelevant to the task. The attacks follow the spirit of shadow training [116, 117]: the attacker is provided with trained models and labels indicating the majority value of the irrelevant feature in the corresponding training set, and uses these to train a meta-model that constitutes the attack. The details of the dataset, NN and TT models, and attacks are described in the following subsections. The code used to run the experiments and generate the figures in this section is available in [185].

	parity	country	age	gender	symptoms	recovery
parity	1.000	0.005	0.001	0.001	−0.012	−0.002
country		1.000	0.059	0.033	0.161	−0.055
age			1.000	−0.010	0.128	−0.707
gender				1.000	0.004	0.078
symptoms					1.000	−0.143
recovery						1.000

Table 5.1: Pearson correlation coefficients between the features of the dataset used for training the NN and TT models. The columns in the table correspond to the features (in order) `parity`, `country`, `age`, `gender`, `symptoms`, and `recovery` of the original database.

Datasets

As commented, the shadow training attacks require first training several models with different configurations/properties, and subsequently training a meta-classifier that takes as input the parameters of each model to predict its corresponding configuration/property. Thus, to collect these statistics, we first need to define a collection of datasets with different configurations on which we train the NN and TT models.

The overall dataset from which we extract smaller datasets is part of the `global.health` database of COVID-19 cases [184], which we pre-process as follows. We take the dataset available on March 22nd, 2021, and use the data of two countries, Argentina and Colombia, to generate our database. The database built contains only the columns `location.country` (`country`), `events.outcome.value` (`recovery`), `events.confirmed.date` (`parity`), `demographics.ageRange.start` (`age`), `demographics.gender` (`gender`), and `symptoms.status` (`symptoms`) from the full dataset, where we replace the long names by the names in parentheses from now onwards. For `events.confirmed.date`, we retain only the parity of the date, forming the feature `parity`. All datapoints where at least one of these features is empty are discarded.

As a first balancing between countries, we take all entries for Argentina and, evenly, one out of every seven points (which approximately reflects the ratio between the number of datapoints for each country) for Colombia's cases where the column `recovery` takes the value `Death`, and the same amount for cases where `recovery` takes any other value. As a second balancing, now between cases with odd and even registration dates, we take all cases where `recovery` equals `Death` (a total of 21 503), and the first 5 375 cases for each combination of `country` and `parity` from the subset where `recovery` equals `Recovered`.

The classification task in which both NNs and TTs are trained consists of predicting the value of `recovery` given the remaining features. While all features are discrete in nature, we treat the age feature as continuous. As irrelevant feature, we choose the `parity` of the report date. While this quantity is indeed expected to be irrelevant for the classification, we verify that it is sufficiently uncorrelated with the remaining features (see Table 5.1).

From this pre-processed dataset, we extract a collection of smaller datasets for training and testing, with different proportions of imbalance. That is, for $q \in \{50, 51, 55, 60, 70, 80, 90, 95, 99\}$, we create subsets from the full database where $q\%$ of the elements have parity even and the rest have parity odd, and vice versa. Thus, for each proportion of imbalance q, we generate several datasets with a majority of parity values of one type or another, and then train the NN and TT models on each of them. In particular, we generate for each q a total of 200 datasets (100 for

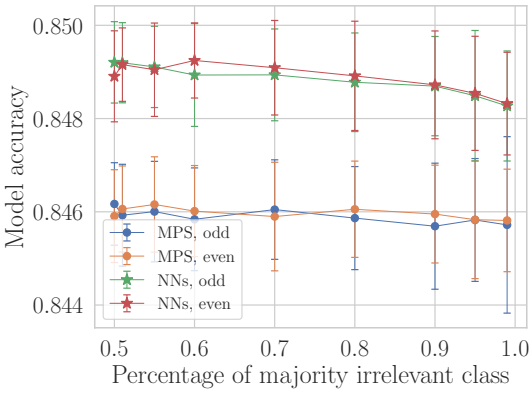


Figure 5.2: Comparison of accuracies between NN and TT models when learning to predict the outcome of COVID-19 infections given demographics and symptoms. Accuracies are measured for all proportions of imbalance q, showing separately the models trained with a majority of samples having odd or even `parity`. For each q, several trainings are performed on multiple datasets, and statistics are computed over the full ensemble of resulting models. The figure shows the average accuracies of all models trained on datasets with the same imbalance. Here, we adopt the physics nomenclature of MPS instead of TT. The fact that models trained on datasets biased towards different values of the irrelevant feature achieve similar performance indicates that the feature is indeed irrelevant.

each majority class), randomly sampled from the original dataset.

Models

The NN model used consists of a five-layer architecture with structure 16-16-8-4-2, totaling 614 trainable parameters, with ReLU activations in each intermediate layer. Training optimizes the cross-entropy between predictions and labels in batches of 8 datapoints, using the Adam optimizer with a learning rate of $3 \cdot 10^{-4}$ and ℓ_2 regularization of magnitude $6 \cdot 10^{-3}$, for at most 1 250 epochs. The final model selected is the one that achieves the highest accuracy on a held-out validation set of 5 000 samples from the original database.

The TT model consists of six tensors, corresponding to the six input features, with all dimensions—input and ranks—of cardinality 2. All entries in the tensors (a total of 40) are free, trainable parameters. Training optimizes the cross-entropy in batches of 100 datapoints, using the Adam optimizer with a learning rate of 10^{-1}, for 20 epochs.

The encoding of categorical variables differs for both architectures. For NNs, we perform standard one-hot encoding. For TTs, we use a noisy encoding: each datapoint is encoded as a random value in either $\left[0, \frac{1}{2} - \varepsilon\right]$ or $\left[\frac{1}{2} + \varepsilon, 1\right]$, with $\varepsilon = 5 \cdot 10^{-2}$. Each input variable is then mapped into a two-dimensional vector via the embedding $\phi(x) = [1 - x, x]$, which serves as input to the TT. For the `age` column, a min-max normalization is applied prior to encoding.

For each of the 200 datasets generated for every proportion q, as described in the previous section, we train 100 NN and TT models, whose accuracies are shown in Fig. 5.2. This produces, for each of the percentages (the horizontal axis of Fig. 5.2), 20 000 trained models.

The computation of representatives of the parameterization orbits of a given TT model—or, put simply, its reparameterization—is performed in three ways: using the SVD-based canonical

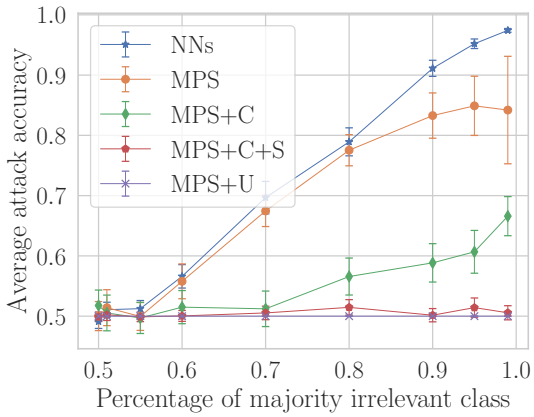


Figure 5.3: Accuracies of attacks attempting to predict the majority value of the irrelevant feature in the training dataset of NN and TT models. Each attack is repeated 1 000 times on randomly selected sets of models, trained on datasets representing 80% of the total. The average test accuracies are shown in the figure. The lines represent, in order: NNs, MPS, MPS after SVD-based canonical form (MPS+C), MPS after SVD-based canonical form and random sampling of residual gauge freedom (MPS+C+S), and MPS after univocal canonical form (MPS+U), where we adopt the physics nomenclature of MPS instead of TT.

form (the green curve, denoted by MPS+C in Fig. 5.3), additionally sampling uniformly over the residual gauge freedom after canonicalization (the red curve, denoted by MPS+C+S), and using the univocal canonical form developed in Section 5.4 (the purple curve, denoted by MPS+U). Here, we adopt the physics nomenclature of MPS instead of TT.

Attacks

The attacks we consider are in the spirit of shadow modeling attacks [116, 117], whereby the attacker is provided with a large collection of models trained on datasets that share the statistics of the dataset where the victim model has been trained. This represents a scenario where the attacker has much more power than what is realistic, so the results serve as upper bounds to realistic attacks.

In order to produce Fig. 5.3, we take, from the 20 000 models trained for each q and model type, as indicated in the previous section, those corresponding to 80% of the datasets, along with their corresponding majority class. These models are then used to train the attacker, while the models for the remaining 20% of datasets are those to be attacked.

The attacker, depending on the type of model, employs a different kind of attack. For NNs, the attack consists of a logistic regressor applied to the full set of models' weights and biases after proper normalization, with ℓ_2 regularization and using the LBFGS solver. This attack is arguably simple, but this only reinforces the argument that NNs are highly vulnerable to leaking the nature of irrelevant features. Removing parameters and keeping only those in the initial layer, or applying Principal Component Analysis (PCA) for dimensionality reduction, did not offer any advantage over using the full set of model parameters as input.

For TT architectures, the attacker trains a deep feedforward neural network that takes as input the model weights (after proper normalization) and outputs the corresponding label. This

approach was chosen to provide the attack with greater generality, potentially allowing it to extract more information from the model parameters. The attacker's NN consists of six layers with structure 20-20-10-10-2-2, using ReLU activations in the intermediate layers. Training optimizes the cross-entropy between predictions and labels in batches of 1 000 datapoints, using the Adam optimizer with a learning rate of 10^{-3} and ℓ_2 regularization of magnitude 10^{-4}, for at most 1 000 epochs. The set of models corresponding to the 80% of the datasets is further split randomly into an 80%-20% proportion to generate a validation set for early stopping the training of the attack.

In order to provide statistics, we train and evaluate the attacks for several random choices of the 80% of datasets used to produce the models given to the attacker during training. We perform a total of 1 000 such repetitions, which provide the confidence intervals shown in Fig. 5.3.

As can be seen in Fig. 5.3, the vulnerability to attacks for NN and TT models is markedly different: both NN and TT architectures straight out from training are equally vulnerable, but TTs in canonical form are not. For instance, at 80% imbalance of the irrelevant feature, attacks on both models achieve around 78% accuracy, while in the canonical-form description of the TT via SVD the accuracy drops to around 56%, close to the limit of random guessing, but still demonstrating that some information remains in the residual gauge left by the SVD. The limit of random guessing is achieved by randomly sampling the residual gauge freedom (this is the curve denoted by MPS+C+S), via adding decompositions of the identity in the form of diagonal matrices with random entries in $\{-1, 1\}$, or by using the univocal canonical form described in Section 5.4 (corresponding to the curve denoted by MPS+U). Notably, the protection occurs despite the attacks performed on TT architectures being more general than those on NNs: while linear regression models are able to extract information on the irrelevant feature in the case of NNs, for TTs the extraction accuracy is not matched even when using more powerful DNNs as attack models.

One could be tempted to attribute the difference in robustness to the attacks to the difference in the number of parameters of the models. As explained earlier, the NN models contain 614 trainable parameters, while the TT models contain only 40. This could, in principle, lead to NN models having a larger capacity that makes them more vulnerable. However, the line corresponding to TT models without post-processing (orange circles) in Fig. 5.3 demonstrates that this is not the case: if the parameters of the TT models are not brought to a canonical form, the models are vulnerable to the attacks in a way analogous to the NN models. Moreover, as indicated in Fig. 5.2, both NN and TT models perform very similarly in the target task, suggesting that the lower accuracy of the privacy attacks for TTs is not due to a difference in expressive capacity.

Therefore, this experiment shows that models trained via gradient descent, including TTs, are vulnerable to the proposed attack. However, reparameterizing the TTs, especially via the univocal canonical form, erases all remaining information and drops the accuracy of the attacks to random guessing. Thus, as expected, these results coincide with the affirmations of Lemma 5.4.1 and Theorem 5.5.1.

5.5.2 Audio data

The experiments conducted in the previous section were set in a relatively simplistic scenario. The dataset consisted of only five features, plus an additional irrelevant variable introduced to perform the attacks. It could be argued that, in more realistic scenarios, such an irrelevant fea-

ture would likely be removed during an initial pre-processing stage, typically involving techniques like PCA to reduce dimensionality and select relevant features.

To address this limitation, we present a much more realistic scenario in this section, where the objective is to classify voices by gender. In this case, each voice sample is represented as a vector in $[0,1]^{500}$, and the irrelevant feature targeted for the attack is one that would not typically be removed during standard pre-processing and does not correspond to a single input variable: the accent. To demonstrate the privacy leakage, we train NN models via gradient descent using training datasets where the accents are predominantly of one type or another. As a solution, we show that instead of training TT models from scratch, TT-RSS can be used to tensorize the original models, reconstructing models that are private while maintaining nearly the same accuracies. In cases where the accuracies drop, it is possible to re-train the TT models for a few additional epochs. In either case, this approach is more efficient and straightforward than training new TT models from the ground. The code employed to train and tensorize the models, and to generate the figures in this section, is provided in [175].

Datasets

For this experiment, we use the public dataset CommonVoice [93], which provides a collection of speech audios in mp3 format. Each audio sample is associated with demographic information, including gender, age, and accent. Additionally, each audio is tagged with a unique identifier, `client_id`, corresponding to each speaker. To simplify the task, we train classifiers with the goal of differentiating voices by gender. To reduce the problem to binary classification, we keep only voices labeled as either woman or man. The variable targeted by the attacker is the accent. To simplify this variable to a binary classification, we consider only "England English" and "Canadian English" accents, referred to here simply as "English" and "Canadian".

To enable statistical analysis, we partition the dataset into several subsets, training multiple NNs on the different subsets. The partitions are also stratified based on the proportion of the `accent` variable. First, we extract a subset from the predefined `train` partition of CommonVoice and balance the `gender` variable across the accents. Denoting q as the percentage of English speakers, we form, for each $q \in \{1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 95, 99\}$, 10 datasets. For each value of q, we create an imbalanced dataset according to the specified accent proportion, and sample 80% of the `client_id` identifiers to form the 10 datasets, splitting each of them into training and validation subsets. Consequently, the 10 datasets created for each q value are distinct but not disjoint. From the remaining 20% of IDs in each case, we select around 200 samples to serve as the pool from which the N pivots for tensorizing the NN models are chosen. Thus, the pivots used in TT-RSS exhibit the same imbalances as those present in the corresponding training datasets. These subsets of 200 samples will be used also to re-train the TT models.

For pre-processing the audio samples, we first resample all audios to a frequency of 1 000 Hz and crop each to retain only the middle 1 000 values, corresponding to one second of audio per person. Next, we apply the Fast Fourier Transform (FFT) and retain only half of the values, since the FFT of real-valued functions is Hermitian-symmetric. This results in feature vectors with 500 variables. Finally, we renormalize the values to restrict them to the interval $[0,1]$.

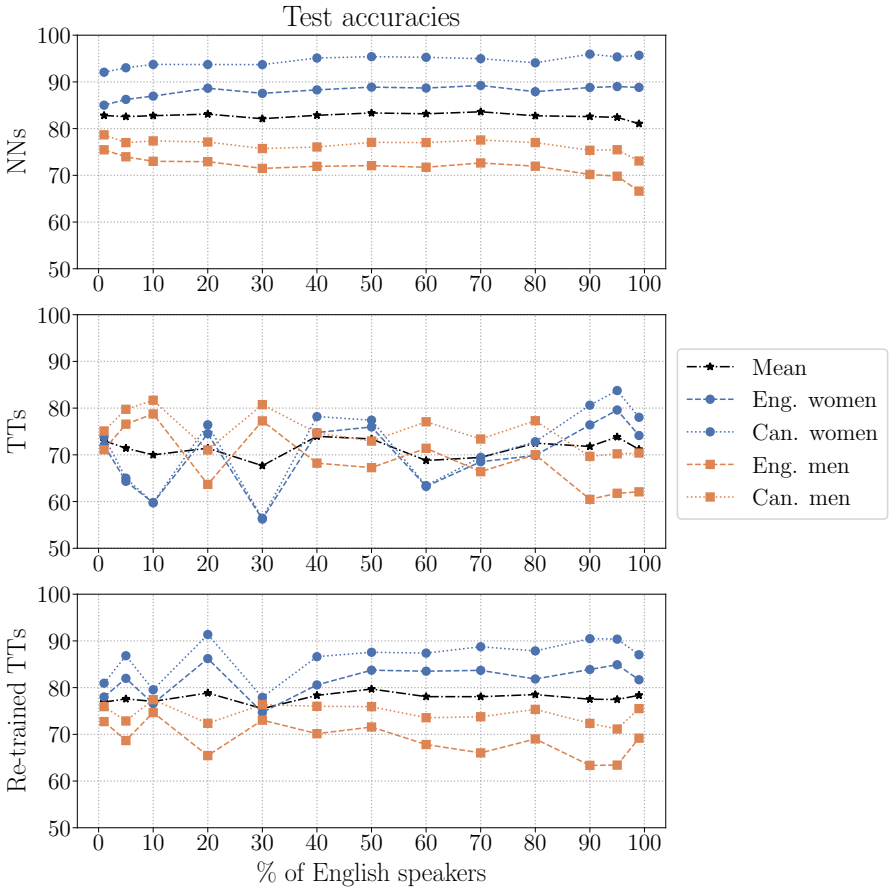


Figure 5.4: Accuracy of the different models evaluated on a test dataset consisting of 300 points, equally distributed across the 4 subgroups: English woman, Canadian woman, English man, and Canadian man. From top to bottom, the models are: the original NN models, TT models output by TT-RSS, and TT models re-trained for 10 epochs. Accuracies are measured for all percentages q of English speakers present in the datasets.

Models

For the classifiers, we use shallow fully connected neural networks consisting of a single hidden layer with 50 neurons and an output layer with 2 neurons (corresponding to the 2 classes), followed by a softmax layer. Between these layers, we include dropout and ReLU activation functions.

The models are trained using the Adam optimizer to minimize the negative log-likelihood. To determine appropriate hyperparameters, we perform an optimization procedure to tune the hyperparameters by training 100 models for 5 epochs each, using balanced datasets ($q = 50$). The optimal hyperparameters obtained are: dropout rate = 0.1, batch size = 8, learning rate = 3.6×10^{-3}, and weight decay = 2.6×10^{-5}.

Using these hyperparameters, we train 25 models for 20 epochs on each of the 10 datasets corresponding to different imbalance levels q. The resulting models achieve an average accuracy of approximately 82%.

Tensorization

To ensure a realistic tensorization scenario, we consider having only black-box access to the NN model, meaning that we can only observe its classifications. While this might seem restrictive, it establishes a minimal-information scenario. For model providers with full access to their models, tensorization could use the output probabilities for each class instead of just the predicted class. However, in other cases, such as when the individual interested in tensorizing the model is merely a user, this black-box approach applies. Additionally, for tensorization, we select as pivots points that are not part of the training or validation sets. Thus, we assume knowledge of the training dataset's distribution but no direct access to the dataset itself.

Remark 5.5.1. Tensorizing models with only black-box access can be considered a form of model-stealing attack, as it provides a method to reconstruct models using only user-level access [186].

The function to be tensorized is thus defined as:

$$\hat{q}_{\text{NN}}(x,y) := \begin{cases} 1, & \text{if } \hat{p}_{\text{NN}}(x,y) \geqslant \frac{1}{2}, \\ 0, & \text{otherwise.} \end{cases} \tag{5.5.1}$$

For TT-RSS, we choose hyperparameters that balance accuracy and computational cost, as suggested by the results in Section 4.4.3. Specifically, we set $d = 2$, $r = 5$ with $c = 0.99$, and $N = 100$, always using the polynomial embedding.

Tensorized models initially exhibit reduced accuracies, typically around 70%-75%. To address this, we re-train the models for 10 epochs using a small dataset containing the pivots but disjoint from the training and validation sets. This process improves accuracy to approximately 78%-80%. While this represents a slight drop compared to the original NN models, it is an acceptable tradeoff for enhanced privacy. Figure 5.4 compares the accuracies of the original NN models to the tensorized and re-trained models.

Previously, we discussed potential issues arising from overfitting the pivots in TT-RSS. As highlighted in Section 4.4.2, overfitting the NN model's outputs does not necessarily imply overfitting of the TT model to the pivots, evidenced by higher accuracies on these points compared to other unseen samples. In our experiments, we verify this by comparing accuracies of TT models on pivots and test samples. Although both plain TT models from TT-RSS and re-trained models show minimal accuracy differences, there is a slight bias towards higher accuracy for pivots in the plain TT models. However, this behavior becomes unclear after re-training (see Fig. 5.5).

Upon re-training, we proceed to obfuscate the TT models' parameters to eliminate any lingering patterns. The approach proposed in Section 5.5.1 involves using the univocal canonical form of the TT. This method provides new TT cores based on a small number of evaluations, ensuring that the TT representation contains the same information as the black-box model. However, as demonstrated in Fig. 5.6, if exploitable patterns exist in the black-box model, they will persist in the TT cores. To mitigate this, we propose a different approach.

To obfuscate the TT cores, we introduce random orthogonal (in general, unitary) matrices W_k and their transposes (equivalently, adjoints), $W_k^\top$, between contractions. This results in new cores defined as

$$\widetilde{G}_k(\alpha_{k-1}, x_k, \alpha_k) := W_{k-1}^\top(\alpha_{k-1}, \theta_{k-1}) G_k(\theta_{k-1}, x_k, \theta_k) W_k(\theta_k, \alpha_k). \tag{5.5.2}$$

Since the added matrices are orthogonal, they cancel out with their transposes in adjacent contractions, preserving the TT's overall behavior. Choosing W_k as random orthogonal matrices

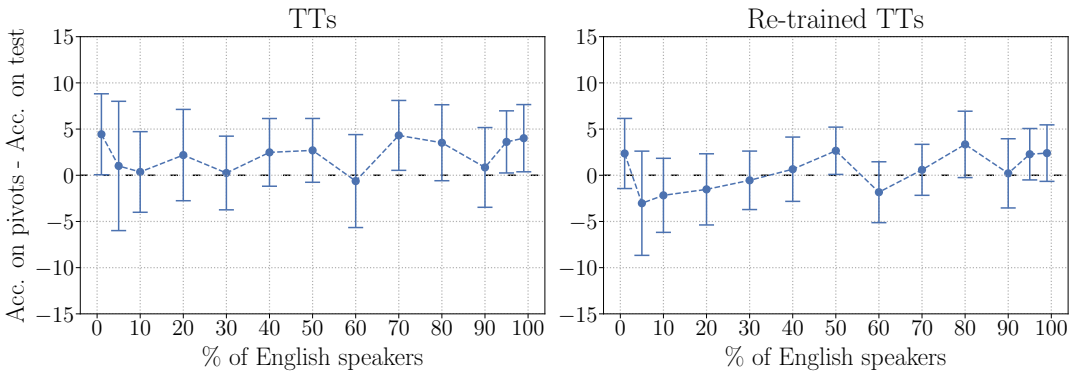


Figure 5.5: Difference between the accuracy measured only on the pivots used in TT-RSS and the accuracy on test samples. On the left, accuracies are measured for TT models directly output from the TT-RSS decomposition. On the right, accuracies are for TT models after re-training them for 10 epochs on a small dataset containing the pivots. These differences are measured for all percentages q of the presence of English speakers in the datasets. For each q, mean differences in accuracy across all models trained for that q are shown, with error bars at $\pm 0.5\sigma$, where σ denotes the standard deviation.

introduces randomness into the cores $\widetilde{G}_k$, effectively removing hidden patterns while maintaining stability by preserving core norms. This new representation is referred to as Private-TT.

Attacks

The property we aim to infer from the training dataset is the majority accent present in it. Specifically, we aim to create a function that, given some information about a model, predicts the majority accent. To achieve this, we train a logistic regression model.

As detailed earlier, we trained 250 NNs for each percentage q. To train the attack model, we merge all models corresponding to q and $100 - q$, for $q \in \{1, 5, 10, 20, 30, 40\}$, and create a dataset by storing some information from each model, x, along with its corresponding label, y. We define the label for each model as $y = 0$ if $q < 50$, and $y = 1$ otherwise. For the information representing the model, x, we consider two scenarios: (i) black-box access, where we can only observe classifications made by the model, and (ii) white-box access, where we have full access to the model's parameters.

For the black-box scenario, we evaluate each model on 100 samples, equally divided among four groups: woman or man, English or Canadian speakers. The resulting 100-dimensional vector represents the information we observe from each model. To ensure fairness, the same 100 samples are used to evaluate all models. These samples are selected from a predefined partition of the CommonVoice dataset, denoted as `validated`, ensuring no overlap with the `train` partition to avoid including training points from any model.

For the white-box scenario, we assume complete access to the model parameters. Although an attacker with white-box access can also evaluate the model (thus accessing black-box information), we conduct these experiments separately to assess the performance of attacks based solely on one type of information. Stronger attacks combining both information sources are possible but beyond the scope of this analysis.

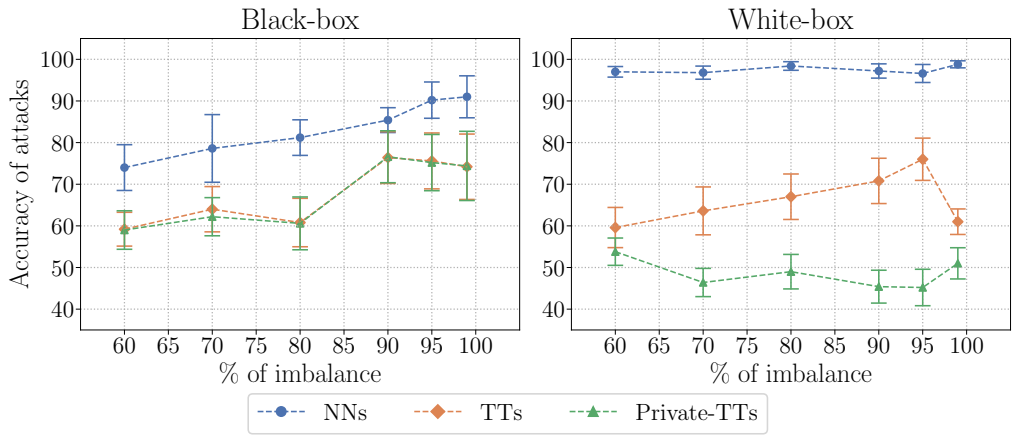


Figure 5.6: Accuracy of logistic regression models trained to classify whether a model was trained on a dataset with a majority of English speakers or a majority of Canadian speakers. For each percentage of imbalance, a dataset is formed from 500 models trained on 20 different datasets (10 datasets per percentage q, with 25 models per dataset). For each configuration, attacks are evaluated via K-fold validation with $K = 10$, and mean accuracies are displayed with error bars at $\pm 0.5\sigma$, where σ denotes the standard deviation.

To compare attack accuracies, we create such datasets for three model types: NNs, TTs, and Private-TTs. Unlike what was observed in Section 5.5.1, our results reveal that accent influences the models' gender classification even at a black-box level (see Fig. 5.6). Models trained with a majority of English speakers can be distinguished from those trained on datasets dominated by Canadian speakers based on their performance differences on individuals with these accents.

Additionally, although TT models are constructed via TT-RSS using only black-box information—implying that NNs and TTs allegedly represent the same black-boxes—black-box attacks on TT models perform worse than on NNs. This discrepancy could be attributed to the rank restriction r of the TT, which limits the model's expressive power and thereby reduces its tendency to memorize training data. Consequently, this constraint slightly lowers the accuracy of the TT models, as shown in Fig. 5.4. Notably, black-box attacks on Private-TT models yield identical results to those on TT models, as the black-box outputs remain the same.

The most revealing results arise from white-box attacks, as they demonstrate the expected behavior for each case. For NN models, a simple logistic regression attack achieves nearly 100% accuracy across all imbalance percentages. For TT models, attack accuracy increases with greater imbalance, mirroring the trend observed in black-box attacks. This similarity arises because TT-RSS reconstructs the TT using only black-box evaluations of the model. Thus, white-box attacks on TT models produce results comparable to black-box attacks. As discussed throughout this chapter, this could already be considered private to some extent. However, it remains unclear whether the information in the black-box and white-box representations is equivalent or complementary. An attacker might achieve better results by combining both. To mitigate such risks, all information in the white-box can be obfuscated by applying the proposed obfuscation method for TTs, which is computationally inexpensive. This is reflected in the attacks on Private-TTs, where accuracy drops to 50%, equivalent to random guessing.

It is important to note, however, that these attacks rely on simple logistic regression models

and should be considered lower bounds for what an attacker could achieve in this setting. On the other hand, the attacks assume access to extensive information, such as hundreds of trained models, enabling statistical analysis and training of powerful classifiers. From this perspective, the results in Fig. 5.6 may represent a pessimistic lower bound on the potential privacy risks.

5.6 Conclusions and future directions

In this chapter, we have shown that global information about the training dataset is embedded in the parameters of deep learning models, even when this information is irrelevant to the task at hand. This idea was first motivated by the toy example presented in Section 5.2, where it can be shown analytically that patterns are imprinted in the parameters depending on the initialization point and the dataset used. Although we have also presented a series of experiments to support this observation, it is further corroborated by recent, stronger theoretical results, which demonstrate that gradient-descent training of NNs leads to this vulnerability [119, 120]. Specifically, these works show that NN parameters satisfy the KKT conditions of a minimization problem that depends on the training data, and that these conditions can be exploited to reconstruct training points from the model parameters by solving a system of equations. This suggests that our proposed vulnerability may apply in even more general settings.

What we remark in our work is that even information irrelevant to the task, and unobservable from the black-box perspective, leaves an imprint in the parameters. Thus, it would be desirable to reparameterize the models in such a way that this information is also eliminated from the white-box, implying that the final parameters contain no more information about the training dataset than what is strictly necessary to produce the output. This is precisely the statement of Lemma 5.4.1. For NNs, one could envision obtaining a canonical form by proceeding in the spirit of model extraction attacks [186, 187]. While it would be necessary to carefully assess whether the robustness requirements are met in this scenario, one must also consider the increased computational cost and data requirements involved (since a second model must be trained from input-output queries to the vulnerable one, ideally using data points not involved in the original training), as well as the potential loss of accuracy introduced by the process. In contrast, TT models are inherently reparameterizable due to their explicit characterization of gauge freedom, which enables the definition of canonical forms that eliminate all unnecessary information, as supported by Theorem 5.5.1. Additionally, as shown in the results of Section 5.5.2, the TT-RSS can be used to reparameterize the black-box of a NN, as it provides a mechanism to construct a new function, in TT form, solely from black-box information of the NN.

Our results nevertheless apply to any architecture in which different parameterizations correspond to the same model, and where a global, smooth, and one-to-one mapping can be defined between the space of black-box representations and a representative of each possible model. If such a mapping exists, it can be used to eliminate information that the final model should not retain. This part of the proof is not restricted to TT architectures, general TN architectures, or even NNs. However, given the extensive expertise of the quantum many-body physics community, it is expected that model classes satisfying the required conditions are more easily found within the TN family [188]. Identifying such architecture classes, which are both sufficiently expressive and easy to train, remains an important open task. A promising family is that of canonical polyadic decompositions of tensors, which correspond to common architectures [189, 190], are invariant under suitable rescalings, and for which, under mild conditions, a unique factorization exists [191].

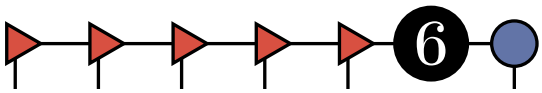
As discussed, the attacks presented in this chapter are in the spirit of property inference attacks, as the majority value of a feature in the dataset constitutes a property of the entire training dataset. Therefore, it remains to be determined whether certain DP mechanisms would also provide protection against this type of vulnerability, given that the primary goal of DP is to safeguard individual privacy. However, even if DP mechanisms prove effective, our solution offers the advantage of preserving model performance, whereas DP mechanisms operate by injecting noise into the data [179], the training process [48, 180], or the model parameters [47], often resulting in greater performance degradation. In any case, TN architectures can also be trained with DP mechanisms, thereby providing individual datapoint privacy in addition to the white-box protection achieved through canonical forms.

An interesting side effect of the tensorization via TT-RSS in the voice classification experiment of Section 5.5.2 is the reduction in the accuracy of black-box attacks on the TT models compared to the NN models. This raises an important question: while the reparameterization of TTs removes information from the white-box that was not present in the black-box, can we further reduce the information accessible through the black-box by compressing the models? Interestingly, some recent works have proposed pruning techniques in NN models as an alternative means to achieve DP without significantly impacting the training process [125, 126]. For comparison, the reduction of ranks in a TT model—which effectively occurs when imposing a low-rank TT form on a NN—can be viewed as a form of pruning. Further investigating this relationship opens an interesting direction to endow TT models with DP.

Additionally, as mentioned above, TT-RSS can be interpreted as the map $\alpha : \mathcal{B} \to \mathcal{W}$ that produces a new parameterization, as a single TT, from the black-box of a NN. However, this approach may be limiting in realistic scenarios, where compressing highly expressive DNNs into a single TT can significantly affect performance. In contrast, as will be detailed in Section 6.3, a common tensorization of NNs that avoids such degradation involves decomposing each linear layer of the model into a TTM form. Nevertheless, this tensorization does not serve as an obfuscation method, since it only allows for reparameterization at the layer level, rather than for the full model. As a consequence, it may occur that certain features present in the dataset are canceled by the combined effect of consecutive layers, rendering them irrelevant for the black-box output, while the layer-wise tensorization lacks the ability to merge layers and effectively remove the patterns induced by these features. To address this issue, as well as to analyze the obfuscation capabilities of SVD-based canonical forms of TTs, an interesting direction to explore would be to provide probabilistic results in the spirit of DP. Specifically, this would involve quantifying the probability that an attacker can extract information from the parameters, assuming a certain capacity to randomize them—e.g., by randomizing the gauge in a TT or in each TT layer of a layer-wise tensorized NN.

Beyond machine learning, the equivalence between black-box and white-box access to functions represented by TTs, as established in this chapter, has the potential for significant implications in cryptography. In particular, the equivalence between black- and white-box access is a fundamental concept in obfuscation [192]. Typically discussed in the context of programs or circuits, obfuscation refers to providing an alternative object that performs the same function but whose description is "unintelligible", in the sense that any information extractable from the description could equally be obtained from the input-output behavior. The line of reasoning presented in Lemma 5.4.1 and Theorem 5.5.1 follows an approach closely related to a proof of obfuscation, motivating deeper exploration of the connections between TNs and Boolean circuits, and suggesting a pathway for establishing obfuscation proofs in restricted circuit classes.

Further applications



In this chapter, we show that tensorizing NN models into TTs offers notable advantages beyond privacy. For instance, TTs provide a high degree of interpretability, as they allow direct access to efficiently computable quantities that yield insightful information about the distribution or system they represent. To illustrate this capability in a physics-based context, we demonstrate how TT representations, derived via TT-RSS, enable the estimation of topological phases. In particular, in Section 6.1, we use TT-RSS to reconstruct the TT representation of the AKLT state from black-box access to the amplitudes of a limited number of spin configurations. From this representation, we calculate an order parameter that allows us to estimate the topological phase, following an approach similar to that presented in Ref. [193]. While this serves as a proof of principle, the methodology could be extended to estimate topological phases in higher-dimensional settings.

Additionally, as a side effect of tensorizing ML models via TT-RSS, we observe two phenomena that could be independently exploited and deserve further analysis. First, following the methodology outlined in Section 5.5.2, if the TT obtained through TT-RSS exhibits lower accuracies compared to the original NN models, these TTs could still serve as starting points for further optimization. This suggests that TT-RSS may serve as an initialization mechanism for TT models. We explore this potential by comparing different initializations (corresponding to different embeddings) in Section 6.2.

Secondly, TT-RSS allows us to leverage one of the most significant advantages of TNs: their efficiency. By tensorizing NN models, we obtain TT models characterized by fewer parameters, leading to reduced computational costs. In Section 6.3, we compare the compression capabilities of this approach with other common techniques that tensorize NNs in a layer-wise fashion to reduce parameters, showing that constructing a single TT model with TT-RSS may offer superior memory-time efficiency trade-offs in specific scenarios.

All experiments presented in this chapter were conducted on an Intel Xeon CPU E5-2620 v4 with 256GB of RAM and an NVIDIA GeForce RTX 3090, using the implementation of TT-RSS

available in the TensorKrowch Python package [174]. The code for the experiments is publicly available in [175].

6.1 Interpretability from tensor trains

Tensor networks have been acclaimed in the field of condensed matter physics due to their interpretability, which allows to easily study properties of quantum systems. In particular, TN representations of quantum states have been pivotal in studying topological phases of matter by capturing how global symmetries affect the local tensors constituting the network.

Specifically, as commented in Section 2.4.3, it can be seen that two states are equal if there is a gauge transformation and a phase that transforms one representation to the other [60, 194]. This implies that, for translationally invariant TTs, under the action of global on-site, reflection, or time-reversal symmetries, the local tensors building up the TT must transform trivially up to a phase. This leads to writing TT tensors in terms of irreducible representations of a symmetry group G [195]. These irreducible representations on the virtual degrees of freedom form non-linear projective representations [196–199].

Further work showed that injective, TI TTs [60] belong to the same phase if they can be interpolated without topological obstructions. However, when symmetry is imposed, the interpolation path is constrained and topological obstructions might occur, leading to phase transitions at the points where the interpolating TT becomes non-injective [197, 198]. These phases, known as Symmetry-Protected Topological (SPT) phases, are classified by the second cohomology group for gapped quantum spin chains [200, 201].

Unlike symmetry-breaking phases, which can be distinguished by local order parameters, SPT phases require non-local observables, like string order parameters, for their detection [202]. A key example is the AKLT state, which exhibits non-trivial projective symmetry representations protected by multiple symmetries, including on-site $\mathbb{Z}_2 \times \mathbb{Z}_2$ symmetry [203]. Reference [193] introduced an order parameter to estimate the SPT phase of the AKLT state, which can be computed from the TI TT representation of the state. This is the approach followed in this section to estimate the AKLT SPT phase from the TT representation obtained via TT-RSS. Tensor networks, as such, remain invaluable due to their capacity to encode both the entanglement structure and symmetry properties of quantum states. For a more comprehensive exploration of this field, including its extensions to higher dimensions, we refer readers to the review by Cirac *et al.* [51] and the references therein.

In recent years, several approaches have emerged to approximate quantum states via NN representations, enabling the efficient representation of complex systems [27–31]. However, these representations often lack the capacity to analyze properties of the states directly, requiring additional models to estimate each property separately [32–34]. In contrast, the methodology we propose leverages pre-trained Neural Network Quantum States (NNQS) to derive a TN representation, enabling the direct study of such properties.

Beyond phase classification, the interpretability capacities of TNs extend to other areas. Loop-less networks like TTs and Trees offer direct and efficient access to critical information such as marginal or conditional distributions, mutual information, and entropy, which has found applications in explainable machine learning [69, 70].

6.1.1 AKLT Model

We present this methodology in a simplified scenario that may serve as a proof of principle. In particular, we will estimate the SPT phase of the AKLT state [203]. Although this state admits a TT form, this is not necessary for the method to succeed, since we will only allow black-box access to this function. Indeed, we will exploit the TT structure of the state only when obtaining the pivots and only for computational efficiency. Thus, this setup is analogous to having access to other less interpretable models, such as a NN, which we can query with spin configurations to obtain the associated amplitudes. Using this function, we will reconstruct a new TT representation of the state via TT-RSS and use this representation to compute the order parameter defined in Ref. [193].

The AKLT model, named after its authors Affleck–Kennedy–Lieb–Tasaki, is defined by the following Hamiltonian:

$$H = \sum_{k=1}^{n} P^{(2)}_{k,k+1}, \tag{6.1.1}$$

where $P^{(2)}_{k,k+1}$ is the projector operator that projects the total spin of the neighboring sites k and $k+1$, both of them spin-1 particles, onto the spin-2 subspace. Consequently, its ground state, referred to as the AKLT state, is formed by spin-1/2 singlets between nearest neighbor sites, with projector operators at each site projecting the two spin-1/2's into the spin-1 subspace.

This state can be represented in TT/MPS form. For the representation of the model and the subsequent description of the order parameter computation, we use Dirac's bra-ket notation. Additionally, instead of representing the TT cores as tensors $G_k(\alpha_{k-1}, i_k, \alpha_k)$, we will simply denote them as matrices indexed by the input label, $G_k^{i_k}$. Thus, a TI TT representation of the AKLT state is defined as:

$$|\Psi\rangle = \sum_{i_1,\dots,i_n} \mathrm{Tr}[G^{i_1}\cdots G^{i_n}]\,|i_1\cdots i_n\rangle, \tag{6.1.2}$$

where subscripts are omitted as all cores are identical. The cores are given by

$$G^- = -\sqrt{\frac{2}{3}}\sigma_-, \quad G^0 = -\sqrt{\frac{1}{3}}\sigma_0, \quad G^+ = \sqrt{\frac{2}{3}}\sigma_+. \tag{6.1.3}$$

After a suitable change of basis, these tensors become

$$G^{i_k} = \sqrt{\frac{1}{3}}\sigma_{i_k}, \quad i_k \in \{1,2,3\}, \tag{6.1.4}$$

as detailed in Example 1.e of the Appendix of Ref. [51]. For clarity in the presentation of Section 6.1.3, we assume the TT cores are expressed in this basis.

In the above definitions, σ denotes the Pauli matrices, defined as:

$$\sigma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{6.1.5}$$

The matrices σ_-, σ_0, and σ_+ are then defined as:

$$\sigma_- = \sigma_1 - i\sigma_2, \quad \sigma_0 = \sigma_3, \quad \sigma_+ = \sigma_1 + i\sigma_2. \tag{6.1.6}$$

Remark 6.1.1. Until now, we have focused on applying TT-RSS to real-valued functions. However, it is important to note that the tensorization process is equally applicable to complex-valued functions, as it involves only linear projectors, SVDs, and the solving of matrix equations. Consequently, while in this example we use TT-RSS to recover a TT representation of the AKLT state, which has real coefficients, the methodology can be straightforwardly extended to general quantum states with complex coefficients.

6.1.2 Tensorization

The definition given in Eq. (6.1.2) forms a TT with PBC. However, TT-RSS cannot generate TTs with this type of loop. Therefore, the function we decompose is defined by imposing OBC, as follows:

$$|\Psi\rangle = \sum_{i_1,\dots,i_n} \langle 0| G^{i_1} \cdots G^{i_n} |0\rangle |i_1 \cdots i_n\rangle, \tag{6.1.7}$$

from which we can define $G_1^{i_1} = \langle 0| G^{i_1}$ and $G_n^{i_n} = G^{i_n} |0\rangle$.

To apply TT-RSS, we fix the input dimension and rank as defined by the AKLT state, i.e., $d = 3$ and $r = 2$ (with $c = 1 - 10^{-5}$). Since the function has a discrete domain, we use the trivial embedding for all sites. For pivot selection, we assume no knowledge of the state, and thus, we sample random configurations $(i_1, \dots, i_n)$ from the distribution defined by the AKLT state. While we assume no access to the TT representation of the state during the entire process, we leverage it for computational efficiency when obtaining the pivots, as it enables direct sampling. Alternatively, this step could be performed via Monte Carlo sampling, as required for NNQS. To analyze the performance of TT-RSS with different numbers of pivots, we use values of N ranging from 2 to 14. Additionally, we run the experiments for three different values of n: 100, 200, and 500. For each configuration, TT-RSS is applied 10 times.

Upon tensorization, we obtain approximate versions of $|\Psi\rangle$, given by

$$|\widehat{\Psi}\rangle = \sum_{i_1,\dots,i_n} \widehat{G}_1^{i_1} \cdots \widehat{G}_n^{i_n} |i_1 \cdots i_n\rangle. \tag{6.1.8}$$

Note that in the reconstructed TT the cores may be different, as no restrictions are imposed in Eq. (4.2.4) to obtain equal cores. To attempt a TI representation, we can bring the TT into a canonical form [60, 140]. In this case, we only fix the gauge to the left of each tensor by requiring

$$\sum_{i_k} \widehat{G}_k^{i_k \dagger} \widehat{G}_k^{i_k} = \mathbb{I}. \tag{6.1.9}$$

This condition can be achieved iteratively through SVD, as described in Ref. [60], resulting in what we refer to as the SVD-based canonical form. In this form, the representation of the AKLT state obtained is equivalent to the one in Eq. (6.1.4) up to conjugation by unitaries, i.e.,

$$\widehat{G}_k^{i_k} = W_{k-1}^\dagger G^{i_k} W_k, \tag{6.1.10}$$

where $W_k \in \mathrm{U}(2)$ for all $k \in [n-1]$.

To evaluate the accuracy of the reconstruction, we measure the fidelity between the recovered state $|\widehat{\Psi}\rangle$ and the original state $|\Psi\rangle$, as defined in Eq. (4.4.3). Remarkably, we observe that using as few as 12 pivots is sufficient to achieve a fidelity of 1, meaning that the AKLT state is recovered exactly, regardless of the system size n (see Fig. 6.1).

6.1.3 Order parameter

To estimate the SPT phase of the AKLT state, we compute the order parameter described in Ref. [193], which distinguishes the equivalence classes of the projective representations of the symmetry group $\mathbb{Z}_2 \times \mathbb{Z}_2$. This order parameter can be computed using the TT representation of the state and is insensitive to unitary conjugations of the form of Eq. (6.1.10). As such, it provides a natural method for estimating SPT phases within our approach.

Specifically, if u_g denotes a unitary representation of $\mathbb{Z}_2 \times \mathbb{Z}_2$, due to the symmetry, the system satisfies

$$u_g^{\otimes n} |\Psi\rangle = |\Psi\rangle . \tag{6.1.11}$$

This implies, by virtue of the Fundamental Theorem of MPS [60]—which characterizes the gauge freedom introduced in Eq. (2.4.26)—that there must exist a TI representation of $u_g^{\otimes n} |\Psi\rangle$ given by tensors H, such that

$$H^{i_k} = V_g^\dagger G^{i_k} V_g, \tag{6.1.12}$$

where the physical transformation u_g induces a transformation on the virtual degrees of freedom, represented by V_g. These transformations are not unique; transformations of the form $\widetilde{V}_g = \beta(g) V_g$, with $\beta(g) \in \mathrm{U}(1)$, also satisfy Eq. (6.1.12). Consequently, they yield projective representations of the symmetry group, which can be classified into equivalence classes. These equivalence classes form a group isomorphic to the second cohomology group, $H^2(\mathbb{Z}_2 \times \mathbb{Z}_2, \mathrm{U}(1))$, which labels the inequivalent representations.

For the AKLT state with global on-site $\mathbb{Z}_2 \times \mathbb{Z}_2$ symmetry, we aim to compute an order parameter to distinguish whether the transformations V_g and V_h, corresponding to different group elements g and h, commute or anti-commute. In this example, it is straightforward to verify, using the TI representation of the AKLT state (6.1.4), that the Pauli matrices, which anti-commute, form a projective representation of $\mathbb{Z}_2 \times \mathbb{Z}_2$. The relationship between the elements of the symmetry group and their corresponding representations as Pauli matrices is shown in Table 6.1.

Element of $\mathbb{Z}_2 \times \mathbb{Z}_2$	Unitary rep.	Projective rep.
$g_3 = (1,0)$	$u_3 = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$V_3 = \sigma_3$
$g_2 = (0,1)$	$u_2 = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$	$V_2 = \sigma_2$
$g_1 = (1,1)$	$u_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$	$V_1 = \sigma_1$

Table 6.1: Correspondence between elements g_α of the symmetry group $\mathbb{Z}_2 \times \mathbb{Z}_2$, elements u_α of a unitary representation, and elements V_α of a projective representation.

However, since the TT representation returned by TT-RSS is not TI and retains the gauge freedom described in Eq. (6.1.10), projective representations may not be readily identifiable. To address this, we employ the order parameter proposed in [193], which determines whether the matrices of the projective representation commute or anti-commute without requiring direct access to their explicit forms.

Due to the finite correlation length of TTs, it can be shown that if U_g denotes the action of the symmetry under renormalization (e.g., blocking L sites, where $U_g = u_g^{\otimes L}$), then

$$G^{i_k} \cdots G^{i_{k+L}} U_g(i_k, \ldots, i_{k+L}, j_k, \ldots, j_{k+L}) \bar{G}^{j_k} \cdots \bar{G}^{j_{k+L}} \approx (\Lambda V_g) \otimes (\Lambda \bar{V}_g), \tag{6.1.13}$$

where $\Lambda = \sum_s \lambda_s |s\rangle\langle s|$, with $\lambda_s = \frac{1}{\sqrt{2}}$ for $s \in [2]$ being the Schmidt values. This approximation converges exponentially fast with the blocking size L. For the subscript notation, we use the Pauli matrix index $\alpha \in \{1, 2, 3\}$ to label the different group elements and their corresponding representations, as shown in Table 6.1, that is, $g_\alpha \leftrightarrow u_\alpha \leftrightarrow V_\alpha \leftrightarrow \sigma_\alpha$.

Using this, we can compute the following order parameter:

$$\mathcal{E}_L(\Psi) := \langle \Psi | (U_3 \otimes U_3 \otimes \mathbb{I}) \mathbb{F}_{13} (U_1 \otimes U_1 \otimes \mathbb{I}) | \Psi \rangle, \tag{6.1.14}$$

where $\mathbb{F}_{13}$ is the operator that swaps the first and third sites. This value effectively computes

$$\mathcal{E}_L(\Psi) \approx \mathrm{Tr}[V_3 V_1 V_3^\dagger V_1^\dagger \Lambda^4] \mathrm{Tr}[\Lambda^4], \tag{6.1.15}$$

yielding

$$\mathcal{E}(\Psi) \approx \pm \mathrm{Tr}[\Lambda^4]^2, \tag{6.1.16}$$

where the sign reveals whether the elements V_1 and V_3 commute or anti-commute.

Remark 6.1.2. In the description of the order parameter $\mathcal{E}_L$, we assume that the TT is TI. However, as pointed out earlier, the result from TT-RSS after canonicalization still retains a U(2) gauge freedom. Nevertheless, when computing the order parameter while accounting for the unitary transformations W_k acting on the virtual indices as shown in Eq. (6.1.10), these transformations cancel out, yielding the same result as in the TI case.

In our experiment, we measure $16\mathcal{E}_{10}(\Psi) \approx \pm 1$ to distinguish the trivial from the non-trivial phase. For the AKLT state, which is known to be in the non-trivial phase due to the anti-commutation relations of the Pauli matrices, the order parameter is -1. This is the value we must observe from the reconstructed TT representation of the state as well. As shown in Fig. 6.1, when the fidelity is 1—indicating an exact reconstruction of the AKLT state—the order parameter attains the value of -1. However, since our method does not rely on prior knowledge of the exact representation, it is applicable to cases where the achieved fidelity is less than 1. Notably, our results demonstrate that the order parameter reaches -1 with only 6 pivots in TT-RSS, even when fidelity remains below 1.

This demonstrates that our method can estimate order parameters of systems with hundreds of sites in just a few seconds, in sharp contrast to other machine learning techniques that require training new models to learn specific system properties [32–34]. With our approach, one can leverage black-box access to a pre-trained model, such as a NNQS [27–31], to construct a TN representation via TT-RSS and use it to study system properties with well-established tools in the field.

For systems well-approximated by TTs, as in the example shown, it may be more beneficial to perform variational optimization directly using the TT form, which can be achieved very

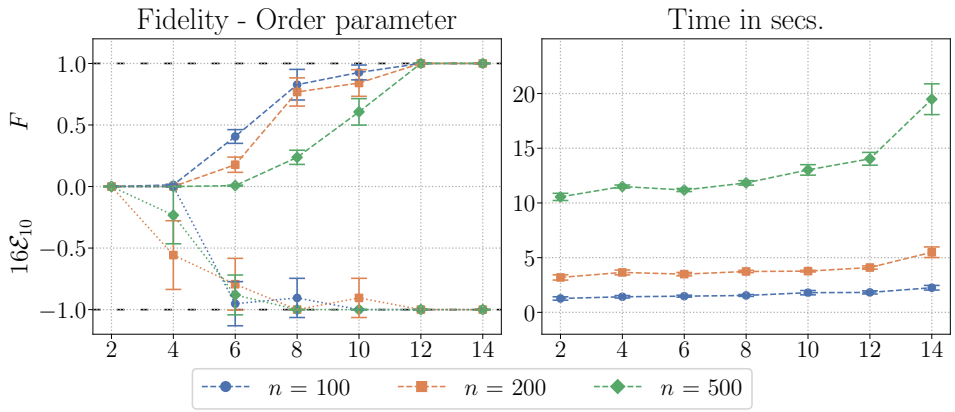


Figure 6.1: Fidelity between the AKLT state $|\Psi\rangle$ and the approximated TT representation obtained via TT-RSS, $|\widehat{\Psi}\rangle$, along with the order parameter $16\mathcal{E}_{10}$ to distinguish the trivial from the non-trivial phase. The right figure shows the time taken to perform TT-RSS in seconds. For each value of N, the decomposition is performed 10 times, displaying the mean values with error bars at $\pm 0.5\sigma$, where σ denotes the standard deviation. Three configurations are considered with different numbers of variables, $n = 100$, $n = 200$, and $n = 500$, represented by different colors.

efficiently [159, 160]. However, for systems in 2D or higher dimensions, this approach becomes highly complex, as it has been shown that contracting PEPS is #P-complete [204, 205]. If TT-RSS could be extended to such higher-dimensional scenarios, this methodology could be used to construct PEPS while avoiding their optimization. Although no current adaptation exists for PEPS, ideas involving sketching and cross interpolation have already been employed to construct Tree TNs [152, 153, 158], which could be implemented for higher-dimensional layouts.

6.2 Initialization mechanism

When training ML models via gradient descent methods, the initialization of variational parameters is crucial for ensuring stable optimization and achieving high-quality solutions. For DNNs with ReLU activations, it has been demonstrated that initializing weights from a Gaussian distribution with appropriate standard deviations results in the outputs of the initial network being distributed according to the Gaussian density $\mathcal{N}(0, 1)$ [206]. Furthermore, over-parameterized DNNs with such random initialization have been shown to converge to near-global minima in polynomial time when optimized using SGD [207].

Inspired by the success of these methods, a similar approach was proposed for TNs in Ref. [66], aiming to ensure that the outputs of randomly initialized TNs follow the $\mathcal{N}(0, 1)$ distribution. However, the sequential contraction of tensors in a TN involves multiple additions and multiplications of the parameters within the tensors, making the outputs highly sensitive to perturbations. This sensitivity often leads to rapid explosion or vanishing of both the outputs and gradients, which can render optimization infeasible. Consequently, this type of initialization is only effective for shallow networks with relatively few input variables. As a partial remedy for the vanishing or exploding values, recent work has introduced a method to maintain the total norm of the TN within a desired range by appropriately normalizing the cores [67]. In

general, if TN models are employed in scenarios where the total norm is not a critical factor, the intermediate computations during contraction can be stabilized by normalizing the results at each step. However, this normalization process is typically performed independently for each input sample, which may undermine the ability to compare the magnitudes of different outputs. In particular, if the TN model represents a probability distribution, it is preferable to use a consistent scaling factor, or partition function, across all samples to preserve the comparability of outputs.

Using TT-RSS, we can generate TT models that produce outputs within a desired range for inputs drawn from a specified distribution and for a chosen set of embedding functions. As an example, we train TT models for the same classification task described in Section 5.5.2 with various initializations and embeddings, and compare their performance against TT models initialized via TT-RSS.

In all cases, we set $d = 2$ and $r = 5$. For the embeddings, we consider two options: (i) the polynomial embedding $\phi(x) = [1, x]$, as proposed in Ref. [59], and (ii) the unit embedding $\phi(x) = [\cos(\frac{\pi}{2}x), \sin(\frac{\pi}{2}x)]$, introduced in Ref. [58] and commonly used for machine learning tasks. For the initializations, we explore the following settings:

- **TT-RSS:** The TT model is obtained by tensorizing a pre-trained NN with the architecture described in Section 5.5.2. This initialization provides an initial configuration of ranks $r_1, \ldots, r_{n-1}$ adjusted to the given distribution. For all other initializations, the rank r is fixed across all internal dimensions.

- **TT-RSS random:** The TT is generated via TT-RSS from a random function p that outputs random probabilities, only ensuring that the conditional probabilities $p(y|x)$ sum to 1. Consistent with prior cases, the function to be tensorized is the square root of p.

- **Id. + noise:** The first matrix in each core is initialized as the identity, i.e., $G_k^1 = \mathbb{I}$. Gaussian noise with a small standard deviation (ranging from 10^{-5} to 10^{-9}) is then added to the entire tensor, ensuring that outputs remain stable. This initialization, combined with the polynomial embedding, was suggested in Ref. [59].

- **Unitaries:** Each core is initialized randomly as a stack of orthogonal matrices or, more generally, unitaries. This approach corresponds to the initialization described in Section 4.4.1.

- **Canonical:** Each core is initialized randomly according to a Gaussian distribution, and the TT is subsequently brought into SVD-based canonical form.

Remark 6.2.1. Although the methods described in this work have primarily been used to tensorize NNs, they are equally applicable to other types of models. For example, instead of pre-training a complex model such as a NN, one could tensorize a simpler, easier-to-train model. While this approach may not yield the highest accuracies, it could suffice for initialization purposes, providing an appropriate output range that is replicated by the TT model. Alternatively, one could use as initialization for the TT model the tensorization of an initialized, un-trained NN.

The training accuracies of the TT models trained with the different configurations are shown in Fig. 6.2. While some combinations, such as using the polynomial embedding with the "Id. + noise" initialization or the unit embedding initialized with "Unitaries", appear to perform well

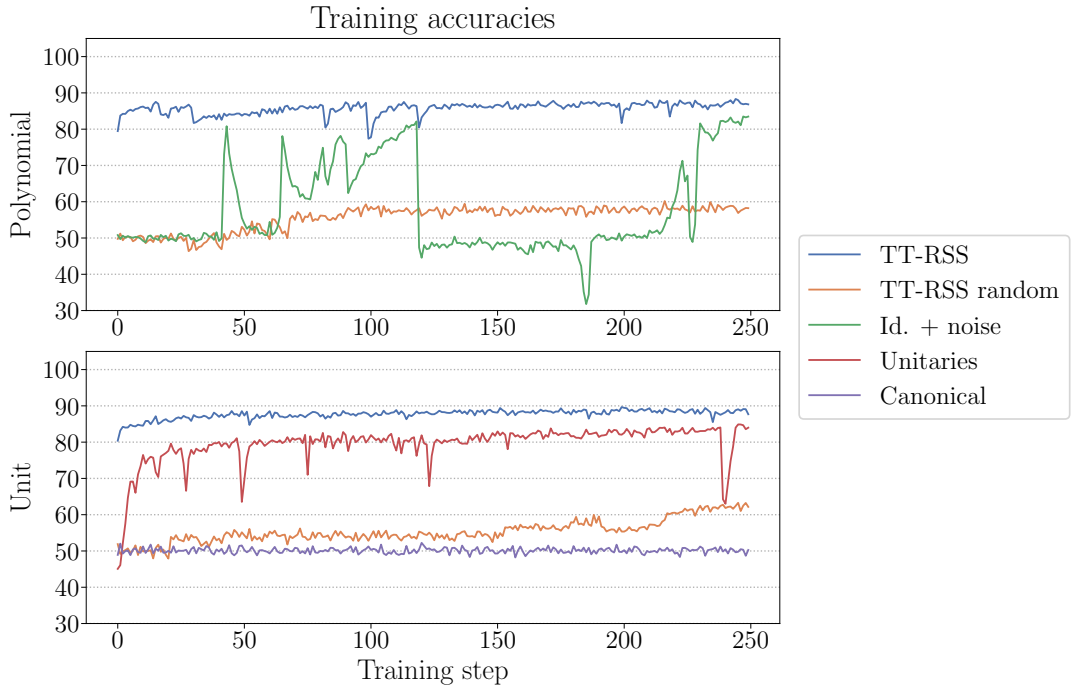


Figure 6.2: Evolution of training accuracies of TT models trained via the Adam optimizer (learning rate = 10^{-5}, weight decay = 10^{-10}) for a binary classification task. The top figure shows results using the polynomial embedding, while the bottom figure corresponds to the unit embedding. Different initialization methods are employed for both embeddings, with each method represented by a distinct color.

during training, TT-RSS consistently yields the highest and most stable accuracies. However, this does not always translate into the best test accuracies. It is possible that other random initialization points allow for models that generalize better, whereas TT-RSS initialization may bias the training process of the obtained TT model.

Conversely, TT-RSS from random functions achieves significantly lower accuracies. Nevertheless, it is worth highlighting that this is an agnostic random initialization scheme—generalizable across different model configurations and independent of pre-trained models. Therefore, we consider the observed gradual improvement in accuracies a promising result. To further explore the potential of this initialization method, a more exhaustive search of training hyperparameters could be performed. Additionally, tensorizing alternative random functions that exhibit greater consistency across different inputs may enhance the method.

In both cases, "TT-RSS" and "TT-RSS random", tensorization enables the initialization of a TT model such that, for inputs drawn from the data distribution, it produces outputs within a controlled range that can be predetermined. This approach is also compatible with a large number of variables n, diverse embedding functions, and varying input/output or internal dimensions.

6.3 Compression of neural networks

Tensor networks, particularly acyclic structures such as TTs and other tree-like networks, have been highly regarded in quantum information theory and condensed matter physics. Their popularity arises not only from the theoretical insights they enable but also from their computational efficiency, which makes it possible to simulate complex systems with relatively low computational effort. This efficiency has led to the adoption of TNs in various other fields to approximate complex computations effectively.

In deep learning, where models are composed of intricate combinations of linear layers followed by non-linear activations, TNs have typically been employed to reduce the number of parameters in the inner linear layers. Among the available tensor decomposition techniques, the most common approach involves converting each matrix in a linear layer into a MPO/TTM [74, 76–78, 80, 81]. Other decomposition methods, such as the CP or Tucker decompositions, have also been successfully applied [75, 79]. This kind of tensorized models are commonly referred to as Tensor Neural Networks (TNNs).

Although this approach has shown promising results in compressing deep CNNs [74, 75, 78] and transformer-based language models [79–81], this type of compression retains two key drawbacks: (i) since the NN architecture is preserved, the privacy and interpretability challenges of the original model remain present in the tensorized version, and (ii) although the decomposition significantly reduces the number of parameters that need to be stored—enabling edge computing on small devices—it does not always lead to a reduction in time complexity.

For example, in the base BERT model [14], some linear layers consist of matrices of size $768 \times 3\,072$. Both the storage and computational complexities of multiplying a vector by such a matrix scale with its dimensions. Specifically, storing the matrix requires 2 359 296 floating-point numbers, and the multiplication involves 2 359 296 floating-point operations.

On the other hand, if the same matrix were represented as a TTM, whose cores are of the form $G_k(\alpha_{k-1}, i_k, \alpha_k, j_k)$, where j_k represents the output index of each core and all ranks are set to $r = 10$, we would need to store only 3 850 numbers. This represents a reduction of more than 99.8% in the number of parameters. However, contracting this TTM with an input vector (not in TT form) would require 2 810 880 operations, which exceeds the computational cost of using the original matrix. This higher complexity arises even with a relatively small rank $r = 10$, which may already lead to significant approximation errors. The increased computational cost is primarily due to the need to recover the intermediate results as a full vector in order to apply non-linear activations element-wise, as these activations cannot be applied directly in TT form. The calculations assume the following decompositions for the input and output dimensions: $d^{in} = (1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 1)$ and $d^{out} = (2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3)$.

To illustrate this phenomenon, we have tensorized three NN models with varying numbers of hidden layers, from 1 to 3, using both TT-RSS and layer-wise tensorization with TTMs. The models are trained for the binary classification task described in Section 5.5.2. For both tensorization methods, we consider three scenarios by setting $r = 2$, $r = 5$, and $r = 10$. In the case of layer-wise tensorization, this entails fixing the chosen rank for all layers. While this choice may not be optimal—since different layers have varying sizes, and a uniform rank may affect them differently, thereby influencing the final accuracy—our primary goal is to compare the ability of these tensorization methods to reduce the number of parameters and computational costs. Nonetheless, to emphasize the expressive capacity of the resultant models, we re-train all tensorized models for 10 epochs, achieving accuracies comparable to those of the original NNs.

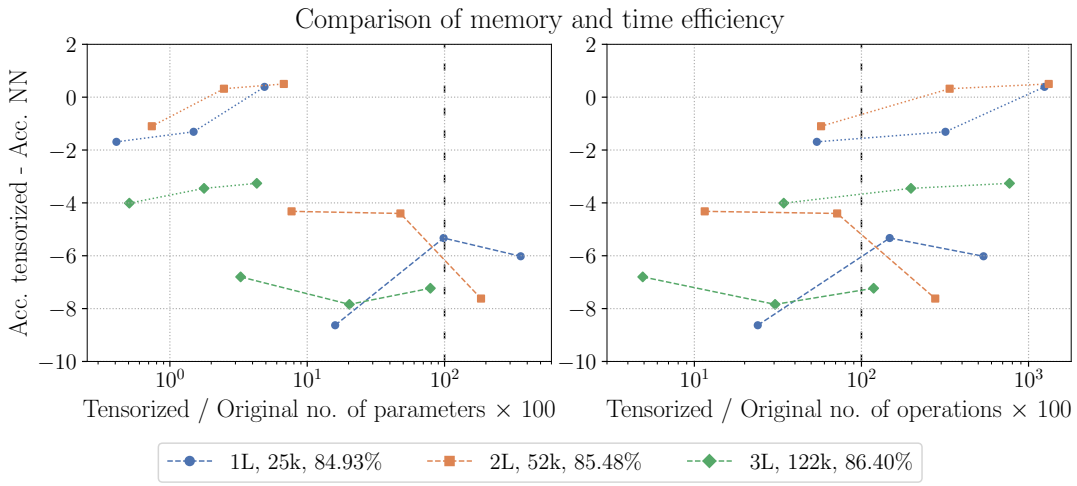


Figure 6.3: Comparison of memory and time efficiency between different tensorized models and various choices of ranks r. The x-axis represents the ratio of parameters/operations in the tensorized models to those in the original models. The y-axis marks the accuracy difference, where negative values indicate accuracy drops in the tensorized models compared to the originals. Each color corresponds to a different NN model being tensorized. The legend indicates the number of hidden layers, approximate number of parameters, and test accuracy of each model. Dashed lines represent TT models obtained via TT-RSS, while dotted lines denote layer-wise tensorized TNNs. Points connected by these lines indicate, from left to right, increasing ranks: $r = 2$, $r = 5$, and $r = 10$, with corresponding increases in the number of parameters and operations.

The calculations focus solely on the linear layers of the models, assigning a constant cost to the application of the non-linear activations in the NN models.

The results are shown in Fig. 6.3. We observe similar results to those in the BERT-layer example discussed above. Although layer-wise tensorization can achieve orders-of-magnitude improvements in memory efficiency, it can also lead to orders-of-magnitude decreases in time efficiency. On the other hand, since it retains the same structure as the original model, it is capable of achieving similar accuracies. This demonstrates that layer-wise tensorization is a viable method for eliminating redundancies, thereby uncovering a finer internal structure within the model.

For TT-RSS, the expressive power of the resultant TT models may differ significantly from that of the original NNs, leading to larger accuracy drops. However, TT-RSS occasionally offers a better balance between memory and time efficiency, showcasing its potential as an alternative tensorization method for applications where such trade-offs are critical.

Remark 6.3.1. The calculations in this section assume the input vector is not in TT form. In practice, it is common to first represent the input vector as a TT, typically through iterative application of truncated SVD, which incurs a cost of $O(d^n rn)$. Subsequently, the TT input vector is contracted with the TTM layer, which can be performed efficiently with a cost of $O(d^2 r^4 n)$. However, the output vector still needs to be returned as a full vector to apply the element-wise non-linear activations, which requires $O(d^n r^2 n)$ operations. In contrast, contracting the full

input vector with the TTM layer, as shown in our calculations, can be done in $O(d^{n+1}r^2n)$. This demonstrates that both strategies have comparable complexities, and our results also extend to the former case, particularly in our scenario where $d = 2$.

6.4 Conclusions and future directions

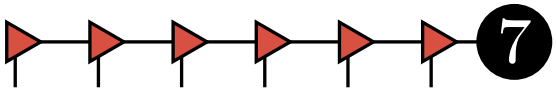
In this chapter, we have shown that the applications of a decomposition like TT-RSS extend beyond privacy, as illustrated through a series of examples which, although set in simplified scenarios, point to promising directions that deserve further investigation.

First, we have found that TT-RSS leads to interpretable representations. In particular, it allows the estimation of physical quantities (in our example, the SPT phase of the AKLT state) from the obtained models. While the AKLT example demonstrates the potential of the proposed methodology, the greatest advantage of our approach is likely to arise in higher-dimensional scenarios. In such cases, training TNs becomes computationally expensive, whereas NNs can be trained more efficiently. Therefore, extending these methods to recover TN models from NNQS for higher-dimensional layouts, such as 2D data, could offer a significant advantage over existing TN methods. Thus, investigating tensor decompositions that are extendable to PEPS opens an interesting future direction for studying 2D quantum systems, as well as for ML applications.

Secondly, we observe that TT-RSS can serve as an effective initialization technique for training TT models in general settings. In particular, one can first train a NN model, or simpler models such as linear or logistic ones, and then decompose it via TT-RSS to obtain a function that returns values within a controlled range across the data domain. More generally, if one seeks a random initialization of a TT that does not impose any bias—as may occur when tensorizing a particular type of model—one can tensorize a random function with the desired consistency. For instance, as in the case of "TT-RSS random", one can decompose a random, vector-valued function whose output vectors are only required to be normalized. Although the results of this approach were not comparable to other initializations such as "TT-RSS" or "Unitaries", as shown in Fig. 6.2, we believe this opens promising avenues toward a generalized initialization mechanism that enables the construction of random TT models with arbitrary input/output dimensions, ranks, and embeddings, while providing stable results within the domain of interest.

Finally, we compare TT-RSS as a model compression technique with the common approach of tensorizing NNs layer-by-layer. Although the latter can reduce the number of parameters by orders of magnitude, and even decrease training and inference times by 50% and 25%, respectively, as reported in Ref. [82], such improvements are only attainable when extremely low ranks can be imposed without significant loss of accuracy, and when combined with sophisticated computational methods that enable parallelization and GPU acceleration. However, disregarding these potential practical advantages, it is easy to see that in certain situations, even if layer-wise tensorization reduces the parameter count by 90%, the number of required operations may still increase. In contrast, we show that TT-RSS, yielding a single TT model, can offer a better balance between memory and time efficiency, demonstrating its potential as an alternative tensorization method for applications where such trade-offs are critical. Moreover, TT-RSS can also be applied layer-wise instead of to the full model. That is, one could obtain TT representations of each NN layer $\sigma(W^{(l)}h^{(l-1)}+b^{(l)})$ and concatenate the resulting TT layers, forming a complete TN with a tree structure. Although the applicability and advantages of such decompositions remain open for further investigation, this shows that more intricate tensorization schemes can be derived from TT-RSS, and we are not restricted to constructing only simple TT models.

TensorKrowch



Another major contribution of this thesis is the development of an open-source Python package for the integration of TN models into PyTorch, one of the most widely adopted machine learning frameworks. In this chapter, we introduce the package, TensorKrowch, presenting its fundamental components for creating and training TN models, and detailing some of the optimizations developed to improve model performance. It should be noted, however, that this chapter is not intended as a comprehensive tutorial on all the capabilities of the package, nor as a complete description of its full functionality. Such information can be found in the TensorKrowch documentation, available online at https://joserapa98.github.io/tensorkrowch.

7.1 Motivation

Throughout this thesis, we have introduced TN models and justified their use in machine learning for specific purposes. However, TN models also present several limitations that, to date, have hindered their widespread adoption. On the one hand, there are intrinsic limitations of TNs that complicate their training. Although we believe TT-RSS can be used to develop general initialization schemes, this has been a major drawback, often requiring training to rely on trial and error in search of suitable combinations of embeddings and initializations that yield stable optimizations. This has restricted the expressiveness of these models primarily to polynomial functions with coefficient vectors in TT form, which is insufficient for several applications and typically results in lower accuracies than NN models. Other architectures, such as Trees [62, 63] and PEPS [65], have also been proposed for their beneficial properties and increased expressive power for image data, but they are accompanied by increased computational costs and the resulting trainability challenges. Thus, from a machine learning perspective, extensive research is still needed to identify the scenarios in which TNs can be most effectively leveraged, as well as to determine the most efficient training methods, initialization strategies, and optimal architectures (including architectures beyond those commonly used in physics).

The other major limitation was the lack of appropriate software to enable rapid exploration of these issues, particularly with a focus on ML applications. Since TNs are widely adopted for simulating quantum systems, most available software is tailored to this domain, and although some knowledge is transferable from physics to ML, the latter field is far more exploratory and heuristic. For comparison, the widespread adoption and rapid development of deep learning have been accompanied by user-friendly software providing tools for building custom models, testing different parameter initializations, performing hyperparameter tuning, and designing optimization schemes. While some libraries already integrate TNs into ML frameworks, they are often optimized for specific TNs, such as TTs [83], or primarily focus on tensorization [84, 85]. On the other hand, libraries like TensorNetwork [86] offer greater architectural generality, but at the cost of added complexity when integrating them into specific frameworks, which limits the use of native modules or optimization tools provided by these frameworks.

These limitations motivated the development of an open-source software package, TensorKrowch, which integrates TN utilities into PyTorch [88]. TensorKrowch is designed to provide a comprehensive framework that enables rapid prototyping of TN layers both as standalone models and as components integrated within deep machine learning models. Its main characteristics are the following:

Generality: TensorKrowch allows users to construct arbitrary TNs by creating and connecting nodes. Users can selectively choose which nodes to train and define arbitrary operations between their components. In addition, TensorKrowch provides pre-built classes for the most common types of networks.

Ease of use: At the core of TensorKrowch is a Pythonic approach, offering a simple interface with building blocks and operations that can be combined to create complex models and training pipelines. These primary objects and operations are described in Sections 7.4 and 7.5, respectively.

Optimization: While the interface remains simple, numerous optimizations are implemented in the background to perform operations efficiently and reduce redundant computations during training. Section 7.7 details the optimizations performed.

Integration: TensorKrowch is built on top of a well-established deep learning framework, namely PyTorch [88]. This integration enables the creation of `TensorNetwork` models that function like any other PyTorch layer, seamlessly combining with existing components. Consequently, TensorKrowch fully leverages the capabilities of PyTorch, including GPU acceleration, automatic differentiation, and easy composition of multi-layer networks.

Moreover, TensorKrowch can also be used for tensorization purposes by substituting or approximating dense matrices in deep PyTorch models and applying the built-in matrix factorization techniques.

7.2 Installation and requirements

TensorKrowch is a Python package available on Linux, Mac and Windows operating systems. It can be installed via `pip` with the following command:

```
pip install tensorkrowch
```

The basic requirement of TensorKrowch is PyTorch [88], which is used as machine learning backend. Additionally, TensorKrowch requires opt_einsum [208], used in the implementation of the `einsum` operation and that allows for the automatic search of good network contraction paths via greedy algorithms.

The source code for TensorKrowch is hosted on GitHub at

https://github.com/joserapa98/tensorkrowch

and is distributed under the MIT License. More details about the software, packaging information, and guidelines for contributing to TensorKrowch are included in Section 7.8.

7.3 Basic usage

TensorKrowch provides a set of basic components, namely `Node`, `Edge`, `TensorNetwork`, and variants of these, that can be combined to build trainable models. The usual workflow consists in the following steps: 1) Define the structure of the graph by creating nodes and connecting them; and initialize the tensors within those. 2) Specify which nodes will be used to store the input tensors coming from the training dataset. 3) Define the contraction algorithm to reduce the whole network to a single output tensor, which one can input to any other layer that might follow the TN.

To carry out these steps, one needs to know how to create and combine the different building blocks of the model, and how to operate them to contract the network. These topics will be covered in detail in Sections 7.4 and 7.5, respectively.

Once the custom TN is defined, the process of training it is analogous to what one would do in vanilla PyTorch. To illustrate this with an example, let us introduce one of the built-in classes provided by TensorKrowch, `MPSLayer`. This class is a variation of the traditional MPS with an additional node that has a dangling edge representing an output dimension. As a result, the `MPSLayer` is contracted into a vector of the specified dimension.

```
import torch
import tensorkrowch as tk

# Instantiate model
mps = tk.models.MPSLayer(n_features=1001,
                         in_dim=2,
                         out_dim=10,
                         bond_dim=10)
```

Tensor network models expect as input a rank-3 tensor with dimensions $b \times n \times d$, being b the size of the batch, n the number of feature vectors, and d their dimension. This tensor, which is a batch of sequences of vectors, represents a TN itself, given by the tensor product of all the vectors corresponding to each batch element. These vectors will be placed in specific nodes so that the network can be contracted with these new data.

However, data tensors tend to come in the form of $b \times n$ matrices, requiring a previous transformation to get the proper rank-3 tensor. That is, for each batch element there is a vector of features, that has to be turned into a sequence of feature vectors. To accomplish this, each feature has to be embedded into a vector space. TensorKrowch provides five common embedding functions, namely `unit`, `add_ones`, `poly`, `discretize` and `basis`, the first two being introduced in [58] and [59], respectively.

```
# Shape: batch_size x n_features
data = torch.randn(100, 1000)

# Shape: batch_size x n_features x in_dim=2
data = tk.embeddings.unit(data)

# Shape: batch_size x out_dim=10
labels = torch.randn(100, 10)
```

To reduce repetition of ancillary computations due to dealing with the TN structure, TensorKrowch introduces some extra steps that need to be carried out before starting training. These involve setting memory modes and tracing the model; both advanced features will be covered in Section 7.7.

```
# Set memory modes
mps.auto_stack  = True
mps.auto_unbind = False

# Trace
example = torch.zeros(1, 1000, 2)

# Shape: batch_size=1 x n_features x in_dim=2
mps.trace(example)
```

To be able to train, one needs to set a loss function to minimize and an optimizer to update the parameters of the model according to their gradients. It is important that the optimizer is set after the model has been traced, since the parameters of the model might change during this process.

```
loss_fun  = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mps.parameters(),
                             lr=1e-4,
                             weight_decay=1e-2)
```

Finally, the above ingredients can be put together in the training loop.

```
for epoch in range(n_epochs):
    # Contract TN with input data
    scores = mps(data)

    # Compute the loss
    loss = loss_fun(scores, labels)

    # Backpropagate and update parameters
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

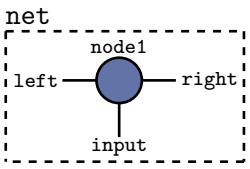
7.4 The building blocks

The main structure of TensorKrowch is similar to that of TensorNetwork [86]. Namely, the main object in TensorKrowch is a `TensorNetwork`, that is populated with `Nodes`. These have `Edges` that are connected according to the desired structure. Creating a `TensorNetwork` is done as follows:

```
import torch
import tensorkrowch as tk
net = tk.TensorNetwork()
```

`Nodes` are the basic elements that make up a `TensorNetwork`. They serve as containers for PyTorch's `torch.Tensor` objects and hold essential information for building, operating, and training the network. Key aspects associated with nodes include shape, tensor, axes, edges, network membership, and successors. Tensors themselves are not actually contained within nodes; instead, they are stored in the shared memory system that is accessible to all nodes in the `TensorNetwork` (see Section 7.7 for details). To create a `Node` one can specify its name, shape, names for its axes, network, and an initialization method to create a new tensor for it. Together with the code we present a graphical notation to depict not only the TN but all the elements that form the `TensorNetwork` object.

```
node1 = tk.Node(name="node1",
                shape=(2, 5, 2),
                axes_names=("left", "input", "right"),
                network=net,
                init_method="randn")
```

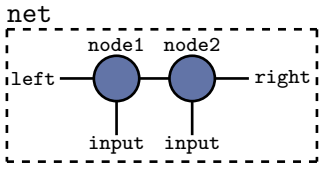


Specifying a shape automatically creates a set of edges for the `Node`. An `Edge` is nothing more than an object that wraps references to the nodes it connects. Thus it stores information like the nodes it connects, the corresponding nodes' axes it is attached to, its size, etc.

To connect nodes, one can access the desired edges using their names, and connect them with the caret operator:

```
# Equivalent to initialize Node with
# "randn" init_method
node2 = tk.randn(name="node2",
                 shape=(2, 5, 2),
                 axes_names=
                     ("left", "input", "right"),
                 network=net)

node1["right"] ^ node2["left"]
```



It is important to note that the operation `^` does not perform contractions, since it may be done on edges of empty nodes. As it will be shown below, contraction between tensors is performed by using the `@` operator, that contracts tensors along all the edges that connect both nodes.

Edges can be designated as *batch* edges if one includes the string `"batch"` in the name of the corresponding axis. This allows for performing batch contractions if the nodes involved in the

operation both have batch edges sharing the same name. This functionality is analogous to that of PyTorch functions like, for instance, `torch.bmm`, used to perform batch matrix multiplication.

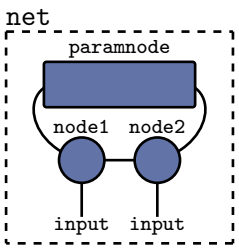
7.4.1 Types of nodes

TensorKrowch features several types of nodes, that have different functionalities and goals within the package.

Parameter nodes: In analogy with PyTorch having `torch.Tensor` to denote a tensor of non-trainable entries and `torch.nn.Parameter` for tensors of trainable parameters, TensorKrowch has `Node` to denote containers of tensors with non-trainable entries and `ParamNode` to denote containers of tensors with trainable entries. One can instantiate a `ParamNode` with:

```
paramnode = tk.ParamNode(name="paramnode",
                         shape=(2, 2),
                         axes_names=("left", "right"),
                         network=net,
                         init_method="randn")

paramnode["left"] ^ node1["left"]
paramnode["right"] ^ node2["right"]
```



In the graphical notation we will use squares for denoting trainable nodes.

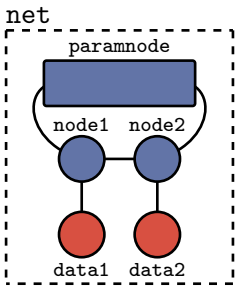
Leaf nodes: These are the nodes that form the `TensorNetwork`, along with the data nodes (see below). Usually, leaf nodes will be the trainable nodes. All leaf nodes contain PyTorch leaf tensors, this is, tensors that are not generated by operations tracked by PyTorch's automatic differentiation engine. By default, all nodes are leaf. In the graphical notation, these are the blue nodes.

Data nodes: These are nodes that are used to store the tensors coming from input data. It is possible to instantiate data nodes directly by specifying `data=True` in the initialization of `Node`. However, data nodes will be usually created when specifying where they should be put in the network using `set_data_nodes` (see Section 7.6). Graphically, data nodes will be depicted as red circles.

```
data1 = tk.Node(name="data1",
                shape=(5,),
                axes_names=("feature",),
                data=True,
                network=net)

data2 = tk.Node(name="data2",
                shape=(5,),
                axes_names=("feature",),
                data=True,
                network=net)

# We can add input data tensors to the nodes
data1.tensor = torch.randn(5)
data2.tensor = torch.randn(5)
```

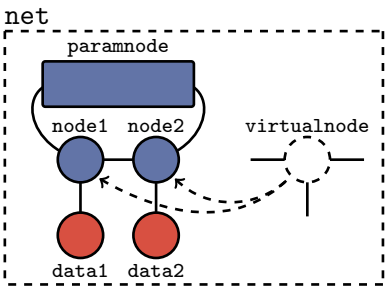


```
node1["input"] ^ data1["feature"]
node2["input"] ^ data2["feature"]
```

Virtual nodes: These nodes are internal nodes that TensorKrowch uses as shortcuts when one wishes to have the same tensor in different nodes, or to process auxiliary information without adding new leaf nodes to the network. This is useful when defining uniform (equivalently, TI) TNs [209], as all their nodes simply use a single tensor stored in a virtual node instead of one tensor per node. Virtual nodes are intended to be mainly internal, but they can be created manually using the argument `virtual=True`. This is needed when defining custom uniform architectures beyond those natively implemented, such as `UMPS`, `UMPO`, `UPEPS` or `UTree`. Virtual nodes are represented by empty nodes with dashed borders.

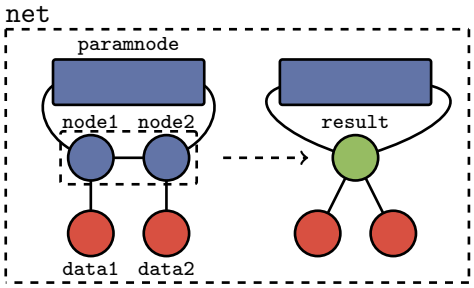
```
virtual = tk.Node(name="virtualnode",
                  shape=(2, 5, 2),
                  axes_names=
                   ("left", "input", "right"),
                  virtual=True,
                  network=net,
                  init_method="randn")

# Put virtual's tensor into node1 and node2
node1.set_tensor_from(virtual)
node2.set_tensor_from(virtual)
```



Resultant nodes: The result of a contraction of tensors (stored in their respective nodes) is another tensor that must be stored in a new node. These nodes are resultant nodes, which coexist in the `TensorNetwork` object together with the original nodes, inheriting their edges. This allows for subsequent contraction with other neighbouring nodes. Resultant nodes are automatically created upon contracting two nodes on a same network, or when tracing the model on its first run (see Section 7.5.1). Resultant nodes are displayed as green circles.

```
# Nodes are contracted along their
# connected edges with the @ operator
result = node1 @ node2
```



7.5 Operations between nodes

Operations between TensorKrowch `Nodes` are instances of a class, `Operation`, that is optimized to avoid unnecessary repeating of basic computations (see Section 7.7 for details). Operations create resultant nodes that inherit information from the parent nodes. TensorKrowch implements many operations that are extensions of those in vanilla PyTorch, like `permute`, `tprod` (`outer` in PyTorch), `mul`, `div`, `add` and `sub`. The complete list of implemented operations

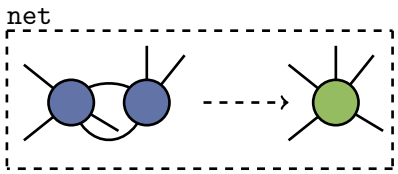
is available in the corresponding page of the documentation. In addition to these, TensorKrowch includes implementations of operations that are inherent to TNs. These are:

Contract: Enables contraction of all the edges connecting two nodes, although it is also possible to use it to contract only a selected range of edges. Furthermore, batch contraction is automatically performed if some edge is of batch type (recall, with the string `"batch"` in its name). This is the most basic operation one can use, allowing for the contraction of the whole network just by implementing a contraction path.

```
node1 = tk.randn(shape=(2, 4, 3, 6, 2))
node2 = tk.randn(shape=(3, 2, 5, 4))

# Edges can also be accessed by index
node1[2] ^ node2[0]
node1[4] ^ node2[1]
# node1 and node2 will belong to the same
# network upon connection

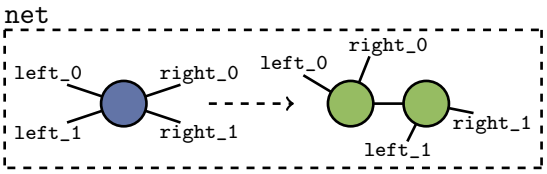
result = node1 @ node2
# result will have all the non-contracted
# edges from node1 and node2
```



Split: As an inverse of the `contract` operation, `split` factorizes the tensor in a node into two, selecting which edges should go to each resultant node. Factorization algorithms include singular value and QR decompositions, with additional functionalities such as selecting the desired amount of singular values to keep, modifying the rank of the resultant nodes accordingly. This can be useful to reduce the bond dimension in cases where it might explode during the contraction algorithm. Furthermore, by iterative application of `split`, arbitrary tensors can be decomposed into TN formats, which enables defining custom tensorization routines [74, 75, 79].

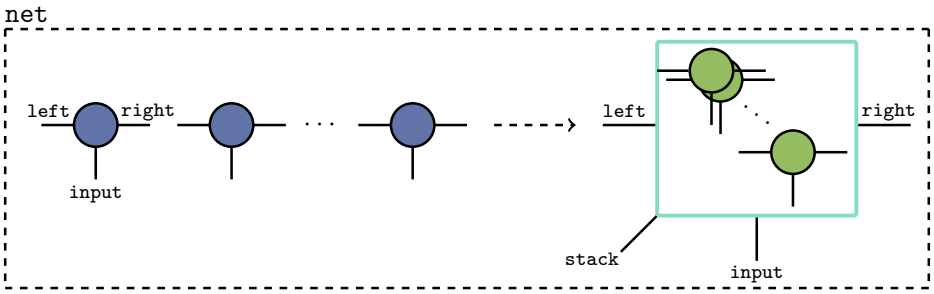
```
node = tk.randn(
    shape=(2, 7, 3, 4),
    axes_names=
        ("left_0", "left_1",
         "right_0", "right_1")
)

node_left, node_right = tk.split(
    node,
    ["left_0", "right_0"],
    ["left_1", "right_1"],
    mode="svd",
    rank=5
)
```



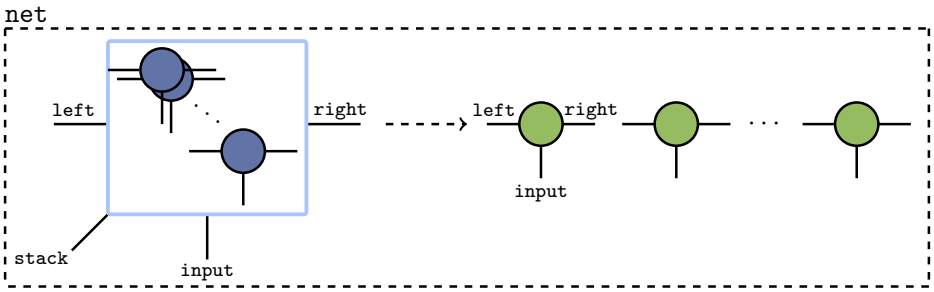
Stack: In many situations, stacking a set of tensors into a larger tensor with one extra dimension speeds up operations by allowing for parallelizing contractions (see Section 7.7 for more details). Creating such stacks is achieved with `stack`. The node resulting from this operation is a special type of node, namely a `StackNode` or `ParamStackNode`, that is only intended for internal use.

```
net    = tk.TensorNetwork()
nodes = [tk.randn(shape=(2, 4, 2),
                  axes_names=("left", "input", "right"),
                  network=net)
         for _ in range(n_nodes)]
stacknode = tk.stack(nodes)
```



Unbind: As the counterpart of `stack`, this enables one to unbind a stack of resultant nodes, returning them in a list. Only `StackNodes` and `ParamStackNodes` can be unbound.

```
stacknode = tk.stack(nodes)
result = tk.unbind(stacknode)
```



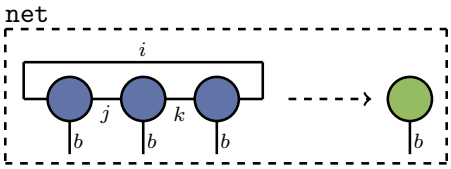
Einsum: Allows for the implementation of complex contractions along sets of edges. Instead of contracting along a connected edge, one can define a contraction path following the Einstein summation convention to contract along several edges at once. This operation uses opt_einsum [208] at its core, making specific checks and simplifications beforehand to adhere to the rules and structures defined in TensorKrowch. For example, indices used twice can only correspond to nodes already connected by an edge in the specified axes. There is a variant of this operation, `stacked_einsum`, which allows to use lists of nodes as inputs, to perform `stack` followed by `einsum` in the same operation.

```
node1 = tk.randn(shape=(10, 15, 100),
                 axes_names=("left", "right", "batch"))
node2 = tk.randn(shape=(15, 7, 100),
                 axes_names=("left", "right", "batch"))
```

```
node3 = tk.randn(shape=(7, 10, 100),
                 axes_names=("left", "right", "batch"))

node1["right"] ^ node2["left"]
node2["right"] ^ node3["left"]
node3["right"] ^ node1["left"]

result = tk.einsum("ijb,jkb,kib->b", node1, node2, node3)
```



For a comprehensive explanation of all these operations and the arguments they admit, the reader is referred to the TensorKrowch documentation.

7.5.1 Trace and reset

Every operation in TensorKrowch returns a new resultant `Node` that stores the output tensor and inherits the non-contracted edges of its parents. Therefore, in order to reduce the creation of redundant nodes and the amount of memory used for this purpose, it is useful to generate void containers for the resultant tensors. This is achieved with the `trace` operation, which can be called using an example input with batch dimension 1 in order to create all the necessary resultant nodes in the fastest way possible. Further details and explicit comparisons can be found in Section 7.7.

```
# Shape: batch_size x n_features x feature_dim
data = torch.randn(100, 20, 5)
example = torch.zeros(1, 20, 5)

net.trace(example)
```

The inverse of the `trace` function is `reset`. This function deletes all the resultant nodes created during training, resetting the network to its initial state. This is useful when one wants to make changes to the structure of the network, to switch on/off the memory modes, or to save a trained model (otherwise, calling `torch.save(net.state_dict())` will save a *traced* model, whose parameters might not be exactly the same as the ones in the original model, due to internal optimizations).

```
# After training
net.reset()

# Save model
torch.save(net.state_dict(), "net.pt")
```

```
# Load model
new_net = TensorNetwork()
new_net.load_state_dict(torch.load("net.pt"))
```

7.6 Building and training TN models

Sections 7.4 and 7.5 showed how one can create nodes belonging to a `TensorNetwork`, and operate them to contract the network. However, although this functionality might be useful for experimentation, the main usage of TensorKrowch will be to define custom models as subclasses of `TensorNetwork`. This allows, for instance, to instantiate TNs that work as any other PyTorch layer.

The workflow to define custom TNs is similar to how one defines custom layers in PyTorch. There, one needs to subclass `torch.nn.Module`, to define the parameters and architecture of the layer in the `__init__` method, and to specify how input data is processed by the layer in the `forward` method.

Similarly, for defining a custom TN in TensorKrowch one needs to subclass `TensorNetwork`, overriding the following methods:

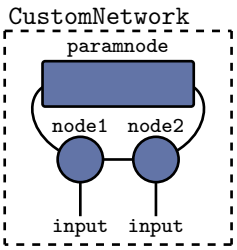
`__init__`: Defines the graph of the TN and initializes the tensors of the nodes.

```
class CustomNetwork(tk.TensorNetwork):

    def __init__(self):
        super().__init__(name="CustomNetwork")

        self.node1 = tk.randn(
            shape=(2, 5, 2),
            axes_names=("left", "input", "right"),
            name="node1",
            network=self
        )
        self.node2 = tk.randn(
            shape=(2, 5, 2),
            axes_names=("left", "input", "right"),
            name="node2",
            network=self
        )
        self.paramnode = tk.randn(
            shape=(2, 2),
            axes_names=("left", "right"),
            name="paramnode",
            network=self,
            param_node=True
        )

        self.node1["right"] ^ self.node2["left"]
        self.paramnode["left"] ^ self.node1["left"]
        self.paramnode["right"] ^ self.node2["right"]
```



`set_data_nodes` **(optional):** Creates the data nodes where the data tensors will be placed. Usually, it will just select the edges to which the data nodes should be connected, and call the parent method.

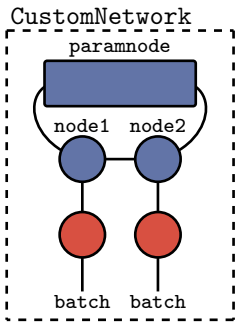
```
class CustomNetwork(tk.TensorNetwork):

    # ...

    def set_data_nodes(self):
        # Collect edges to which data
        # nodes will be connected
        input_edges = [self.node1["input"],
                       self.node2["input"]]

        # Define number of batch indices
        # for the input
        num_batch_edges = 1

        # Call parent method
        super().set_data_nodes(input_edges,
                               num_batch_edges)
```



`add_data` **(optional):** Places input data into the previously specified data nodes. Commonly, all data nodes will have the same shape, namely $b \times d$, being b the batch size and d the feature dimension. Assuming there are n of these nodes (typically one per feature), the input to `add_data` must be a tensor of dimension $b \times n \times d$. On default operation, the input tensor will be then unbound at its second axis, delivering each slice of shape $b \times d$ to each of the data nodes. However, this method can be overridden to customize how data is set into the data nodes.

`contract`: Very much like the `forward` method in PyTorch, this is the main method that describes how the components of the network are combined. In contrast to vanilla PyTorch, however, in a TensorKrowch `TensorNetwork` the `forward` method shall not be overriden, since its goal is to just call `set_data_nodes`, if needed, `add_data`, and `contract`, and then it will return the tensor corresponding to the last resultant node. Instead, one should customize the `contract` method. TensorKrowch does not implement algorithms for searching optimal contraction paths [210]. Thus, one must specify custom contraction algorithms for each user-defined TN, via `einsum` (recall Section 7.5) or by any other means. As will be detailed in Section 7.7, the order in which TensorKrowch `Operations` appear in the algorithm is significant, being mandatory that the last `Operation` is the one returning the final node.

```
class CustomNetwork(tk.TensorNetwork):

    # ...

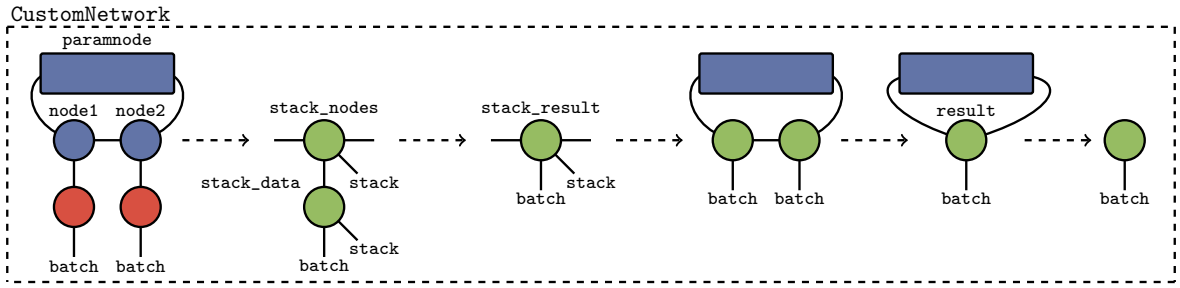
    def contract(self):
        stack_nodes = tk.stack([self.node1, self.node2])
        stack_data  = tk.stack(list(self.data_nodes.values()))
```

```
        # Stacks need to be reconnected before contraction
        stack_nodes["input"] ^ stack_data["feature"]
        stack_result = stack_nodes @ stack_data

        stack_result = tk.unbind(stack_result)

        result = stack_result[0] @ stack_result[1]

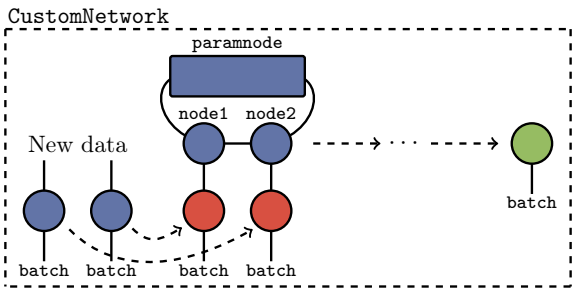
        # Last operation must return the output node
        result @= self.paramnode
        return result
```



With the subclass correctly defined, one can now instantiate the custom network and feed it with new input data tensors:

```
net = CustomNetwork()

# Pass data to the model
# Shape: batch_size x n_features x feature_dim
data = torch.randn(100, 2, 5)
result = net(data)
```



As mentioned in the beginning of the section, creating a custom network as a subclass of `TensorNetwork` makes the integration of TN layers within PyTorch models straightforward:

```
import torch.nn as nn
```

```
# Combine built-in custom TN layer with PyTorch layers
model = nn.Sequential(
    tk.models.MPSLayer(n_features=101,
                       in_dim=2,
                       out_dim=10,
                       bond_dim=5),
    nn.ReLU(),
    nn.Linear(10, 10)
)
```

The last codeblock contains a built-in class, `MPSLayer`, that readily implements a MPS architecture. Similar implementations are available for MPOs, PEPS and Trees, both uniform and non-uniform. This direct integration also allows to efficiently define tensorized neural network models through the interleaving of MPOs, or simply `MPSLayers`, with the common nonlinearities of neural networks.

As discussed, TNs have gauge symmetries that allow fixing preferred sets of parameters defining the same model, which we call canonical forms, and is the property that makes TNs robust against the privacy vulnerability presented in Chapter 5. TensorKrowch allows, in the built-in MPS and Tree implementations, to compute canonical forms using the function `canonicalize` (or `canonicalize_univocal` for the univocal canonical form described in Section 5.4).

7.6.1 Model training

The provided code enables users to create custom `TensorNetwork` models. To train them, different approaches can be followed. In PyTorch, it is customary to pass the model parameters to native optimizers that implement different gradient descent algorithms. Gradients are automatically computed via the automatic differentiation engine, which tracks all operations in which model parameters are involved and computes gradients by applying the chain rule. Since `TensorNetwork` models are defined as subclasses of `torch.nn.Module`, and the `ParamNodes` within these are of type `torch.nn.Parameter`, this type of gradient-based optimizations are effortlessly implemented to train the `ParamNodes`. This approach enables the implementation of models similar to the one presented in Ref. [59], where all MPS nodes are optimized at the same time. This, however, is not the only way to train TNs in TensorKrowch. With the capability to parameterize and de-parameterize `Nodes` and `ParamNodes`, respectively, and assign different tensors to specific `Nodes`, various optimization schemes can be explored. For instance, DMRG-like approaches, as described in Ref. [58], can be easily implemented by freezing all MPS nodes, except for a trainable block that can traverse the matrix chain. The TensorKrowch documentation includes examples illustrating how to train MPS models in different fashions. Besides those, implementations of hybrid NN-TN models [162] and TNNs [74] can be found.

7.7 Time and memory optimizations

Operating TNs requires a careful handling of memory, since the memory requirement may vary drastically with the contraction path. In addition to this, it is always desirable to have fast and efficient operations in machine learning pipelines. To ensure efficient memory utilization, TensorKrowch employs a memory management scheme where nodes do not possess their own

memory. Instead, the memory is stored within the `TensorNetwork` object, and nodes are just pointers to the corresponding addresses in this shared memory. This design choice enables memory sharing among all elements of the model, facilitating operations and allowing nodes to utilize tensors from other nodes. However, it is important to note that memory sharing is limited to elements within the same object, meaning that nodes created in different `TensorNetworks` will not share memory. By adopting this memory management approach, TensorKrowch incorporates a range of optimizations that effectively reduce both time and memory overheads:

7.7.1 Operations

Tensor network operations in TensorKrowch are not simple functions, but rather instances of a class, `Operation`, that is designed to minimize redundant steps during training. Each node operation consists of two functions: one that is executed the first time the operation is called, and one that is executed in every subsequent call with the same arguments. Although these functions are similar, the former makes extra computations regarding the creation of the resultant nodes and some auxiliary operations that yield the same result in every call. For instance, when contracting two nodes, tensors are typically permuted first; how this permutation is carried out is always the same, in spite of the fact that the tensors themselves may be different.

Furthermore, to keep track of repeated calls to an `Operation`, a new object is created during the first run: `Successor`. This is a class intended for internal use that acts as a cache memory, storing the arguments that were used to call the operation, some hints regarding the auxiliary tensor-network-related computations, and a reference to the resultant nodes created. Hence, once an operation has been called, both the parent and children nodes are determined, and only their tensors will change in further contractions. This enables reducing all the code of the contraction algorithm, which may include plain Python code to collect parent nodes, into a sequence of calls to TensorKrowch `Operations`. Because of this simplification, the order in which these operations are called is relevant. Consequently, the last operation must always be the one returning the final node that corresponds to the contraction of the entire network.

These two optimizations (having different functions for different calls, and exploiting cache memory) break the whole contraction into a set of basic tensor operations that are computed sequentially, thus improving the efficiency of the training process.

7.7.2 Tracing

In Section 7.5.1 the `trace` operation was introduced as a means of keeping heavy auxiliary operations involved in the first run of the contraction out of the training loop. Tracing the model not only saves time, but also saves memory. While tracing a `TensorNetwork` model, a new memory is created to keep track of which nodes are involved in each operation of the contraction algorithm. This allows to free up the memory of data or resultant nodes that have already taken part in some operation but are not going to be needed any more in the contraction. An explicit example of the difference in memory usage that tracing the model produces can be found in Fig. 7.1.

7.7.3 Memory Modes

Additionally, there are two modes that can change how TNs utilize their memory. These modes, `auto_stack` and `auto_unbind`, allow the `stack` and `unbind` operations, respectively,

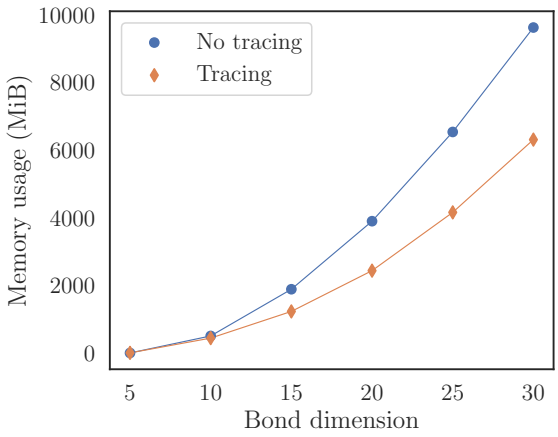


Figure 7.1: Comparison of the maximum memory usage of one contraction of the built-in `MPSLayer` class, when the model is traced or not, using different bond dimensions. Contraction is performed in a training regime: 1) an example data tensor is passed through the model, 2) gradients are computed via backpropagation, and 3) parameters are updated according to the gradients. All the contractions are computed in CPU using a batch size of 500, with both memory modes `auto_stack` and `auto_unbind` set to `True`, both contraction arguments `inline_input` and `inline_mats` also set to `True`, and the following arguments for the `MPSLayer` model: `n_features`=1 000, `in_dim`=2, `out_dim`=10. All the experiments were run on an Intel Xeon CPU E5-2620 v4 with 256GB of RAM.

to reuse information stored in other parts of the memory instead of recalculating it in every contraction. This helps accelerating both, training and inference. To illustrate how these modes affect the efficiency of the TN model, Fig. 7.2 presents a comparison of running times for the built-in `MPSLayer` class, when these modes are activated or deactivated.

7.8 Additional package information

This section provides additional information about the TensorKrowch package, including the available online documentation and guidelines for contributing to the development of the package. These resources are intended to support both users and developers in effectively using and extending TensorKrowch.

7.8.1 Documentation for TensorKrowch

The documentation for TensorKrowch is available online at https://joserapa98.github.io/tensorkrowch. It consists of a comprehensive user's guide and an API glossary. The user's guide provides detailed information that expands upon the topics covered in this chapter. It includes more information on the installation and in-depth tutorials with examples ranging from the basic usage of `Nodes` and `Edges` to building advanced hybrid NN-TN models like the one discussed in Ref. [162]. The API glossary is automatically generated from the docstrings (formatted comments to code objects), containing detailed information about the public functions and classes defined in TensorKrowch.

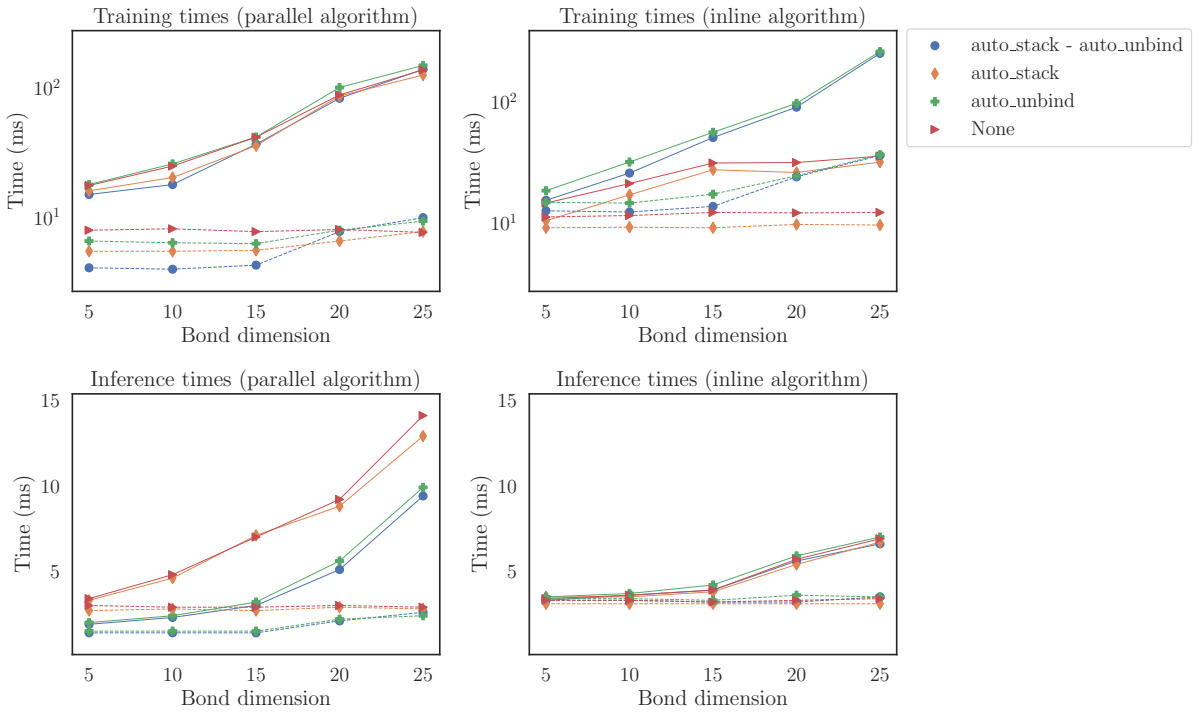


Figure 7.2: Comparison of running times of one contraction of the built-in `MPSLayer` class, using different bond dimensions. Contraction is performed in different regimes: training/inference, parallel/inline algorithm, CPU/GPU execution, and using different combinations for the options `auto_stack` and `auto_unbind`. For training, 1) an example data tensor is passed through the model, 2) gradients are computed via backpropagation and 3) parameters are updated according to the gradients. For inference, only the example data tensor is passed to perform one contraction of the model. "Parallel" and "inline" refer to the two possible algorithms that can be used to contract `MPSLayer`, specified by the argument `inline_mats`. When `inline_mats = True` (inline), the matrices of the MPS are contracted sequentially; when `inline_mats = False` (parallel), matrices are iteratively stacked, contracted in parallel and unbound until the chain of matrices is reduced to a single one. The argument `inline_input` is always set to `False`, indicating that the contraction of the MPS nodes with the respective input nodes is performed in parallel. Solid lines represent CPU times, dashed lines represent GPU times. All the contractions are computed using a batch size of 100 and the following arguments for the `MPSLayer` model: `n_features`=100, `in_dim`=2, `out_dim`=10. All the experiments were run on an Intel Xeon CPU E5-2620 v4 with 256GB of RAM and an NVIDIA GeForce RTX 3090.

7.8.2 Contribution guidelines

We welcome contributions to TensorKrowch from the wider communities interested in integrating TNs within machine learning frameworks, and in quantum information theory. Contributions can include feedback about the package, feature requests, bug fixes, or code contributions via pull requests.

Feedback and feature requests can be done by opening an issue on the TensorKrowch GitHub

repository [174]. Bug fixes and other pull requests can be done by forking the TensorKrowch source code, making changes, and then opening a pull request to the TensorKrowch GitHub repository. Pull requests are peer-reviewed by TensorKrowch's core developers to provide feedback and/or request changes.

Contributors are expected to adhere to TensorKrowch development practices including style guidelines and unit tests. Tests are written with the PyTest Python framework and are implemented outside the module. To test installation or changes, one can download the source code from the repository, and use standard PyTest functions. For example, executing the following in a Unix terminal in the test folder runs all the tests:

```
python -m pytest -v
```

7.9 Conclusions and future directions

Machine learning research relies heavily on rapid prototyping and iteration. By building on top of PyTorch, TensorKrowch enables these features for ML architectures based on TNs. With it, it is possible to use, off the shelf, standard TNs (MPS, MPOs, PEPS and Trees) either as standalone architectures or as layers in deep models, as well as defining custom networks. In the latter case, the user has access to customizing fine details of how the network processes input data, such as specifying which parts of the input are sent to which nodes, or fully customizing the contraction path. Our aim is that TensorKrowch contributes to the wide adoption of TNs by the machine learning community, and that allows to go beyond (and helps advancing) the body of knowledge generated by the communities working in quantum information theory and quantum many-body physics.

In this chapter we have described the main logic behind TensorKrowch, as well as the broad families of cases in which it can be used. We have also detailed the features of its building blocks, and many optimizations that are performed behind the scenes in order to obtain a fast and efficient operation. However, for scenarios where efficiency is paramount, such as quantum computer simulation [55, 56] or tensorization of NNs for edge computing [77], there may be libraries with better resource management. Nevertheless, TensorKrowch remains useful even in those cases, as it can be part of a preliminary prototyping step.

Since TensorKrowch has a strong focus on machine learning, the priority has been to implement TNs understood as architectures that parameterize intricate families of functions. Alternative perspectives, such as considering TNs as quantum states, have assumed a secondary role. Despite this, given the importance that the quantum information perspective can have in ML explainability [62, 69, 70], TensorKrowch supports tensors with complex coefficients and implements the calculation of reduced density matrices and entanglement entropy. Building more elaborated explainability tools, and exploiting TensorKrowch for quantum simulation (for instance, by leveraging PyTorch's automatic differentiation engine for calculating ground-state energies), will be the first steps to be addressed in the future.

On the other hand, other interesting operations include multiple tensor decompositions known in the numerical linear algebra community [130]. TensorKrowch allows for automatic splitting of tensors appearing in ML (such as those appearing in linear and convolutional layers) into MPO forms via SVDs. Additionally, the continuous version of TT-RSS, which also applies to discrete cases when using trivial embeddings, is implemented in TensorKrowch. Other tensoriza-

tions that are relevant in the machine learning literature, such as the canonical polyadic [189] and the Tucker decompositions [211], are not yet natively implemented. This is another upcoming objective to develop.

Finally, there are numerous other methods that will progressively be implemented, such as allowing for the modification of the number of edges a node has through a reshape operation, or incorporating visual feedback for observing the constructed graph, akin to the graphical notation used throughout this thesis.

Further information on the package, as well as end-to-end examples using state-of-the-art TN and hybrid architectures on standard datasets, can be found on the documentation website. We strongly encourage interested users to contribute to the development of the package via its repository [174].

Bibliography

[1] J. J. Hopfield. "Neural networks and physical systems with emergent collective computational abilities". In: *Proc. Natl. Acad. Sci. USA* 79.8 (1982), pp. 2554–2558.

[2] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. "A learning algorithm for Boltzmann machines". In: *Cogn. Sci.* 9.1 (1985), pp. 147–169.

[3] G. E. Hinton. "Training Products of Experts by Minimizing Contrastive Divergence". In: *Neural Comput.* 14.8 (2002), pp. 1771–1800.

[4] G. E. Hinton, S. Osindero, and Y.-W. Teh. "A fast learning algorithm for deep belief nets". In: *Neural Comput.* 18.7 (2006), pp. 1527–1554.

[5] Y. Bengio et al. "Greedy Layer-Wise Training of Deep Networks". In: *Adv. Neural Inf. Process. Syst.* Vol. 19. MIT Press, 2006.

[6] Y. LeCun et al. "Handwritten digit recognition with a back-propagation network". In: *Adv. Neural Inf. Process. Syst.* Vol. 2. Morgan-Kaufmann, 1989.

[7] D. E. Rumelhart and J. L. McClelland. "Learning internal representations by error propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. 1986, pp. 318–362.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Adv. Neural Inf. Process. Syst.* Vol. 25. Curran Associates, Inc., 2012.

[9] V. Nair and G. E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proc. 27th Int. Conf. Mach. Learn.* Omnipress, 2010, pp. 807–814.

[10] K. Simonyan and A. Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556.

[11] C. Szegedy et al. "Going deeper with convolutions". In: *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.* 2015, pp. 1–9. arXiv: 1409.4842.

[12] K. He et al. "Deep residual learning for image recognition". In: *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.* 2016, pp. 770–778. arXiv: 1512.03385.

[13] A. Vaswani et al. "Attention is All you Need". In: *Adv. Neural Inf. Process. Syst.* Vol. 30. Curran Associates, Inc., 2017. arXiv: 1706.03762.

[14] J. Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proc. NAACL HLT 2019, Vol. 1.* Association for Computational Linguistics, 2019, pp. 4171–4186. arXiv: 1810.04805.

[15] T. Brown et al. "Language Models are Few-Shot Learners". In: *Adv. Neural Inf. Process. Syst.* Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. arXiv: 2005.14165.

[16] H. Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971.

[17] A. Chowdhery et al. "PaLM: Scaling Language Modeling with Pathways". In: *J. Mach. Learn. Res.* 24.240 (2023), pp. 1–113. arXiv: 2204.02311.

[18] G. Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2024. arXiv: 2312.11805 [cs.CL].

[19] G. Aad, T. Abajyan, B. Abbott, et al. "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC". In: *Phys. Lett. B* 716 (2012), pp. 1–29.

[20] P. Baldi, P. Sadowski, and D. Whiteson. "Searching for exotic particles in high-energy physics with deep learning". In: *Nat. Commun.* 5.1 (2014), p. 4308. arXiv: 1402.4735.

[21] J. Schmelzle et al. *Cosmological model discrimination with Deep Learning*. 2020. arXiv: 1707.05167 [astro-ph.CO].

[22] F. Noé et al. "Machine Learning for Molecular Simulation". In: *Annu. Rev. Phys. Chem.* 71 (2020), pp. 361–390.

[23] J. Carrasquilla et al. "Reconstructing quantum states with generative models". In: *Nat. Mach. Intell.* 1.3 (2019), pp. 155–161. arXiv: 1810.10584.

[24] H.-Y. Huang, R. Kueng, and J. Preskill. "Predicting many properties of a quantum system from very few measurements". In: *Nat. Phys.* 16.10 (2020), pp. 1050–1057. arXiv: 2002.08953.

[25] T. Schmale, M. Reh, and M. Gärttner. "Efficient quantum state tomography with convolutional neural networks". In: *npj Quantum Inf.* 8 (2022), p. 115. arXiv: 2109.13776.

[26] D. Koutný et al. "Neural-network quantum state tomography". In: *Phys. Rev. A* 106 (2022), p. 012409. arXiv: 2206.06736.

[27] Y. Nomura et al. "Restricted Boltzmann machine learning for solving strongly correlated quantum systems". In: *Phys. Rev. B* 96 (2017), p. 205152. arXiv: 1709.06475.

[28] X. Gao and L.-M. Duan. "Efficient representation of quantum many-body states with deep neural networks". In: *Nat. Commun.* 8 (2017), p. 662. arXiv: 1701.05039.

[29] G. Carleo and M. Troyer. "Solving the quantum many-body problem with artificial neural networks". In: *Science* 355.6325 (2017), pp. 602–606. arXiv: 1606.02318.

[30] M. Hibat-Allah et al. "Recurrent neural network wave functions". In: *Phys. Rev. Res.* 2 (2020), p. 023358. arXiv: 2002.02973.

[31] D. Fitzek et al. *RydbergGPT*. 2024. arXiv: 2405.21052.

[32] J. Carrasquilla and R. G. Melko. "Machine learning phases of matter". In: *Nat. Phys.* 13.5 (2017), pp. 431–434. arXiv: 1605.01735.

[33] E. P. L. van Nieuwenburg, Y.-H. Liu, and S. D. Huber. "Learning phase transitions by confusion". In: *Nat. Phys.* 13.5 (2017), pp. 435–439. arXiv: 1610.02048.

[34] M. Hibat-Allah, R. G. Melko, and J. Carrasquilla. "Investigating topological order using recurrent neural networks". In: *Phys. Rev. B* 108 (2023), p. 075152. arXiv: 2303.11207.

[35] J. Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (2021), pp. 583–589.

[36] R. Guidotti et al. "A Survey of Methods for Explaining Black Box Models". In: *ACM Comput. Surv.* 51.5 (2018), p. 93. arXiv: 1802.01933.

[37] Y. Zhang et al. "A Survey on Neural Network Interpretability". In: *IEEE Trans. Emerg. Top. Comput. Intell.* 5.5 (2021), pp. 726–742. arXiv: 2012.14261.

[38] C. Rudin. "Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead". In: *Nat. Mach. Intell.* 1 (2019), pp. 206–215. arXiv: 1811.10154.

[39] C. Chen et al. "This looks like that: deep learning for interpretable image recognition". In: *Adv. Neural Inf. Process. Syst.* Curran Associates Inc., 2019. arXiv: 1806.10574.

[40] A. Ramesh et al. "Zero-Shot Text-to-Image Generation". In: *Proc. 38th Int. Conf. Mach. Learn.* Vol. 139. PMLR, 2021, pp. 8821–8831. arXiv: 2102.12092.

[41] R. Rombach et al. "High-Resolution Image Synthesis with Latent Diffusion Models". In: *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.* IEEE Computer Society, 2022, pp. 10674–10685. arXiv: 2112.10752.

[42] L. Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155.

[43] N. Carlini et al. "Extracting training data from diffusion models". In: *USENIX Security '23*. USENIX Association, 2023. arXiv: 2301.13188.

[44] M. Nasr et al. *Scalable Extraction of Training Data from (Production) Language Models*. 2023. arXiv: 2311.17035.

[45] C. Dwork. "Differential Privacy". In: *Automata, Languages and Programming*. Vol. 4052. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 1–12.

[46] C. Dwork and A. Roth. "The Algorithmic Foundations of Differential Privacy". In: *Found. Trends Theor. Comput. Sci.* 9.3–4 (2014), pp. 211–407.

[47] C. Dwork. "Calibrating Noise to Sensitivity in Private Data Analysis". In: *Theory of Cryptography*. Vol. 3876. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 265–284.

[48] M. Abadi et al. "Deep Learning with Differential Privacy". In: *Proc. ACM Conf. Comput. Commun. Secur.* ACM, 2016, pp. 308–318. arXiv: 1607.00133.

[49] R. Orús. "A practical introduction to tensor networks: Matrix product states and projected entangled pair states". In: *Ann. Phys.* 349 (2014), pp. 117–158. arXiv: 1306.2164.

[50] J. C. Bridgeman and C. T. Chubb. "Hand-waving and interpretive dance: an introductory course on tensor networks". In: *J. Phys. A: Math. Theor.* 50.22 (2017), p. 223001. arXiv: 1603.03039.

[51] J. I. Cirac et al. "Matrix product states and projected entangled pair states: Concepts, symmetries, theorems". In: *Rev. Mod. Phys.* 93 (2021), p. 045003. arXiv: 2011.12127.

[52] Y.-Y. Shi, L.-M. Duan, and G. Vidal. "Classical simulation of quantum many-body systems with a tree tensor network". In: *Phys. Rev. A* 74.2 (2006), p. 022320. arXiv: 0511070 [quant-ph].

[53] L. Tagliacozzo, G. Evenbly, and G. Vidal. "Simulation of two-dimensional quantum systems using a tree tensor network that exploits the entropic area law". In: *Phys. Rev. B* 80 (2009), p. 235127. arXiv: 0903.5017.

[54] V. Murg et al. "Simulating strongly correlated quantum systems with tree tensor networks". In: *Phys. Rev. B* 82 (2010), p. 205105. arXiv: 1006.3095.

[55] F. Pan, K. Chen, and P. Zhang. "Solving the Sampling Problem of the Sycamore Quantum Circuits". In: *Phys. Rev. Lett.* 129 (2022), p. 090502. arXiv: 2111.03011.

[56] C. Oh et al. "Tensor network algorithm for simulating experimental Gaussian boson sampling". In: *Nat. Phys.* 20 (2024), pp. 1461–1468. arXiv: 2306.03709.

[57] J. Tindall et al. "Efficient Tensor Network Simulation of IBM's Eagle Kicked Ising Experiment". In: *PRX Quantum* 5 (2024), p. 010308. arXiv: 2306.14887.

[58] E. Stoudenmire and D. J. Schwab. "Supervised Learning with Tensor Networks". In: *Adv. Neural Inf. Process. Syst.* Vol. 29. Curran Associates, Inc., 2016, pp. 4799–4807. arXiv: 1605.05775.

[59] A. Novikov, M. Trofimov, and I. V. Oseledets. "Exponential machines". In: *Bull. Pol. Acad. Sci. Tech. Sci.* 66.No 6 (Special Section on Deep Learning: Theory and Practice) (2018), pp. 789–797. arXiv: 1605.03795.

[60] D. Pérez-García et al. "Matrix Product State Representations". In: *Quantum Inf. Comput.* 7.5 (2007), pp. 401–430. arXiv: 0608197 [quant-ph].

[61] I. Oseledets. "Tensor-train decomposition". In: *SIAM J. Sci. Comput.* 33.5 (2011), 2295 –2317.

[62] D. Liu et al. "Machine learning by unitary tensor network of hierarchical tree structure". In: *New J. Phys.* 21.7 (2019), p. 073059. arXiv: 1710.04833.

[63] S. Cheng et al. "Tree tensor networks for generative modeling". In: *Phys. Rev. B* 99 (2019), p. 155131. arXiv: 1901.02217.

[64] J. A. Reyes and E. M. Stoudenmire. "Multi-scale tensor network architecture for machine learning". In: *Mach. Learn. Sci. Technol.* 2.3 (2021), p. 035036. arXiv: 2001.08286.

[65] T. Vieijra, L. Vanderstraeten, and F. Verstraete. *Generative modeling with projected entangled-pair states*. 2022. arXiv: 2202.08177.

[66] F. Barratt, J. Dborin, and L. Wright. *Improvements to Gradient Descent Methods for Quantum Tensor Network Machine Learning*. 2022. arXiv: 2203.03366.

[67] A. Mata Ali et al. *Efficient Finite Initialization for Tensorized Neural Networks*. 2024. arXiv: 2309.06577.

[68] J. Wang et al. *Anomaly detection with tensor networks*. 2020. arXiv: 2006.02516.

[69] J. Tangpanitanon et al. "Explainable natural language processing with matrix product states". In: *New J. Phys.* 24.5 (2022), p. 053032. arXiv: 2112.08628.

[70] B. Aizpurua, S. Palmer, and R. Orus. *Tensor Networks for Explainable Machine Learning in Cybersecurity*. 2024. arXiv: 2401.00867.

[71] A. Mossi, B. Žunkovic, and K. Flouris. *A Matrix Product State Model for Simultaneous Classification and Generation*. 2024. arXiv: 2406.17441.

[72] A. Kehoe et al. "Defence against adversarial attacks using classical and quantum-enhanced Boltzmann machines". In: *Mach. Learn. Sci. Technol.* 2.4 (2021), p. 045006. arXiv: 2012.11619.

[73] A. Pozas-Kerstjens et al. "Privacy-preserving machine learning with tensor networks". In: *Quantum* 8 (2024), p. 1425. arXiv: 2202.12319.

[74] A. Novikov et al. "Tensorizing Neural Networks". In: *Adv. Neural Inf. Process. Syst.* Vol. 28. Curran Associates, Inc., 2015. arXiv: 1509.06569.

[75] V. Lebedev et al. *Speeding-up convolutional neural networks using fine-tuned CP decomposition*. 2014. arXiv: 1412.6553.

[76] Z.-F. Gao et al. "Compressing deep neural networks by matrix product operators". In: *Phys. Rev. Res.* 2 (2020), p. 023300. arXiv: 1904.06194.

[77] C. Hawkins, X. Liu, and Z. Zhang. "Towards compact neural networks via end-to-end training: A Bayesian tensor approach with automatic rank determination". In: *SIAM J. Math. Data Sci.* 4.1 (2022), pp. 46–71. arXiv: 2010.08689.

[78] S. Singh, S. S. Jahromi, and R. Orus. *Tensor network compressibility of convolutional models*. 2024. arXiv: 2403.14379.

[79] X. Ma et al. "A Tensorized Transformer for Language Modeling". In: *Adv. Neural Inf. Process. Syst.* Vol. 32. Curran Associates, Inc., 2019. arXiv: 1906.09777.

[80] V. Chekalina et al. "Efficient GPT Model Pre-training using Tensor Train Matrix Representation". In: *Proc. 37th Pac. Asia Conf. Lang. Inf. Comput.* Association for Computational Linguistics, 2023, pp. 600–608. arXiv: 2306.02697 [cs.AI].

[81] B. Aizpurua et al. *Quantum Large Language Models via Tensor Network Disentanglers*. 2024. arXiv: 2410.17397.

[82] A. Tomut et al. *CompactifAI: Extreme Compression of Large Language Models using Quantum-Inspired Tensor Networks*. 2024. arXiv: 2401.14109 [cs.CL].

[83] J. Miller. *TorchMPS*. https://github.com/jemisjoky/TorchMPS. 2019.

[84] M. Usvyatsov, R. Ballester-Ripoll, and K. Schindler. "tntorch: Tensor network learning with PyTorch". In: *J. Mach. Learn. Res.* 23.208 (2022), pp. 1–6. arXiv: 2206.11128.

[85] J. Kossaifi et al. "TensorLy: Tensor Learning in Python". In: *J. Mach. Learn. Res.* 20.26 (2019), pp. 1–6. arXiv: 1610.09555.

[86] C. Roberts et al. *TensorNetwork: A library for physics and machine learning*. 2019. arXiv: 1905.01330.

[87] J. Gray. "quimb: A Python package for quantum information and many-body calculations". In: *J. Open Source Softw.* 3.29 (2018), p. 819.

[88] A. Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Adv. Neural Inf. Process. Syst.* Vol. 32. Curran Associates, Inc., 2019. arXiv: 1912.01703.

[89] Y. Hur et al. "Generative modeling via tensor train sketching". In: *App. Comput. Harmon. Anal.* 67 (2023), p. 101575. arXiv: 2202.11788.

[90] I. Oseledets and E. E. Tyrtyshnikov. "TT-cross approximation for multidimensional arrays". In: *Linear Algebra Appl.* 432.1 (2010), pp. 70–88.

[91] D. V. Savostyanov. "Quasioptimality of maximum-volume cross interpolation of tensors". In: *Linear Algebra Appl.* 458 (2014), pp. 217–244. arXiv: 1305.1818.

[92] L. Deng. "The MNIST database of handwritten digit images for machine learning research". In: *IEEE Signal Process. Mag.* 29.6 (2012), pp. 141–142.

[93] R. Ardila et al. "Common Voice: A Massively-Multilingual Speech Corpus". In: *Proc. 12th Lang. Resour. Eval. Conf.* European Language Resources Association, 2020, pp. 4218–4222. arXiv: 1912.06670 [cs.CL].

[94] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[95] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[96] D. P. Kingma and M. Welling. "Auto-Encoding Variational Bayes". In: *2nd Int. Conf. Learn. Represent.* 2014. arXiv: 1312.6114 [stat.ML].

[97] I. J. Goodfellow et al. "Generative Adversarial Nets". In: *Adv. Neural Inf. Process. Syst.* Vol. 27. Curran Associates, Inc., Dec. 2014. arXiv: 1406.2661 [stat.ML].

[98] L. Dinh, J. Sohl-Dickstein, and S. Bengio. *Density estimation using Real NVP*. 2017. arXiv: 1605.08803 [cs.LG].

[99] D. P. Kingma and P. Dhariwal. "Glow: Generative Flow with Invertible 1x1 Convolutions". In: *Adv. Neural Inf. Process. Syst.* Vol. 31. Curran Associates, Inc., 2018. arXiv: 1807.03039 [stat.ML].

[100] J. Sohl-Dickstein et al. "Deep Unsupervised Learning using Nonequilibrium Thermodynamics". In: *Proc. 32nd Int. Conf. Mach. Learn.* PMLR, 2015, pp. 2256–2265. arXiv: 1503.03585 [cs.LG].

[101] J. Ho, A. Jain, and P. Abbeel. "Denoising Diffusion Probabilistic Models". In: *Adv. Neural Inf. Process. Syst.* Vol. 33. Curran Associates, Inc., 2020, pp. 6840–6851. arXiv: 2006.11239 [cs.LG].

[102] J. MacQueen. "Some Methods for Classification and Analysis of Multivariate Observations". In: *Proc. Fifth Berkeley Sympos. Math. Statist. Probab.* Vol. 1. University of California Press, 1967, pp. 281–298.

[103] A. Y. Ng, M. I. Jordan, and Y. Weiss. "On Spectral Clustering: Analysis and an Algorithm". In: *Adv. Neural Inf. Process. Syst.* Vol. 14. Neural Information Processing Systems Foundation, 2001.

[104] L. van der Maaten and G. Hinton. "Visualizing Data using t-SNE". In: *J. Mach. Learn. Res.* 9.86 (2008), pp. 2579–2605.

[105] L. McInnes et al. "UMAP: Uniform Manifold Approximation and Projection". In: *J. Open Source Softw.* 3.29 (2018), p. 861.

[106] C. Doersch, A. Gupta, and A. A. Efros. "Unsupervised Visual Representation Learning by Context Prediction". In: *Proc. IEEE Int. Conf. Comput. Vis.* 2015, pp. 1422–1430. arXiv: 1505.05192 [cs.CV].

[107] X. Liu et al. "Self-supervised Learning: Generative or Contrastive". In: *IEEE Trans. Knowl. Data Eng.* (2021), pp. 1–1.

[108] C. Cortes and V. Vapnik. "Support-Vector Networks". In: *Machine Learning* 20.3 (1995), pp. 273–297.

[109] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain". In: *Psychological Review* 65.6 (Nov. 1958), pp. 386–408.

[110] A. L. Maas, A. Y. Hannun, and A. Y. Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. 30th Int. Conf. Mach. Learn.* Vol. 30. 1. 2013, p. 3.

[111] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units*. 2015. arXiv: 1511.07289 [cs.LG].

[112] D. Hendrycks and K. Gimpel. *Gaussian Error Linear Units (GELUs)*. 2016. arXiv: 1606.08415 [cs.LG].

[113] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[114] K. Hornik, M. Stinchcombe, and H. White. "Multilayer Feedforward Networks Are Universal Approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366.

[115] M. Fredrikson et al. "Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing". In: *USENIX Security '14*. Aug. 2014, pp. 17–32.

[116] G. Ateniese et al. "Hacking Smart Machines with Smarter Ones: How to Extract Meaningful Data from Machine Learning Classifiers". In: *Int. J. Secur. Netw.* 10.3 (2015), pp. 137–150.

[117] R. Shokri et al. "Membership Inference Attacks Against Machine Learning Models". In: *2017 IEEE Symposium on Security and Privacy*. 2017, pp. 3–18. arXiv: 1610.05820.

[118] B. Balle, G. Cherubin, and J. Hayes. "Reconstructing Training Data with Informed Adversaries". In: *CoRR* abs/2201.04845 (2022). arXiv: 2201.04845 [cs.CR].

[119] N. Haim et al. "Reconstructing Training Data from Trained Neural Networks". In: *Adv. Neural Inf. Process. Syst.* Vol. 35. 2022, pp. 22911–22924.

[120] Y. Oz et al. "Reconstructing Training Data from Real-World Models Trained with Transfer Learning". In: *CoRR* abs/2407.15845 (2024). arXiv: 2407.15845 [cs.LG].

[121] P. C. Mahawaga Arachchige et al. "Local Differential Privacy for Deep Learning". In: *IEEE Internet Things J.* 7.7 (2020), pp. 5827–5842. arXiv: 1908.02997.

[122] E. Bagdasaryan and V. Shmatikov. "Differential Privacy Has Disparate Impact on Model Accuracy". In: *Adv. Neural Inf. Process. Syst.* Vol. 32. Curran Associates Inc., 2019. arXiv: 1905.12101.

[123] D. Yu et al. "Large Scale Private Learning via Low-rank Reparametrization". In: *Proc. 38th Int. Conf. Mach. Learn.* Vol. 139. PMLR, 2021, pp. 12208–12218. arXiv: 2106.09352.

[124] E. J. Hu et al. "LoRA: Low-Rank Adaptation of Large Language Models". In: *Int. Conf. Learn. Represent.* 2022. arXiv: 2106.09685.

[125] Y. Huang et al. *Privacy-preserving Learning via Deep Net Pruning*. 2020. arXiv: 2003.01876.

[126] K. Adamczewski and M. Park. *Differential Privacy Meets Neural Network Pruning*. 2023. arXiv: 2303.04612.

[127] M. Jagielski, J. Ullman, and A. Oprea. "Auditing Differentially Private Machine Learning: How Private is Private SGD?" In: *Adv. Neural Inf. Process. Syst.* Vol. 33. Curran Associates, Inc., 2020, pp. 22205–22216. arXiv: 2006.07709.

[128] N. D. Sidiropoulos et al. "Tensor Decomposition for Signal Processing and Machine Learning". In: *IEEE Trans. Signal Process.* 65.13 (2017), pp. 3551–3582. arXiv: 1607.01668.

[129] S. Dolgov and D. Savostyanov. "Parallel cross interpolation for high-precision calculation of high-dimensional integrals". In: *Comput. Phys. Commun.* 246 (2020), p. 106869. arXiv: 1903.11554.

[130] T. G. Kolda and B. W. Bader. "Tensor decompositions and applications". In: *SIAM Rev.* 51.3 (2009), pp. 455–500.

[131] F. Verstraete et al. "Criticality, the area law, and the computational power of projected entangled pair states". In: *Phys. Rev. Lett.* 96 (2006), p. 220601. arXiv: 0601075 [quant-ph].

[132] C. Damm, M. Holzer, and P. McKenzie. "The complexity of tensor calculus". In: *Comput. Complex.* 11 (2002), pp. 54–89.

[133] B. Pirvu et al. "Matrix product operator representations". In: *New J. Phys.* 12.2 (2010), p. 025012. arXiv: 0804.3976.

[134] J. Håstad. "Tensor Rank is NP-Complete". In: *J. Algorithms* 11.4 (1990), pp. 644–654.

[135] C. J. Hillar and L.-H. Lim. "Most Tensor Problems Are NP-Hard". In: *J. ACM* 60.6 (2013).

[136] Q. Zhao et al. *Tensor Ring Decomposition*. 2016. arXiv: 1606.05535 [cs.NA].

[137] K. Batselier. *The Trouble with Tensor Ring Decompositions*. 2018. arXiv: 1811.03813 [cs.NA].

[138] J. J. García-Ripoll. "Quantum-inspired algorithms for multivariate analysis: from interpolation to partial differential equations". In: *Quantum* 5 (2021), p. 431. arXiv: 1909.06619.

[139] M. K. Ritter et al. "Quantics Tensor Cross Interpolation for High-Resolution Parsimonious Representations of Multivariate Functions". In: *Phys. Rev. Lett.* 132 (2024), p. 056501. arXiv: 2303.11819.

[140] G. Vidal. "Efficient Classical Simulation of Slightly Entangled Quantum Computations". In: *Phys. Rev. Lett.* 91 (2003), p. 147902. arXiv: 0301063 [quant-ph].

[141] A. Acuaviva et al. "The minimal canonical form of a tensor network". In: *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*. 2023, pp. 328–362. arXiv: 2209.14358.

[142] Y. Núñez Fernández et al. *Learning tensor networks with tensor cross interpolation: new algorithms and libraries*. 2024. arXiv: 2407.02454.

[143] G. H. Golub and C. F. Van Loan. *Matrix Computations*. 4th ed. Johns Hopkins University Press, 2013.

[144] C. Eckart and G. Young. "The Approximation of One Matrix by Another of Lower Rank". In: *Psychometrika* 1.3 (1936), pp. 211–218.

[145] W. B. Johnson and J. Lindenstrauss. "Extensions of Lipschitz mappings into Hilbert space". In: *Conference on Modern Analysis and Probability*. Vol. 26. Contemporary Mathematics. American Mathematical Society, 1984, pp. 189–206.

[146] N. Halko, P.-G. Martinsson, and J. A. Tropp. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions". In: *SIAM Rev.* 53.2 (2011), pp. 217–288. arXiv: 0909.4061.

[147] M. W. Mahoney. "Randomized Algorithms for Matrices and Data". In: *Found. Trends Mach. Learn.* 3.2 (2011), pp. 123–224. arXiv: 1104.5557.

[148] D. P. Woodruff. "Sketching as a tool for numerical linear algebra". In: *Found. Trends Theor. Comput. Sci.* 10.1–2 (2014), pp. 1–157. arXiv: 1411.4357.

[149] T. Shi, M. Ruth, and A. Townsend. "Parallel algorithms for computing the tensor-train decomposition". In: *SIAM J. Sci. Comput.* 45 (2023), pp. C101–C130. arXiv: 2111.10448.

[150] Y. Sun et al. *Tensor Random Projection for Low Memory Dimension Reduction*. 2021. arXiv: 2105.00105.

[151] Y. Ren et al. "High-dimensional density estimation with tensorizing flow". In: *Res. Math. Sci.* 10 (2023), p. 30. arXiv: 2212.00759.

[152] X. Tang et al. "Generative Modeling via Tree Tensor Network States". In: *Res. Math. Sci.* 10 (2023), p. 19. arXiv: 2209.01341.

[153] Y. Peng et al. *Generative Modeling via Hierarchical Tensor Sketching*. 2023. arXiv: 2304.05305.

[154] E. E. Tyrtyshnikov. "Incomplete cross approximation in the mosaic-skeleton method". In: *Computing* 64 (2000), pp. 367–380.

[155] S. A. Goreinov, E. E. Tyrtyshnikov, and N. L. Zamarashkin. "A theory of pseudoskeleton approximations". In: *Linear Algebra Appl.* 261.1 (1997), pp. 1–21.

[156] M. W. Mahoney and P. Drineas. "CUR matrix decompositions for improved data analysis". In: *Proc. Natl. Acad. Sci. U.S.A.* 106.3 (2009), pp. 697–702.

[157] S. A. Goreinov and E. E. Tyrtyshnikov. "The maximum-volume concept in approximation by low-rank matrices". In: *Structured Matrices in Mathematics, Computer Science, and Engineering I*. Vol. 280. Contemporary Mathematics. American Mathematical Society, 2001, pp. 47–52.

[158] J. Tindall, M. Stoudenmire, and R. Levy. *Compressing multivariate functions with tree tensor networks*. 2024. arXiv: 2410.03572.

[159] S. R. White. "Density matrix formulation for quantum renormalization groups". In: *Phys. Rev. Lett.* 69 (1992), pp. 2863–2866.

[160] U. Schollwöck. "The density-matrix renormalization group". In: *Rev. Mod. Phys.* 77 (2005), pp. 259–315. arXiv: cond-mat/0409292.

[161] A. Meiburg et al. "Generative Learning of Continuous Data by Tensor Networks". In: *SciPost Phys.* 18 (2025), p. 096. arXiv: 2310.20498 [quant-ph].

[162] I. Glasser, N. Pancotti, and J. I. Cirac. "From probabilistic graphical models to generalized tensor networks for supervised learning". In: *IEEE Access* 8 (2020), pp. 68169–68182. arXiv: 1806.05964.

[163] Z.-Y. Han et al. "Unsupervised Generative Modeling Using Matrix Product States". In: *Phys. Rev. X* 8.3 (2018), p. 031012. arXiv: 1709.01662.

[164] J. Miller, G. Rabusseau, and J. Terilla. "Tensor Networks for Probabilistic Sequence Modeling". In: *Proc. 24th Int. Conf. Artif. Intell. Statist.* Vol. 130. PMLR, 2021, pp. 3079–3087. arXiv: 2003.01039.

[165] T. Vieijra et al. "Direct sampling of projected entangled-pair states". In: *Phys. Rev. B* 104 (23 2021), p. 235141. arXiv: 2109.07356.

[166] M. Hashemizadeh et al. *Adaptive Learning of Tensor Network Structures*. 2021. arXiv: 2008.05437 [cs.LG].

[167] J. Zeng et al. "Bayesian Tensor Network Structure Search and Its Application to Tensor Completion". In: *Neural Netw.* 175 (2024), p. 106290.

[168] A. Acharya et al. *Qubit seriation: Improving data-model alignment using spectral ordering*. 2022. arXiv: 2211.15978.

[169] C. Li et al. "Permutation Search of Tensor Network Structures via Local Sampling". In: *Proc. 39th Int. Conf. Mach. Learn.* Vol. 162. 2022, pp. 13106–13124. arXiv: 2206.06597.

[170] M. Kodryan, D. Kropotov, and D. Vetrov. "MARS: Masked Automatic Ranks Selection in Tensor Decompositions". In: *Proc. 26th Int. Conf. Artif. Intell. Statist.* Vol. 206. PMLR, 2023, pp. 3718–3732. arXiv: 2006.10859.

[171] S. Holtz, T. Rohwedder, and R. Schneider. "The Alternating Linear Scheme for Tensor Optimization in the Tensor Train Format". In: *SIAM J. Sci. Comput.* 34.2 (2012), A683–A713.

[172] J. R. Pareja Monturiol, A. Pozas-Kerstjens, and D. Pérez-García. *Tensorization of neural networks for improved privacy and interpretability*. 2025. arXiv: 2501.06300 [math.NA].

[173] F. Mezzadri. "How to generate random matrices from the classical compact groups". In: *Not. Am. Math. Soc.* 54.5 (2007), pp. 592–604. arXiv: 0609050 [math-ph].

[174] J. R. Pareja Monturiol, D. Pérez-García, and A. Pozas-Kerstjens. "TensorKrowch: Smooth integration of tensor networks in machine learning". In: *Quantum* 8 (2024). https://github.com/joserapa98/tensorkrowch, p. 1364. arXiv: 2306.08595.

[175] J. R. Pareja Monturiol, A. Pozas-Kerstjens, and D. Pérez-García. "Computational Appendix of *Tensorization of neural networks for improved privacy and interpretability*". In: *Zenodo* 15639937 (2025).

[176] D. Savostyanov and I. Oseledets. "Fast adaptive interpolation of multi-dimensional arrays in tensor train format". In: *The 2011 International Workshop on Multidimensional (nD) Systems*. 2011, pp. 1–8.

[177] Apple. *Differential Privacy Overview*. Accessed: 2021-12-02. 2021. URL: https://www.apple.com/privacy/docs/Differential_Privacy_Overview.pdf.

[178] Google. *How we're Helping Developers with Differential Privacy*. Accessed: 2021-12-02. 2021. URL: https://developers.googleblog.com/2021/01/how-were-helping-developers-with-differential-privacy.html.

[179] S. L. Warner. "Randomized Response: A Survey Technique for Eliminating Evasive Answer Bias". In: *J. Am. Stat. Assoc.* 60.309 (1965), pp. 63–69.

[180] N. Phan, X. Wu, and D. Dou. "Preserving Differential Privacy in Convolutional Deep Belief Networks". In: *Mach. Learn.* 106.9 (2017), pp. 1681–1704. arXiv: 1706.08839 [cs.LG].

[181] C. Collberg et al. "Toward Digital Asset Protection". In: *IEEE Intell. Syst.* 26.6 (2011), pp. 8–13.

[182] J. Haegeman et al. "Geometry of Matrix Product States: Metric, Parallel Transport, and Curvature". In: *J. Math. Phys.* 55.2 (Feb. 2014), p. 021902. arXiv: 1210.7710.

[183] K. B. Marathe and G. Martucci. *The Mathematical Foundations of Gauge Theories*. North-Holland, 1992.

[184] Global.health. *A Data Science Initiative*. Accessed: 2021-03-22. 2021. URL: https://global.health.

[185] A. Pozas-Kerstjens, S. Hernández-Santana, and D. Pérez-García. "Computational Appendix of *Privacy-preserving machine learning with tensor networks*". In: *Zenodo* 6302728 (2022).

[186] F. Tramèr et al. "Stealing Machine Learning Models via Prediction APIs". In: *USENIX Security '16*. USENIX Association, 2016, pp. 601–618. arXiv: 1609.02943.

[187] M. Jagielski et al. "High Accuracy and High Fidelity Extraction of Neural Networks". In: *USENIX Security '20*. USENIX Association, Aug. 2020, pp. 1345–1362. arXiv: 1909.01838 [cs.LG].

[188] A. Molnar et al. "Normal projected entangled pair states generating the same state". In: *New J. Phys.* 20.11 (Nov. 2018), p. 113017. arXiv: 1804.04964.

[189] N. Kargas and N. D. Sidiropoulos. "Supervised Learning and Canonical Decomposition of Multivariate Functions". In: *IEEE Trans. Signal Process.* 69 (2021), pp. 1097–1107.

[190] F. Wesel and K. Batselier. "Large-Scale Learning with Fourier Features and Tensor Decompositions". In: *Adv. Neural Inf. Process. Syst.* Vol. 34. Curran Associates, Inc., 2021, pp. 17543–17554. arXiv: 2109.01545 [cs.LG].

[191] N. D. Sidiropoulos and R. Bro. "On the uniqueness of multilinear decomposition of N-way arrays". In: *J. Chemometrics* 14.3 (May 2000), pp. 229–239.

[192] B. Barak et al. "On the (im)possibility of obfuscating programs". In: *J. ACM* 59.2 (May 2012).

[193] J. Haegeman et al. "Order Parameter for Symmetry-Protected Phases in One Dimension". In: *Phys. Rev. Lett.* 109 (2012), p. 050402. arXiv: 1201.4174.

[194] D. Pérez-García et al. "String Order and Symmetries in Quantum Spin Lattices". In: *Phys. Rev. Lett.* 100 (2008), p. 167202. arXiv: 0802.0447.

[195] M. Sanz et al. "Matrix product states: Symmetries and two-body Hamiltonians". In: *Phys. Rev. A* 79 (2009), p. 042308. arXiv: 0901.2223.

[196] F. Pollmann et al. "Entanglement spectrum of a topological phase in one dimension". In: *Phys. Rev. B* 81 (2010), p. 064439. arXiv: 0910.1811.

[197] X. Chen, Z.-C. Gu, and X.-G. Wen. "Classification of gapped symmetric phases in one-dimensional spin systems". In: *Phys. Rev. B* 83 (2011), p. 035107. arXiv: 1008.3745.

[198] N. Schuch, D. Pérez-García, and I. Cirac. "Classifying quantum phases using matrix product states and projected entangled pair states". In: *Phys. Rev. B* 84 (2011), p. 165139. arXiv: 1010.3732.

[199] F. Pollmann et al. "Symmetry protection of topological phases in one-dimensional quantum spin systems". In: *Phys. Rev. B* 85 (2012), p. 075125. arXiv: 0909.4059.

[200] X. Chen et al. "Symmetry protected topological orders and the group cohomology of their symmetry group". In: *Phys. Rev. B* 87 (2013), p. 155114. arXiv: 1106.4772.

[201] Y. Ogata. "Classification of symmetry protected topological phases in quantum spin chains". In: *Curr. Dev. Math.* 2020.1 (2022), pp. 41–104. arXiv: 2110.04671.

[202] F. Pollmann and A. M. Turner. "Detection of symmetry-protected topological phases in one dimension". In: *Phys. Rev. B* 86 (2012), p. 125441. arXiv: 1204.0704.

[203] I. Affleck et al. "Rigorous results on valence-bond ground states in antiferromagnets". In: *Phys. Rev. Lett.* 59 (1987), pp. 799–802.

[204] N. Schuch et al. "Computational Complexity of Projected Entangled Pair States". In: *Phys. Rev. Lett.* 98 (2007), p. 140506. arXiv: quant-ph/0611050.

[205] J. Haferkamp et al. "Contracting projected entangled pair states is average-case hard". In: *Phys. Rev. Res.* 2 (2020), p. 013010. arXiv: 1810.00738.

[206] K. He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *Proc. IEEE Int. Conf. Comput. Vis.* 2015, pp. 1026–1034. arXiv: 1502.01852.

[207] Z. Allen-Zhu, Y. Li, and Z. Song. "A Convergence Theory for Deep Learning via Over-Parameterization". In: *Proc. 36th Int. Conf. Mach. Learn.* Vol. 97. PMLR, 2019, pp. 242–252. arXiv: 1811.03962.

[208] D. G. A. Smith and J. Gray. "opt_einsum - A Python package for optimizing contraction order for einsum-like expressions". In: *J. Open Source Softw.* 3.26 (2018), p. 753.

[209] V. Zauner-Stauber et al. "Variational optimization algorithms for uniform matrix product states". In: *Phys. Rev. B* 97 (2018), p. 045145. arXiv: 1701.07035.

[210] J. Gray and S. Kourtis. "Hyper-optimized tensor network contraction". In: *Quantum* 5 (2021), p. 410. arXiv: 2002.01935.

[211] Y. Liu and M. K. Ng. "Deep neural network compression by Tucker decomposition with nonlinear response". In: *Knowl.-Based Syst.* 241 (2022), p. 108171.