

Chapter 4: Tensors Structures and Modelling Applications

Chapter one, accompanying source code, has introduced why tensors are essential for high dimensional dataset representations enabling multi-way analysis that is not possible when high dimensional data are vectorised or metricised. Representing tensors in full dimensionality suffers from the dimensionality curse. This chapter will start with tensor decomposition methods that factorise the high-dimensional tensor space into the most dominating components in each dimension, providing low-rank compression. These components can be represented in memory using sparse factor matrices and low-order core tensors, called factors or blocks. The most common tensor decomposition algorithms that will be covered are CP and Tucker.

Then, the second section will introduce graphical tensor notations as graph data structures. This section explains how a network of tensors can be contracted into one tensor using graphical notation and Einstein indices. Then the third section will introduce tensor networks as decomposition algorithms that represent large-scale tensors hierarchically using lower-rank core tensors. We can work with networks of tensors, such as each tensor representing a multi-way dataset; particular indices/features in a tensor connect to other indices/features in another multi-way dataset tensor representation as summation indices enabling contraction or left as free indices in the final tensor shape. The final tensor shape is the dataset a machine learning or deep learning algorithm should use, identifying some indices/features as predictors and others as target/outcome variables. We can also work with tensor networks that are a factorisation of a given large tensor. The following section then introduces two tensor Network decomposition approaches. Tensor Train (TT) decomposition reduces the complexity of the tensor decomposition algorithms presented in chapter three when working on large tensors. TT uses permutations of tensor dimensions doing sequential multilinear products over latent tensor cores. Tensor Ring (TR) decomposition optimises the operations of TT by doing circular multilinear products over a sequence of low-dimensional tensor cores. TT and TR reduce a large-scale optimisation problem to tractable more minor problems (lower-order smaller core tensors of order at most 3), similar to how ALS reduced a non-convex problem to convex subproblems. TT and TR methods build on Hierarchical Tucker methods in which lower-order rank core tensors form a tree representing the large tensor.

The chapter will then conclude with applications. The first section applies tensor completion by using tensor decomposition methods, and the other introduces tensor regression methods.

Then, a final section will introduce artificial neural networks (ANN) and how they can benefit from tensor higher-order representations of data and the tensor decomposition algorithms.

4.1 Tensor Decomposition Methods

The previous section explains the projection to a lower dimension as the Multilinear subspace learning method. In this section, learning the projection matrices that effectively capture the dataset structure and optimising the representation of the approximate learned structure to the high dimensional one is explained. Low-rank matrix factorization presented in chapter two does not enable the recovery of the underlying components. In contrast, some tensor decompositions, such as CP decomposition, provide an essentially unique decomposition suitable for many problems, such as Blind Source Separation (BSS) and others. The general framework requires the following steps:

1. The learning paradigm to follow: supervised, unsupervised, semi-supervised, or active learning.
2. The multilinear projection to employ: VVP, TVP, or TTP.
3. The criterion to be optimized: such as maximising scatter measures as practised in LSL.
4. The order of tensor representation: 2-D, 3-D, or 4-D are the most natural choices, but some datasets or application requirements require higher orders.
5. The additional model/constraints to be imposed: PCA maximises the variance captured as the optimisation criteria while keeping orthogonality between PCs as the constraints (uncorrelated). ICA assumes independent sources mixed in the received dataset. Other constraints, optimisation objectives, and assumptions can be extended to each mode in the higher dimension or even in the interactions between modes.

4.1.1 CANDECOMP/PARAFAC (CP)

As explained in chapter two, the SVD of a matrix is computed by $X = USV^T = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \dots + \sigma_r u_r v_r^T$. This can be expressed as the summation of outer products of the vectors of the most dominating columns in U, and most dominating rows in V, in the order of the singular values σ from the highest σ_1 to the lowest given rank σ_r that is diagonalized in S as illustrated in Figure 1.

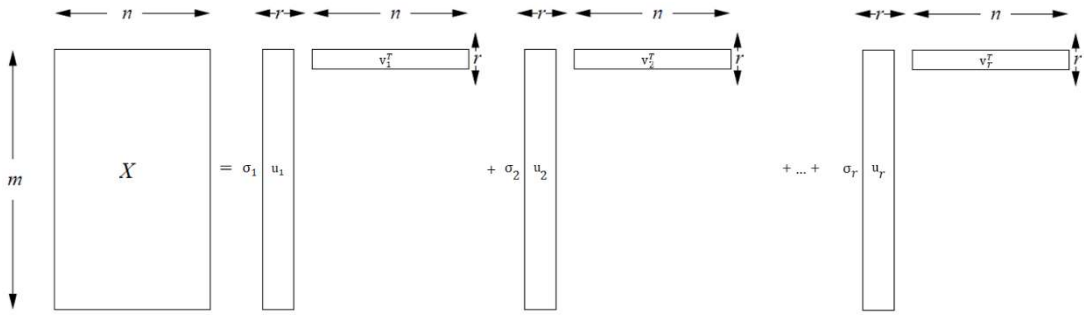


Figure 1: 2-way SVD generalised to enable the higher order SVD

This enables the approximate reconstruction of matrix χ as $\hat{\chi}$ from its dominant components:

$$\hat{\chi} = \sum_{j=1}^r \sigma_j u_j v_j^T$$

This means matrix $\hat{\chi}$ is a linear combination of the rank one vectors of the two dimensions of the given matrix χ . The higher-order representation of this process is illustrated in Figure 2 for 3 dimensions. The most dominating components in each dimension are represented as rank-one vectors along with each mode, the multiplication with the singular values is absorbed in the factor vectors, and the summation of their outer products forms the reconstruction.

Hitchcock, in 1927 proposed the idea of the polyadic form of a tensor, i.e., expressing a tensor as the sum of a finite number of rank-one tensors as the Canonical decomposition (CANDECOMP). This was later redefined in the psychometrics community as parallel factors (PARAFAC), which is an approximation technique suitable for matrices that have a collection of the same number of columns and different rows. PARAFAC relaxes some of the CP constraints and fits the covariance matrices' cross-products to the original data. Both have the acronym CP. CP factorises a tensor χ using the outer product \circ of rank-1 vectors across each mode, for R as the tensor rank. For a tensor of order 3 this is defined as follows:

$$\chi \in \mathbb{R}^{I \times J \times K} = \sum_{r=1}^R a_r \circ b_r \circ c_r \approx \sum_{r=1}^R a_{ir} b_{jr} c_{kr} \text{ for all } a_r \in \mathbb{R}^I, b_r \in \mathbb{R}^J, \text{ and } c_r \in \mathbb{R}^K.$$

This is defined as the Khatri-Rao product \odot for the matrix-form on each mode as follows: $\chi_{(1)} \approx A(C \odot B)^T$, $\chi_{(2)} \approx B(C \odot A)^T$, $\chi_{(3)} \approx C(B \odot A)^T$. This is concisely expressed as $\chi \approx \hat{\chi} = [\lambda; A, B, C] = \sum_{r=1}^R \lambda_r a_r \circ b_r \circ c_r$. This three-way model is expressed as the frontal slices of χ . It is often useful to assume that the columns of A , B , and C are normalized to length one with the weights absorbed into the vector $\lambda \in \mathbb{R}^R$ that can be added to the equation as below.

The N dimensions generalisation is defined as $\chi \in \mathbb{R}^{I_1 \times \dots \times I_N} \approx [\lambda; A^{(1)}, A^{(2)}, \dots, A^{(N)}] = \sum_{r=1}^R \lambda_r a_r^{(1)} \circ a_r^{(2)} \circ \dots \circ a_r^{(N)} = \Lambda \times_1 A^{(1)} \times_2 A^{(2)} \dots \times_N A^{(N)}$, where $\Lambda \in \mathbb{R}^{r \times \dots \times r}$ is a diagonal core tensor such that $\lambda_r = \Lambda_{r,r,\dots,r}$.

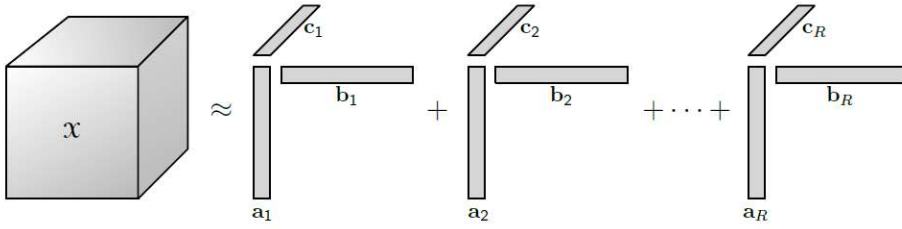


Figure 2: The CANDECOMP/PARAFAC decomposition of a third-order tensor.

Just like the matrix rank is defined as the fewest number of linearly independent columns, the tensor rank should be defined the same, but this will be NP-hard to identify the rank computationally. Iteratively, the tensor rank is computed as the fewest number of rank-one tensors that generate the approximated tensor as their sum with the smallest error $\|\chi - \hat{\chi}\|^2$, i.e. the fewest number of components in the exact CP decomposition with $R = \text{rank}(\chi)$. Finding the tensor rank is an NP-hard problem, and methods like Alternating Least Squares (ALS) are used. The method is based on fixing A and B to solve C, then fixing A & C to solve B, then fixing B and C to solve A, and repeating until convergence, which is defined as the case when the error is not significantly decreasing. ALS reduces a non-convex optimisation problem to convex subproblems.

Repeat Until Convergence:

$$\min_A \sum_{ijk} \left(x_{ijk} - \sum_l a_{il} b_{jl} c_{kl} \right)^2 = \min_A \|X_{(1)} - A(C \odot B)'\|_F^2, \text{ such that } (C \odot B) = \sum_{i=1}^r c_i \otimes b_i$$

$$\min_B \sum_{ijk} \left(x_{ijk} - \sum_l a_{il} b_{jl} c_{kl} \right)^2 = \min_B \|X_{(1)} - B(C \odot A)'\|_F^2, \text{ such that } (C \odot A) = \sum_{i=1}^r c_i \otimes a_i$$

$$\min_C \sum_{ijk} \left(x_{ijk} - \sum_l a_{il} b_{jl} c_{kl} \right)^2 = \min_C \|X_{(1)} - C(B \odot A)'\|_F^2, \text{ such that } (B \odot A) = \sum_{i=1}^r b_i \otimes a_i$$

A sample algorithm is defined as follows for a 4th-Order Tensor (Ji et al., 2019), where χ_{mx} is mode x vectorisation of tensor Y:

Input:

The 4th-order tensor $\chi \in \mathbb{R}^{I \times J \times K \times L}$

Output:

Factor matrices A,B,C,D and the core tensor Λ

1. Initialize A,B,C,D and CP rank R, where $R \leq \min\{IJ, JK, IK\}$;
2. **while** the iteration threshold does not reach or the algorithm has not converged **do**
 - a. $A = \chi_{m1}[(D \odot_R C \odot_R B)^T]$;
 - b. Normalize column vectors of A to unit vector;
 - c. $B = \chi_{m2}[(D \odot_R C \odot_R A)^T]$;
 - d. Normalize column vectors of B to unit vector;
 - e. $C = \chi_{m3}[(D \odot_R B \odot_R A)^T]$;
 - f. Normalize column vectors of C to unit vector;
 - g. $D = \chi_{m4}[(C \odot_R B \odot_R A)^T]$;
 - h. Normalize column vectors of D to unit vector;
 - i. Save the value of the norms of the R column vectors in the factor matrix C to the core tensor Λ ;
3. end while
4. **return** Factor matrices A, B, C, D and the core tensor Λ

And can be generalised to N order as:

Input: N-order tensor $\chi \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$, tensor rank R

Output: coefficients $\lambda_{n=1}^N$, factor matrices $A^n \in \mathbb{R}^{I_n \times R}_{n=1}^N$

Initialize: randomly initialize $A^n_{n=1}$

repeat

for $n = 1, \dots, N$ do

$$T_n = A^{(1)T} A^{(1)} * \dots * A^{(n-1)T} A^{(n-1)} * A^{(n+1)T} A^{(n+1)} * \dots * A^{(N)T} A^{(N)}$$

$$A^{(n)} = \chi_{(n)}(A^{(N)} \odot \dots \odot A^{(n+1)} \odot A^{(n-1)} \odot \dots \odot A^{(1)}) T_n$$

end for

until the convergence criterion is satisfied

Generally, the CP decomposition has a computational complexity of $O(Nlr)$, where N is the dimension, $l \in \mathbb{R}^N$ is a vector of the shape of the tensor, and r is the rank, such that $r \ll N$.

This is linear to tensor order. For example, the discretization of the 5-variate (columns) dataset over 100 sample points (rows) on each axis would yield $100^5 = 10\,000\,000\,000$ sample points, while a rank-2 CP representation would require only $5 \times 2 \times 100 = 1000$ sample points. It is also worth noting that CP optimization is difficult for high-order tensors, converges slowly, may produce an unstable estimation of its components, and generally, factor matrices can be arbitrarily reordered and scaled.

Example CP applications:

- Time-varying electroencephalographic (EEG) spectrum arranged as a three-dimensional array with modes corresponding to time, frequency, and channel, is compared to space/time ICA and PCA in (Miwakeichi *et al.*, 2004).
- Vowel-sound data where different individuals (mode 1) spoke different vowels (mode 2) and the formant (i.e., the pitch) was measured (mode 3) in (Harshman, 1970).
- Multi-subject fMRI data is analysed with a three-way extension of independent component analysis (ICA) and CP, and the differences in terms of the higher-order statistical properties were identified (Stegeman, 2007).

4.1.2 Tucker Decomposition

The Tucker decomposition is most cited and is considered a higher-order (or multi-way) PCA. It decomposes a tensor into a core tensor (not a diagonal core tensor of weights as in CP decomposition) multiplied by a factor matrix along each mode. This enables capturing the arbitrary interaction of factors among each mode independently from each other (mixed modes). For a given tensor $\chi \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$, it is computed using mode- n multiplication of $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times \dots \times R_N}$ as the core tensor such that the values R_k denotes the rank along the k^{th} mode, and $A^{(k)} \in \mathbb{R}^{I_k \times R_k}$ as the orthogonal factor matrices for each mode k . Given a tensor of order 3, its Tucker decomposition is defined as:

$$\chi \in \mathbb{R}^3 \approx \mathcal{G} \times_1 A \times_2 B \times_3 C$$

It is defined using outer products on the vector level as follows:

$$\chi \approx \sum_{i=1}^{R_1} \sum_{j=1}^{R_2} \sum_{k=1}^{R_3} g_{ijk} a_i \circ b_j \circ c_k$$

The scalar representation is defined as follows:

$$x_{ijk} \approx \sum_{i=1}^{R_1} \sum_{j=1}^{R_2} \sum_{k=1}^{R_3} g_{ijk} a_i b_j c_k$$

And the compact form is $[[\mathcal{G}; A, B, C]]$, where.

The N-Dimensional generalisation is $X \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ as $\chi \approx [[\mathcal{G}; A^{(1)}, A^{(2)}, \dots, A^{(N)}]]$, such that $\hat{\chi}$ is estimated from $\hat{x}_{i_1 i_2 \dots i_N} = \sum_{r_1=1}^{R_1} \dots \sum_{r_N=1}^{R_N} g_{r_1 \dots r_N} a_{i_1 r_1}^{(1)} \circ a_{i_2 r_2}^{(2)} \circ \dots \circ a_{i_N r_N}^{(N)}$.

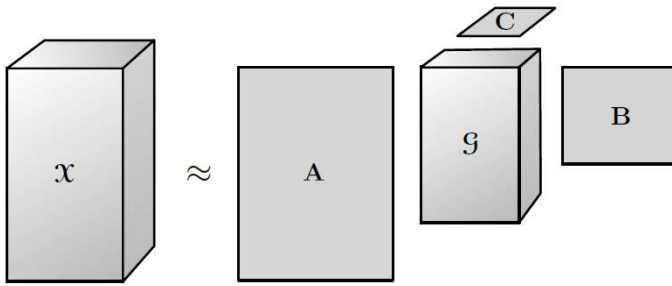


Figure 3: The Tucker decomposition of a third-order tensor.

This can be expressed in matrix form by unfolding χ and G and using the Kronecker product \otimes as follows: $\chi = A^{(n)} \mathcal{G}_{(n)} \cdot (A^{(n+1)} \otimes \dots \otimes A^{(N)} \otimes A^{(1)} \otimes \dots \otimes A^{(n-1)})^T$

The CP decomposition can be considered a special case Tucker decomposition in which the core tensor Λ is super diagonal such that $R_1 = R_2 = \dots = R_N$, containing the weights/coefficient along the diagonal describing an interaction between one mode factor matrix and another mode factor matrix. While in Tucker, we have Multiple Linear Ranks (Tucker Rank: (R_1, R_2, \dots, R_N)) for each mode and the core tensor captures the underlying multi-way structure of tensor data. Standard 2-dimensional PCA is a Tucker 1 decomposition by setting the factor matrices to the Identity matrix I and capturing the variance in mode-1 independent of the other modes. Using ALS, Tucker 2 alternates the I matrix to find other modes' variance. The HOSVD used for tensor compression requires that the factor matrices and the core tensor be all orthogonal. Higher-order Factor Analysis is also considered a special case of the Tucker decomposition.

TensorFaces which will be reviewed in detail in chapter six is an example of higher-order SVD based on the Tucker decomposition. It takes facial images of different people from different angles, lighting, facial expressions, and more modes as required. It is significantly more accurate than PCA face recognition methods and valuable for compression and removing irrelevant effects (Vasilescu and Terzopoulos, 2002).

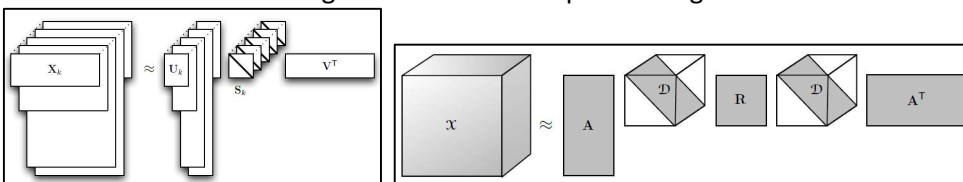
The Tucker decomposition generally has a computational complexity of $O(Nr + r^N)$, where N is the dimension, $\mathbf{l} \in \mathbb{R}^N$ is a vector of the shape of the tensor, and r is the rank. This is exponential to tensor order but is more stable and produces a better approximation than CP decomposition. Tucker decomposition does not produce unique factor matrices and can be rotated along each mode.

In `ch3.ipynb`, Tensorly Python package implementation of CP and Tucker examples are presented. Other Python packages provide tensor decomposition implementation, such as `scikit-tt` (https://github.com/PGelss/scikit_tt), and `HOTTBOX` (<https://github.com/hottbox/hottbox>).

4.1.3 Other Tensor Decomposition Approaches

Other tensor decomposition approaches have been proposed in the literature. These include but are not limited to the following:

- Individual Differences in SCALing (INDSCAL): the method is a special case of 3-way CP suitable for 3-way tensors that are symmetric in 2 modes. This is implemented in scikit-tensor (<https://github.com/mnick/scikit-tensor>).
- CANonical Decomposition with LINear Constraints (CANDELINC): the method imposes linear constraints in one or more of the factor matrices, such as being orthonormal or replaced with one that generates the same orthogonal column space. It is useful for large-scale datasets for compression and regularisation. Its applications include multicollinearity in chemometric datasets.
- PARAFAC2 simultaneously decomposes a collection of matrices, with each having an equal number of columns but a different row size, allowing for distinct factors associated with different frontal slices in the first mode. Example Application: PARAFAC2 handles time shifts in resolving chromatographic data with spectral detection. In this application, the first mode corresponds to elution time, the second mode to wavelength, and the third mode to samples
- DEDICOM is a decomposition into directional components that describe tensor χ as an asymmetric relationship between I objects. Such as I countries, and the I_{ij} is the exports from country i to country j . The method identifies that Latent components in R and groups the I objects accordingly, and A is factor loadings (for example, the interactions of countries in groups) applying scaling and rotations as required to maximise the variance across matrix A . Example Application: (Bader, Harshman and Kolda, 2007) applied their ASALSAN method for computing DEDICOM on email communication graphs over time. In this case, x_{ijk} corresponded to the (scaled) number of email messages sent from person i to person j in month k . This is also implemented in scikit-tensor.
- PARATUCK2, combines CP and Tucker 2 to generalise DEDICOM to capture the interactions between two sets of interacting objects, similar to PLS in the LSL context.
- Nonnegative variants are suitable for datasets in which the interpretation requires nonnegativity for physical or psychological reasons. Tensorly provides several non-negative tensor decomposition algorithms.



(a) (b)

Figure 4; Parafac2 in (a) and DEDICOM in (b)

The important other tensor decomposition formats that the next chapter will build on are:

- The Hierarchical Tucker (HT) decomposition decomposes a tensor hierarchically similar to a binary tree split. In HT decomposition, the core tensor must be less than or equal to the third order and no more than three-factor matrices can be connected to the core tensor. In tensor networks (explained in chapter four), this can be achieved by a hierarchy of nested separation to split a higher order tensor into lower levels in the tree, such that each tensor has three modes only. For example, given a tensor $\chi \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times \dots \times I_N}$, the objective is to define disjoint subsets u, v such that, $u = \{1, \dots, N_0\}$, and $v = \{N_0 + 1, \dots, N\}$, where $t = u \cup v \subset 1, 2, \dots, N$ and $1 \leq N_0 \leq N$. For example, given a fourth-order tensor χ , $N_0 = 2$, such that $u = \{1, 2\}$, $v = \{3, 4\}$, and the HT is defined as $\chi = \sum_{r_{12}=1}^{R_{12}} \sum_{r_{34}=1}^{R_{34}} g_{r_{12}r_{34}}^{(1234)} a_{r_{12}}^{(u)}(x_1, x_2) \circ a_{r_{34}}^{(v)}(x_3, x_4)$, such that $a_{r_{12}}^{(u)}(x_1, x_2) = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} g_{r_1 r_2 r_{12}}^{(12)} a_{r_1}^{(1)} a_{r_2}^{(2)}$ and $a_{r_{34}}^{(v)}(x_3, x_4) = \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} g_{r_3 r_4 r_{34}}^{(34)} a_{r_3}^{(3)} a_{r_4}^{(4)}$.
- The Tree Tensor Network States (TTNS) format extends HT, creating many disjoint subsets.

Detailed algorithms for computing these decompositions and example applications for tensors of different orders can be found in (Cichocki *et al.*, 2016, p. 1).

Figure 5, Figure 6, and Figure 7 show taxonomies of some proposed higher-order PCA algorithms as an unsupervised learning method, higher-order LDA algorithms as a supervised learning method, and ICA, CCA, PLS higher-order MLS some proposed algorithms, illustrating the projection used, the optimisation criteria, the number of modes it applies to, the model name, and the learning model. Chapter four and chapter eight in (Lu, Plataniotis and Venetsanopoulos, 2014) discuss more details about these algorithms and list their original publications for even more details.

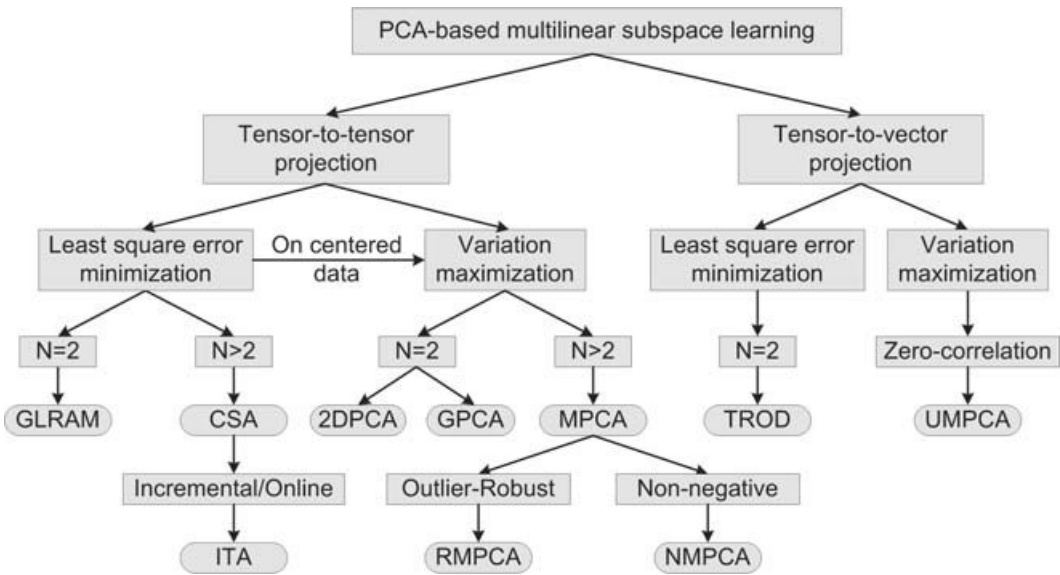


Figure 5: A taxonomy of PCA-based MSL algorithms

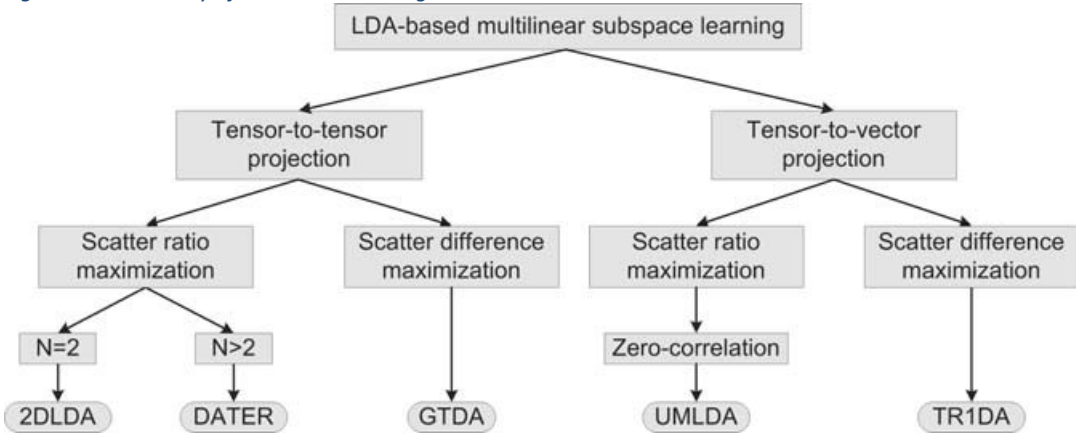


Figure 6: A taxonomy of LDA-based MSL algorithms.

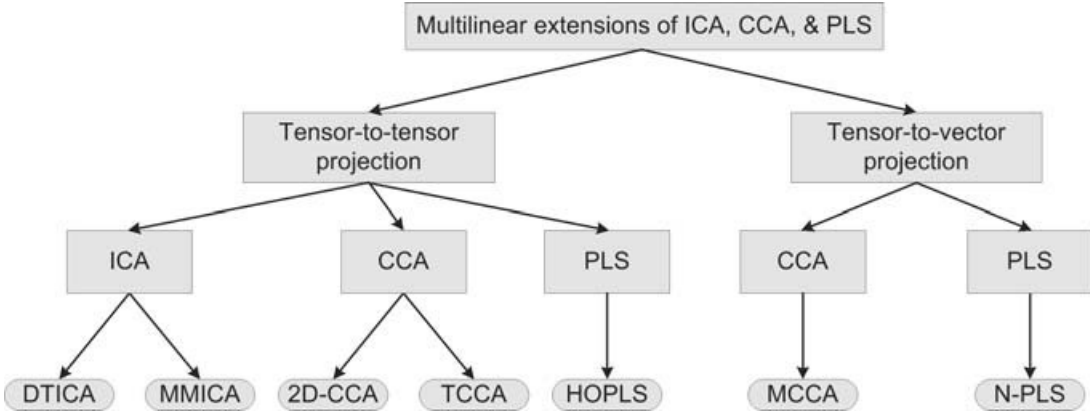


Figure 7: Taxonomy of ICA, CCA, PLS based MSL algorithms.

Python notebook ch3.ipynb, multi-wayExamples.ipynb and tensorisation.ipynb show examples of the tensor decompositions methods discussed in this section.

4.1.4 Summary

All presented methods are very introductory. Most of them are computationally expensive for larger tensors. Researchers adopted the Alternating Least Square (ALS) method using approximations of the rank, multiplying the input matrix by random Gaussian matrices, and by iterative methods until convergence. Good approximations have been achieved with very low-rank factors. Many approaches have been proposed to handle large tensors, such as: decomposing only a compressed representation of the original tensor, sparse tensor multiplications for sparse tensors, decomposing a sample (sub-tensor) of the original tensorial data, parallelizing or distributing the decomposition computation, or using Tensor Networks as will be presented in the next section (Hou, 2017).

4.2 Tensor Graphical Notation

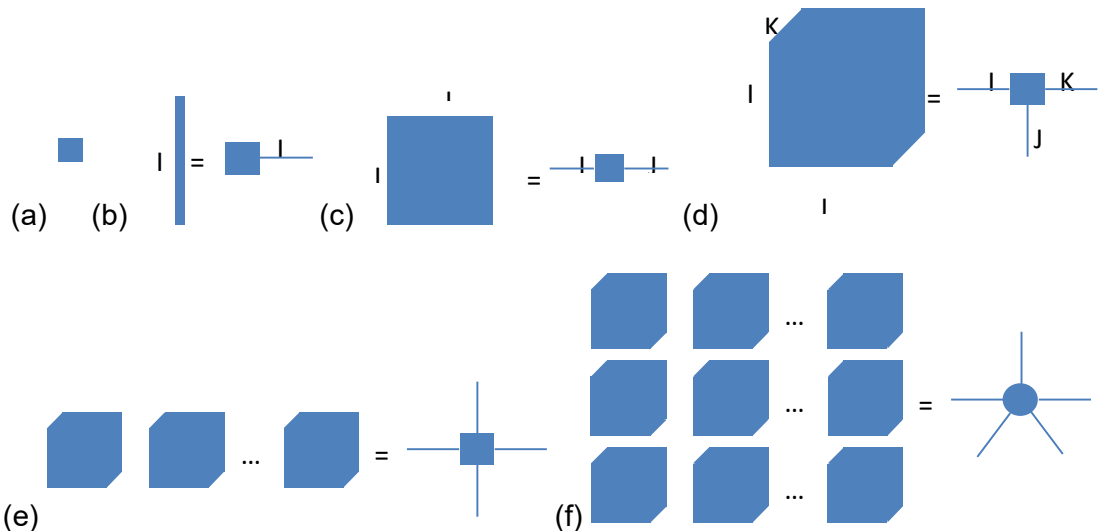


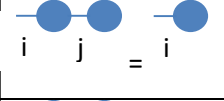
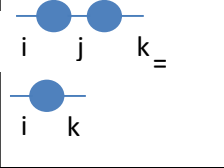


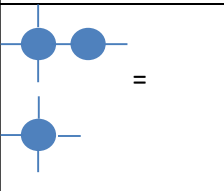
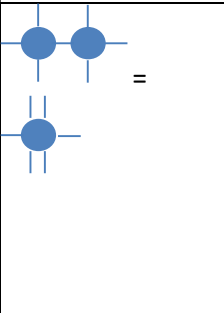
Figure 8: Tensor Network representations, for (a) scalar, (b) vector, (c) matrix, and (d) 3-way tensor, (e) 4-way tensor, (f) 5-way tensor.

The graphical tensor notation uses graph data structures of nodes and edges. The graph nodes describe tensors (can be matrices, vectors or even scalars) as a circle, a square, a dot or a polygon such as a pentagon or a hexagon. Each outgoing edge of the tensor represents the index of a particular dimension, and the order is the number of edges. Figure 8 illustrates some tensor graphical possible representations of tensors up to order 5. This notation enables the graphical illustration of Tensor contractions, from which some are

CHAPTER 4

listed in Table 1, showing their equations, graphical tensor notation, and some explanations. Tensor contraction is denoted as a connection between two nodes. As explained in chapter one, this is the multilinear product operator between two tensors on a specific mode, which corresponds to the summation over the indices of that mode.

Table 1: Some Tensor contractions illustrations using Tensor Networks Notation

	$\sum_j A_{ij} x_j = b_i$	Matrix-vector multiplication
	$\sum_j A_{ij} B_{jk} = C_{ik}$	Matrix-Matrix Multiplication
	$\text{tr}(A) = a$	Connecting two index lines of the same tensor corresponds to a trace. This produces a scalar value. Similarly, any network of all tensor indices connected will result in a scalar. Only unpaired indices/edges will count towards the resulting tensor order.
	$A_{ij} B_{ji} = \text{Tr}[AB] = a$	Transpose of Matrix-Matrix Multiplication
	$\sum_k T_{ijkl} V_{km} = R_{ijml}$	Tensor-Matrix Multiplication
	$\sum_k T_{ijkl} V_{kmno} = R_{ijmno}$	Tensor-Tensor Multiplication with the third mode in the first tensor = first mode of the second tensor, The resulting contracted tensor is shown.

CHAPTER 4

	$\sum_{\alpha_1, \alpha_2, \alpha_3} \mathcal{A}_{\alpha_1}^{s_1} \mathcal{B}_{\alpha_1, \alpha_2}^{s_2} \mathcal{C}_{\alpha_2, \alpha_3}^{s_3} \mathcal{D}_{\alpha_3}^{s_4} = \mathcal{R}_{s_1 s_2 s_3 s_4}$	<p>Tensor-Tensor Contraction shows upper indices not involved in the contraction and lower indices involved in different contractions.</p>
	$CP(\chi_{ijkl}) = \sum_{r=1}^R a_r \circ b_r \circ c_r \circ d_r$	<p>The CP decomposition for a 4th-order tensor</p>
	$\chi \in \mathbb{R}^{R_1 I_1 \times R_2 I_2}$ (a matrix of matrices), which comprises block matrices $X_{r_1, r_2} \in \mathbb{R}^{I_1 \times I_2}$	<p>Hierarchical matrix structures, a 4th-order tensor representation for a block matrix $\chi \in \mathbb{R}^{R_1 I_1 \times R_2 I_2}$ (a matrix of matrices), which comprises block matrices $X_{r_1, r_2} \in \mathbb{R}^{I_1 \times I_2}$</p>
<p>Vector (each entry is a block matrix)</p>		<p>A 5th-order hierarchical tensor</p>
<p>Matrix</p>		<p>A 6th-order hierarchical tensor</p>

Ingoing or outgoing arrows onto index lines illustrate the Einstein notation showing contravariant/up versus covariant/down indices, respectively. The physics community uses

unique tensors' icons to denote different tensor types, such as isometric tensors as triangles and unitary tensors as rectangles. A diagonal line through the middle of the tensor shape illustrates a diagonal tensor. Hyper-edges or multiple indices which are "locked" together can be notated by introducing "Kronecker delta" or "spider" tensors, often notated by small black dots. These tensors can have an arbitrary number of indices, and only diagonal elements are equal to one ("The Tensor Network," n.d.). The last three rows in the table represent hierarchical tensors and their graphical representations (Cichocki et al., 2016, p. 1).

The Python notebook `ch4.ipynb`, uses `tensortrace` application created in a gamemaker studio environment that is available for download from <https://www.tensortrace.com/>, to graphically create four examples of random networks. Then, the tool generates the python code for their contraction using the `ncon` Python function ("Network CONtractor") (Pfeifer et al., 2015).

4.3 Tensor Networks

Chapter three discussed various tensor decomposition algorithms that factorise tensor objects into factor matrices. Tensor Networks (TN) decompose higher-order tensors into sparsely interconnected lower-order core tensors. The lower-order core tensors are the dominant components in the large-scale tensorial dataset. This decomposition enables data approximation that captures the relevant/important/dominating multi-way interactions to compress large-scale data by removing irrelevant information. This decomposition also enables the distributed storage and computation of large-scale tensors. As discussed in chapter two, large-scale high dimensional datasets often contain subspace of much lower dimensionality embedded in the ample high dimensional space. TN decomposition learns this subspace approximating the original high dimensional space in full format and distributes it on lower rank cores in sparse format.

For example, an N -variate function $f(x) = f(x_1, x_2, \dots, x_N)$ can be approximated by a finite sum of products of individual functions, each depending on only one or a very few variables, such as $f(x_1, x_2, \dots, x_N) \approx f^{(1)}(x_1)f^{(2)}(x_2) \dots f^{(N)}(x_N)$. These are coordinate functions, as explained in chapter three. The sparse format reduces the storage requirement from $\prod_{j=1}^N I_j$ to $\sum_{j=1}^N I_j$, which is $\ll I_N$. The sparse format is robust in the presence of noise and missing data flexible enough to incorporate various constraints as required. TN extends the 2-way (matrix) Component Analysis (2-way CA) methods to multi-way component analysis (MWCA), capturing the relations between the different modes while growing linearly with the dataset size. This decomposition can be performed in tensorial datasets or

after tensorising vector or matrix form datasets. All tensor operations after that can be performed on the core tensors. TN formats enable emerging optimisation algorithms such as random coordinate descent (RCD) schemes, sub-gradient methods, alternating direction method of multipliers (ADMM), and proximal gradient descent methods.

There are many algorithms for Tensor Networks. This section will discuss Matrix Product State / Tensor Train and Tensor Rings. Other methods include Matrix Product Operator (MPO), Tree Tensor Network / Hierarchical Tucker, Projected Entangled Pair States (PEPS) and Multi-scale Entanglement Renormalization Ansatz (MERA).

4.3.1 The Tensor Train Decomposition

The Tensor Train (TT) and Tensor Chain (TC) decomposition are special cases of the Hierarchical Tucker Decomposition introduced in chapter three. In these methods, each core tensor is chained in series and aligned by having the same dimension, and all the factor matrices are unit matrices as leaf nodes. This decomposition transforms a large problem into several tractable small-scale problems. TC connects the last tensor with the first, making all tensors of the same dimension, while TT has its first and last tensors of one less dimension than the intermediate tensors of order 3. The physics communities refer to TC decomposition as the Matrix Product State (MPS) decomposition with periodic boundary conditions (PBC), and the TT decomposition as the MPS decomposition with the Open Boundary Conditions (Oseledets, 2011).



Figure 9: Tensor Train decomposition showing first and last core tensors of order-3 and internal tensors of order 4.

Given an Nth-order tensor $\chi \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times \dots \times I_N}$, the TT decomposition is:

$$\chi = \mathcal{A}_1 \times_{3,1} \mathcal{A}_2 \dots \times_N \mathcal{A}_N$$

Such that each entry in χ is:

$$\chi_{i_1, \dots, i_N} = \sum_{r_1}^{R_1} \dots \sum_{r_N}^{R_N} \mathcal{A}_1(r_0, i_1, r_1) \mathcal{A}_2(r_1, i_2, r_2) \dots \mathcal{A}_N(r_{N-1}, i_N, r_N)$$

Where $\mathcal{A}_n \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$, $R_0 = R_N = 1$; $n = 1, 2, \dots, N$, such that \mathcal{A}_1 and \mathcal{A}_N are of lesser rank than the internal core tensors. The mode ordering of the core tensors is important, and their permutations should be optimised. $\mathcal{A}_1 \times_{3,1} \mathcal{A}_2$ means mode-3 from \mathcal{A}_1 multiplied by mode-1 from \mathcal{A}_2 and similar notation is used elsewhere. The approximately reconstructed

tensor from its TT decomposition is $\hat{\chi}$ with each entry is defined in terms of entries in \mathcal{A} as follows:

$$\hat{\chi}_{i_1, i_2, \dots, i_N} = \sum_{r_1, r_2, \dots, r_{N-1}}^{R_1, R_2, \dots, R_N} a_{1, i_1, r_1}^1 a_{r_1, i_2, r_2}^2 a_{r_2, i_3, r_3}^3 \dots a_{r_{N-2}, i_{N-1}, r_{N-1}}^{N-1} a_{r_{N-1}, i_N, 1}^N$$

Similar to Tucker decomposition, the TT rank is defined for each mode as (R_1, R_2, \dots, R_N) , where $R_n = \text{rank}(\chi_{mcn})$, where m means matricization, c means canonical, and n means mode-n. $\chi_{mcn} = \text{mat}(\chi)_{cn}$, such as mat is the mode-n canonical matricisation of the tensor, which extracts n dimensions from the original tensor as the first dimension of the resulting matrix and the remaining (N-n) dimensions as the second dimension. This notation is used in the following equations. The TT rank increases in proportion to the dimension of the original data tensor χ .

Expressing large tensors in their TT decomposition enables tensor operations on the smaller core tensors, reducing the number of parameters to estimate. For example, Given two tensors expressed in their TT core tensors:

$$\begin{aligned} \chi &\in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N} = \chi_1 \times_{3,1} \chi_2 \times_{3,1} \dots \times_{3,1} \chi_N && \text{with } \chi_n \in \mathbb{R}^{R_{n-1} \times I_n \times R_n} && \text{and} \\ \mathcal{Y} &\in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N} = \mathcal{Y}_1 \times_{3,1} \mathcal{Y}_2 \times_{3,1} \dots \times_{3,1} \mathcal{Y}_N && \text{with } \mathcal{Y}_n \in \mathbb{R}^{Q_{n-1} \times I_n \times Q_n} \end{aligned}$$

For $n = 1, 2, \dots, N$ and R_n are the tensor's χ TT rank, and Q_n are the tensor's \mathcal{Y} TT rank.

- **Their Hadamard product** is $\mathcal{Z} = \chi \odot \mathcal{Y} = \mathcal{Z}_1 \times_{3,1} \mathcal{Z}_2 \times_{3,1} \dots \times_{3,1} \mathcal{Z}_N$, where each core tensor $\mathcal{Z}_n \in \mathbb{R}^{R_{n-1} Q_{n-1} \times I_n \times R_n Q_n} = \chi_n \otimes \mathcal{Y}_n$ for $n=1, 2 \dots N$.
- **Their sum** is $\mathcal{Z} = \chi + \mathcal{Y}$, such that $\mathcal{Z}_n = \begin{bmatrix} \chi_n & 0 \\ 0 & \mathcal{Y}_n \end{bmatrix}$, with first and last core tensors are defined as $\mathcal{Z}_1 = [\chi_n \ \mathcal{Y}_n]$ and $\mathcal{Z}_n = \begin{bmatrix} \chi_n \\ \mathcal{Y}_n \end{bmatrix}$. The TT rank of \mathcal{Z} equals the sum of the TT rank of χ and \mathcal{Y} : $(R_1 + Q_1, R_2 + Q_2, \dots, R_n + Q_n)$
- **Their scalar/dot/quantitative product** is $\mathcal{Z} = \mathcal{X} \cdot \mathcal{Y}$, $(\mathcal{Z}_n)_{m1} = \mathcal{X}_{n-1}(\mathcal{Y}_n)_{m1}$ by using cumulative array variable $a_1 = 0$, and subsequent values $a_n = \mathcal{X}_n \times_{1,2} (\mathcal{Y}_n \times_{1m} a_{n-1}) \in \mathbb{R}^{R_n \times Q_n}$, for $n = 2, \dots, N$, such that the last value in the array a_N will be equal to $\mathcal{X} \cdot \mathcal{Y}$ (Oseledets, 2011).

Various methods solve the TT decomposition, including:

1. SVD-based TT algorithm (TT-SVD) performs mode-n matricisation on the input tensor and then performs HOSVD decomposition to compute a minimum possible compression rank.
2. Algorithms based on low-rank matrix decomposition(LRMD) are similar to TT-SVD but simplify the SVD decomposition using matrix cross-approximation or CR decomposition.
3. Restricted Tucker-1 decomposition(RT1D), converting the original input tensor into a 3rd-order tensor and then performing Tucker-1 and Tucker-2 decomposition.

4. A generalised alternating least squares (ALS) algorithm and modified ALS (MALS) algorithm facilitate the self-adaptation of ranks by using SVDs or employing a greedy algorithm.

The TT decomposition has a computational complexity of $O(Nlr^2)$, where N is the dimension, $l \in \mathbb{R}^N$ is a vector of the tensor's shape, and r is the rank. The number of parameters is linear to the tensor order. The increased rank increases the computational complexity, and methods such as TT Truncation are used to approximate the solution using a smaller rank. TT Truncation performs the N th canonical matricisation of the core tensor and performs a low-rank matrix approximation (SVD and QR). Several modified tensor representations based on TT have been proposed in the literature. These propositions include the quantised tensor-train format (QTT), the block tensor-train format (BTT), and the cyclic tensor-train (CTT). TT representation has been applied to large-scale problems in numerical analysis, such as the optimisation of the Rayleigh quotient, e.g., density matrix renormalisation group (DMRG), and the approximate solution of linear systems, e.g., alternating minimal energy (AME).

TT/MPS are the most popular solutions for tensor networks in 1D. As seen above, the factors are linearly connected in 1D. TT and HT do not allow cycles, but TC allows cycles. Various layered tensor networks have been introduced in the literature enabling analysis on a 2D lattice or deeper layers to reduce the TT rank by increasing the number of core tensors but with more minor ranks. The 3rd-order core tensors of TT were replaced by 5th-order core tensors in the Projected Entangled Pair States (PEPS) method that connects higher-order tensors to represent a physical state in a two-dimensional network. Also, TT 3rd-order core tensors were replaced by 6th-order core tensors in the Projected Entangled Pair Operators (PEPO) method. The Honey-Comb Lattice (HCL) uses 3rd-order core tensors, and the Multi-scale Entanglement Renormalization Ansatz (MERA) uses 4th-order tensors. The computational complexity increases in proportion to the number of cycles, with MERA, generally having a smaller size and dimension, reducing the number of parameters. The cycles explain correlations between variables. These developments are mainly contributed by the researchers from quantum physics communities studying the interactions of many particle systems (Ji et al., 2019). MERA is implemented in Python at <https://www.tensors.net/mera>, MPS Python implementation can be found at <https://www.tensors.net/mps>, <https://www.tensors.net/mps-vumps>

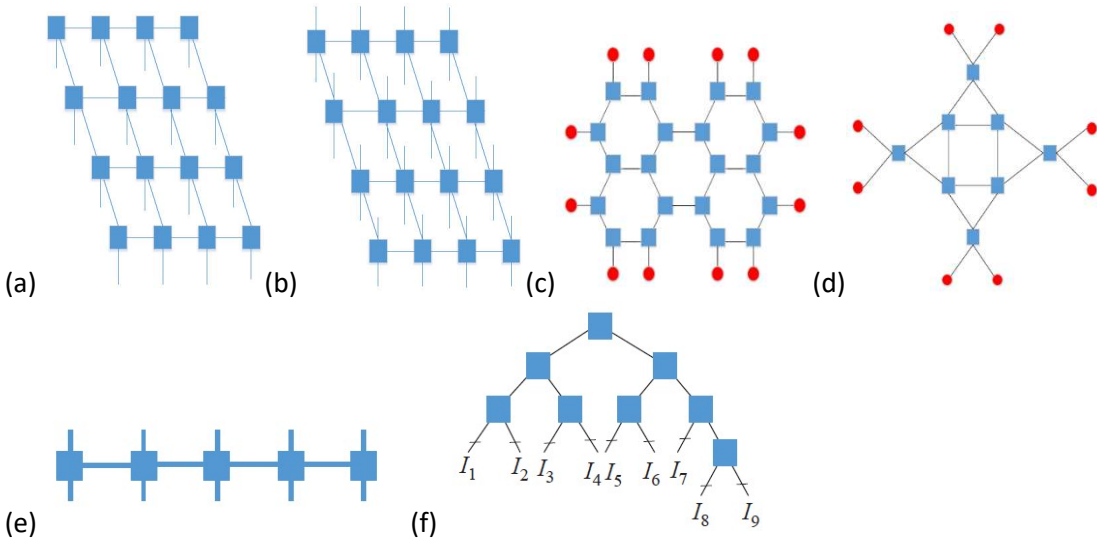


Figure 10: (a) PEPS uses 5th-order core tensors, (b) PEPO uses 6th-order core tensors, (c) HCL uses 3rd-order core tensors, (d) MERA uses 4th-order tensors. The blue rectangle represents core tensors, and the red circle represents factor matrices, (e) MPO of leaf core tensors of order 3, and internal ones of order 4, and (f) TTNS binary tree (Ji et al., 2019).

TNs enable change of topology while keeping the modes and their interactions intact. HT/TT allow sequential contractions of the core tensors to reduce the computational complexity. Algorithms that enable cycles can be modified to eliminate the cycles by contractions and reduce the complexity. Converting tensor networks with cycles to trees reduces the computational complexity as well.

The Python notebook ch4.ipynb shows Tensorly Python package implementation of TT. Tensorly also implemented a regression example in TT format, published at https://github.com/tensorly/Proceedings_IEEE_companion_notebooks/blob/master/tt-compression.ipynb.

The original TT paper has its Python package “tpty” Python package published at <https://github.com/oseledets/tpty>. There are other Python packages as well, such as:

- “scikit_tt” is published at (https://github.com/PGelss/scikit_tt/), and its requirements are demonstrated in ch4.ipynb. The TT class has many interesting features, and the package has many examples for solvers, regression, and Data Analysis methods.
- “ttrecipes” Python package is built on top of tpty. It was built by (Cichocki et al., 2016, p. 1) and published at <https://github.com/rballester/ttrecipes>. They have examples for tensor completion problems, and tensor compression.

4.3.2 Tensor Rings

Like tensor chain (TC), Tensor Ring (TR) decomposition employs a circular dimensional permutation invariance and equivalently makes first and last core tensors of similar shape as central core tensors. This decomposition generalises TT decomposition using linear combinations of its core tensors. TR relaxes the constraints that $R_1 = R_{N+1} = 1$, and no dimension permutation optimisation is required, enabling circular multiplications of the trace of all core tensors.



Figure 11: Tensor Ring Illustration showing that only order-3 core tensors are used and connected circularly.

Given an Nth-order tensor $\chi \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times \dots \times I_N}$ The TR decomposition is:

$$\chi = \mathfrak{R}(\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_N) = \sum_{\alpha_1, \alpha_2, \dots, \alpha_N=1}^{R_1, R_2, \dots, R_N} a_1(\alpha_1, \alpha_2) \circ a_2(\alpha_2, \alpha_3) \circ \dots \circ a_N(\alpha_N, \alpha_1)$$

Where the TR rank is defined for each mode as (R_1, R_2, \dots, R_N) , core tensors $\mathcal{A}_n \in \mathbb{R}^{R_n \times I_n \times R_{n+1}}$, $n = 1, 2, \dots, N$, and last one $\mathcal{A}_N \in \mathbb{R}^{R_N \times I_N \times R_1}$ completes the circular connection with the first one, such that $R_1 = R_{N+1}$. The symbol \circ denotes the outer product of vectors and $a_k(\alpha_k, \alpha_{k+1}) \in \mathbb{R}^{I_k}$ denotes the $(\alpha_k, \alpha_{k+1})^{\text{th}}$ mode-2 fibre of tensor \mathcal{A}_k . \mathcal{A}_1 and \mathcal{A}_2 is multiplied along one dimension indexed by α_2 , which is thus denoted by a connection with the size of that mode (i.e., R_2). This decomposition indicates that the whole tensor can be composed of the sum of rank-1 tensors generated from N vectors taken from each core respectively. The main difference with TT decomposition is the circular dimension permutation invariance such that the TR decomposition of $\chi \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times \dots \times I_N} = \mathfrak{R}(\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_N)$, if we shift the dimensions of χ k positions to the left as $\hat{\chi}^k \in \mathbb{R}^{I_{n_{k+1}} \times \dots \times I_N \times I_1 \times \dots \times I_{n_k}}$, then the TR decomposition of $\hat{\chi}^k$ is defined as $\hat{\chi}^k = \mathfrak{R}(\mathcal{A}_{k+1}, \dots, \mathcal{A}_N, \mathcal{A}_1 \dots \mathcal{A}_k)$.

In TR decompositions, all shifting is equivalent such that:

$$\begin{aligned} \chi &= \mathfrak{R}(\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_N) = \mathfrak{R}(\mathcal{A}_2, \mathcal{A}_3, \dots, \mathcal{A}_N, \mathcal{A}_1) = \dots = \mathfrak{R}(\mathcal{A}_{k+1}, \dots, \mathcal{A}_N, \mathcal{A}_1 \dots \mathcal{A}_k) \\ &= \dots = \mathfrak{R}(\mathcal{A}_N, \mathcal{A}_1 \dots \mathcal{A}_{N-1}) \end{aligned}$$

The approximately reconstructed tensor from its TR decomposition is $\hat{\chi}$ with each entry is defined in terms of entries in A as follows:

$$\hat{\chi}_{i_1, i_2, \dots, i_N} = Tr\{\mathcal{A}_1(i_1)\mathcal{A}_2(i_2) \dots \mathcal{A}_N(i_N)\} = Tr\left\{\prod_{k=1}^N \mathcal{A}_k(i_k)\right\}$$

where the ‘Tr’ is the trace function, $\mathcal{A}_k(i_k)$ denotes the i_k^{th} lateral slice matrix of the latent tensor \mathcal{A}_k , which is of size $r_k \times r_{k+1}$. Any two adjacent core tensors are of equal dimension r_{k+1} on their corresponding mode.

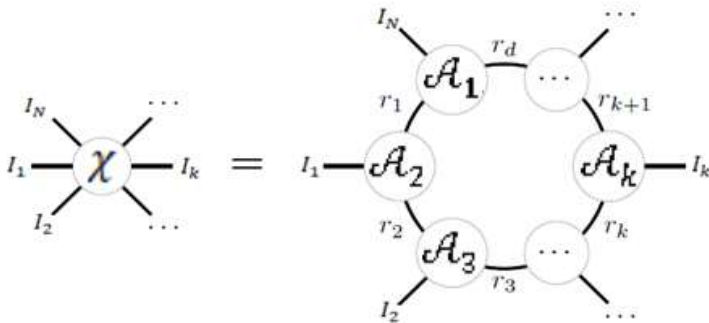


Figure 12: A circular graphical representation of the tensor ring decomposition (Zhao et al., 2016)

Tensors represented in their TR decomposition can perform tensor operations such as those defined for the TT decomposition, including addition, multilinear product, Hadamard product, inner product and Frobenius norm, performed efficiently on each core.

Minimising the ranks and the error solves the TR decomposition:

$$\min_{A_1, \dots, A_N} r$$

$$s. t.: \|\chi - \mathfrak{R}(A_1, A_2, \dots, A_N)\|_F \leq \epsilon \|\chi\|_F$$

Several approaches are used in the literature, including:

- TR-SVD is a sequential SVD-based approach. The algorithm uses subchains of the core tensor multiplications formed by n-unfolding and n-mode matricisation steps, then merging into a single core by multilinear products. These subchains form several TT representations on which to apply the TT_SVD algorithm. This approach does not guarantee optimal TR-ranks but converges fast.
- TR-ALS builds on ALS concepts given a fixed pre-defined rank by optimising one core while fixing the others and alternating between them until some defined convergence stopping criteria. This approach is faster but uses fixed pre-defined ranks.
- ALS-AR also builds on ALS concepts but uses adaptive ranks from initial values that get updated during the iterations to minimise the error.
- BALS is a block-wise ALS algorithm based on truncated SVD on blocks (subchains) formed by merging adjacent core tensors, optimising for one block and fixing the others while updating the ranks. This algorithm is better in finding the most optimal rank without increasing iterations due to the block-wise optimisation iterations of merged adjacent cores rather than one-by-one.

The number of parameters, and hence the computational complexity in TR representation, is $O(Nr^2)$, which is linear to the tensor order N, similar to TT. However, TR is a more

generalised, flexible and powerful representation with usually more minor ranks than TT (Zhao et al., 2016).

The Python notebook `ch4.ipynb` shows “tednet” Python package implements various tensor decomposition algorithms, including CP, Tucker2, TT, and TR. Their code and documentation are published at <https://github.com/tnbar/tednet>.

There are several Matlab implementations for TR decomposition, such as <https://qibinzhao.github.io/> and <https://github.com/oscardickelin/tensor-ring-decomposition>.

4.4 Machine Learning Tensor Decomposition Applications:

This section will present two tensor decomposition machine learning applications. The first one is the tensor completion example, based on CP decomposition. The second application is a class of algorithms to efficiently extend regression approaches to tensorial datasets.

4.4.1 Tensor Completion Application

Tensor completion is an extension to the matrix completion class of problems that aims to interpolate missing values in a dataset from the given values. This is also called inpainting or imputation of missing values. A 3-way association tensor completion application example is presented in (Huang *et al.*, 2021). It is important to understand the dataset and the objective of the analysis that needs to be performed on it. The dataset used in this paper is collected from the HMDD (the Human microRNA Disease Database), which is a database that curates biological lab experiment-supported evidence for human microRNA (miRNA) and disease associations. Auxiliary data are the disease descriptors collected from Medical Subject Heading (MeSH), a comprehensive controlled vocabulary thesaurus about life science, to calculate disease semantic similarity. MicroRNAs (miRNAs) is a small single-stranded non-coding RNA molecule (containing about 22 nucleotides) found in plants, animals and some viruses that functions in RNA silencing and post-transcriptional regulation of gene expression. They play crucial roles in various biological processes associated with human diseases, such as cell growth and division, tissue differentiation, embryonic development and apoptosis, cell cycle regulation, inflammation, and stress response. For example, miR-129, miR-142-5p, and miR-25 are differentially expressed (found in the human cell in different quantities) between paediatric/children's central nervous system neoplasms and normal tissue, indicating their role in oncogenesis. Identifying potential miRNA-disease

associations contributes to understanding the molecular mechanisms of miRNA-related diseases in order to discover new biomarkers and/or develop new therapies. The MiRNA role in disease prominently diverges. For instance, Genetic variants of microRNA (mir-15) may affect miRNAs' expression level, leading to B cell chronic lymphocytic leukaemia. In comparison, circulating miRNAs have the potential to detect breast cancer in the early stage (the quality and quantity changes of circulating miRNAs are associated with the initiation and progression of cancer and can be easily detected by basic molecular biology techniques). The four different types of miRNA-disease associations considered in this paper are illustrated in Figure 13.

Previous work focused on predicting whether a miRNA-disease association exists or not (binary classification/prediction). These methods can be grouped mainly into three categories: (1) methods based on score function, (2) methods based on complex network or graph algorithms, and (3) methods based on machine learning algorithms. Instead of building a binary graph association, this paper presented the dataset in a 3-way structure of miRNA-disease-type triplets as a tensor. It introduced Tensor Decomposition methods to solve the prediction task, such that the type explains the roles of miRNAs in disease development or identification. A miRNA-disease type can be naturally modelled as a binary tensor where every element represents whether the corresponding entry (miRNA, disease, type) exists or not. The authors formulated the multi-type miRNA-disease association prediction as a tensor completion task. Their goal was to complete the tensor for exploring the unobserved triple associations using Tensor Decomposition Methods.

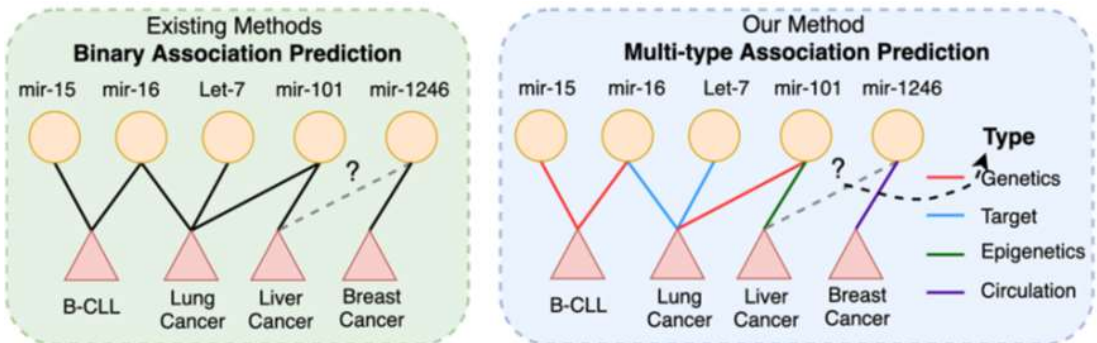


Figure 13: The figure on the left illustrates the binary association graph, and the figure on the right illustrates the MiRNA role in diseases' defined four types

The paper proposed a novel method, Tensor Decomposition with Relational Constraints (TDRC), which incorporates biological features (miRNA-miRNA similarity and disease-disease similarity) as relational constraints to further the existing tensor decomposition methods. TDRC employs the alternating direction method of multipliers (ADMM) framework and resorts to the conjugate gradient (CG) method to avoid computing an inverse matrix in inner iterations of ADMM for lower time complexity.

The authors compared TDRC performance against two previous tensor decomposition methods and a third binary network-based approach:

- 1) CP decomposition without auxiliary information, as explained earlier in this chapter.
- 2) Tensor Factorization Using Auxiliary Information (TFAI) considers incorporating auxiliary information into the CP model via introducing graph Laplacian regularizations.
- 3) NLPMMDA is the latest heterogeneous **Network-based Label Propagation MiRNA MiRNA Disease Association** method that predicts each type of miRNA-disease association by label propagation on the miRNA-miRNA disease-disease similarity network.

Compared with the NLPMMDA, tensor decomposition-based methods significantly improve because these models dissect the data in a higher dimensional perspective through the tensor decomposition and capture complicated ternary relationships of miRNA-disease-type triples. TFAI works slightly worse on HDMM v3.2 than CP, which may be attributed to its weak ability to incorporate the auxiliary information, while TDRC achieved much better performance as it makes sufficient use of the auxiliary information.

The authors concluded that TDRC could produce better performance while being more efficient. The reconstructed tensor's low-rank property may help further improve the performance. The python code of the paper is found at <https://github.com/BioMedicalBigDataMiningLab/TDRC>. The mathematical formulation of the paper is as follows and is illustrated in Figure 14:

- Given a set of miRNAs $\mathcal{E} = \{e_1, e_2, \dots, e_m\}$, a set of diseases $\mathcal{D} = \{d_1, d_2, \dots, d_n\}$ and a set of association types $\mathcal{R} = \{r_1, r_2, \dots, r_t\}$, we can construct a multi-relation bipartite graph \mathcal{G} . A triple (e_i, d_j, r_t) as a link in the graph \mathcal{G} denotes that the association between the miRNA e_i and the disease d_j with the type r_t .
- \mathcal{G} is a binary three-way tensor $\mathcal{X} \in \{0, 1\}^{m \times n \times t}$ with miRNA mode, disease mode and type mode, where each slice is the adjacency matrix with regard to a type of miRNA-disease associations.
- An entry x_{ijt} of the tensor is set to 1 if $(e_i, d_j, r_t) \in \mathcal{G}$. Otherwise, the entries are set to 0.
- The tensor \mathcal{X} is extremely sparse with many unknown entries, and thus it is challenging to reach the goal only by using known links.
 - Hence, the authors considered biological similarities as auxiliary information to tackle the challenge.
- Given the miRNA-disease-type tensor \mathcal{X} , the CP decomposition model can be represented as the following optimization problem (solved by Alternating Least Squares (ALS) method):
 - $\min_{C, P, F} \|\mathcal{X} - \llbracket C, P, F \rrbracket\|^2$
 - Here, $\|\cdot\|$ is the norm of a tensor. $C \in \mathbb{R}^{m \times r}$, $P \in \mathbb{R}^{n \times r}$ and $F \in \mathbb{R}^{t \times r}$ are the factor matrices with respect to the miRNA (the i^{th} miRNA is encoded as a

vector c_i), disease (the j^{th} disease is represented as a vector p_j) and type mode in F , which are usually considered as latent representations for the corresponding modes.

- $\llbracket C, P, F \rrbracket$ is the reconstructed tensor, and its $(i, j, t)^{\text{th}}$ element is calculated by $\sum_l^r c_{il} p_{jl} f_{tl}$
 - where c_{il} , p_{jl} and f_{tl} denote respectively the $(i, l)^{\text{th}}$ element of C , the $(j, l)^{\text{th}}$ element of P and the $(t, l)^{\text{th}}$ element of F . We call r the rank of the approximated tensor $\llbracket C, P, F \rrbracket$.
- In general, r is set much lower than $\min(m, n)$ so that the low-rank property of the latent representations is enforced.
- Then they added the auxiliary information:
 - MeSH descriptors, the hierarchical relationships of diseases, are represented as Directed Acyclic Graphs (DAGs), where nodes represent the diseases and edges represent the relationships between different diseases to calculate the disease semantic similarity.
 - For a disease d , a DAG denoted as $DAG(d) = (N(d), E(d))$ is constructed, where $N(d)$ is the set of all ancestors of d (including itself) and $E(d)$ is the set of links from ancestor disease to their children.
 - The semantic contribution of disease $d_i \in N(d)$ to disease d can be calculated as:

$$C(d, d_i) = \begin{cases} 1 & \text{if } d = d_i \\ \max(\Delta \times C(d, d_j) | d_j \in \text{children of } d_i) & \text{if } d \neq d_i \end{cases}$$

where Δ is the semantic contribution factor, and it was set to 0.5 in the paper. Then the semantic value of disease d is defined as:

- $SV(d) = \sum_{d_i \in N(d)} C(d, d_i)$
- Finally, the semantic similarity between the two diseases d_i and d_j is calculated by:

$$s_{ij}^n = S_{disease}(d_i, d_j) = \frac{\sum_{d \in N(d_i) \cap N(d_j)} (C(d_i, d) + C(d_j, d))}{SV(d_i) + SV(d_j)}$$

- The miRNA functional similarity between two miRNAs e_i, e_j are calculated as follows:

$$s_{ij}^m = S_{miRNA}(e_i, e_j) = \frac{\sum_{d \in \mathcal{D}(e_i)} S_{disease}(d, d_i^*) + \sum_{d \in \mathcal{D}(e_j)} S_{disease}(d, d_j^*)}{|\mathcal{D}(e_i)| + |\mathcal{D}(e_j)|},$$

where $\mathcal{D}(e_i)$ represents the set of diseases that are associated with e_i in at least one association type, $|\mathcal{D}(e_i)|$ is the number of elements in the set $d_i^* = \underset{d_i \in \mathcal{D}(e_i)}{\operatorname{argmax}} S_{disease}(d, d_i)$

- Both similarities are denoted as:

- $S_m \in \mathbb{R}^{m \times m}$ as the miRNA-miRNA functional similarity matrix with s_{ij}^m as its $(i, j)^{\text{th}}$ element
- And $S_n \in \mathbb{R}^{n \times n}$ as the disease-disease semantic similarity matrix with s_{ij}^n as its $(i, j)^{\text{th}}$ element.
- A real-valued function $f(x, y) = xMy^T$ is used to approximate the similarity between two miRNAs (or diseases) for high-quality relational learning, where M is a projection matrix, x and y are the row vectors of C (or P).
- The approximation errors are minimized by:

$$\min_{C,P,M} \alpha \sum_{i=1}^m \sum_{j=1}^m (s_{ij}^m - c_i M_1 c_j^T)^2 + \beta \sum_{i=1}^n \sum_{j=1}^n (s_{ij}^n - p_i M_2 p_j^T)^2$$
- The optimization problem is reformulated in matrix form:

$$\min_{C,P,M} \alpha \|S_m - CM_1 C^T\|_F^2 + \beta \|S_n - PM_2 P^T\|_F^2$$
- Then adding ℓ_2 regularization to the combined optimisation: $\min_{C,P,F,M} \frac{1}{2} \|\mathcal{X} - [C, P, F]\|^2 + \frac{\lambda}{2} (\|M_1\|_F^2 + \|M_2\|_F^2) + \frac{\alpha}{2} \|S_m - CM_1 C^T\|_F^2 + \frac{\beta}{2} \|S_n - PM_2 P^T\|_F^2$

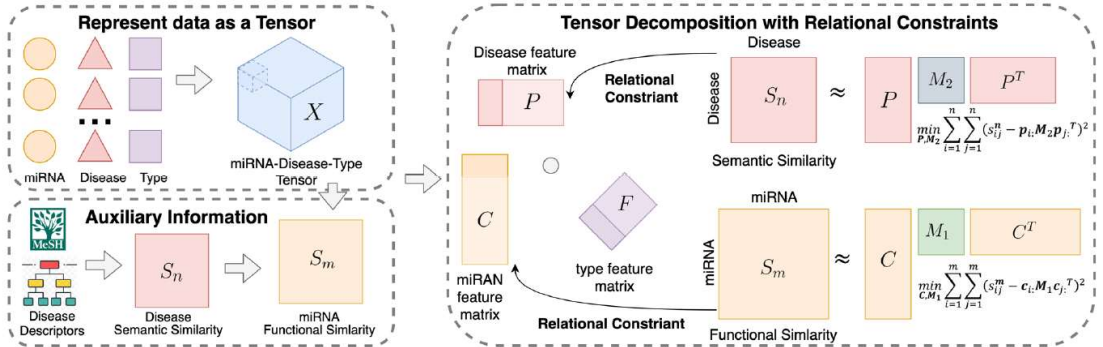


Figure 14: TDRC method to incorporate biological similarities as constraints into the CP model

This application illustrates a method that can be generalised to tensorizing any relational dataset such that there are several types of relationships involved. Example datasets are found in social networks friendships as the two subjects involved, and the type of relationship can be likes, retweets, comments, content sharing, and tagging. A 3-way tensor slice might capture each available relationship by fixing the index of the two subjects involved and varying the relationship index.

4.4.2 Tensor Regression

As chapter one explains, regression is one of the most fundamental supervised machine learning algorithms. it fits a dataset of labels including y (dependent/ target/ outcome/

response) feature as a function of x (independent/ predictor) features. The fitting can be a linear equation such as $y = \epsilon + \sum_{i=0}^N w_i x_i$, such that the w is the estimated regression parameters, including w_0 as the bias and $x_0 = 1$. Chapter two explains that it can also be a non-linear regression using a higher degree polynomial equation or any other non-linear parameters such as exponential, trigonometric, and power functions. A multivariate regression model is when x is multiple predictors and y is multiple responses.

A non-parametric regression model does not assume a specific form of the parameters of the predictors. These algorithms do not need to define a linear or non-linear function to fit the data. They evaluate the mean outcome of the predictor covariates without pre-assumptions and hence misspecification errors. These methods expand to various machine learning algorithms, such as Gaussian Processes (GP), Artificial Neural Networks (ANN), Decision Trees, and Support Vector Regression (SVR). These methods usually require much more sample data than parametric methods (Hou, 2017).

If the dataset is in a matrix form, a regression model will estimate the regression parameters based on the correlation between each feature predictor and the target feature by estimating a parameter value for each predictor. Most classical regression algorithms can be applied to a dataset in tensor after vectorising or matricising the tensor. This is not successful in capturing the multi-way structural information, leading to lower accuracy and higher error. These methods will also use more parameters than tensor regression methods, requiring more storage and computational complexity. This section surveys the tensor regression algorithms that capture the multi-way inherent structures in tensorial data while being more efficient computationally and more accurate in avoiding overfitting. These methods are based on CP and Tucker decompositions of tensors, as presented earlier in this chapter. The tensorial regression models reduce the number of parameters from $O(I^N)$ to $O(NIR)$, where N is the number of modes, I is the number of elements in each mode, and R is the rank, which is usually $R \ll I$. These fewer parameters will be associated with each mode independently. Conversely, the traditional regression models on vectorised or matricised tensorial datasets combine the modes and the estimated extra parameters, which are difficult to interpret.

A simple tensor regression model is defined as: given an N th-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times \dots \times I_N}$, and the output \mathcal{Y} could be a tensor of any order; the regression equation is:

$$\mathcal{Y} = f(\mathcal{X}) + \epsilon$$

The f function for linear regression can be the dot product defined in the generalised linear tensor regression model as:

$$\mathcal{Y} = \langle \mathcal{X}, \mathcal{B} \rangle + \epsilon$$

Such that the dot product of the predictor and β as the coefficient tensor in the same size as the predictor χ , capturing its tensor covariate, and is added to ϵ as the tensor representation error or bias. It can also be non-linear, as defined in chapter two, by using polynomial equations or other non-linear functions such as exponential, logarithmic and others. Prediction or reconstruction/interpolation can occur based on a dataset of M samples as follows:

$$\hat{\chi}_{i_1, i_2, \dots, i_N} = \sum_{k=1}^M \langle \chi_k, \beta_k \rangle$$

Solving $\langle \mathcal{X}, \mathcal{B} \rangle$ by vectorising, both tensors will produce a huge number of parameters. For example, an MRI dataset $\mathcal{X} \in \mathbb{R}^{128 \times 128 \times 128}$ will require 2,097,152 + and five usual covariates parameters to estimate, which is intractable. Using the unsupervised PCA produces the most dominating principal components that are irrelevant to the input, lose the multi-way structural relationship, and are difficult to interpret.

The CP Tensor Regression defines the \mathcal{B} tensor in terms of its rank- R CP decomposition a $[B_1, B_2, \dots, B_N]$ with $B_n = [b_1^{(n)}, \dots, b_R^{(n)}] \in \mathbb{R}^{I_n \times R}$

such that $y = \langle \mathcal{X}, \mathcal{B} \rangle + \epsilon = \left\langle \mathcal{X}, \sum_{r=1}^R b_r^{(1)} \circ b_r^{(2)} \circ \dots \circ b_r^{(N)} \right\rangle + \epsilon$ where y is a scalar output.

This reduces the number of parameters from $O(I^N)$ to the scale of $O(NR)$ while also producing reasonable reconstruction accuracy. For example, the previous MRI example parameters can be reduced to $389 = 5 + 128 \times 3$ for a rank-1 model, and to $1,157 = 5 + 3 \times 128 \times 3$ for a rank-3 model.

As discussed earlier in this chapter, Tucker decomposition is more flexible than CP and accurately captures the multi-way structural relationships in the core tensors. Tucker Tensor Regression defines the \mathcal{B} tensor in terms of its Tucker decomposition as:

$$\sum_{r_1=1}^{R_1} \dots \sum_{r_n=1}^{R_n} \dots g_{r_1 \dots r_n} b_{r_1}^{(1)} \circ b_{r_2}^{(2)} \circ \dots \circ b_{r_n}^{(N)}$$

such that $y = \langle \mathcal{X}, \mathcal{B} \rangle + \epsilon = \left\langle \mathcal{X}, \sum_{r_1=1}^{R_1} \dots \sum_{r_n=1}^{R_n} \dots g_{r_1 \dots r_n} b_{r_1}^{(1)} \circ b_{r_2}^{(2)} \circ \dots \circ b_{r_n}^{(N)} \right\rangle + \epsilon$ where $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times \dots \times R_n}$ with entries $\{g_{r_1 \dots r_n}\}_{r_1=1, \dots, r_n=1}^{R_1, \dots, R_n}$. The factor matrices are defined as $B_n \in \mathbb{R}^{I_n \times R_n}$ along different modes. This reduces the number of parameters from $O(I^N)$ to the scale of $O(Nr + r^N)$, which is higher than the CP regression parameters of $O(NR)$ but more parsimonious modelling of the input data when $R \ll N$. An example application presented in (Li, Zhou and Li, 2013) shows that for a tensorial dataset representing neuroimaging data as a 3D signal $\mathcal{X} \in \mathbb{R}^{16 \times 16 \times 16}$ using a Tucker model with multilinear rank = (2, 2, 5), the number of parameters is 131, while using a 5-component CP regression model yields 230 parameters.

Other tensor regression algorithms are as follows:

Other tensor regression algorithms are as follows:

1) Linear Tensor Regression Methods:

- Hierarchical Tucker Tensor Regression reduce the $O(NR^3 + NIR)$
- Higher-order partial least squares (HOPLS) model a tensor response from tensor predictors by extracting a small number of common latent variables that capture the maximum covariance, followed by a regression step against them. This is done using block Tucker decompositions of both the predictor and outcome tensors. This model is computational prohibitive for large tensors. Incremental higher-order partial least squares (IHOPLS) are suitable for infinite online streams of tensors by incrementally clustering the learned latent variables to summarise previous data in the core tensors and projection matrices.

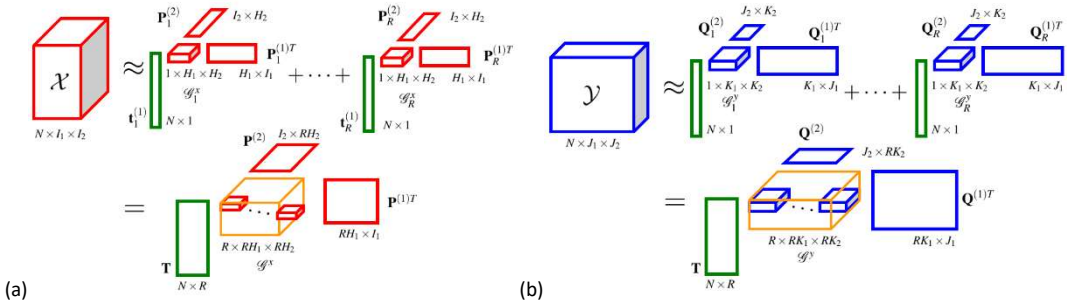


Figure 15: An illustration of high-order partial least squares (HOPLS) for predictor \mathcal{X} of order $M=2$ and $M=2$, and outcome \mathcal{Y} of L order and $L=2$, (a) shows HOPLS decomposes \mathcal{X} into a sum of rank- $(1, H_1, \dots, H_M)$ Tucker blocks, and (b) shows HOPLS decomposes \mathcal{Y} into a sum of rank- $(1, K_1, \dots, K_L)$ Tucker blocks (Hou, 2017).

- Generalised and Penalised Tensor Linear Regression.
- Bayesian Tensor Linear Regression.
- Online Local Gaussian process for tensor-variate regression (OLGP).

2) Non-Linear Tensor Regression Models:

- Incremental higher-order partial least squares (IHOPLS)
- Recursive higher-order partial least squares (RHOPLS) conduct a consecutive blockwise calculation by merging the new data into the previous low-rank approximation of the model, storing only fewer factors than the complete observation.
- Kernel-based multiblock tensor partial least squares (KMTPLS) predict dependent tensor blocks from a set of independent tensor blocks through the extraction of a small number of common and discriminative latent components by fusing the information from multiple tensorial data sources and unifies the single and multiblock tensor regression scenarios into one general model.
- Kernel-based Tensor Partial Least Squares (KTPLS)
- Kernel-based multiblock tensor partial least squares (KMTPLS)
- Kernel-based higher-order linear ridge regression (KHOLRR)
- Tensor Regression Networks: In deep learning (DL), Convolution Neural Networks (CNN) have been used in regression models. Tensor Regression Layer can update the weight matrix in the last Fully Connected (FC) layer with regression coefficient tensor in low-rank Tucker decomposition format as shown in Figure 16.

Simple Linear Tensor Regression Methods are solved by approaches such as Rank Minimisation, Alternating Least Squares (ALS), Greedy Low Rank, and Projected Gradient Descent. Various Tensor Regression applications are found in the literature, such as Tensor on Vector Regression, Vector on Tensor Regression, Tensor on Tensor Regression, and Multiple Tensor-on-Tensor Regression.

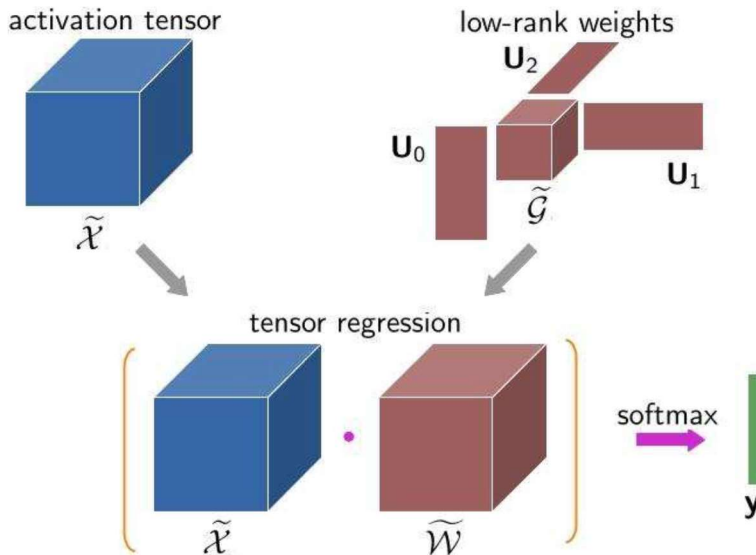


Figure 16: An illustration of tensor regression layer (TRL).

A tensor regression is implemented in the Tensorly package as shown at https://github.com/tensorly/Proceedings_IEEE_companion_notebooks/blob/master/tensor_regression_layer.ipynb. Another implementation is provided by scikit_tt Python package for three different regression algorithms that can be found at https://github.com/PGelss/scikit_tt/blob/master/scikit_tt/data_driven/regression.py. Also, ttrecipes Python package provides a tensor regression and completion implementation that can be found at <https://github.com/rballester/ttrecipes/blob/master/ttrecipes/core/completion.py>. T3F implements tensor completion as well https://t3f.readthedocs.io/en/latest/tutorials/tensor_completion.html. T3F is built on top of Tensorflow providing Tensor Train decomposition for neural networks, with Riemannian optimization. This will be explained next chapter. There is a collection of tensor completion methods maintained in <https://github.com/zhaoxile/reproducible-tensor-completion-state-of-the-art>.

4.5 Deep Learning Applications

The tensor decomposition algorithms presented in this chapter and the previous chapter are data analysis algorithms that reveal complex relationships from the available data without adding any other machine learning algorithms. Tensor completion and regression discussed in the previous chapter can already do prediction and classification. Other machine learning algorithms that benefit from these concepts include support tensor machines, canonical correlation analysis, higher-order partial least squares, generalised eigenvalue decomposition, Riemannian optimisation, and the optimisation of deep neural networks. Tensor decomposition models are analogous to machine learning models. Some examples are as follows:

- TT/MPS is analogous to Hidden Markov Models (HMM).
- HT/TTNS is analogous to Deep Learning Neural Networks and Gaussian Mixture Model (GMM).
- PEPS is analogous to Markov Random Field (MRF) and Conditional Random Field (CRF).
- MERA is analogous to Wavelets and Deep Belief Networks (DBN).
- ALS, DMRG/MALS Algorithms are analogous to Forward-Backward Algorithms and Block Nonlinear Gauss-Seidel Methods (Cichocki et al., 2016, p. 1).

For Artificial Neural Networks (ANN) algorithms, many building blocks contribute together to provide the non-parametric performance that requires no pre-defined assumptions about the function to approximate. ANN uses no rule-based programming, such as if statements or pre-defined equations used in classical machine learning (ML) algorithms, such as those used in chapter two methods. ML relied on developing new algorithms to address new problems, or suitable feature engineering to inject the algorithms with human knowledge about the dataset. Adding neurons and layers and choice of activation functions and error functions are the methods to capture the complexity of the data structures and their correlations. These functions can be linear or non-linear based on the problem requirement. The construction of the network is massively parallel by definition and enables the employment of supercomputers and very-large-scale-integrated (VLSI) technologies. ANN depends on the forward propagation (mapping) of the input from the input layer, to the estimated output on the output layer, through a computational graph (of any layers and neurons per layer) of dot products between weights and input connections (whether fully connected or convoluted). The estimated output is compared to the expected output to calculate the loss at the output layer. The loss is used to update the weights in backward propagation. The mapping from input to output is adaptive to dynamic changes in a batched

dataset or a changing environment in an online learning model and is subject to many hyperparameters, such as the learning rate, the number of epochs, learning algorithm, choice of activation and loss functions, validation process, convergence criteria and others.

The human-brain-inspired model is empowered with the kernel methods from statistics introduced in chapter three, which will be further explored in chapter five. ANN is introduced in first theorised in the 1940s and implemented in the 1950s with various advancements through the initial decades, but with lousy performance due to the lack of large datasets and computing power. The SVM Kernel approaches were state-of-the-art during these decades. The big data scale datasets available now in the public domain and the exponential growth in computing power are the main reasons ANN picked up again in 2006 to become state of the art in various ML tasks. The computing power that was available to supercomputers in the late 2010s is now available in individual desktops and laptops with multiple cores and GPUs or accessible through dense GPU systems in the cloud. This enabled deeper large networks to be established and trained on large datasets, with increased accuracy sometimes exceeding human performance, such as winning in the Go game against humans. For example, the 2017 Google neural machine translation achieved near human language translation using 100 ExaFLOPS supercomputers to train a network to estimate 8700 million meters. The 100 ExaFLOPs (Exascale = 10^{18} Floating Point Operation per second) supercomputer processing is equivalent to two years of processing on a dual CPU server.

This section introduces ANN in general and how tensor decomposition methods can enhance their performance. Then tensorising a data set example will be presented. Chapter six will discuss other detailed applications.

4.5.1 Introduction to NN:

This section provides a very concise description of artificial neural networks to introduce how tensor decomposition methods can be used in these models. The simplest explanation of ANN, which is relatively more straightforward than many other machine learning algorithms, is that it comprises six steps. 1) The NN takes input data and passes it through the neurons of the input layer by multiplying the data by a set of weights. 2) The received set of weights and biases are added in each neuron, and the neuron applies a simple non-linear activation function to the result. 3) Then, the results from the neurons in one layer are stacked up to the following layers to capture more complex patterns/interactions. 4) the final layer is designed to produce the required predicted output of the machine learning task. 5) The predicted output is compared to the pre-labelled output to calculate the loss

and use the loss to backpropagate through the network, adjusting the weights. 6) The forward/backward process is repeated until convergence. The width (number of neurons) of each hidden layer (not the input and the output layers as these are specific to the input and output size required), and the depth of the NN, are learned from best practice papers and models specific to the complexity of the data structure and the learning objective.

For more details on the theoretical and mathematical foundations of these models, books such as (Haykin, 2009) can be explored. For more programming exercises and essential building blocks of neural networks, books such as (Ekman, 2021) can be explored. The origin of the artificial neuron as a mathematical model is the biological neuron, as illustrated in Figure 17 (a), in which the body of the neuron is a node that performs two functions: it computes the sum of the weighted input signals, and it applies an output function to the sum. The neuron output/activation function is usually non-linear; examples are:

- Converting the neuron's output to a set of discrete values.
- Limiting the range of the output values.
- Normalising the range of output values.

The input layers should have enough neurons for the number of input features. The output layer is based on the objective of the model. If it is classification, then the number of classes of the supervised prelabelled data set is the number of neurons in the output layer or other encoding methods to reduce the number of neurons. If it is a regression model, then one output neurone will contain the estimated predicted outcome continuous value. Adding more hidden layers between input and output, as illustrated in Figure 17 (b, c), enables more complex functions, not just simple monotonic ones, to be estimated. In computer vision, adding more hidden layers can extract more complex shapes, such as object detection. Recurrent connections that take the output of a neuron to the input of a neuron in the same layer (including itself) can be used to implement memory, as illustrated in Figure 17 (d). ANN implement spatial filters for a dataset of pixels of an image, such that nearby pixels will be input to adjacent neurons. For example, a 16x16 image will require an input layer containing 256 neurons that will read the vectorised image data as intensities of grey shade or any other scalar representation. Spatial filters extract local features such as differences of intensity between adjacent pixels in an image, and this local property can be used for tasks like identifying edges in the image. More details about recurrence and convolution will follow.

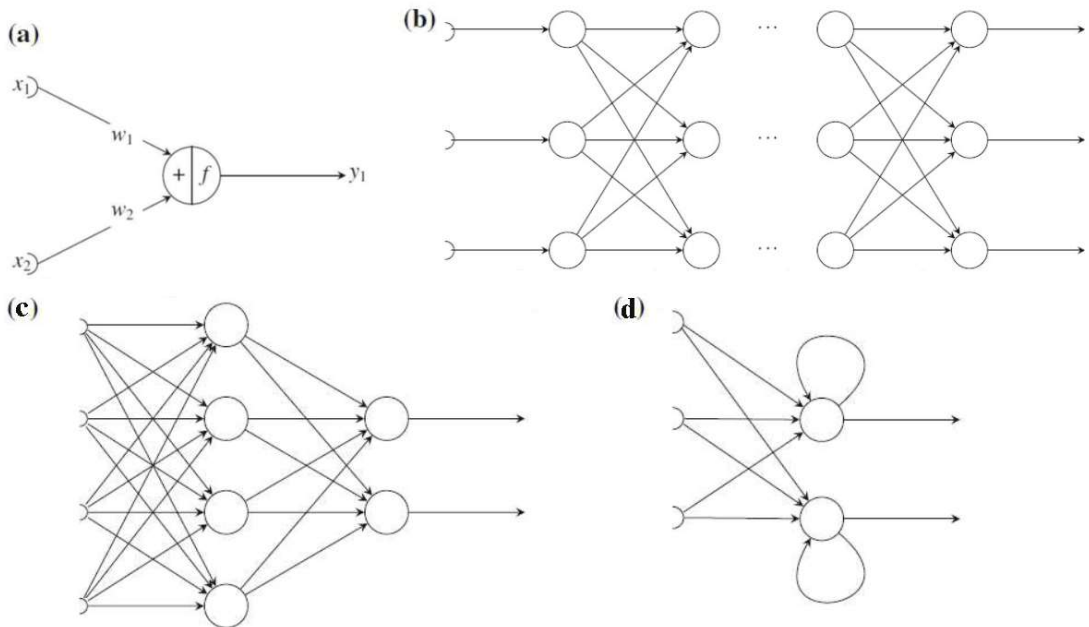


Figure 17: (a) ANN: one neuron with two inputs, (b) Fully Connected Neural network for deep learning such that many hidden layers can be added, (c) Another example of a 3-layer ANN, (d) ANN with memory

The weights/parameters/coefficients of the input features are learned/estimated using various approaches. The main classes of approaches are:

- **Supervised Learning:** The output required for every set of inputs is pre-defined. The network is trained to estimate the weights by reducing a quantified error while learning from a prelabelled training dataset and generalising for unseen datasets by approximating a function of the inputs and the weights: $y = w_1x_1 + w_2x_2 + \dots + w_nx_n$.
- **Reinforcement Learning:** A good output vs bad output is generally defined as using a utility function that generalises the objective without specific details. An environment of the problem specifies possible actions, rewards and states. The network is trained to maximise the utility function/reward and store the state/action/reward as samples or strategies to learn from in the future. For example, in a collision-avoidance navigation algorithm with an unknown map, in which the utility function is the shortest path, it learns by attempting possible strategies/actions (such as driving forward until an object is sensed ahead, then turning right). All sensed objects' positions are recorded, and better shortest paths are calculated later. Then strategies are updated as the map change, object positions change, and utility function changes.
- **Unsupervised Learning:** There is no specific output, but the inputs have patterns that can be estimated, such as in clustering.

CHAPTER 4

The Hebbian rule updates the ANN connections' weights between two neurons, such as increasing or decreasing them in proportion to the product of their activation:

$$\Delta w_{kj} = \alpha y_k x_j$$

Such that neuron k and neuron j having connection weight w_{kj} with Δw_{kj} is the change of w_{kj} , y_k is the output of neuron k, x_j is the input of neuron j, and α is a constant that defines the speed of learning.

A perceptron is an artificial neuron with a weighted summation unit with inputs $\{x_1, x_2, \dots, x_n\}$ and each x_i is multiplied by a factor w_i before summation. An additional input x_0 has the constant value 1 for setting a bias independent of the inputs. The output is obtained by applying a function f to the addition result. For binary classification, this output can be the sign of the summation indicating a + class or a - class:

$$y = \text{sign} \left(\sum_{i=1}^n w_i x_i \right) = \pm 1$$

Perceptrons are the basic building blocks of deep learning, and the Hebbian learning rule is the basic rule applied in the stochastic gradient descent algorithm using partial derivatives for the high dimensional datasets: $w_i(t+1) = w_i(t) + \eta \left(\frac{dy}{dx_i} \right)$ that is added or subtracted in the opposite direction of the gradient. The algorithm starts by initialising the weights arbitrarily, such as 0.1. Then iteratively update the weights until convergence. The learning phase/iterations are terminated when the model converges. Convergence is defined as reaching 98% of the samples correctly classified or when the magnitudes of the corrections to the weights become small below a specified threshold.

Multiple Layer Perceptrons draw a new decision line boundary by each perceptron that can be simply effective in multiclass classification, such as Multiclass Linear Discriminant Analysis (LDA), or causing the decision boundary to be non-linear such as the Quadratic Discriminant Analysis (QDA). Both LDA and QDA are parametric in that they measure the data against a pre-defined form (linear or quadratic) and similar distributions in each class. ANN does not require these assumptions.

Computing partial derivative for multiple neurons for each dimension in the dataset, in multiple layers are computationally expensive. This is solved by using a backpropagation algorithm: 1) The forward pass, where we present a learning example to the network and compare the network output to the desired value (the ground truth). 2) The backward pass, where we compute the partial derivatives with respect to the weights. These derivatives are then used to adjust the weights to make the network output closer to the ground truth. The gradient descent algorithm tracing over a simple network is animated in <https://developers->

dot-devsite-v2-prod.appspot.com/machine-learning/crash-course/backprop-scroll. Instead of the non-differentiable sign function, the tanh or continuous sigmoid function is used.

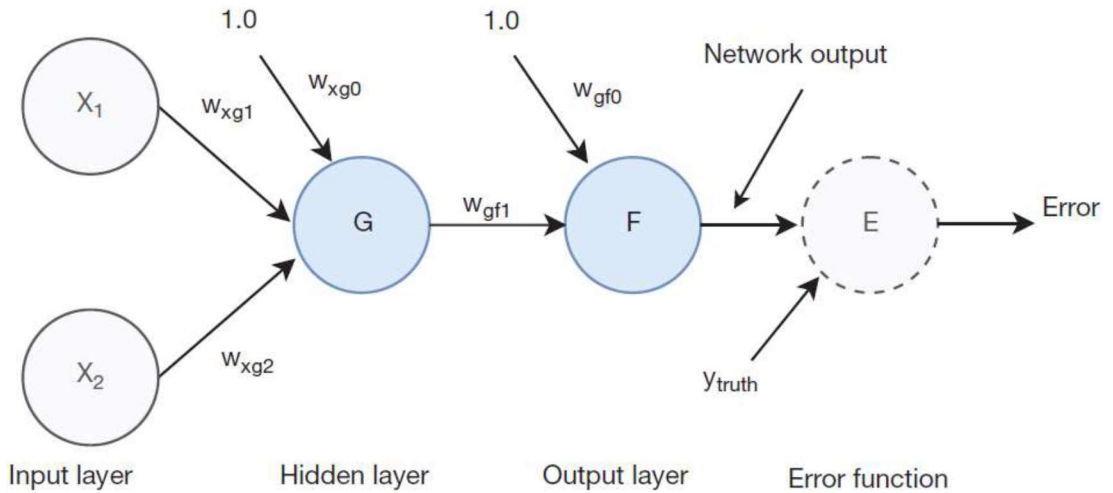


Figure 18: Simple two-layer network used to explain backpropagation. The last unit (dashed) is not a part of the network but represents the error function that compares the output to ground truth (Ekman, 2021).

Figure 18 shows a two-layer ANN with 2 inputs and one output for a dataset with m samples. Each sample is applied to the network in each iteration to update the weights. In designing an ANN model, various building blocks need to be put together, such as:

- **The input aggregation functions/activation function in each neurone:** for example, for the ANN in Figure 18: a Sigmoid function can be applied for hidden layer F neurone and a tanh for input layer G neurone, such that the predicted outcome can be computed as: $\hat{y} = \text{Sigmoid}(w_{gf} + w_{gf1} \text{tanh}(w_{xg0} + w_{xg} x_1 + w_{xg} x_2))$
- **The number of hidden layers** required (and their type) and the number of neurons in each layer, and their connections.
- **The loss/Error Function:** for example: Mean squared error $MSE = \frac{1}{m} \sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2$
- **The minimisation algorithm of the error function:** for example the minimisation can be done analytically using calculus, such as computing a composite function by applying the chain rule in calculating the derivatives: $\text{Error}(w_{gf}, w_{gf}, w_{xg}, w_{xg} x_1, w_{xg2} x_2) = e \circ f \circ z_f \circ g \circ z_g$, where z_f is the activation function of neuron F, and z_g is the activation function of neuron G. The iterative method is to use gradient descent class of algorithms to minimise an error until a threshold is met, or a number of iterations are performed.
- **Other parameters to fine-tune**, such as the learning rate, the number of iterations, the convergence criteria, and the regularisation parameter to avoid overfitting. Each

implementation comes with default values, but learning a fine tuned values to a particular dataset or task can be achieved using search algorithms.

A simple visualisation can be found in <https://aegeorge42.github.io/>. Another Multiclass classification ANN is shown in Figure 19. The network takes input from the MNIST dataset of handwritten digits containing 3D arrays of images, where the first dimension selects one of the 60,000 training images or 10,000 test images. The other two dimensions represent the 28×28 pixel (flattened as 784 pixels) values (integers between 0 and 255). Each pixel is fed to one neurone in the input layer. A fully connected network (i.e. each neuron in one layer connects to all neurons in the next layer) with one hidden layer is constructed. One Hot encoding output layer, in which only one is activated to 1 as the estimated class/digit, and the remaining nine neurons/units are zeros (Ekman, 2021).

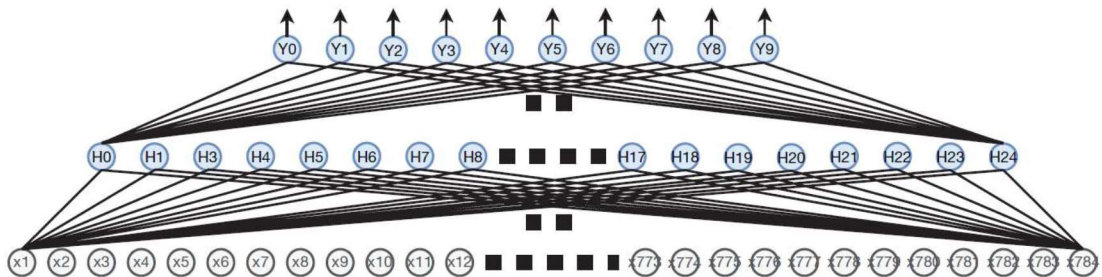


Figure 19: Network for digit classification. A large number of neurons and connections have been omitted from the figure to make it less cluttered. In reality, each neuron in a layer is connected to all the neurons in the next layer (Ekman, 2021).

Researchers have been proposing new NN models to handle different data types, machine learning tasks and complexity. Major neural networks architectures are convolutional neural networks (CNN) that are suitable to spatial datasets such as images, maps, or any n-D datasets where the position of an input is related to the positions of spatially related inputs. CNN complexity is much less than fully connected layers and all other remaining architectures. The remaining models are discussed in the order of increasing complexity. Recurrent Neural Networks (RNN) are suitable to sequential datasets, such as time series, sentences, audio and video files. There is also Recursive NN which sometimes uses the same acronym (RNN) and suitable to learning on graphs. There is also long/short term memory (LSTM), and their variant simplified Gated Recurrent Units (GRUs) that solves the vanishing gradient problems. There are also multi-component NNs such as encoder/decoder models, Generative Adversarial Neural Networks (GAN), among many others. In the following, some of the major architectures will be concisely introduced.

4.5.1.1 CNN:

ANN input layer uses a neuron per input variable, such as a column in a dataset. This requires flattening a 2D image input, as seen in the MNIST example illustrated in Figure 19. The spatial information that connects a pixel to its eight neighbours spatially on a 2D grid is important to capture. This is what CNN is doing. CNN input layer uses a receptive field of the same size as the convolution filter size that divides an image input, for example (or any spatial data) using the kernel size, the stride (how much overlap in the image between the filter application), and requires padding in case of residuals. There is a 1D convolution to connect a data point to its previous and following data points, but images usually require 2D convolutions, and we also have 3D convolutions for video files with an extra frame/time dimension. A 2D convolution animation of different kernels can be found at <https://setosa.io/ev/image-kernels/>.

The grid of neurons creates a feature map for the image, such that each layer has multiple output channels. Each neuron acts as a feature/pattern identifier (such as vertical line, horizontal, ... etc.) and is activated if the particular feature is found in the location covered by that neuron's receptive field. This provides translation invariance, i.e. identifying an object found in any special position in an image, by sharing the weights between all neurons in a single channel. The next chapter will discuss translation invariance from different perspectives and other forms of invariance. The sparse connection (each neuron works on its assigned receptive field only) leads to efficient computation costs. CNN layers are stacked such that each layer receives inputs from multiple input channels of the previous layer and produces multiple-output channels, such that a neuron in subsequent layers receives $N \times M \times M$ inputs (+bias) for N output channels from the previous payer, and the kernel size of $M \times M$. The resolution (number of neurons per channel) of the first convolutional layer is lower than the resolution of the image, and further CNN layers need to be of lower resolution than previous layers. For example, use a stride greater than 1, or use a max-pooling layer to reduce the size of a layer by combining the output neurons within channels such as every 2×2 neurons and outputs the max value of these four neurons. Stacking CNN layers will recognise more complex objects than in the initial CNN layer where simple vertical/horizontal lines and other kernels are used. CNN layers compute fewer parameters than Fully connected layers. For example, assume an image $32 \times 32 \times 3$ and a CNN layer, the weights to learn for the 3×3 kernel is $3 \times 3 \times 3 + 1 = 28$, such that the third factor (3) represents the three channels in the previous layer. The +1 is the bias weight.

Figure 20 illustrates the receptive fields in 2D convolution over images and the different kernel sizes and strides effect over the convolution through the image size. Figure 21 shows an example CNN network learning the different features by stacking the CNN layers, then flattening to merge the learned objects and produce a class prediction, and also shows the

CHAPTER 4

receptive field as it grows deeper by stacking the CNN layers merging identified objects into more complex objects. A fully connected layer would have required $32 \times 32 \times 3 + 1 = 3,073$ weights to learn.

There is an extensive history behind the choice of kernel sizes, layers and their types in many of the CNN models and classical computer vision concepts that can be reviewed from multiple other references. For online references: there is an article diving into the different types: <https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215>. There is also an article explaining the math: http://d2l.ai/chapter_convolutional-neural-networks/channels.html (Ekman, 2021).

There are also now many pre-trained models that can be used directly or can be tuned to a particular dataset or ML task that can benefit from the learned weights. Using the pre-trained models, the initial layers are most general, and the later layers are more specific. The final layers can be changed as required to be suitable for a new ML task, by selecting which layers to keep and which to remove and add more layers as required, or freeze some layers' weights from being updated during the fine-tuning. For example, VGGNet (16 layers) and GoogLeNet (22 layers) are close to human-level identification since 2014, then beaten by ResNet-152 consisting of 152 layers in 2015. Many other pre-trained models on various machine learning tasks can be fine-tuned as required, such as those posted in online repositories like [TensorFlow Hub](#), or TensorFlow specific to Keras, [Pytorch Hub](#), the [NGC Catalog](#), GitHub and <http://paperswithcode.com>. There is also a low code/no-code framework for using transfer learning in new applications provided by [the NVIDIA TAO Toolkit](#).

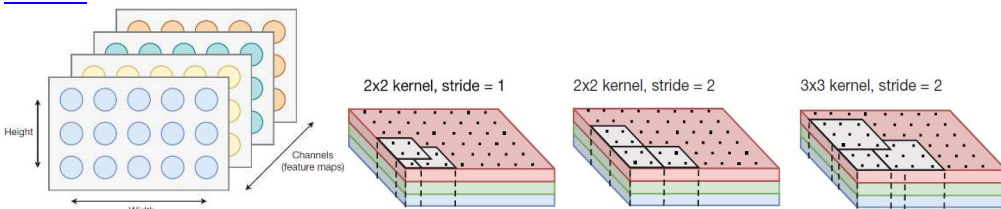
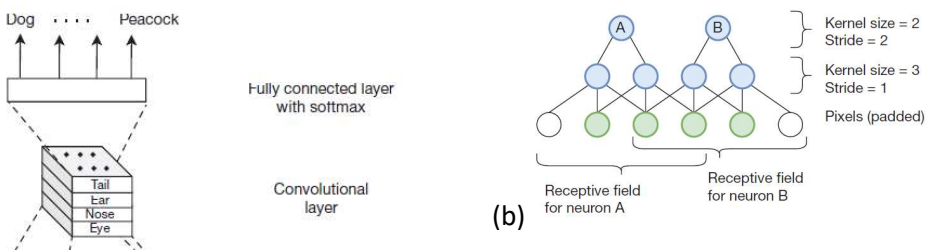
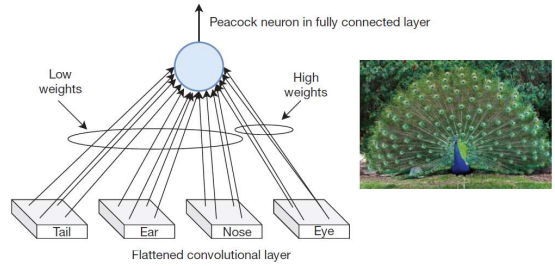


Figure 20: CNN Receptive fields, kernels, strides, multiple channel outputs, sharing weights



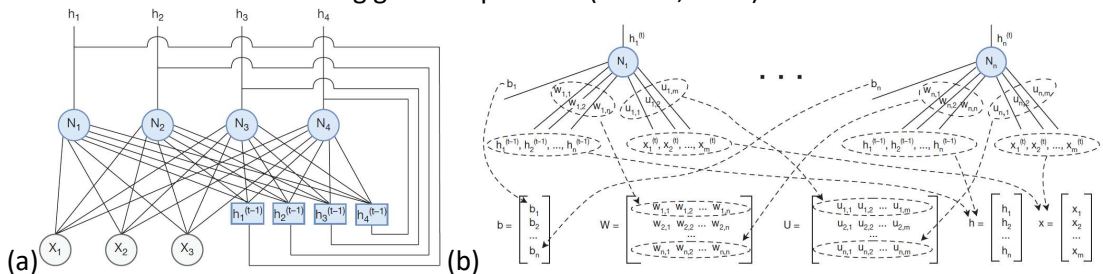


(c)

Figure 21: (a) CNN Layers stacking identifying different features, then combining features to identify more complex objects, then flattening using Fully Connected Layer to produce the identified object type (b) The receptive field increases deeper into the network. Although the neurons in the topmost layer have a kernel size of only 2, their receptive fields are four pixels. Note the padding of the input layer. (c) How a neuron in the fully connected layer combines multiple features into an animal classification (Ekman, 2021).

4.5.1.2 RNN:

A Fully connected RNN layer connects the outputs from a dense layer to the inputs of that same layer, such that the number of inputs (weights) to a single neuron is now a function of both the size of the input vector and the number of neurons in the layer. This enables the weight to learn the interaction with previous values keeping the sequential data dependence such as time series and input value changes over time steps. Multiple recurrent layers can be stacked after each other to create a deep RNN, and a combination of recurrent layers, regular fully connected feedforward layers, and convolutional layers in the same network are also possible based on the machine learning task at hand. For example, for a regression ML task, one neuron is required in the output layer with Linear activation to produce the weighted sum of the predictive features. The regression accuracy measure in the output layer can be Mean Absolute Error to measure how far a predicted continuous value is from the ground truth. More details about the computation in RNN is illustrated in Figure 22. Bi-directional RNN learns the sequential dependence in both directions (look at future words) by having two RNN layers operating in parallel, but they receive the input data in different directions. Because of the successive weight multiplications over the time-steps, RNN suffer from the vanishing gradient problem (Ekman, 2021).



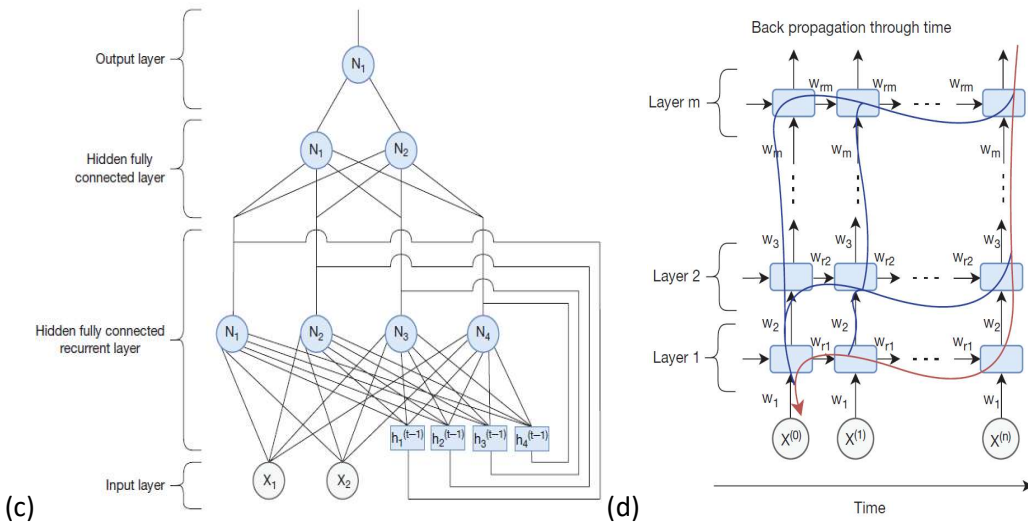


Figure 22 (a) Simple 4-neuron RNN layer with 3 inputs, reading back their output along with every new input applied to the network (b) The activation function in Dense Layers is: $y = \tanh(Wx)$, while in RNN Layers, it is: $h^{(t)} = \tanh(W_h h^{(t-1)} + Ux^{(t)} + b)$, in which $h^{(t-1)}$ is the previous output and their Weights $W, x^{(t)}$ are current input with their Weights U , and the bias for each neuron to produce the new output $h^{(t)}$ (c) shows a network that learns the book sales prediction from two inputs: historical book sales and overall consumer spending, using dense RNN 4-neurons input layer that read their own output along with the new iteration of new input (time step), such that they continuously update their learned weight (hidden states) with every time step, then pass on their output to a fully connected layer of 2 neurons to simplify the network, then to a final output layer of a single continuous variable using simple linear activation. (d) A flattened through time RNN ($n+1$ time-steps in each layer) that shows how the backpropagation through time (BPTT) algorithm takes weight sharing into account when training RNN layers. The figure shows weights connecting the layers and recurrent weights back to the same layer: $w_{r1}, w_{r2}, \dots, w_{rm}$. The error from the output propagates backward both through the network (vertically) and through time (horizontally) (Ekman, 2021)..

4.5.1.3 LSTM:

Long Short-Term Memory (LSTM) is a more complex unit that acts as a drop-in replacement for a single neuron in a recurrent neural network (RNN). LSTM addresses the gradient vanishing and exploding problems from training both vertically and horizontally in Deep RNN over long time-steps, and enables capturing attention. If the LSTM is fed long paragraphs of text (or any sequential data), it remembers the key elements of that text or data, while RNN would be more affected by the most recent elements. For intuition on how LSTM works, watch the video in <https://www.youtube.com/watch?v=8HyCNIVRbSU>. The LSTM cell has no less than five non-linear functions: Three are logistic sigmoid functions known as the gates in the unit, making it a gated unit. Two can be any regular activation functions, with popular choices being tanh and ReLU. It has four weighted sums, so the number of weights is four times as many as in a simple RNN. Multiplying a value by the output of a logistic sigmoid function results in the logistic sigmoid function acting as a gate: 0 to close, and 1 to open and capture the value ($X^{(t)}$). tanh activation function is popular because many RNNs are not as deep as feedforward networks, without severe vanishing

gradient problems. ReLU function can also be used as input and output activation functions in the LSTM. Multiple LSTM cells can be connected into a recurrent network layer, just like a regular RNN, but each neuron has been replaced by the LSTM cell. This results in a network with two sets of states. We have the internal state (c) inside each LSTM cell, and we also have the state (h) in the global recurrent connections, just as in an RNN based on simple neurons. Figure 23 illustrates an LSTM cell, an LSTM layer, and their unrolling over time. The complexity of LSTM is 4-times the complexity of RNN (learning more parameters); Gated Recurrent Units (GRUs) simplify LSTM cells by not having an internal cell state. GRUs have only a single activation function, and the forget and remember gates are combined into a single update gate. LSTM and GRU are the most popular units used in RNNs, among other variations.

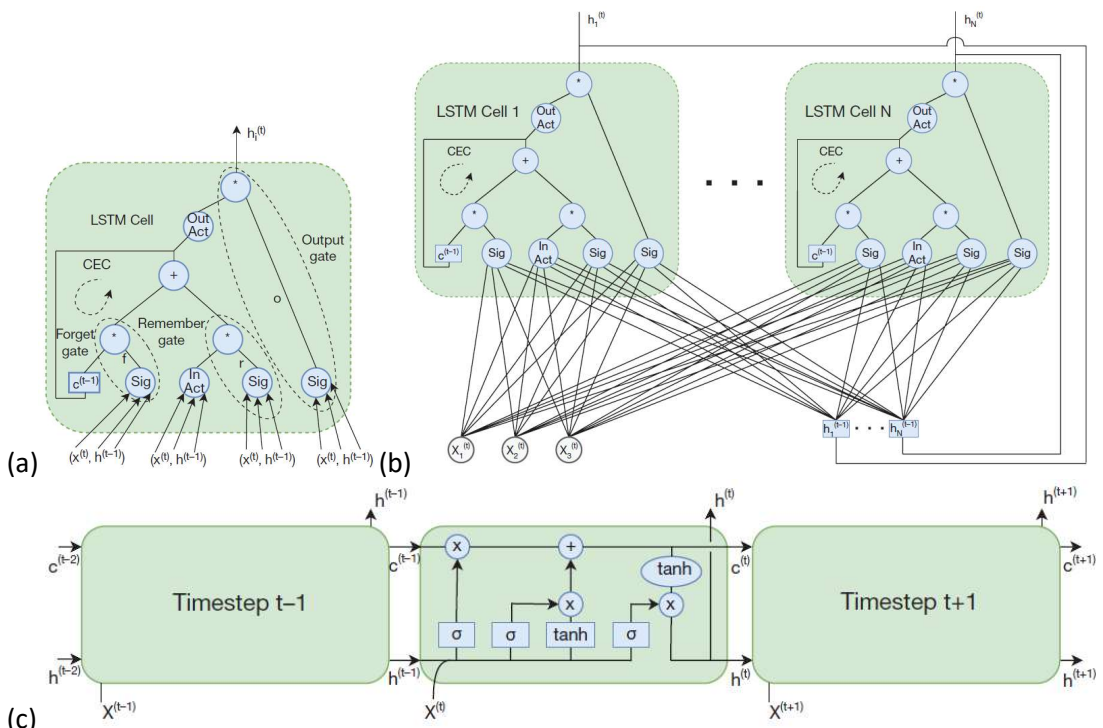


Figure 23: (a) LSTM Cell showing 5 activations functions and their input and output, (b) LSTM Layer showing connections of LSTM Cells, (c) LSTM unrolled in time

4.5.1.4 Multi-component NNs:

Having two networks in one model is useful, and the most popular architectures are the encoder/decoder and the Generative Adversarial Network (GAN). Sequence to Sequence (Seq2Seq) problems have input sequences to learn from how to produce an output sequence. The encoder network learns the hidden states' representation of an input sequence, such as an embedding of the input space (a lower dimension generalised equivalent) to pass on to the decoder network to learn the corresponding output sequence

from samples of prelabelled data. The most common application is in machine language translation, such that the input length can be different from the output length, and word order and context vary from one language to another. Various stacking of models can create an encoder/decoder model, such as RNN encoder to RNN decoder, CNN encoder and CNN decoder, LSTMs and many combinations based on the objective of the network and type/structure of the input and the output. The Transformer model is an encoder/decoder model that learns input data representation in the encoder by using multi-head self-attention layers and exposing all encoder states to the decoder to have different weights of the different states. This will be further discussed in chapter six.

Generative models in Deep learning are similar to the generative models in machine learning; it is about learning the data distribution and how samples are generated in a class, rather than just being discriminative among the classes. Autoencoder networks generate latent vectors that describe the network input. Variational autoencoders learn the latent representation (distribution parameters of the latent space of the input) such that it can generate output that interpolates the given input, such as continuing sentences or drawing images.

GANs apply some game theory models, using a generator and discriminator components as players. The generator learns the input data representation and generates new samples, labelled as fake or generated, but as close as possible to the genuine/real data. On the other side, the discriminator keeps learning how to identify the generated from real data as both continuously learn their weights. The model is called Adversarial because of the competition between the players to perfect their output, the generator aims to make the discriminator fail, and the discriminator aims to identify generated from real data correctly.

4.5.2 Tensorisation Benefits:

The tensor decomposition methods can be applied in any building block to reduce the number of parameters providing compression of the NN model and capturing multi-way structures that increase the model's expressive power, increasing the accuracy and reducing overfitting. The data can already be collected in tensor forms, such as from multiple experiments or multimodal readings. A vector or matrix form dataset can also be tensorised to tree or chain structures to benefit from the tensorised algorithms. This will be explained in this section.

Low-rank matrix factorization of weights in fully connected layer

$$\mathbf{W} \approx \mathbf{A}\mathbf{B} \quad \text{Compression: } \mathcal{O}(MN) \rightarrow \mathcal{O}(r(M+N))$$

$(M \times N) \quad (M \times r) \quad (r \times N)$

Low-rank tensor network factorization of weights

- ▶ Step 1: $\mathbf{W} \rightarrow \mathcal{W}$ (Matrix to d -order tensor)
- ▶ Step 2: $\mathcal{W} \approx \text{TT}(\mathcal{G}_1 \cdots \mathcal{G}_d)$ (Tensor network representation)

Loss function:

$$\mathcal{L}(\mathbf{W}, \mathbf{x}, \mathbf{y}) \rightarrow \mathcal{L}(\{\mathcal{G}_1, \dots, \mathcal{G}_d\}, \mathbf{x}, \mathbf{y})$$

Compression:

$$\mathcal{O}(MN) \rightarrow \mathcal{O}(dr^2 \sqrt[d]{M} \sqrt[d]{N})$$

Order of tensor
TT-rank

Train very “wide” model

Figure 24: Tensorising Neural Networks (Novikov et al., 2015)

In the first ANN building block, activation and loss functions choice can be tensorised, as shown in Figure 24. The standard weighted sum of dot products of input vectors with weight vectors and then aggregating them is suitable for vector and matrix form datasets. A tensorised activation function for a tree data structure can start from the node assigned to a neuron to recursively compute the weighted sum of its children with weight sharing between neurons to reduce the model complexity. This model, called a recursive neuron, is first modelled for binary trees and leads to a higher-order generalised n -ary tree using tensorised aggregation. CP decomposition or Tensor Train (TT) decomposition can further decompose the full format tensor aggregation. Chapter six will introduce more details about tensorised activation functions (Bacciu and Mandic, 2020). The loss function as well can use the decomposed tensor cores of the weights tensor.

In the second ANN building block, the choice of the number of hidden layers and the number of neurons per layer can also benefit from the compressive nature of tensor decompositions algorithms. Tensor Networks can compress the whole Deep Neural Network (DNN) using a suitable tensor decomposition algorithm and then map back to the uncompressed form. Another approach is to update only the final fully connected layer (or specific layers of interest) of a model with a tensor decomposition layer, such as TT. Usually,

compressed models benefit from wider fewer layers (shallower networks). Chapter six will also introduce more details about layers' compression (Bacciu and Mandic, 2020).

Back to tensorising datasets, tensor network representations often allow for super-compression of datasets as large as 10^{50} entries, down to the affordable levels of 10^7 or even less. The previous section shows an example of the MNIST dataset of handwritten digits containing images of 28×28 pixels (flattened as 784 pixels) values representing grey shade as integers between 0 and 255. Data in the tensor form can represent a coloured image with pixel values for red content, green content, and blue content as three different values in the (RGB) frames stacked into a 3rd-order tensor. Similarly, a video dataset can include the 3rd-order coloured image frames extended with the time dimension in a 4th-order tensor. This new arrangement of the data will require an alignment of the data slicing into the different epochs. These blocks of tensorised data need to represent the multi-way structures in the dataset identifying latent variables so that the learning iterations can reduce the error.

Domain-transform methods can achieve this representation. The values stored in a given index vector capture interaction between the dimensions/modes. This means symmetric or partially symmetric tensors might be sufficient to capture the inter-mode interactions, ignoring the values in a permuted index (same modes in a different order) in which the value might be redundant. For example, in the EEG dataset, we have F frequency measures collected over T time samples from S channels, forming a 3rd-order tensor. Transforming the domain of this 3rd-order tensor to get the time-frequency decomposition can be achieved using a short-time Fourier transform (STFT) that uses a fixed window size or wavelet transform (WT) that uses variable window sizes inversely proportional to the frequency resolution (high or low). Other transformations can represent data at multi-scale and orientation levels, such as the Gabor, contourlet, or pyramid steerable transformations. More details about change of basis and representation learning will be discussed in next chapter.

Furthermore, datasets can contain statistically independent latent variables. Hence, the dataset can be represented by higher-order statistics (cumulants) or by partial derivatives of the observations' Generalised Characteristic Functions (GCF). Using a suitable tensor format such as the lower rank core tensors presented in this chapter enables all the above-discussed transformations and others while keeping the number of parameters smaller. The monograph in (Cichocki et al., 2017, p. 2) provides detailed discussions with examples of various forms of tensorisation that prepare a dataset for a compressed tensorised deep neural network model. This section will present one example in detail.

Chapter one presents tensor vectorisation and matricisation functions and the reverse operation of tensorising a vector or a matrix. The tensorisation depends on considering the vector or matrix as a hierarchical object of data blocks. The segmentation is a process of identifying the index ranges of a block, and then each block range of values becomes the size of the dimension/mode in the resulting tensor. For example, a vector of four elements can be thought of as containing two blocks each of two elements. This vector can be matricised such that:

$$M(i,j) = v(i * \text{cols} + j), \text{ such that } M(0,0) = v(0), M(0,1)=v(1), M(1, 0) = v(2), M(1, 1)=v(3).$$

M matrix is considered a folding of vector v. A higher-order folding yielding tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ is considered a folding of a vector v of length $I_1 \times I_2 \times \dots \times I_N$, if $\mathcal{X}(i_1, i_2, \dots, i_N) = v(i)$, for $0 \leq i_n \leq I_n$ for $1 \leq n \leq N$, and for $0 \leq i \leq (I_1 \times I_2 \times \dots \times I_N)$, $i = 1 + \sum_{n=1}^N (i_n - 1) \prod_{k=1}^{n-1} I_k$ as a linear index of (i_1, i_2, \dots, i_N) .

Data or signals come in many structured forms that can be assumed in the tensorisation details. For example, it is observed that a higher-order folding (quantisation) of a vector of length q^N ($q = 2, 3, \dots$), sampled from an exponential function $y_k = az^{k-1}$, yields an N^{th} -order tensor of rank 1. Many functions are formed from products and/or sums of trigonometric, polynomial and rational functions. These can be quantised as shown above to yield (approximate) low-rank tensor train (TT) network formats. These two assumptions can be used in Blind Source Separation (BSS) problem when given a mixture $y(t)$ and it is assumed to be composed of J sources as $y(t) = a_1x_1(t) + a_2x_2(t) + \dots + a_jx_j(t) + n(t)$, where $n(t)$ is added Gaussian noise. Extracting the sources by estimating the mixing matrix A, can be done by assuming a higher-order folding \mathcal{X} of the sources with low-rank representations using one of the known tensor decomposition algorithms, such as CP, Tucker, TT, or TR. The multilinearity of this tensorisation keeps the relation between the tensorised mixture \mathcal{Y} and the tensorised sources \mathcal{X} as $\mathcal{Y} = a_1\mathcal{X}_1 + a_2\mathcal{X}_2 + \dots + a_j\mathcal{X}_j + \mathcal{N}$, where \mathcal{N} is the tensorised noise. Decomposing \mathcal{Y} into any TN format, the separate decomposed components will represent the sources up to a scaling ambiguity.

For example, if the mixture signal length was $L = 2^d J^2$, and the sources are contributing to the mixture equally: $a_1\|\mathcal{X}_1\| = a_2\|\mathcal{X}_2\| = \dots = a_j\|\mathcal{X}_j\|$, we can tensorise the mixture \mathcal{Y} to the d^{th} -order of size $2R \times 2 \times \dots \times 2 \times 2R$. It is observed that An N^{th} -order tensor of size $I_1 \times I_2 \times \dots \times I_N$, where $I_n \geq 2$, which is reshaped from a sinusoid signal, can be represented by a multilinear rank-(2,2,...,2) tensor. The modulated variants of the Harmonic sinusoidal signals, such as the exponentially decaying signals $\exp(\gamma t)$, are fundamental in many practical applications. This tensorisation enables the tensors of $\mathcal{X}_n(t)$ to be represented by tensors in the TT format of rank-(2,2, ..., 2). In order to separate the J signals $\mathcal{X}_n(t)$ from the mixture $\mathcal{Y}(t)$, the process requires fitting \mathcal{X}_n sequentially to the

residual $\mathcal{Y}_n = \mathcal{Y} - \sum_{s \neq n} \mathcal{X}_s$ calculated by the difference between the data tensor \mathcal{Y} and its approximation by the other TT-tensors \mathcal{X}_s , where $s \neq n$. Then, a minimisation step of the reconstructed mixtures from $\operatorname{argmin}_{\mathcal{X}_n} \|\mathcal{Y}_n - \mathcal{X}_n\|_F^2$ for $n=1, \dots, J$. In this setup, the signal

length L determines the quality of the extraction of the sources and the d value to use in the tensorisation. Other tensorisation methods are suitable for short-length signals, such as multi-way Toeplitz or Hankel tensors (Cichocki et al., 2017, p. 2).

An example to BSS problem using tensor decomposition is presented in (Böttcher et al., 2018). The authors built a Python Package “Decompose” that generalises the PCA, ICA, and NMF solutions to the BSS problem. These methods are built on statistical assumptions that might be found in a dataset or might not be found. Each of them would produce different sources when applied on the same dataset. Expert knowledge is usually needed to identify the correct statistical assumptions of a given dataset in an application domain. The authors built a probabilistic BSS model that estimates the priors of every source, can extend to new prior distributions, scale well to large datasets, assuming each source has a different sparsity level, and efficiently estimates the posterior adopted to the dataset. Their code is published in <https://github.com/bethgelab/decompose>.

The python notebook `ch4.ipynb` shows various NN architectures and links to sample tensorial CNN, RNN and others.

References

- Bacciu, D., Mandic, D.P., 2020. Tensor Decompositions in Deep Learning.
- Böttcher, A., Brendel, W., Englitz, B., Bethge, M., 2018. Trace your sources in large-scale data: one ring to find them all.
- Cichocki, A., Lee, N., Oseledets, I.V., Phan, A.-H., Zhao, Q., Mandic, D., 2016. Low-Rank Tensor Networks for Dimensionality Reduction and Large-Scale Optimisation Problems: Perspectives and Challenges PART 1. *FNT in Machine Learning* 9, 249–429. <https://doi.org/10.1561/22000000059>
- Cichocki, A., Phan, A.-H., Zhao, Q., Lee, N., Oseledets, I.V., Sugiyama, M., Mandic, D., 2017. Tensor Networks for Dimensionality Reduction and Large-Scale Optimisations. Part 2 Applications and Future Perspectives. *FNT in Machine Learning* 9, 249–429. <https://doi.org/10.1561/22000000067>
- Ekman, M., 2021. Learning deep learning: theory and practice of neural networks, computer vision, nlp, and transformers using tensorflow, First edition. ed. Addison-Wesley, Boston.
- Haykin, S.S., 2009. Neural networks and learning machines, 3rd ed. ed. Prentice Hall, New York.

CHAPTER 4

- Ji, Y., Wang, Q., Li, X., Liu, J., 2019. A Survey on Tensor Techniques and Applications in Machine Learning. IEEE Access 7, 162950–162990. <https://doi.org/10.1109/ACCESS.2019.2949814>
- Novikov, A., Podoprikin, D., Osokin, A., Vetrov, D., 2015. Tensorizing Neural Networks. NIPS-2015, Advances in Neural Information Processing Systems 28.
- Oseledets, I.V., 2011. Tensor-Train Decomposition. SIAM J. Sci. Comput. 33, 2295–2317. <https://doi.org/10.1137/090752286>
- Pfeifer, R.N.C., Evenbly, G., Singh, S., Vidal, G., 2015. NCON: A tensor network contractor for MATLAB.
- The Tensor Network [WWW Document], n.d. . Tensor Network. URL <https://www.tensornetwork.org> (accessed 9.10.22).
- Zhao, Q., Zhou, G., Xie, S., Zhang, L., Cichocki, A., 2016. Tensor Ring Decomposition.