User Guide for Polynomial Regression via Latent Tensor Reconstruction(LTR)

Document Version 0.24

July 29, 2022

Contents

1 Installation

1.1 Requirements

The application of the LTR assumes the **Python** interpreter, version at least 3.7, and the **Numpy** package, version at least 1.20.

To run the examples also requires the **matplotlib** and **scikit-learn packages**. All these packages can be freely downloaded and installed from pypi.org.

1.2 Installation

The LTR package might be installed by the following procedures.

1.2.1 Directly from the Github

This can be done via this command.

```
pip3 install
  git+https://github.com/aalto-ics-kepaco/LTR_multiview.git#egg=LTR_multiview
```

1.2.2 Downloading from Github

In the first step the LTR package needs to be downloaded from the github.

```
mkdir ltrpath
cd ltrpath
git clone https://github.com/aalto-ics-kepaco/LTR_Multiview
```

After downloading, the LTR can be installed by the following command:

```
pip3 install ltrpath
```

To install the LTR package the latest version of the Python packages **pip** and **build** need to be installed.

```
pip3 install --upgrade pip
pip3 install --upgrade build
```

The LTR can be imported as

```
import ltr_solver_multiview as ltr
```

and the solver object can be created by

```
cmodel = ltr.ltr_solver_cls(norder=2, rank=10)
```

Further details about the application of the LTR package can be found in Section 3, and in Section 4, and in the example files, in the directory of examples.

2 Background: learning polynomial regression models

In this introduction of LTR we closely follow this paper [15] and its preprint [16]. In [16] a large scale application is presented for drug interaction prediction in cancer research.

The LTR is variant of the polynomial regression models

$$\pi(\mathbf{x}) = \sum_{j=1}^{n} w_j x_j + \sum_{j,k=1}^{n} w_{jk} x_j x_k + \cdots + \sum_{j_1,j_2,\ldots,j_{n_d}=1}^{n} w_{j_1,\ldots,j_{n_d}} x_{j_1} \cdots x_{j_{n_d}}, \tag{1}$$

where w's are the regression coefficients to be learned, n is the number of input variables and n_d is the degree of the polynomial.

Polynomial regression models have high representation power, they are capable of accurately representing continuous functions with a fixed L_∞ norm-based tolerance. The Stone-Weierstrass theorem and its generalizations, [11], allows to approximate those functions by polynomials on a compact subset with an accuracy not worse than a given arbitrary small error.

An arbitrary multivariate polynomial defined on the field of real numbers can be described by $\binom{n+n_d}{n}$ parameters, where n is the number of variables, and n_d is the maximum degree of the polynomial. The complexity relating to the size of the underlying parameter tensor is $O(n^{n_d})$, which grows exponentially in the number of degree.

To tackle this exponential complexity the authors of [12, 3] propose a special representation of the coefficients as inner products of factors: e.g., for the second order terms

$$w_{i_1,i_2} = \langle \mathbf{p}_{i_1}, \mathbf{p}_{i_2} \rangle = \sum_{j=1}^{n_t} p_{i_1,j} p_{i_2,j}$$

where $\mathbf{p}_i \in \mathbb{R}^{n_t}$ encodes the participation of i'th variable in n_t factors. For higher degree terms, the same is given by a generalized inner product given by a multilinear function,

$$w_{i_1,\ldots,i_m} = \langle \mathbf{p}_{i_1}, \ldots, \mathbf{p}_{i_m} \rangle = \sum_{j=1}^{n_t} p_{i_1,j} \cdots p_{i_m,j}.$$

This approach is known as Higher Order Factorization Machine(HOFM). This factorized representation significantly reduces the number of parameters to $O(n_d \cdot n_t \cdot n)$ [3]. The HOFM was recently demonstrated to be able to accurately predict drug combination responses [8]. The model of HOFMs exploits the theory built upon the Newton's identities connecting symmetric polynomials to power sums, and elementary symmetric polynomials. A function, a polynomial is a symmetric functions if its value is invariant to

Prediction	Learning
Given : $\mathbf{T}$ tensor, $\mathbf{x}$ Output : y	Given : $\{(y_i, \mathbf{x}_i) \vert i \in [m]\}$ Output : $\mathbf{T}$ tensor
$\pi(\mathbf{x}) = \langle \mathbf{T}, \otimes^{n_d} \mathbf{x} \rangle \Rightarrow y$	$\min_{\lambda, \mathbf{P}} \sum_i \vert\vert y_i - \langle \mathbf{T}, \otimes^{n_d} \mathbf{x}_i \rangle \vert\vert^2$ $s.t.\ \mathbf{T} = \sum_{t=1}^{n_t} \lambda_t \otimes_{d=1}^{n_d} \mathbf{p}_t^{(d)}$

Table 1: The general scheme of Latent Tensor Reconstruction-based regression (LTR). Given an n_d-order parameter tensor $\mathbf{T}$ and data point $\mathbf{x}$, the prediction entails computing an inner product between $\mathbf{T}$ and an n_d-order tensor product of the data point with itself. Learning entails finding a factorization of $\mathbf{T} = \sum_{t=1}^{n_t} \lambda_t \otimes_{d=1}^{n_d} \mathbf{p}_t^{(d)}$ with the lowest regression error.

permutation of features. The background built on the symmetric polynomials imposes some restrictions on the HOFM model as a general regression model.

LTR follows an alternative, but closely relating, approach for factorizing the parameter representation [14] , that starts from the full-order tensor representation of the unknown regression coefficients and learning a factorization into rank-one tensors that optimizes the regression error (Table 1).

2.1 Tensor-based representation of polynomial functions

Since the dual space of the homogeneous polynomials defined on the real numbers is naturally isomorphic to vector space of symmetric tensors, a polynomial function over the real numbers with degree n_d and with n variables can also be written in a compact form

$$\pi(\mathbf{x}) = \langle \mathbf{T}, \mathbf{x} \otimes_{d=1}^{n_d} \mathbf{x} \rangle \,, \tag{2}$$

where $\mathbf{T}$ is a symmetric tensor of order n_d, and with dimension $\times_{d=1}^{n_d} n$. If the vector $\mathbf{x}$ is given in homogeneous form, extended with a constant 1, then (2) covers all possible polynomials up to degree n_d. The tensor $\mathbf{T}$ can be given in a decomposed, HOSVD form, [10, 9],

$$\begin{aligned} \mathbf{T} \quad &= \sum_{t=1}^{n_t} \lambda_t \otimes_{d=1}^{n_d} \mathbf{p}_t^{(d)} \\ \text{s.t.} \quad &||\mathbf{p}_t^{(d)}|| = 1,\ \mathbf{p}_t^{(d)} \in \mathbb{R}^n,\ t \in [n_t]. \end{aligned} \tag{3}$$

This representation is generally not unique, see ([10, 6]). By replacing $\mathbf{T}$ with its decomposed form, the polynomial function turns into the following expressions

$$\pi(\mathbf{x}) \quad = \sum_{t=1}^{n_t} \lambda_t \, \langle \otimes_{d=1}^{n_d} \mathbf{p}_t^{(d)}, \otimes_{d=1}^{n_d} \mathbf{x} \rangle = \sum_{t=1}^{n_t} \lambda_t \prod_{d=1}^{n_d} \langle \mathbf{p}_t^{(d)}, \mathbf{x} \rangle \,,$$

where we exploit the well known identity connecting the inner products and the tensor products [7]. This form only consists of terms of scalar factors, where each scalar is the value of a linear functional acting on the space $\mathcal{X}$. This transformation eliminates the difficulties which arise in working directly with full tensors. Observe that the function π is linear in each of the vector-valued parameters, $\mathbf{p}_t^{(d)}, t \in [n_t],\ d \in [n_d]$.

We can further transform the polynomial representation, (2.1), into a form which does not contain any reference to tensor product. The following simple statement, allows us to introduce an additional model of factorization within the polynomial function to reduce further the number of parameters.

Proposition 1. *The polynomial function $\pi(\mathbf{x})$ can be expressed only by the help of matrix and pointwise, Hadamard, products, namely*

$$\pi(\mathbf{x}) \quad = \sum_{t=1}^{n_t} \lambda_t \prod_{d=1}^{n_d} \langle \mathbf{p}_t^{(d)}, \mathbf{x} \rangle = \boldsymbol{\lambda}^T \circ_{d=1}^{n_d} \mathbf{P}^{(d)} \mathbf{x}, \tag{4}$$

where $\mathbf{P}^{(d)}$ is a matrix of size $n_t \times n$ for any d, whose rows are given as $\mathbf{P}_t^{(d)} = \mathbf{p}_t^{(d)}$, and $\boldsymbol{\lambda}$ is a vector with components $\lambda_t, t \in [n_t]$.

Proof. The matrix-vector product $\mathbf{P}^{(d)}\mathbf{x}$ yields a vector with components $(\langle \mathbf{p}_t^{(1)}, \mathbf{x} \rangle, \cdots, \langle \mathbf{p}_t^{(n_d)}, \mathbf{x} \rangle)$, and after a rearrangement, the original form can be restored. □

2.2 Latent Tensor Reconstruction - basic form

The LTR-based polynomial regression method [14] exploits the representation shown in Proposition 1, which leads to the following optimization problem:

$$\begin{aligned} \min \quad & \frac{1}{mn_y} \sum_{i=1}^{m} \left\| \mathbf{y}_i - \left(\circ_{d=1}^{n_d} \mathbf{P}^{(d)} \mathbf{x}_i \right) \mathbf{D}_\lambda \mathbf{Q} \right\|^2 \\ & + \frac{C_p}{n_t n_d n} ||\mathbf{P}^{(d)}||^2 + \frac{C_q}{n_t n_y} ||\mathbf{Q}||^2 \\ \text{w.r.t. } & \boldsymbol{\lambda}, \mathbf{Q},\ \mathbf{P}^{(d)},\ d \in [n_d], \end{aligned} \tag{5}$$

where C_p and C_q are penalty constants, and matrix $\mathbf{Q}$ projects the vector given by the polynomial function of dimension n_t into the output space.

2.3 Reparametrization of the polynomial representation

In the LTR model, the predictor is implemented via a polynomial function π acting on vectors $\mathbf{x}$ of dimension n. The parameter space corresponding to matrices $\mathbf{P}^{(d)}, d \in [n_d]$ has dimension $n_d n_t n$ which is large enough to fit the polynomial to a nonlinear function with complex structure, but it requires a large sample to achieve a proper estimation of those parameters.

The LTR framework can be improved to increase the flexibility, and in the same time, to reduce the dimension of the parameter space. To this end, let the polynomial function of (4) be reformulated

$$\begin{aligned} \pi(\boldsymbol{\phi}(\mathbf{x})) &= \sum_{t=1}^{n_t} \lambda_t \prod_{d=1}^{n_d} \langle \mathbf{v}_t^{(d)}, \mathcal{A}(\mathbf{U}^{(d)T}\mathbf{x}) \rangle \\ &= \boldsymbol{\lambda}^T \circ_{d=1}^{n_d} \mathbf{V}^{(d)} \mathcal{A}(\mathbf{U}^{(d)T}\mathbf{x}), \end{aligned} \tag{6}$$

where $\mathcal{A}$ is a pointwise activation function, and the matrix $\mathbf{U}^{(d)T}$ is a linear transformation projecting the original input vector into a space with lower dimension, n_k, for each $d \in [n_d]$. That projection can enforce a bottleneck within the polynomial function. This modification preserves the linear dependence on the matrix valued parameters. The expression $\boldsymbol{\phi}(\mathbf{x}) = \mathcal{A}(\mathbf{U}^{(d)T}\mathbf{x})$ might be viewed as a layer of a multilayered perceptron. The main difference is that the layers within the LTR are joined by a polynomial function in a parallel way instead of being connected sequentially. Figure 1 demonstrates the structural differences of the multilayered perceptron and the LTR.

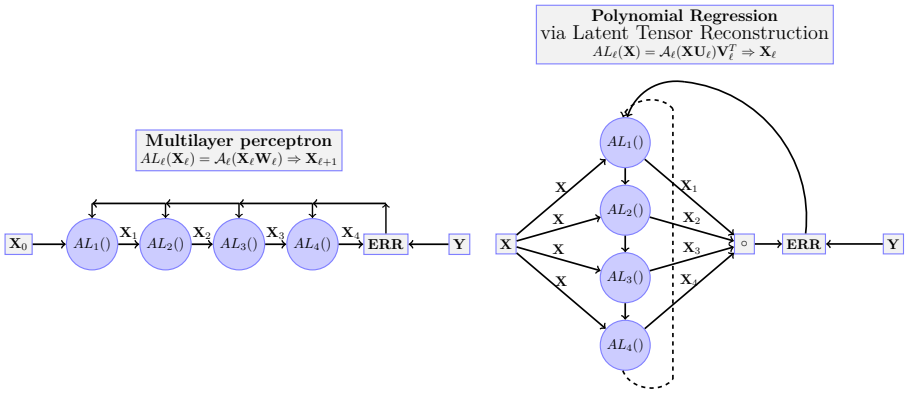


Figure 1: The structural comparison of the Multilayer Perceptron and LTR. The former one follows a sequentially layered model and the sequential error backpropagation. In the latter one, the layers are parallel and combined via a polynomial function which can give the nonlinear power of the representation. The error propagation is circular between the layers. The model of the parallel layers also allows a simple implementation of a multiview learning framework for distinct data sources.

The following table summarizes the matrices describing the extended

LTR problem.

Polynomial	parameters	$\mathbf{V}^{(d)}\ d \in [n_d]$	$\in \mathbb{R}^{n_t \times n_k}$,
Output	projection	$\mathbf{Q}$	$\in \mathbb{R}^{n_t \times n_y}$,
Scaling		$\mathbf{D}_\lambda = \mathrm{diag}(\boldsymbol{\lambda})$	$\in \mathbb{R}^{n_t \times n_t}$,
In layers	projection	$\mathbf{U}^{(d)}\ d \in [n_d]$	$\in \mathbb{R}^{n \times n_k}$,
	scaling	$\mathbf{D}_{\lambda^{(u)}} = \mathrm{diag}(\boldsymbol{\lambda}^{(\boldsymbol{u})})$	$\in \mathbb{R}^{n \times n}$,
	value	$\mathbf{A}^{(d)} = \mathcal{A}\left(\mathbf{X}\mathbf{D}_{\lambda^{(u)}}\mathbf{U}^{(d)}\right)$	$\in \mathbb{R}^{m \times n_k}$,
Full	polynomial	$\mathbf{F} = \circ_{d=1}^{n_d}\mathbf{A}^{(d)}\mathbf{V}^{(d)T}$	$\in \mathbb{R}^{m \times n_t}$,
Error		$\mathbf{E} = \mathbf{Y} - \mathbf{F}\mathbf{D}_\lambda\mathbf{Q}$	$\in \mathbb{R}^{m \times n_y}$.

(7)

The parameter $\boldsymbol{\lambda}$ plays a role similar to the singular values of the tensor decomposition, see in (3).

The extended LTR problem now takes the following form

$$\begin{array}{ll} \min & \dfrac{1}{2mn_y}\left\|\mathbf{Y} - \mathbf{F}\mathbf{D}_\lambda\mathbf{Q}\right\|_F^2 + \dfrac{C_\lambda}{2n_t}||\boldsymbol{\lambda}||_2^2 \\ \text{w.r.t.} & \mathbf{U}^{(d)},\ \boldsymbol{\lambda}^{(u)},\ \mathbf{V}^{(d)},\ \boldsymbol{\lambda},\ \mathbf{Q},\ d = [n_d], \\ \text{s.t.} & ||\mathbf{U}_j^{(d)}|| = 1,\ ||\boldsymbol{\lambda}^{(u)}|| = 1, \\ & ||\mathbf{V}_t^{(d)}||_2 = 1,\ ||\mathbf{Q}_t|| = 1, \\ & j \in [n], t \in [n_t], d \in [n_d], \end{array} \tag{8}$$

where C_λ is a penalty constant relating to the scale factor $\boldsymbol{\lambda}$, and the L_2 norm based constraints are enforced on the rows of the corresponding matrices.

3 Basic class

This section describes the interface of the LTR solver module. Program examples detailing the interface and the use of LTR are in the *examples* subdirectory:

example_main_oneview.py	Training and test on randomly generated nonlinear, vector valued polynomial function, report on the accuracy, and a simple demonstrating graph representing of prediction. It presents some alternative calling designs as well.
example_main_multiview.py	The training and test procedures of *example_main_oneview.py* are extended to cover multiview examples.
data_generator.py	Provides the randomly generated data. It allows to select between simple linear or nonlinear, high degree, interrelated samples.

.

LTR solver is implemented by the following class:

class ltr_solver_cls

This class contains a training function, *fit()*, the predictor function *predict()*, and the a function to modify the learning parameters, *update_parameters*.

After importing the LTR the constructor of the corresponding object can be called with this parameters:

```
import ltr_solver_multiview as ltr
cmodel = ltr.ltr_solver_cls(norder = 1, rank = 10, rankuv = None)
```

The input parameters:

norder : integer; the order(degree) of the polynomial, (default = 1). In the multiview design this parameter is overwritten, see Section 5.1 detailing the multiviews.

rank : integer; the rank of the tensor decomposition, (default = 10).

rankuv : integer; the rank of the decomposition of the parameter matrix of the polynomial. (default = rank).

4 Methods and parameters

4.1 Parameter setting

The learning parameters can be modified via this class method
`cmodel.update\_parameters(self,**dparams)`
or directly setting or getting them in the object `cmodel`.
The potentially important parameters are sorted in the following classes, where the most important parameters are marked with "*".

4.1.1 Optimization parameters

sigma0 * : float; the initial learning speed, the parameter **sigma** stores the changing, diminishing learning speed during the optimization procedure (default=0.05)

nsigma : integer; update frequency of the learning speed, (default=10)

mblock * : integer; block(**mini-batch**) size (default=100)

mblock_gap : integer; if it is equal to the block size then consecutive blocks are processed, if it less than the block overlap each other, if it is more then there is gap between the blocks

gamma : float; discount factor of the polynomial parameters updated (default=1.0)

gammanag : float; discount factor for the ADAM gradient update, (default=0.95)

gammanag2 : float; discount factor for the norm update in the ADAM gradient update, it should be equal to *gammanag* in most of the applications, (default=0.95)

perturb : float; Gradient random perturbation, (default=0). The length of the perturbation is proportional to the Frobenius length of the gradient multiplied by perturb. See further details in Section 9.

ibias : binary; =1 bias term is computed for the polynomial function, =0 no, (default=1).

4.1.2 Loss and regularization

lossdegree * : float; = 0 squared L_2 norm, = 1 L_2 norm, (default=0)

iquantile * : binary; = 1 Pinball loss for L_1 norm, and quantile regression, = 0 no, (default=1), smoothed L_1 norm regression. See further details in Section 12.

If **iquantile=1** then it overwrites the **lossdegree**.

quantile_alpha : float; quantile confidence parameter, e.g. 0.05, 0.95, in case of smoothed L_1 norm regression it is 0.5, (default=0.5). See further details in Section 12.

quantile_smooth : float; smoothing parameter of the quantile, pinball loss, if it is greater then smoother the loss is, if it is smaller then it is closer to the Pinball loss, (default=1). See further details in Section 12.

regdegree * : integer; degree of the regularization norm, (default=1, Lasso)

cregular * : float; regularization constant, (default=0.000005) for the $\boldsymbol{\lambda}$ optimization parameter, see C_λ in (8).

4.1.3 Data transformations

ixmean : binary; = 1 input is centralized by mean, = 0 not (default=0)

ixscale * : binary; = 1 scale the input variables by L_∞ norm, = 0 not, (default=1)

ixl2norm : binary; = 1 scale the input examples by L_2 norm, = 0 not, (default=0)

iymean : binary; = 1 output is centralized by mean, = 0 not (default=0)

iyscale * : binary; = 1 scale the output variables by L_∞ norm, = 0 not, (default=1). The scale factor is saved in the ıfit() function, and in the prediction that factor is applied to rescale the predicted value, thus the finale scale will be correct.

ihomogeneous * : binary; = 1 input examples are homogenize, = 0 not, (default=1).

iactfunc : integer; selector of the activation function, the currently implemented ones: =0 identity, =1 arcsinh, =2 $2 * sigmoid - 1$, =3 tanh, =4 Relu. (default = 0).

norm_type : integer; =0 after updates the rows of parameter $\mathbf{U}$ and $\mathbf{V}$ are projected onto a ball. See other alternative in Section 10.

4.1.4 Log report

report_freq : integer; the frequency of the verbosing report about the convergence. The frequency counts the min-batches processed, (default = 100). The larger frequency requires more computational time of the reported parameters. The parameters reported are in this order:

1	*rank,*
2	*the total counter of the mini-batches processed,*
3	*the number of epoch processed,*
4	*counter of the mini-batches within one epoch,*
5	*the RMSE error on the currently processed mini-batch,*
6	*the Pearson correlation of the prediction and the observed output computed on the currently processed mini-batch,*
7	*the norm of* $\boldsymbol{\lambda}$*,*
8	*the total norm of* $\mathbf{U}^{(D)}$ *for all order,*
9	*the total norm of* $\mathbf{V}^{(D)}$ *for all order,*
10	*the total norm of* $\boldsymbol{\lambda}^{(U)}$ *for all order.*

See the role of parameters in (7).

4.1.5 Example

The parameters can be modified via this function:

```
## set learning parameters
cmodel.update_parameters(nsigma=1, \
                         mblock=100, \
                         sigma0=1, \
                         gamma=1.0, \
                         gammanag=0.9, \
                         gammanag2=0.9)
```

See further details in *example_main_oneview.py* and *example_main_multiview.py* .

4.2 Training by the $fit()$ function

```
cmodel.fit(lXtrain, Ytrain, \
           llinks = None, xindex = None, yindex = None, \
           nepoch = 10, idata_add = 0, imultitask = 0)
```

It implements of the training phase of the learning, namely fits the tensor represented polynomial to the data.

Input parameters:

lXtrain : list of 2d arrays; List of the input data tables represented by 2d arrays. If there is only one table then the list can be replaced with 2d array defining the table. See further details in Section 5.1.

Ytrain : 2d array; The 2d array of the output data table, if the output is provided as vector then it will be converted into 2d array and in the predictor phase the output is returned as 2d array as well.

llinks : list; It describes the design, how the data tables are combined into views, (default = `None`). If it is `None` then the the number of views is set to `norder`, and each view contains the first table of the list `lXtrain`. See further details in Section 5.1.

xindex : 2d array; If it is not `None`, then it contains the joining indexes relative to the data tables. Data examples of different views are joined if their indexes are in the same row of this array, (default = None). See further details in Section 5.1.

yindex : vector; If it is not `None` then it contains the indexes to the rows of output data table. This index vector allows to store the output examples only ones in Ytrain, e.g., if there are 10000 examples but only 10 different outputs, only the 10 different examples need to appear in Ytrain. See further details in Section 5.1.

nepoch : integer; the number of epoch, the repetition of the entire processing of the full input and output data.

idataadd : binary$(0, 1)$; $= 0$ cold start, all optimization parameters are initialized, $= 1$ the data is extended with new rows without initialization of the optimization parameters. (default $= 0$).

imultitask : ‘ binary$(0, 1)$; $= 0$ cold start, all optimization parameters are initialized. $= 1$ warm start, the latest gradient can be modified with external adjustment and the training can be continued. In the first run it needs to be 0. (default $= 0$).

See further details in *example_main_oneview.py* and *example_main_multiview.py* .

4.3 Prediction by the predict function

```
cmodel.predict(lXtest, Ytrain = None, \
                llinks = None, xindex = None, \
                itestmode = 0)
```

It provides the prediction based on the polynomial parameters computed in the training.

Input parameters:

lXtest : list of 2d arrays; list of the input data tables represented by 2d arrays. This list has to contain the same number of tables as the input parameter **lXTrain** in the *fit()* function. The order of those tables has to follow the corresponding order in **lXTrain**.

Ytrain : 2d array; if it is **None** then the prediction directly compute the regression result, if it is provided as a 2d array of the training output data then that training output example is selected as prediction which is the closest in sense of the highest correlation to the direct regression result. (default = None)

llinks : list; It needs to be the same as the **llinks** parameter in the *fit()* function. If **llink** is **None** in the *fit()* then it has to be **None** in this function. (default = None)

xindex : 2d array; It has the same structure as the **xindex** in the *fit()* function. The indexes stored here have to point into the tables stored in **lXTest**. (default = None). See further details in Section 5.1.

itestmode : integer; It is considered when the training outputs **YTrain** is provided, if not it is **None**. If it is $= 0$ then the prediction is given as the training example with the highest correlation to raw regression

prediction. If it is = 1 then the prediction is an indicator vector with 1 in the position of the highest raw regression prediction value. For example, it can be applied in a multiclass prediction problem. (default = 0)

Output parameters:

Ypredict	2d array	The predicted output vectors.

Example 2. *Let the order be 3, the rank 10, and only one view is given.*

```
import tensor_vector_multi_cls as tensor_cls

## Xtrain contains training input vectors in the rows
## Ytrain contains training output vectors in the rows
## Xtest  contains test input vectors in the rows

cmodel = tensor_latent_vector_cls(norder=3, rank=10)
cmodel.update_parameters(mblock=100, \
                         sigma0=1, \
                         gammanag=0.99, \
                         gammanag2=0.99)

lXtrain=[ Xtrain for _ in range(norder)]
cmodel.fit(lXtrain, Ytrain, nepoch=10)

lXtest=[ Xtest for _ in range(norder)]
Ypred=predict(lXtest, Ytrain=None)
```

See further details in *example_main_oneview.py* and *example_main_multiview.py* .

4.4 Reading out the optimization parameters

The optimization parameters can be extracted from the learning object:

```
## d = 0,...,cmodel.norder)-1
U_d = cmodel.xU[d].xT
V_d = cmodel.xV[d].xT
LambdaU_d = cmodel.xLambdaU[d].xT
Q = cmodel.xQ.xT
Lambda = cmodel.xlambda.xT
```

5 Multiview learning

The parallel representation of polynomial function opens up a relatively simple way of integrating different data sources as view into one learning problem. Those sources might be represented by data tables sampled from different distributions. The basic concept of the multiview learning models the **join** operations of the SQL database systems, [5] [13]. In that model we might have several data tables represented by 2d arrays, matrices, and there is a table, the index, which allows to combine those tables to realize the multivariate relationships between the rows of the data tables. The model is demonstrated on Figure 2. The potential general connection between the SQL operations and the different methods of machine learning is discussed for example by [4].

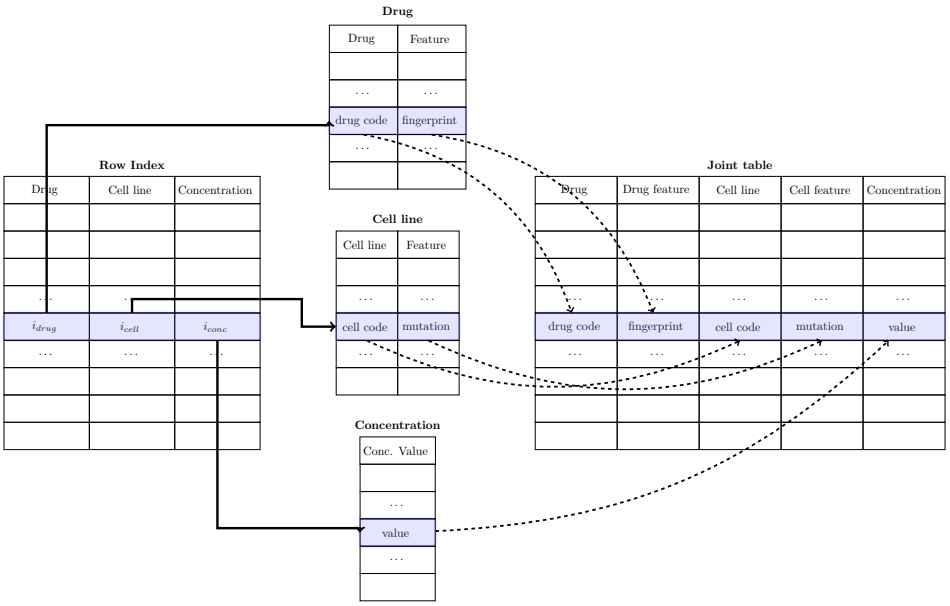


Figure 2: The basic structure of the multiview learning. In this example we have 3 tables, *Drug, Cell line, Concentrations*. The rows are connected via experiments where the *drugs* are tried on certain *cell lines* with given *concentrations*. The index table, *Row links*, corresponds to those experiment. That table contains the indexes of the relating rows in the data tables. Finally we have the joint table representing the relationships between the elements of different tables.

In the LTR system the **Join** operation of the multiview learning is realized on the level of the min-batches. This approach can reduce the memory requirements of the learning method with several magnitudes. For example, in the learning problem of [16] the demand for memory is reduced from $200GByte$ to $2GByte$. The LTR Join operation is demonstrated in Figure

3.

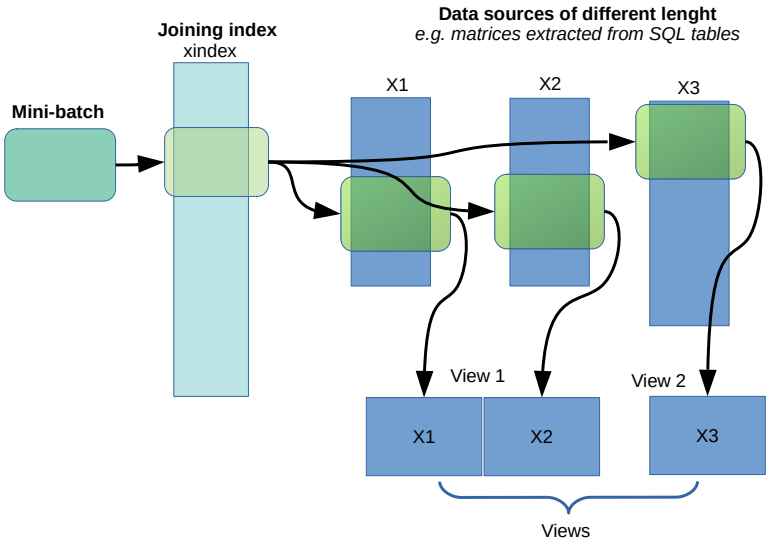


Figure 3: The tables are combined in the level of the mini-batches based on the joining index, **xindex**. The design corresponding this figure is given by $lXtrain = [X1, X2, X3]$ containing 3 tables, and $llinks = [\,[0, 1], 2]$ containing 2 views, the first view is a concatenation of the Table X1 and X2.

5.1 Multiview design

This parametrization of the multiview learning in LTR is valid from version: `ltr_solver_multiview_010.py`

5.2 The parameters

lXtrain, (lXTest):	$[X_1, \ldots, X_{n_x}]$
$X_i, i = 1, \ldots, n_x$:	2D arrays representing the tables
llinks:	$[view, \ldots, view]$
view:	index_x | list_of_view_elements
index_x:	index in the list of lXtrain(lXtest)
list_of_view_elements:	[view_element, ..., view_element]
view_element:	index_x | list_degree
list_degree:	[index_x, degree]
degree:	$1, 2, 3, \ldots$

- The elements of lXtrain have to be 2d arrays, if it is a vector then the shape has to be $(n, 1)$.

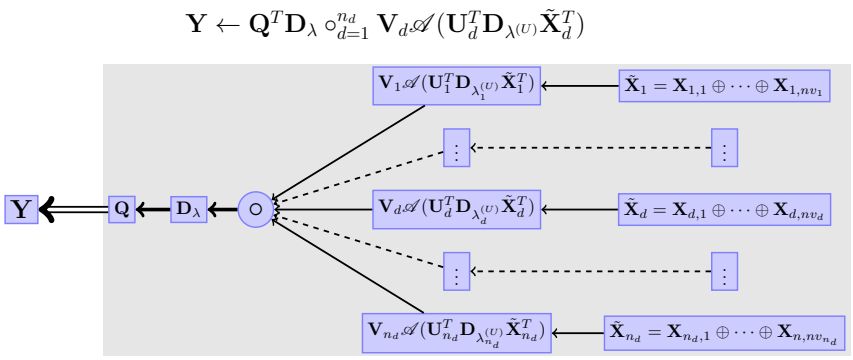


Figure 4: In the LTR system each view is represented as a distinct factor of the polynomial function, and it has its own optimization parameters. The views can also combine several tables via concatenating the rows of the corresponding tables.

- The 2d arrays in the list_of_view_elements are concatenated by the **hstack** function of the **numpy**.
- If the indexes in the list_of_view_elements point to arrays of lXtrain with different length then all arrays in that view truncated to the shortest one.
- If in the view_element the degree is missing then it is set to 1.

5.3 xindex structure

- The array **xindex** is a 2D array assuming a structure similar to the output of the SQL inner join operation.
- **xindex** has as many columns as many views appearing in the **llinks**, and each column corresponds to the views in the same order as they occur in the **llinks**.
- **xindex** rows contain row indexes of the 2d arrays corresponding to the views.
- The rows of **xindex** can contain any combination of the row indexes of the views.

5.4 Degree(order) of the polynomial generated via a design

If the design, **llinks** is given then the maximum degree of the polynomial computed in the LTR is the sum of the degrees of the views appearing in the design. Let the degrees of the views be $d_1, \ldots, d_{n_v}$, where n_v is the number of views, thus the maximum degree is equal to $\sum_{v=1}^{n_v} d_v$. If the homogeneous

setting is true, **ihomogeneous = 1**, which is the default case, then all terms with degree $\sum_{v=1}^{n_v} \tilde{d}_v$, $(d)_v = 0, 1, \ldots d_v$ is also included, see examples below.

6 Multi-view learning via the LTR

The multi-view learning can eliminate the unnecessary interaction in the LTR training. For example interactions within the variables of the *d1, d2, conc1, conc2 and c* are pointless since they are categorical variables. We need those interactions which act between these entities, and the fingerprints and the DTI features.

To implement the multi-view learning is straightforward in the LTR. It requires one more parameter in the training, in the function $fit()$, thus in this case we have

$\mathbf{I}[itrain]$	the indexes can be used to join the data arrays
larrays	the list of data arrays [**D1**, **D2**, **Conc1**, **Conc2**, **C**, **FP**, **PER**]
multiview	is a list of lists of indexes pointing into the list of data arrays, *larrays*, **D1** $\rightarrow 0, \ldots,$ **PER** $\rightarrow 6$ for example: $[[0,5],[1,5],[2],[3],[4,6]]$, where we would like to have interaction between the five views, (D1,FP), (D2,FP), Conc1, Conc2, (C,PER). the pairs like (D1,FP) means that drug 1 is concatenated to the corresponding fingerprints. No interaction is computed within the pairs.

In this setting the LTR parameters, $\mathbf{U}^{(d)}, \mathbf{V}^{(d)}$ are assigned only to the corresponding views and not to all variables.

If we would like to find interactions within one view, then that view needs to be repeated in the list of views, e.g. second order interaction of the finger prints can be realized by this list: $[[0,5],[0,5],[1,5],[2],[3],[4,6]]$

7 Multi-view design examples

Multiview design examples

The design of the multi-view learning:

- There could be a list of tables within each view.
- Interaction is computed between views but not within.
- Every table can appear in any number of views.
- Any view can contain any number of tables.
- Homogeneous representation of the views is used to express all monomials up to the maximum order.

Examples assuming each view contains one variable of these ones,x, y, z, thus we have 3 views. The views are indexed by 0:x, 1:y, 2:z.

Design	Monomials	
	ihomogeneous = 0	ihomogeneous = 1
$[[0, 1, 2]]$	x, y, z	$1, x, y, z$
$[[0], [1], [2]]$	xyz	$1, x, y, z, xy, xz, yz, xyz$
$[[0], [0], [0]]$	x^3	$1, x, x^2, x^3$
$[[0], [0], [1]]$	x^2y	$1, x, y, x^2, xy, x^2y$
$[[0, 1], [1, 2]]$	xy, xz, y^2, yz	$1, x, y, z, xy, xz, y^2, yz$

If the views consist of several variables then the expressions are valid for all variables contained in the views.

8 Training

- The views are concatenated only in the mini-batch level, thus no huge data table is needed, **160GB reduced to 2GB in case of the ISMB data set.**
- Homogeneous coordinates, input and output normalization are moved into the mini-batch related computations. No explicit normalization is needed before the training is called.
- Random perturbation of the gradient to reduce to fall into a local optimum.
- Some constraints on the parameters to reduce the chance of overfitting
  - The projection of the output space is fixed to constants in case of scalar output.

9 Gradient perturbation

With ADAM(Adaptive moment estimation)

$$\boxed{\text{Parameter}} \Leftarrow \boxed{\text{Parameter}} - \boxed{\text{ADAM}} \leftarrow \begin{array}{c}\boxed{\text{Gradient}} \\ \uparrow \\ \boxed{\text{Perturbation}}\end{array} \tag{9}$$

Without the ADAM we might write

$$\begin{aligned} \mathbf{P}_{t+1} &= \mathbf{P}_t - s\left(\nabla f(\mathbf{P})|_{\mathbf{P}_t} + \boldsymbol{\xi}\right), \\ \boldsymbol{\xi} &\sim \mathcal{N}(\mathbf{0}, \mu||\nabla f(\mathbf{P})|_{\mathbf{P}_t}||_2),\ \mu \in (0,1). \end{aligned} \tag{10}$$

- It mimics an Markov Chain Monte Carlo(MCMC) sampling with additional manifold regulation, e.g. Hamiltonian Monte Carlo sampling.
- The perturbation enforces exploration of domain of non-convex nonlinear optimization problem.

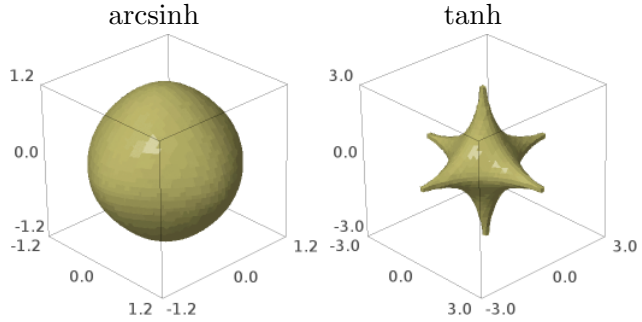
10 Alternative constraints of the design parameters in the optimization

In the optimization algorithm after every iteration the parameters $\mathbf{Q}$, $\mathbf{U}^{(d)}$, $\mathbf{V}^{(d)}$ and $\boldsymbol{\lambda}^{(u)}$ are normalized, see their description in Table 7 of Section 2.3. The matrix represented parameters $\mathbf{Q}$, $\mathbf{U}^{(d)}$ and $\mathbf{V}^{(d)}$ normalized row wise.

Let $\mathbf{r}$ denote a row of any parameter matrix then the default normalization $\mathbf{r} \overset{\text{Project}}{\rightarrow} ||\mathbf{r}||_2 = 1$, thus every row is projected onto a L_2 sphere of radius 1. Some alternative normalizations are available as well

$$\begin{array}{ll} \textbf{norm\_type=0, Deafult} & ||\mathbf{r}||_2 \\ \textbf{norm\_type=1} \ , \ L_1 & ||\mathbf{r}||_1 \\ \textbf{norm\_type=2} \ , \ L_\infty & ||\mathbf{r}||_\infty \\ \textbf{norm\_type=3} & ||\arcsin(\mathbf{r})||_2 \\ \textbf{norm\_type=4} & ||\tanh(\mathbf{r})||_2 \end{array} \tag{11}$$

Some examples:



11 Learning speed

The learning speed is set by this formula in the LTR:

$$\begin{array}{l} t = 0 \\ \text{sigma}_t = \textit{sigma0} \\ \textbf{Do} \\ \quad \textbf{If } \text{modulo}(t, \textit{nsigma}) == 0 : \\ \qquad \text{sigma}_{t+1} = \text{sigma}_t - \dfrac{\text{sigma}_t^2}{\textit{dscale}} \\ \quad t = t + 1. \end{array} \tag{12}$$

with the parameters

sigma0 * : float; the initial learning speed, the parameter **sigma** stores the changing, diminishing learning speed during the optimization procedure (default=0.05)

nsigma : integer; update frequency of the learning speed, (default=10)

dscale : float; scales the learning speed update, (default=2). The greater the value implies slower the decay of the learning speed.

See similar learning speed updates for example in [2].

12 Pinball loss

An additional loss function, the **Pinball loss** can be used in LTR. The Pinball loss is given by the function

$$\ell(z) = \begin{cases} -(1-\alpha)z & z < 0, \\ \alpha z & z \geq 0, \end{cases}$$

where $\alpha \in (0, 1)$ is a parameter which can be used to define confidence intervals for the quantile regression. For example, it could be chosen as 0.05, or 0.95. See further details about the *quantile regression* in [1].

L_1 norm loss can also be expressed as subcase of the Pinball loss of the Quantile Regression by setting $\alpha = 0.5$.

The parameters controling the Pinball loss in the LTR are the following:

iquantile : binary; = 1 Pinball loss for L_1 norm, and quantile regression, = 0 not, (default=1), smoothed L_1 norm regression.

quantile_alpha : float; quantile confidence parameter, e.g. 0.05, 0.95, in case of smoothed L_1 it is 0.5, (default=0.5). See Figure 6.

quantile_smooth : float; smoothing parameter of the quantile, pinball loss, if it is greater then smoother the loss is, if it is smaller then it is closer to the Pinball loss, (default=1). See Figure 5.

12.1 Pinball loss via logistic function

The Pinball loss is not differentiable at 0, Generally, logistic function based approximation is applied to derive a loss with smooth gradient.:

$$\frac{1}{1 + \exp(-t(u - u_0))} - (1 - \alpha), \; u_0 = \frac{1}{t} \log\left(\frac{\alpha}{1-\alpha}\right), \tag{13}$$

where t is a smoothing parameter. $t \to \infty$ leads to original Pinball loss.

12.2 Pinball loss via hyperbola based approximation

In large scale application the exponential function in the logistic approximation of the Pinball loss can be numerically unstable, e.g. overflow error can frequently occur. An alternative, more stable, approach can be applied by using the upper sheet of a hyperbola. It contains only the $\sqrt{()}$ function which is less sensitive on large arguments.

Let us assume the same α parameter of the Pinball loss. The approximation of the Pinball loss can be represented by rotating the upper sheet of the hyperbola around the origin, see Figure 7. To implement that rotation we can use the following parameters. Let $\beta_{Left} = \arctan(1 - \alpha)$, $\beta_{Right} =$

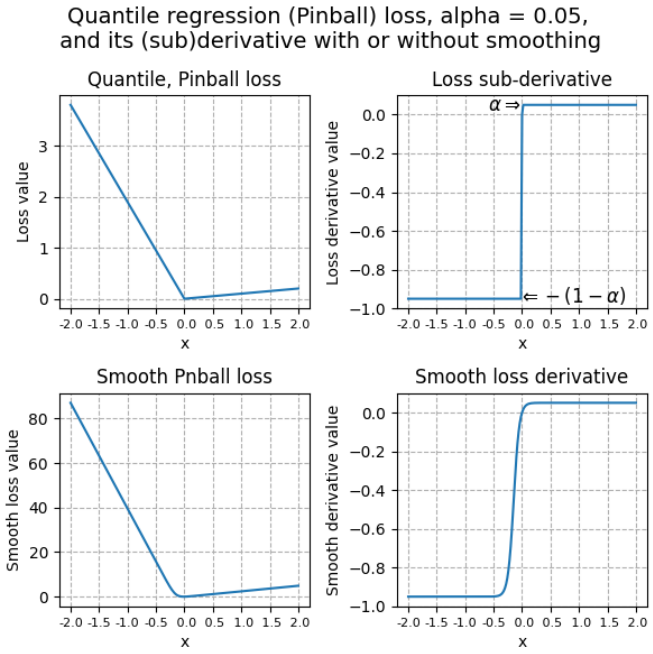


Figure 5: Pinball loss and its smooth approximation

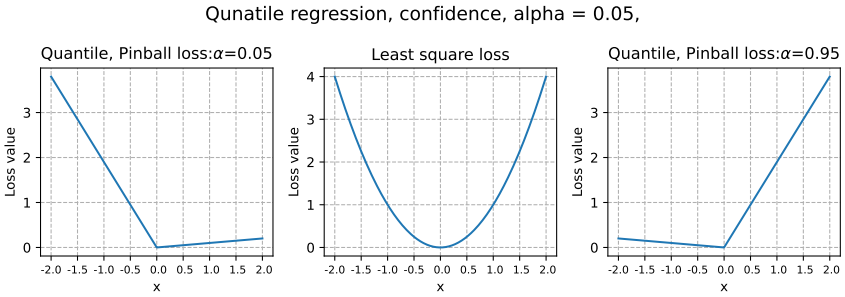


Figure 6: The shapes of the Pinball loss approximation for confidence intervals belonging to $\alpha = 0.05$

$\arctan(\alpha)$. The angle of the axis of the hyperbola relative to the Y coordinate axis after rotation is given by: $\beta_{y_{axis}} = (\pi - \beta_{Left} - \beta_{Right})/2 + \beta_{Right}$, where the rotation angle is $\beta_{\frown} = \beta_{y_{axis}} - \pi/2$, The angle relative to the X axis is $\beta_{x_{axis}} = \beta_{Right} - \beta_{\frown}$, We can apply a smoothing parameter t to adopt the hyperbola.

With these parameter the expression of the hyperbola can be derived.

The approximation of the Pinball loss as a rotated hyperbola is given by

$$y(u) = \frac{Lu + \sqrt{S}}{D}, \tag{14}$$

and the corresponding gradient takes this form

$$\nabla_u y = \frac{L + \frac{\tan(\beta_{shape})^2 u}{\sqrt{S}}}{D}, \tag{15}$$

where we used these shorthand notations, the shape of the hyperbola is given by

$$\beta_{shape} = (\beta_{Right} + \beta_{Left})/2, \tag{16}$$

the denominator looks like this

$$D = \cos(\beta_{\frown})^2 - \sin(\beta_{\frown})^2 \tan(\beta_{shape})^2, \tag{17}$$

the linear term is

$$L = \sin(\beta_{\frown}) \cos(\beta_{\frown})(1 + \tan(\beta_{shape})^2), \tag{18}$$

and the expression in the square root takes this form

$$S = \left(\tan(\beta_{shape})^2 u^2 + Dt^2\right). \tag{19}$$

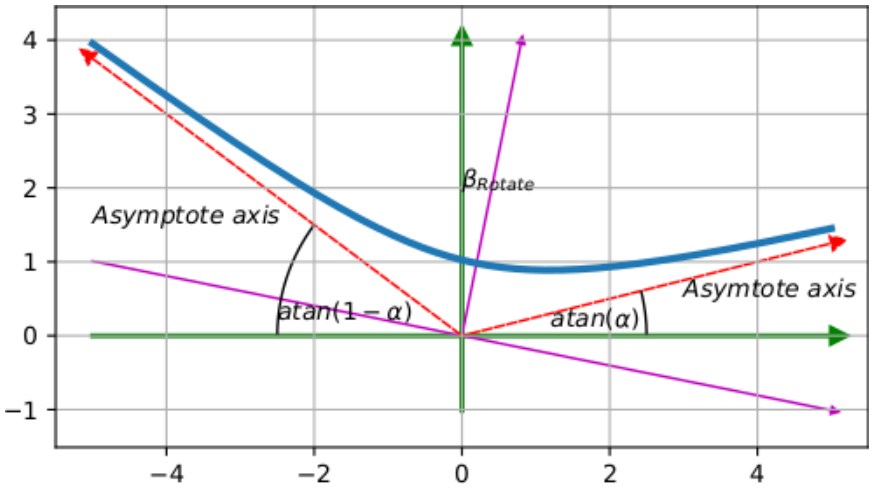


Figure 7:

References

[1] Anastasios N. Angelopoulos and Stephen Bates. A gentle introduction to conformal prediction and distribution-free uncertainty quantification, 2021.

[2] D.P. Bertsekas. *Nonlinear Programming*. Athena Scienctific, second edition, 1999.

[3] Mathieu Blondel, Akinori Fujino, Naonori Ueda, and Masakazu Ishihata. Higher-order factorization machines. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 3359–3367, USA, 2016. Curran Associates Inc.

[4] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. Towards linear algebra over normalized data. *Proc. VLDB Endow.*, 10(11):12141225, aug 2017.

[5] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377387, jun 1970.

[6] Vin de Silva and Lek-Heng Lim. Tensor rank and the ill-posedness of the best low-rank approximation problem. *SIAM J. Matrix Anal. Appl.*, 30(3):1084–1127, September 2008.

[7] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 4th edition edition, 2013.

[8] Heli Julkunen, Anna Cichonska, Prson Gautam, Sandor Szedmak, Jane Douat, Tapio Pahikkala, Tero Aittokallio, and Juho Rousu. Leveraging multi-way interactions for systematic prediction of pre-clinical drug combination effects. *Nature Communications*, 11, 6136, 2020.

[9] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM REVIEW*, 51(3):455–500, 2009.

[10] L. De Lathauwer, B. De Moor, and J. Vandewalle. A multilinear singular value decomposition. *Journal of Matrix Anal. Appl.*, 21(4):1253–1278, 2000.

[11] P.M. Prenter. A Weierstrass theorem for real, separable Hilbert spaces. *Journal of Approximation Theory*, 3:341–351, 1970.

[12] Steffen Rendle. Factorization machines. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 995–1000. IEEE Computer Society, 2010.

[13] Abraham Silberschatz, Henry F Korth, and S Sudarshan. *Database system concepts / Abraham Silberschatz, Henry F. Korth, S. Sudarshan.* McGraw-Hill Higher Education, Boston, 5th ed. edition, 2006.

[14] Sandor Szedmak, Anna Cichonska, Heli Julkunen, Tapio Pahikkala, and Juho Rousu. A solution for large scale nonlinear regression with high rank and degree at constant memory complexity via latent tensor reconstruction. *arXiv:2005.01538*, cs.LG, 2020.

[15] Tianduanyi Wang, Sandor Szedmak, Haishan Wang, Tero Aittokallio, Tapio Pahikkala, Anna Cichonska, and Juho Rousu. Modeling drug combination effects via latent tensor reconstruction. *Bioinformatics*, 37:93–101, July 2021. Publisher Copyright: © 2021 The Author(s). Published by Oxford University Press.

[16] Tianduanyi Wang, Sandor Szedmak, Haishan Wang, Tero Aittokallio, Tapio Pahikkala, Anna Cichonska, and Juho Rousu. Modeling drug combination effects via latent tensor reconstruction. *bioRxiv*, 2021.