Aalto University
School of Science
Master's Programme in Life Science Technologies

Santeri Mentu

Analyzing Performance of Latent Tensor Methods on Large High-Rank Data Sets

Master's Thesis
Espoo, July 31st, 2020

Supervisor:	Professor Juho Rousu
Advisor:	Sandor Szedmak, PhD

Aalto University
School of Science
Master's Programme in Life Science Technologies

ABSTRACT OF
MASTER'S THESIS

Author:	Santeri Mentu		
Title: Analyzing Performance of Latent Tensor Methods on Large High-Rank Data Sets			
Date:	July 31st, 2020	**Pages:**	76
Major:	Bioinformatics and Digital Health	**Code:**	SCI3092
Supervisor:	Professor Juho Rousu		
Advisor:	Sandor Szedmak, PhD		

Linear regression techniques are very popular in the natural sciences thanks to their scalability, well understood mathematical properties, and interpretable parameters. However including multivariate interactions in the form of feature transformation leads to an explosion in the number of model parameters. In general the problem of nonlinear regression with higher order interactions on high dimensional complex data is quite challenging with few good general purpose solutions. Impressive results have been achieved using deep neural networks and kernel methods, but both have significant shortcomings and unsolved issues in certain scenarios.

In this thesis I analyze and compare the efficacy of different latent tensor models in this task. I focus on the recently published method of latent tensor reconstruction which has not yet been extensively studied on large (10^7 examples) real world data sets. I present a new software implementation of this model and experiment with different training configurations to achieve optimal performance. I used both synthetic and real world data to study the characteristics of the model and the limitations of the training procedure.

I used the publicly available NCI-ALMANAC dataset to compare the performance of latent tensor reconstruction to higher order factorization machines which has previously been used to achieve state of the art performance in this task, as well as deep neural networks. Latent tensor reconstruction was able to exceed the predictive accuracy of higher order factorization machines, but deep neural networks achieved the highest performance overall. The results are promising in terms of latent tensor reconstruction achieving accuracy on par with deep neural networks with further development.

Keywords: machine learning, latent tensor methods, drug development

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Master's Programme in Life Science Technologies

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Santeri Mentu		
Työn nimi: Latent Tensor Reconstruction -menetelmien käyttö korkeaulotteisen ja korkearankisen datan mallinnuksessa			
Päiväys:	31. heinäkuuta 2020	**Sivumäärä:**	76
Pääaine:	Bioinformatics and Digital Health	**Koodi:**	SCI3092
Valvoja:	Professori Juho Rousu		
Ohjaaja:	Tohtori Sandor Szedmak		

Lineaarinen regressio on luonnontieteissä yleisesti käytetty menetelmä sen hyvin kartoitettujen matemaattisten ominaisuuksien ja helposti tulkittavien parametrien ansiosta. Muuttujien välisten interaktioiden mallintaminen piirretransformaatioiden avulla johtaa parametrien määrän räjähdysmäiseen kasvuun. Epälineaarinen regressio korkea-asteisilla yhteisvaikutuksilla on yleisesti ottaen vaikea haaste, johon ei ole hyviä yleisiä ratkaisuja. Syviä neuroverkkoja ja kernelmenetelmiä käyttämällä on saavutettu vaikuttavia tuloksia, mutta molempien lähestymistapojen käytössä on puutteita tietyissä skenaarioissa.

Tässä diplomityössä analysoin ja vertailen latentteihin tensoreihin pohjautuvien mallien toimivuutta edellämainituissa sovelluksissa. Keskityn vastikään julkaistuun latent tensor reconstruction -menetelmään, jonka käyttöä suurilla dataseteillä (10^7 datapistettä) ei ole toistaiseksi tutkittu yksityiskohtaisesti. Esittelen uuden ohjelmistototeutuksen tästä mallista ja suoritan kokeita eri konfiguraatioilla optimaalisen suorituskyvyn aikaansaamiseksi. Olen käyttänyt sekä synteettistä että oikeista lähteistä peräisin olevaa dataa mallin ominaisuuksien ja koulutusmenetelmän rajojen kartoittamiseen.

Käytin julkisesti saatavilla olevaa NCI-ALMANAC tietoaineistoa latent tensor reconstruction -menetelmän ja aiemmin parhaan tuloksen saavuttaneiden korkea-asteisten faktorointimenetelmien vertailuun. Myös syvä neuroverkkomalli on sisällytetty vertailuun. Latent tensor reconstruction suoriutui paremmin kuin korkea-asteiset faktorointimenetelmät, mutta ei yltänyt syvien neuroverkkojen tasolle ennustustarkkuudessa. Tulokset ovat lupaavia sen kannalta että latent tensor reconstruction -menetelmällä voitaisiin saavuttaa tulevaisuudessa syviä neuroverkkoja vastaava tarkkuus.

Asiasanat:	koneoppiminen, tensorimenetelmät, lääkekehitys
Kieli:	Englanti

Acknowledgements

I want to give special thanks to my thesis advisor Sandor Szedmak for his extensive feedback, advice, and support, sometimes on rather short notice. This work as it is would not been possible without his patient guidance. I also want to thank professor Juho Rousu, Tiandianyui Wang, and the other members of the Aalto University KEPACO group for sharing their knowledge and expertise. I'm also very grateful for my girlfriend Ziyi for her patience and understanding during the long and arduous process of writing the thesis.

Helsinki, July 31st, 2020

Santeri Mentu

Terms and Notation

Symbols

$x \in \mathbb{R}$	scalar
$\mathbf{x} \in \mathbb{R}^n$	vector
$\mathbf{X} \in \mathbb{R}^{n \times m}$	matrix
$\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_m}$	tensor

Operators

$\langle \cdot, \cdot \rangle$	vector inner product
$\circ$	Hadamard product
$\otimes$	tensor outer product
$\|\| \cdot \|\|$	induced norm in a Hilbert space $\mathcal{H}$
$\mathrm{tr}(\cdot)$	trace of matrix
$\mathrm{vec}(\cdot)$	column-wise vectorization of matrix

Conventions

n	dimension of the feature space
m	number of data observations
t	index referring to a specific rank of a tensor
n_t	rank of a tensor decomposition
n_d	maximum degree of interactions included in a model

Contents

Chapter 1

Introduction

With the rapid development of information technology, there is an ever increasing volume of complex data. Making use of this data necessitates the development of analysis methods which can effectively capture and model its structure and properties. The end goal is to convert data into knowledge and insights that allow us to learn about the systems under study, as well as predict unseen observations.

One of the most important application areas of these data analysis techniques is biomedical and health informatics. An unprecedented wealth of data is becoming available thanks to high-throughput sequencing technologies, wearable sensors, digital imaging, electronic health records, and national and international biobanks [4]. Transforming these data into biological and diagnostic information is a serious challenge, since the data is often heterogeneous, high dimensional, and contains multivariate interactions. Often it is not enough to simply model the data or produce accurate predictions, but we would also like to be able to gain insights from the trained model. [35]

Currently deep neural networks are popular machine learning models to use for modelling and making predictions about complex data and have achieved impressive results in this task in biology and medicine [36]. However they are famously difficult to interpret [58] and often vulnerable to adversarial attacks [46]. Due to these reasons and other limitations, such as problems in dealing with small datasets, high sparsity, and low quality data, there is much interest in developing other models with similar performance but more transparent parameters [12]. Kernel methods have been found to be very successful in dealing with some of these shortcomings, especially learning small data sets, but many of them do not scale well in their basic form [48].

The research objective of this thesis is the comprehensive analysis and comparison of machine learning methods in the problems of regression for large high-rank data sets. This kind of modelling can be applied in many situations in the biological sciences as well as other domains. I test and compare these methods using synthetic data as well as a real world data set that exhibits the aforementioned properties. This thesis is in part a continuation of earlier efforts to apply latent tensor methods to drug combination prediction by Heli Julkunen in her Master's thesis. Julkunen found that higher-order factorization machines (HOFM) achieved state-of-the art performance in predicting the effectiveness of drug combination treatments [29], and my central objective is to investigate whether the recently published latent tensor reconstruction (LTR) [53] can further improve these results owing to it's more expressive model space.

I give a short general introduction to machine learning and the training of machine learning methods using empirical risk minimization. Later I give an extensive overview of the most common models used for linear and nonlinear regression on high-dimensional data. I compare their design and empirical performance characteristics based on the literature. I give a detailed description of the theory and implementation of different latent tensor methods and their unifying connection to tensor decompositions. Included is a brief introduction to the mathematics of tensors to the extent that it is relevant for the reader to understand the theory behind the models. I go through the design of the recently published latent tensor reconstruction algorithm [53] and compare it to other latent tensor methods. In addition to this I discuss several non-tensor based machine learning techniques, such as kernel methods and deep neural networks. I highlight the situations where existing models do not produce adequate results and thus new approaches are necessary.

As part of this thesis I also present a new software implementation of Latent Tensor reconstruction that supports hardware acceleration with GPUs. I demonstrate significant performance increases compared to training the same model on CPUs. The modular design of this implementation is utilized to compare different heuristics, optimizers and training strategies in order to find the optimal configuration for the model. I analyze both the predictive performance of this model as well as the computational efficiency of this implementation.

The experiments are split into finding the optimal configuration my implementation of latent tensor reconstruction and comparisons to other models, namely higher order factorization machines and deep neural networks. I conclude by reviewing my results and discussing possible avenues for further research.

Chapter 2

Literature Review

2.1 Applications and challenges of nonlinear regression

Applications for algorithms that can efficiently learn high dimensional nonlinear interactions from high-dimensional data exist in a variety of fields. A much studied and well known example of are recommender systems, which aim to predict a score of how well a given item matches the preferences of a user [13]. In the natural sciences examples include predicting the properties of molecules [21], propeties of metamaterials [31], and efficacy of combination treatments [29].

Neural networks are a very popular method for nonlinear regression and classification tasks, due to their expressive power and the development efficient training algorithms. Neural networks are often used as a black box model for nonlinear regression, because they are difficult to interpret and identifying which predictors are relevant to the output is challenging [58].

In applications such as medical diagnostics the interpretability and robustness of models is often at least as important as the prediction accuracy, so there exists a clear need to develop models that are able to model complex nonlinear interactions, but also be interpretable in their parameters, and ideally have verifiable guarantees on convergence and robustness. [46]

Another challenge is the so-called curse of dimensionality. In the context of regression this means that the number of training examples required to train a regression grows exponentially with the dimensionality of the data. The only way to overcome this is to introduce restrictions to the class of models considered by the training algorithm. This is implicitly done by almost all multivariate estimation procedures, including neural networks. [24]

The aforementioned application areas also often involve large data sets.

This imposes its own demands on learning algorithms, namely that they should scale favorably with the number of training data. Notably this does not hold for kernel ridge regression, which can very efficiently model some complex nonlinear interactions, but cannot be used without on large data sets without significant modification. [48]

2.2 Machine learning

Machine learning is the general term to describe approaches where a model is shown example data to make it learn the patterns present in it. This is in contrast to traditional computer algorithms, where the programmer explicitly defines rules and procedures that are used to solve a problem or perform a task. In machine learning the idea is that if we are able to design a procedure for fitting a model to our data, then we can leave the machine to figure out the specifics and, hopefully, to find patterns that would be hard or impossible for a human to discover. The core objective of machine learning research is to develop algorithms that can generate accurate predictions for unseen observations and do this efficiently and robustly [39].

From a theoretical perspective machine learning is an application of statistical theory since inferences are made using a finite sample of observations. We can use statistical theory to make statistical convergence guarantees and derive optimal learning procedures, but in many instances optimal configurations for learning algorithms are found empirically. The no free lunch theorem states that no machine learning algorithm performs for all data, and that some prior knowledge needs to be incorporated into the structure of the model and learning algorithm in order to achieve superior results [49]. Thus in machine learning research the model and data to which it is applied to are always interlinked.

We can categorize machine learning based on the training procedure used. These include supervised learning, unsupervised learning, on-line learning, and reinforcement learning. Supervised learning means that in the training phase the learner is given examples as well as target labels, which it attempts to reproduce as accurately as possible. In unsupervised machine learning the goal is to infer the structure or shape of the data distribution and in semi-supervised learning the learner receives both labeled and unlabeled data, and the goal is to improve model performance by supplementing the labeled data with unlabeled examples. In online learning and reinforcement learning the training and evaluation phases are intermixed and the model attempts to maximize a reward function based on the information gathered during each cycle. [39]

Machine learning models on the other hand can be divided into parametric and non-parametric. Parametric modelling assumes that the sample drawn from a parametric distribution. Incorporating this assumption to our model allows us to estimate this finite set of parameters from on the data and once the parameters have been established, the full distribution is defined and we can use it to make predictions. However the assumption that the model is valid over the input space needs to be verified separately to ensure the reliability of predictions.

Non-parametric modelling takes a different approach, assuming instead that similar inputs have similar outputs. Predictions for new data are made by finding past observations that are similar and then interpolating the target value based on proximity. The method of interpolation is the main differentiating factor between nonparametric modelling methods. The main advantage of this approach is that the only assumption we have to make about the data distribution is that it conforms to some smoothness criterion, but the downside is that computing proximity to known observations becomes increasingly expensive as the number of training examples grows. Thus nonparametric methods are generally poorly suited for modelling very large data sets. [2]

Parameters that control some aspect of the training procedure are called hyperparameters [39]. Common ones are the size of individual parameter updates, often called the learning rate, what kind of a sample is used for making each update, and various features specific to the optimization method, such as the strength of possible momentum terms.

2.3 Training machine learning models

For parametric models the task of training a model consists of finding parameters that maximize the accuracy measure being used. For simple models such as linear regression it is possible to do this analytically by minimizing mean squared error using matrix algebra. For complex models and large datasets however, it is generally unfeasible or needlessly labor intensive to find formulas for analytical solutions. In these cases we generally turn to stochastic optimization. This involves evaluating the model on a sample of the data, and using numerical optimization methods to update the parameters in a way that improves model performance.

Several types of optimization methods are used for machine learning, such as stochastic gradient descent, coordinate descent, and higher order optimization methods like Newton's method. Higher order methods are unfeasible to use with large data sets due to their computational complexity, so they are excluded here. Stochastic gradient descent and it's extensions, such as Adam, are very popular in science and engineering, since they only require only the computation of first order derivatives and are therefore relatively efficient for minimizing differentiable objective functions [30]. They have also been instrumental in the rise to popularity of deep learning methods.

2.3.1 Empirical risk minimization and overfitting

The formal term to describe the aforementioned paradigm is empirical loss minimization (ERM). The components of this approach are the training set S, the unknown true distribution $\mathcal{D}$, the loss function ℓ, and the predictor $h_S : X \to Y$. For supervised learning the goal is to find an predictor h that minimizes the error between the the predictions and true values. Since we cannot access D, we use an error measure on the training set as a proxy for the accuracy of the predictor. This error L_S is commonly called the empirical error or empirical risk, giving this paradigm its name. [49]

Formally a loss function is a function $\ell : \mathcal{Q} \times Z \to \mathbb{R}_+$, where $\mathcal{Q}$ is any set that plays a role in our hypotheses or models. For general prediction problems $Z = \mathcal{X} \times \mathcal{Y}$. We then define the risk function as the expected loss over the true distribution

$$L_D(h) := \mathbb{E}_{z \sim \mathcal{D}}[\ell(h, z)].$$

The empirical risk over the training data is defined as the sample version of this

$$L_S(h) := \frac{1}{m} \sum_{i=1}^{m} [\ell(h, z_i)].$$

The loss function most commonly used in regression is the square loss

$$\ell_{\mathrm{sq}}(h, (x, y)) := (h(x) - y)^2.$$

The empirical risk with this loss function is called the mean squared error (MSE)

$$\mathrm{MSE}(h) := \frac{1}{m} \sum_{i=1}^{m} (h(x) - y)^2. \tag{2.1}$$

The main problem with ERM is the possibility of overfitting. This occurs when the model has the capacity to learn properties of the training set S that are not true for the distribution $\mathcal{D}$, but are instead a result of the finite sampling or noise. An example of overfitting in the case of polynomial regression is presented in figure 2.1. Many different theoretical and heuristic approaches exist for preventing overfitting. One of the most common is to restrict our hypothesis space. Choosing an appropriate restriction or hypothesis class is a common problem, and solving it generally requires some domain knowledge. [49]

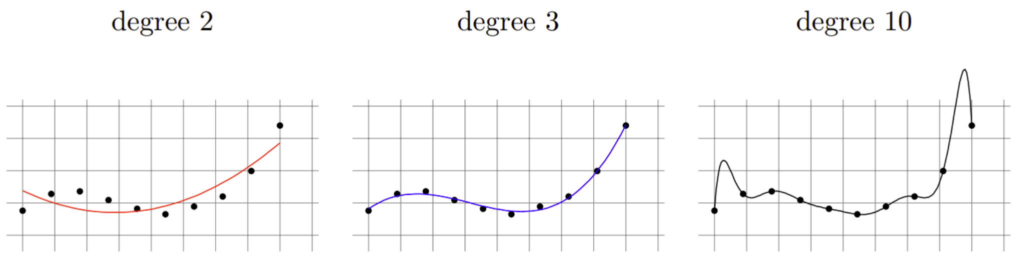

Figure 2.1: Example of underfitting and overfitting a polynomial to a small data set by Bishop [7].

Convex learning problems are important family of learning problems, mainly because many problems that can be learned efficiently fall into it. To define convex learning problems we first have to define convex sets and convex functions.

Definition 1. *A set C in a vector space is convex if for any two vectors $\boldsymbol{u}$ and $\boldsymbol{w}$ the line segment connecting them is contained in the set, meaning that for every $\alpha \in [0, 1]$ we have $\alpha\boldsymbol{u} + (1 - \alpha)\boldsymbol{v} \in C$.*

Definition 2. *Let C be a convex set. Then a function $f : C \to \mathbb{R}$ is convex if for every $\alpha \in [0, 1]$ it holds that*

$$f(\alpha\boldsymbol{u} + (1 - \alpha)\boldsymbol{v}) \leq \alpha f(\boldsymbol{u}) + (1 - \alpha)f(\boldsymbol{v}).$$

An important and useful property of convex functions is that every local minimum is also a global minimum. Another property that will become relevant in the next section is that if f is differentiable, then we can go against the gradient of the function to reach parameters which give decrease the value of the function. A convex learning problem is defined as

Definition 3. *A learning problem is called convex if the hypothesis class $\mathcal{H}$ is convex, and for all $z \in X$, the loss function $\ell(\cdot, z)$ is a convex function.*

If ℓ is a convex function and the hypothesis class is convex, then the ERM problem is a convex optimization problem. [49]

An alternative to restricting the hypothesis space is to introduce a second term to the target function that penalizes extreme parameter values. This is called regularization, a common example of which is Tikhonov regularization, also known as ridge regression. The objective then becomes

$$\arg\min_{\boldsymbol{w}}(L_S(\boldsymbol{w}) + R(\boldsymbol{w})),$$

where $\boldsymbol{w}$ is the parameter vector and R is the regularization term. A simple and popular regularization method is to use the ℓ_2 norm on the parameters. Parameterizing our hypotheses with a parameter vector w we can write the objective function here as

$$\arg\min_{\boldsymbol{w}}(L_S(\boldsymbol{w}) + \lambda \left\|\boldsymbol{w}\right\|^2),$$

where λ is a hyperparameter controlling the strength of the regularization. [49]

Preventing overfitting can also be applying some heuristic during the learning process. Commonly used heuristics are early stopping [39], injecting noise into the target [22], and dropout for neural networks [22].

2.3.2 Stochastic gradient descent

As mentioned in the introduction to this chapter, it is often not possible to minimize empirical risk analytically. Gradient descent is an iterative algorithm where we minimize a differentiable objective function $L_{\mathcal{D}}$ by moving the parameters in a direction opposite the gradient of the objective function. We can imagine a ball on an uneven surface that always moves down in the direction of steepest descent. Denoting the parameter vector as $\boldsymbol{w}$, we have the gradient

$$\nabla_{\boldsymbol{w}} L_D := \left(\frac{\partial L_D(\boldsymbol{w})}{\partial w_1}, \ldots, \frac{\partial L_D(\boldsymbol{w})}{\partial w_d}\right)$$

A single step of the gradient descent algorithm looks like

$$\boldsymbol{w} = \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} L_D(\boldsymbol{w}), \tag{2.2}$$

where X is the training data. η is a hyperparameter called step size in mathematical optimization and often learning rate in machine learning research. The parameter update is computed across the full set of observations. Again since we do not know the true data distribution, we use the gradient computed on the training set S as a proxy. . In practice we often don't want to use the full data set for computing each update, particularly when it is too large to fit in memory. Stochastic gradient descent (SGD) computes the gradient and the update using a single observation. This works as long as the expected value of the gradient on the test set is in the same direction as the true gradient [49]. The parameter update becomes

$$\boldsymbol{w} := \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} J(x_i; \boldsymbol{w}), \tag{2.3}$$

where x_i is the ith datum point. The advantages of stochastic gradient descent are that it requires very little memory and that updates to parameters happen quickly and frequently. The basic gradient descent algorithm consistently moves towards a local minimum of the objective function on the training set, but convergence is generally slow and inefficient, since the gradient is computed for multiple similar observations before the parameters are adjusted. Figure 2.2 shows typical behaviour of the loss function over time for gradient descent and stochastic gradient descent.

Since parameter updates are made based on single observations, the updates performed by stochastic gradient descent have high variance and can easily overshoot. This is desirable in the case of non-convex objective functions, since the learner can be disturbed away from local minima into finding better ones, although a clear downside is that it might not converge to an exact optimal solution. This can be remediated by reducing the step size according to some rule as the model approaches convergence.

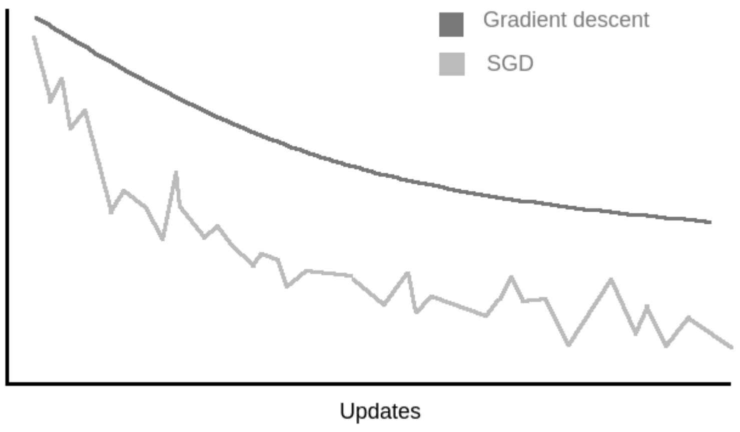

Figure 2.2: Example of how the objective function behaves compared between gradient descent and stochastic gradient descent.

Mini-batch gradient descent is a compromise between gradient descent and SGD, where each update uses a subset of the full training set, commonly called a mini-batch. The parameter update for each mini batch X_i is

$$\boldsymbol{w} = \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} J_i(X_i; \boldsymbol{w}). \tag{2.4}$$

This achieves lower variance compared to SGD but with significantly lower memory requirements than that of full-batch gradient descent. In practice this method significantly exceeds SGD in speed, since observations can be retrieved from memory in batches which reduces overhead, and for many machine learning methods we can formulate batch updates as matrix operations that are very efficiently implemented on modern GPUs or TPUs. Mini-batch sizes typically vary between 50 and 250 [47]. In deep learning applications it is not uncommon to have batch sizes of 500 or more since this typically leads to faster per-epoch times, although some recent findings suggest that smaller batch sizes such as 32 lead to a wider range of suitable learning rates and lead to better generalization performance [37].

Vanilla mini-batch training has several issues. Firstly choosing a good learning rate can be difficult, and having a fixed learning rate during the whole optimization procedure is probably not optimal, since we want large updates in the beginning and smaller ones towards the end. Furthermore all parameters are updated using the same learning rate, which can significantly hinder training when different parameters are updated with different frequencies. Finally it is difficult for mini-batch SGD to escape saddle points of the target function. [47]

2.3.3 Momentum methods

Momentum methods are extensions of SGD that attempt to fix some of its shortcomings by using an additional momentum term to the update formula to incorporate information about previous updates. The basic idea is to accelerate learning when consecutive gradients are small but pointing in the same direction, and slow it down when there is oscillation in the gradient. A physical analogy for this is to think of a ball with mass rolling towards a lower area, but having a certain amount of mass causing it to pick up or lose momentum depending on the geometry of the surface. [41]

The simplest formulation of this is

$$\begin{aligned} \boldsymbol{v}_t &= \gamma \boldsymbol{v}_{t-1} + \eta \nabla_{\boldsymbol{w}} L_S(\boldsymbol{w}) \\ \boldsymbol{w} &= \boldsymbol{w} - \boldsymbol{v}_t, \end{aligned} \tag{2.5}$$

where v_t is the velocity of the parameter update, and γ is a hyperparameter that controls how much weight the previous updates are given. 0.9 is a typical value for γ [47].

AdaGrad is a gradient based optimization method that uses adaptive learning rates to improve performance when some parameters are updated infrequently, for example in the case of sparse data [16]. It does this by scaling each coordinate by a sum of squared past gradient values, which reduces gradients for frequently updated parameters and relatively increases the size of updates for sparsely updated ones. The update rule for AdaGrad is

$$\boldsymbol{w}_i^{(t+1)} = \boldsymbol{w}_i^{(t)} - \frac{\eta}{\sqrt{\boldsymbol{G}_t + \epsilon}} \cdot \boldsymbol{g}^{(t)}, \tag{2.6}$$

where $\boldsymbol{G}$ is a diagonal matrix where each diagonal element is the sum of the squares of the gradients of $\boldsymbol{w}_i$ up to the time step t, and ϵ is a small number to prevent division by zero. [47]

AdaGrad has been shown to greatly improve performance on sparse optimization problems compared to SGD [17], but under-performs in deep learning tasks [57]. The primary issue with AdaGrad is the accumulation of the gradients in the denominator which eventually shrinks the learning rate so much that the learner is no longer able to acquire new information [47].

Adaptive moment estimation (Adam) is currently one of the most popular optimizer for training deep neural networks [18]. The goal behind Adam is to have an adaptive learning rate for each variable, that is invariant to scaling of the gradients. It is primarily based on the earlier adaptive optimization methods AdaGrad and RMSProp. Unlike AdaGrad, the learning rate does not accumulate gradients from the duration of the training, but instead uses a moving average of previous updates. [30]

Adam stores an exponential moving average of the gradient and the square of the gradient, which are then used as estimates of the 1st moment (mean) and 2nd moment (variance) of the gradient. These estimates are corrected to eliminate the bias resulting from them being initialized as zeros. The momentum updates are

$$\begin{aligned} \boldsymbol{m}_t &= \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t \\ \boldsymbol{v}_t &= \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2, \end{aligned} \tag{2.7}$$

where β_1 and β_2 are hyperparameters controlling the contribution of the current gradient. The bias correction formula is

$$\begin{aligned} \hat{\boldsymbol{m}}_t &= \frac{\boldsymbol{m}_t}{1-\beta_1^t} \\ \hat{\boldsymbol{v}}_t &= \frac{\boldsymbol{v}_t}{1-\beta_2^t}. \end{aligned}$$

Finally the actual parameter update is

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon} \hat{\boldsymbol{m}}_t, \tag{2.8}$$

where η is the learning rate and ϵ is a small constant meant to prevent numerical instability. It is easy to see from equations (2.7) and (2.8) that any diagonal scaling of the gradient is canceled out by the moment terms.

The original Adam paper by Kingma provides only a rudimentary analysis of the convergence properties of the method and the proof provided has been questioned in other research. Many papers have been published about the topic since the original Adam paper, with different approaches to finding convergence guarantees. Défossez et al. (2020) give a proof that Adam has convergence rate $\mathcal{O}(\ln(m)/\sqrt{m})$ with learning rate $1/\sqrt{m}$ smooth gradients and an almost sure uniform bound on the ℓ_∞ norm of the gradients. [18]

2.3.4 Nesterov accelerated gradient

Nesterov accelerated gradient (NAG) uses the momentum term to compute the gradient using effectively the parameter values at the next step of the optimization procedure. This can be expressed as

$$\begin{aligned} \boldsymbol{g}_t &= \nabla_{\theta_t} J(\boldsymbol{w}_t - \gamma v_{t-1}) \\ \boldsymbol{v}_t &= \gamma \boldsymbol{v}_{t-1} + \eta \boldsymbol{g}_t \\ \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \boldsymbol{v}_t. \end{aligned} \tag{2.9}$$

The proposition is that this allows the algorithm to anticipate changes in direction that are ahead, and thus prevent the parameter updates from going too fast in one direction. The description of NAG given by equation (2.9) is easy to understand, but the algorithm can be made more efficient and easier to implement by changing the order of operations. In the basic version we need to apply the momentum step twice, once to compute the look-ahead gradient, and another time to perform the parameter update. In his doctoral thesis on training recurrent neural networks, Ilya Sutskever proposes that the look-ahead momentum is applied in the third step directly into the parameter update [52]. Thus the full computation becomes

$$\begin{aligned} \boldsymbol{g}_t &= \nabla_{\theta_t} J(\theta_t) \\ \boldsymbol{v}_t &= \gamma \boldsymbol{v}_{t-1} + \eta \boldsymbol{g}_t \\ \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - (\gamma \boldsymbol{v}_t + \eta \boldsymbol{g}_t). \end{aligned} \tag{2.10}$$

This version is less intuitive than previous one, but the logic can be seen more clearly by expanding the parameter update in (2.5), which gives

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - (\gamma \boldsymbol{v}_{t-1} + \eta \boldsymbol{g}_t).$$

We can now see that in (2.5) we are using the previous momentum and current gradient to update our parameters, whereas in (2.10) we are in effect using the current momentum vector. Sutskever also provides empirical evidence that this algorithm outperforms SGD and traditional momentum in traditionally difficult optimization tasks [52].

Timothy Dozat presents an adaptation of Adam that applies Nesterov acceleration to improve speed of convergence and quality of learned models [15]. Expanding the Adam parameter update (2.8) gives us

$$\begin{aligned}\boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_t}+\epsilon}\left(\frac{\beta_1 \boldsymbol{m}_{t-1}}{1-\beta_1^t} + \frac{(1-\beta_1)\boldsymbol{g}_t}{1-\beta_1^t}\right) \\ &= \boldsymbol{w}_t - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_t}+\epsilon}\left(\beta_1 \hat{\boldsymbol{m}_{\boldsymbol{t-1}}} + \frac{(1-\beta_1)g_t}{1-\beta_1^t}\right).\end{aligned}$$

We can then simply replace the bias-corrected momentum term of the previous step $\hat{m}_{t-1}$ with the bias corrected momentum term of the current update, which gives the Nadam update formula

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_t}+\epsilon}\left(\beta_1 \hat{\boldsymbol{m}}_t + \frac{(1-\beta_1)\boldsymbol{g}_t}{1-\beta_1^t}\right). \tag{2.11}$$

2.4 Tensors

To begin the treatment of tensors I refer to the definition of tensors given by Bocci and Chiantini [10].

Definition 4. *A tensor $\mathcal{T}$ over a field K is a multidimensional table of elements of K where each element is determined by a multi-index $(i_1, \ldots, i_n)$. Alternatively a tensor can be defined as a multi-linear map*

$$\mathcal{T} : K^{n_1} \times \ldots \times K^{n_m} \to K.$$

This thesis deals exclusively with tensors over $\mathbb{R}$ with the standard basis. The order of a tensor is the number of indices needed to specify an element in the tensor. Thus a first order tensor is a vector, a second order tensor is a matrix and so on. A fiber of a tensor is equivalent to a row of a matrix, and is defined by the set of tensor elements where all indices except one are fixed. Slices are likewise two-dimensional sections of a tensor, where all but two indices are constant. There are several different tensor products, and we give definitions for those that are relevant for the next sections.

Definition 5. *The inner product of two tensors of equal size and shape is defined as the sum of the products of their elements*

$$\langle \mathcal{X}, \mathcal{Y} \rangle := \sum_{i_1, i_2, \ldots, i_n} x_{i_1, i_2, \ldots, i_n} y_{i_1, i_2, \ldots, i_n} \tag{2.12}$$

Definition 6. *The Hadamard product is an elementwise product of two tensors of equal size and shape. It is denoted by the circle operator* $\circ$.

Definition 7. *The outer product of vectors is a tensor where each element is the product of corresponding vector elements. If* $\mathcal{X} = a^{(1)} \otimes a^{(2)} \otimes \cdots \otimes a^{(n)}$, *then*

$$x_{i_1,i_2,\ldots,i_n} = a^{(1)}_{i_1} a^{(2)}_{i_2} \cdots a^{(n)}_{i_n}$$

Thus we can build of arbitrary order and shape as an outer product of vectors. Figure 2.3 gives an example of this for a three-way tensor. This ties closely into the definition of the rank of a tensor and the rank decomposition

Definition 8. *An n-way tensor is rank-one if it can be written as an outer product of n vectors.*

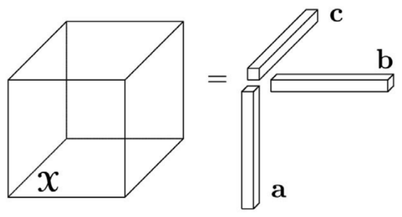

Figure 2.3: A pictorial representation of a three-way rank-one tensor as the outer product of tree vectors by Kolda et al. [33]

A special class of tensors that will become relevant later due to its connection to symmetric polynomials is what we refer to as symmetric tensors.

Definition 9. *A tensor is symmetric if all elements are constant under permutation of the indices.*

Several tensor decompositions exist, many of which can be seen as generalizations of matrix decompositions. The relevance of these tensor factorization machines in the context of latent tensor methods is explained in section 2.5.7. The tensor rank decomposition, also commonly called CANDECOMP or PARAFAC, factorizes a tensor into a sum of rank-one tensors. This decomposition provides the definition of rank for tensors.

Definition 10. *The rank of a tensor* $\mathcal{X}$ *is the smallest number of rank-one tensors that are required to generate* $\mathcal{X}$ *as their sum.*

Figure 2.4 demonstrates a rank-three decomposition of a three-way tensor. A compact way of representing rank decompositions is to collect the rank-one components from each rank into so-called factor matrices. Furthermore

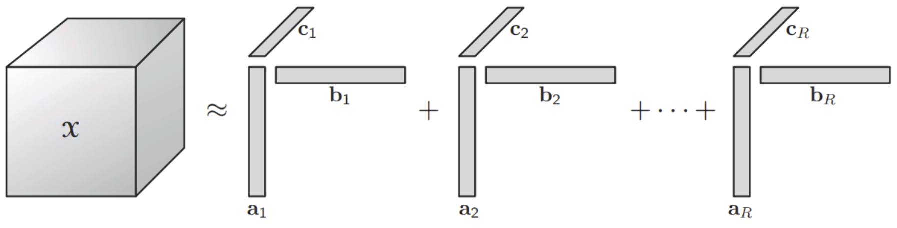

Figure 2.4: A pictorial representation of rank-decomposition by Kolda et al. [33].

we can normalize the columns of the factor matrices to one and absorb the scaling into a weight vector λ without loss of generality. For example we can denote the factorization shown in figure 2.4 as

$$\mathcal{X} \approx [\![\lambda; \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}]\!] = \sum_i \lambda_i \, a_i \otimes b_i \otimes c_i, \tag{2.13}$$

where $\boldsymbol{A}, \boldsymbol{B}$ and $\boldsymbol{C}$ are the factor matrices.

In contrast to many matrix decompositions, the rank decomposition of many high rank tensors is in fact unique [33]. For matrices it holds that the best rank-k approximation is given by the first k-factors of the support vector decomposition [19], but this does not generally hold for tensors of higher order [34]. What this means is that searching for an optimal rank decomposition rank-wise will not lead to the best possible model. It is also possible that a best possible rank-k approximation may not even exist, in what's known as degeneracy [33]. A tensor is degenerate if it can be approximated arbitrarily well by a lower rank factorization.

These concepts have practical relevance when developing algorithms for computing rank-decompositions. Most procedures for computing rank-decompositions compute decompositions of different rank until sufficient accuracy is reached [33]. This approach is quite inefficient since it generally requires us to compute many useless decompositions of lower rank before arriving at a satisfactory one. Furthermore since many decompositions can be approximated arbitrarily closely with lower ranks, we cannot say that we have found the true rank of the tensor even if the error is small. Another complicating factor is that in the presence of noise the rank of the original data may not give the best accuracy for a rank-decomposition. Thus real world data is often noisy, so the rank of the underlying tensor cannot be ascertained in this way.

Alternating least squares (ALS) can be used for computing rank decompositions of a given rank. This method was introduced by Carroll and Chang [11] in their paper on decomposing psychometrics tensors. The general procedure for computing rank decompositions with ALS is to fix all but one of the factor matrices and optimize that one by the use of tensor algebra.

Tensor decomposition is fundamentally different from the tensor reconstruction type problems that are at the core of latent tensor based machine learning methods, but some of the insights from research into the former can be leveraged when developing and analyzing the latter. I next introduce factorization machines and their higher order extension, before going through an unifying reformulation that will allow us to see the underlying connection between factorization machines and latent tensor reconstruction.

2.5 Methods for multivariate regression

2.5.1 Linear regression methods

Multivariate regression analysis in the biological sciences is most commonly done using linear regression techniques. These methods have been studied extensively and there is a robust statistical framework for analyzing model fit. They rely on verifiable statistical assumptions about the data, and learned parameters can be easily interpreted and understood. However these models quickly become inadequate in situations where the prerequisite assumptions do not hold, such as when the data contains nonlinearities and complex multivariate interactions.

Regression is the task of predicting real-valued labels for observations as closely as possible. In regression the measure of error is based on the magnitude of the difference between the label and the model prediction. A common loss function to use in regression is the squared loss, or L_2 loss, given by the formula

$$L_2(x, y) := |x - y|^2. \tag{2.14}$$

The associated empirical loss is the mean squared error (MSE) defined as

$$\mathrm{MSE}(x, y) := \frac{1}{m}\sum_{i=1}^{m} L_2(x_i, y_i), \tag{2.15}$$

where m is the sample size. Given a hypothesis space of mappings between the data and training labels, the task in regression is to find mappings which minimize empirical error on the labeled data. Linear functions are a commonly used and extensively studies hypothesis space for regression. Algorithms that use this hypothesis space include linear regression, ridge regression, kernel ridge regression, support vector machines and LASSO [39].

It is a general principle in machine learning that when choosing a model there is a trade off between bias and variance of the trained models. Larger sample sizes guarantee better generalization, but also for a given empirical error smaller hypothesis sets also improve generalization performance, in what can be seen as an instance of Occam's razor. With certain conditions on the hypothesis space and loss functions it is possible to derive generalization bounds by using the Rademacher complexity or VC-dimension of the hypothesis space [39]. Although these complexity measures are rather technical and often difficult or impossible to compute for any given hypothesis space, they do allow us to show certain useful general results and to derive simpler generalization bound formulas for some commonly used hypothesis spaces.

Linear regression is perhaps the most popular regression technique and commonly used in many scientific disciplines. Linear regression can be written concisely as

$$\min_{\boldsymbol{W}} \left\|\boldsymbol{X}^T\boldsymbol{W} - \boldsymbol{Y}\right\|^2, \tag{2.16}$$

where $\boldsymbol{X}$ is a matrix of features, $\boldsymbol{W}$ is a matrix of weights, and $\boldsymbol{Y}$ is a matrix of potentially multivariate labels. Here as in the rest of the thesis we follow the convention that observations are represented as rows of the matrix X. The linearity is really only in the parameters, and the features can be arbitrary transformations of the data. For example in the case of polynomial regression we can use monomials of the original variables as the features. Additionally we can add a bias term to the regression by adding a constant term, such as ones, to the end of each feature vector. The objective function (2.16) is affine and differentiable, so we can find the optimum by solving for the zero of the gradient. This solution is

$$\boldsymbol{W} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y} \tag{2.17}$$

assuming $(\boldsymbol{X}\boldsymbol{X}^T)$ is invertible. Computing the matrix $(\boldsymbol{X}\boldsymbol{X}^T)$ has computational complexity $\mathcal{O}(md^2)$ and the matrix inversion has complexity $\mathcal{O}(d^3)$, so the overall solution has complexity $\mathcal{O}(md^2 + d^3)$. The generalization performance of this suffers from the lack of regularization, and it generally performs poorly in real world applications. [39]

Ridge regression, or Tikhonov regularization is a modification of the basic linear regression loss function that adds an extra term to regularize the weight parameters. The formula for ridge regression can be written in a form similar to (2.16) as

$$\min_{\boldsymbol{W}} \left[\lambda \left\|\boldsymbol{W}\right\|^2 + \left\|\boldsymbol{X}^T\boldsymbol{W} - \boldsymbol{Y}\right\|^2\right]. \tag{2.18}$$

Like basic linear regression, the ridge regression problem is convex and differentiable, and admits a closed solution both for its primal and dual problem. In addition to linear regression and ridge regression, popular models using a linear hypothesis space are support vector regression and Lasso.

If we want to model nonlinear relationships between observations and labels using linear techniques, we can to turn to nonlinear feature maps. We can also model interactions between variables by constructing additional features that combine two or more variables. Polynomial regression constructs new features as monomials of the original features [7]. The issue with this approach is that the number of interaction terms grows exponentially and with it the number of parameters. In the case of polynomial regression, the number of unique monomials in polynomial of degree d in n variables is

$$\binom{n+d}{n},$$

which gives space complexity $\mathcal{O}(n^d)$. This means that even for fairly small d and n_d the number of coefficients in the model becomes unfeasibly large. This is another example of the so-called curse of dimensionality [7]. Reducing this number of coefficients while maintaining the expressive power of polynomial regression is the primary motivation for the development of latent tensor methods. [53]

2.5.2 Kernel methods

Kernel methods are a family of machine learning methods that allow us to get many of the benefits of high-dimensional feature transformations without having to explicitly compute the feature mapping. As noted in the previous section, we can transform many nonlinear problems into linear ones by using a suitable mapping to a higher dimensional feature space. The issue with this approach is that computing this feature map quickly becomes very expensive or requires much feature engineering.

Kernel methods overcome this issue by using a special class of functions to leverage implicit feature mappings. The core principle of kernel methods can be seen in the definition of kernel functions. To qualify as a kernel, a bivariate function must be expressible as an inner product in some feature space. To define this more exactly, we also need the concept of a Hilbert space.

Definition 11. *A Hilbert space is a complete metric space with respect to the distance function induced by the inner product*

Definition 12. *A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a kernel if and only if there exists a Hilbert space $\mathcal{H}$ and a feature map $h : \mathcal{X} \to \mathcal{H}$ such that $\forall x, x' \in \mathcal{X}$,*

$$k(x, x') = \langle h(x), h(x') \rangle_{\mathcal{H}}.$$

What this means is that by using kernels we are implicitly mapping the features into a higher dimensional space and computing similarity as an inner product in that space. If we can reformulate a problem using kernels, we never actually need to compute the high-dimensional feature vector explicitly [25]. Fortunately many algorithms can be kernelized, such as PCA, k-means, canonical correlation analysis, and ridge regression [50].

When defining new kernels we generally don't need to use show the existence of a suitable feature map and Hilbert space. The sum and product of kernels is also a kernel, so in many cases we need only construct our proposed kernel from existing kernels. Additionally, the Moore-Aronszajn theorem states that a bivariate function is a kernel if and only if it is symmetric and positive definite. [23]

The advantage of using kernels instead of explicitly computing the feature map and inner product is that the kernel function generally has much lower computational cost or memory cost. Thanks to this, kernels allow us to get the benefit of using feature maps that would be prohibitively expensive to compute explicitly. This is commonly referred to as the *kernel trick* [25]. For example for the second order polynomial kernel

$$\begin{aligned} k_{\text{poly}}(\boldsymbol{x}, \boldsymbol{x}') &:= (\langle \boldsymbol{x}, \boldsymbol{x}' \rangle)^2 \\ &= \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle \end{aligned} \tag{2.19}$$

with corresponding feature map

$$\phi(\boldsymbol{x}) = (x_1 x_1, x_1 x_2, \ldots, x_n x_n),$$

the feature map has length n^2 and the computational complexity of the inner product in the target space is $\mathcal{O}(n^2)$, but evaluating the kernel function only has complexity $\mathcal{O}(n)$.

I give the definitions of some popular kernels that were used in the experiments. Polynomial kernels of arbitrary order d are defined as

$$k_{\text{poly}}(\boldsymbol{x}, \boldsymbol{x}') := (\langle \boldsymbol{x}, \boldsymbol{x}' \rangle + c)^d. \tag{2.20}$$

The Gaussian kernel is defined as

$$k_{\text{G}}(\boldsymbol{x}, \boldsymbol{x}') := \exp\left(-\gamma^{-2} \left\| x - x' \right\|^2\right). \tag{2.21}$$

Kernel ridge regression (KRR) is a popular and powerful regression method that takes advantage of the kernel trick to perform nonlinear regression without the need for combination features or feature engineering. KRR can has a closed form solution that can be computed efficiently as long as the data set is fairly small.

Deriving the solution to kernel ridge regression is much easier if we introduce the repeating kernel Hilbert space (RKHS) and the representer theorem.

Definition 13. *A Hilbert space $\mathcal{H}$ of real valued functions $f : \mathcal{X} \to \mathbb{R}$ is a repeating kernel Hilbert space, if for every $f \in \mathcal{H}$ there is a function $k(\cdot, x) \in \mathcal{H}$ such that*

$$\forall x \in \mathcal{X}, f(x) = \langle f, k(\cdot, x)\rangle.$$

This is called the reproducing property, and the kernel $k(\cdot, x)$ is called the reproducing kernel.

The feature mapping and Hilbert space specified in definition 12 are generally not unique, but if we require this Hilbert space to have the reproducing property, then $\mathcal{H}$ and ϕ are uniquely defined [25].

A very useful property is that the RKHS norm of a function controls the smoothness and thus enables us to control the complexity of our hypothesis space for regularization. By applying the Cauchy-Schwarz inequality we get the following

$$\begin{aligned} |f(x) - f'(x)| &= |\langle f, k(\cdot, x)\rangle - \langle f, k(\cdot, x')\rangle| \\ &= |\langle f, k(\cdot, x) - k(\cdot, x')\rangle| \\ &\leq \|k(\cdot, x) - k(\cdot, x')\|_{\mathcal{H}} \|f\|_{\mathcal{H}} . \end{aligned} \tag{2.22}$$

Thus a larger RKHS norm leads to more expressive functions, but also higher variance. We also have that two functions close in RKHS norm are pointwise similar [5].

The representer theorem states that the solution of a regularized empirical risk minimization problem where the feature space is a RKHS can be written as a linear combination of the repeating kernel evaluated at the training points. More formally, if the empirical loss minimization problem is given as

$$\arg\min_{f} \frac{1}{m} \sum_{i=1}^{m} L(y_i, f(x_i)) + \lambda \|f\|_{\mathcal{H}}^2 , \tag{2.23}$$

then the optimal solution $\hat{f}_{\mathcal{H}}$ is of the form

$$\hat{f}_{\mathcal{H}}(\cdot) = \sum_{i=1}^{m} \alpha_i k(\cdot, x_i). \tag{2.24}$$

Now that we know the solution has this form we can write

$$f_{\mathcal{H}}(x_j) = \sum_{i=1}^{m} \alpha_i k(x_i, x_j) = [\boldsymbol{K}\boldsymbol{\alpha}]_j,$$

and

$$\|f\|_{\mathcal{H}}^2 = \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j k(x_i, x_j) = \boldsymbol{\alpha}^T \boldsymbol{K}\boldsymbol{\alpha}.$$

Using mean squared error we can write the optimization problem (2.23) as

$$\arg\min_{\boldsymbol{\alpha}\in\mathbb{R}^m} \frac{1}{m}(\boldsymbol{K}\boldsymbol{\alpha} - \boldsymbol{y})^T(\boldsymbol{K}\boldsymbol{\alpha} - \boldsymbol{y}) + \lambda \boldsymbol{\alpha}^T \boldsymbol{K}\boldsymbol{\alpha}. \tag{2.25}$$

This is the target function of kernel ridge regression. The formula is differentiable and has analytical solution

$$\alpha = (\boldsymbol{K} + \lambda m \boldsymbol{I})^{-1}\boldsymbol{y}. \tag{2.26}$$

This is very convenient and allows kernel methods to be extensively analyzed using learning theory. The obvious downside is that if the number of training examples is large, then computing the matrix inversion in (2.26) becomes expensive, since it has computational complexity of $\mathcal{O}(m^3)$ (Gauss-Jordan elimination) [48] [14].

Kernel methods have a clear advantage in situations where the number of training examples is limited and the kernel trick is significantly less computationally expensive than computing the feature map explicitly, but they do not scale well to large training data. For this reason it is often better to use some approximation of the true feature map when the data sets in question are big. Several such approaches have been proposed such as Nyström approximation and random Fourier features [53]. For Gaussian kernels Mercer's theorem states that we can approximate the true infinite dimensional feature map with a finite sum of orthogonal functions. [50]

2.5.3 Neural networks

Neural networks are a very popular supervised learning method to use for problems that involve complex nonlinear interactions. In recent years deep neural networks in particular have produced impressive results in many application areas [36] but they are by no means the only type of neural network models being actively researched and applied [6].

The core idea in neural networks is to have multiple computational nodes, with each node performing only basic computations, but together able to model complex functions. Nodes are the basic units of neural networks. The output of each node is a weighted sum of its inputs, fed through an activation function. We can express the computations performed by a single node as

$$f(\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x} + b),$$

where $\boldsymbol{w}$ is the weight vector, b is a bias term, and σ is the activation function. Figure 2.5 shows the shape of the sigmoid and linear rectifier activation functions.

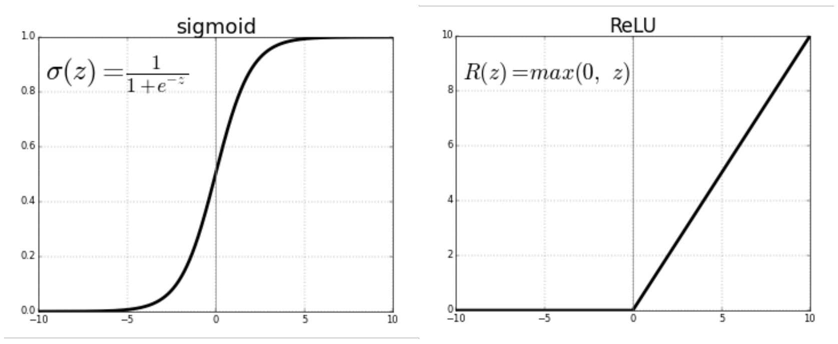

Figure 2.5: The sigmoid and linear rectifier (ReLu) activation functions commonly used in artificial neural networks.

Nonlinear activation functions enable the neural network to express nonlinear relationships between inputs. In modern neural network research and especially deep learning settings it is common to use linear rectification (ReLu) as it has been found to lead to faster learning and not require unsupervised pre-training unlike the smoother activation functions that were more popular previously. [36]

The expressive power of neural networks comes from the interconnections between nodes. The most basic neural network design, or architecture, is the feedforward network. In this type of network each layer of nodes only receives inputs from the preceding layer(s), meaning that there are no feedback connections. Figure 2.6 illustrates a basic design with two dense hidden layers and a single output node. Despite this restriction, these types of networks are extremely powerful and have found tremendous success in a wide variety of applications. The complexity of feedforward networks can be controlled by the number and size of layers. Intuitively feedforward networks build increasingly complex abstractions of the input features in each subsequent layer. This allows neural networks to learn good combination features, without a need for feature engineering. [22]

Hornik et al. showed that a multilayer layer feedforward network can approximate functions of arbitrary complexity in what is known as the uni-

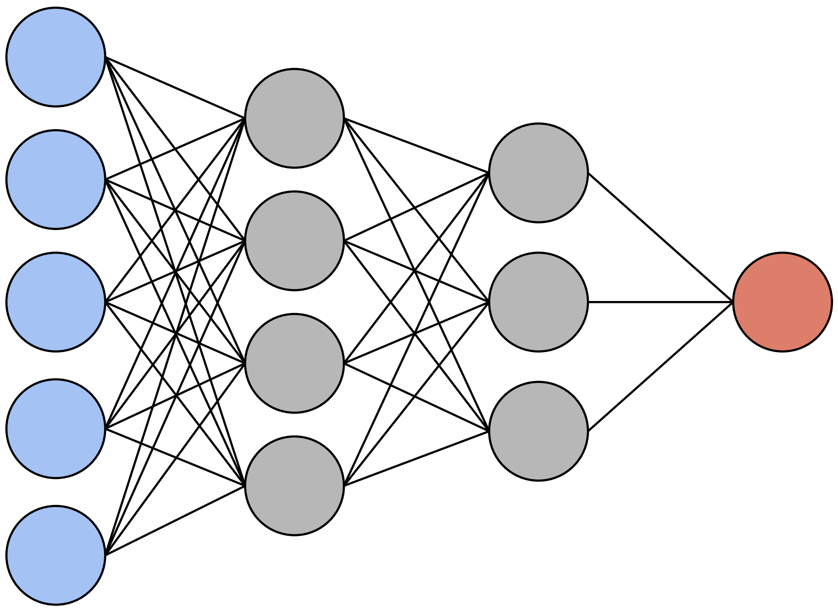

Figure 2.6: Illustration of a multilayer perceptron with a single output. The input layer is colored blue, hidden layers grey, and output red.

versal approximation theorem. They concluded that a neural network being unable to learn a specific mapping must be due to inadequate training, insufficient number of hidden units, or a lack of deterministic relation between input and output. [27]

Neural networks are trained the backpropagation algorithm, which applies the chain-rule for derivatives to compute the gradient of the weights of each layer successively. The gradients are then used to train the connection weights using an optimization procedure, such as Adam. [22]

Deep learning is a general term to describe architectures with a large number of hidden layers. These models use multiple processing layers to learn representations of the data at several different levels of abstraction. Deep learning methods have had tremendous success in various applications requiring the modelling of complex, nonlinear problems [36].

A major and much discussed problem with deep learning is the difficulty in interpreting their predictions. This is a significant weakness, since we are not able to gain insights about the structure of the data, unlike for example in the case of linear or logistic regression. Another major issue which partially pertains to the first one is that it is difficult to measure and predict the robustness of neural networks, with many studies and experiments demonstrating the weakness of neural networks towards adversarial attacks. For both of these reasons there is significant scepticism towards the reliability of deep neural networks both from the academia and those applying machine learning to safety-critical applications such as medical diagnostics. [46]

2.5.4 Factorization machines

Factorization machines (FM) are a class of supervised machine learning models proposed by Steffen Rendle in 2010 [43]. They can achieve empirical accuracy on par with polynomial regression and kernel-based methods, but with models that are significantly smaller and faster to evaluate [9]. They can be used for regression and classification, as well as other tasks, such as ranking. Factorization machines have been found to be highly effective in learning data with very high sparsity and have training times that are linear in the number of features. The basic model equation for factorization machines of order $d = 2$ is

$$y(\boldsymbol{x}) = w_0 + \langle \boldsymbol{w}, \boldsymbol{x} \rangle + \sum_{i'>i}^{n} \langle \boldsymbol{p}_i, \boldsymbol{p}_{i'} \rangle x_i x_{i'}, \tag{2.27}$$

where w_0 is a global bias term, w_i are single variable weights, and $\boldsymbol{v}_i$ models the interactions between variables. Thus the learnable parameters for the model are $b \in \mathbb{R}$, $\boldsymbol{w} \in \mathbb{R}^n$, and $\boldsymbol{P} \in \mathbb{R}^{n \times k}$.

Alternatively we can express the model equation as

$$y(\boldsymbol{x}) = w_0 + \langle \boldsymbol{w}, \boldsymbol{x} \rangle + \sum_{i'>i}^{n} (\boldsymbol{P}\boldsymbol{P}^T)_{ij} x_i x_{i'}, \tag{2.28}$$

which reveals the underlying factorization of the interaction matrix. It is a well known result that for any positive definite matrix $\boldsymbol{W}$ there exists $\boldsymbol{P}$ such that $\boldsymbol{W} = \boldsymbol{P} \cdot \boldsymbol{P}^T$ if k is sufficiently large [43]. This means that factorization machines have the potential to model arbitrary interaction matrices.

The choice of k is used in factorization machines to restrict the complexity of interactions. Smaller k leads to better generalization performance, and is better suited for learning highly sparse data. The original formulation of factorization machines is only capable of modelling second order interactions, making them comparable to second order polynomial regression. The main advantage of FMs is that in polynomial regression the algorithm needs to observe every variable combination at least once to learn the interactions terms whereas factorization machines are able to learn even from data where some interactions are missing. This is because the interaction terms are factorized by the parameter matrices P and can thus be inferred from related interactions at least to some degree.

The computational complexity of the basic formula (2.27) is $\mathcal{O}(kn^2)$, but it can be reformulated such that the model equation can be evaluated in time

$\mathcal{O}(kn)$. This also allows the model equation to be expressed in a simpler form. From Rendle's original paper we have

$$\begin{aligned}
\sum_{i'>i}^{n} \langle \boldsymbol{p}_i, \boldsymbol{p}_{i'} \rangle x_i x_{i'} &= \frac{1}{2} \sum_{i'>i}^{n} \langle \boldsymbol{p}_i, \boldsymbol{v}_{i'} \rangle x_i i' - \frac{1}{2} \sum_{i=1}^{n} \langle \boldsymbol{p}_i, \boldsymbol{p}_j \rangle x_i x_j \\
&= \frac{1}{2} \left(\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{f=1}^{k} p_{i,s} p_{j,s} x_i x_j - \sum_{i=1}^{n} \sum_{s=1}^{k} p_{i,s} pi, s x_i x_i \right) \\
&= \frac{1}{2} \sum_{s=1}^{k} \left(\left(\sum_{i=1}^{n} p_{i,s} x_i \right) \left(\sum_{j=1}^{n} p_{j,s} x_j \right) - \sum_{i=1}^{n} p_{i,s}^2 x_i^2 \right) \\
&= \frac{1}{2} \sum_{s=1}^{k} \left(\left(\sum_{i=1}^{n} p_{i,s} x_i \right)^2 - \sum_{i=1}^{n} p_{i,s}^2 x_i^2 \right) \\
&= \frac{1}{2} \left(\left\| \boldsymbol{P}^T \boldsymbol{x} \right\|^2 - \sum_{s=1}^{k} \left\| \boldsymbol{p}_s \circ \boldsymbol{x} \right\|^2 \right).
\end{aligned} \tag{2.29}$$

Another advantage of this formulation is that if x is highly sparse, we have to perform a reduced number of computations, since we only need to compute the terms inside the sums with nonzero x_i.

In the original FM paper Rendle proposes training factorization machines using mean squared error loss and training the model using stochastic gradient descent. A 2012 follow-up paper also describes methods for training FMs using alternating least-squares (ALS) and Markov Chain Monte Carlo (MCMC) [44]. An implementation of the algorithms described in this paper is published under the name libFM.

The direct approach to training factorization machines for regression is by using mean squared error loss with an L_2 regularization term

$$\frac{1}{n} \sum_{i=1}^{n} l(y_i, f_{\mathrm{FM}}(y)) + \frac{\beta_1}{2} \left\| \boldsymbol{w} \right\|^2 + \frac{\beta_2}{2} \left\| \boldsymbol{P} \right\|^2, \tag{2.30}$$

where l is a loss function, and β_1 and β_2 are hyperparameters that determine the strength of the regularization. Owing to the reformulation of the model equation (2.29) it is possible to write simple and efficiently computed partial derivatives for each model parameter. A single parameter update has computational complexity $\mathcal{O}(kn)$ or $\mathcal{O}(kn(\boldsymbol{x}))$ with sparse $\boldsymbol{x}$ and thus the runtime complexity per epoch is $\mathcal{O}(kmn(\boldsymbol{x}))$ [9].

The alternating least squares algorithm involves optimizing each parameter individually in a cycle. For factorization machines there is a closed form expression for the optimal value of each individual parameter. Solving this expression naively is quite computationally expensive, but using dynamic programming some of the terms of the expression can be reused which enables the update to be computed in time $\mathcal{O}(mn(x))$.

The third training procedure involves a Bayesian formulation of FMs which is trained using MCMC. The specifics of this model and training algorithm are beyond the scope of this thesis.

2.5.5 Higher order factorization machines

Factorization machines have found much success in many fields in modelling sparse and noisy data, however they are limited to capturing only second order interactions between parameters. The second order FM defined in (2.27) can be straightforwardly generalized to to a higher order factorization machine (HOFM) of order d by adding similarly factorized higher order interaction terms. Let $P^{(t)} \in \mathbb{R}^{n \times k_t}$ be the factor matrix for the interactions of order $t \in \{2, \dots, d\}$. We also adopt the notation $\langle x_1, x_2, \dots, x_n \rangle = \sum_i \prod_i x_i$. The higher order factorization machine is then defined as

$$y(\boldsymbol{x}) = w_0 + \langle \boldsymbol{w}, \boldsymbol{x} \rangle + \sum_{i'>i}^{n} \langle \boldsymbol{p}_i, \boldsymbol{p}_{i'} \rangle x_i x_{i'} + \dots + \sum_{i_d > \dots > i_1} \langle \boldsymbol{p}_{i_1}, \dots, \boldsymbol{p}_{i_d} \rangle x_{i_1} \dots x_{i_d}.$$

Through a similar procedure than that described in (2.29) it can be shown that this formula can be computed in linear time [43]. Even after the reformulation the model equation is relatively inefficient to evaluate and train, since each order of interactions is modelled using it's own factor matrix, and computing the model equation requires a large number of parameters to be evaluated. The basic objective function for training FMs is

$$\frac{1}{n} \sum_{i=1}^{n} l(y_i, f_{\text{HOFM}}(y)) + \frac{\beta_1}{2} \|\boldsymbol{w}\|^2 + \sum_{t=1}^{d} \frac{\beta_t}{2} \left\| \boldsymbol{P}^{(t)} \right\|^2. \tag{2.31}$$

In order to avoid having to find suitable hyperparameters β_t and k_t terms separately, it Blondel et al. [8] suggest setting $\beta_1 = \dots = \beta_d$ and $k_1 = \dots = k_t$.

Blondel et al. [8] propose an efficient generic algorithm for training HOFMs. Their approach is based around the reformulation of factorization machines using the ANOVA kernel presented in their earlier 2016 paper on polynomial networks and factorization machines [9]. This allows the gradient of the objective function to be computed efficiently using dynamic programming.

In the earlier paper Blondel et al. [9] use two different carefully picked kernel functions, the homogeneous polynomial kernel defined as

$$\begin{aligned}\mathcal{H}^d(\boldsymbol{x}, \boldsymbol{p}) &:= \langle \boldsymbol{p}, \boldsymbol{x} \rangle^d \\ &= \sum_{i_1=1}^{n} \dots \sum_{i_m=1}^{n} p_{i_1} x_{i_1} \dots p_{i_m} x_{i_m}\end{aligned} \tag{2.32}$$

and the ANOVA kernel

$$\mathcal{A}^d(\boldsymbol{x}, \boldsymbol{p}) := \sum_{i_m > \dots > i_1} p_{i_1} x_{i_1} \dots p_{i_m} x_{i_m}, \tag{2.33}$$

with special cases $m = 0$ and $m = 1$ being defined as $\mathcal{A}^0(\boldsymbol{x}, \boldsymbol{p}) := 1$ and $\mathcal{A}^1(\boldsymbol{x}, \boldsymbol{p}) := \langle \boldsymbol{p}, \boldsymbol{x} \rangle$. We can now write polynomial networks and second order factorization machines respectively as

$$f_{\text{PN}} = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + \sum_{t=1}^{n_t} \lambda_t \mathcal{H}^2(\boldsymbol{x}, \boldsymbol{p}_t) \tag{2.34}$$

and

$$f_{\text{FM}} = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + \sum_{t=1}^{n_t} \lambda_t \mathcal{A}^2(\boldsymbol{x}, \boldsymbol{p}_t). \tag{2.35}$$

Factorization machines fix $\lambda = 1$ so their expressiveness can be limited from what this formulation would allow [9]. This relation extends to higher dimensions, meaning that we can write higher order factorization machines using the ANOVA kernel as

$$f_{\text{HOFM}} = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + \sum_{t=1}^{n_t} \mathcal{A}^2(\boldsymbol{x}, \boldsymbol{p}_t^{(2)}) + \dots + \sum_{t=1}^{n_t} \mathcal{A}^d(\boldsymbol{x}, \boldsymbol{p}_t^{(d)}).$$

The homogeneous polynomial kernel and the ANOVA kernel are both homogeneous and the ANOVA kernel is multi-linear, meaning that $\mathcal{A}^d(\boldsymbol{x}, \boldsymbol{p})$ is affine in each p_t separately. Using a convex loss function l we can write an error function

$$\sum_{i=1}^{m} l(y_i, f_{\text{HOFM}}(\boldsymbol{x}_i; \boldsymbol{P}),$$

which can then be optimized with regard to the rows of $\boldsymbol{P}$ using gradient descent thanks to the multi-linearity of $\mathcal{A}^d$ [9].

In their latter 2016 paper, Blondel et al. propose a dynamic programming algorithm for computing the gradient of HOFMs in time $\mathcal{O}(nd)$. The key observation is that $\mathcal{A}^d$ can be defined recursively using lower order kernels, which also allows the gradient to be recursively defined. The lower order gradient terms can then be memoized to significantly reduce the number of computations required [8]. An implementation that is similar in design to the one proposed by Blondel et al. is implemented using tensorflow under the name tffm [38], although it was developed independently and prior to the first appearance of the paper.

In the next chapter I review another training procedure based on framing the problem as a low-rank tensor factorization problem.

2.5.6 Latent tensor methods

In this section I present an unifying perspective that allows us to view many of the models presented in previous section as different approaches to the same idea, and provides motivation for discussing the latent tensor reconstruction method in the next chapter.

Latent tensor methods are a family of machine learning methods, united by the property that variable interactions are parameterized as tensors which are inferred during the learning process. Blondel et al. [9] show that Polynomial Networks and Factorization Machines can both be cast as tensor reconstruction problems.

In naive polynomial regression the original data is transformed through a polynomial feature map into new features that allow us to use to use linear regression to train polynomial models. The model equation then becomes

$$f_{PR}(\boldsymbol{x}) = w_0 + \sum_{d=1}^{n_d} \sum_{i_d \geq \ldots \geq i_1}^{n} \mathcal{W}^{(d)}_{i_d,\ldots,i_1} x_{i_1} x_{i_2} \cdots x_{i_d}, \tag{2.36}$$

where $\mathcal{W}^{(d)}$ is the weight tensor corresponding to interaction terms of degree d. I define two classes of polynomials that are relevant to the discussion that follows.

Definition 14. *A polynomial is homogeneous if all the monomials have equal degree.*

Definition 15. *A polynomial is symmetric if it is invariant under all permutations of its arguments.*

Any non-homogeneous polynomial of degree d can be expressed as a homogeneous polynomial of degree $d+1$ by augmenting the variables with a constant term. We are thus able to capture all interactions up to degree n_d, as well as any bias terms with a single parameter tensor by the use of augmented data. I will perform all following analysis on homogeneous polynomials with the understanding that they can be made non-homogeneous with this augmentation. We can now simplify the model equation for naive polynomial regression to

$$f_{PR}(\boldsymbol{x}) = \sum_{i_d \geq \ldots \geq i_1}^{n} \mathcal{W}_{i_d,\ldots,i_1} x_{i_1} x_{i_2} \cdots x_{i_d}. \tag{2.37}$$

For the sake of simplifying notation I ignore the ordering in the index. It should be noted that elements of $\mathcal{W}$ which have the same combination of indices are redundant. We can now identify the expression as an elementwise product between two tensors, specifically

$$\begin{aligned} f_{PR}(\boldsymbol{x}) &= \sum_{i_d,\ldots,i_1}^{n} W_{i_d,\ldots,i_1} x_{i_1} x_{i_2} \cdots x_{i_d} \\ &= \mathcal{W} \circ \boldsymbol{x}^{\otimes d}, \end{aligned} \tag{2.38}$$

where $\boldsymbol{x}^{\otimes d}$ is the outer product of $\boldsymbol{x}$ d times with itself. Thus we see that we can use tensors of order d to represent homogeneous polynomials of degree d. Additionally the algebra of symmetric tensors is isomorphic to the algebra of homogeneous polynomials defined on $\mathbb{R}$ [53]. This is quite intuitive when looking at the definitions and comparing definitions 15 and 9.

Another interesting result of the reformulation 2.35 is that we can cast factorization machines as a low-rank tensor estimation problem. It follows directly from the definition of $\mathcal{H}^m$ (2.32) that

$$\mathcal{H}^d(\boldsymbol{x}, \boldsymbol{p}) = \langle \boldsymbol{p}^{\otimes d}, \boldsymbol{x}^{\otimes d} \rangle, \tag{2.39}$$

and similarly for the ANOVA kernel

$$\mathcal{H}^d(\boldsymbol{x}, \boldsymbol{p}) = \langle \boldsymbol{p}^{\otimes d}, \boldsymbol{x}^{\otimes d} \rangle_{>}, \tag{2.40}$$

where $\langle \cdot, \cdot \rangle_{>}$ is shorthand for

$$\langle \mathcal{X}, \mathcal{W} \rangle_{>} := \sum_{i_d > \ldots > i_1} \mathcal{W}_{i_1,\ldots,i_d} \mathcal{X}_{i_1,\ldots,i_d}.$$

Now if we write

$$\mathcal{W} = \sum_{t=1}^{n_t} \lambda_t p_t^{\otimes d}, \tag{2.41}$$

then by the linearity of $\langle \cdot, \cdot \rangle$ and $\langle \cdot, \cdot \rangle_{>}$ we can rewrite (2.34) and (2.35) as

$$f_{\mathrm{PN}} = \langle \mathcal{W}, \boldsymbol{x}^{\otimes d} \rangle \tag{2.42}$$

and

$$f_{\mathrm{FM}} = \langle \mathcal{W}, \boldsymbol{x}^{\otimes d} \rangle_{>}. \tag{2.43}$$

We can see that by the definition of the rank decomposition (2.13) the tensor $\mathcal{W}$ is in fact a rank decomposition. We can thus reframe the training of factorization machines as the problem of reconstructing the tensor $\mathcal{W}$.

Blondel et al. use this reformulation to come up with a more efficient training procedure for HOFMs. This reformulation and the application of a symmetrization trick results in a multi-convex problem that can be solved efficiently with alternating minimization.

This perspective of viewing the factorization machines and higher order factorization machines as problems of computing a low rank approximation of the interaction tensor $\mathcal{W}$ also allows for easy interpretability of trained models. We can compute the full $\mathcal{W}$ from the parameter vectors $\boldsymbol{p}_t$ using equation (2.41) which can then be analyzed to identify multivariate interactions.

2.5.7 Latent tensor reconstruction

Latent tensor reconstruction (LTR) is a supervised machine learning model that uses tensor rank decompositions to reduce the number of model parameters in higher order polynomial regression. In section 2.5.6 I reviewed the result by Blondel et al. [9] that polynomial networks and factorization machines can be reformulated as low rank matrix factorization problems. LTR starts from this formulation, but instead of constructing each rank of the latent tensor as an outer product $p_t^{\otimes d}$, it uses a full rank decomposition, which is described in section 2.4. This has the effect of removing the restriction of being limited to representing homogeneous polynomials. Table 2.1 shows a condensed representation of using LTR for regression.

Decoupling the factors of the polynomial expands the hypothesis class of the model and thus leads to greater expressive power, but also invariably requires more data for convergence, as mentioned in section 2.1. This has also been verified experimentally [53]. The potential for overfitting is likewise greater compared to HOFM, which most likely needs to be taken into account via the use of regularization or other measures.

The model equation for LTR is

$$\begin{aligned} f(\boldsymbol{x}) :&= \sum_{t=1}^{n_t} \lambda_t \langle \boldsymbol{p}_1^{(t)} \otimes \cdots \otimes \boldsymbol{p}_{n_d}^{(t)}, \boldsymbol{x}^{\otimes n_d} \rangle \\ &= \sum_{t=1}^{n_t} \lambda_t \langle \boldsymbol{p}_1^{(t)}, \boldsymbol{x} \rangle \cdot \ldots \cdot \langle \boldsymbol{p}_{n_d}^{(t)}, \boldsymbol{x} \rangle. \end{aligned} \tag{2.44}$$

Polynomial from tensor	Tensor from sample examples
Given : $\boldsymbol{T}$ tensor, $\boldsymbol{x}$	Given : $\{(y_i, \boldsymbol{x}_i) \mid i = 1, \ldots, m\}$
Output : y	Output : $\boldsymbol{T}$ tensor
$\boldsymbol{T} \quad \boldsymbol{x}$	$\{y_i\} \quad \{\boldsymbol{x}_i\}$
$\downarrow \quad \downarrow$	$\downarrow \quad \downarrow$
$f(\boldsymbol{x}) = \langle \boldsymbol{T}, \otimes^{n_d} \boldsymbol{x} \rangle \Rightarrow y$	$\min_{\boldsymbol{T}} \sum_i \lvert\lvert y_i - \langle \boldsymbol{T}, \otimes^{n_d} \boldsymbol{x}_i \rangle \rvert\rvert^2 \Rightarrow \boldsymbol{T}$

Table 2.1: The general scheme of Latent Tensor Reconstruction based regression from the LTR paper [53].

Where the second line follows from a well known tensor identity [54]. Comparing equation (2.44) to the reformulation of HOFM in equation (2.41) clearly shows the difference in model design. Adding a Tikhonov-type regularization term to prevent overfitting and the problem of degeneracy mentioned in section 2.4 gives the target equation

$$\begin{aligned}&\min \frac{1}{m} \sum_{i=1}^{m} \left\| y_i - \sum_{t=1}^{n_t} \lambda_t f^{(t)}(\boldsymbol{x}_i) \right\|^2 + \frac{C_p}{n_t n_d n} \sum_{t=1}^{n_t} \sum_{d=1}^{n_d} \left\| \boldsymbol{p}_d^{(t)} \right\|^2 . \\ &\text{w.r.t. } \lambda_t, (\boldsymbol{p}_1^{(t)}, \ldots, \boldsymbol{p}_{n_d}^{(t)}), t = 1, \ldots, n_t.\end{aligned} \tag{2.45}$$

If we collect the parameter vectors $\boldsymbol{p}_d$ in to matrices as $P_d = [\boldsymbol{p}_d^T]_{t=1}^{n_t}$, weights λ_t as a column vector $\boldsymbol{\lambda} = [\lambda_t]_{t=1}^{n_t}$, and write $\boldsymbol{F} = \circ_{d=1}^{n_d} \boldsymbol{X} \boldsymbol{P}_d^T$ then we can write the target function concisely as

$$\begin{aligned}&\min \frac{1}{2mn_y} \|\boldsymbol{Y} - \boldsymbol{F}\boldsymbol{\lambda}\|^2 + \frac{C_p}{n_t n_d n} \sum_{d=1}^{n_d} \|\boldsymbol{P}_d\|^2 \\ &\text{w.r.t. } \boldsymbol{\lambda}, \boldsymbol{P}_d, d = 1, \ldots, n_d.\end{aligned} \tag{2.46}$$

Szedmak et al. propose both a rank-wise training algorithm and one where all parameters are optimized simultaneously. The rank-wise algorithm has the advantage of having a constant memory requirement with respect to rank, and allowing us to control the complexity of the model during the training process, which helps prevent overfitting. However based on the fact that optimal solutions to tensor rank-decompositions cannot be computed rank-wise, as explained in section 2.4, it seems likely that this training algorithm would not converge optimally in most situations. For the full rank-training algorithm, the authors showed that at a fixed number of epochs the computational complexity of training this model is $\mathcal{O}(mnn_dn_t)$. Being linear in sample size, rank, and order of the model is quite good, especially compared to naive polynomial regression. [53].

In the next chapter I describe a new GPU implementation of this model, which includes both training procedures, and in chapter 5 I analyze the performance of the implementation in various scenarios comparing it against various other supervised learning models for multivariate data.

Chapter 3

Model Implementation

As part of my thesis developed a new implementation of the latent tensor reconstruction algorithm that uses the TensorFlow [1] library to enable hardware accelerated tensor computations using GPUs and TPUs. This implementation is highly modular enabling the integration of LTR as a component in other TensorFlow models, such as neural networks, as well as fast prototyping.

Modern GPUs have enable highly parallelized algorithms for matrix computations which can in result in extreme performance improvements over single-threaded computations. For example under ideal circumstances the computation time of matrix multiplication can be $\mathcal{O}(m)$ compared to $\mathcal{O}(m^3)$ in sequential computation, with similarly dramatic improvements possible for other matrix and vector operations [3]. Since the model equation of LTR (2.46) and gradient computations can be formulated entirely as matrix operations, it is possible to gain significant performance improvements by utilizing GPU computing.

TensorFlow provides highly optimized algorithms for tensor operations as well as automatic gradient computation. TensorFlow also includes efficient implementation for operating on sparse tensors by the use of the module. Thus it is possible to achieve significant performance and memory improvements by using the specialized sparse data types provided by TensorFlow.

The use of TensorFlow allows for high levels of flexibility and modularity in the configuration of the algorithm, making it trivial to modify the algorithm to use different optimization procedures, loss functions, or regularizers. This makes it easy to perform extensive hyperparameter optimization and by extension to identify optimal training procedures for a variety of scenarios. It is also straightforward to implement extensions that use different output spaces, such as binary classification or ranking by modifying adding different decision functions and choosing an appropriate loss function in a manner similar to that described by Rendle in his original FM paper [43].

TensorFlow also comes with an extensive suite of logging and performance analysis tools called Tensorboard. It allows us to easily and conveniently track loss during training, efficiency and possible bottlenecks in the computational graph, and compare model performance under different hyperparameters for model selection.

3.1 Design and training procedure

The implementation is closely based on the matrix formulation given by Szedmak et al. in the original LTR paper [53]. The various components are represented as tensorflow tensors and computations are carried out using native libraries.

I used the 2.0 version of TensorFlow for the implementation which introduces fairly significant changes the the way in which models are defined when compared to TensorFlow 1. Namely the user does not manually specify the structure of the computational graph, which TensorFlow uses internally to represent computations, but instead specifies which parts of the code form distinct functional components, which are then automatically turned into computational graphs and optimized by Tensorflow. In order to maximal flexibility while still gaining the performance advantages associated with this procedure, I specified the computation of the term

$$f(\boldsymbol{X}) = \boldsymbol{F}\boldsymbol{\lambda} = \left(\circ_{d=1}^{n_d}\boldsymbol{X}\boldsymbol{P}_d^T\right)\boldsymbol{\lambda}$$

as one such component. I also manually specified the gradient computation for this term instead of relying on automatic gradient computation, which provided a slight performance increase. The gradients of this component are

$$\begin{aligned}\nabla_{\boldsymbol{\lambda}} f(\boldsymbol{X}) &= -(\boldsymbol{1}_m\boldsymbol{F})^T\\ \nabla_{\boldsymbol{P}_d} f(\boldsymbol{X}) &= -(\boldsymbol{F}_{\backslash k}\boldsymbol{\lambda})^T\boldsymbol{X}\end{aligned}$$

where $\boldsymbol{1}$ is a vector of ones and $\boldsymbol{F}_{\backslash k} = \circ_{d=1,d\neq k}^{n_d}\boldsymbol{X}\boldsymbol{P}_d^T$. Rather than computing each $\boldsymbol{F}_{\backslash k}\boldsymbol{\lambda}$ separately we can compute an intermediate result

$$\boldsymbol{F}_{\text{int}} = \left(\circ_{d=1}^{n_d}\boldsymbol{X}\boldsymbol{P}_d^T\right)\text{diag}(\boldsymbol{\lambda}),$$

which is a matrix unlike $\boldsymbol{F}$. We can then compute the desired terms as

$$\boldsymbol{F}_{\backslash k}\cdot\boldsymbol{\lambda} = (\boldsymbol{F}_{\text{int}}/(\boldsymbol{X}\boldsymbol{P}_k))\boldsymbol{1}_{n_t},$$

where / denotes elementwise division. The implementation of the above in Tensorflow is quite short and is included in appendix B.

The first training algorithm presented by Szedmak et al. successively solves rank-one subproblems and deflating the target. My implementation includes both this learning procedure and training all ranks simultaneously. During experiments I found that training all ranks simultaneously results in much better performance, so in most scenarios there is little reason to train the model rank-wise. In section 5.1 I present experimental results from comparing the two training procedures on synthetic data of known rank.

Since the magnitude of the gradient is directly proportional to the magnitude of the parameters, which in turn are multiplicative with each other, the model is quite prone to exploding gradients. For this reason it is difficult or impossible to find learning rates and initialization rules that would lead to stable convergence when using stochastic gradient descent. I found that using optimization algorithms with adaptive learning rates, such as Adam, resulted in much more robust convergence and better overall performance.

3.2 Running time comparisons

I conducted experiments to measure the real world difference between running times between different configurations. For the comparison I used a highly sparse large dataset, with a non-zero rate of 2.3%. I let the training run for 10 epochs in order for the epoch times to settle, and then averaged the epoch times over the next 10 epochs. The system used for the test was a workstation with an Intel i7-4790K CPU, 16GB of DDR3 RAM, and an NVIDIA RTX 2070 Super GPU. I measured the difference between using the CPU and GPU, automatic gradient computation and user defined gradient formula, as well as using special data types for handling sparse data. The results are presented in table 3.1.

comparison	A	B	Δ
CPU vs GPU	15.31	6.90	-54.9%
automatic vs custom gradient	6.90	6.79	-1.6%
dense vs sparse	6.90	9.06	+31.3%

Table 3.1: Epoch time comparisons for LTR.

As expected, running the training on a GPU resulted in a speedup of over 50%, although this will of course vary greatly depending on the hardware used. Using the custom gradient gave only a marginal decrease in execution time, suggesting that the automatic gradient computation is well optimized.

Perhaps surprising was that using a sparse representation for the data increased epoch times despite dedicated operations being used. The reason for this is difficult to ascertain without deeper knowledge of the implementation of the libraries used, or runtime analysis of the computational graph.

Chapter 4

Experiments

In this section I describe the experimental setup, data, and benchmarks used to analyze the performance of the LTR algorithm. I analyzed the performance of LTR alone as well as against several other machine learning algorithms using both synthetic and real-world data. Cross-validation was used in the

4.1 Testing environment

All experiments were performed using the Python programming language. The scikit-learn ridge regression implementation used as a benchmark in the experiments. For higher-order factorization machines I used the tffm TensorFlow implementation of Trofimov and Novikov [38]. The implementation of HOFMs uses TensorFlow 1, and at time of writing there is no good port to TensorFlow 2. I used TensorFlow 2 for my implementation of LTR as described in chapter 3 because of the simpler and more flexible interface. There should be little difference in efficiency due to this difference in versions, with the more knowledgeable implementation of HOFM likely having some advantage in terms of optimal resource usage. The names and versions of the main software libraries used in the experiments are listed in table 4.1.

4.2 Data

The main experiments were conducted using a real world dataset, but synthetic data was also used for investigating the performance characteristics of LTR and HOFM in controlled scenarios.

name	version
Python	3.7.7
TensorFlow 1	1.15.0
TensorFlow 2	2.2.0
numpy	1.18.1
scikit-learn	0.22.1

Table 4.1: Versions of the main software libraries used.

4.2.1 Generating synthetic data

In some of the experiments I used synthetically generated data to test the performance of LTR and HOFM. This data was generated by using the tensor-to-polynomial procedure shown in the first part of table 2.1. A random interaction tensor $\mathcal{W}$ of a given order and rank was generated and then applied to randomly generated multivariate data $\boldsymbol{X}$ to produce the output y. More specifically

$$\begin{aligned}
\boldsymbol{X} &\sim \mathcal{N}_n(0, \operatorname{diag}(\mathbf{1}_n)) \\
\boldsymbol{P}_d &\sim \mathcal{U}(-1, 1) \\
\boldsymbol{\lambda} &= \mathbf{1}_{\text{rank}}/n_t \\
\boldsymbol{y} &= \left(\circ_{d=1}^{n_d} \boldsymbol{X}\boldsymbol{P}_d^T\right) \cdot \boldsymbol{\lambda}
\end{aligned}$$

Finally $\boldsymbol{y}$ is normalized to zero mean and unit variance. The script for generating the synthetic data is given in appendix C. The resultant distribution of $\boldsymbol{y}$ is shown in figure 4.1. Except for first order, all orders of polynomial result in similarly shaped distributions after normalization.

4.2.2 NCI-ALMANAC

The NCI-ALMANAC (A Large Matrix of Anti Neoplastic Combinations) dataset [26] is a publicly available data set of pairs of drugs for cancer treatment. Combinations of FDA approved small-molecule cancer drugs were tested in a laboratory environment against a variety of human tumor cell lines. The cancer cells were exposed to the drugs *in vivo* and the subsequent change in the size of the sample was measured using the NCI-60 testing protocol. The time interval between the application of the drugs and the effect measurement was two days.

The motivation behind the compilation of the data set is the study and development of combination therapies for cancer. Tumours often exhibit a

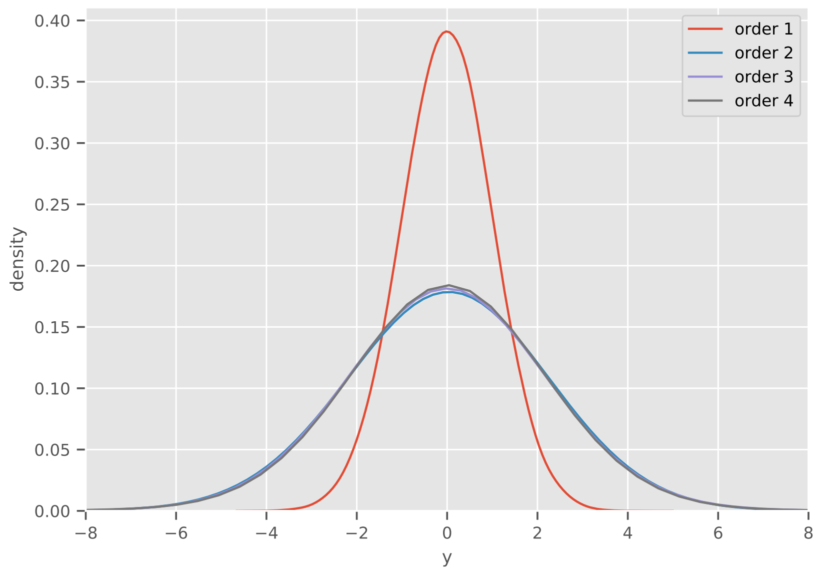

Figure 4.1: Density plots of synthetic y corresponding to different orders.

level of robustness due to various factors, such as genetic instability, feedback connections, and redundancy. This allows tumors to persist and even dynamically develop resistance to drugs meant to damage them. Findings in various areas of cancer research have revealed the complex nature of the regulatory mechanisms present in cancer and suggest that drugs targeting any single molecule are unlikely to be magic bullets. In addition to overcoming drug resistance, combination therapies promise higher effectiveness and less toxicity in healthy tissues when compared to monotherapies. [32]

One way to approach this issue is to apply multiple drugs which impose different selective pressures on the tumor. Combination therapies are not a new idea, but developments in biology and omics have rapidly increased the study of combination therapies [20]. Finding these combination requires large amounts of data both from high-throughput bioinformatics and *in vivo* experiments. Computational strategies for predicting drug synergy are critical to guide experimental approaches, since the cost of evaluating drug combinations in-vitro or in clinical trials increases exponentially as the with the number of drugs included in the search.

The NCI-ALMANAC experiments used 60 well-characterized human tumor cell lines from the NCI-60 screen and pairwise combinations of 104 FDA-approved cancer drugs. Drugs were administered at multiple concen-

Feature	length
Drug 1	50
Drug 2	50
Concentration 1	46
Concentration 2	46
Cell line	60

Table 4.2: Table of the lengths of the one-hot encodings for each of the features.

trations with the minimum and maximum concentrations being based on clinical guidelines. Each drug pair was tested at either a 3x3 or 3x5 grid of concentrations resulting in almost three million individual data points.

For the experiments I used a modified version of the original data set created by Heli Julkunen for her master's thesis [29]. This design imposes various restrictions on which measurements are included, which brings the number of data points down to 333 180 combination measurements and 222 120 monotherapy measurements. The feature vector consists of the identities of the two drugs, the concentrations of the drugs, and the identity of the cell line. All five are one-hot encoded and concatenated to form the final feature vector. An example of what a single encoded feature vector looks like is

$$\boldsymbol{x}_i = \underbrace{0, 1, \ldots, 0}_{\text{drug 1}} \underbrace{0, 0, \ldots, 0}_{\text{drug 2}}, \ldots, \underbrace{1, 0, \ldots, 0}_{\text{cell line}}.$$

The lengths of the one-hot encodings are listed in table 4.2.

Each of the drugs appear in the data set an equal number of times. The target variable for the regression is the percentage growth of the cancer after the drugs were administered. The distribution of this variable is shown in figure 4.2. For the experiments I normalized the target variable to have zero mean and unit variance.

4.3 Accuracy measures

In order to measure the accuracy of model predictions in the regression task I used the following methods.

Mean squared error

Mean squared is a distance measure that is generally used to measure the difference between two real-valued vectors. It is always non-negative. I is

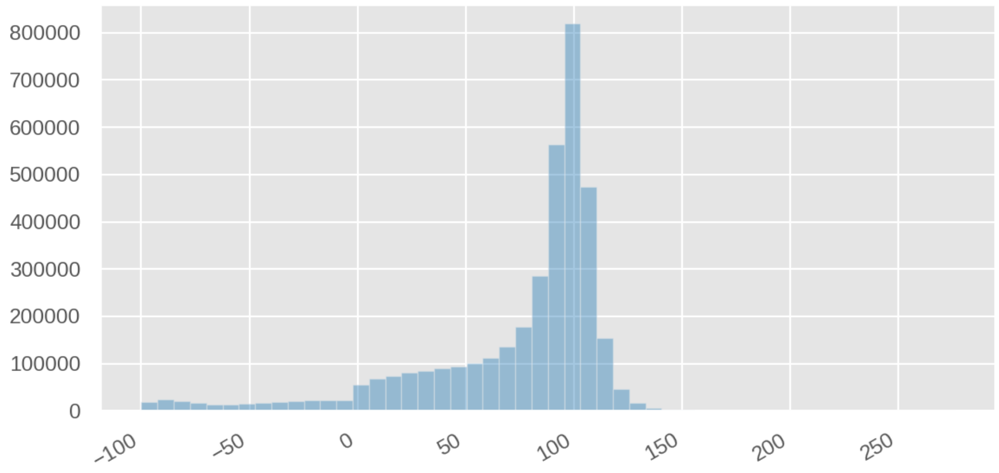

Figure 4.2: Histogram of the percentage growth variable.

defined as

$$\mathrm{MSE}(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{m} \sum_{i=1}^{m} (x_i - y_i)^2.$$

It is not scale-independent and therefore should not be used to compare accuracy of predicting data with different scales. Root mean squared error (RMSE) is defined as the square root of mean squared error, and is often preferred as it is on the same scale as the data. [28]

Pearson correlation

Pearson correlation coefficient (PCC) measures the linear correlation between two variables. It is defined as the covariance of the two variables divided by the product of their standard deviations. It can have values between -1 and +1. The sample Pearson correlation coefficient is computed using the formula

$$\mathrm{PCC}(\boldsymbol{x}, \boldsymbol{y}) = \frac{\sum_{i=1}^{m} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{m} (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^{m} (y_i - \bar{y})^2}}.$$

Pearson correlation coefficient is a standard measure of correlation in regression analysis. There are various ways of interpreting the Pearson correlation but a illustrative and basic one is to say that the correlation coefficient is positive if the values tend to lie on the same side of their respective means. Pearson correlation is invariant to linear transformations, such as scaling. [45]

Coefficient of determination

The coefficient of determination or R^2 score measures the proportion of variance in the dependent variable that is predictable from the independent variable. It can be defined as the square of the Pearson correlation coefficient. [51]

Spearman correlation

Spearman's rank correlation coefficient is a measure of the degree of similarity between the rankings of two variables. It is defined as the Pearson correlation coefficient between the rank variables. The rank variables here refer to the ranking of each item in order of value. It has no connection to the rank of any associated data distribution. [40]

4.4 Cross-validation

K-fold cross-validation is an effective method for estimating performance of a machine learning model and is also commonly used for optimizing hyperparameters. The data set is successively split into separate training and testing sets, and a model is trained multiple times using the different splits. We can perform this procedure with different hyperparameters to get a distribution of accuracies instead of a single score on a validation set. The issue with is that if we are using the full data set for the hyperparameter optimization procedure, then there is no data left that could be used to give an unbiased estimate of the generalization performance. [56]

Nested cross validation is a procedure for computing an unbiased estimate of the prediction error of a machine learning algorithm proposed by Varma and Simon [55]. The solution is to set aside a validation set that is not used for the hyperparameter selection and thus gives an unbiased performance estimate. Nested cross-validation works by repeating this procedure such that the division of the data into the validation set and the cross-validation data that is used in hyperparameter selection acts as its own 'outer' cross-validation loop. Figure 4.3 illustrates the procedure.

Nested cross-validation gives the most unbiased estimate of model performance, but is also very expensive to perform as it requires multiple runs of the full cross-validation procedure on the inner folds. It has been argued that performing nested cross-validation in situations where maximum rigor is not required is overzealous. Wainer and Cawley tested a variety of models and training scenarios and found that in most situations regular cross-validation

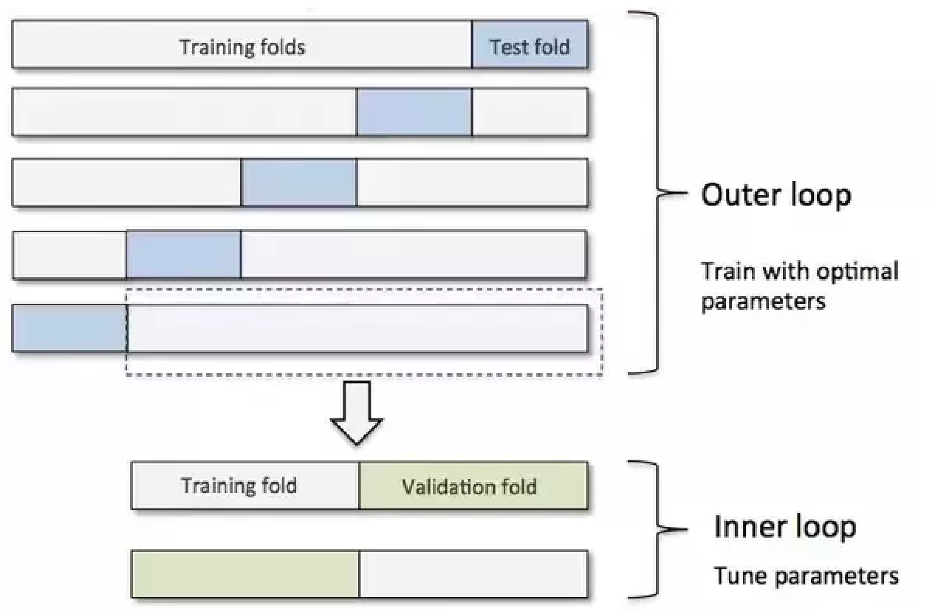

Figure 4.3: Illustration of the nested cross-validation procedure by Sebastian Raschka [42].

lead to the choice of a model that had performance on par with that selected using nested CV [56]. Furthermore as long as we are applying the same cross-validation procedure for all models tested, the same positive bias is induced on all models, and thus conclusions based on this analysis are valid as long as we are only comparing the models to each other and not making claims about generalization performance. For these reasons I chose to use regular cross-validation for the model comparison experiment.

4.5 Neural Network

In the final model comparison experiment I used a deep feedforward neural network as a point of comparison for performance. This was implemented using the Keras Python neural network library. The network consisted of four fully connected hidden layers of decreasing size similar to the network in figure 2.6. The layer sizes and activation functions are listed in table 4.3. All layers were dense and used the linear rectifier activation function. I trained the neural network using the TensorFlow Adam optimizer with a learning rate of 0.01 and otherwise default parameters.

Layer	type	size	activation
1	dense	512	ReLu
2	dense	128	ReLu
3	dense	64	ReLu
4	dense	24	ReLu
5	dense	1	None

Table 4.3: Architecture of the neural network used in the model comparison.

Chapter 5

Results

In this chapter I present my experimental results. I conducted experiments with three different goals: finding the optimal range of hyperparameters for LTR, testing the performance of LTR in various synthetic scenarios, and comparing LTR to other regression methods on real world data. These experiments should provide a comprehensive overview of the performance characteristics of LTR when dealing with high-dimensional, high-rank data, as well as how it compares to other commonly used methods on real world data. Notably I found that LTR showed state-of-the-art performance in the NCI-ALMANAC task with my implementation having only moderately worse run times compared to the tffm implementation of HOFM.

5.1 Hyperparameter search

Due to the non-negligible computation times of LTR on large data, I first searched for the rough range of valid hyperparameters by varying only one parameter at a time. I then performed a grid search on a selected set of hyperparameters to get closer to optimal values. With the exception of batch size, which was fixed at 500 for all experiments, I assumed that the hyperparameters are independent enough of each other such that good candidate values for them can be found by this method at least to an accuracy of a few orders of magnitude. This assumption was necessary for me to be able to search through a wide range of values for a number of hyperparameters. Table 5.1 lists all the different hyperparameter values tested individually.

I used the same testing procedure for all cross-validation runs. I used a constant learning rate of 0.01 and trained the model for 100 epochs. I used a batch size of 500 for all experiments to maintain consistency and to not interfere with the learning rate parameter. When using the NCI-ALMANAC

Parameter	Values
Training procedure	{rank-wise, full-rank}
Regularization	{100, 10, 1, 0.1, 0.01}
Learning rate	{0.01, 0.001, 0.0001}
Learning rate scheduling	{constant, inverse time, exponential}

Table 5.1: Table listing all the hyperparameter values tested with cross-validation.

dataset I used the same split indices for each of cross-validation run.

5.1.1 Training procedure

In order to compare performance between training the model rank-wise or all ranks simultaneously I generated random data of a fixed known rank and tested how well the model learned the data at various ranks with both training procedures. I generated the data by feeding five-dimensional data drawn from a unit normal distribution to a 3rd order rank four polynomial with the weights drawn from a uniform distribution. This data was then used to train two models corresponding to the different training procedures for 100 epochs each. This procedure was repeated 5 times for each rank with new random data for each repetition. The results are presented in figure 5.1.

Training all ranks simultaneously results in a test error that plateaus at the true data rank. Rank-wise training is not able to match this accuracy even at ranks above the true rank. This result is under idealized conditions on completely noise free data, but still serves as empirical evidence in support of the remarks I made in section 2.5.7.

5.1.2 Regularization

The purpose of regularization is to reduce overfitting as explained in section 2.3.1, and in the case of LTR, to eliminate the occurrence of degeneracy as explained in section 2.4. The optimal strength of regularization generally depends on the size of the data set [7], so I ran a cross-validation experiment using both the full NCI-ALMANAC dataset and a random subset to see if overfitting would be a factor at smaller sample sizes. The results are shown in figure 5.2.

For the full dataset a regularization parameter of 1 or lower gives roughly the same result, with the exception of one outlier. The situation is similar with the 10% subset with a parameter value of 10 arguably showing a slight

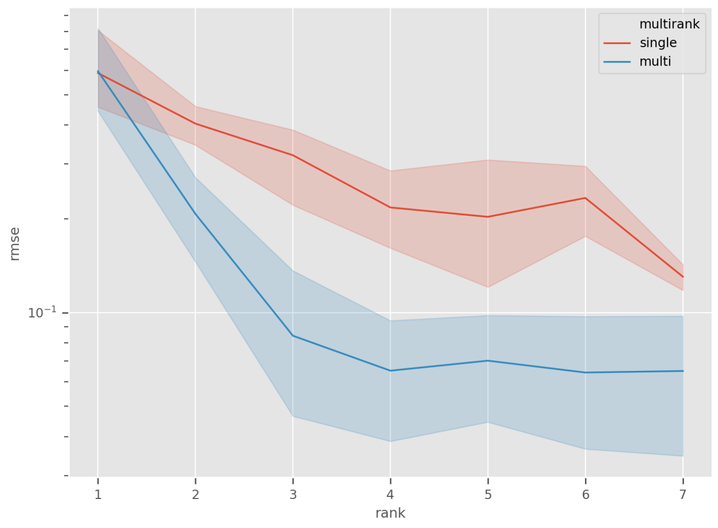

Figure 5.1: Accuracy as a function of rank for LTR models trained rank-wise and all ranks simultaneously. The true rank of the polynomial that was used to generate the data is four.

advantage. Investigation of the error graphs reveals signs of overfitting, which is not present with the full dataset. This is in spite of the fact that even the smaller data set contains 100k training examples.

5.1.3 Optimizer

I found that in most cases, using the Adam optimizer resulted in relatively fast convergence and easily optimized hyperparameters. I was generally able to get superior results with Adam compared to Nadam, even though it has been shown to result in superior convergence in some cases [15]. Additionally run times when using Nadam were roughly twice greater, making Adam a much more pragmatic choice especially when dealing with large data sets.

Learning rate and learning rate annealing

For finding the optimal learning rate I used the Adam optimizer, a regularization parameter of 10, and trained 200 epochs without early stopping. I first experimented with different constant learning rates, the results of which are presented in figure 5.3. From this experiment it is quite clear that, at

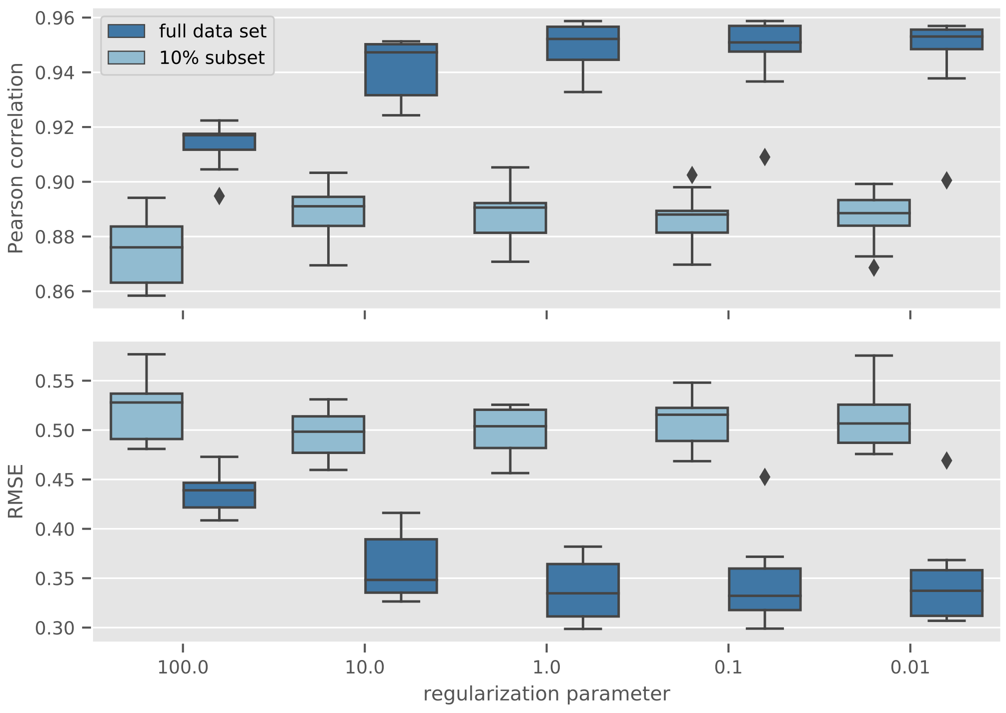

Figure 5.2: Cross-validation results for different regularization parameter values for a 5th order LTR model.

least for the NCI-ALMANAC dataset, a learning rate of around 0.001 yields the best results.

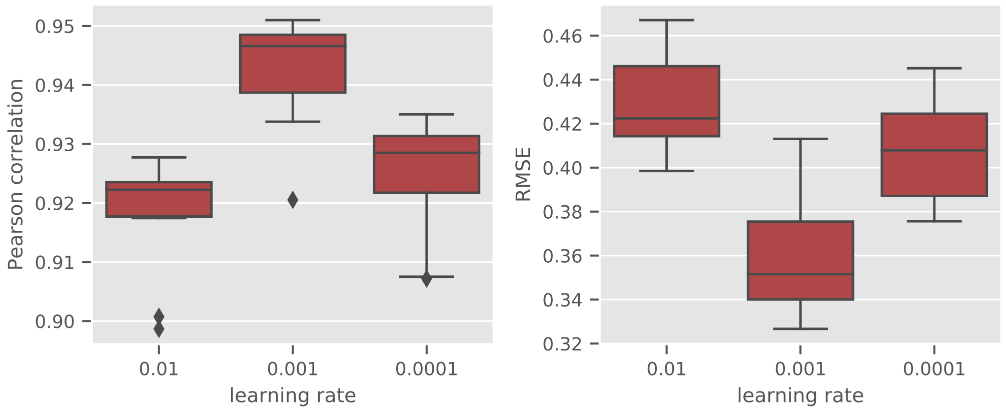

Figure 5.3: Cross-validation of different constant learning rates with Adam optimizer.

I also experimented with learning rate annealing in the form of learning rate scheduling. Learning rate scheduling allows the model to progress quickly near the beginning, but slow down later, which can lead to better convergence [47]. I found that simple schedulers did not produce significantly higher accuracy overall, but did speed up convergence, as would be expected. A representative example of this is shown in figure 5.4. The results of the cross-validation experiment are shown in figure 5.5. The inverse time

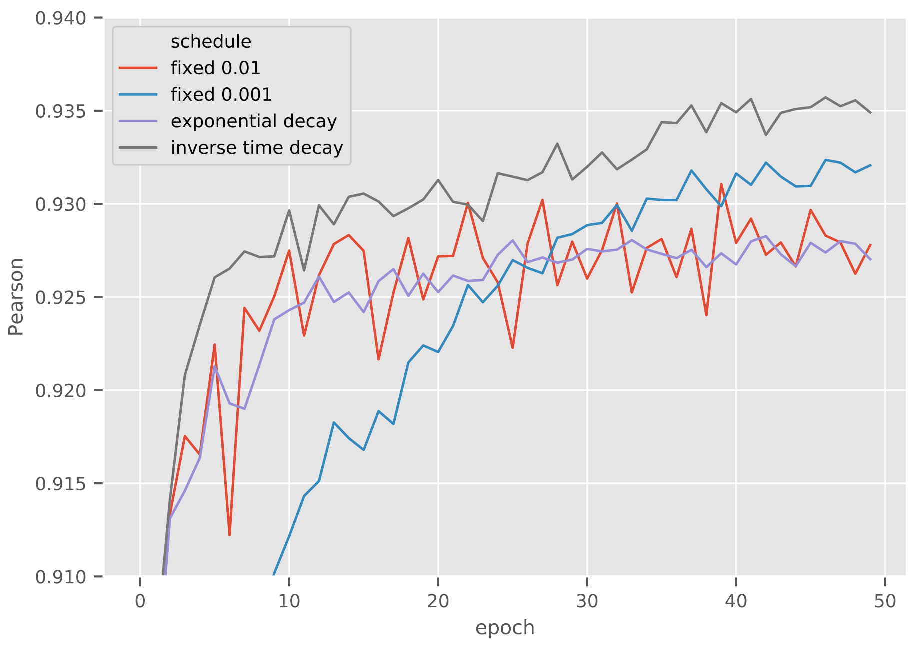

Figure 5.4: Typical example showing the effect of different learning rate annealing strategies on training LTR of order 5 and rank 10 on the full ALMANAC data set.

scheduling produces offers no improvement, whereas exponential scheduling results in a higher mean accuracy, with some notable outliers. Inspection of the learning rate graph showed signs of overfitting, so the issue should be resolvable with some sort of regularization or heuristic, such as early stopping.

5.2 Experiments with synthetic data

I conducted several experiments with artificial data to map the performance of LTR in various scenarios. The data generation procedure is described in

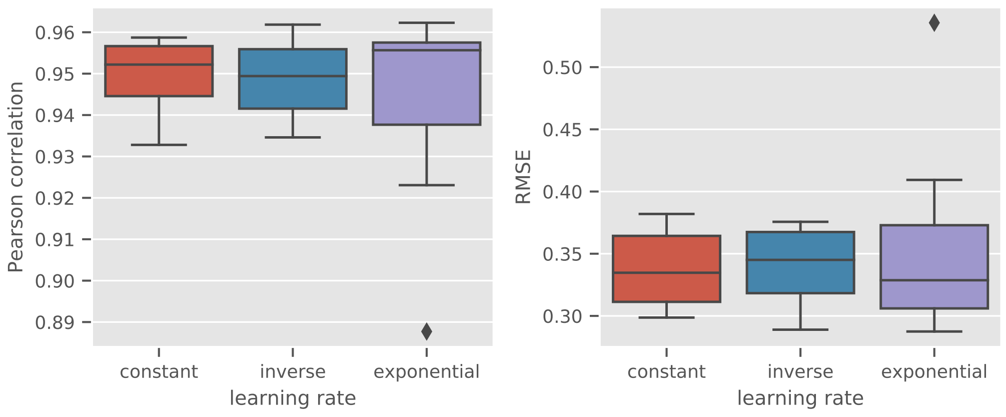

Figure 5.5: Cross validation results from using inverse and exponential learning rate scheduling.

section 4.2.1. In some of the experiments I also used HOFM as a point of comparison in order to see how the difference in design affects results.

5.2.1 Impact of rank

I trained LTR models with different ranks to see how performance varies around the rank that was used to generate the data. The synthetic data was generated using observations of dimension 20 and polynomials of order 3. The results are presented in figure 5.6.

As expected, there is a general pattern of higher performance with higher model rank, with only a few anomalous data points. It should also be pointed out that I did not use cross-validation here, so there is no measure of how much variance there is in the results. What is perhaps more surprising is that a rank 30 and even rank 20 models are able to achieve very high performance when the data is generated using a rank 50 polynomial. This would suggest that at least with data of intermediate dimensional containing fairly low order interactions (≤ 3) good Pearson coefficients and RMSE can be achieved even if the model rank is around half of the rank of the generating distribution. I also repeated the experiment with higher order factorization machines. The results are shown in figure 5.7.

Clearly HOFM is unable to learn the data at any of the ranks. This result is not particularly surprising, and is highly biased towards LTR, due to the artificial and unrealistic nature of the dataset.

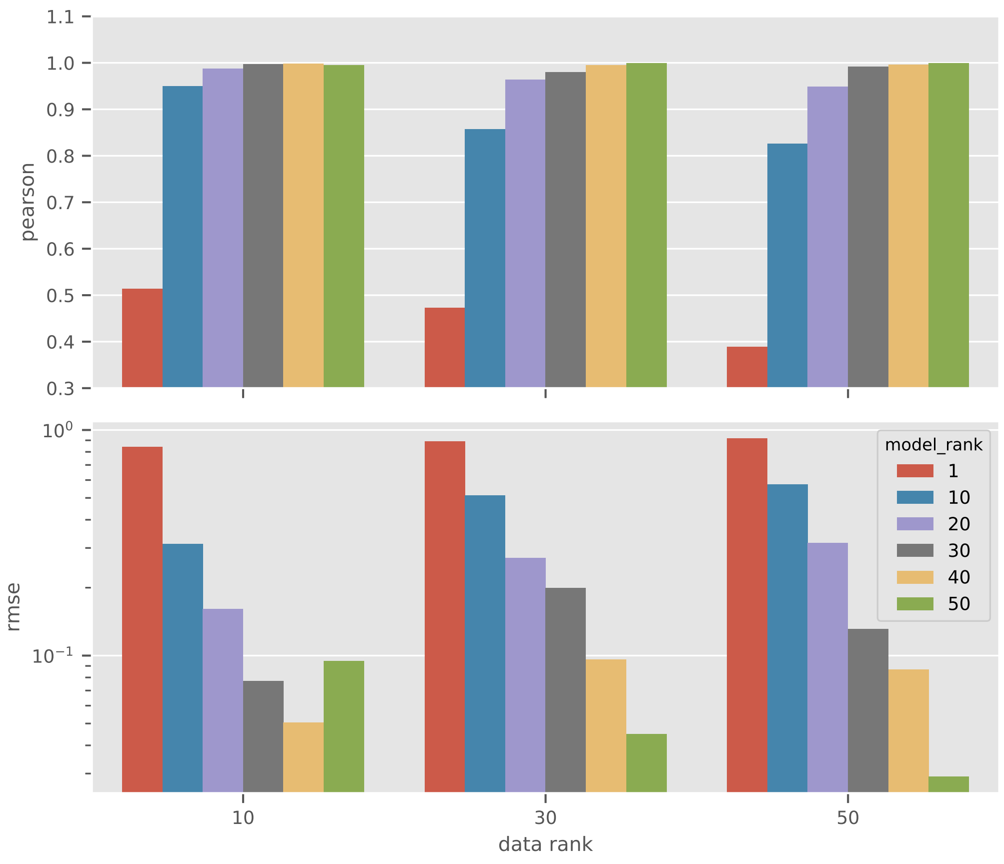

Figure 5.6: Prediction accuracy for LTR on the artificial data set.

5.2.2 Training time depending on order

A very important consideration is how long it actually takes to train models of increasing complexity. Just because gradient computations scale linearly with order does not mean training times do, if convergence takes longer. I conducted an experiment to test this by generating data using a polynomial of rank 10 and dimension 20. I then measured how long it would take a model of the correct order and rank to learn this data. A Pearson correlation coefficient of 0.95 or above was used as the criterion of the model having 'learned' the data. The results are presented in figure 5.8.

While there is a significant increase in required training time between orders 2 and 3, there is only a moderate increase past this. It is also notable that increasing the rank of the data and the model to 30 even speeds up learning at lower orders, and does not result in a particularly big increase in training times even at fifth order. I was unable to compare results with

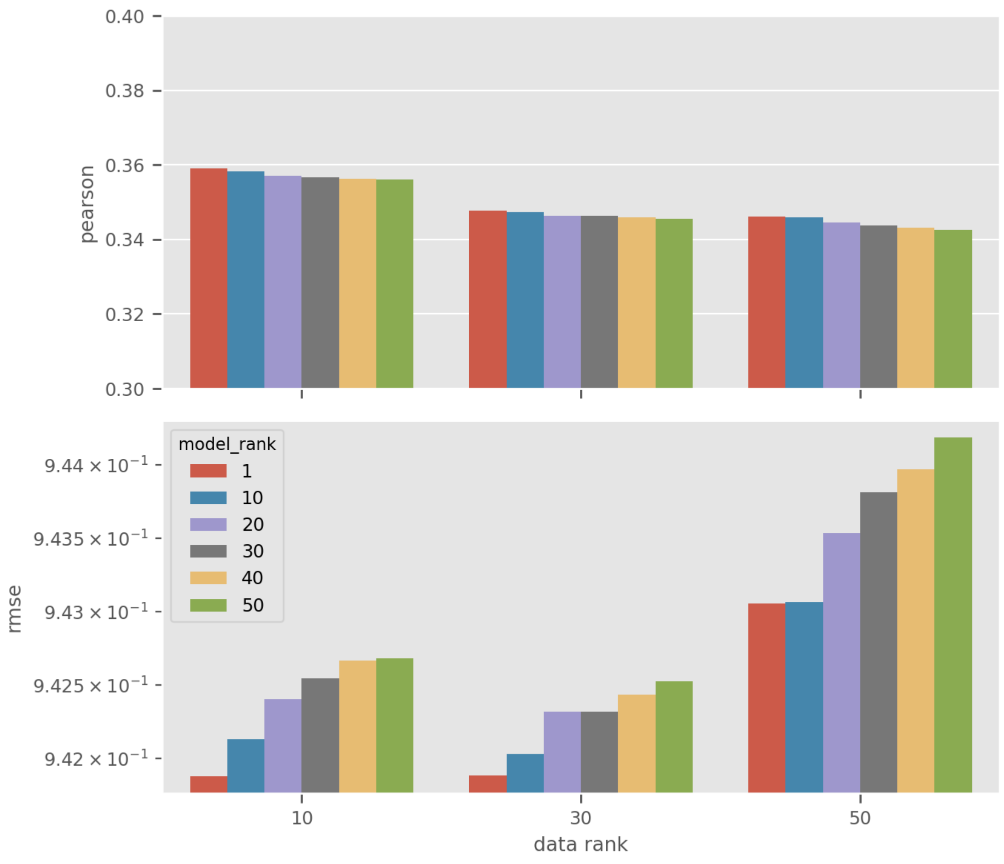

Figure 5.7: Prediction accuracy for HOFM on the artificial data set.

HOFM here, since as seen in the previous experiment, it was unable to learn the data.

5.3 Comparing LTR to other models

5.3.1 Comparison with HOFM

The natural point of comparison for LTR are higher order factorization machine, since they are both based on using a rank-one tensor factorization of a given order and rank. However comparing the models fairly and accurately is not trivial due to the different designs of the model. Despite the underlying tensor representation of the polynomials being the same, LTR has a significantly higher number of trainable parameters which, especially at higher orders, provides it with an advantage when in terms of expressive potential.

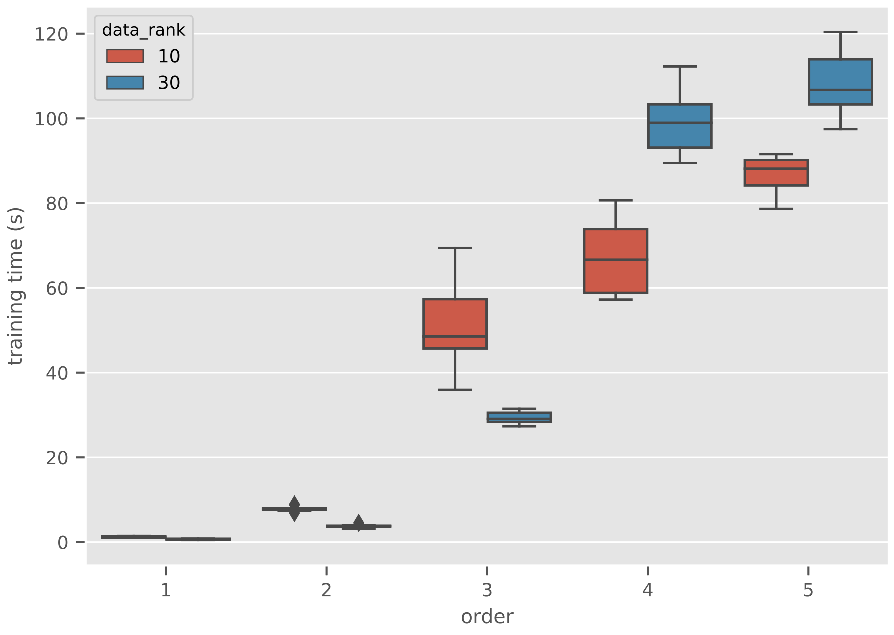

Figure 5.8: Training times for LTR as a function of order for the synthetic data.

The number of parameters in an LTR model is greater than that of HOFM by a factor corresponding to the order of the model.

I compared the performance of LTR and HOFM at different orders and ranks to see how they compare in terms of performance and training times. Figures 5.9 and 5.10 show a comparison between HOFM and LTR models of the same order at different ranks.

LTR consistently outperforms HOFM at the same order and rank, which is expected given that LTR is essentially a more expressive version of HOFM as explained in section 2.5.7. Notably LTR has a significantly smaller advantage in the 3rd order case, even having a higher mean RMSE at rank 50.

Another interesting question is how much data is needed to train LTR compared to HOFM. As mentioned in sections 2.1 and 2.5.7 it is to be expected that LTR requires more observations to learn the data due to the larger number of parameters. I conducted an experiment to see how this differs between LTR and HOFM by randomly sampling different numbers of observations from the training data. I used the same random samples for both algorithms at each cross-validation fold to keep the comparison accurate

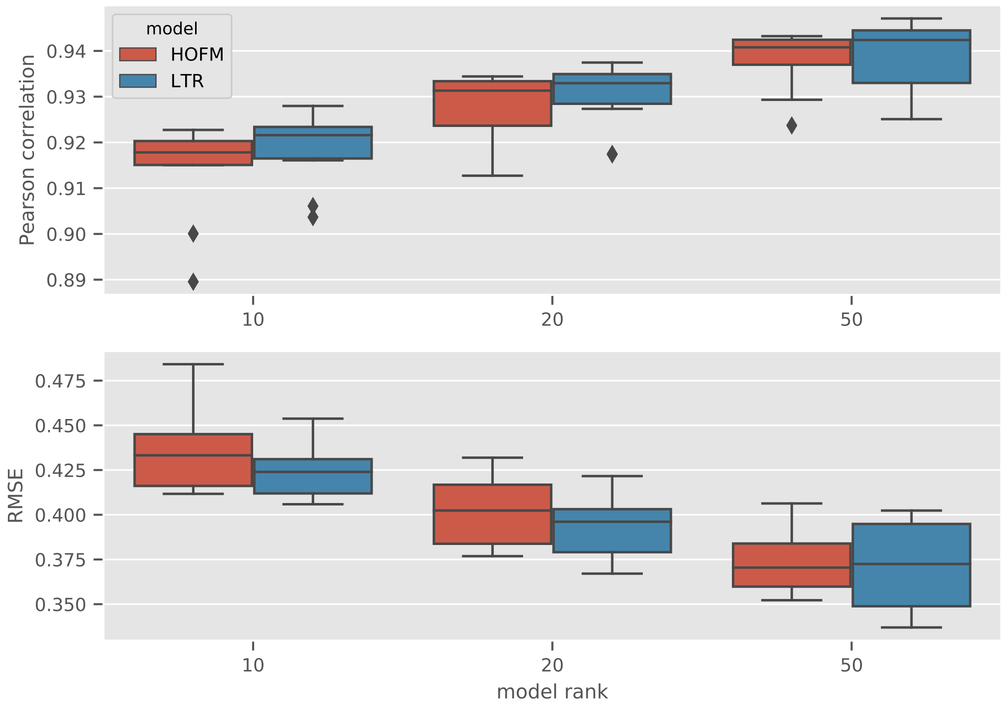

Figure 5.9: Comparison between 3rd order HOFM and LTR at rank 10, 20, and 50.

and reproducible. I chose to use rank 50 for both models, since it was already observed that LTR beats HOFM at this rank, but I wanted to observe whether this changes at lower ranks. I random subsamples of various sizes of the NCI-ALMANAC data set and trained both models with the subsample. The results are shown in figure 5.11.

As expected we see that HOFM has much better prediction accuracy when using subsets of 10% and 30% but when using the full data set LTR is able to pass HOFM in accuracy.

5.3.2 Multi-model comparison

In the final experiment I compared HOFM, LTR, and a neural network model side by side, using cross-validation to give an estimate of the relative generalization performance. For HOFM and LTR I performed a grid search with previously identified candidate hyperparameters. For HOFM the hyperparameters tried were the same as used by Heli Julkunen in her final experiment [29] and for LTR I based the set on the results gained in section 5.1. The

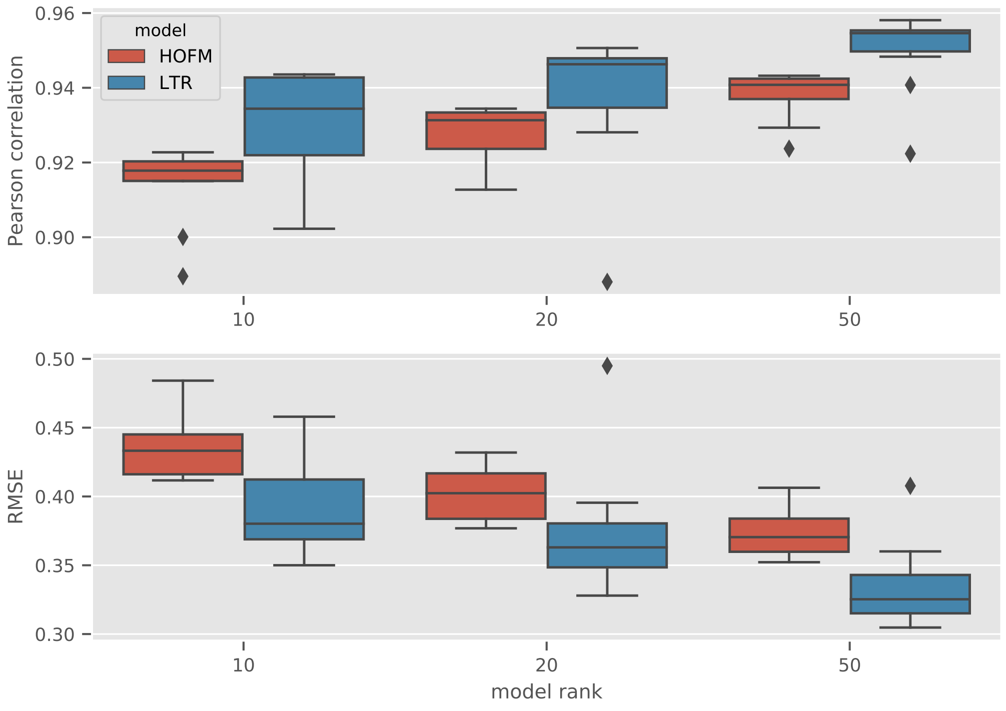

Figure 5.10: Comparison between 5th order HOFM and LTR at rank 10, 20, and 50.

hyperparameters optimized using grid search are listed in table 5.2.

For the LTR grid search I kept batch size fixed at 500, since it significantly affects the optimal values for other hyperparameters and this value was used in the preceding analysis. Adam optimizer with a fixed learning rate of 0.001 was used since it was found to produce good results and this was the optimization procedure used by Julkunen for HOFM. The hyperparameters optimized using the grid search were the regularization parameter and the rank of the model, since these were found to not have unambiguous optimal values and the choice of one affects the other. The number and size of layers for the neural network were mostly picked using trial and error. I found that performance did not significantly improve when increasing the number of layers or their size.

Mean Pearson correlation coefficient was used as the selection criterion for the hyperparameter optimization for LTR and HOFM. The hyperparameters chosen through this procedure were rank 75 for both models and regularization parameters 10^4 and 0.01 for HOFM and LTR respectively. Somewhat coincidentally the number of parameters for the best performing LTR model

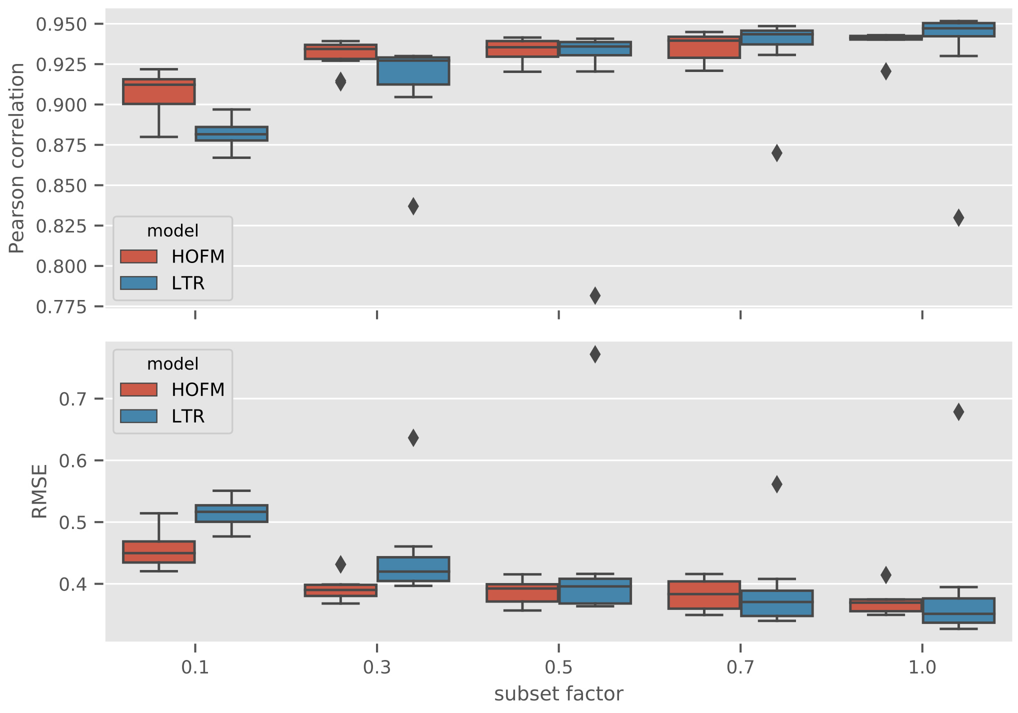

Figure 5.11: HOFM and LTR trained on various subsamples of the NCI-ALMANAC data.

and the neural network model are quite close to each other, making the comparison between them more fair. Table 5.3 lists the per-epoch training times and numbers of parameters for each of the models.

Remarkably all models had per-epoch running times in the same order of magnitude despite significant differences in structure and number of parameters. One rather likely explanation for this is that the speed of retrieving data from memory was the bottleneck during training and thus the underlying computation times might differ significantly. To investigate the matter further would require the analysis of GPU resource usage.

The final results of the experiment are presented in figure 5.12 and in table 5.4.

hyperparameter	HOFM	LTR
optimizer	Adam	Adam
learning rate	0.001	0.001
n epochs	200	200
order	5	5
regularization parameter	$\{10^3, 10^4, 10^5\}$	$\{1.0, 0.1, 0.01\}$
rank	$\{50, 75, 100\}$	$\{50, 75, 100\}$

Table 5.2: Hyperparameter values used in grid search.

Model	number of parameters	average epoch time (s)
HOFM	37500	7.78
LTR	187500	9.24
Neural network	205553	8.84

Table 5.3: Number of parameters and per-epoch training times for each of the three models compared.

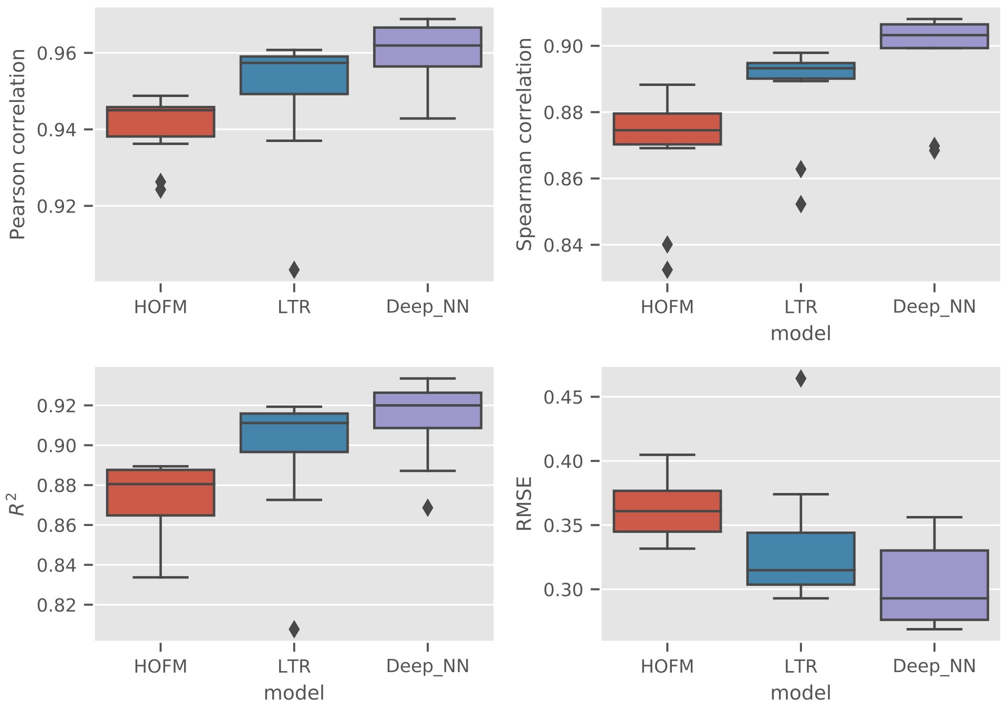

Figure 5.12: Comparison between the top performing HOFM and LTR models and a deep feedforward network.

Model	Pearson	Spearman	R^2	RMSE
HOFM	0.941	0.869	0.872	0.364
LTR	0.949	0.887	0.897	0.336
Neural network	**0.960**	**0.897**	**0.913**	**0.303**

Table 5.4: Mean values for the various accuracy metrics used in the comparison with the best values in each category in boldface.

The neural network model achieved the best results on all metrics, with HOFM being the worst performing.

Chapter 6

Conclusions

In this thesis I reviewed a variety of methods for performing regression on high-dimensional, high-rank, large data sets. I analyzed the definitions, theoretical performance characteristics, and reviewed relevant existing literature for all of them while comparing their strengths and weaknesses. I reviewed in detail the family of latent tensor based machine learning methods and how they can be reformulated in a way that revels their connection as variations of the same underlying design. I included the recently published method of latent tensor reconstruction and designed experiments to study how it performs in various scenarios.

I presented a new software implementation of latent tensor reconstruction which supports GPU acceleration. I found that training the model using a GPU resulted in significant performance improvements compared to training on a CPU. The design of the implementation is modular and flexible in a way that allows for easy extension and modification. In my experiments this implementation was comparable to the best available HOFM implementation in terms of running time.

I conducted a variety of experiments using both synthetic and real world data to study the absolute and relative performance characteristics of LTR. I found that LTR is able to learn polynomials of high order and rank in synthetic scenarios, a task where HOFMs failed. The running times to learn this type of synthetic data were found to scale favorably. I explored the range of optimal hyperparameters for LTR by performing cross-validation on the NCI-ALMANAC dataset. I identified approximately optimal hyperparameters for LTR for performing regression on this data set which I later used in evaluating relative performance compared to other models. I found that there is likely potential for improvement especially in terms of preventing overfitting at higher ranks by the use of different regularization methods or by the addition of heuristics to the training procedure.

I conducted an experiment with the aim of comparing the best performance of HOFMs, LTR and a feedforward neural networks on the NCI-ALMANAC prediction task. I performed a hyperparameter grid search to select optimal hyperparameters for HOFMs and LTR. For HOFMs I used the candidate hyperparameters identified by Heli Julkunen in her master's thesis [29] and for LTR I used the hyperparameters identified by my earlier exploration. I found that latent tensor reconstruction is able to outperform HOFMs in the task, but was unable to match the performance of a feedforward neural network with a similar number of parameters. HOFMs and LTR have an advantage over neural networks in terms of transparency, with parameters that are intuitive and can be analyzed to gain insights into the data.

The results I achieved with LTR almost certainly still have potential for improvement. I found the model to suffer from overfitting at higher ranks even on large data sets ($\sim 10^6$ observations). From a software perspective my implementation is quite basic and could likely be analyzed and fine tuned to optimize resource use and minimize overhead. It is possible that HOFMs could be subject to similar improvements, but considering the more refined software implementation and already considerable effort put into maximizing performance on this particular dataset by previous work [29] it seems less likely that dramatic improvements are possible. The neural network used in the comparison was not subject to any stringent hyperparameter optimization, so it is likely that performance could be improved further by changes to the network architecture, hyperparameters, or heuristics. This result is interesting by itself, and suggests that neural networks have potential in being applied to the combination treatment prediction task.

While the NCI-ALMANAC data set does exhibit the properties that formed the scope of the thesis, it still represents only a single dataset with no particular guarantees that performance in this particular task generalizes to other similar regression tasks. Thus many more experiments on datasets with a variety of structures and characteristics are required to truly understand and verify the strengths and weaknesses of each model for regression on high-rank, high-dimensional large data sets.

The topic of tensor based machine learning is under active research with the efficient formulation of higher order factorization machines by Blondel being published in 2016 and the significantly better performing LTR only being published this year. Thus it seems likely that improvements will continue to be to this class of methods that will extend their applicability.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Alpaydin, E. *Introduction to Machine Learning*, 2nd ed. The MIT Press, 2010.

[3] Amarís, M., de Camargo, R. Y., Dyab, M., Goldman, A., and Trystram, D. A comparison of gpu execution time prediction using machine learning and analytical modeling. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)* (2016), pp. 326–333.

[4] Andreu-Perez, J., Poon, C. C. Y., Merrifield, R. D., Wong, S. T. C., and Yang, G. Big data for health. *IEEE Journal of Biomedical and Health Informatics 19*, 4 (2015), 1193–1208.

[5] Babbar, R. Lecture notes of kernel methods in machine learning, January 2020.

[6] Baskin, I. I., Winkler, D., and Tetko, I. V. A renaissance of neural networks in drug discovery. *Expert Opinion on Drug Discovery 11*, 8 (2016), 785–795. PMID: 27295548.

[7] Bishop, C. M. *Pattern recognition and machine learning.* springer, 2006.

[8] Blondel, M., Fujino, A., Ueda, N., and Ishihata, M. Higher-order factorization machines, 2016.

[9] Blondel, M., Ishihata, M., Fujino, A., and Ueda, N. Polynomial networks and factorization machines: New insights and efficient training algorithms. *arXiv preprint arXiv:1607.08810* (2016).

[10] Bocci, C., and Chiantini, L. *An Introduction to Algebraic Statistics with Tensors*. Springer, 01 2019.

[11] Carroll, J. D., and Chang, J.-J. Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition. *Psychometrika 35* (1970), 283–319.

[12] Chen, D., Liu, S., Kingsbury, P., Sohn, S., Storlie, C. B., Habermann, E. B., Naessens, J. M., Larson, D. W., and Liu, H. Deep learning and alternative learning strategies for retrospective real-world clinical data. *NPJ digital medicine 2*, 1 (2019), 1–5.

[13] Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., Anil, R., Haque, Z., Hong, L., Jain, V., Liu, X., and Shah, H. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems* (New York, NY, USA, 2016), DLRS 2016, Association for Computing Machinery, p. 7–10.

[14] Datta, B. N. *Numerical linear algebra and applications*, vol. 116. Siam, 2010.

[15] Dozat, T. Incorporating nesterov momentum into adam.

[16] Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research 12*, 7 (2011).

[17] Duchi, J., Jordan, M. I., and McMahan, B. Estimation, optimization, and parallelism when data is sparse. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 2832–2840.

[18] Défossez, A., Bottou, L., Bach, F., and Usunier, N. On the convergence of adam and adagrad, 2020.

[19] Eckart, C., and Young, G. M. The approximation of one matrix by another of lower rank. *Psychometrika 1* (1936), 211–218.

[20] Foucquier, J., and Guedj, M. Analysis of drug combinations: current methodological landscape. *Pharmacology Research & Perspectives 3* (2015).

[21] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212* (2017).

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

[23] Gretton, A. Introduction to rkhs, and some simple kernel algorithms, October 2019.

[24] Györfi, L., Kohler, M., Krzyzak, A., and Walk, H. *A distribution-free theory of nonparametric regression*. Springer Science & Business Media, 2006.

[25] Hofmann, T., Schölkopf, B., and Smola, A. J. Kernel methods in machine learning. *The annals of statistics* (2008), 1171–1220.

[26] Holbeck, S. L., Camalier, R., Crowell, J. A., Govindharajulu, J. P., Hollingshead, M., Anderson, L. W., Polley, E., Rubinstein, L., Srivastava, A., Wilsker, D., Collins, J. M., and Doroshow, J. H. The national cancer institute almanac: A comprehensive screening resource for the detection of anticancer drug pairs with enhanced therapeutic activity. *Cancer Research 77*, 13 (2017), 3564–3576.

[27] Hornik, K., Stinchcombe, M., White, H., et al. Multilayer feedforward networks are universal approximators. *Neural networks 2*, 5 (1989), 359–366.

[28] Hyndman, R. J., and Koehler, A. B. Another look at measures of forecast accuracy. *International journal of forecasting 22*, 4 (2006), 679–688.

[29] Julkunen, H. Predictive modeling of anticancer efficacy of drug combinations using factorization machines. Master's thesis, Aalto University, Espoo, Finland, 5 2019.

[30] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization, 2014.

[31] Kitai, K., Guo, J., Ju, S., Tanaka, S., Tsuda, K., Shiomi, J., and Tamura, R. Designing metamaterials with quantum annealing and factorization machines. *Physical Review Research 2*, 1 (2020), 013319.

[32] Kitano, H. Cancer as a robust system: implications for anticancer therapy. *Nature reviews. Cancer 4*, 3 (March 2004), 227—235.

[33] Kolda, T. G., and Bader, B. W. Tensor decompositions and applications. *SIAM Review 51*, 3 (2009), 455–500.

[34] Kolda, T. G., Bader, B. W., and Kenny, J. P. Higher-order web link analysis using multilinear algebra. In *Fifth IEEE International Conference on Data Mining (ICDM'05)* (2005), pp. 8 pp.–.

[35] Larrañaga, P., Calvo, B., Santana, R., Bielza, C., Galdiano, J., Inza, I., Lozano, J. A., Armañanzas, R., Santafé, G., Pérez, A., and Robles, V. Machine learning in bioinformatics. *Briefings in Bioinformatics 7*, 1 (03 2006), 86–112.

[36] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *Nature 521* (2015), 436–444.

[37] Masters, D., and Luschi, C. Revisiting small batch training for deep neural networks. *CoRR abs/1804.07612* (2018).

[38] Mikhail Trofimov, A. N. tffm: Tensorflow implementation of an arbitrary order factorization machine. `https://github.com/geffy/tffm`, 2016.

[39] Mohri, M., Rostamizadeh, A., and Talwalkar, A. *Foundations of Machine Learning.* The MIT Press, 2012.

[40] Myers, J. L., Well, A., and Lorch, R. F. *Research design and statistical analysis.* Routledge, 2010.

[41] Qian, N. On the momentum term in gradient descent learning algorithms. *Neural networks 12*, 1 (1999), 145–151.

[42] Raschka, S. Machine learning faq: How do i evaluate a model.

[43] RENDLE, S. Factorization machines. In *2010 IEEE International Conference on Data Mining* (2010), pp. 995–1000.

[44] RENDLE, S. Factorization machines with libFM. *ACM Trans. Intell. Syst. Technol. 3*, 3 (May 2012), 57:1–57:22.

[45] RODGERS, J. L., AND NICEWANDER, W. A. Thirteen ways to look at the correlation coefficient. *The American Statistician 42*, 1 (1988), 59–66.

[46] ROSS, A. S., AND DOSHI-VELEZ, F. Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients. *CoRR abs/1711.09404* (2017).

[47] RUDER, S. An overview of gradient descent optimization algorithms, 2016.

[48] SCHLKOPF, B., SMOLA, A. J., AND BACH, F. *Learning with kernels: support vector machines, regularization, optimization, and beyond.* the MIT Press, 2018.

[49] SHALEV-SHWARTZ, S., AND BEN-DAVID, S. *Understanding machine learning: From theory to algorithms.* Cambridge university press, 2014.

[50] SHAWE-TAYLOR, J., CRISTIANINI, N., ET AL. *Kernel methods for pattern analysis.* Cambridge university press, 2004.

[51] STEEL, R. G., AND TORRIE, J. H. Principles and procedures of statistcs with special reference to the biological sciences. Tech. rep., McGraw-Hill Book Company, Inc., 1960.

[52] SUTSKEVER, I. *Training Recurrent Neural Networks.* PhD thesis, University of Toronto, 2013.

[53] SZEDMAK, S., CICHONSKA, A., JULKUNEN, H., PAHIKKALA, T., AND ROUSU, J. A solution for large scale nonlinear regression with high rank and degree at constant memory complexity via latent tensor reconstruction. *arXiv preprint arXiv:2005.01538* (2020).

[54] VAN LOAN, C. F., AND GOLUB, G. H. *Matrix computations.* Johns Hopkins University Press Baltimore, 1983.

[55] VARMA, S., AND SIMON, R. Bias in error estimation when using cross-validation for model selection." bmc bioinformatics, 7(1), 91. *BMC bioinformatics 7* (02 2006), 91.

[56] Wainer, J., and Cawley, G. Nested cross-validation when selecting classifiers is overzealous for most practical applications. *arXiv preprint arXiv:1809.09446* (2018).

[57] Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., and Recht, B. The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 4148–4158.

[58] Zhang, Z., Beck, M. W., Winkler, D. A., Huang, B., Sibanda, W., Goyal, H., et al. Opening the black box of neural networks: methods for interpreting neural network models in clinical applications. *Annals of translational medicine 6*, 11 (2018).

Appendix A

Definitions

Observation	data point, row, or sample in a data set.
Feature vector	a list of features describing an observation
Model	mathematical model and associated data
Learner	combination of model and optimizer
Learning algorithm	an algorithm which produces a trained model
Loss function	a measure of prediction error
Step	operations that result in a single parameter update
Epoch	feeding the whole dataset to a learner once
Input space	the set of all possible inputs to a model
Hypothesis space	the set of all mappings from data to labels
Hyperparameter	parameter for the learning algorithm

Appendix B

Model Equation in TF2

```
import tensorflow as tf

@tf.custom_gradient
def model_equation(lambda_vec, *P_list):

    #  Compute the XP^T matrices
    constituents = [tf.matmul(X, tf.transpose(P)) for P in P_list]
    #  Compute the F matrix
    F = tf.math.reduce_prod(tf.stack(constituents, axis=2), axis=2)
    #  Multiply by lambda but don't sum reduce
    pre_F = F * tf.transpose(lambda_vec)

    def grad(dy):
        lambda_grad = tf.expand_dims(tf.reduce_sum(F * dy, axis=0),
                                     axis=-1)
        P_grads = [
            tf.transpose(tf.matmul(tf.transpose(X),
                tf.math.divide(pre_F, c) * dy))
            for c in constituents]
        #  No need for gradient w.r.t. X so we don't compute it
        return (None, lambda_grad, *P_grads)

    return tf.expand_dims(tf.reduce_sum(pre_F, axis=1), axis=-1), grad
```

Appendix C

Synthetic Data Generation

```
import numpy as np
from functools import reduce

def compute_F(_X, P_list):
    constituents = [np.matmul(_X, np.transpose(P)) for P in P_list]
    return reduce(lambda a, b: np.multiply(a, b), constituents)

def generate_data(nsamples, dimension, order, rank):
    P_matrices = [np.random.uniform(-1 ,1, size=(rank, dimension))
                  for i in range(order)]
    X = np.random.normal(size=(nsamples, dimension))

    y = np.matmul(compute_F(X, P_matrices), np.ones(rank) / rank)
    y = (y - np.mean(y)) / np.sqrt(np.var(y))

    return X, y
```