VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

some kind of rnn/tensor mess

Paul Francis Cunninghame Mathews

Supervisors: Marcus Frean and David Balduzzi

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

A short description of the project goes here.

Acknowledgments

Any acknowledgments should go in here, between the title page and the table of contents. The acknowledgments do not form a proper chapter, and so don't get a number or appear in the table of contents.

Contents

Figures

Chapter 1

Introduction

This chapter gives an introduction to the project report.

In Chapter **??** we explain how to use this document, and the `vuwproject` style. In Chapter **??** we say some things about LaTeX, and in Chapter 6 we give our conclusions.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Feed-Forward Neural Networks

This is to be pretty brief. Cover useful things – what they look like, gradient descent (in brief) and outline some results on the expressive power.

2.1.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) of the form considered here generalise feed-forward networks to address problems in which we wish to map a *sequence* of inputs $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \dots \mathbf{x}_T)$ to a sequence of outputs $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2, \dots \mathbf{y}_T)$. They have been applied successful to a wide range of tasks which can be framed in this way including statistical language modelling [29] (including machine translation [5]), speech recognition [13], polyphonic music modelling [3], music classification [6], image generation [14], automatic captioning [44, 46] and more.

Original Formulation

An RNN is able to maintain context over a sequence by transferring its hidden state from one time-step to the next. We refer to the vector of states at time t as $\mathbf{h}_t$.

The classic RNN (often termed "vanilla") originally proposed in [11] computes its hidden states with the following recurrence:

$$\mathbf{h}_t = f(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}) \tag{2.1}$$

where $f(\cdot)$ is some elementwise non-linearity, often the hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.

Equation (2.1) bears a striking resemblance to the building block of a feed-forward network. The key difference is the (square) matrix $\mathbf{W}$ which contains weights controlling how the previous state affects the computation of the new activations.

Training

We can train this (or any of the variants we will see subsequently) using back-propagation. Often termed "Back Propagation Through Time" [45] which requires using the chain rule to determine the gradients of the loss with respect to the network parameters in the same manner as for feedforward networks.

To understand what is required to perform this, consider a loss function for the whole sequence of the form

$$\mathcal{L}(\hat{\mathbf{y}}_1, \hat{\mathbf{y}}_2, \ldots, \hat{\mathbf{y}}_T, \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_T) = \sum_{i=1}^{T} \mathcal{L}_i(\hat{\mathbf{y}}_i, \mathbf{y}_i),$$

which is a sum of the loss accrued at each time-step. This captures all common cases including sequence classification or regression, as the $\mathcal{L}_i$ may simply return 0 for all but the last time-step. To find gradients of the loss with respect the parameters which generate the hidden states, we must first find the gradient of the loss with respect to the hidden states themselves. Choosing a hidden state i somewhere in the sequence we have:

$$\nabla_{\mathbf{h}_i} \mathcal{L} = \sum_{j=i}^{t} \nabla_{\mathbf{h}_i} \mathcal{L}_j$$

from the definition of the loss and the fact that a hidden state may affect all future losses. To determine each $\nabla_{\mathbf{h_i}} \mathcal{L}_j$ (noting $j \geq i$), we apply the chain rule, to back-propagate the error from time j to time i. This is the step from which the algorithm derives its name, and simply requires multiplying through adjacent timesteps. Let $\mathbf{z}_k = \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}$ be the pre-activation of the hidden states. Then

$$\begin{aligned}
\nabla_{\mathbf{h}_i} \mathcal{L}_j &= (\nabla_{\mathbf{h}_j})^{\mathsf{T}} \mathcal{L}_j \left(\prod_{k=i+1}^{j} \frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}} \right) \\
&= (\nabla_{\mathbf{h}_j} \mathcal{L}_j)^{\mathsf{T}} \left(\prod_{k=i+1}^{j} \nabla_{\mathbf{z}_k} f \cdot \mathbf{W} \right). \qquad (2.2)
\end{aligned}$$

This has two key components: $\nabla_{\mathbf{h}_j} \mathcal{L}_j$ quantifies the degree to which the hidden states at time j affect the loss (computing this will most likely require further back-propagation through one or more output layers) while the second term in equation (2.2) measures how much the hidden state at time i affects the hidden state at time j.

We can now derive an update rule for the parameters by observing

$$\begin{aligned}
\nabla_{\mathbf{W}} \mathcal{L} &= \sum_{i=1}^{T} \nabla_{\mathbf{W}} \mathcal{L}_i \\
&= \sum_{i=1}^{T} \sum_{j=1}^{i} \nabla_{\mathbf{h}_j} \mathcal{L}_i \nabla_{\mathbf{W}} \mathbf{h}_j
\end{aligned}$$

shapes, get them right

and applying the above. For the input matrix and the bias the process is the same.

Issues

Equation (2.2) reveals a key pathology of the vanilla RNN – vanishing gradients. This occurs when the gradient of the loss vanishes to a negligibly small value as we propagate it backward in time, leading to a negligible update to the weights. $(\nabla_{\mathbf{h}_j} \mathcal{L}_j)^{\mathsf{T}}$ (a vector) is multiplied by a long product of matrices, alternating between $\nabla_{\mathbf{z}_k} f$ and $\mathbf{W}$. If we assume for illustrative purposes that $f(\cdot)$ is the identity function (so we have a linear network), then the loss vector is multiplied by $\mathbf{W}$ taken to the $(j-i)$th power. If the largest eigenvalue of $\mathbf{W}$ is large, then this will cause the gradient to eventually explode. If the largest eignevalue is small, then the gradient will vanish. This issue was first presented in 1994 [2], for a thorough treatment including necessary conditions for vanishing and the complementary exploding problem, see [35].

In the non-linear case, this remains a serious issue. While exploding gradients are often mitigated by using a *saturating* non-linearity so that the gradient tends to zero as the hidden states grow, this only exacerbates the vanishing problem.

A second issue when training RNNs can be illustrated by viewing them as iterated non-linear dynamical systems and thus susceptible to the "butterfly effect": seemingly negligible changes in initial conditions can lead to catastrophic changes after a number of iterations [27]. In RNNs this manifests as near-discontinuity of the loss surface [35] as a change (for example, to a weight during back-propagation) which may even reduce the loss for a short period can cause instabilities further on which lead to steep increases in loss. This problem is not as well studied as vanishing gradients although some partial solutions exist such as clipping the norm of gradients [35] or using a regulariser to encourage gradual changes in hidden state [25].

Alternate Architectures

To address these fundamental problems a number of alternate architectures have been proposed. Here we will outline two popular variants: the Long Short Term Memory (LSTM) and the Gated Recurrent Unit (GRU). Both of these belong to a class of *gated* RNNs, which have a markedly different method of computing a new state.

The LSTM was proposed to alleviate the vanishing gradient problem. It uses a number of gates to control the flow of information during the computation of new states. Although a large number of variants exist, the standard LSTM we will consider here has the form: [21, 12]

$$\begin{aligned}
\mathbf{h}_t &= \mathbf{o}_i \odot \tau(\mathbf{c}_t) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \\
\mathbf{g}_t &= \tau(\mathbf{W}_g \mathbf{c}_{t-1} + \mathbf{U}_g \mathbf{x}_t + \mathbf{b}_g) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_o \mathbf{c}_{t-1} + \mathbf{U}_o \mathbf{x}_t + \mathbf{b}_o) \\
\mathbf{f}_t &= \sigma(\mathbf{W}_f \mathbf{c}_{t-1} + \mathbf{U}_f \mathbf{x}_t + \mathbf{b}_f) \\
\mathbf{i}_t &= \sigma(\mathbf{W}_i \mathbf{c}_{t-1} + \mathbf{U}_i \mathbf{x}_t + \mathbf{b}_i)
\end{aligned}$$

where $\tau(\cdot)$ refers to the elementwise tanh, $\sigma(\cdot)$ the elementwise logistic sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ and $\cdot \odot \cdot$ is used to denote elementwise multiplication between vectors. These equations can be hard to take at face value – the key elements are $\mathbf{i}_t, \mathbf{o}_t, \mathbf{f}_t$, termed the *input*, *output* and *forget* gates respectively, are computed in the same fashion as the activations of a vanilla neural network but use sigmoid activation function which varies smoothly between zero and one. Combining this with elementwise multiplication has the eponymous gating effect, attenuating the contributions of other components.

The output gate is fairly straightforward, it simply allows the network to prevent its hidden state from being exposed. The forget and input gates have a more difficult role to characterise. Taken together, these control the acceptance or rejection of new information by modulating the amount by which the new candidate state $\mathbf{g}_t$ is accepted into the hidden state $\mathbf{c}_t$.

A closely related architecture proposed much more recently by Cho et al. [5] is the GRU, which computes its state as follows:

$$\begin{aligned}
\mathbf{h}_t &= \mathbf{f}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{f}_t) \odot \mathbf{z}_t \\
\mathbf{z}_t &= \tau(\mathbf{W}_z(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{U}_z\mathbf{x}_t + \mathbf{b}_z) \\
\mathbf{f}_t &= \sigma(\mathbf{W}_f\mathbf{h}_{t-1} + \mathbf{U}_f\mathbf{x}_t + \mathbf{b}_f) \\
\mathbf{r}_t &= \sigma(\mathbf{W}_r\mathbf{h}_{t-1} + \mathbf{U}_r\mathbf{x}_t + \mathbf{b}_r).
\end{aligned}$$

This is a slightly simpler form than the LSTM although the alterations go beyond simply removing the output gate. Notably, the forget gate now controls both parts of the state update. Further, in the computation of $\mathbf{z}_t$ there is a departure from the vanilla RNN-style building block that makes up all of the LSTM's operations. This is interesting as a half of the model's parameters, and therefore a large part of its computational power, is dedicated towards computing state updates. However, the mechanism of their computation places significant emphasis on using temporally local information to do so – the *reset* gate $\mathbf{r}_t$ provides the model with the ability to ignore parts of its state.

The key shared component of these architectures is an *additive* state update. Another way of phrasing this is that while the vanilla RNN attempts to learn an opaque function $\mathbf{h}_t = \mathcal{F}(\mathbf{x}_t, \mathbf{h}_{t-1})$, these gated architectures instead learn a *residual* mapping $\mathbf{h}_t = \mathcal{F}(\mathbf{x}_t, \mathbf{h}_{t-1}) + \mathbf{h}_{t-1}$. By itself, this would alleviate the main cause of vanishing gradients [22, 21]. This is equivalent to adding a skip connection, allowing the state to skip a time-step and is directly analogous to the residual connections now commonly used to address the vanishing gradient problem in very deep feed-forward networks [16, 10, 40]. Unfortunately the presence of the gate complicates this perspective – this will be analysed in detail in chapter 4.

2.2 Related Work

2.2.1 Long Time Dependencies

The key symptom of vanishing gradients in RNNs is that it makes it much harder to learn to store information for long time periods [2]. This makes RNNs often struggle to solve simple-seeming tasks in which the solution requires remembering an input for many time-steps. There are two main categories of solutions to this issue – architectural and algorithmic.

Architectural Solutions

These attempts to solve the problem focus on alleviating the issue by changing the manner in which hidden states are calculated. The aforementioned LSTM and GRU are the most widespread, but several alternatives have been proposed. An early attempt to solve the problem involves feeding inputs to different units at different rates. The primary motivation for this seems to be to force the network to attend to longer time-scales by passing some parts of it information at a slower rate, although the authors note an intuition that more abstract representations of the sequence will change slower and hence not require re-calculation at every step [18]. This idea has been revisited recently in the form of the Clockwork RNN [24] which flattens the network, simply forcing the various hidden states to be updated at different predefined rates. Further approaches in this direction attempt to have the network learn the time scale at which it should update – recent examples include Hierarchical Multiscale RNNs [7] and Gated Feedback RNNs [8] which suggest various ways of connecting, via additional gated connections, the outputs of various layers in a stacked RNN architecture.

These approaches have been successful, although it is not precisely clear how adding extra multiplicative gates could alleviate the vanishing gradient problem. Secondly, many of these approaches add significant extra structures to the network (typically a large LSTM) to force it to behave in a manner which it was already capable of. That significant extra structures need to be added to able to reliably train it to express such behaviours seems to suggest that a more fundamental change in the model would be wise.

A simpler recent architecture of note is the Unitary Evolution RNN [1] which guarantee the eigenvalues of the recurrent weight matrix have a magnitude of one. While this leads to provably non-vanishing or exploding gradients, in practice they still seem to struggle to learn to store information for very long time periods. This is potentially due to the seemingly ad-hoc composition of unitary operators used to allow unconstrained optimisation of parameters while maintaining the desired properties of the recurrent matrix.

Strongly Typed

Another line of enquiry which can help with long time-dependencies focuses on the initialisation of the network. IRNNs [26], which simply initialise the recurrent weights matrix to the identity (presaged by the nearly diagonal initialisation in [30]), perform remarkably well on pathological tasks. The importance of the initialisation of the recurrent weights matrix was further emphasised in [17] where marked differences were found between initialising a slightly modified vanilla RNN with the identity matrix or a random orthogonal matrix. While simply initialising the network to have certain beneficial properties provides no guarantees on final performance after training, it is important to note the significant results reported especially as they are likely to at least be partially transferable to any given architecture.

Algorithmic Solutions

The second class of attempts to address the issue focuses on techniques for training the network that help overcome the problems with the gradients. The first of this is to use approximate second-order methods to try and rescale the gradients appropriately. A number of notable attempts used Hessian-Free Optimization [28, 3] and achieved remarkable results. Perhaps more interestingly, it was subsequently found that many of the results could in fact be achieved with very careful tuning of standard stochastic gradient descent with momentum and careful initialisation [39].

Another interesting approach which seems to help learn networks learn tasks requiring longer time dependencies is to add a regularisation term into the loss to penalise the difference between successive hidden states [25]. This encourages the network to learn smooth transitions and may help it "bootstrap" itself past the vanishing gradients.

2.2.2 Memory

Something about using memory, needing to add extra memory/weird ways of using it to make LSTMs et al. do cool stuff. Repeat observation that having to go to all this trouble indicates the LSTM is the issue.

2.2.3 Tensors in Neural Networks

Including gated networks, MRNN and so on.

Chapter 3

Tensors

In order to represent functions of two arguments with neural networks, three-way tensors are unavoidable. In this chapter discuss the vector-tensor-vector bilinear product. We look at various low-rank tensor decompositions to avoid having to store the tensor in full with a view towards assessing their suitability to be learnt in situ with gradient descent.

It is also important to understand conceptually what the bilinear product does. It admits a surprising number of interpretations including theorem 1 – that it allows us to operate on a pairwise exclusive-or of features. Additionally, using different tensor decompositions to represent the tensor corresponds to emphasising different modes of interpretation. Making these relationships clear is then an important step in making a principled choice of decomposition to insert into a network architecture.

Finally, we undertake preliminary experiments to investigate empirically the feasibility of learning tensor decompositions in practice.

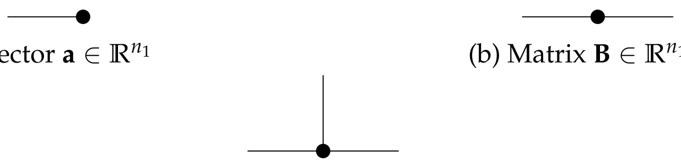
3.1 Definitions

Formally, we refer to a multi-dimensional array which requires n indices to address a single (scalar) element as a n-way tensor and occasionally refer to n as the number of dimensions of the tensor. In this sense a matrix is a two-way tensor and a vector a one-way tensor, although we will use their usual names for clarity. Notationally, we attempt to stick to the notation used in [23] deviating only where it would become unwieldy. We denote tensors with three or more dimensions with single calligraphic boldface letters such as $\boldsymbol{\mathcal{W}}$. Matrices and vectors will be denoted with upper and lower case boldface letters while scalars will be standard lower case letters. Commonly we will need to address particular substructures of tensors. This is analogous to pulling out individual rows or columns of a matrix. To perform this we fix some of the indices to specific values and allow the remaining indices to vary across their range. We denote by $\cdot$ the indices allowed to vary, the rest will be provided with a specific value. For example, $\mathbf{A}_{i\cdot}$ would denote the i-th row of the matrix $\mathbf{A}$. For three-way tensors we refer to the vector elements produced by fixing two indices as *threads*. It is also possible to form matrices by fixing only one index – we refer to these as *slices*. Table 3.1 provides examples of the possibilities.

When dealing with tensor-tensor products in general, it is important to be precise as there are often a number of possible permutations of indices that would lead to a valid operation. The downside of this is that it leads to unwieldy notation. Fortunately, we are only concerned with a a couple of special cases. In particular, we need to multiply a three-tensors by vectors and a matrices by vectors. Matrix-vector multiplication consists of taking the dot product of the vector with each row of the matrix. For example, with a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ and a vector

Description	*Example*
scalar	a
vector	$\mathbf{b}$
matrix	$\mathbf{C}$
higher order tensor	$\boldsymbol{\mathcal{D}}$
element of vector (scalar)	b_i
element of matrix (scalar)	C_{ij}
element of 3-tensor (scalar)	D_{ijk}
row of matrix (vector)	$\mathbf{C}_{i\cdot}$
column of matrix (vector)	$\mathbf{C}_{\cdot i}$
fiber of 3-tensor (vector)	$\mathbf{D}_{\cdot jk}$
slice of 3-tensor (matrix)	$\mathbf{D}_{\cdot\cdot k}$

Table 3.1: Example of notation for tensors.



(a) Vector $\mathbf{a} \in \mathbb{R}^{n_1}$

(b) Matrix $\mathbf{B} \in \mathbb{R}^{n_1 \times n_2}$

(c) Three-way tensor $\boldsymbol{\mathcal{C}} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$

Figure 3.1: Example tensor network diagrams.

$\mathbf{x} \in \mathbb{R}^n$, if $\mathbf{y} = \mathbf{A}\mathbf{x}$ (with $\mathbf{y}$ necessarily in $\mathbb{R}^m$), then

$$\begin{aligned} y_i &= \sum_j^n A_{ij} x_j \\ &= \langle \mathbf{A}_i, \mathbf{x} \rangle \end{aligned}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner (dot) product. This can be viewed as taking all of the vectors formed by fixing the first index of $\mathbf{A}$ while allowing the second to vary and computing their inner product with $\mathbf{x}$. To perform the same operation using the columns of $\mathbf{A}$ we need to fix the second index, the would typically be done by exchanging the order: $\mathbf{x}^\mathsf{T}\mathbf{A}$. This kind of operation is sometimes referred to as a *contractive* product, especially in the physical sciences [33]. This name arises because we form the output by joining a shared dimension (by elementwise multiplication) and contracting it with a sum.

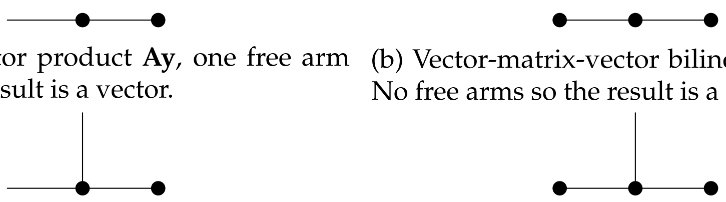
We can generalise the operation to tensors: choose an index over which to perform the product, collect every thread formed by fixing all but that one index and compute their inner product with the given vector. If the tensor has n indices, the result will have $n-1$. A three tensor is in this way reduced to a matrix. Kolda and Bader introduce the operator $\cdot \bar{\times}_i \cdot$ for this, where i represents the index to vary [23]. For the bilinear forms we are concerned with, this leads to the following notation:

$$\begin{aligned} \mathbf{z} &= \boldsymbol{\mathcal{W}} \;\bar{\times}_1\; \mathbf{x} \;\bar{\times}_2\; \mathbf{y} && (3.1) \\ &= \boldsymbol{\mathcal{W}} \;\bar{\times}_3\; \mathbf{y} \;\bar{\times}_1\; \mathbf{x}. && (3.2) \end{aligned}$$

When $\boldsymbol{\mathcal{W}}$ is a three-way tensor, we prefer a more compact notation

$$\mathbf{z} = \mathbf{x}^\mathsf{T}\boldsymbol{\mathcal{W}}\mathbf{y}. \tag{3.3}$$

This loses none of the precision of the more verbose notation, provided we make clear that we intend $\mathbf{x}$ to operate along the first dimension of the tensor and $\mathbf{y}$ the third. That is to say,



(a) Matrix-vector product $\mathbf{Ay}$, one free arm indicates the result is a vector.

(b) Vector-matrix-vector bilinear form $\mathbf{x}^\mathsf{T}\mathbf{Ay}$. No free arms so the result is a scalar.

(c) Tensor-vector product produces a matrix.

(d) Vector-tensor-vector product produces a vector.

Figure 3.2: Various products expressed as Tensor Network Diagrams.

$(\mathbf{x}^\mathsf{T}\boldsymbol{\mathcal{W}})\mathbf{y}$ exactly corresponds with equation (3.1) while $\mathbf{x}^\mathsf{T}(\boldsymbol{\mathcal{W}}\mathbf{y})$ corresponds to equation (3.2). With either notation, for a tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, we must have that $\mathbf{x} \in \mathbb{R}^{n_1}$, $\mathbf{y} \in \mathbb{R}^{n_3}$ and the result $\mathbf{z} \in \mathbb{R}^{n_2}$.

An intuitive way to illustrate these ideas is using Tensor Network Diagrams [9, 33]. In these diagrams, each object is represented as a circle, with each free 'arm' representing an index used to address elements. A vector therefore has one free arm, a matrix two and so on. Scalars will have no arms. Figure 3.1 has examples for these simple objects.

Where these diagrams are especially useful is for representing contractive products where we sum over the range of a shared index. We represent this by joining the respective arms. As an example, a matrix-vector product $\mathbf{y} = \mathbf{Ax}$ has such a contraction: $y_i = \sum_j A_{ij}x_j$. This is shown in figure 3.2a – it is clear that there is only a single free arm, so the result is a vector as it should be. Figure 3.2 shows some examples of these kinds of products.

3.2 Bilinear Products

There are several ways to describe the operation performed by the bilinear products we are concerned with. These correspond to different interpretations of the results. The following interpretations provide insight both into what is actually being calculated and how the product might be applicable to a neural network setting. In all of the below we use the above definitions of $\mathbf{x}, \mathbf{y}, \mathbf{z}$ and $\boldsymbol{\mathcal{W}}$.

3.2.1 Interpretations

Stacked bilinear forms

If we consider the expression for a single element of $\mathbf{z}$, we get

$$z_j = \sum_i^{n_1} \sum_k^{n_3} W_{ijk} x_i y_k$$

as expected. We can re-write this in terms of the slices of $\boldsymbol{\mathcal{W}}$:

$$z_j = \mathbf{x}^\mathsf{T} \boldsymbol{\mathcal{W}}_{\cdot j \cdot} \mathbf{y}$$

which reveals the motivation behind the notation in equation (3.3). It also reveals that each element of $\mathbf{z}$ is itself linear in $\mathbf{x}$ or $\mathbf{y}$ if the other is held constant.

This provides an interpretation in terms of similarities. If we consider the standard dot product of two vectors $\mathbf{a}$ and $\mathbf{b}$ of size m:

$$\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^\mathsf{T}\mathbf{b} = \sum_{i=1}^{m} a_i b_i = \cos\theta ||\mathbf{a}||_2 ||\mathbf{b}||_2$$

where θ is the angle in the angle between the vectors. If the product is positive, the two vectors are pointing in a similar direction and if it is negative they are in opposite directions. If it is exactly zero, they must be orthogonal. The dot product therefore provides us with some notion of the similarity between two vectors. Indeed if we normalise the vectors by dividing each component by their l_2 norm we recover exactly the widely used cosine similarity, common in information retrieval [37, 41] . Note that we can generalise this idea by inserting a matrix of (potentially learned) weights $\mathbf{U}$ which enables us to define general scalar bilinear forms

$$\langle \mathbf{a}, \mathbf{U}\mathbf{b} \rangle = \langle \mathbf{a}^\mathsf{T}\mathbf{U}, \mathbf{b} \rangle = \mathbf{a}^\mathsf{T}\mathbf{U}\mathbf{b}.$$

In a bilinear tensor product, each component of the result takes this form. We can therefore think of the product as computing a series of distinct similarity measures between the two input vectors. With this in mind the obvious question is: what is the role of the matrix? The first thing to note is that inserting a matrix into the inner product allows the two vectors to be of different dimension. We also observe that a matrix-vector multiplication consists of taking the dot product of the vector with each row or column of the matrix. Given our current interpretation of the dot product as an un-normalised similarity measure, we can also interpret a matrix-vector multiplication as computing the similarity of the vector with each row or column of the matrix.

We can then think of the rows of the matrix $\mathbf{U}$ in the above as containing patterns to look for in the $\mathbf{b}$ vector and the columns to contain patterns to test for in the $\mathbf{a}$ vector. If we consider the vectors $\mathbf{a}$ and $\mathbf{b}$ to come from different feature spaces, the matrix $\mathbf{U}$ provides a conversion between them allowing us to directly compare the two. We can then interpret each coordinate of the result of the bilinear tensor product as being an independent similarity measure based on different interpretations of the underlying feature space. In this sense, where a matrix multiplication looks for patterns in a single input space, a bilinear product looks for *joint* patterns in the combined input spaces of $\mathbf{x}$ and $\mathbf{y}$.

Choosing a matrix

Following on from the above discussion we claim that for each coordinate of the output we are computing a *similarity vector* which we compare to the remaining input to generate a scalar value. If we consider all coordinates at once, we see that this amounts to having one input choose a matrix, which we then multiply by the remaining input vector. We have refrained from making these points in terms of the specific vectors referenced above to make the point that the operation is completely symmetrical. While it aids interpretation to think of one vector choosing a matrix for the other vector, we can always achieve the same intuition after switching the vectors, give or take some transposes.

This interpretation is very clear from the expression of the product in equation eqrefeqrefeq:bilinearkolda. Simply by inserting parentheses we observe that we are first generating a matrix in a way somehow dependent on the first input and multiplying the second input by that matrix. In this sense we allow one input to choose patterns to look for in the other input.

This intuition of choosing a matrix is suggested in [38] in the context language modelling with RNNs. It is suggested that allowing the current input character to choose the hidden-to-hidden weights matrix should confer benefits. This intuition (and the factorisation of the implicit tensor) was put to use earlier in the context of Conditional Restricted Boltzmann Machines [42] which actively seek to model the conditional dependencies between two types of input.

Although this provides a powerful insight into the bilinear product, it is worth reinforcing that the product is entirely symmetrical. We can not think purely about it as $\mathbf{x}$ choosing a matrix for $\mathbf{y}$ as the converse is equally true.

Tensor as an independent basis

Extending the above to try and capture the symmetricity of the operation, we introduce the notion that the coefficients of the tensor represent a basis in which to compare the two inputs, independent of both of them. This idea is mentioned in [43] which considered the problem of data that can be described by two independent factors. The tensor then contains a basis which characterises the interaction by which the factors in the input vectors combine to create an output observation.

This is a somewhat abstract interpretation – to attempt to create a more concrete intuition consider the case when both vectors have a single element set to 1 and the remainder 0. In this case the first tensor-vector product corresponds to taking a slice of the tensor, resulting in a matrix. The final matrix-vector product corresponds to picking a row or column of the matrix. Consequently the whole operation is precisely looking up a fibre of the tensor. To generalise from such a "one-hot" encoding of the inputs to vectors of real coefficients, we simply replace the idea of a *lookup* with that of a *linear combination*. The vectors then represent the coefficients of weighted sums; first over slices and then over rows or columns. The final product is then a distinct representation of both vectors in terms of the independent basis expressed by the tensor.

Operation on the outer product

n this section we describe the bilinear tensor product as operating on pairwise products of inputs. This interpretation is essential for understanding the expressive power of the operation as it gives rise to an obvious XOR-like behaviour. It also serves as a useful reminder that a bilinear form is linear in each input only when the other is held constant – when both are allowed to vary we can represent complex non-linear relations. A way to approach this interpretation arises from a method of implementing the bilinear product in terms of straightforward matrix operations.

To discuss this we introduce the notion of the *matricisation* of a tensor. Intuitively this operation is a way of *unfolding* or *flattening* a tensor into a matrix, preserving its elements. Specifically the mode-n matricisation of a tensor is defined as an operation which takes all mode-n fibres of the tensor and places them as columns to create a matrix. We denote an mode-n matricisation of a tensor $\boldsymbol{\mathcal{W}}$ as $\text{mat}_n(\boldsymbol{\mathcal{W}})$. While the operation is fairly straightforward, describing the exact permutation of the indices is awkard – for a robust treatment of the general case (and the source of the above definition) see [23].

Although this notion captures and generalises the vectorisation operator often encountered in linear algebra, we retain the classical vec operator for clarity. This flattens a matrix into a vector by stacking its columns. For some matrix $\mathbf{A}$ with n columns:

$$\text{vec}(\mathbf{A}) = \begin{bmatrix} \mathbf{A}_{\cdot 1} \\ \mathbf{A}_{\cdot 2} \\ \vdots \\ \mathbf{A}_{\cdot n} \end{bmatrix}.$$

For our purposes is is sufficient to note that a mode-2 matricisation of a three-way tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ must have shape $n_2 \times n_1 n_3$. This gives us the following lemma, which shifts this line of thinking slightly sideways.

Lemma 3.2.1 (Matricisation/vectorisation)**.** *The j-th row of the mode-2 matricisation of the three-way tensor $\boldsymbol{\mathcal{W}}$ is equivalent to the vectorisation of the slice formed by fixing the second index at* *j*:

$$\text{mat}_2(\boldsymbol{\mathcal{W}})_{j\cdot} = \text{vec}(\mathbf{W}_{\cdot j \cdot}).$$

Proof. By the above definition of the vectorisation operator, each index (i,k) in some matrix $\mathbf{U} \in \mathbb{R}^{n_1 \times n_3}$ maps to element $i + (k-1)n_1$ in $\text{vec}(\mathbf{U})$. By the definition of the mode-2 matricisation we would expect to find tensor element W_{ijk} at index $(j, i + (k-1)n_3)$. Hence if we fix j, we have precisely the vectorisation of the j-th slice of the tensor.

The indices into $\text{mat}_2(\boldsymbol{\mathcal{W}})$ can be though of as arising from the following construction process: first fix all indices to 1. Construct a column by sweeping the second index, j, through its full range. Then increment the first index i and repeat the procedure, placing the generated columns with index i. Only when i has swept through its full range increment the final index k and repeat the procedure. The generated columns should then be at positions $i + (k-1)n_1$. □

These flattenings are important as they allow us to implement many operations involving tensors in terms of a small number of larger matrix operations when compared to the naive approach.

Lemma 3.2.2 (Matricised product). *For a tensor* $\boldsymbol{\mathcal{W}}$ *and vectors* $\mathbf{x}, \mathbf{y}$ *as above, we can describe the product* $\mathbf{z} = \mathbf{x}^\mathsf{T}\boldsymbol{\mathcal{W}}\mathbf{y}$ *in terms of the mode-2 matricisation of* $\boldsymbol{\mathcal{W}}$ *as follows:*

$$\mathbf{z} = \text{mat}_2(\boldsymbol{\mathcal{W}})\text{vec}\left[\mathbf{y}\mathbf{x}^\mathsf{T}\right] \tag{3.4}$$

Proof. To prove this we can compare the expressions for a single element of the result. An element z_j from equation (3.4) is formed as the inner product of the j-th row of the flattened tensor and the vectorised outer product of the inputs. By lemma 3.2.1:

$$z_j = \sum_{s=1}^{n_1 n_3} (\text{mat}(\boldsymbol{\mathcal{W}})_{js} \left(\text{vec}\left[\mathbf{y}\mathbf{x}^\mathsf{T}\right]\right)_s$$

We replace the sum over s with a sum over two indices, i and k, and using them to appropriately re-index the flattenings as described in lemma 3.2.1 we derive

$$\begin{aligned} z_j &= \sum_{i=1}^{n_1}\sum_{k=1}^{n_3} W_{ijk} x_i y_k \\ &= \mathbf{x}^\mathsf{T}\mathbf{W}_{\cdot j \cdot}\mathbf{y} \end{aligned}$$

□

Therefore to understand the bilinear product it helps to understand the matrix $\mathbf{y}\mathbf{x}^\mathsf{T}$. Each element is of the following form:

$$(\mathbf{y}\mathbf{x}^\mathsf{T})_{ki} = x_i y_k,$$

it contains all possible products of pairs of elements, one from each vector. This captures some interesting interactions.

Theorem 1 (Bilinear exclusive-or). *Bilinear tensor products operate on the pair-wise exclusive-or of the signs of the inputs.*

Proof. Consider the sign of a scalar product $c = a \cdot b$. If one of the operands a or b is positive and the other negative, then the sign of c is negative. If both are positive or both are negative, then the result is positive. This captures precisely the "one or the other but not both" structure of the exclusive-or operation.

By lemma 3.2.2 a bilinear product can be viewed as a matrix operation on the flattened outer product of the two inputs. As each element in the outer product is the product of two scalars, the signs have an exclusive-or structure. □

Corollary 1.1 (Bilinear conjunctions). *If the inputs are binary, bilinear products operate on pair-wise conjunctions of the inputs.*

Proof. If $a, b \in \{0, 1\}$, then $a \cdot b = 1$ if and only if both a and b are 1. If either or both are 0 then their product must be zero. Following the same structure as the proof of theorem 1, for binary inputs we must have this conjunctive relationship. □

There are a number of remarks worth making about these results. Firstly they suggest considering the case where both inputs are the same: $\mathbf{z} = \mathbf{x}^\mathsf{T}\mathcal{W}\mathbf{x}$. Replacing bilinear forms with quadratic forms may invalidate some of the similarity based interpretations, but it might also provide an interesting class of very powerful feed-forward networks. As an example, note that a plain perceptron has been long known to be incapable of learning the exclusive-or mapping [31] – a neural network to solve the problem requires either a hidden layer or an additional input feature (specifically the conjunction of the inputs) [36]. By corollary 1.1 this quadratic tensor form would implicitly and naturally capture the additional conjunctive feature and be capable of solving the exclusive or problem without hidden layers or hand-engineered features.

Secondly, this captures the notion that the bilinear tensor product operates implicitly on higher level features, constructed by combining both inputs. This indicates that it is capable of capturing complex relationships between the two inputs.

3.3 Tensor Decompositions

The ultimate goal is to learn the tensor that parameterises a bilinear product. Unfortunately such a tensor is often very large; an $n \times n \times n$ would require $O(n^3)$ space to store explicitly. Further, it is highly likely that we will be looking to model simpler interactions than the full tensor might represent. This leads to the idea of a parameterised decomposition – an ideal solution would be a method of representing the tensor with some way of specifying the complexity of the bilinear relationship and hence making explicit the relationship between the number of parameters required and the expressive power of the model.

We will investigate two methods of tensor decomposition with a view towards their use as a building block for neural networks. The outcomes we wish to evaluate are: the savings in storage, how convenient it is to specify the number of parameters, convenience and efficiency of implementation (in terms of matrix operations) and whether the gradients suggest any benefits or hindrances when learning by gradient descent. There is a significant amount of research into tensor decompositions, [23] provides a thorough review. Here we consider some of the most general decompositions and families of decompositions without going into detail outside of where it is relevant to the above concerns.

3.3.1 CANDECOMP/PARAFAC

This method of decomposing a tensor dates as far back as 1927 [20, 19] and was rediscovered repeatedly across a range of communities over the next 50 years [23]. First referred to as 'polyadic form of a tensor' we refer to it as the CP-decomposition after two prominent publications in 1970 referring to the technique as 'canonical decomposition' (CANDECOMP) [4] or 'parallel factors' (PARAFAC) [15]. It is a fairly straightforward extension of a matrix rank decomposition to the more general case, representing a tensor as a sum of rank one tensors. This section contains a formal description of the three-way case as well as a note on computing bilinear products with the decomposed tensor. Some interesting results on the uniqueness of the decompositions can be found in [23], but they are not relevant to the discussion here.

Description

Given a three-way tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ we wish to approximate it as

$$\boldsymbol{\mathcal{X}} \approx \sum_{r=1}^{R} \mathbf{a}_r \otimes \mathbf{b}_r \otimes \mathbf{c}_r$$

where R is the rank of the decomposition and $\cdot \otimes \cdot$ denotes the tensor product. This product expands the dimensions; for the first two vectors it is the outer product $\mathbf{ab}^\mathsf{T}$ which generates a matrix consisting of the product of each pair of elements in $\mathbf{a}$ and $\mathbf{b}$. With three such products we proceed analogously, except now our structure must contain the product of each possible triple of elements. Hence we need three indices to address each entry, so it best described as a three-way tensor. Each individual element has the form

$$X_{ijk} = \sum_{r=1}^{R} a_{ri} b_{rj} c_{rk}.$$

It is convenient to stack these factors into matrices – we differ slightly from [23] by defining the *factor matrices* of the form $\mathbf{A} = [\mathbf{a}_1 \; \mathbf{a}_2 \; \cdots \; \mathbf{a}_R]^\mathsf{T} \in \mathbb{R}^{R \times n_1}$ (and equivalently for $\mathbf{B}$ and $\mathbf{C}$) such that the *rows* of the matrix correspond to the R vectors. We denote a tensor decomposed in this format $\boldsymbol{\mathcal{X}} = [\mathbf{A}, \mathbf{B}, \mathbf{C}]_{CP}$.

Bilinear Product

We wish to compute a product of the form $\mathbf{z} = \mathbf{x}^\mathsf{T} \boldsymbol{\mathcal{W}} \mathbf{y}$ where $\boldsymbol{\mathcal{W}} = [\mathbf{A}, \mathbf{B}, \mathbf{C}]_{CP}$ is represented as a CP-decomposition. Conveniently, this can be done with simple matrix products.

Proposition 3.3.1.

$$\begin{aligned} \mathbf{z} &= \mathbf{x}^\mathsf{T} \boldsymbol{\mathcal{W}} \mathbf{y} \\ &= \mathbf{B}^\mathsf{T} (\mathbf{Ax} \odot \mathbf{Cy}) \end{aligned}$$

when $\boldsymbol{\mathcal{W}} = [\mathbf{A}, \mathbf{B}, \mathbf{C}]_{CP}$ *and* $\odot$ *represents the Hadamard (elementwise) product.*

Proof. This follows from straightforward rearranging. Firstly

$$\begin{aligned} z_j &= \sum_i^{n_1} \sum_k^{n_3} W_{ijk} x_i y_k \\ &= \sum_i^{n_1} \sum_k^{n_3} \sum_r^{R} A_{ri} B_{rj} C_{rk} x_i y_k \\ &= \sum_r^{R} B_{rj} \left(\sum_i^{n_1} A_{ri} x_i \cdot \sum_k^{n_3} C_{rk} y_k \right). \end{aligned} \tag{3.5}$$

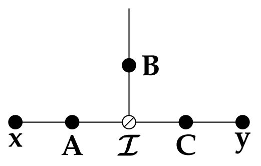


Figure 3.3: One way of representing a bilinear product in the CP-decomposition.

This implies we can compute the quantity inside the brackets for all r at once simply by $\mathbf{Ax} \odot \mathbf{Cy}$ which results in a length R vector. Equation (3.5) then notes that for each output j we take the j-th column of the $R \times n_2$ factor matrix $\mathbf{B}$ and perform a dot product with our earlier result. A series of dot products can be easily expressed with matrix multiplication, noting that we have to transpose $\mathbf{B}$ to keep it on the left but still sum over each column. Hence:

$$\mathbf{z} = \mathbf{B}^\mathsf{T}(\mathbf{Ax} \odot \mathbf{Cy}).$$

□

We can express the bilinear product above as a tensor network diagram, although we have to insert an auxiliary $R \times R \times R$ tensor, denoted $\mathcal{I}_R$ which is defined as

$$I_{ijk} = \begin{cases} 1 & \text{if } i = j = k \\ 0 & \text{otherwise} \end{cases}$$

in a manner analogous to the identity matrix. Taking a bilinear product with this tensor simply represents elementwise multiplication of the two vectors. In the diagram this is represented with the symbol $\oslash$ to emphasise the diagonality (and to emphasise that it it is different as we do not need to store its parameters). This is presented in figure 3.3, which can be compared to figure 3.2d.

Gradients

As the final aim is to attempt to learn the coefficients of the decomposition using gradient descent, it makes sense to check that the gradients of the parameters are well-behaved with respect to the output of the bilinear product. We break the gradient into three parts, one for each of the parameter matrices. do this properly

Let $\mathbf{z}$ be defined as above. The gradient of $\mathbf{z}$ with respect to $\mathbf{A}$ has entries of the form:

$$\begin{aligned} \frac{\partial z_j}{\partial A_{lm}} &= x_m \cdot \sum_k^{n_3} C_{lk} y_k \\ &= x_m \cdot \mathbf{C}_{l\cdot}^\mathsf{T} \mathbf{y}. \end{aligned}$$

The gradients with respect to $\mathbf{C}$ have the same form:

$$\begin{aligned} \frac{\partial z_j}{\partial C_{lm}} &= y_m \cdot \sum_i^{n_1} A_{li} x_i \\ &= y_m \cdot \mathbf{A}_{l\cdot}^\mathsf{T} \mathbf{x}. \end{aligned}$$

The gradient of the final parameter matrix $\mathbf{B}$ has entries

$$\frac{\partial z_j}{\partial B_{lm}} = \mathbb{I}_{m=j} \cdot (\mathbf{Ax} \odot \mathbf{Cy})$$

where $\mathbb{I}_{m=j}$ denotes the indicator function that is one if $i = j$ and zero otherwise. This has the curious effect that the gradient of the product with respect to **B** is the same for all columns of **B**. While this seems like it may hinder learning, in practice there will be additional terms due to the loss function which ameliorate this.

Storage Requirements

The number of stored coefficients for a rank R CP-decomposed tensor of size $I \times J \times K$ will be $RI + RJ + JK$. Explicitly storing the tensor would require IJK numbers to be stored. To illustrate the significance of this, consider the case when the tensor being decomposed has all dimensions and rank of equal size. Denoting the size as I, then the explicit tensor will have an I^3 storage requirement while the decomposed tensor needs only $3I^2$.

3.3.2 Tensor Train

The *tensor-train* decomposition has a much shorter history – it was first proposed in 2011 as a simpler way of representing a slightly earlier tree based form derived from a generalisation of the singular value decomposition [34]. It is proposed as an alternative to the CP-decomposition which purportedly provides benefits in terms of numerical stability. It has been used to learn compressed weight matrices in large neural networks [32] suggesting it is a prime candidate to learn with gradient descent.

Description

The tensor-train decomposition (TT-decomposition) of a general tensor $\boldsymbol{\mathcal{X}}$ with d indices is the tensor $\boldsymbol{\mathcal{Y}} \approx \boldsymbol{\mathcal{X}}$ with elements expressed as a product of slices of three-way tensors (so a product of matrices):

$$Y_{i_1 i_2 \dots i_d} = \mathbf{G}[1]_{\cdot i_1 \cdot}\mathbf{G}[2]_{\cdot i_2 \cdot} \cdots \mathbf{G}[d]_{\cdot i_d \cdot}.$$

If dimension j is of size n_j, then $\boldsymbol{\mathcal{G}}[j]$ is size $r_{j-1} \times n_j \times r_j$ so that each slice $\mathbf{G}[j]_{\cdot i \cdot}$ is size $r_{j-1} \times r_j$. The collection of r_i is the 'tt-rank' of the decomposition and controls the number of parameters. In order to make the result of the chain of matrix products a scalar, we have to ensure $r_0 = r_d = 1$ [34].

The three-way case presents no obvious simplifications from the general case but we present it here to ensure consistent notation through the remainder of this section. The TT-decomposition of a three-way tensor of size $I \times J \times K$ has three 'cores' with shapes $1 \times I \times r_1$, $r_1 \times J \times r_2$ and $r_2 \times K \times 1$. For convenience, we can treat these as a matrix, a three-way tensor and a matrix by ignoring dimensions of size one. This leads to the following expression of equation 3.3.2 where $\boldsymbol{\mathcal{W}}$ is the three way tensor in its decomposed form:

$$W_{ijk} = \mathbf{A}_{i\cdot}\mathbf{B}_{\cdot j \cdot}\mathbf{C}_{\cdot k}.$$

In a manner consistent with the CP-decomposition we denote such a decomposed tensor $\boldsymbol{\mathcal{W}} = [\mathbf{A}, \boldsymbol{\mathcal{B}}, \mathbf{C}]_{TT}$. It is important to note that the shapes are less consistent: **A** is an $I \times r_1$ matrix, $\boldsymbol{\mathcal{B}}$ a $r_1 \times J \times r_1$ tensor and **C** an $r_2 \times K$ matrix.

Bilinear Product

Computing a bilinear product between two vectors and a tensor in the TT-decomposition does not have quite such an efficient form as the CP-decomposition. Primarily this is due to the presence of the three-way tensor $\boldsymbol{\mathcal{B}}$, which means we are still eventually computing a full tensor product, albeit involving a smaller tensor. We denote the product as

$$\mathbf{z} = \mathbf{x}^\mathsf{T}\mathbf{A}\boldsymbol{\mathcal{B}}\mathbf{C}\mathbf{y}.$$

For a specific index this has the form

$$\begin{aligned} z_j &= \mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{B}_{\cdot j \cdot}\mathbf{C}\mathbf{y} \\ &= \sum_{i=1}^{I}\sum_{k=1}^{K}\sum_{m=1}^{r_1}\sum_{n=1}^{r_2} A_{im}B_{mjn}C_{nk}x_i y_k. \end{aligned}$$

This provides a clear intuition about what the TT-decomposition is doing in the three-way case. Matrices **A** and **C** project the inputs into a new (potentially smaller) space where the bilinear product is carried out with $\boldsymbol{\mathcal{B}}$ ensuring the result has been pushed back to the appropriate size.

Gradients

In the TT-decomposition the gradients are very similar. The gradient of **z** with respect to **A** has entries of the form:

$$\frac{\partial z_j}{\partial A_{lm}} = x_l \cdot \left(\mathbf{B}^\mathsf{T}_{mj\cdot}\mathbf{C}\mathbf{y}\right).$$

The components of the gradient with respect to **C** have a similar form:

$$\frac{\partial z_j}{\partial C_{ml}} = y_l \cdot \left(\mathbf{B}^\mathsf{T}_{\cdot jm}\mathbf{A}\mathbf{x}\right).$$

While the gradient of the elements of $\boldsymbol{\mathcal{B}}$ behave surprisingly similarly to the CP-decomposition:

$$\frac{\partial z_j}{\partial B_{lmn}} = \mathbb{I}_{m=j} \cdot \left(\mathbf{A}\mathbf{x} \odot \mathbf{C}\mathbf{y}\right).$$

3.3.3 Comparison

In this section we first prove a result on the conditions required for the decompositions to be equivalent. This is followed by a brief discussion of the theoretical differences and similarities between the two decompositions and whether we can draw any conclusions about their suitability for the task at hand.

It is also worth noting briefly that the gradients are very nearly equivalent. This implies that attempting to learn the parameters directly using gradient descent should work for both or neither, since it seems reasonable to expect similar dynamics.

Equivalence

One condition for the tensor-train to be equivalent to the CP-decomposition is if the central tensor in the TT-decomposition has diagonal, square slices. Intuitively this reduces the tensor product to a matrix product. To the best of our knowledge this result has not been presented elsewhere. It can be found in the appendix, proposition **??**.

Space Requirements

The CP-decomposition is appealing because it has a single hyper-parameter r. Further, the storage of a CP-decomposed tensor is linear in r. By comparison the TT-decomposition has two parameters and the total storage requirement grows linearly in their product. This makes choosing the parameter potentially easier with the CP-decomposition. If we take the easy option and set both ranks of the TT-decomposition to the same value, then the number of elements is proportional to the square of that value. As the rank has to be an integer, this potentially gives us fewer feasible options.

3.4 Learning decompositions by gradient descent

Multiplicative dynamics = instability?

Chapter 4

Proposed Architectures

4.1 Incorporating tensors for expressivity

4.2 Gates and Long Time Dependencies

4.3 Proposed RNNs

Chapter 5

Evaluation of architectures

5.1 Synthetic Tasks

Pathological, exercise specific features of the architecture.

5.1.1 Addition

5.1.2 Variable Binding

5.1.3 MNIST

is really dumb

5.2 Real-world Data

Mostly testing rank as regulariser

5.2.1 Polyphonic Music

5.2.2 PTB

5.2.3 War and Peace

Chapter 6

Conclusions

The conclusions are presented in this Chapter.

Bibliography

[1] ARJOVSKY, M., SHAH, A., AND BENGIO, Y. Unitary Evolution Recurrent Neural Networks. *arXiv* (Nov. 2015), 1–11. arXiv: `1511.06464`.

[2] BENGIO, Y. Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks* 5, 2 (1994), 157–166.

[3] BOULANGER-LEWANDOWSKI, N., VINCENT, P., AND BENGIO, Y. Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription. *Proceedings of the 29th International Conference on Machine Learning (ICML-12)* Cd (June 2012), 1159–1166. arXiv: `1206.6392`.

[4] 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, 3 (Sept. 1970), 283–319.

[5] CHO, K., ET AL. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (June 2014), 1724–1734. arXiv: `1406.1078`.

[6] CHOI, K., ET AL. Convolutional Recurrent Neural Networks for Music Classification. *arXiv preprint* (Sept. 2016). arXiv: `1609.04243`.

[7] CHUNG, J., AHN, S., AND BENGIO, Y. Hierarchical Multiscale Recurrent Neural Networks. *arXiv preprint* (2016). arXiv: `1609.01704`.

[8] CHUNG, J., ET AL. Gated feedback recurrent neural networks. *Proceedings of the 32nd International Conference on Machine Learning* 37 (2015), 2067–2075. arXiv: `1502.02367`.

[9] CICHOCKI, A., ET AL. Low-Rank Tensor Networks for Dimensionality Reduction and Large-Scale Optimization Problems: Perspectives and Challenges PART 1. *arXiv preprint* (2016), 100. arXiv: `1609.00893`.

[10] DUVENAUD, D., ET AL. Avoiding pathologies in very deep networks. *Mlg.Eng.Cam.Ac.Uk* (Feb. 2014), 9. arXiv: `1402.5836`.

[11] ELMAN, J. l. Finding Structure in Time. *COGNITIVE SCIENCE* 14 (1990), 179–211.

[12] GRAVES, A. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* (2013), 1–43. arXiv: `arXiv:1308.0850v5`.

[13] GRAVES, A., ET AL. Connectionist Temporal Classification : Labelling Unsegmented Sequence Data with Recurrent Neural Networks. *Proceedings of the 23rd international conference on Machine Learning* (2006), 369–376.

[14] GREGOR, K., ET AL. DRAW: A Recurrent Neural Network For Image Generation. *Icml-2015* (Feb. 2015), 1462–1471. arXiv: `1502.04623`.

[15] HARSHMAN, R. A. Foundations of the PARAFAC procedure: Models and conditions for an explanatory multimodal factor analysis. *UCLA Working Papers in Phonetics* 16, 10 (1970), 1–84.

[16] HE, K., ET AL. Deep Residual Learning for Image Recognition. *arXiv* (Dec. 2015). arXiv: 1512.03385.

[17] HENAFF, M., SZLAM, A., AND LECUN, Y. Orthogonal RNNs and Long-Memory Tasks. *arxiv* (2016). arXiv: 1602.06662.

[18] HIHI, S. E., AND BENGIO, Y. Hierarchical Recurrent Neural Networks for Long-Term Dependencies. *Nips* (1995), 493–499.

[19] HITCHCOCK, F. L. Multiple Invariants and Generalized Rank of a P-Way Matrix or Tensor. *Journal of Mathematics and Physics* 7, 1-4 (Apr. 1928), 39–79.

[20] HITCHCOCK, F. L. The Expression of a Tensor or a Polyadic as a Sum of Products. *Journal of Mathematics and Physics* 6, 1-4 (Apr. 1927), 164–189.

[21] HOCHREITER, S., AND SCHMIDHUBER, J. Long Short-Term Memory. *Neural Computation* 9, 8 (Nov. 1997), 1735–1780. arXiv: 1206.2944.

[22] JOZEFOWICZ, R., ZAREMBA, W., AND SUTSKEVER, I. An Empirical Exploration of Recurrent Network Architectures. In: *ICML*. 2015.

[23] KOLDA, T. G., AND BADER, B. W. Tensor Decompositions and Applications. *SIAM Review* 51, 3 (Aug. 2009), 455–500.

[24] KOUTNIK, J., ET AL. A Clockwork RNN. *Proceedings of The 31st International Conference on Machine Learning* 32 (2014), 1863–1871. arXiv: arXiv:1402.3511v1.

[25] KRUEGER, D., AND MEMISEVIC, R. Regularizing RNNs by Stabilizing Activations. *International Conference On Learning Representations* (Nov. 2016), 1–8. arXiv: 1511.08400.

[26] LE, Q. V., JAITLY, N., AND HINTON, G. E. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. *arXiv preprint arXiv:1504.00941* (2015), 1–9. arXiv: arXiv:1504.00941v1.

[27] LORENZ, E. N. Deterministic Nonperiodic Flow. *Journal of the Atmospheric Sciences* 20, 2 (Mar. 1963), 130–141.

[28] MARTENS, J., AND SUTSKEVER, I. Learning recurrent neural networks with Hessian-free optimization. *Proceedings of the 28th International Conference on Machine Learning, ICML 2011* (2011), 1033–1040.

[29] MIKOLOV, T. Statistical Language Models Based on Neural Networks. PhD thesis. 2012, 1–129. arXiv: 1312.3005.

[30] MIKOLOV, T., ET AL. Learning Longer Memory in Recurrent Neural Networks. *Iclr* (Dec. 2015), 1–9. arXiv: arXiv:1412.7753v1.

[31] MINSKY, M., AND PAPERT, S. *Perceptrons.* M.I.T. Press, 1969.

[32] NOVIKOV, A., ET AL. Tensorizing Neural Networks. *Nips* (2015), 1–9. arXiv: arXiv:1509.06569v1.

[33] ORS, R. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics* 349 (June 2014), 117–158. arXiv: 1306.2164.

[34] OSEDELETS, I. V. Tensor Train Decomposition. *SIAM J Sci. Comput.* 33, 5 (2011), 2295–2317.

[35] PASCANU, R., MIKOLOV, T., AND BENGIO, Y. On the difficulty of training recurrent neural networks. *Proceedings of The 30th International Conference on Machine Learning* 2 (2012), 1310–1318. arXiv: arXiv:1211.5063v2.

[36] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533–536. arXiv: `arXiv:1011.1669v3`.

[37] Singhal, A. Modern Information Retrieval: A Brief Overview. *Bulletin of the Ieee Computer Society Technical Committee on Data Engineering* 24, 4 (2001), 1–9.

[38] Sutskever, I. Training Recurrent neural Networks. PhD thesis. 2013, 101. arXiv: `1456339 [arXiv:submit]`.

[39] Sutskever, I., et al. On the importance of initialization and momentum in deep learning. *Jmlr* 28, 2010 (2013), 1139–1147.

[40] Szegedy, C., Ioffe, S., and Vanhoucke, V. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *Arxiv* (2016), 12. arXiv: `1602.07261`.

[41] Tan, P.-N., Steinbach, M., and Kumar, V. *Introduction to data mining*. Pearson Addison Wesley, 2006, 769.

[42] Taylor, G. W., and Hinton, G. E. Factored conditional restricted Boltzmann Machines for modeling motion style. In: *Proceedings of the 26th International Conference on Machine Learning (ICML 09)*. 2009, 1025–1032.

[43] Tenenbaum, J. B., and Freeman, W. T. Separating style and content with bilinear models. *Neural computation* 12, 6 (2000), 1247–1283.

[44] Vinyals, O., et al. Show and Tell: Lessons learned from the 2015 MSCOCO Image Captioning Challenge. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 99, PP (2016), 1–1. arXiv: `1609.06647`.

[45] Werbos, P. J. Backpropagation Through Time: What It Does and How to Do It. *Proceedings of the IEEE* 78, 10 (1990), 1550–1560.

[46] Xu, K., et al. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *arXiv preprint* (2015). arXiv: `arXiv:1211.5063v2`.