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

Feed-forward neural networks are a class of parameterised function approximators which consist of artificial neurons arranged in layers. They arise from generalisations of the perceptron [55], indeed simple feed-forward networks are often termed 'multi-layer perceptrons' (MLPs). MLPs solve the key limitations of the early perceptron [48], but it was not until 1986 when a reliable and general purpose procedure (*back-propagation*) was proposed [56] to train them.

Conceptually it is possible to consider MLPs as a directed acyclic graph, where each node consists of a 'neuron' or 'unit' with the direction of the edges indicating the way information flows from the input of the function being modelled to the output. For an MLP this graph is divided into layers. The first layer represents the input (with one unit per coordinate) while subsequent layers represent individual neurons. In a standard fully-connected network, every node is connected to all nodes in the subsequent layer. When an input is presented to the network, each neuron computes an activation by a weighted sum over its inputs followed by the application of some non-linear function. Specifically a unit that receives inputs $(x_1, x_2, \ldots, x_n)$ will compute its activation h as:

$$h = f\left(\sum_{i=1}^{n} x_i w_i\right) \tag{2.1}$$

where $(w_1, w_2, \ldots, w_n) + b$ are the weights the neuron gives its inputs and b is a bias, commonly added to enhance the expressive power. $f : \mathbb{R} \to \mathbb{R}$ is some nonlinear function, common examples would be a threshold, the logistic sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ or a linear rectifier $\rho(x) = \max(0, x)$ [49].

While it is convenient to consider the network in this manner, it is computationally and notationally more efficient to consider to express many neurons at once, using vectors and matrices. If we consider the input to a network as a vector of features $\mathbf{x}$, we can express eq. (2.1) with a matrix multiplication as

$$h = f\left(\mathbf{w}^\mathsf{T}\mathbf{x} + b\right)$$

where $\mathbf{w}$ is the weights from eq. (2.1) as a vector. We can generalise this to compute an entire layer of activations at once. Let $\mathbf{W}$ be a matrix whose rows contain the weight vectors for each neuron in a given layer and $\mathbf{b}$ a vector containing all of their biases. We can then compute

their activations at once with

$$\mathbf{h} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$$

provided we ensure f is applied elementwise to the result.

This gives us a single layer, a classic MLP will perform this process twice, although the last layer is often linear:

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)} f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}.$$

This can be though of as proceeding in stages – first compute $\mathbf{h}$ as before and then using another set of weights compute a new transformation of it to get the final output.

Although it can not increase the class of functions a neural network can approximate, it is often beneficial in practice to increase the depth as this can allow to express more complex relationships with only a moderate increase in parameters to be learned. [68] This is done by recursively applying the above transform with a fresh set of weights and biases. Omitting the biases for clarity this gives us the following for a network with $l-1$ hidden layers:

$$\hat{\mathbf{y}} = \mathbf{W}^{(l)} f(\mathbf{W}^{(l-1)} f(\dots f(\mathbf{W}^{(1)}\mathbf{x}))).$$

Naively adding depth in this fashion can cause the network to become very challenging to train. [15, 57] Several methods have been proposed which successfully alleviate this, such as layer-wise pre-training [26, 72] and more recently residual connections which change the structure of the graph to allow better flow of information. [22]

Training

These networks are typically trained in a supervised fashion. Denote the parameters of the network $\theta = \{\mathbf{W}^{(l)}, \dots, \mathbf{W}^{(1)}, \mathbf{b}^{(l)}, \dots, \mathbf{b}^{(l)}\}$ and the output of the network for a given input $\mathbf{x}$ and set of parameters as $\hat{\mathbf{y}} = \mathcal{F}(\mathbf{x};\ \theta)$. Suppose we have a set of m data items $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^m$ where each item is an input-output pair. Let $\mathcal{X}$ be the set of possible $\mathbf{x}$ values and $\mathcal{Y}$ the set of possible $\mathbf{y}$ values. We wish to learn a mapping which will produce each $\mathbf{y}_i$ from the given $\mathbf{x}_i$. To do this we define a loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ which maps pairs of valid outputs to a real number, such that $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = 0$ implies $\mathbf{y} = \hat{\mathbf{y}}$. Typical loss functions are the squared error

$$\mathcal{L}_{SE}(\mathbf{y}, \hat{\mathbf{y}}) = ||\mathbf{y} - \hat{\mathbf{y}}||_2^2 = \sum_d (y_d - \hat{y}_d)^2$$

or the cross entropy

$$\mathcal{L}_{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_d y_d \log(\hat{y}_d).$$

The cross entropy is typically preferred when the targets can be interpreted as probabilities.[1] This requires converting the output of the network to a probability, typically done with either the sigmoid function mentioned above or the softmax $s_i(\mathbf{x}) = e^{x_i} / \sum_j e^{x_j}$.

The best set of parameters for the network, θ^*, is then the set which minimises the loss across the data items. Training the network amounts to solving the following optimisation problem:

$$\theta^* = \arg\min_\theta \sum_{i=1}^m \mathcal{L}(\mathbf{y_i}, \mathcal{F}(\mathbf{x}_i;\ \theta)).$$

[1] For example, discrete class labels – we can convert them to a one-of-n (often termed "one-hot") taking discrete labels between 1 and n to n dimensional vectors with 0 in all positions bar one, which has its element set to 1. We can then view these vectors as a (very certain) probability distribution over classes. In this case the cross entropy corresponds exactly to the negative log-likelihood of the correct class.

For neural networks this is typically done using variations on gradient descent. In its simplest form, each weight is updated at each iteration as follows:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \cdot \nabla_{\mathbf{W}} \mathcal{L}$$

where η is a hyper parameter adjusting the step size (called the *learning rate*) and $\nabla_{\mathbf{W}} \mathcal{L}$ is the gradient of the loss with respect to the $\mathbf{W}$. Typically the gradient is evaluated over a small subset or "mini-batch" of the data per iteration which is akin to using a noisy estimate of the true gradient. The gradients of deep networks can be calculated using *back-propagation*, which amounts to the chain rule of calculus and the observation that neural networks consist of many composed functions. [56] In practice this is assisted greatly by software packages which can determine the derivatives of a given function using automatic or symbolic differentiation such as Tensorflow [1] or Theano . [70]

Mention improved SGDs? Adagrad, Adam, RMSProp?

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 [46] (including machine translation [6, 76]), speech recognition [18], polyphonic music modelling [4], music classification [7], image generation [20], automatic captioning [74, 78] and more.

Classical 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 [14] computes its hidden states with the following recurrence:

$$\mathbf{h}_t = f(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}) \tag{2.2}$$

where $f(\cdot)$ is some elementwise non-linearity, often the hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. Final outputs are then computed from these states in the same fashion as the feed-forward networks above. This remains the same for the extended architectures considered below, the key difference is the manner in which the states are produced.

Equation (2.2) 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" [75] 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

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

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

$$\begin{aligned}\nabla_{\mathbf{h}_i}\mathcal{L}_j &= (\nabla_{\mathbf{h}_j})^\mathsf{T}\mathcal{L}_j \left(\prod_{k=i+1}^{j} \frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}}\right) \\ &= (\nabla_{\mathbf{h}_j}\mathcal{L}_j)^\mathsf{T} \left(\prod_{k=i+1}^{j} \nabla_{\mathbf{z}_k} f \cdot \mathbf{W}\right). \end{aligned} \tag{2.3}$$

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 (2.3) 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 (2.3) 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 [3], for a thorough treatment including necessary conditions for vanishing and the complementary exploding problem, see [53].

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 [40]. In RNNs this manifests as near-discontinuity of the loss surface [53] 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 [53] or using a regulariser to encourage gradual changes in hidden state [37].

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: [29, 16]

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

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. [6] 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 [32, 29]. 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 [22, 12, 65]. 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 [3]. 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 [24]. This idea has been revisited recently in the form of the Clockwork RNN [36] 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 [8] and Gated Feedback RNNs [9] 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 [2] 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 [39], which simply initialise the recurrent weights matrix to the identity (presaged by the nearly diagonal initialisation in [47]), perform remarkably well on pathological tasks. The importance of the initialisation of the recurrent weights matrix was further emphasised in [23] 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 [42, 4] 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 [64].

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 [37]. This encourages the network to learn smooth transitions and may help it "bootstrap" itself past the vanishing gradients.

2.2.2 Memory

Another line of research which attempts to enhance the capabilities of RNNs focuses on augmenting them with additional memory structures. There are two key motivations for this kind of work. One aim is to decouple the memory from the network so that it can store and interact with arbitrary quantities of information. The second is related to vanishing gradients in the hope that a novel method of storing or addressing information will be more robust to long time delays.

A notable effort to decouple RNNs from their memory comes in Neural Turing Machine proposed in [17]. This uses a neural network (which may or may not be recurrent itself) to attend to a large matrix of memory. Addressing for both reading and writing is done via a address mechanism mostly based on the softmax. It was evaluated on a number of synthetic tasks which mostly involved accumulation such as remembering a sequence of arbitrary patterns and outputting them after a number of time-steps in a particular order.

Subsequently, Grefenstette et al. propose a set of neural Deques, Queues and Stacks which attempt to address the limitations of the fixed size memory of the Neural Turing Machine [19]. A similar approach is also found in Joulin and Mikolov's Stack Augmented RNN [31]. These allow for unbounded memory with efficient access, at the cost of the random access nature of the memory. Primarily these are proposed as augmentations to an RNN, with the idea being that a standard RNN could operate as a small working memory while the more complex data structure (with more awkward access semantics) can be used for long-term, more exact storage. This is an exciting direction as it enables something akin to a learnable Von Neumann architecture [17].

Other such approaches have also arisen such as the Neural Random-Access Machine, which is similar to a Neural Turing Machine except that it solves a number of the inefficiencies with a clever addressing mechanism [38]. Other related works include Pointer Networks [73] and Neural GPUs [33]. These tend to involve adding fairly complex structures on top of an LSTM. While they are highly successful on synthetic tasks when compared to an unaugmented LSTM, very few of the tasks seem like the kind of task that an RNN should inherently be incapable of solving. Nearly all of the tasks addressed require a fixed size memory, and solutions can be imagined composed of carefully writing to it and reading from it. Hence it seems as though it is worth revisiting the basic architecture, rather than searching for distinct modules to add on.

An approach in this direction is Danihelka et al.'s Associative LSTM [11]. This work incorporates an associative memory model into the LSTM by adjusting several of the operations to reflect something close to a Holographic Reduced Representation [54]. This is a very encouraging angle – to re-think the way in which the RNN uses its memory at a basic level seems like the most efficient and sensible way to address the issues. The results were very promising on some interesting synthetic tasks, focusing on the ability of the network to do key-value retrieval which should be a strength of the Holographic Reduced Representation. Unfortunately simply adjusting the network to sue such a reduced representation was insufficient for good performance and a number of complicating factors had to be incorporated, including keeping many redundant copies of memory which diminishes somewhat the elegance of the solution.

Many of these ideas seem to be founded on an intuition which has not been clearly formalised. In general, the task of an RNN is to learn a mapping from one sequence to another sequence one step at a time using an intermediate memory (the hidden states). Intuitively a solution must consist of writing to and reading from memory, both conditioned on input. In an RNN, reading from the memory is trivially attended to as typically the entire memory is used to produce the output at each time step. However, we would suggest identifying two separate types of writing to memory: accumulating and forgetting. Many of the architectures discussed below (and the GRU and LSTM above) have separate mechanisms for these two tasks. We consider an accumulation to be an updating of the state, typically additively, to incorporate new information. Forgetting is simply wiping clean a section of the memory – this tends to be achieved multiplicatively. Different tasks will require a different balance of the two. Instinctively, a classification tasks in which the network is only evaluated on its final output would favour more accumulative solutions. Conversely, tasks which may not even have a fixed length but rather require constant prediction should be better suited to networks with an ability to forget information that is no longer relevant.

2.2.3 Tensors in Neural Networks

The final set of related works provide the key motivation for the next section. In general, many of the above architectures contain multiplicative gates. Multiplicative gating structures are not novel, introduced by Hinton as early as 1981. [25] Sigaud et al. separate modern uses into two categories based on the intuition behind the application. [58] The first class seeks to use the gate in a manner akin to a transistor in a digital circuit, seeking to switch the flow of information between computational units on or off. This captures the way such connections are used in LSTMs as well as their feed-forward cousins Highway Networks. [61] The second class of gated networks aims to implement a multiplicative connection between inputs. The latter class strictly generalises the former. Intuitively, the former class captures those models where two inputs to a computational unit are multiplied before they are processed, while the second admits some transformations before the multiplication is realised. Two excellent

reviews can be found in [43, 58] and rather than repeat in detail the history we attempt to bring out the most salient examples which make clear the intuitions behind incorporating multiplicative interactions. Finally, we mention a few particularly very recent applications that have achieved significant successes after publication of the aforementioned reviews.

The earliest examples of multiplicative interactions in the context of the current Deep Learning movement is in the Gated Boltzmann machine [44] which generalised Restricted Boltzmann Machines [60] (a class of probabilistic graphical models very closely related to neural networks) to model conditional relationships between pairs of images. These parameterise a multiplicative connection with a three-way tensor $\mathcal{W}$. The likelihood of a hidden state being 1 given a pair of inputs $\mathbf{x}$ and $\mathbf{y}$ is defined as

$$p(h_k|\mathbf{x}, \mathbf{y}) = \sigma\left(\sum_{ij} W_{ijk} x_i y_j\right) \tag{2.4}$$

where $\sigma(\cdot)$ is the sigmoid discussed earlier. [44] This architecture very naturally extends the operation of a feed-forward network to deal with two inputs (compare equations (2.1) and (2.4)), but the size of the tensor $\mathcal{W}$ is a serious limitation. This work outlines three key points: multiplicative interactions are somehow a natural way of handling multiple distinct inputs, multiplicative interactions are properly parameterised by tensors and tensors can be prohibitively large. This last point is addressed in later works in the same field by factorising the tensor into three matrices. [67, 45].

The Multiplicative RNN [63, 62] picks up both the notion that multiplicative interactions are sensible ways to model interactions between two inputs and the factorisation from the above. Applied in a language modelling context, the reasoning is that it makes sense to allow the current input symbol to affect the recurrent transition matrix. These MRNNs achieved significant results, especially when combined with Hessian Free [42], a method of approximating second order optimisation.

Very recent methods have revisited a number of these ideas, although often with little to no mention of the underlying tensorial structure. Of note is the Multiplicative Integration RNN which proposes adjusting the building block of eq. (2.2) to use element-wise multiplication instead of addition. [77] This corresponds to factorising the tensor into two matrices and has an extreme gating effect, especially during back-propagation. In order to successfully learn, it was found necessary to add additional bias terms to the point where the final model required four weight matrices and five bias vectors per unit. In some experiments they incorporate this approach into an LSTM, leading to a model which would have nearly 16 times the parameters of the simplest RNN proposed by eq. (2.2). We suggest that paying more attention to the tensorial structure of the desired interaction would lead to more thoroughly grounded models, and propose one such model in chapter 4.

Chapter 3

Three-way Tensors and Bilinear Products

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.

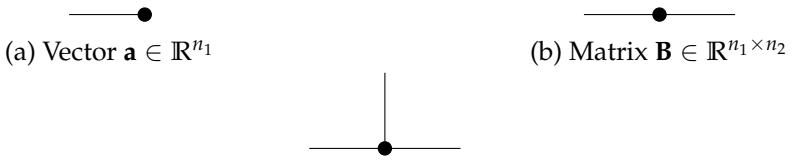
3.1 Definitions

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 [35] deviating only where it would become unwieldy. We denote tensors with three or more dimensions with single calligraphic boldface letters such as $\mathcal{W}$. 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

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.

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 $\mathbf{x} \in \mathbb{R}^n$, 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 [51]. This name arises because we form the output by joining a shared dimension (by elementwise multiplication) and contracting it with a sum.

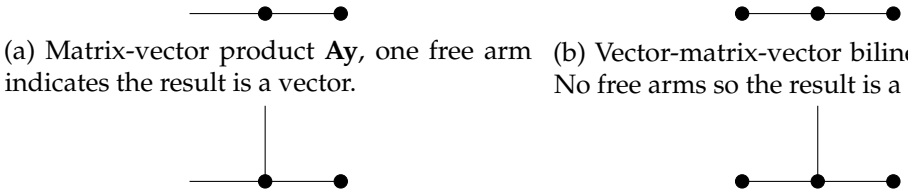
We can generalise the operation to tensors: choose an index over which to perform the product, collect every thread formed by fixing all but that one index and compute their inner product with the given vector. If the tensor has n indices, the result will have $n - 1$. A three tensor is in this way reduced to a matrix. Kolda and Bader introduce the operator $\cdot \bar{\times}_i \cdot$ for this, where i represents the index to vary [35]. For the bilinear forms we are concerned with, this leads to the following notation:

$$\mathbf{z} = \boldsymbol{\mathcal{W}} \bar{\times}_1 \mathbf{x} \bar{\times}_2 \mathbf{y} \tag{3.1}$$

$$= \boldsymbol{\mathcal{W}} \bar{\times}_3 \mathbf{y} \bar{\times}_1 \mathbf{x}. \tag{3.2}$$

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

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



(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 producing a matrix.

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

Figure 3.2: Various products expressed as Tensor Network Diagrams.

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, $(\mathbf{x}^\mathsf{T}\boldsymbol{\mathcal{W}})\mathbf{y}$ 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 [10, 51]. 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 [59, 66] . 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 [62] 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 of Conditional Restricted Boltzmann Machines [67] 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 symmetry 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 [69] 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

Perhaps the most powerful interpretation of the product is achieved by observing that it can be seen as a linear operation 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 need to 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 [35].

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 $\mathcal{W} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ must have shape $n_2 \times n_1 n_3$. The following lemma helps us describe the components of this matricisation.

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

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

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 $\text{mat}_2(\mathcal{W})$ can be though of as arising from the following construction. 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 $\mathcal{W}$ and vectors $\mathbf{x}, \mathbf{y}$ as above, we can describe the product $\mathbf{z} = \mathbf{x}^\mathsf{T} \mathcal{W} \mathbf{y}$ in terms of the mode-2 matricisation of $\mathcal{W}$ as follows:*

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

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:

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

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. □

Remarks on theorem 1

Firstly, 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 binary relationships based on correlations between both inputs. This allows it to model a rich class of binary relations between $\mathbf{x}$ and $\mathbf{y}$.

Secondly this suggests 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 also provides an interesting building block for feed-forward networks. As an example, note that a plain perceptron has been long known to be incapable of learning the exclusive-or mapping [48] – a neural network to solve the problem requires either a hidden layer or an additional input feature (specifically the conjunction of the inputs) [56]. 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.

Finally we will point out that this gives an indication of what we term the 'apparent depth' of the tensor product. While it has long been known that neural networks with a single hidden layer and appropriate non-linearity (and therefore two weight matrices) are universal function approximators [30], recent results have suggested that the depth of the network is important in keeping the number of parameters bounded [13, 68]. In particular, Eldan and Shamir show that for a particular class of radial basis-like functions – which depend only on the squared norm of the input [1], – there exist networks with two hidden layers and a number of nodes polynomial in the input dimension which can perfectly approximate the function. They then show that a shallower network with a single hidden layer can only perfectly approximate the function with a number of nodes exponential in the input dimension. [13]

Eldan and Shamir's proof constructs a three-layer network in the following manner: for an input $\mathbf{x}$, the first two layers learn to approximate the map $\mathbf{x} \to ||x||_2^2 = \sum_i x_i^2$ while the final layer learns a univariate function on the squared norm. [13]

It is straightforward to see that a quadratic tensor layer as discussed above could compute the squared norm. Consider the product $\mathbf{x}^\mathsf{T} \mathcal{W} \mathbf{x}$ and let $\mathcal{W} = [\mathbf{A}, \mathbf{B}, \mathbf{C}]_{CP}$ be represented

[1] Precisely, functions $f : \mathbb{R}^n \to \mathbb{R}$ such that $f(\mathbf{x}) = f(\mathbf{y})$ implies that $||\mathbf{x}||^2 = ||\mathbf{y}||^2$. Eldan and Shamir deal specifically with cases where the norm is the Euclidean norm (as is the case for the common squared-exponential kernel). They also require some additional constraints on the boundedness and support of the function, see [13], theorem 1, for the precise construction.

in the CP-decomposition with rank d, where d is also the size of the inputs. Then if we let $\mathbf{A} = \mathbf{C} = \mathbf{I}_d$ and $\mathbf{B}$ be a $d \times 1$ vector of ones, we have

$$\begin{aligned}\mathbf{x}^\mathsf{T}\mathcal{W}\mathbf{x} &= \mathbf{B}^\mathsf{T}\left(\mathbf{A}\mathbf{x} \odot \mathbf{C}\mathbf{x}\right) \\ &= \mathbf{1}^\mathsf{T}(\mathbf{x} \odot \mathbf{x}) \\ &= \sum_{i=1}^{d} x_i^2 = ||\mathbf{x}||_2^2.\end{aligned}$$

Thus we can compute the squared norm with only a single tensor layer, while it requires two standard layers. This is a useful intuition: using tensors in a neural network allows us to limit the number of hidden layers while maintaining much of the expressive power and polynomial parameters.

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 $\Theta(n^3)$ space to store explicitly and a cubic number of operations to use to compute a bilinear product. 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, [35] 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 [28, 27] and was rediscovered repeatedly across a range of communities over the next 50 years. [35] 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) [5] or 'parallel factors' (PARAFAC). [21] 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 [35], 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 $\mathbf{a}\mathbf{b}^\mathsf{T}$ 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 [35] 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.

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

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

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

This implies we can compute the quantity inside the brackets for all r at once simply by $\mathbf{A}\mathbf{x} \odot \mathbf{C}\mathbf{y}$ 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:

$$\mathbf{z} = \mathbf{B}^\mathsf{T} (\mathbf{A}\mathbf{x} \odot \mathbf{C}\mathbf{y}).$$

□

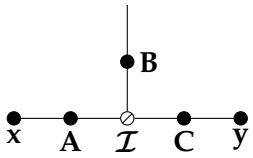


Figure 3.3: One way of representing a bilinear product in the CP-decomposition.

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. As the end goal is the gradient of a vector with respect to a matrix it should naturally be represented by a three-way tensor. Many authors avoid this by vectorising the matrix [41], but for the purposes below it is sufficient to note that the gradients are highly structured, and we can write a simple form for each element one at a time.

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

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

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

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

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

$$\frac{\partial z_j}{\partial B_{lm}} = \sum_i^{n_1} \sum_k^{n_3} A_{li} C_{lk} x_i y_k. \tag{3.6}$$

Curiously, the j and m indices do not appear in the right hand side of eq. (3.6). This appears to suggest that $\mathbf{B}$ may learn a significantly redundant structure. However, during gradient descent this will be multiplied by the back-propagated loss, will alter the update.

A final point to make about these gradients is the degree of multiplicative interactions present. This has been noted to occasionally cause slightly problematic dynamics during

gradient descent. The simplest example is to consider gradient descent on the product of two variables: $c = ab$. If a is very small and b is very large then the update applied to b will be very small and the update applied to a will be proportionally very large. Sutskever [62] uses this as motivation for using second-order information during optimisation of RNNs containing similar structures. We prefer to counsel patience – after the above step, the gradients will become perfectly aligned to the scale of the parameters and subsequent updates should be better aligned.

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 $3I^2$, a remarkable reduction.

3.3.2 Tensor Train, Tucker

The *tensor-train* decomposition has a much shorter history – it was first proposed in 2011 as a simpler way of representing a slightly hierarchical form derived from a generalisation of the singular value decomposition. [52] 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 [50] suggesting it is a prime candidate to learn with gradient descent. In the three-way case it is nearly equivalent to the more venerable Tucker decomposition. [71] The tensor train turns out to be simpler – beyond showing the similarity, we consider solely the tensor-train.

It is also important to note that the tensor-train can be used more creatively. Novikov et al. [50] use the decomposition to compress the final layers of a large convolutional neural network by reshaping the matrices into high dimensional tensors. The reduction in parameters for the tensor-train is most notable with particularly high-dimensional tensors, so this approach allows for significant compression. They also show methods of computing the required matrix vector products without having to expand the tensor. While this approach is highly successful, the implementation (especially of back-propagation) is somewhat involved and it admits a very large number of ways of applying the decomposition. To keep our search space of decompositions feasible, we consider only the straight-forward application of the tensor-train.

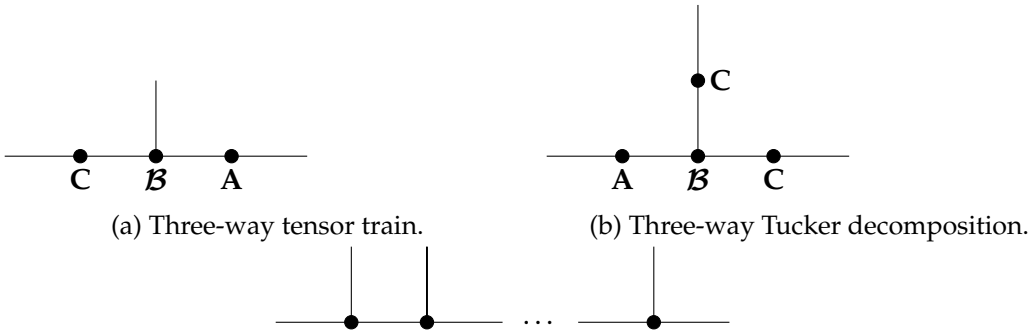
Description

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):

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

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 ensure the result of the chain of matrix products is a scalar, it must be that $r_0 = r_d = 1$. [52]

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



(a) Three-way tensor train. (b) Three-way Tucker decomposition.

(c) General Tensor Train

Figure 3.4: Diagrams of the Tensor Train and Tucker decompositions.

TT-decomposition of a three-way tensor of size $n_1 \times n_2 \times n_3$ has three 'cores' with shapes $1 \times n_1 \times r_1$, $r_1 \times n_2 \times r_2$ and $r_2 \times n_3 \times 1$. 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 $\mathcal{W}$ 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 $\mathcal{W} = [\mathbf{A}, \mathcal{B}, \mathbf{C}]_{TT}$. It is important to note that the shapes are less consistent: $\mathbf{A}$ is an $n_1 \times r_1$ matrix, $\mathcal{B}$ a $r_1 \times n_2 \times r_1$ tensor and $\mathbf{C}$ an $r_2 \times n_3$ matrix.

As this decomposition only contains contractive products, it is very well expressed as a Tensor Network diagram. Figure 3.4 shows both the general case and the three-way case – the general case shows clearly how we can build a tensor with a large number of dimensions purely out of relatively small three-way tensors. The Tucker decomposition is also presented to illustrate the similarities in the three-way case. It is also worth contrasting with figure 3.3 to see how the CP-decomposition can arise as a special case, a notion which is formalised later.

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 $\mathcal{B}$, which means we are still eventually computing a full tensor product, simply with a smaller tensor. We denote the product as

$$\mathbf{z} = \mathbf{x}^\mathsf{T}\mathbf{A}\mathcal{B}\mathbf{C}\mathbf{y}. \tag{3.7}$$

For a specific index this has the form

$$\begin{aligned} z_j &= \mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{B}_{\cdot j\cdot}\mathbf{C}\mathbf{y} \\ &= \sum_{i}^{n_1}\sum_{k}^{n_3}\sum_{\alpha_1}^{r_1}\sum_{\alpha_2}^{r_2} A_{i\alpha_1}B_{\alpha_1 j\alpha_2}C_{\alpha_2 k}x_i y_k. \end{aligned}$$

Eq. (3.7) provides a clear intuition about what the TT-decomposition is doing in the three-way case. Matrices $\mathbf{A}$ and $\mathbf{C}$ project the inputs into a new (potentially smaller) space where the bilinear product is carried out with $\mathcal{B}$ ensuring the result has been pushed back to the appropriate size.

Gradients

The gradient of $\mathbf{z}$ with respect to $\mathbf{A}$ has entries of the form:

$$\begin{aligned}\frac{\partial z_j}{\partial A_{lm}} &= \sum_{k}^{n_3}\sum_{\alpha_2}^{r_2} B_{mj\alpha_2} C_{\alpha_2 k} x_l y_k \\ &= x_l \cdot \left(\mathbf{B}^{\mathsf{T}}_{mj\cdot}\mathbf{C}\mathbf{y}\right).\end{aligned}$$

The components of the gradient with respect to $\mathbf{C}$ have a similar form:

$$\begin{aligned}\frac{\partial z_j}{\partial C_{lm}} &= \sum_{i}^{n_1}\sum_{\alpha_1}^{r_1} A_{i\alpha_1} B_{\alpha_1 jl} x_i y_m \\ &= y_m \cdot \left(\mathbf{B}^{\mathsf{T}}_{\cdot jl}\mathbf{A}\mathbf{x}\right).\end{aligned}$$

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

$$\frac{\partial z_j}{\partial B_{lmn}} = \sum_{i}^{n_1}\sum_{k}^{n_3} A_{il} C_{nk} x_i y_k.$$

This is very similar to the CP-decomposition, very little further can be added. Again the central object – which is in this case a three-way tensor – has highly redundant gradients, although it is not clear what effect this may have on learning. Finally, there is slightly more summation in the tensor-train, but it does not appear to be a dramatic enough recasting to avoid any potential instabilities due to the proliferation of multiplicative dynamics in the gradients.

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.

Proposition 3.3.2. *The rank R CP-decomposition of a tensor $\boldsymbol{\mathcal{X}} = [\mathbf{A}, \mathbf{B}, \mathbf{C}]_{CP}$ is equivalent to a TT-decomposition $[\mathbf{A}', \boldsymbol{\mathcal{B}}', \mathbf{C}']_{TT}$ with both ranks equal to R and $\boldsymbol{\mathcal{B}}'_{ijk} = 0$ except where $i = k$.*

Proof. Consider the slices of $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ formed by fixing the second index and allowing the first and third to vary. If $\boldsymbol{\mathcal{X}} = [\mathbf{A}, \mathbf{B}, \mathbf{C}]_{CP}$ then these slices can be expressed concisely as

$$\mathbf{X}_{\cdot j \cdot} = \mathbf{A}\,\mathrm{diag}(\mathbf{B}_{j\cdot})\mathbf{C}^{\mathsf{T}}$$

where $\mathrm{diag}(\mathbf{v})$ denotes the matrix with vector $\mathbf{v}$ along the leading diagonal and zero elsewhere. In the above the vector is in fact the j-th row of the factor matrix $\mathbf{B}$. It is clear the the diagonal matrix must then have shape $R \times R$ so the result has the expected shape $n_1 \times n_3$. We also verify this gives us the appropriate expression for a single element of the tensor:

$$\begin{aligned} X_{ijk} &= \left(\mathbf{A}\,\mathrm{diag}(\mathbf{B}_j.)\mathbf{C}^\mathsf{T}\right)_{ik} \\ &= \sum_{\alpha=1}^{R} A_{i\alpha} \cdot \left(\sum_{\beta=1}^{R} \mathrm{diag}(B_j.)_{\alpha\beta} C_{k\beta}\right) \\ &= \sum_{\alpha=1}^{R} A_{i\alpha} B_{j\alpha} C_{k\alpha} \end{aligned}$$

where the last step follows from diagonality.

We now construct a TT-decomposition $\boldsymbol{\mathcal{X}} = [\mathbf{A}', \boldsymbol{\mathcal{B}}', \mathbf{C}']_{TT}$ of the same tensor. The ranks of the decomposition will be $r_1 = r_2 = R$ for R the rank of the corresponding CP-decomposition. Let $\mathbf{A}' = \mathbf{A}$ and $\mathbf{C}' = \mathbf{C}^\mathsf{T}$. Construct tensor $\boldsymbol{\mathcal{B}}'$ by

$$B'_{ijk} = \begin{cases} B_{ij} & \text{if } i = k \\ 0 & \text{otherwise.} \end{cases}$$

An expression for the central slices of $\boldsymbol{\mathcal{X}}$ in terms of its TT-decomposition which follows from the element-wise definition is

$$\begin{aligned} \mathbf{X}_{\cdot j \cdot} &= \mathbf{A}'\mathbf{B}'_{\cdot j \cdot}\mathbf{C}' \\ &= \mathbf{A}\,\mathrm{diag}(B_j.)\mathbf{C}^\mathsf{T} \end{aligned}$$

by construction of $\boldsymbol{\mathcal{B}}$. □

As is demonstrated in figure 3.4b the Tucker decomposition is very similar to the tensor-train in the three-way case. For brevity we refrain from presenting the full Tucker decomposition – it suffices to note that it has matrices for all three dimensions rather than just two. In order to convert the Tucker decomposition to the tensor-train, it is enough to multiply the additional matrix with the central tensor. For that reason, we prefer the simpler tensor-train.

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

We now present some experimental results, with two aims in mind. Firstly, it is instructive to compare the performance of the two tensor decompositions when learnt by gradient descent on some straightforward tasks. Secondly we wish to verify that a tensor decomposition is capable of learning complicated structure by applying it to a classic benchmark and a real world example.

3.4.1 Random Bilinear Products

The first test uses the following procedure: generate a fixed random tensor $\mathcal{T}$ in the chosen decomposition with rank r_T. Then generate a second tensor $\mathcal{W}$ with rank r_W. At each stage t generate a pair of random input vectors $\mathbf{x}_t$ and $\mathbf{y}_t$ and compute the bilinear products $\mathbf{z}_t t = \mathbf{x}_t^\top \mathcal{T} \mathbf{y}_t$ and $\hat{\mathbf{z}}_t = \mathbf{x}_t^\top \mathcal{W} \mathbf{y}_t$. Finally update the parameters of $\mathcal{W}$ to minimise the mean squared error between $\mathbf{z}^t$ and $\hat{\mathbf{z}}^t$ using stochastic gradient descent. In this way we hope to learn an approximation to $\mathcal{T}$ in a similar setting to what might be found inside a neural network.

In practice it is computationally expedient to deal with more than one vector at a time, in all of the results for this section we use a batch size of 32. We use a fairly aggressive learning rate of 0.1. Training continues for $250,000$ parameter updates, although most of the tensors have stopped making significant improvements by that time. In all tests the input vectors had elements drawn from a uniform distribution over the range $[-1, 1]$ while unless otherwise noted the elements of the decomposition are drawn from a normal distribution with mean 0 and standard deviation 0.1.

The results of attempting to approximate a rank 100 tensor with dimensions $100 \times 100 \times 100$ are shown in figure 3.5 and are more or less as anticipated when attempting to approximate a CP-decomposition with another CP-decomposition. When the approximator is a TT-decomposition, it was found to be very difficult to achieve a good approximation – when the learning rate was dropped to account for the apparent high sensitivity of the loss the still failed to approach the error rates of the CP-decomposition. While we might expect the CP-decomposition to better represent another CP-decomposition due to the shared structure, it is nonetheless remarkable how difficult the TT-decomposition was to train.

In general we see that the quality of the approximation drops consistently with the number of parameters in the approximator, which is to be expected. More unexpected is the variance apparent in the training curves of the CP-decomposition – the very low rank approximations seem more sensitive to small changes in their coefficients. This suggests that some care may need to be taken with the learning rates and the initialisation especially in deeper models as when such difficulties will be compounded. It is also useful to note that over-specifying the rank of the decomposition appears to still aid learning.

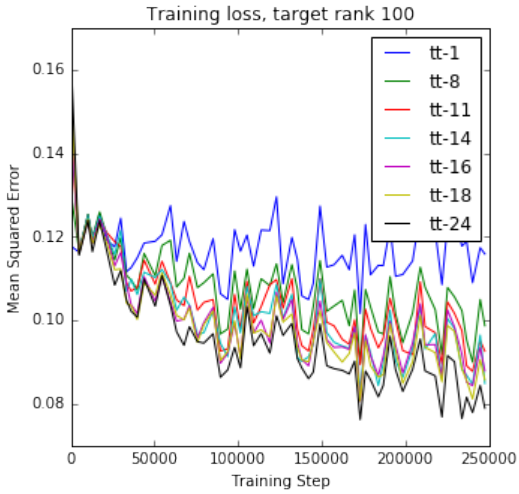
The experiment was repeated with the target tensor expressed is a TT-decomposition with ranks $r_1 = r_2 = 16$ (28,800 coefficients). Results can be seen in figure 3.6. As should be expected, the TT-decomposition performed better. In particular a TT-decomposition matching the rank of the target tensor reached a very low error extraordinarily quickly, performing much better than the CP-decomposition under the same conditions.

These results imply both decompositions are potentially useful. The CP-decomposition seems to be capable of learning reasonable approximations even when it does not necessarily reflect the structure of the underlying problem. The TT-decomposition appears to struggle in this situation, but when the problem is structured in a more favourable way it seems to take greater advantage.

3.4.2 Learning to multiply

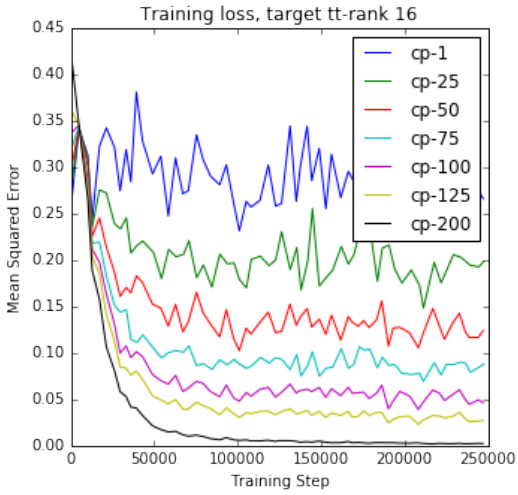
Learning bilinear functions provides a way to linearly approximate functions of two variables. A motivating example of such a function is elementwise multiplication, also known as the Hadamard product. In this section we investigate briefly the ability of the two decompositions to model such a product.

(a) Attempting to approximate by directly learning factor matrices representing a CP-decomposition of various ranks

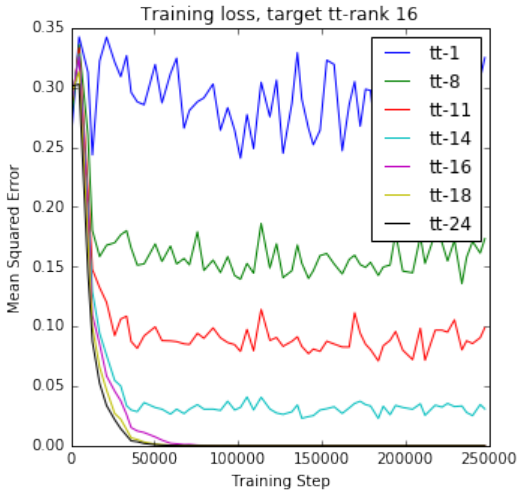


(b) Attempting to approximate by directly learning a TT-decomposition of various ranks

Figure 3.5: Results for learning direct approximations of a CP-rank 100 tensor.



(a) Attempting to approximate by directly learning a CP-decomposition of various ranks



(b) Attempting to approximate by directly learning a TT-decomposition of various ranks

Figure 3.6: Results for learning direct approximations of a tt-rank 16 tensor.

Theory

We wish to determine first how to construct a tensor that represents an elementwise product and then determine the rank of the decomposition required to represent it. Note that the tensor must have the same size in all dimensions as the inputs and output of the product must be the same size. This tensor must be *diagonal* with the diagonal elements having value one (see proposition A.0.1 in the appendix for a proof). This means all entries ijk such that $i = j = k$ must be one, all other entries zero. We denote such a tensor by $\mathcal{H}$.

Now we consider how the two proposed decompositions could model this tensor. For the CP-decomposition this is straightforward: $\mathcal{H} = [\mathbf{I}, \mathbf{I}, \mathbf{I}]_{CP}$ where $\mathbf{I}$ is the $N \times N$ identity matrix. This is can be easily verified as the definition of a bilinear product with a CP-decomposed tensor includes an elementwise product so it suffices to ensure the factor matrices do not alter the inputs. The CP-decomposition reduces the required parameters to $3N^2$ as opposed to the original N^3. This is a remarkable reduction, considering that an analogous diagonal *matrix* is the worst case for the corresponding two-way decomposition.

The TT-decomposition is not capable of reducing the parameters. In order to ensure that the central tensor product passes through the appropriate elements it must in fact be equal to $\mathcal{H}$ itself.

Experiments

Learning exact elementwise products is a special case and can clearly be done with the CP-decomposition without sacrificing parameter reduction. The question that remains is how closely a given decomposition can approximate an elementwise product, especially if there is some kind of helpful latent structure in the inputs to exploit. We test this by inducing a random structure on the inputs.

First we verify our theories about the capacity of the decompositions to represent elementwise multiplication. This is performed identically to the experiments in section 3.4.1 except that that rather than generating a random tensor to provide the targets for training, we simply multiply elementwise the input vectors. We also simplify slightly by drawing the input vectors from $\{0, 1\}$ with equal probability (elementwise multiply in this sense corresponds to a bitwise AND). Again the squared error loss is used, although a momentum term (with coefficient 0.9) was found to greatly assist learning. For a further test, the target output has a fixed permutation applied which removes the diagonal structure from the target but should still be a straightforward task. For this task we would expect that the CP-decomposition drastically outperforms the TT-decomposition.

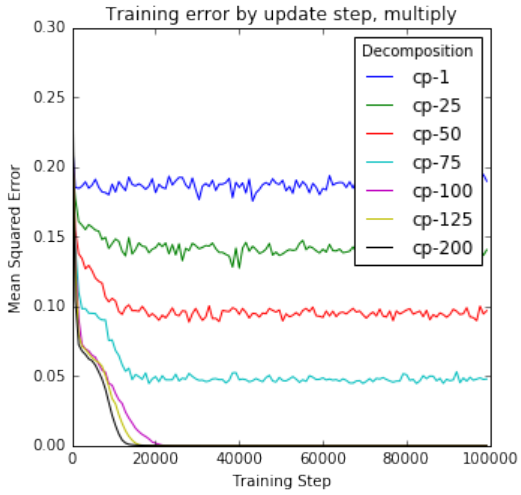
The results, figures 3.7 and 3.8, affirm our expectations. The CP-decomposition is able to consistently reduce error as rank increases while the TT-decomposition appears to struggle.

For a more realistic test we introduce a random structure to the input vectors. This is done by creating smaller vectors and expanding them to the appropriate size by copying the elements to random positions. The mapping of positions is fixed for the duration of a training run. We then experiment with varying the size of these smaller vectors and hence varying the degree of correlation while keeping the rank of the decomposition fixed. Here we have fewer preconceptions regarding the relative performance of the decompositions.

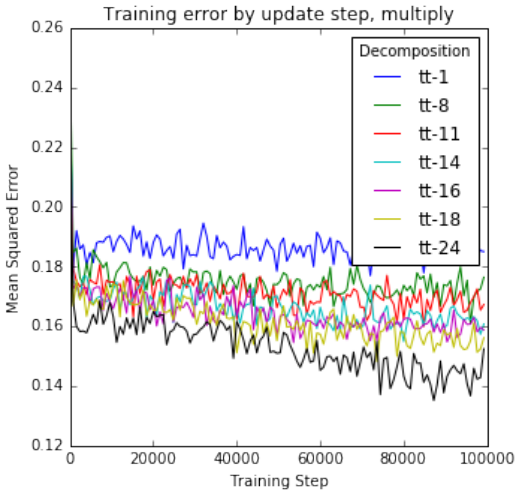
The results again show the CP-decomposition outperforming the TT-decomposition dramatically. In this case this is slightly more surprising as there is more likely to be solutions obtainable in the TT-decomposition.

3.4.3 XOR

Should go here?

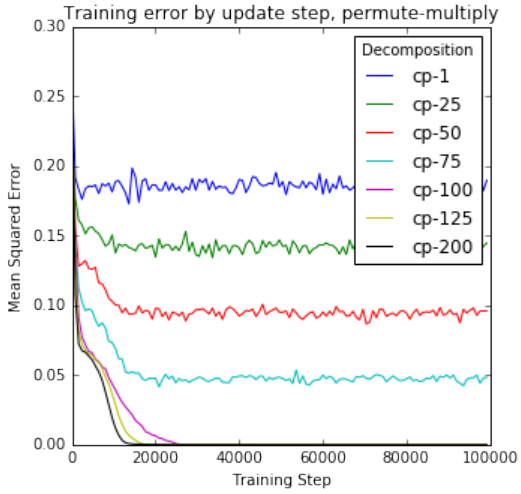


(a) Learning curves for CP-decompositions learning elementwise multiplication

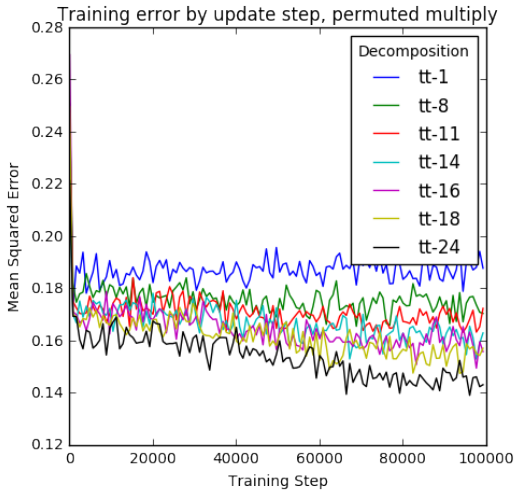


(b) Learning curves for TT-decompositions learning elementwise multiplication

Figure 3.7: Training error for various rank decompositions on the elementwise multiplication task.

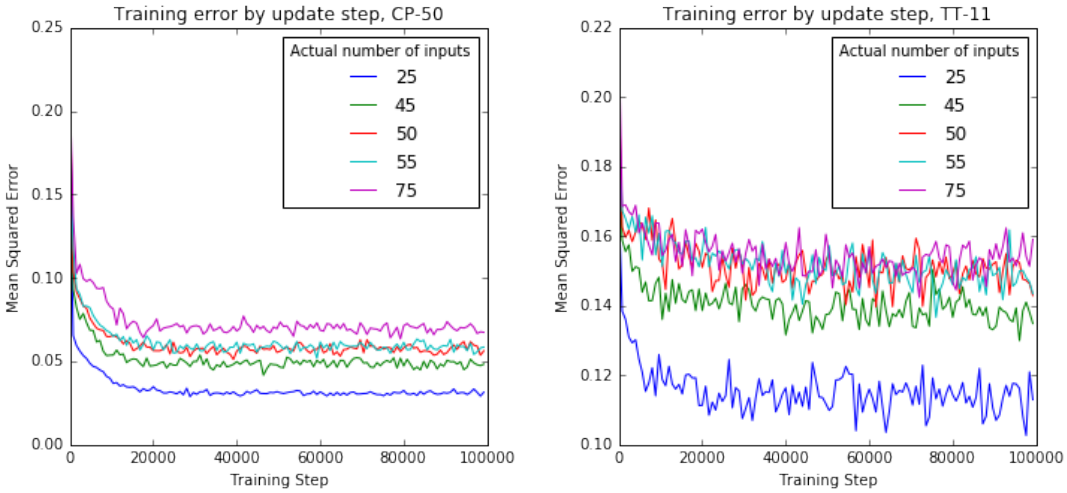


(a) Learning curves for CP-decompositions learning permuted elementwise multiplication



(b) Learning curves for TT-decompositions learning permuted elementwise multiplication

Figure 3.8: Training error for various rank decompositions on the permuted elementwise multiplication task.



(a) Learning curves for a rank 50 CP-decomposition.

(b) Learning curves for a rank 11 TT-decomposition.

Figure 3.9: Training error for two decompositions learning elementwise multiplication with varying amounts of structure.

3.4.4 Separating Style and Content

In this section we apply the tensor decompositions to a less trivial task. In particular, we address the *extrapolation* problem presented by Tenenbaum and Freeman. [69] The focus in this task in on data that is naturally presented as a function of two factors, termed *style* and *content*. Tennenbaum and Freeman propose the use of bilinear models for such data as follows.

Let $\mathbf{y}_{sc}$ be an observation vector with style s and content c. It is modelled in bilinear form:

$$\mathbf{y}_{sc} = \mathbf{a}_s^\mathsf{T} \mathcal{W} \mathbf{b}_c$$

where $\mathbf{a}_s$ and $\mathbf{b}_c$ are style and content parameter vectors respectively and $\mathcal{W}$ represents a set of basis functions independent of the parameters. A number of tasks involving learning such a model were proposed, but we focus on extrapolation: learning both the parameter vectors and the basis tensor at once in such a way that the model successfully generalises to style and content pairs not seen during training.

We model the style and content parameter vectors as fixed length, dense vectors with a unique vector for each style label and each content label. The parameters in these vectors can be learnt by gradient descent at jointly with the tensor $\mathcal{W}$. This is equivalent to representing them in a one-of-N encoding (a vector with one value per possible label, all zero apart from the entry corresponding to the label in question) and then multiplying it by a matrix. We refer to such a matrix as an *embedding matrix*. It is clear that if we were to attempt to learn the model without these embedding matrices and simply representing $\mathbf{a}$ and $\mathbf{b}$ with one-of-N encodings that the model would fail completely to generalise. A bilinear product with two such vectors would amount to selecting a single fibre from the tensor, so the tensor could only ever hope to directly store the training data. Introducing a decomposition changes this by forcing the elements of the tensor to depend on one another.

Data

One dataset explored in [69] is typographical – this provides a natural source of data defined by two factors: font and character. We refer to the character as the content and the font as the

style. We collected a small dataset of five fonts and their italic/oblique variants for a total of ten styles. Variation between styles comes from stroke width, slant and the presence of absence of serifs. From each font the uppercase and lowercase letters of the English alphabet were extracted, totalling 52 different content labels. Data was saved as 32 pixel by 32 pixel greyscale images. To represent images as vectors for the purposes of the above model they are flattened in scanline order: left to right and top to bottom.

Experiments

This is a very difficult task as we are expecting the model to produce pictures similar to ones that have never been seen in training. Further, we are trying to train a model with potentially thousands of free parameters based only on a few hundred examples. As this suggests, stopping the model from overfitting was the major challenge. The aim of the experiments was not necessarily to achieve state-of-the-art performance, but to quickly ensure that using a tensor decomposition to model non-trivial interactions was a feasible goal.

To begin we verify that a fairly small model is capable of representing the data. For this we use embeddings of size 25 and a rank 25 CP-decomposition. This model has 78,350 parameters, while a model that contained the explicit $25 \times 1024 \times 25$ tensor would have 641,550. We train the model using ADAM [34], a variant of stochastic gradient descent to minimise the squared error:

$$E = \sum_{i}^{B} ||\hat{\mathbf{y}}_{sc} - \mathbf{y}_{sc}||_2^2$$

where B is the number of elements in a batch (26 was used in the following) and $\hat{\mathbf{y}}_{sc} = \mathbf{a}_s^\mathsf{T} \mathcal{W} \mathbf{b}_c$ is the predicted image. Both the central tensor and the embedding vectors are updated at every update step. A small amount of l_2 regularisation was found to help prevent overfitting, this involves adding a penalty term to the loss of the form $\lambda ||\mathbf{X}||_F^2 = \lambda \sum_i^m \sum_j^m X_{ij}^2$ for each matrix in the model.

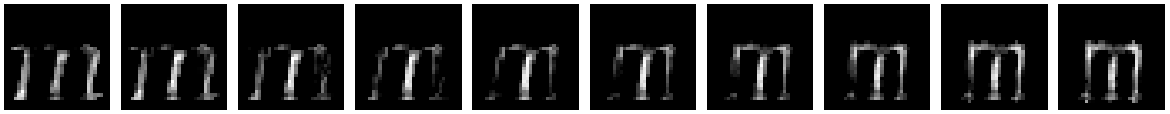
Figure 3.10 provides a quick visual inspection of what the model was able to learn. Images were generated by finding the style and content labels for a two examples and linearly interpolating first the content vector and then the style vector, generating an image with the intermediate vectors at each step.

The start and end images are not perfect; the model was small and unable to capture the training data exactly. During experimentation larger models were found to fit the data very well, but to begin to overfit after only one or two epochs of training while a smaller model tended to perform better on validation data for longer.

Intermediate stages of the interpolation have no real meaning but they indicate that the model has not simply learned to recall individual pairs of labels. We see that as either the content or the style shifts from one to another, the elements of the source image which are not shared with the target fade out and are replaced.

Results on unseen data are shown in figure 3.11. These show that the model does appear to capture some of the salient information for separating the style and content. In particular, the general shapes of the letters are roughly appropriate and it has begun to capture some information about the presence or absence of serifs and general slant. Figure 3.12 shows a harder example – the lowercase 'a' has considerable variation among various fonts and the model is unable to guess what would be appropriate for an unseen example.

(a) Linearly interpolating between two content vectors ('T' to 'm') with style vector fixed.



(b) Linearly interpolating between two style vectors with fixed content.

Figure 3.10: Learned style and content representations.

(a) Actual images from the dataset.

(b) Model output, having never seen these specific pairs during training.

Figure 3.11: Example of images generated from unseen style and content pairs, three letters from three different fonts.

(a) From the data

(b) Generated

Figure 3.12: A difficult example.

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.

Appendix A

Additional Proofs

Proposition A.0.1 (Identity Tensor). *$\boldsymbol{\mathcal{H}} \in \mathbb{R}^{N\times N\times N}$ such that*

$$\mathbf{x}^\mathsf{T}\boldsymbol{\mathcal{H}}\mathbf{y} = \mathbf{x} \odot \mathbf{y}, \qquad \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^N$$

implies

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

Proof. We prove briefly, by inspecting one component of the result. Let $\mathbf{z} = \mathbf{x}^\mathsf{T}\boldsymbol{\mathcal{H}}\mathbf{y}$. Then

$$\begin{aligned} z_j &= \mathbf{x}^\mathsf{T}\mathbf{H}_{\cdot j \cdot}\mathbf{y} \\ &= \sum_i^N \sum_k^N x_i H_{ijk} y_k \end{aligned}$$

If $z_j = x_j y_j$ as in the elementwise product, then it is clear we want H_{ijk} to be 1 if $i = j = k$. Further, if we ensure H_{ijk} is 0 when this is not the case we can see that the rest of the terms in the sums will disappear. □

Bibliography

[1] ABADI, M., ET AL. TensorFlow: Large-scale machine learning on heterogeneous systems. *arXiv preprint* (2015). arXiv: arXiv:1603.04467v2.

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

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

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] DANIHELKA, I., ET AL. Associative Long Short-Term Memory. *arXiv* (Feb. 2016). arXiv: 1602.03032.

[12] DUVENAUD, D., ET AL. Avoiding pathologies in very deep networks. *AISTATS* (Feb. 2014), 202–210. arXiv: 1402.5836.

[13] ELDAN, R., AND SHAMIR, O. The Power of Depth for Feedforward Neural Networks. In: *29th Annual Conference on Learning Theory*. Dec. 2016, 907–940. arXiv: 1512.03965.

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

[15] GLOROT, X., AND BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)* 9 (2010), 249–256.

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

[17] GRAVES, A., WAYNE, G., AND DANIHELKA, I. Neural Turing Machines. *arXiv preprint* (2014), 1–26. arXiv: arXiv:1410.5401v2.

[18] 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.

[19] GREFENSTETTE, E., ET AL. Learning to Transduce with Unbounded Memory. *arXiv preprint* (2015), 12. arXiv: 1506.02516.

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

[21] 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.

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

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

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

[25] HINTON, G. E. *A Parallel Computation that Assigns Canonical Object-Based Frames of Reference*. 1981.

[26] HINTON, G. E., AND SALAKHUTDINOV, R. R. Reducing the Dimensionality of Data with Neural Networks. *Science* 313, 5786 (2006), 504–507. arXiv: 20.

[27] 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.

[28] 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.

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

[30] HORNIK, K., STINCHCOMBE, M., AND WHITE, H. Multilayer Feedforward Networks Are Universal Function Approximators. *Neural Networks* 2 (1989), 359–366.

[31] JOULIN, A., AND MIKOLOV, T. Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. *arXiv* (2015), 1–10. arXiv: arXiv:1503.01007v4.

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

[33] KAISER, ., AND SUTSKEVER, I. Neural GPUs Learn Algorithms. *arXiv:1511.08228 [cs]* (2015), 1–9. arXiv: 1511.08228.

[34] KINGMA, D. P., AND BA, J. L. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION (2014). arXiv: 1412.6980.

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

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

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

[38] KURACH, K., ANDRYCHOWICZ, M., AND SUTSKEVER, I. Neural Random-Access Machines. *ICLR* (2016), 17. arXiv: 1511.06392.

[39] 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.

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

[41] MAGNUS, J., AND NEUDECKER, H. *Matrix Differential Calculus with Applications in Statistics and Econometrics*. 3rd ed. Wiley, 2007.

[42] 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.

[43] MEMISEVIC, R. Learning to relate images: Mapping units, complex cells and simultaneous eigenspaces. *arXiv preprint arXiv:1110.0107* (2011), 1–32. arXiv: 1110.0107.

[44] MEMISEVIC, R., AND HINTON, G. Unsupervised Learning of Image Transformations. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on* (2007), 1–8.

[45] MEMISEVIC, R., AND HINTON, G. E. Learning to represent spatial transformations with factored higher-order Boltzmann machines. *Neural computation* 22, 6 (2010), 1473–1492.

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

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

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

[49] NAIR, V., AND HINTON, G. E. Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning* 3 (2010), 807–814.

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

[51] 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.

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

[53] 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.

[54] PLATE, T. Holographic Reduced Representations. *IEEE Transactions on Neural Networks* 6, 3 (1995), 623–641.

[55] ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* 65, 6 (1958), 386–408.

[56] 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.

[57] SAXE, A. M., MCCLELLAND, J. L., AND GANGULI, S. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *Advances in Neural Information Processing Systems* (Dec. 2013), 1–9. arXiv: 1312.6120.

[58] SIGAUD, O., ET AL. Gated networks: an inventory. *arXiv* (2015). arXiv: 1512.03201.

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

[60] SMOLENSKY, P. Information processing in dynamical systems: Foundations of harmony theory. *Parallel Distributed Processing Explorations in the Microstructure of Cognition* 1, 1 (1986), 194–281.

[61] SRIVASTAVA, R. K., GREFF, K., AND SCHMIDHUBER, J. Highway Networks. *arXiv:1505.00387 [cs]* (2015). arXiv: 1505.00387.

[62] SUTSKEVER, I. Training Recurrent neural Networks. PhD thesis. 2013, 101. arXiv: 1456339 [arXiv:submit].

[63] SUTSKEVER, I., MARTENS, J., AND HINTON, G. Generating Text with Recurrent Neural Networks. *Neural Networks* 131, 1 (2011), 1017–1024. arXiv: 9809069v1 [arXiv:gr-qc].

[64] SUTSKEVER, I., ET AL. On the importance of initialization and momentum in deep learning. *Jmlr* 28, 2010 (2013), 1139–1147.

[65] 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.

[66] TAN, P.-N., STEINBACH, M., AND KUMAR, V. *Introduction to data mining*. Pearson Addison Wesley, 2006, 769.

[67] 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.

[68] TELGARSKY, M. Benefits of Depth In Neural Networks. In: *29th Annual Conference on Learning Theory*. Vol. 49. 1. 2016, 1–19. arXiv: 1602.04485.

[69] TENENBAUM, J. B., AND FREEMAN, W. T. Separating style and content with bilinear models. *Neural computation* 12, 6 (2000), 1247–1283.

[70] THE THEANO DEVELOPMENT TEAM, ET AL. Theano: A Python framework for fast computation of mathematical expressions. *arXiv* abs/1605.0 (2016), 19. arXiv: 1605.02688.

[71] TUCKER, L. R. Some mathematical notes on three-mode factor analysis. *Psychometrika* 31, 3 (1966), 279–311.

[72] VINCENT, P., ET AL. Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion. *Journal of Machine Learning Research* 11, 3 (2010), 3371–3408. arXiv: 0-387-31073-8.

[73] VINYALS, O., FORTUNATO, M., AND JAITLY, N. Pointer Networks. In: *Advances in Neural Information Processing Systems*. 2015. arXiv: arXiv:1506.03134v1.

[74] 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.

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

[76] WU, Y., ET AL. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint* (2016), 1–23. arXiv: 1609.08144.

[77] WU, Y., ET AL. On Multiplicative Integration with Recurrent Neural Networks. *arXiv* (June 2016). arXiv: 1606.06630.

[78] XU, K., ET AL. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *arXiv preprint* (2015). arXiv: arXiv:1211.5063v2.