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.

WHO??

Contents

Figures

Chapter 1

Introduction

Recurrent Neural Networks (RNNs) are powerful machine learning models for learning mappings between pairs of sequences. Such tasks are typically highly challenging and include language modelling, machine translation, speech recognition and audio classification. They have achieved remarkable success even at immense scale [102], but the most successful architectures are opaque, poorly understood and seemingly redundant while simple clear models tend to have serious issues learning complex sequence asks [6]. Understanding how RNNs are able to solve complex tasks and designing new, simpler architectures that are well-understood from the outset is therefore of high importance.

RNNs make computations at each time step of the input sequence based on the current input value and a fixed size vector of hidden states. At each step a new state is produced and passed forward. This allows the network to model complicated temporal dynamics. The most successful architectures have two key components to this computation: additive state updates and multiplicative gating. The former allows easy integration of new information at each time step while the latter provides the ability to selectively ignore elements of the state or input.

Multiplicative gating involves computing element-wise products of vectors. We can generalise this as computing a product involving a three-way tensor of weights, termed a bilinear product. Looking to understand and generalise the structures prevalent in existing RNNs we investigate in detail the properties of the bilinear product. This leads to the remarkable result, expressed in theorem 1, that these bilinear products operate on the exclusive-or of features. The implications of this are clear and strong – bilinear products operate implicitly at a higher level than the standard linear operations composed to form neural networks.

The downside of these tensor products is that using them explicitly requires storing the tensor which can be very large. We therefore look to the multi-linear algebra literature to investigate methods of tensor decomposition. Finding a useful decomposition would allow us to incorporate this highly expressive tensor product in a neural network with a feasible number of parameters. We find that the simplest decomposition, CANDECOMP/PARAFAC [8, 34], fullfils our criteria both theoretically and experimentally.

Equipped with this tensor decomposition we now propose two new classes of RNNs: the Generalised Multiplicative RNN (GMR) and the Tensor Gate Unit (TGU). The GMR is a very simple way to incorporate a tensor into a neural network and generalises a number of recently proposed architectures [90, 103]. The TGU includes an additive state update and standard multiplicative gates in addition to the bilinear product. The architecture is based on the observation that the gate is the most important part of the architecture and that if we ensure it is sufficiently expressive, we can remove a number of other dependencies. This results in a unique, clearly defined architecture with full expressive power and a minimum of extraneous elements.

These architectures are then evaluated empirically. We consider standard synthetic tasks and achieve excellent results including successfully training a TGU to capture time dependencies up to 10,000 steps, more than 10 times the best previously reported. None of the synthetic tasks common in the literature adequately exercise the ability of the model to store and retrieve arbitrary patterns. As this is an essential task for solving many complex problems we propose a novel synthetic form of *variable binding* which tests the ability of the models to use their hidden units in this fashion.

Finally, we demonstrate that our proposed RNNs are highly competitive with the best existing techniques on real world data. In doing so, we investigate the hypothesis that controlling the rank of the tensor decomposition can provide an important regularisation effect, helping greatly to reduce overfitting. These results justify the intuitions and theoretical analysis behind the proposed architectures as well as indicating they provide a useful class of novel RNNs with clear performance benefits.

Chapter 2

Background and Related Work

2.1 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 [80], indeed simple feed-forward networks are often termed 'multi-layer perceptrons' (MLPs). MLPs solve the key limitations of the early perceptron [67], but it was not until 1986 when a reliable and general purpose procedure (*back-propagation*) was proposed [81] 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. Computationally and notationally it is efficient to express many neurons at once, using vectors and matrices. If we consider the input to a network as a vector of features $\mathbf{x}$, and let $\mathbf{W}$ be a matrix whose rows contain the weights for each neuron, the output is computed by

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

where $f : \mathbb{R}^n \mapsto \mathbb{R}^n$ is a non-linear function applied element-wise. Common examples include the logistic sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ or a linear rectifier $\rho(x) = \max(0, x)$ [68]. We also include a bias term $\mathbf{b}$ which is also learned, enhancing the range of the possible transformations greatly.

This gives us a single layer of neurons, a classic MLP will perform this process twice. 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.

Such neural networks are universal function approximators [44]. However 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 [95]. 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 [26, 83]. Several methods have been proposed which successfully alleviate this, such as layer-wise pre-training [40, 98] and more recently residual or skip connections which change the structure of the graph to allow better flow of information [35, 45].

2.1.1 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 transform each $\mathbf{x}_i$ into $\mathbf{y}_i$. To do this we define a loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}$ 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 $\mathcal{L}_{SE}(\mathbf{y}, \hat{\mathbf{y}}) = \sum_d (y_d - \hat{y}_d)^2$ or the cross entropy $\mathcal{L}_{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_d y_d \log(\hat{y}_d)$, ensuring some safeguards are put in place to avoid attempting to compute $\log 0$. 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 the set which minimises the loss across the data items. Training the network amounts to solving the following optimisation problem:

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

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 $\mathbf{W}$. Typically the gradient is evaluated over a small subset or "mini-batch" of the data per iteration which gives 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 [81]. 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 [97].

2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) considered here generalise feed-forward networks to address problems in which we wish to map a *sequence* of inputs vectors $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \dots \mathbf{x}_T)$

[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.

to a sequence of output vectors $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2, \ldots \mathbf{y}_T)$. They have been applied successful to a wide range of tasks which can be framed in this way including statistical language modelling [65] (including machine translation [10, 102]), speech recognition [30], polyphonic music modelling [7], music classification [12], image generation [33], automatic captioning [100, 104] and more.

2.2.1 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 [23] computes its hidden states with the following recurrence:

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

where $f(\cdot)$ is some elementwise non-linearity, often the hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. 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 (2.1) is closely related 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.

2.2.2 Training

We can train this (or any of the variants we will see subsequently) using back-propagation. Often termed "Back Propagation Through Time" [101], this requires using the chain rule to determine the gradients of the loss with respect to the network parameters in the same manner as feed-forward networks.

To illustrate this, consider a loss function for the whole sequence of the form

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

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

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

from the definition of the loss and the fact that a hidden state may affect all future losses. To determine each $\nabla_{\mathbf{h}_i} \mathcal{L}_j$ (noting $j \geq i$), we apply the chain rule, to back-propagate the error from time j to time i. This is the step from which the algorithm derives its name, and simply requires multiplying through adjacent time steps. 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

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

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

We can now derive an update rule for the parameters by observing[2]

$$\begin{aligned}\nabla_{\mathbf{W}}\mathcal{L} &= \sum_{i=1}^{T}\nabla_{\mathbf{W}}\mathcal{L}_i \\ &= \sum_{i=1}^{T}\sum_{j=1}^{i}(\nabla_{\mathbf{W}}\mathbf{h}_j)(\nabla_{\mathbf{h}_j}\mathcal{L}_i)\end{aligned} \tag{2.3}$$

and applying the above. For the input matrix $\mathbf{U}$ and the bias the process is the same.

2.2.3 Issues with Vanilla RNNs

Vanishing and Exploding Gradients

Equation (2.2) reveals a key pathology of the vanilla RNN – vanishing gradients. This occurs when the gradient of the loss vanishes to a negligibly small value as we propagate it backward in time, leading to a negligible update to the weights. $\nabla_{\mathbf{h}_j}\mathcal{L}_j$ (a column 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 rapidly explode. If the largest eigenvalue is small, then the gradient will vanish. This issue was first presented in 1994 [6] where it was shown to be unavoidable if the network needs to both store information and be robust to noise. However, this has not precluded further research into alleviating the main symptoms, for a thorough treatment including necessary conditions for vanishing and the complementary exploding problem, see [76].

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.

The key symptom of vanishing gradients in RNNs is that it makes it much harder to learn to store information for long time periods [6]. This makes RNNs struggle to solve simple-seeming tasks in which the solution requires remembering an input for many time steps.

Butterfly Effect

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 [57]. In RNNs this manifests as near-discontinuity of the loss surface [76] as a change (for example, to a weight during back-propagation) which may be in a downward direction on the loss surface can cause instabilities 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

[2]For clarity, when we consider the gradient of anything with respect to a matrix, such as $\nabla_{\mathbf{W}}\mathcal{L}$ we implicitly vectorise the matrix. More precise notation would be $\nabla_{\text{vec}(\mathbf{W})}\mathcal{L}$, but this adds extra distraction into already unfortunately cluttered equations. For a definition of the vectorisation operation see section 3.2. This means $\nabla_{\mathbf{W}}\mathcal{L}$ has shape $nn \times 1$, where n is the number of hidden states. Correspondingly, $\nabla_{\mathbf{W}}\mathbf{h_j}$ has shape $nn \times n$, so the shapes of the matrix multiplication in eq. (2.3) are appropriately defined.

norm of gradients [76] or using a regulariser to encourage gradual changes in hidden state [53].

2.3 Popular Alternatives: LSTM and GRU

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 number of subtle variants exist, the standard LSTM we will consider here has the form: [43, 28]

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

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 element-wise 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. [10] 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 [47, 43]. 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 [35, 20, 92]. Unfortunately the presence of the gate complicates this perspective – this will be analysed in detail in chapter 4.

These architectures are often used at large scale to solve very hard real world tasks. However, the synthetic tasks on which the LSTM was originally validated [43] overlook a key requirement of a successful RNN architecture. The tasks focus on problems involving long time dependencies, but all involve remember either symbols from a fixed, small vocabulary or a single real number. We suggest that it is also necessary to evaluate the ability of RNNs to learn to use all of their memory cells at once to store distributed representations of the current context from which elements of the sequence can be reconstructed. While this ability is used implicitly, especially in architectural advancements targeted at language modelling [11, 9, 58], no specific empirical validation of this ability is regularly carried out.

2.4 Other Ideas

While the LSTM and the GRU are by far the most popular RNN architectures, many other solutions have been proposed to the problems that plague the Vanilla RNN. We divide these into two categories: training and memory augmentation. Training solutions focus on adjusting the structure of the RNN or the training algorithm to avoid vanishing or exploding gradients and hence allow it to learn to solve difficult tasks. Memory augmentation approaches view an RNN as a device which interacts with a memory (in the form of its hidden states) and either augment the memory with additional structures or impose specific access patterns to encourage more appropriate use of this memory.

2.4.1 Enhancing Training

Methods of addressing issues when training RNNs can be further divided into two categories: architectural and algorithmic. Architectural solutions, such as the LSTM and GRU, seek to alter the way in which hidden states are calculated. Algorithmic solutions look to change the training process itself to better avoid or handle the various pathologies. These are complementary lines of enquiry – advanced architectures can still benefit from more sophisticated training techniques.

Architectural Solutions

An early attempt to solve the vanishing gradient 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 [38]. This idea has been revisited recently in the form of the Clockwork RNN [52] 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 [13] and Gated Feedback RNNs [15]

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 closely related approach is augmenting RNNs to add an attention mechanism. This is especially popular in translation tasks, in which a common approach is to use two separate RNNs in an *encoder-decoder* arrangement where the first network encodes the sentence into a fixed length vector and the second expands the vector into a new sentence in a different language [11, 58]. Content-based attention is often used to improve these models, by allowing the decoder access to all previous states of the encoder via a weighted sum. Weights are derived from similarities between each encoder state and a vector produced by the decoder at each step [9, 21].

While this achieves excellent performance at extraordinary scale [102], it also requires computing similarities for every step of the output with every step of the input, which will scale quadratically in sentence length. Consequently, it seems like a useful avenue of research to allow the encoder RNN to learn a more useful reduced representation of the sentence to remove the need for the attention mechanism.

A simpler recent architecture of note is the Unitary Evolution RNN [3] 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.

Another principled way to approach the design of RNNs is to use *strongly typed* architectures. [4] This approach is inspired by the desire to alter the main source of problems in classic RNNs – the recurrent weight matrix. The argument is that continually applying the same weight matrix to the hidden states leads to incoherent features in the hidden state. The solution is to attempt to *diagonalise* the state updates. This corresponds to ensure all operations on the hidden state apply elementwise, and that the input should be the only item projected through weight matrices.

Another line of enquiry which can help with long time-dependencies focuses on the initialisation of the network. IRNNs [55], which simply initialise the recurrent weights matrix to the identity (presaged by the nearly diagonal initialisation in [66]), perform remarkably well on pathological tasks. The importance of the initialisation of the recurrent weights matrix was further emphasised in [37] 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. The first of these is to use approximate second-order methods to try and rescale the gradients appropriately. A number of notable attempts used Hessian-Free Optimization [61, 7] 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 [91].

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

A notable recent work applies the recently proposed path-SGD [69] to Vanilla RNNs with ReLU activations. This involves re-casting the RNN as a very deep feed-forward network with shared weights across times-steps. The authors then regularise the update based on the scale of the individual paths through the network. This allows them to achieve significant results with an architecture that is otherwise very challenging to train – the linear rectifier is unbounded in one direction which typically leads rapidly to exploding gradients [70].

2.4.2 Memory

This line of research attempts to enhance the capabilities of RNNs by focusing on the way they interact with their memory. 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 [29]. 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 [31]. A similar approach is also found in Joulin and Mikolov's Stack Augmented RNN [46]. 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 [29].

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 [54]. Other related works include Pointer Networks [99] and Neural GPUs [48]. 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 [18]. 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 [78]. 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.

2.5 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 [39]. Sigaud et al. separate modern uses into two categories based on the intuition behind the application [84]. 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 [88]. 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 [62, 84] 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 [63] which generalised Restricted Boltzmann Machines [86] (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

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

where $\sigma(\cdot)$ is the sigmoid discussed earlier [63]. This architecture very naturally extends the

operation of a feed-forward network to deal with two inputs, 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 [94, 64].

The Multiplicative RNN [90, 89] 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 [61], 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. (2.1) to use element-wise multiplication instead of addition [103]. 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. (2.1). 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.

LSTM-style gating also continues to receive attention, being employed in Gated-Attention Readers to combine representations of questions and potential answers [19]. It is also used in PixelCNN and WaveNet [73, 72], allowing to model to choose how particular features propagate upwards through layers.

It is important to realise that all of these multiplicative structures have an inherent tensor structure, whether it is made implicit or not. In chapter 3 we investigate with more rigour the types of tensor products which generalise many of these multiplicative interactions.

Chapter 3

Three-way Tensors

In order to represent functions of two arguments with neural networks, three-way tensors are unavoidable. In this chapter we discuss the vector-tensor-vector bilinear product and assess its suitability, both theoretically and empirically, to be used inside a neural network,

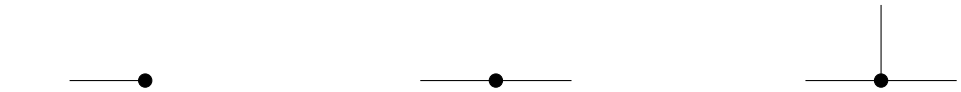
This product admits a surprising number of interpretations including theorem 1 – that it allows us to operate on a pairwise exclusive-or of features. This is a powerful result which shows that admitting tensors into neural networks can solve the exclusive-or problem without any hidden units. Further, we demonstrate the expressive power of the tensor by showing that a polynomially sized network with a single tensor product is capable of expressing a class of functions which standard two layer networks can not approximate accurately without exponential size.

The downside of these tensor products is the storage requirements. We investigate methods for decomposing tensors into products of smaller objects. Finally we conduct experiments to show that learning the coefficients of a decomposed tensor can be done in situ with gradient descent. We attempt to investigate the tradeoff between the complexity of the decomposition and the expressivity of the product and find that it scales as expected.

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, mostly follow [51] 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. We will often 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 B.1 in the appendix provides examples of the possibilities.

When dealing with tensor-tensor products, 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



(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 $\mathcal{C} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$

Figure 3.1: Example tensor network diagrams.

with a a few of special cases. In particular, we need to multiply a three-tensor by a vector and a matrix by a vector. 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

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

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 [74]. This name arises because we form the output by joining a shared dimension (by element-wise 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 [51]. For the bilinear forms we are concerned with, this leads to the following notation:

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

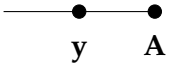
When $\mathcal{W}$ is a three-way tensor, we prefer a more compact notation

$$\mathbf{z} = \mathbf{x}^\mathsf{T} \mathcal{W} \mathbf{y}. \tag{3.2}$$

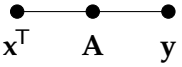
This loses none of the precision of the more verbose notation provided we make clear that we intend $\mathbf{x}$ to operate along the first dimension of the tensor and $\mathbf{y}$ the third such that $(\mathbf{x}^\mathsf{T}\mathcal{W})\mathbf{y}$ exactly corresponds with equation (3.1). With either notation, for a tensor $\mathcal{W} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, 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 [16, 74]. 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 provides examples for these simple objects.

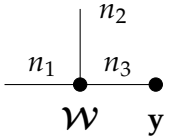
Where these diagrams are especially useful is for representing contractive products where we sum over the range of a shared index. We represent this by joining the respective arms. As an example, a matrix-vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$ has such a contraction: $y_i = \sum_j A_{ij} x_j$. 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.



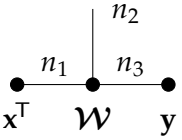
(a) Matrix-vector product $\mathbf{Ay}$, one free arm indicates the result is a vector.



(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.

3.2 Interpreting 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 $\mathcal{W}$.

3.2.1 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 $\mathcal{W}$ as $z_j = \mathbf{x}^\mathsf{T} \mathcal{W}_{\cdot j \cdot} \mathbf{y}$ which reveals the motivation behind the notation in equation (3.2). It also shows clearly each element of $\mathbf{z}$ is 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:

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

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 a 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 [85, 93] . 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

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

In a bilinear tensor product, each component of the result takes this form. 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 therefore interpret each coordinate of the result of the bilinear tensor product as being an independent similarity measure with a different interpretation 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 $\mathbf{y}$. This becomes clear if we write out each element of the result vector

$$\mathbf{z} = \begin{bmatrix} \mathbf{x}^\mathsf{T}\mathbf{W}_{\cdot 1}.\mathbf{y} \\ \mathbf{x}^\mathsf{T}\mathbf{W}_{\cdot 2}.\mathbf{y} \\ \vdots \\ \mathbf{x}^\mathsf{T}\mathbf{W}_{\cdot n_2}.\mathbf{y} \end{bmatrix} = \begin{bmatrix} \langle \mathbf{x}, \mathbf{W}_{\cdot 1}.\mathbf{y} \rangle \\ \langle \mathbf{x}, \mathbf{W}_{\cdot 2}.\mathbf{y} \rangle \\ \vdots \\ \langle \mathbf{x}, \mathbf{W}_{\cdot n_2}.\mathbf{y} \rangle \end{bmatrix} = \begin{bmatrix} \cos\theta_1 \cdot ||\mathbf{x}||_2 \cdot ||\mathbf{W}_{\cdot 1}.\mathbf{y}||_2 \\ \cos\theta_2 \cdot ||\mathbf{x}||_2 \cdot ||\mathbf{W}_{\cdot 2}.\mathbf{y}||_2 \\ \vdots \\ \cos\theta_{n_2} \cdot ||\mathbf{x}||_2 \cdot ||\mathbf{W}_{\cdot n_2}.\mathbf{y}||_2 \end{bmatrix}.$$

3.2.2 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.

This interpretation is clear from the expression of the product in equation (3.1). Simply by inserting parentheses we observe that we are first generating a matrix in 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 [89] in the context of 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 proposed factorisation of the tensor) was put to use earlier in Conditional Restricted Boltzmann Machines [94]in a more feed-forward setting.

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.

3.2.3 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 explained in [96] 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.

To make this concrete 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. Generalising 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 a distinct representation of both vectors in terms of the independent basis expressed by the tensor.

3.2.4 Operation on the outer product

The most powerful interpretation of the product is achieved by observing that it is 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. 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 *matricisation* of a tensor. Intuitively this operation is a way of *unfolding* or *flattening* a tensor into a matrix, preserving its elements. Specifically we define the mode-n matricisation of a tensor as an operation which takes all mode-n fibres of the tensor and places them as columns of 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 straightforward, describing the exact permutation of the indices is awkard – for the general case see [51].

As an example consider the three-way tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{2\times3\times4}$ with slices

$$\mathbf{X}_{1\cdot\cdot} = \begin{bmatrix} a & d & g & j \\ b & e & h & k \\ c & f & i & l \end{bmatrix}, \qquad \mathbf{X}_{2\cdot\cdot} = \begin{bmatrix} m & p & s & v \\ n & q & t & w \\ o & r & u & x \end{bmatrix}. \tag{3.3}$$

To construct $\text{mat}_2(\boldsymbol{\mathcal{X}})$ we fix the first and third indices and place the resulting threads as columns of a new matrix. Fixing the first index corresponds to choosing one of the slices presented above while fixing the third amounts to choosing a column of the slice. The result is:

$$\text{mat}_2(\boldsymbol{\mathcal{X}}) = \begin{bmatrix} a & d & g & j & m & p & s & v \\ b & e & h & k & n & q & t & w \\ c & f & i & l & o & r & u & x \end{bmatrix}. \tag{3.4}$$

Although this 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:

$$\text{vec}(\mathbf{A}) = \begin{bmatrix} \mathbf{A}_{\cdot1}^\mathsf{T} & \mathbf{A}_{\cdot2}^\mathsf{T} & \dots & \mathbf{A}_{\cdot n}^\mathsf{T} \end{bmatrix}^\mathsf{T}.$$

Note that a mode-2 matricisation of a three-way tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{n_1\times n_2\times n_3}$ must have shape $n_2 \times n_1 n_3$. The following lemma helps us describe the resulting matrix.

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

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

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

The indices into $\text{mat}_2(\boldsymbol{\mathcal{W}})$ can be though of as arising from the following construction. 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 $i + (k-1)n_1$. □

To understand the lemma it helps to consider an example. Using the $2 \times 3 \times 4$ tensor $\boldsymbol{\mathcal{X}}$ defined in equation (3.3), fixing the second index to 2 gives a 4×2 transposed slice:

$$\mathbf{X}_{:2:}^{\mathsf{T}} = \begin{bmatrix} b & n \\ e & q \\ h & t \\ k & w \end{bmatrix}.$$

Vectorising this matrix takes each column and stacks them, giving

$$\text{vec}(\mathbf{X}_{:2:}^{\mathsf{T}}) = \begin{bmatrix} b & e & h & k & n & q & t & w \end{bmatrix}^{\mathsf{T}}$$

which is precisely row two in equation (3.4).

These flattenings are important as they allow us to implement many operations involving tensors in terms of large matrix operations which can be efficiently parallelised. The next lemma describes how we use a flattened tensor to implement a bilinear product.

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

$$\mathbf{z} = \text{mat}_2(\boldsymbol{\mathcal{W}}) \text{vec}\left[\mathbf{y}\mathbf{x}^{\mathsf{T}}\right] \tag{3.5}$$

Proof. To prove this we can compare the expressions for a single element of the result. An element z_j from equation (3.5) 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:

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

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

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

□

Therefore to understand the bilinear product it helps to understand the matrix $\mathbf{y}\mathbf{x}^{\mathsf{T}}$. This matrix takes the following form:

$$\mathbf{y}\mathbf{x}^{\mathsf{T}} = \begin{bmatrix} x_1 y_1 & \dots & x_{n_1} y_1 \\ \vdots & \ddots & \vdots \\ x_1 y_{n_3} & \dots & x_{n_1} y_{n_3} \end{bmatrix}$$

which contains all possible products of pairs of elements, one from each vector. This enables the following result.

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 pairwise 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. □

Implications

1. This indicates that the bilinear tensor products operate implicitly on higher level features, constructed by combining both inputs. This indicates that it is capable of capturing complex relationships between both. This allows it to model a rich class of binary relations.

2. The case where both inputs are the same, $\mathbf{z} = \mathbf{x}^\mathsf{T}\mathcal{W}\mathbf{x}$, provides an interesting building block for feed-forward networks. A plain perceptron has been long known to be incapable of learning the exclusive-or mapping [67] – a neural network to solve the problem requires either a hidden layer or an additional input feature (specifically the conjunction of the inputs) [81]. By corollary 1.1 this quadratic tensor form would capture the conjunctive feature and be capable of solving the exclusive or problem without hidden layers or hand-engineered features.

3. 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 [44], recent results have suggested that the depth of the network is important in keeping the number of parameters bounded [22, 95]. We claim that allowing tensor products has the effect of increasing the apparent depth of the network without adding additional hidden layers.

The final claim requires some justification. 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 [22]. This is a very recent result (published in COLT 2016), which is remarkable as it proves an impossibility: there is a realistic class of functions which it is impossible to approximate accurately with a two layer network with polynomial size. However, this restriction can be subverted if we allow tensor semantics in the first layer of the network.

To show this, we first require the following lemma.

Lemma 3.2.3 (Squared norm)**.** *A tensor layer $\mathbf{x}^\mathsf{T}\mathcal{W}\mathbf{x}$ is capable of exactly computing the squared Euclidean norm $||\mathbf{x}||_2^2 = \sum_i x_i^2$.*

Proof. Recalling lemma 3.2.2, $\mathbf{x}^\mathsf{T}\mathcal{W}\mathbf{x} = \text{mat}_2(\mathcal{W})\,\text{vec}(\mathbf{x}\mathbf{x}^\mathsf{T})$. Note that the diagonal elements of $\mathbf{x}\mathbf{x}^\mathsf{T}$ are the components x_i^2 which we need to sum to compute the squared norm. We simply need to define $\text{mat}_2(\mathcal{W})$ such that it picks out the correct elements and sums them.

[1] Precisely, functions $f : \mathbb{R}^n \mapsto \mathbb{R}$ 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 [22] theorem 1, for the precise construction.

If $\mathbf{x} \in \mathbb{R}^d$ then $\mathcal{W} \in \mathbb{R}^{d \times 1 \times d}$ as we only require a single output. The matricisation is then a $1 \times d^2$ row vector and we simply need to set its elements zero apart from those with indices of the form $1 + id$ for $i \in \{1, \ldots, d\}$. The matrix product will then simply pick out the diagonal elements and sum them, computing the squared norm as required. □

Theorem 2 (Exponential expressivity)**.** *There exists a class of functions which can be approximated arbitrarily well by a two layer network including a single tensor layer with a number of nodes polynomial in the size of the input which require an exponential number of nodes to be approximated by a standard feed-forward layer.*

Proof. We make use of Eldan and Shamir's proof [22]. This is highly technical so we only provide a constructive outline here, which shows where we need to fit in. The proof in [22] shows that for a given class of radial basis-like functions a two-layer feed-forward network requires exponential width to be able to approximate below a certain error.

They then construct a three-layer network in the following manner: for an input $\mathbf{x}$, the first two layers learn to approximate the map $\mathbf{x} \mapsto ||x||_2^2 = \sum_i x_i^2$ while the final layer learns a univariate function on the squared norm, it is then proved that this required that each layer have polynomial width.

Using lemma 3.2.3, we can compute the squared norm exactly with a single layer. We can therefore construct a network in the same fashion as Eldan and Shamir, replacing the first two layers with a single tensor layer.

This plugs directly into the proof of lemma 10 in [22, pp. 18–19], simply noting that we must duplicate the tensor layer a polynomial number of times. At most we will need the same number of nodes in the final hidden layer of Eldan and Shamir's three layer network. □

This is a remarkable result, as it shows the increase in expressive power gained by allowing tensor representations. We have shown a single tensor layer can do the work of two feed-forward layers. This provides a strong motivation to explore architectures incorporating these tensor semantics.

3.3 Tensor Decompositions

Bilinear products show great promise. 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 similar number of operations to use to compute a bilinear product. 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, [51] 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 [42, 41] and was rediscovered repeatedly across a range of communities over the next 50 years. [51] 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) [8] or 'parallel factors' (PARAFAC). [34] 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 [51], but they are not relevant to the discussion here.

Description

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

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

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

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

It is convenient to stack these factors into matrices – we differ slightly from [51] 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.

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

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

Proof. This follows from straightforward rearranging. Firstly

$$z_j = \sum_{i}^{n_1} \sum_{k}^{n_3} W_{ijk} x_i y_k = \sum_{i}^{n_1} \sum_{k}^{n_3} \sum_{r}^{R} A_{ri} B_{rj} C_{rk} x_i y_k = \sum_{r}^{R} B_{rj} \left(\sum_{i}^{n_1} A_{ri} x_i \cdot \sum_{k}^{n_3} C_{rk} y_k \right). \tag{3.6}$$

We can compute the quantity inside the brackets for all r at once by $\mathbf{A}\mathbf{x} \odot \mathbf{C}\mathbf{y}$, which results in a length R vector. Equation (3.6) 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. This is naturally expressed by the following matrix multiplication:

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

□

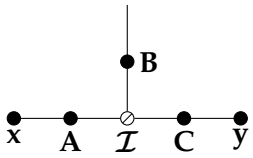


Figure 3.3: Bilinear product in the CP-decomposition.

This proposition casts the CP decomposition into a form that looks very similar to some of the RNNs discussed in chapter 2. A key similarity is with the GRU – the manner in which the reset gate is applied to the hidden states during the production of the candidate update $\mathbf{z}_t$ could be almost be viewed as a non-linear version of the above.

To express the bilinear product above as a tensor network diagram 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 represents element-wise multiplication of the two vectors (see proposition D.1.1 in the appendix). This is represented diagrammatically with the symbol $\oslash$. This is shown in figure 3.3.

Discussion

The CP decomposition does a remarkable job of preserving the expressivity. To illustrate this consider a special case of the form $\mathcal{W} = \sum_r^R \mathbf{a}_r \otimes \mathbf{e}_r \otimes \mathbf{c}_r$, where $\mathbf{e}_i$ are standard basis vectors with all elements zero and a one in position i. Consider $\mathbf{z} = \mathbf{x}^\mathsf{T} \mathcal{W} \mathbf{y}$. Suppose, without loss of generality, that $\mathbf{z} \in \mathbb{R}^R$. Then

$$z_r = \langle \mathbf{a}_r, \mathbf{x} \rangle \cdot \langle \mathbf{b}_r \mathbf{y} \rangle.$$

If we apply a sigmoid non-linearity, then $\sigma(z_r) > 0.5$ implies z_r is positive so the dot products must have the same sign. $\sigma(z_r) < 0.5$ implies z_r is negative. If inserted in a neural network, the CP decomposition has the ability to compute the exclusive-or of features.

Bilinear products with the CP decomposition also retain the ability to express element-wise multiplication. The tensor it represents must be *diagonal* with the diagonal elements having value one (see proposition D.1.1 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 $\mathcal{H} \in \mathbb{R}^{N \times N \times N} element - wise$.

This is straightforward to model with a CP decomposition: $\mathcal{H} = [\mathbf{I}, \mathbf{I}, \mathbf{I}]_{CP}$ 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 element-wise 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.

3.4 Learning decompositions by gradient descent

We now present some experimental results, with two aims in mind. Firstly, we compare the above CP decomposition to another method for decomposing tensors: the *tensor-train* (TT). This decomposition has some promising properties, however, it is ultimately out-performed

by the CP decomposition so we defer description and theoretical discussion of it to to appendix A.

Secondly we wish to verify the expressive power of the decomposed tensor. We verify theoretical results by showing a tensor layer is capable of solving the exclusive-or problem without hidden units. We then apply it to a real world task to illustrate the benefits of the representation.

3.4.1 Gradients

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 [59], 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.

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

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

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

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

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

$$\frac{\partial z_j}{\partial B_{lm}} = \sum_i^{n_1} \sum_k^{n_3} A_{li} C_{lk} x_i y_k. \tag{3.7}$$

Curiously, the j and m indices do not appear in the right hand side of eq. (3.7). 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 [89] 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.

3.4.2 Random Bilinear Products

Goal

The aim for this experiment is to compare the two decompositions. There are two scenarios to investigate: learning a decomposed tensor when the tensor has the same structure as the data and learning a decomposed tensor when it can only approximate the underlying structure.

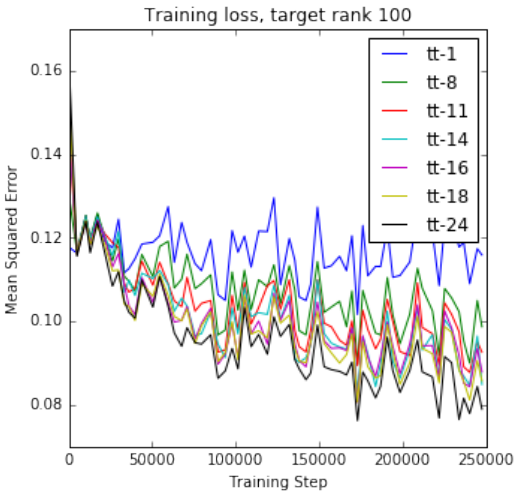
Experiment Details

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.

We use a batch size of 32 and 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. We use the same random seed for each trial, which explains the similar patterns in the TT-decompositions loss curves.

Results

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

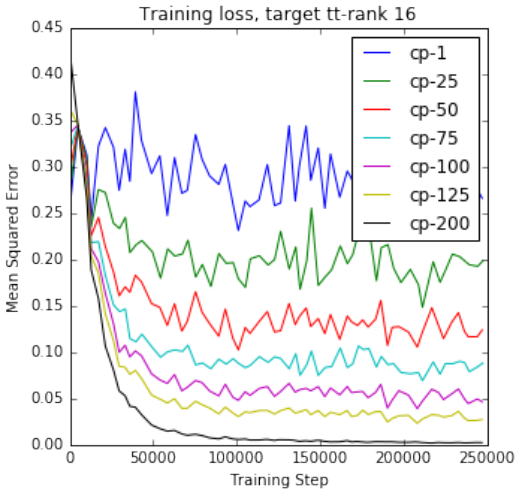


(b) Attempting to approximate by directly learning a TT-decomposition of various ranks

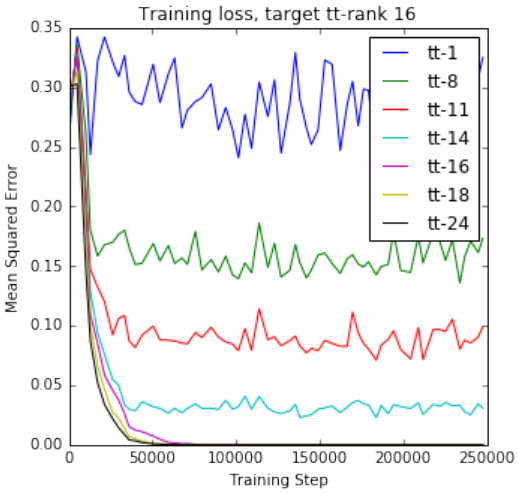
Figure 3.4: Results for learning direct approximations of a CP-rank 100 tensor.

The results of attempting to approximate a (CP) rank 100 tensor with dimensions $100 \times 100 \times 100$ (which has 30,000 independent coefficients) are shown in figure 3.4. The CP-decomposition is able to perfectly represent it when the rank is high enough, but usefully it is able to converge on reasonable approximations when it can not represent the full tensor. The TT-decomposition struggled, both in stability during training and final error. While we might expect the CP-decomposition to better represent another CP-decomposition due to the shared structure, it is 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.

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 3.5. As should be



(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 3.5: Results for learning direct approximations of a tt-rank 16 tensor.

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 learns reasonable approximations even when it does not necessarily reflect the structure of the underlying problem. The TT-decomposition struggles in this situation, but when the problem is structured more favourably it takes greater advantage.

3.4.3 Learning to multiply

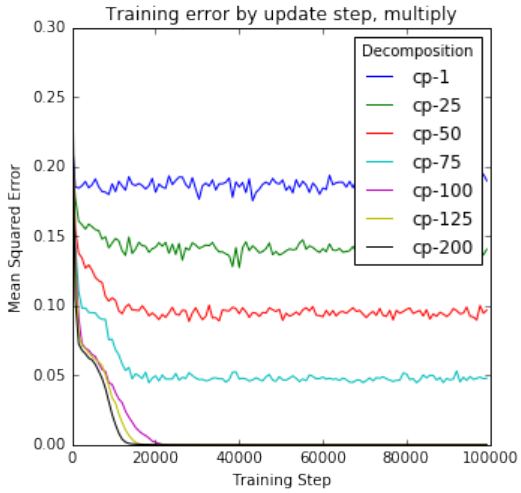
Goal

Learning bilinear functions provides a way to linearly approximate functions of two variables. A motivating example of such a function is element-wise multiplication, also known as the Hadamard product. In this section we investigate briefly the ability of the two decompositions to model such a product. We are particularly concerned with this as generalising this operation provides significant motivation for the current investigation.

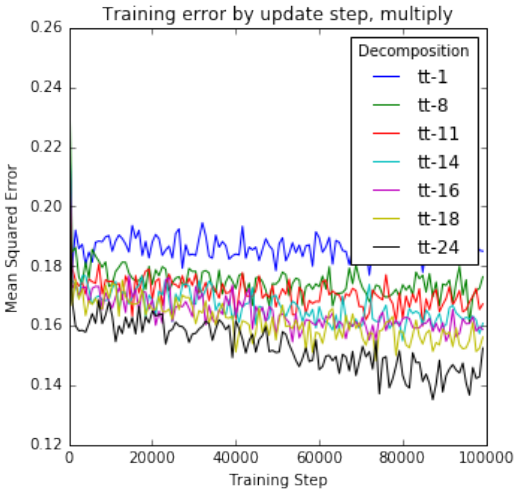
Experiment Details

Learning exact element-wise 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 element-wise 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.

This is performed identically to the experiments in section 3.4.2 except that that rather than generating a random tensor to provide the targets for training, we simply multiply element-wise the input vectors. We also simplify slightly by drawing the input vectors from $\{0, 1\}$ with equal probability (element-wise 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



(a) Learning curves for CP-decompositions learning element-wise multiplication



(b) Learning curves for TT-decompositions learning element-wise multiplication

Figure 3.6: Training error for various rank decompositions on the element-wise multiplication task.

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.

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.

Results

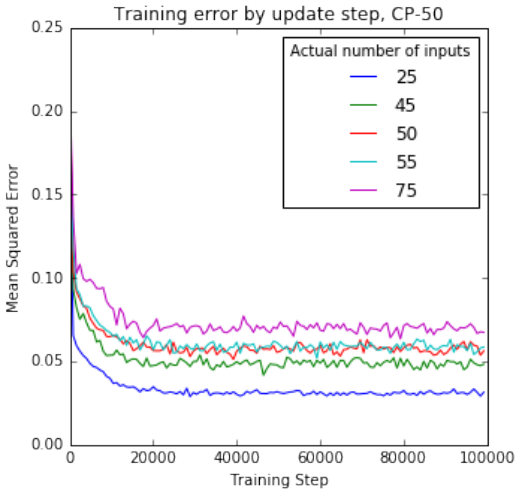
The results figure 3.6, affirm our expectations. The CP-decomposition is able to consistently reduce error as rank increases while the TT-decomposition appears to struggle. Permuting the indices did not make any difference to the relative performance, results are deferred to figure C.1 in the appendix.

When there is significant structure to the inputs, the TT decomposition is notably worse. One hypothesis is that the CP decomposition simply represents higher rank tensors with the same number of parameters and that this is is what allows to better represent more complex structures. This is shown in the other results – the rank 1 decompositions achieve similar performance regardless of the decomposition.

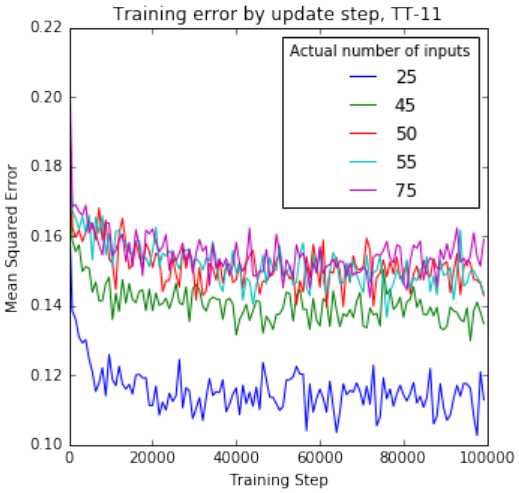
3.4.4 XOR

Goal

Theoretical results suggest that a single tensor layer should be able to solve the exclusive-or problem. This is known to be impossible with a single normal feed-forward layer, so if the tensor layer is successful in this task it would provide a useful demonstration of the difference in expressive power.



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



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

Figure 3.7: Training error for two decompositions learning element-wise multiplication with varying amounts of structure.

Experiment Details

True values were represented as 1 and false as 0. As there are only four data points, all were evaluated for each parameter update. All models were trained using standard gradient descent with a learning rate of 0.5.

Three models were evaluated, a perceptron, a multi-layer perceptron (MLP) and a single layer tensor. All models used a sigmoid nonlinearity on the output, the MLP had 4 hidden units also with sigmoids and the tensor layer used a rank 1 CP decomposition. Models were trained to minimise the cross-entropy, averaged across the data points with training continuing for 1000 epochs. Although not all MLPs had converged by this time, it was sufficient to see the difference between the architectures. To show that the differences are fundamental we repeat the experiments 50 times and observe consistent results.

Results

The baseline cross entropy is 0.6931, corresponding to outputting 0.5 for all inputs (or guessing uniformly at random). As expected, the perceptron converges rapidly to this point. Both the MLP and the tensor are able to solve the task, although it is interesting to note that the tensor does so far more rapidly and reliably than the MLP. This is not surprising in light of theorem 1 as this is precisely the kind of relationship we would expect the tensor product to represent naturally.

3.4.5 Separating Style and Content

Goal

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. [96] 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

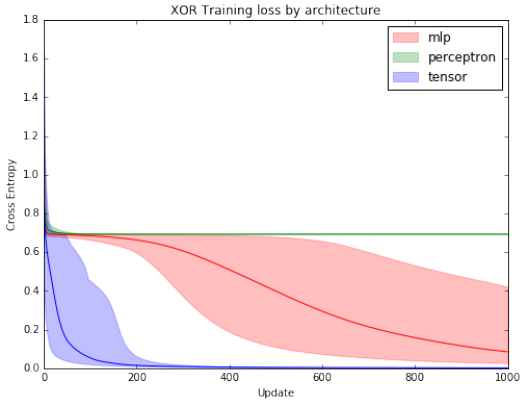


Figure 3.8: Exclusive-or results. Bold line is the mean of 50 runs, shading indicates the minimum and maximum across all runs.

follows. The aim is to verify that we can learn a tensor decomposition of an unknown latent tensor with unknown rank.

Experiment Details

Problem Formulation 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 [96] 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.

Training Details 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 [50], 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 $\hat{\mathbf{y}}_{sc} = \mathbf{a}_s^\top \mathcal{W} \mathbf{b}_c$ 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.

Results

Figure 3.9 provides a 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 overfitted rapidly, which is why a smaller model was used for these results.

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 3.10. These show that the captures much 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 3.11 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.

Overall, these results show the tensor was capable of learning a model for the data. This suggests that the aim of inserting such tensor products into more complex neural networks is feasible, reinforcing the promising theoretical results.

(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 3.9: Learned style and content representations.

(a) Actual images from the dataset.

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

Figure 3.10: Example of images generated from unseen style and content pairs, three letters from three different fonts.

(a) From the data

(b) Generated

Figure 3.11: A difficult example.

Chapter 4

Proposed Architectures

In this chapter we use the intuitions gathered from related works to propose novel classes of architectures employing tensor decompositions to implement multiplicative connections. There are two key principles: simplicity and modularity. We want to design networks that have no extraneous components, such that every element of the network has a clearly defined role. We first propose two novel architectures utilising a tensor product and then explain the decisions that let to their construction.

4.1 Architectures

4.1.1 Biases

Neural networks typically require biases to be able to express a full range of transformations. We would expect a tensor layer to be no different. A common way to conceptualise the addition of a bias for a given layer is to incorporate it into the weight matrix by adding an additional row and inserting a corresponding input which has its value fixed to one. With a bilinear product gives more than just the addition of a bias vector. We use $\tilde{}$ to represent altering the inputs and weights in this way: $\tilde{\mathbf{x}}$ is an input vector with an additional 1 appended and $\tilde{\mathbf{U}}$ is a matrix with a corresponding additional row.

Generalising this construction to a three-way tensor we have to take an $n_1 \times n_2 \times n_3$ tensor to a $n_1 + 1 \times n_2 \times n_3 + 1$. For a three-way tensor $\mathcal{W}$,

$$\tilde{\mathbf{x}}^\mathsf{T} \widetilde{\mathcal{W}} \tilde{\mathbf{y}} = \mathbf{x}^\mathsf{T} \mathcal{W} \mathbf{y} + \mathbf{U}\mathbf{y} + \mathbf{V}\mathbf{x} + \mathbf{b}. \tag{4.1}$$

This is very similar to the manner in which the states are computed in the vanilla RNN with the addition of the multiplicative tensor interactions.

Applying the same process when the tensor is represented in the CP decomposition does not have the same result. Instead it results in adding bias vectors to two of the internal matrix products. Let $\mathcal{W} = [A, B, C]_{CP}$, then appending ones to the inputs results in

$$\tilde{\mathbf{x}}^\mathsf{T} \widetilde{\mathcal{W}} \tilde{\mathbf{y}} = \mathbf{B}^\mathsf{T} \left((\mathbf{A}\mathbf{x} + \mathbf{b}_x) \odot (\mathbf{C}\mathbf{y} + \mathbf{b}_y) \right).$$

This is quite different to equation (4.1). If we wish to insert such a tensor product into a neural network we must choose whether to leave the biases in the decomposition or treat the bias matrices separately. The latter provides less control over the parameters, as there are now two matrices unaffected by the rank, but it might be helpful in that we could use a very low rank decomposition and still maintain a baseline behaviour.

4.1.2 Generalised Multiplicative RNN

The simplest architecture way to incorporate a decomposed tensor is to use it to generalise the Multiplicative RNN, Multiplicative Integration RNN [90, 103] and Vanilla RNN. To do this, we simply replace the various linear operations with a bilinear form with appropriate biases:

$$\mathbf{h}_t = \tau\left(\tilde{\mathbf{x}}_t^\mathsf{T} \tilde{\boldsymbol{\mathcal{W}}} \tilde{\mathbf{h}}_{t-1}\right)$$

Choosing to keep the biases elements separate from the decomposition gives a form which captures vanilla RNNs as well as several types of multiplicative RNNs:

$$\mathbf{h}_t = \tau\left(\mathbf{x}_t^\mathsf{T} \boldsymbol{\mathcal{W}} \mathbf{h}_{t-1} + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{V}\mathbf{x}_t + \mathbf{b}\right).$$

We term this the Generalised Multiplicative RNN (GMR). Unfortunately this network will still exhibit the same vanishing gradients as the vanilla RNN as the state updates still pass through a matrix (albeit one modulated by the new input).

4.1.3 Tensor Gate Unit

Incorporating ideas from gated RNNs such as the LSTM and GRU, we also propose the following architecture, which we call the Tensor Gate Unit (TGU):

$$\begin{aligned}
\mathbf{h}_t &= \mathbf{p}_t \odot \mathbf{h}_{t-1} + (\mathbf{1} - \mathbf{p}_t)\mathbf{z}_t \\
\mathbf{p}_t &= \sigma\left(\tilde{\mathbf{x}}_t^\mathsf{T} \widetilde{\boldsymbol{\mathcal{W}}} \tilde{\mathbf{h}}_{t-1}\right) \\
\mathbf{z}_t &= f(\mathbf{W}_{in}\mathbf{x}_t + \mathbf{b}_{in})
\end{aligned}$$

Where $f(\cdot)$ is either a linear rectifier or the identity function.

4.2 Motivation

The GMR is an elegant generalisation of a number of fairly simple architectures. However, it will still suffer from vanishing gradients and correspondingly we do not expect it to be capable of learning long time dependencies. To solve this, we have proposed the TGU which incorporates several key ideas from LSTMs and GRUs as well as a bilinear product. This section deals with the less clear aspects of the TGU's design, showing the theoretical and conceptual underpinning.

4.2.1 Solving Vanishing Gradients

The biggest issue with the GMR, and that style of RNNs in general is vanishing gradients. A simple approach to avoid this is to have the network compute a candidate vector of hidden states $\mathbf{z}_t$ and compute new hidden states as

$$\mathbf{h}_k = \mathbf{h}_{k-1} + \mathbf{z}_k. \tag{4.2}$$

Recalling equation (2.2), the problematic term was $\nabla_{\mathbf{h}_{k-1}}\mathbf{h}_k$, the gradient of a hidden state with respect to its immediate precursor. If the state is computed by equation (4.2):

$$\nabla_{\mathbf{h}_{k-1}}\mathbf{h}_k = \mathbf{I} + \nabla_{\mathbf{h}_{k-1}}\mathbf{z}_k.$$

Adding the identity matrix pushes the eigenvalues of the gradient up, which should go a long way to avoiding vanishing.

While the gradients may not vanish, they may still explode. Pascanu et al. show that a necessary condition for exploding gradients, when using the hyperbolic tangent activation, is that the largest eigenvalue of the recurrent weight matrix is greater than one [76]. With this additive update all we need is for the the entire $\nabla_{\mathbf{h}_{k-1}} \mathbf{z}_k$ to have at least one positive eigenvalue. This is highly likely, especially from a random initialisation.

Additive connections give us a way to fight vanishing gradients, but we need to temper them somehow to avoid exploding gradients. Fortunately, LSTMs and GRUs already present a class of successful techniques for this.

4.2.2 Gates

LSTMs and GRUs manage their additive connections by applying multiplicative gates. We will therefore investigate the solutions they present, with a view to choosing the best. We refer to the LSTM's gate as a *forget* gate. This type performs best in the presence of other gates [32, 47], which is explained by our theoretical analysis. The gate on the GRU is slightly different and has a number of advantageous properties which lead to its eventual selection. We term this the *convex* gate as it recursively computes convex sums.

Denote by p_t the gate signal at time t (which is a function of both the input and the current state and bounded in $[0, 1]$). We also use h_t to refer to the hidden state at time t, the object of interest, and z_t will refer to the candidate update computed by the rest of the network. In general these elements will be vectors, but in the following we assume scalars with no loss of generality as the vector forms contain only element-wise operations.

We define the forget gate's recurrence as

$$h_t = p_t h_{t-1} + z_t$$

for z_t the new candidate state which is typically a function of both h_{t-1} and the input at time t. This allows the network to wholly replace the state with a new value.

The convex gate makes the following modification:

$$h_t = p_t h_{t-1} + (1 - p_t) z_t.$$

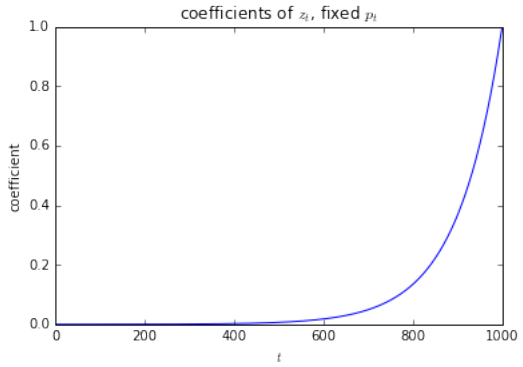
This allows the one gate signal to control the acceptance or rejection of new information.

In order to analyse the behaviour of these gates, we observe that each state h_t can be expanded as a weighted sum over all z_i for $i \leq t$. We can think of the states as being a sliding-window sum over candidates where the shape of the window is defined recursively by the p_i values.[1] With this view in mind, we note that we can think of the gate signals p_i as providing an *attention* mechanism which controls how the network attends to the information from past time steps.

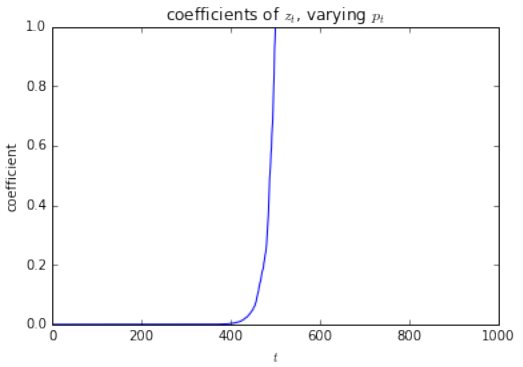
Forget Gate

The forget gate allows the network to completely replace a state. For this to happen the gate signal simply needs to be zero. If we define the initial state $h_0 = z_0$ we can write the first few

[1] There is in fact no sliding window – the sum is always over all values. However given the range of window shapes the gating mechanism is capable of producing it is reasonable to think about it as something close to a one-dimensional convolution.



(a) Coefficients of z_t resulting from a fixed, high value of all p_t under the forget gate scheme.



(b) Coefficients of z_t random p_t up to $t = 500$ and all p_t after 500 set to 1.

Figure 4.1: Different window shapes produced by the forget gate. As each coefficient at time t is the product of all gate values from t down to 1, the range of possible shapes is limited.

states explicitly:

$$\begin{aligned} h_1 &= p_1 z_0 + z_1 \\ h_2 &= p_2 h_1 + z_2 \\ &= p_2(p_1 z_0 + z_1) + z_2 \\ &= p_2 p_1 z_0 + p_2 z_1 + z_2. \end{aligned}$$

In general,

$$h_t = \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} p_j \right) \cdot z_i + z_t.$$

Each state is a weighted sum of the past candidates, with strictly non-increasing weights as we go back in time. The coefficient of each past candidate is the product over all gate signals back from the current time – as the gate signals are in $[0, 1]$ it will reduce very rapidly the further back we go. Figure 4.1 shows some possible resulting coefficients going back up to 1000 steps. This leads to a key conclusion about this form of gate.

Remembering requires cooperation. Consider the case where the network needs to remember an input at one time step and forget all future inputs. Achieving this with the forget gate can be done in two ways. Firstly, the gate values can be clamped to one and the candidates to zero. Clamping the gates to one produces an almost rectangular window; the state will be an evenly weighted sum of all candidates from the point the gate switches on. All subsequent candidates therefore must be zero. The second method of remembering a single candidate state is by reproducing the same candidate state at each time and setting the forget gate to zero.

Both of these require a significant degree of co-operation between the gate and the mechanism that produces candidate activations. Further, the production of candidate activations has to depend on the previous state. The LSTM achieves this with additional gates controlling the candidate production. We hypothesise that this hinders learning as these components have to concurrently learn complementary behaviours.

Gradients are gated. In order to update the weights of the network, we need the gradient of the hidden state at each time t with respect to the gate signal at an earlier time $i < t$:

$$\frac{\partial h_t}{\partial p_i} = \left(\prod_{k=i+1}^{t} \frac{\partial h_k}{\partial h_{k-1}} \right) \frac{\partial h_i}{\partial p_i} = \left(\prod_{k=i+1}^{t} p_k \right) h_{i-1}$$

by equation 4.2.2. This makes it clear that the gating does not just happen during the forward pass. Repeating the process with respect to the candidate activations, it is clear that the same argument applies.

$$\frac{\partial h_t}{\partial z_i} = \left(\prod_{k=i+1}^{t} \frac{\partial h_k}{\partial h_{k-1}} \right) = \left(\prod_{k=i+1}^{t} p_k \right) \cdot 1.$$

If the gates exhibit a strong decay, the network will be forced to update primarily on local information so it will struggle to learn long time dependencies. A common practical solution is to initialise the bias of the unit computing the forget to a high positive value so the gates tend to be higher [47].

Convex Gate

We now repeat the analysis with the convex gate. Let $p_0 = 0$ and $h_0 = z_0$ be some initial state.

$$\begin{aligned} h_0 &= z_0 \\ h_1 &= p_1 z_0 + (1 - p_1) z_1 \\ h_2 &= p_2 p_1 z_0 + p_2 (1 - p_1) z_1 + (1 - p_2) z_2 \end{aligned}$$

giving rise to the general form

$$h_t = \sum_{i=0}^{t} \left(\prod_{j=i+1}^{t} p_j \right) (1 - p_i) z_i$$

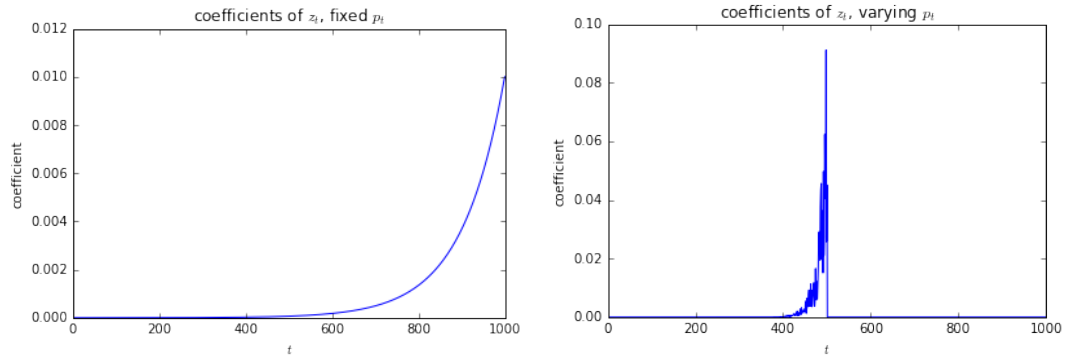
ensuring we define $\prod_{j=i+1}^{t} p_j = 1$ if $i = t$. This is very close to what we had above, but the extra $1 - p_i$ in the coefficients has significant impact. Firstly, we can prove the following statement which shows that the sum over states is a convex sum.

Proposition 4.2.1 (Conservation of attention)**.** *At any time $t > 1$,*

$$\sum_{i=0}^{t} \left(\prod_{j=i+1}^{t} p_j \right) (1 - p_i) = 1.$$

The coefficients of all previous candidates sum to one.

Proof. If we expand the brackets, we can produce a telescoping sum. To ensure it telescopes



(a) Coefficients of z_t resulting from a fixed, high value of all p_t under the convex gate scheme.

(b) Coefficients of z_t random p_t up to $t = 500$ and all p_t after 500 set to 1.

Figure 4.2: Different window shapes produced by the convex gate. A simple adjustment to the formula a much wider range of shapes.

appropriately, we first pull the first and last terms out of the summation.

$$
\begin{aligned}
&\sum_{i=0}^{t} \left(\prod_{j=i+1}^{t} p_j \right) (1 - p_i) \\
&= \prod_{j=1}^{t} p_j (1 - p_0) + (1 - p_t) + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} p_j \right) (1 - p_i) \\
&= \prod_{j=1}^{t} p_j (1 - p_0) + (1 - p_t) + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} p_j \right) - \left(\prod_{j=i}^{t} p_j \right) \\
&= \prod_{j=1}^{t} p_j (1 - p_0) + (1 - p_t) - \prod_{k=1}^{t} p_k + p_t \\
&= \prod_{j=1}^{t} p_j - \prod_{k=1}^{t} p_k + 1 - p_0 + p_t - p_t = 1
\end{aligned}
$$

□

This means that the scale of the states depends only on the scale of the candidate activations. We can make the following remarks about this form of gate.

Remembering is easy. Figure 4.2 illustrates two of the possible shapes this can take. It is clear that this gating scheme provides a different set of possible shapes to work with. Specifically, if the gate mechanism starts outputting values very close to one, the window applied to past states takes the form of a distinct spike over a small range of activations. This occurs because if p_i is very close to one, $1 - p_i$ is very close to zero, so state h_{i-1} is going to be carried over with only a very minor contribution from z_i. Figure 4.3 further emphasises the difference, making it clear that the convex gate is capable of representing a fundamentally different set of window shapes than the forget gate, despite their similar formulation.

This enhanced range of window shapes makes these gates appealing as they may be able to reduce interdependence between the gate mechanism and the candidate production mechanism. This may allow us to remove the candidate's dependence on the previous state which should solve many of the issues with the above gate.

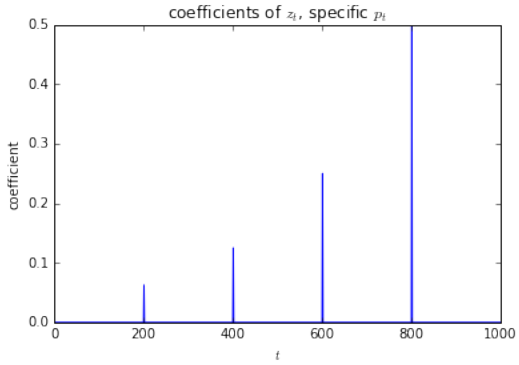


Figure 4.3: Setting specific p_i to 0.5 picks out a set of past candidates, albeit with an exponential decrease in their weighting.

Gradients are still gated. The gradient of the hidden state h_t with respect to a previous gate signal p_i has the form:

$$\frac{\partial h_t}{\partial p_i} = \left(\prod_{k=i+1}^{t} \frac{\partial h_k}{\partial h_{k-1}} \right) \frac{\partial h_i}{\partial p_i} = \left(\prod_{k=i+1}^{t} p_k \right) (h_{i-1} - z_i) \tag{4.3}$$

While the gradient of the state with respect to a prior candidate is

$$\frac{\partial h_t}{\partial z_i} = \left(\prod_{k=i+1}^{t} \frac{\partial h_k}{\partial h_{k-1}} \right) \frac{\partial h_i}{\partial z_i} = \left(\prod_{k=i+1}^{t} p_k \right) (1 - p_i). \tag{4.4}$$

The pattern here is fundamentally the same as the forget gate in that the gradient is simply the coefficient assigned to z_i in the weighted sum that produced h_t. Hence all discussion of the differences in the gates' forward behaviour apply equally during the backward pass.

Equation (4.3) is interesting – the gradient of the state with respect to the gate value depends not just on the preceding state but on the difference between the state and the proposed update. This suggests that during training what drives the updates is the possible changes that could be made to the state, rather than just the state values itself. Meanwhile, equation (4.4) shows that the proposition 4.2.1 also applies to the gradients. This suggests that as long as the scale of the candidate activations are under control there will be no problems with exploding gradients.

The downside of the system is that it may still struggle to learn long time dependencies early. It would be straightforward to initialise the gate so that it has a mean activation of 0.5, but this would correspond to an exponential decay of information. Unlike the forget gate simply increasing the mean activation (for example by initialising the bias to a high positive value) will not correspond to a flatter window.

Conclusions

The convex gate by itself seems more capable than the forget gate as it is able to represent more useful distributions over past candidate states while successfully employing a forget gate requires a more complex method of producing candidate states. Further, the convex gate avoids the issue of requiring two components to learn complementary behaviours by allowing a single component to carry out the job. We find this appealing as it enables simpler and more modular architectures.

4.2.3 Computing the gate

In order to decide whether incoming information should overwrite the current state values, the gate will need to see the current values and the new information. The binary nature of this task suggests the use of a tensor unit as discussed in chapter 3. Secondly if we want the full utility of the convex gate, it seems logical to imbue it with full expressive power – the bilinear tensor is a perfect fit.

It is also necessary to choose a non-linearity. We expect the gate to function mostly as a switch, accepting or rejecting new information. The typical approach is to ensure this smoothly with a sigmoid. Although sigmoids have undesirable properties when used in feed-forward networks [26] they remain the standard choice when this kind of gating is desired [73, 72] and we find no reason to deviate from this.

Using a tensor in this fashion provides an interesting possibility: the ability to write to the memory associatively. The gate is constantly computing weighted similarities between the input and the current state so it is possible for the gate to react if a certain pattern in the input matches a pattern stored in the state. This is a mode of behaviour very difficult to realise with existing gated architectures [18].

4.2.4 Candidate update

All that remains is to derive a candidate state update. In all of the architectures surveyed apart from the Strongly Typed variants [4] it is assumed that this needs to be a binary function of the current input and the previous hidden state. Intuitively this needs to be the case with the LSTM style forget gate, but as we have decided against that it is worth questioning this assumption.

Removing this step makes each role in the network clearly defined. As each input comes in, it is embedded into the state space. The gate then decides whether it is worth keeping. The candidate production mechanism only needs to behave as a feature detector, transforming the input into a representation that captures its essential structure. Conceptually this is very clean and modular and therefore favourable provided it does not strip the model of too much expressive power, which is a question to be determined empirically.

What remains is to determine how to compute the candidate from the input. We wish to model a unary mapping from the input to the candidate, so a logical way to proceed is with a standard feed-forward layer:

$$\mathbf{z}_t = f(\mathbf{W}_{in}\mathbf{x}_t + \mathbf{b}_{in}).$$

We experimented with a number of options for the non-linearity f and found the best performance was consistently achieved with either a linear rectifier or no non-linearity at all. This echoes trends in feed-forward networks away from saturating non-linearities towards unbounded, piecewise linear activation functions [27, 36]. It is interesting that for many tasks a completely linear candidate performed optimally – clearly the expressive power of the tensor combined with the non-linearity applied to the gate is sufficient to maintain the representative power of the network as a whole.

While a single feed-forward layer was sufficient in our experiments, there is scope to expand this depending on the complexity of the task. For example this could be a deeper feed-forward architecture, a convolutional neural network if the inputs have appropriate structure or even an identity map if the input to the network is already high-level features.

Chapter 5

Evaluation of architectures

In this chapter we present experimental results comparing the performance of the proposed architectures. The goal is two-fold: firstly we want to verify the feasibility and ease of use of the models, secondly we want to see if they achieved the goal of improving on existing architectures. The first goal is important as the architectures introduce a new hyper-parameter, the tensor rank, and we need to see if this is a significant hindrance to using the models in practice. The second is equally important, as no matter how appealing and conceptually grounded the architectures are, they are not useful if they do not perform in the real world.

5.1 Synthetic Tasks

The synthetic tasks are designed to be highly difficult. Most synthetic tasks for RNNs explored in the literature have focused on long time dependencies. While these provide a useful measure of the stability of a model's memory, we also wish to determine how capable a model is at using its memory to read and write arbitrary patterns. To do this we have had to propose a new synthetic task which identifies key weaknesses in the LSTM and GRU which are resolved by the TGU. As the tasks in this section are highly pathological, we do not report results for Vanilla RNNs or the GMR as they fail on all of tasks.

5.1.1 Addition

Task

This task is designed to test the networks' ability to store information for long time periods. It first featured in [43], although we use the slightly different formulation found more recently in [55]. The problem is a common benchmark for RNNs and has featured in a number of recent works ([3, 37, 5, 70] for example).

The inputs for this task are sequences of T pairs. One element of the input is sampled from a uniform distribution over the range $[0, 1]$ while the other is zero except for two locations where it is one. The location of the first one is always chosen to be earlier than $T/2$ while the second is after. The goal is to output at the last time step the sum of the two random values that were presented when the second input was one. Pseudocode for an algorithm to generate sequences is presented in the appendix, section F.1 and figure C.2 shows an example. This task becomes harder as the T increases because the length of time the numbers have to be remembered for increases.

Experiment Setup

We test a number of architectures on sequences with T of $250, 500$ and 750. As we are concerned purely with the ability of the networks to solve the task, we generate a new batch of sequences at each step. All models were allowed 8 hidden units and trained on batches of 8 data items to minimise the squared error between the output of the network at the final time step and the target value. The TGU was allowed a rank 4 tensor, giving it the smallest number of parameters of the gated architectures tested. Updates were applied using ADAM [50] with a base learning rate chosen from $\{0.1, 0.01, 0.001, 0.0001\}$ according to speed of convergence. For all runs, training continued for 1800 updates. While some authors have found they require significantly more updates for success with certain architectures (up to 10,000,000 in the case of [55]) we found most successful architectures converge within 500 on most sequences.

While previously reported results for this task stop at $T = 750$, the most successful architectures still converge rapidly to a solution at this stage. In order to more fully test the limits we increase T to 1,000, 5,000 and 10,000. The same parameters were used but only the architectures tested that were able to solve the task at $T = 750$ were applied to these much more challenging tasks. In order to reliably find solutions at length 10,000 we had to increase the number of hidden units to 32, the rank of the tensor decomposition was increased proportionally to 16.

The models tested were the LSTM, GRU, IRNN [55] and our TGU. We use a standard LSTM without peephole connections. For the TGU we separate the bias matrices and use a layer of ReLUs for the candidate production. All models use a linear output layer to reduce the hidden states back to a scalar.

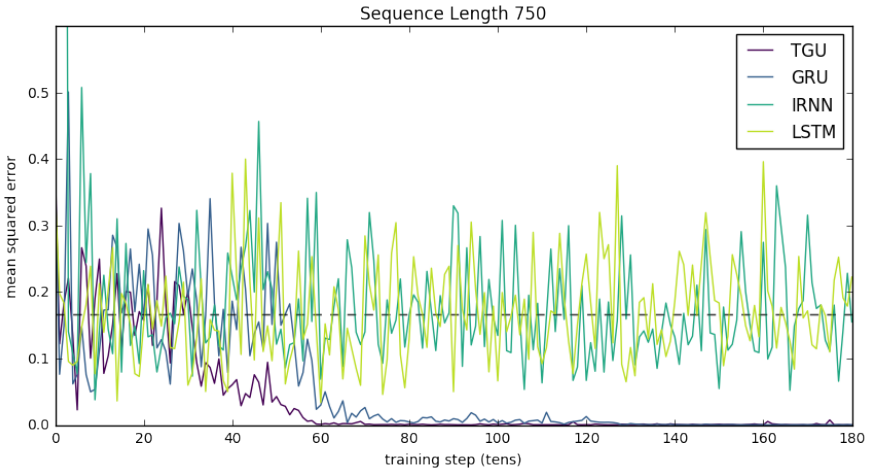
Results

The baseline result for this task is a mean squared error of 0.1767 which corresponds to predicting 1 for every input. Many architectures are unable to perform significantly better than this. Figure 5.1 shows that we were unable to find a solution with the IRNN or LSTM within the time allotted, while the GRU and TGU consistently did so very rapidly – in less than 1,000 iterations which is several orders of magnitude faster than previously published results [55, 3, 37].

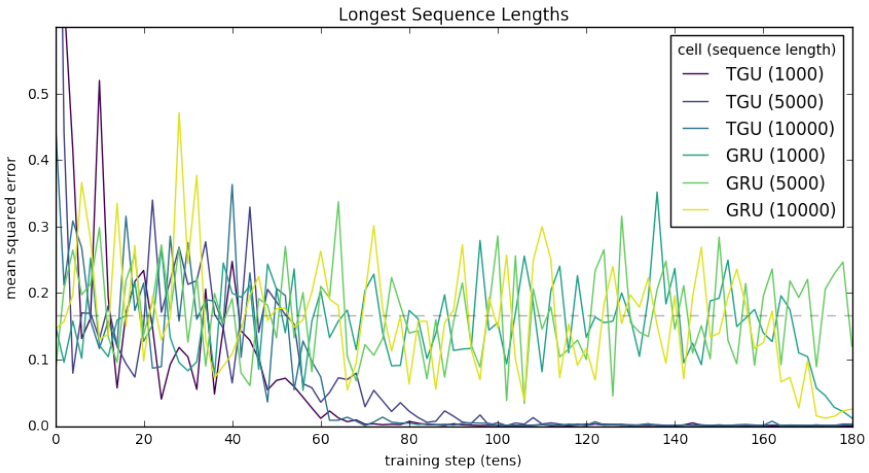
The figures also show that this task can be somewhat frustrating. Due to the online nature, the loss curves are very noisy, so it is almost impossible to determine whether a model is making progress. Further, in accordance with a number of others' results [55, 3] many architectures will sit on or around the baseline showing no signs of progress for thousands of training steps before suddenly and rapidly converging to a solution, such as the GRU on sequence length 1000 in figure 5.1b.

On sequences up 750 steps, there is little to separate the GRU and the TGU. As they share a gating mechanism, it is probable that this is a key factor in their success. As discussed in chapter 4 this convex gating mechanism is naturally capable of representing exactly the behaviour required to solve this task – remembering precisely a small subset of inputs.

We also found that the TGU performed much better on this task with a ReLU layer producing candidate updates, which is contrary to findings on several later tasks in which linear candidates were preferred. A possible reason for this is that it is very easy for the ReLU layer to output exactly 0. If the candidate activation mechanism is able to learn to simply output zeros when the appropriate input is zero, then if the gate is able to stay constant the task is essentially solved – there should be at least one hidden unit that will contain a value proportional to the correct answer. Therefore, we would expect all gated units with an



(a) Sequence length 750.



(b) Very long sequences.

Figure 5.1: Results for addition task

additive state update to perform excellently on this task, those with a rectified state update only more so.

These results reveal a key weakness in the LSTM. While others have shown it is capable of solving the task eventually [37], it was unable to do so in a comparable number of training iterations. This adds weight to the hypothesis that the co-operation between components that the LSTM requires to succeed is tricky to learn by gradient descent and it is preferable to avoid it if possible.

Although the convex gate is excellent at picking out individual time steps to remember, it still has the character of an exponential decay. However, we were unable to find a sequence too long for the TGU. Results for the TGU and GRU on long sequences are shown in figure 5.1b. The GRU begins to beat the baseline on most of the sequences inside 2,000 updates. What is remarkable is that the TGU seems almost completely unaffected by the extreme length of the sequences, still achieving a solution inside 1,000 steps.

For the sequences with length 10,000 the expected time dependency (the number of steps between the two inputs the network has to remember) is 5,000. The network then has to preserve this value in its memory for an average of 2,500 steps. If we consider taking the RNN and unrolling it across all 10,000 time steps, it is equivalent to a feed-forward network

of extraordinary depth and shared weights at each layer.[1] In order to train it successfully we must back-propagate gradients through 7,500 steps on average. While the shared weights undoubtedly assist greatly it is worth observing that this is still more than 5 times the depth of the deepest feed-forward networks reported [45].

5.1.2 Variable Binding

This task is designed to test the architectures' ability to store arbitrary patterns in memory and associate them with a label. We term this "variable binding" as to solve the task the network needs to be capable of associating an arbitrary value with a specific label. This task is a natural sub-problem of many large scale problem domains to which RNNs have already been successfully applied. An example is translation – in an end-to-end translation system, the network must perform tasks such as detecting subjects and objects of sentences, store them appropriately and reproduce them at an arbitrary time.

Curiously, we could not find a satisfactory task in the literature which exercised this ability. Close alternatives include the copy task in [43] and the variants considered in [29], but these are concerned with recalling the order in which symbols arrive. The "variable assignment" task used to evaluate the Associative LSTM [18] has a similar aim in mind but assesses recall of temporal patterns.

Task

We propose the following synthetic task to test the ability of the network to store and retrieve arbitrary patterns: inputs and targets are sequences of length T. Inputs at each time step are binary vectors of size $D + N$. The first D elements of the input are the "pattern bits" while the remaining N are the "label bits". N different binary patterns are presented to the network at randomly chosen time steps in the first half of the sequence, the pattern bits are zero at all other times. Immediately before a pattern is presented, one of the label bits is set. The label bit is then held until a randomly chosen time step where it is cleared. At this point, the network must output the corresponding pattern. Label bits are never reused. The target sequence to match consists of vectors of size D, containing the appropriate pattern at the moment each of the label bits switches off and zeros elsewhere. Section C.2.2 in the appendix includes some examples of this. while algorithm 2 in the appendix demonstrates how to produce appropriate data.

This task involves a number of interesting concepts. A correct solution will need to learn a transformation from input patterns to hidden states and then learn the inverse transform to produce the correct output. The requirement of invertibility suggests a strong reason for the success of the TGU with linear candidate updates, as the correct output can be reconstructed by applying the inverse affine transformation. Storing and recalling multiple distinct items also requires the ability to be selective about updating the hidden state, which suggests gated architectures should perform well.

Experiment Setup

For this task we found the same hyper parameters to be the best for all models (that is to say, grid search was unable to find any with demonstrably better performance). Models were trained with a batch size of 32 using ADAM with a base learning rate of 0.01. We ceased

[1] As the error is only computed on the final output for this task, the analogy is very close – the RNN is precisely a deep feed-forward network with shared weights and additional inputs at each layer.

training after 5000 steps – although some models were only just beginning to learn by this time, it is sufficient to see the difference in how easily they are able to arrive at a solution.

As the targets are binary vectors we use a sigmoid output layer and train to minimise the sum of the cross entropy at each output at each time step. For all results reported we use eight bit patterns and test on sequences of length 100, 200 and 500 with one, two and three patterns leading to nine different experiments.

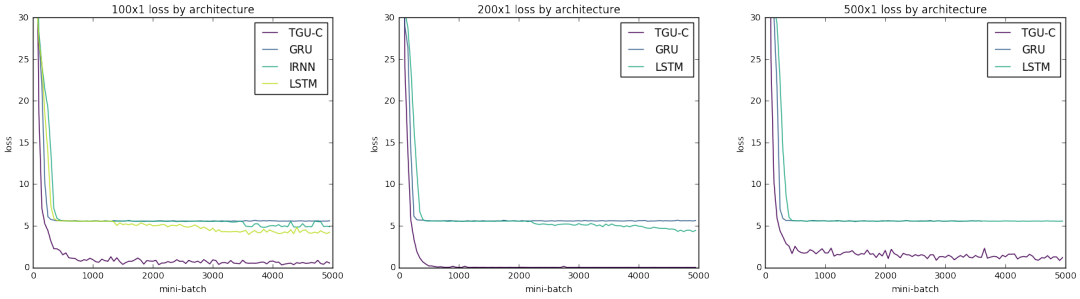
The baseline behaviour which all architectures achieved rapidly is to wait until any of the label bits turns off and output a pattern with mean 0.5. Guessing 0.5 (or randomly with probability 0.5) achieves a cross entropy of $-\log 0.5 = 0.69315$, so this will achieve a total loss on sequences with N eight bit patterns of $-8N \log 0.5$. Examples of a networks exhibiting this behaviour can be found in figure C.6 in the appendix.

As the aim is to assess the ability of the networks to solve the task, we train in an online fashion. This means that rather than generate a training and test set, we generate new sequences as required and only report the training error as in this case it is equivalent to generalisation performance.

We set the number of hidden units to ten times the number patterns to remember for each architecture. This leads to considerable difference in the number of parameters, but this task is primarily concerned with the ability of an RNN to use its memory sensibly, so it is fairest to ensure they all have the same amount of memory. As the patterns are all eight bits, have ten hidden units per pattern ensures it is possible for the networks to store the inputs directly and still have a couple of units to track the state of the label bits. As the patterns are binary, it should be straightforward to learn a compressed representation, so these networks are very much overspecified.

Results

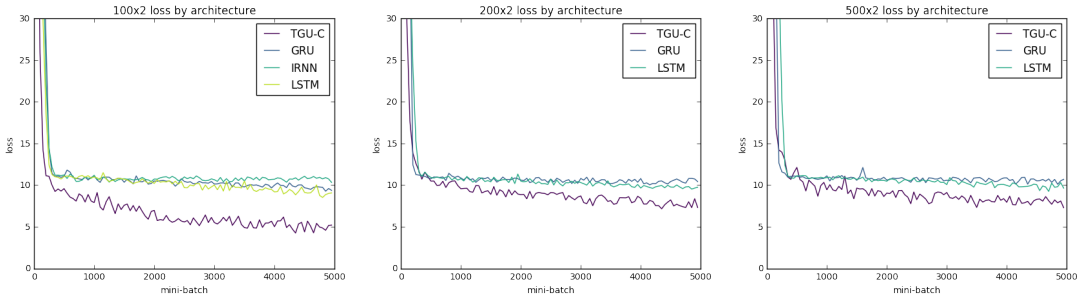
We report results for the LSTM, GRU, IRNN and TGU. For the TGU we incorporate the biases into the decomposition (we denote this TGU-C) and use linear activations which was found to give the most consistent results. We also tested the Vanilla RNN, which was unable to beat the baseline in any tests and several TGU variants – for clarity we only present the results of architectures which showed at least some promise. We were unable to reliably train IRNNs on sequences longer than 100, despite heavy gradient clipping the gradients consistently exploded. Therefore we only report IRNN results on the tasks with length 100. We show the median of five runs as there were occasional outliers.



(a) Sequence length 100. (b) Sequence length 200. (c) Sequence length 500.

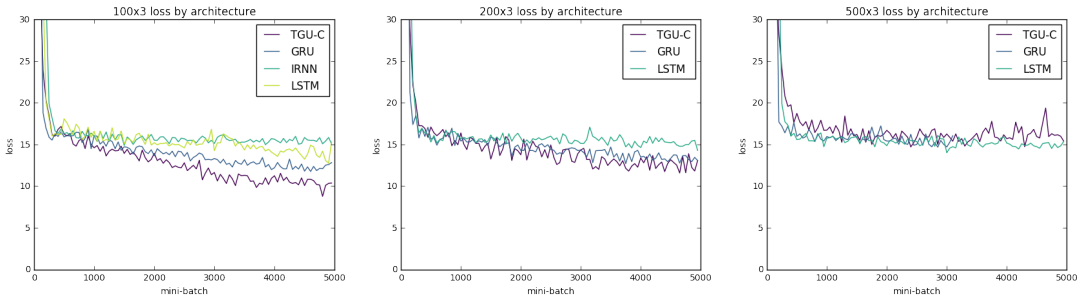
Figure 5.2: Variable binding results for sequences containing 1 pattern to remember. The baseline loss for this task is 5.5452.

With only a single pattern to remember, the TGU dramatically outperformed the other architectures. The LSTM was the most consistent of the other architectures tested, although



(a) Sequence length 100. (b) Sequence length 200. (c) Sequence length 500.

Figure 5.3: Variable binding results for sequences containing 2 patterns to remember. The baseline loss for this task is 11.0904.



(a) Sequence length 100. (b) Sequence length 200. (c) Sequence length 500.

Figure 5.4: Variable binding results for sequences containing 3 patterns to remember. The baseline loss for this task is 16.6355.

in every test by the time it had begun to escape from the baseline the TGU had already converged. The GRU was able to make surprisingly little progress on this task.

Increasing the number of items to remember, even just to two, makes the task considerably harder. This highlights an issue with the manner in which RNNs interact with their memory – they are forced to attend to it all at once. If we consider a TGU-style RNN in which the candidate state does not depend on the previous hidden states, it must learn a redundant mapping to embed the input into the state-space. This is because it can not rely on the label bits to determine which of the two inputs it is processing (this would require access to the hidden states to determine which has switched recently). If we assume different sets of hidden states are used to represent inputs with different associated labels, then the candidate update must propose to update both of them with the new input and it is up to the gate to select which update to apply. This seems like a perfectly reasonable solution, although learning such a redundant mapping is going to be considerably more challenging than simply learning the single reversible transformation required to solve the task when only a single pattern is to be remembered.

With two patterns to remember the TGU still makes the most progress below the baseline in the time allotted. On the longer sequences progress is limited and none of the architectures are close to converging although the TGU is the only architecture to make notable progress.

When the number of patterns is increased to three all the architectures struggle. With a sequence length of 100, there is some separation – the TGU makes the most progress although the GRU and LSTM also begin to progress beyond the baseline. On the longer sequences, there is less of a clear result and when the sequence length is up to 500, no models were capable of surpassing the baseline even when allowed to train for significantly longer than

reported in figure 5.4.

These results affirm that the TGU is competitive with the state of the art approaches. They also suggest that the fundamental change – focusing the expressive power in the gate rather than the candidate update – makes a fundamental difference to the manner in which the network attends to its state. Further, limiting the space of potential solutions by removing the dependence of the candidate update on the hidden state seems to help make solutions easier to find.

5.1.3 Sequential MNIST

A pair of recently proposed tasks which have gathered significant popularity lately are based around attempting to classify the MNIST handwritten image dataset [56]. To turn two dimensional images into a sequence task they are flattened in one of two ways: either in scan-line order beginning from the top-left going one row at a time or simply by taking a random permutation [55].

This gives two tasks in which the input is a one dimensional time series to be classified. The MNIST digits are 28 by 28 digits, so the input sequences are 784 steps long. As the majority of the pixels are black with the activity focused in the central region, the majority of the information in the scan-line version is focused in the middle. Contrastingly, taking a random permutation of the indices is highly likely to induce very long time dependencies, in addition to removing any temporal correlations in the sequence by destroying the morphology. Consequently the permuted version is significantly more challenging. We group these with the synthetic problems because they are somewhat arbitrary and unnatural.

Although this has been widely used recently to benchmark architectural advances in RNNs (see, for example [106, 5, 25, 70, 17] solely from 2016), we argue that it does not necessarily provide a useful test of a model's capabilities. We show this by achieving excellent performance on the more challenging permuted task with a models which fail miserably in real world applications.

Proposed Architecture

We propose an architecture specifically for this task which is a variant on GMR proposed earlier, modified to allow for direct accumulation. Further, we enforce this accumulative behaviour by applying a linear rectifier to the state update. The hidden states are computed by:

$$\mathbf{h}_t = \mathbf{h}_{t-1} + \rho\left(\mathbf{x}_t^\top \boldsymbol{\mathcal{W}} \mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}\right).$$

This architecture should fail – the gradients are almost guaranteed to explode during back-propagation by the argument in section 4.2.1, it can never remove information from its states (the state updates must be non-negative). The only way it can make a context-dependent decision to ignore certain inputs is if the tensor product is able to outweigh the rest of the calculation before the non-linearity and force it to be negative. As we implement the tensor product using a tensor in the CP decomposition, we refer to this architecture as the "CP+".

A model which seems slightly less absurd allows *subtractive* forgetting:

$$\begin{aligned}
\mathbf{a}_t &= \rho\left(\tilde{\mathbf{x}}_t^\top \widetilde{\boldsymbol{\mathcal{W}}}_a \tilde{\mathbf{y}}\right) \\
\mathbf{b}_t &= \rho\left(\tilde{\mathbf{x}}_t^\top \widetilde{\boldsymbol{\mathcal{W}}}_b \tilde{\mathbf{y}}\right) \\
\mathbf{h}_t &= \mathbf{h}_{t-1} + \mathbf{a}_t - \mathbf{b}_t.
\end{aligned}$$

Architecture	Test Accuracy (%)
TGU	88.2
CP+	84.7
CP-Δ	91.5
IRNN (ours)	84.0
LSTM (ours)	86.3
Vanilla (ours)	79.3
*i*RNN [55]	82.0
*u*RNN [3]	91.4
*s*TANH-RNN [106]	94.0
BN-LSTM [17]	**95.4**

Table 5.1: Test accuracy for permuted sequential MNIST.

We denote this architecture "CP-Δ" and consider only the variant in which the bias matrices are incorporated into the tensor decomposition to keep the number of parameters comparable. It appears more sensible than the CP+ as it at least allows for the possibility of a negative state update. However, this model will suffer from the same gradient issues as the CP+, although they might be expected to be less pronounced. In practice, we find this is not the case and the model requires at least as much guidance as the CP+.

To enable training of this model we re-parameterise the weight matrices with a form of *weight normalisation* [82]. Typical weight normalisation learns unconstrained weights but applies a differentiable normalisation transformation before use. Salimans and Kingma normalise each row of the weight matrices to have an $l2$ norm of 1, we achieved best results by dividing each matrix by its Frobenius norm.[2] Back-propagation proceeds as usual through the transformation.

Experiment Setup

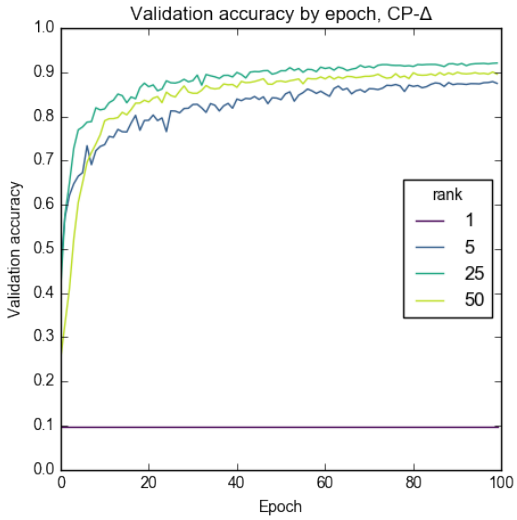
We are primarily concerned with the permuted task as all models tested apart from the Vanilla RNN and GMR achieved 98% ($\pm$1.0) on the simpler task. Learning rate and rank were found using grid-search. We train using ADAM with a batch size of 100. For nearly all models, severe gradient clipping [77] is required to train successfully, the best threshold was also found using grid search. Following [55] all models tested have a single hidden layer of 100 units. We use a grid search to find the best rank for the tensor models.

Results

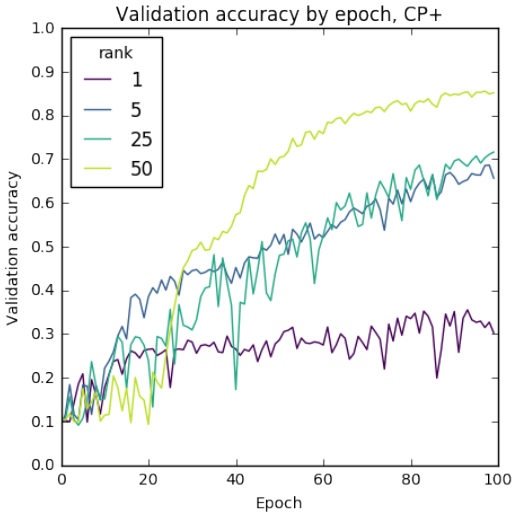
Test accuracy for the best models is reported in table 5.1 which also includes some recent results, all of which were state-of-the-art at the time of their publication. Our models sit in the middle of these results. This is remarkable as the best performing architecture, the CP-Δ has fundamental design flaws which cause it to fail on any other dataset we attempted to train it on.

Figure 5.5 shows how the performance scales with rank of the tensor decomposition. As the CP-Δ incorporates the bias matrices into the decomposition, reducing the rank should greatly affect performance. Indeed, with a rank 1 decomposition the model is incapable of learning anything. What is more curious is that the best performing model is not the highest

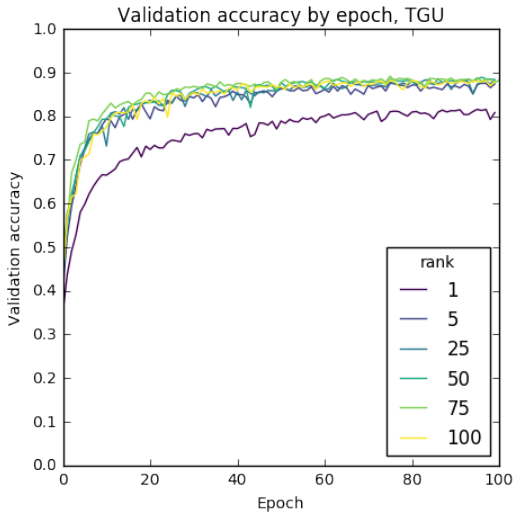
[2] For a $n \times m$ matrix $\mathbf{U}$, the Frobenius norm is $||\mathbf{U}||_F = \sqrt{\sum_i^n \sum_j^m U_{ij}^2} = ||\operatorname{vec}(\mathbf{U})||_2$.



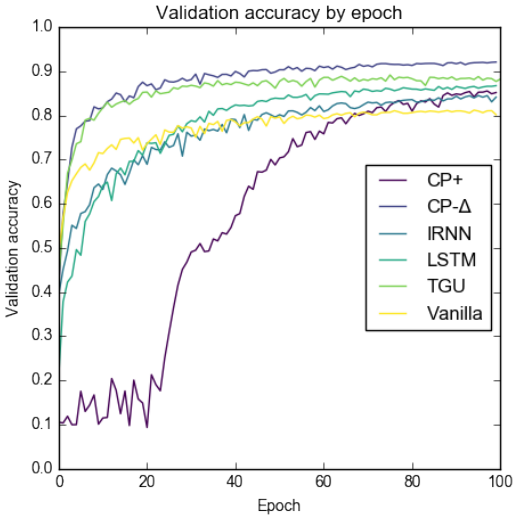
(a) CP-Δ by rank



(b) CP+ by rank



(c) TGU by rank



(d) All architectures

Figure 5.5: Performance (classification accuracy, %) on the validation set of permuted MNIST by rank of tensor decomposition.

rank. None of the models had overfitting problems, we can only hypothesise that reducing the rank might restrict the space of feasible solutions helping gradient descent.

The absurdity of the CP+ unit is clear from figure 5.5b – it trains in an incredibly volatile fashion. That the rank 1 model learns at all is remarkable. The tensor product in this unit is severely restricted, the vast majority of its learning power resides in a feed-forward connection. This goes some way to affirming our suspicions about this task – although it exhibits long time dependencies, it is almost enough to simply accumulate over all inputs.

5.2 Real-world Data

We now move on to more realistic datasets. All of the following tasks are *next-step prediction* tasks – the goal of the network is to estimate the probability of the next item in the sequence

given the sequence so far:

$$p(\mathbf{x}_{t+1}|\mathbf{x}_1, \ldots, \mathbf{x}_t).$$

The output layer of the RNN then has the task of converting the hidden states into probabilities and so requires an appropriate non-linearity. Training proceeds by maximising the log-likelihood of the data producing the correct distribution.

This means that the target sequence is the input sequence shifted across by one. If we prepend a special *start-of-sequence* symbol to the data sequence and append an *end-of-sequence* symbol, then the inputs to the network are simply all but the *end-of-sequence* and the outputs are all but the *start-of-sequence*. The end result is a statistical model of the sequence, approximating the conditional distribution described above.

In all cases the data is split into three sets: training, test and validation. The validation set is used to monitor generalisation performance and to stop training when the model begins to overfit. We do this by checking the loss on the validation set after each epoch. If performance has improved, we save a checkpoint containing the model parameters. After having trained for a fixed number of epochs the best saved model is loaded and evaluated on the test set.

The aim of this section is not to achieve state-of-the-art performance on large datasets. Rather we seek to ascertain whether the theoretical reasoning and intuitions developed in the preceding chapters hold in a more realistic setting. These datasets are less pathological than those previously evaluated on, so it is interesting to see if the benefits of our architectures hold in such a setting.

5.2.1 Polyphonic Music

This task, introduced by [7] consists of modelling four datasets of polyphonic music from a score. This is potentially a challenging task – not only do the melodies and chord progression exhibit long time dependencies, the networks must learn the rules of harmony to produce appropriately consonant arrangements. The likelihood of a given note at a given time step is affected by both simultaneous information (which other notes are one at the same time) and long term information (such as which key the piece is in). Modelling accurately both these dependencies is challenging but reflects well the difficulties inherent int much naturally generated data.

Task

The four datasets are:

Pianomidi is a collection of classical piano scores originally sourced from `http://piano-midi.de`. We split the data according to [79]. This data uses the full range of the piano, there are 88 distinct notes present although the polyphony is limited as the pieces must all be played by a human.

Nottingham is a set of folk tunes converted to MIDI by [7],[3] we use their split of the data. These tunes have the simplest structure and are generally quite short although they span a range of 63 distinct notes.

Muse comes originally from `http://www.musedata.org` and contains both orchestral and piano classical music. This dataset has a range of 84 distinct notes and due to the orchestral nature of many of the scores the polyphony can be quite high. Again we split the data according to [7].

[3] The originals are available at: `http://ifdo.ca/~seymour/nottingham/nottingham.html`

JSB consists of 382 chorales harmonised by J. S. Bach. This dataset has some of the lowest polyphony, as it is written in four part harmony. This dataset also has the smallest range of 54 notes. We use the split of the data given in [2].

The goal of the task is to learn the conditional distribution over notes at the current time, given the preceding sequence of notes.

Experiments

The data is converted from MIDI files (which contain note on/off events) to an appropriate format by sampling the notes present at regular time intervals. This gives binary vectors in which each element indicates with a one or a zero whether a particular note is active or not.[4] For each dataset, we fix the size of these vectors to the range of notes present in the dataset.

Previous work is slightly inconsistent with the size of these vectors. Of those that report it, [7] fix the size of the vectors for all datasets to the range of 88 notes available on a piano while [14] use much larger vectors although it is not clear why. We fix the size of the inputs to the range of the notes present in the data. As the data is written for specific instruments it only makes sense to train the model to attend to an appropriate range for those instruments. This means results can not be directly compared although they are very similar, with nearly all of our RNNs surpassing [14] and achieving performance similar to the RNNs evaluated in [7]. We would expect our RNNs to report slightly higher negative log-likelihoods as we use input/output sizes with no redundancy – there are no outputs the network can learn to switch off constantly bringing down the average loss. Further, the more unexpected elements of the results such as the performance of the Vanilla RNN on Nottingham are present in others' results.

All networks used a single hidden layer. The number of hidden nodes and the rank of the tensor decomposition was chosen so that each network would have as close to 20,000 parameters when applied to the JSB dataset as possible. This was done so that the comparison is not skewed by models such as the LSTM which have a large number of parameters per hidden unit. The rank was set as either 1 or a multiple of the number of hidden states, chosen from $\{1/2, 1, 2\}$. Table C.1 in the appendix shows the number of hidden units and the rank of the tensor decomposition for the models evaluated. The networks used a sigmoid output layer to appropriately model the fact that the multiple output units may need to be on at once.

A preliminary grid search was undertaken on a subset of the architectures to determine sensible ranges for the final experiments. Models reported were trained using ADAM with a grid search over the following hyper-parameters: base learning rate from $\{0.01, 0.001\}$, maximum length of back-propagation through time chosen from $\{75, 100\}$ steps and rank as discussed above. The batch size was set to 8. For nearly all architectures best performance was achieved with the lower learning rate while the best maximum length was task-dependent.

Results

Results for the four datasets are found in table 5.2. We report the average negative log-likelihood the best early-stopped models assign the data vectors over the training, test and validation sets.

In general the tensor units performed better than their counterparts except on Nottingham, the simplest of the datasets. This dataset consists of folk songs which are often very formulaic in their harmony. Correspondingly, simple models seem to be able to perform very well.

[4] Appropriately pre-processed data can be downloaded from `http://www-etud.iro.umontreal.ca/~boulanni/icml2012` courtesy of Boulanger-Lewandowski et al. [7].

	Nottingham		JSB		Pianomidi		Muse	
	Train	*Test*	*Train*	*Test*	*Train*	*Test*	*Train*	*Test*
Vanilla	**3.3278**	**3.8871**	8.0284	8.6381	7.3874	7.8093	6.8954	7.3619
LSTM	3.5826	4.0124	8.2058	8.6633	7.3730	7.7542	7.0777	7.4159
GRU	3.5876	4.0157	8.1689	8.6027	7.3039	7.7652	7.0001	7.3824
GMR	3.3940	3.9055	8.0526	8.6084	7.3598	7.7676	**6.8864**	**7.3089**
GMR-C	3.7902	4.1350	8.0093	8.5369	7.4244	7.8615	6.9135	7.4296
TGU	3.9652	4.2140	**7.9357**	8.6060	**7.2132**	7.7161	7.4524	7.6879
TGU-C	4.0035	4.2531	7.9731	**8.5307**	7.2748	**7.7003**	7.4551	7.6719

Table 5.2: Results on polyphonic music datasets. Numbers are average negative log-likelihood, lower is better. "-C" appended to the tensor units indicates the bias matrices are folded into the decomposition, otherwise they are separate.

It also seems to be the case that there is something about the structure of the data which suits the Vanilla RNN – the best GMR (which was the closest to the Vanilla RNN out of all architectures) had a rank 1 tensor decomposition meaning that it was functionally nearly identical to a Vanilla RNN.

In fact, all of the best GMR models used a rank 1 tensor decomposition. This is interesting as it suggests the extra complication of the tensor product only hindered learning. Alternatively, it may be that it is beneficial to have more hidden units even if that comes at the cost of decreased representational power. This effect was only notable in the GMR.

Interestingly the TGU-C models generally outperformed their counterpart TGUs on the test data but not on the training data. This suggests that having the ability to constrain the representative power of the gate can provide a useful regularising effect.

5.2.2 PTB

The next experiment tests whether reducing the number of parameters in the model by lowering the rank of the tensor decomposition can have a regularising effect. For this we use the Penn Treebank [60] and attempt to model it by word using the next-step prediction framework outlined earlier. This is a relatively small dataset and overfitting becomes a serious challenge. Due to this the dataset has been popular for testing regularisation approaches for recurrent neural networks [105, 24].

We are concerned with how the tensor that computes the gate values in the TGU affects the ability of the network to make use of its memory. To test this we fix the number of hidden units and vary the rank of the tensor.

Experiment Setup

The data is pre-processed following [105] by replacing all but the most common 10,000 words with an 'unknown' symbol. We train using ADAM with a batch size of 20. Base learning rate for each model was found via grid search. All networks trained had 128 hidden units – for this experiment we do not attempt to keep the number of trainable parameters equal between models. We test four variations of the TGU: with and without combined biases with with linear or rectified linear candidate updates. For comparison we also train an LSTM, GRU and Vanilla RNN. The GMR performed unremarkably on this data, so we omit it from our results.

Architecture (rank)	*Train*	*Valid*	*Test*
TGU (64)	82.86	139.54	131.42
TGU-C (64)	92.73	147.90	139.15
lin-TGU (64)	91.81	127.80	**121.93**
lin-TGU-C (128)	99.74	133.77	127.56
LSTM	**78.74**	144.75	134.79
GRU	79.89	**133.55**	127.56
Vanilla	85.10	164.81	156.63

Table 5.3: Results of best early-stopped models on the Penn Treebank

Results

For this task we report the average word-level perplexity over the test set. The perplexity is the exponentiation of the cross entropy between the predicted distribution and the target distribution (lower is better).

Overall results are presented in table 5.3. The best performance across all models was achieved by the TGU with linear candidate updates and separate biases. Three out of four of the TGUs surpassed the LSTM with linear candidate updates having a clear advantage.

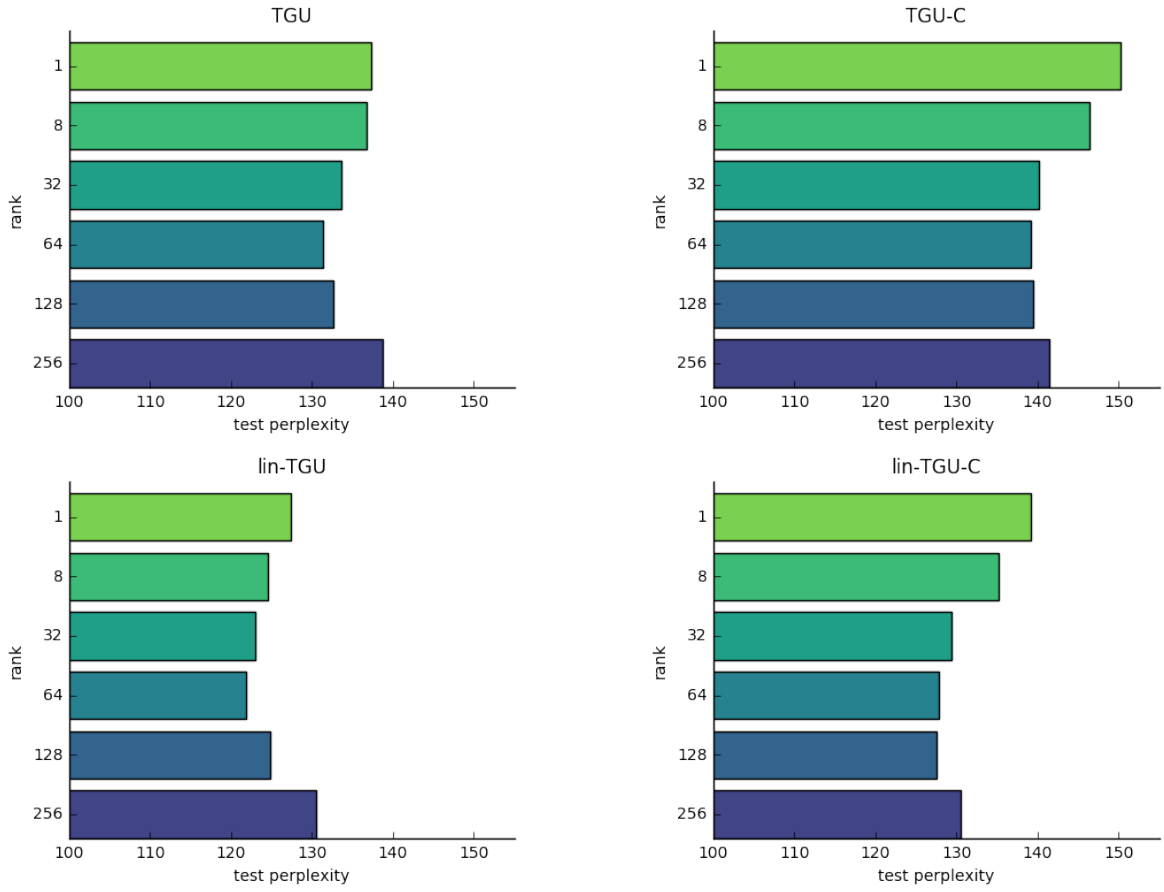


Figure 5.6: Test perplexity of best early stopped models for each type of TGU for a range of ranks. Networks had 128 hidden units.

Figure 5.6 shows results of the TGUs organised by rank. For all models there is a clear benefit to constraining the tensor to half the number of hidden states. When the biases are

combined the difference between rank 64 and 128 is less clear cut, which is to be expected as the expressive power of the tensor unit is much more constrained by the rank. That none of the models with separate biases improve performance with a rank below 64 indicates that the tensor is performing a valuable role as robbing it of power harms performance.

5.2.3 War and Peace

We perform one final experiment to test whether the regularising effect observed so far are due to limiting the expressive power of the tensor or simply reducing the parameters. We do this by adjusting the number of hidden units so that all models have the same number of trainable parameters regardless of rank.

As data for this test we use the novel War and Peace by Leo Tolstoy to build character level language models. This is mostly in English and in order to succeed the models must learn to model long term temporal dependencies such as opening and closing quotations marks, correct punctuation and grammar as well as higher level language concepts.

Experiment Setup

Experiments are set up very similarly to [49]. Accordingly we split the 3,258,246 characters into training/validation/test sets with an 80/10/10 % ratio, giving approximately 300,000 characters in the training and validation sets. The book contained a vocabulary of 83 characters. Again we train using ADAM, we found a learning rate of 0.001 to be robust for all models. We use a batch size of 100 and limit back-propagation of gradients to 100 steps. Models were trained for 50 epochs at most.

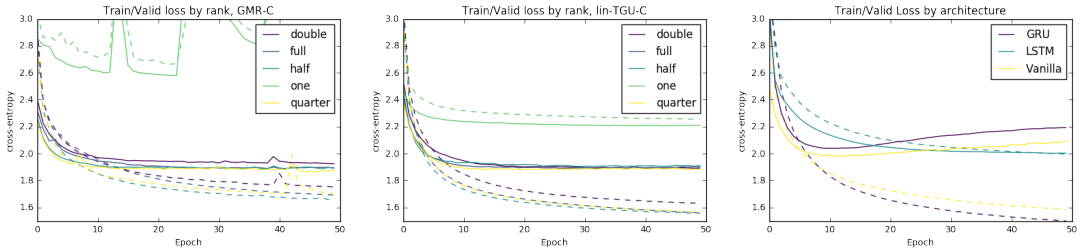
We found even very small models overfit dramatically even after one or two epochs. To approach reasonable baseline performance, we introduce a bottleneck before the network in the form of very small character embeddings. This is equivalent to encoding the input symbols as one-hot vectors (size 83×1) and multiplying by a 8×83 matrix to embed the symbols into an 8 dimensional space. This matrix is learned jointly with the model and has dropout [87] applied to further prevent overfitting.

Models were constrained to have a budget of 25,000 parameters (including the output layer but not including the embedding matrix). This corresponds to an LSTM with 65 hidden units. The rank one GMR with combined biases, by contrast, was able to have 293 hidden units, although it proved incapable of making use of the them. Vanilla RNNs perform well on this task, so we test the GMR in addition to the TGU. As we are interested in the effect of low-rank tensor decompositions on expressive power, we only report results for models with combined biases to exacerbate the effects. We found linear candidate activations consistently outperformed rectifiers for the TGU. Rank was chosen in the same way as the polyphonic music experiment, although we allow ranks as low as a quarter of the number of hidden units.

Results

Training and validation cross entropy are plotted in figure 5.7. The tensor models both show remarkable resistance to overfitting compared to the other architectures. What is particularly interesting is that the validation error does not begin to rise seriously at all, regardless of the rank. Aside from rank one, which struggles to learn at all, the models benefit slightly from lower rank with best test performance being achieved by both tensor models with rank equal to a quarter of their number of hidden units.

Perhaps more unexpectedly, even the tensor model with an excessive rank of twice the number of hidden units still fails to overfit as dramatically as the LSTM or the Vanilla



(a) GMR-C by relative rank.

(b) TGU-C (with linear candidate states) by relative rank.

(c) Other architectures, exhibiting either slow learning or dramatic overfitting.

Figure 5.7: Training and validation curves on War and Peace. Dashed lines indicate training loss, solid lines are the loss on the validation set. 'Full' refers to rank equal to the number of hidden units, 'one' simply means rank 1.

Architecture	Test Cross-Entropy
GMR-C	**2.018**
lin-TGU-C	2.052
Vanilla	2.128
GRU	2.1621
LSTM	2.1271

Table 5.4: Test set cross entropy (per character) for the best War and Peace models.

RNN. This seems to suggest that the tensor units work particularly well with the small embeddings and dropout which were primarily applied to allow the other architectures to report validation cross entropies below three or four at the end of training.

Chapter 6

Conclusions

We have presented a new class of promising architectures for recurrent neural networks. Existing approaches are either seriously flawed or opaque and over-complicated. This necessitates finding more elegant solutions which maintain the performance but keep only components with clearly defined roles and a minimum of confusion and complexity.

The first key observation is that gated RNNs have an inherent tensorial structure. Detailed analysis of this structure indicates that it contributes greatly to their expressive power. This leads to the idea of designing new networks that make explicit this structure, allowing for control over the balance between representative capacity and storage requirements by using a well known tensor decomposition. The aim behind drawing this structure out and making it explicit is that by fully realising the important underlying processes we can construct simplified architectures by removing extraneous parts.

The second key observation is that networks with a gated, additive state update are very appealing. This structure allows the network to sit on the border between vanishing and exploding gradients and should therefore allow excellent learning performance. Correspondingly we decide to keep this structure. We note that this means we have three choices to make: how to implement the gate, how to compute its value and how to compute the additive update. By reasoned analysis, we arrive at clear conclusions for all of them, including taking the novel step of computing an additive update which does not depend on the previous hidden state. This is a key novelty in our proposed class of architectures which allows them to allocate more expressive power to controlling the use of their memory.

Empirically, we find our conclusions are justified. It demonstrates the ability to store items in its memory for long time periods and the ability to store and reproduce arbitrary inputs well in advance of existing architectures. This illustrates the benefits of simplifying the architecture – as each component has a clearly defined role we observe rapid learning even on highly challenging tasks. We also investigate the role of the extra hyper-parameter introduced by utilising a tensor decomposition. This is found to have a useful regularising effect, helping to improve generalisation performance when set appropriately. Further, we are able to derive a useful rule of thumb to be able to set it without excessive exploration.

Now we need to say a few bits and pieces about general life with rnns and how it can better now

Appendix A

Tensor Train

The *tensor-train* (TT) decomposition has a short history – it was first proposed in 2011 as a simpler way of representing an earlier hierarchical form derived from a generalisation of the singular value decomposition. [75] It is proposed as an alternative to the CP-decomposition as it is often easier to find a TT representation of a given tensor.

The tensor-train can also be used more creatively. Novikov et al. [71] 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 (compressing one layer up to $200,000$ times). 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.

A.1 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):

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

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 $r_0 = r_d = 1$. [75]

The three-way case presents no obvious simplifications from the general case but we present it here to ensure consistent notation through the remainder of this section. The TT-decomposition of a three-way tensor of size $n_1 \times n_2 \times n_3$ 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 A.1 where $\boldsymbol{\mathcal{W}}$ is the three way tensor in its decomposed form:

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

In a manner consistent with the CP-decomposition we denote such a decomposed tensor $\boldsymbol{\mathcal{W}} = [\mathbf{A}, \boldsymbol{\mathcal{B}}, \mathbf{C}]_{TT}$. It is important to note that the shapes are less consistent: $\mathbf{A}$ is an $n_1 \times r_1$

(a) Three-way tensor train. (b) General Tensor Train

Figure A.1: Diagrams of the Tensor Train decomposition.

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 A.1 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.

A.2 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

$$\mathbf{z} = \mathbf{x}^\mathsf{T}\mathbf{A}\mathcal{B}\mathbf{C}\mathbf{y}. \tag{A.1}$$

For a specific index this has the form

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

Eq. (A.1) 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.

A.3 Gradients

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

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

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

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

While the gradient of the elements of $\mathcal{B}$ 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. 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.

A.4 Comparison with CP

It is worth comparing the Tensor Train decomposition with the CP decomposition. In the three-way case we find they are very closely related but that the CP has some key advantages.

A.4.1 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 A.4.1. *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

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

where $\operatorname{diag}(\mathbf{v})$ denotes the matrix with vector $\mathbf{v}$ along the leading diagonal and zero elsewhere. 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:

$$X_{ijk} = \left(\mathbf{A} \operatorname{diag}(\mathbf{B}_{j \cdot}) \mathbf{C}^{\mathsf{T}} \right)_{ik} = \sum_{\alpha=1}^{R} A_{i\alpha} \cdot \left(\sum_{\beta=1}^{R} \operatorname{diag}(B_{j \cdot})_{\alpha\beta} C_{k\beta} \right) = \sum_{\alpha=1}^{R} A_{i\alpha} B_{j\alpha} C_{k\alpha}$$

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

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

□

A.4.2 Space Requirements

The number of stored coefficients for a rank R CP-decomposed tensor of size $I \times J \times K$ will be $RI + RJ + RK$. 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 $3RI$, a remarkable reduction.

For a TT decomposition with ranks (R_1, R_2) and dimensions $I \times J \times K$ we need $R_1 I + R_1 J R_2 + R_2 K$ storage. If both ranks are equal, this becomes $RI + R^2 J + RK$, which is quadratic in the rank. As the ranks have to be integers, this gives us less ability to fine-tune the number of parameters in the decomposition as changing the ranks by small amounts may have large implications on the size of the decomposition.

The CP decomposition is clearly the most appealing in this regard. Not only does it only require a single hyper-parameter to control the size, the fact that it grows linearly in that hyper-parameter makes it much more amenable to adjustment.

Appendix B

Additional Tables

This chapter contains additional tables which were omitted from the main text due to space requirements. They are not essential to the main text, but provide illustrative examples.

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 B.1: Example of notation for tensors.

Appendix C

Additional Figures

This chapter contains a number of figures omitted from the main text because the relationships they reveal are very close to earlier figures.

C.1 Learning Tensors

Figure C.1 shows results for learning permuted element-wise multiplication, these are almost identical to the non-permuted version.

C.2 Synthetic Tasks

C.2.1 Addition

Task

Figure C.2 shows an example input sequence for the addition task discussed in section 5.1.1. This was generated by training a TGU with 2 hidden units and rank 1 to solve the task on sequences of length 50. We show the input to the network and the values of the network's hidden states.

Results

Figure C.3 shows results for the addition task on the smaller of the sequences attempted. Results are very similar to those at sequence length 750.

C.2.2 Variable Binding

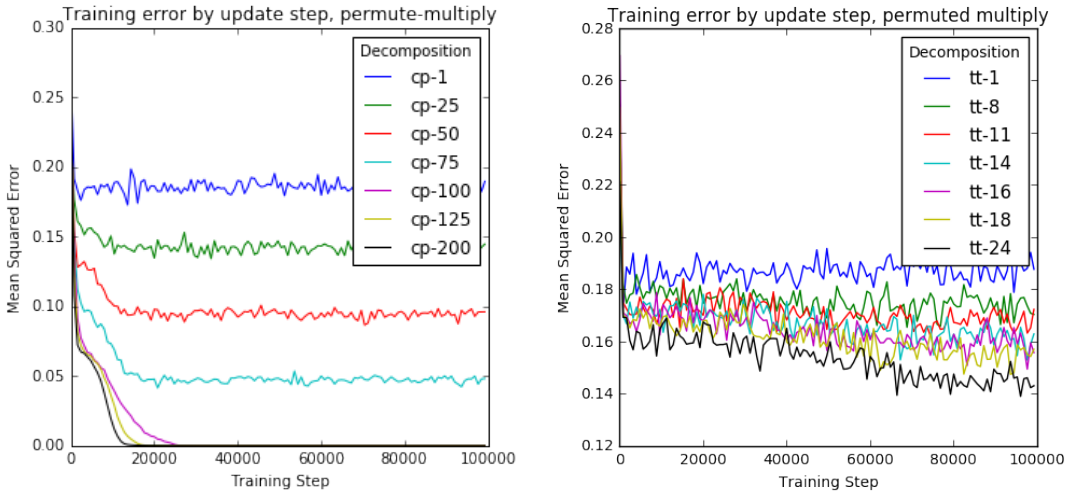
Task

Figures C.4, C.5 and C.6 all show examples of the inputs and outputs for the variable binding task. Including the outputs of a successful network and an network that failed to progress beyond the baseline.

C.3 Real Data

C.3.1 Polyphonic Music

The sizes of the models used are reported in table C.1.



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

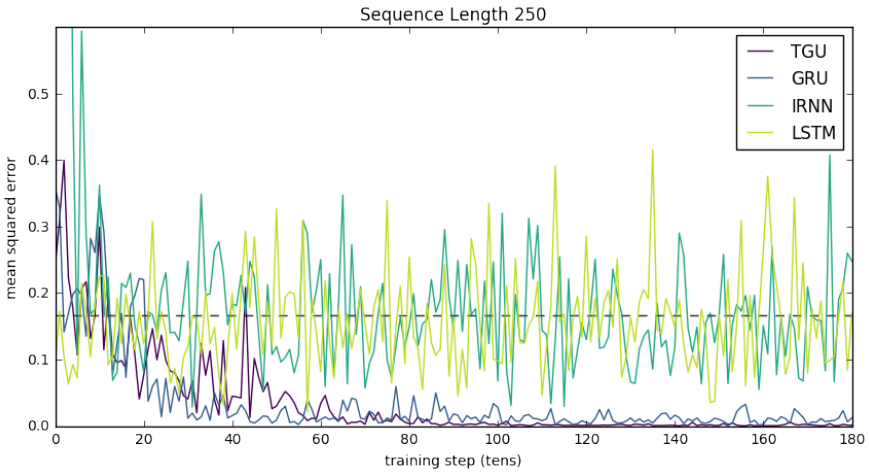
(b) Learning curves for TT-decompositions learning permuted element-wise multiplication

Figure C.1: Training error for various rank decompositions on the permuted element-wise multiplication task.

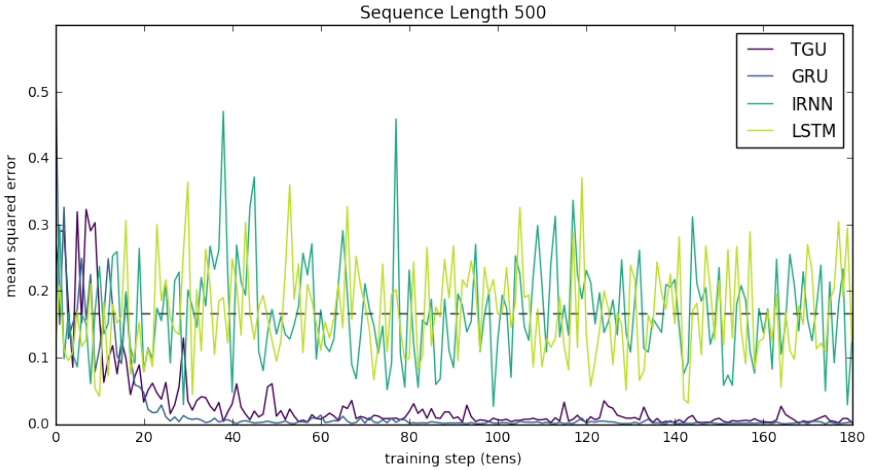
Figure C.2: Example of inputs and states for a network solving the addition task. Images are greyscale, with lower values darker. The bottom line is the input to the network while the top row is its hidden states – clearly it accumulates the appropriate input in a single hidden unit.

Model	hidden units/rank (parameters)			
Vanilla	96 (19927)			
LSTM	44 (20075)			
GRU	52 (19763)			
GMR	95/1 (19870)	71/35 (19872)	58/58 (19775)	45/90 (20125)
GMR-C	349/1 (20005)	106/52 (20142)	76/76 (20119)	53/106 (20248)
TGU	80/1 (20030)	63/31 (20156)	53/53 (20248)	41/82 (19817)
TGU-C	176/1 (20000)	88/44 (20075)	66/66 (19855)	48/96 (20071)

Table C.1: Size of models for polyphonic music modelling. Architectures with -C appended have the bias matrices combined with the decomposition. Parameters are reported for inputs of size 54 as per the JSB dataset. Rank is only reported if applicable.



(a) Sequence length 250.



(b) Sequence length 500.

Figure C.3: Addition results, smaller sequences.

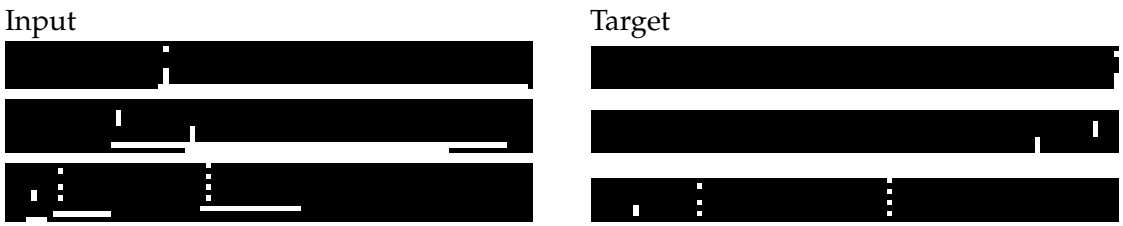


Figure C.4: Example input/target pairs for the three different variable binding tasks. All have length 100, with 8 bit patterns – all that differs is the number of patterns the network may have to remember at once.

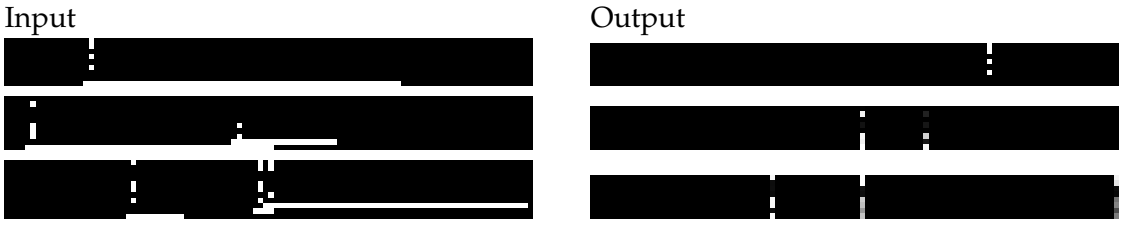


Figure C.5: Outputs from TGU models that that achieved good results.

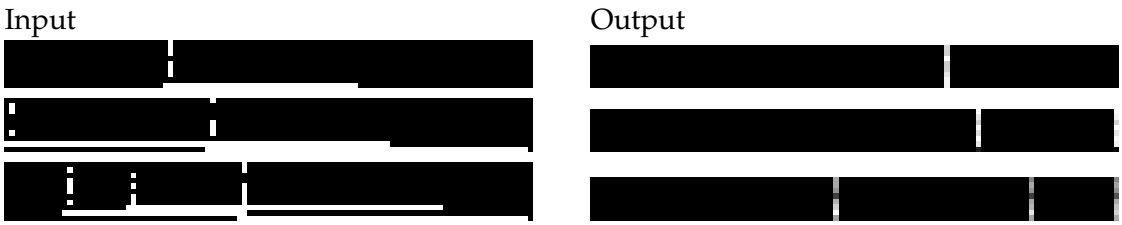


Figure C.6: Output from GRUs demonstrating the baseline behaviour – the timing is correct but the output is not related to the input.

Appendix D

Additional Proofs

There are a number of useful facts which are used throughout the document. Here we provide proofs of those which are not necessarily obvious.

D.1 Element-wise multiplication by bilinear product

This is a useful point about the structure of a tensor implementing element-wise multiplication.

Proposition D.1.1 (Identity Tensor). *$\mathcal{H} \in \mathbb{R}^{N\times N\times N}$ such that*

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

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 $\mathbf{z} = \mathbf{x}^\top \mathcal{H}\mathbf{y}$. Then

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

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. □

Appendix E

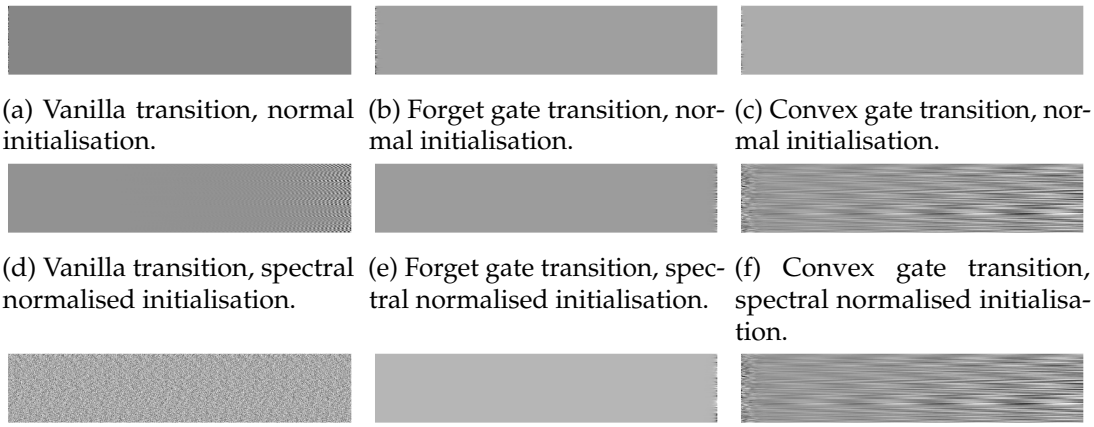
Initialization

RNNs can be quite sensitive to initialisation, especially with regard to learning long time dependencies [55]. It has been suggested that a good initialisation should have eigenvalues on or inside the complex unit circle [107, 66] which would suggest initialising to an orthogonal or orthonormal matrix as per [37].

Figure E.1 shows some simplified simulation to illustrate the effect of different methods of initialisation with different forms of gated recurrence. They are simulated with no non-linearity and a single input at the first time step sampled from a unit normal. If gate values are required, they are sampled uniformly in $[0, 1]$. We then simply run the recurrence over the hidden state for a number of time steps plotting each cell's state as a grayscale value.

We show three possible procedures: random normal (mean 0, standard deviation 0.01), spectral normalised and orthonormal. The spectral normalised matrix is a random normal matrix that has been divided by a fraction of its leading singular value. This ensures the spectral radius of the matrix is slightly greater than one. To generate a random orthonormal matrix we generate a random matrix from a normal distribution, compute its QR decomposition and use the Q.

Results are in figure E.1. Clearly initialising from a normal distribution is insufficient. The spectral normalisation also tends to explode (note that often the initial random state is no longer visible, this is because the degree of normalisation required to display the final states). Correspondingly we prefer the orthonormal initialisation throughout this report, although we note that with Vanilla RNNs which preserve information nearly perfectly from this initialisation we occasionally had to scale the entire matrix down by a constant factor to avoid exploding early in training.



(a) Vanilla transition, normal initialisation.

(b) Forget gate transition, normal initialisation.

(c) Convex gate transition, normal initialisation.

(d) Vanilla transition, spectral normalised initialisation.

(e) Forget gate transition, spectral normalised initialisation.

(f) Convex gate transition, spectral normalised initialisation.

(g) Vanilla transition, orthonormal initialisation.

(h) Forget gate transition, orthonormal initialisation.

(i) Convex gate transition, orthonormal initialisation.

Figure E.1: Example simulations of RNN states from different initialisation. Images are independently normalised, with darker values being more negative and lighter more positive.

Appendix F

Algorithms for Synthetic Tasks

F.1 Addition

Data for the addition task can be generated by algorithm 1. This algorithm produces a single item, it could also be done in batches with slight adjustments.

Algorithm 1: Generating data for addition task

Input: Integer T
Output: Problem instance $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T), y$

Sample integer $i \sim U[1, (T/2) - 1]$
Sample integer $j \sim U[T/2, T]$
$y \leftarrow 0$
for $t \in \{1, \ldots, T\}$ **do**
  Sample float $x_{t1} \sim U[0, 1]$
  if $t = i$ *or* $t = j$ **then**
    $y \leftarrow y + x_{t1}$
    $x_{t2} \leftarrow 1$
  else
    $x_{t2} \leftarrow 0$
return $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T), y$

F.2 Variable binding

Algorithm 2 shows how to generate a single data item for this task. This algorithm is indicative only, there are ways to be more efficient especially if it has to be implemented as a computation graph, for example in Tensorflow.

Algorithm 2: Generating data for variable binding

Input: Integers T, D, N
Output: Problem instance $(\mathbf{x}_1, \ldots, \mathbf{x}_T), (\mathbf{y}_1, \ldots, \mathbf{y}_T)$

intialise $(\mathbf{x}_1, \ldots, \mathbf{x}_T), (\mathbf{y}_1, \ldots, \mathbf{y}_T)$ to zeros
for $i \in \{1, \ldots, N\}$ **do**
  Sample integer $j \sim U[1, (T/2) - 1]$ `/* start position */`
  Sample integer $k \sim U[j + 1, T]$ `/* end position */`
  $\mathbf{z} \leftarrow$ random D bit binary pattern
  $\mathbf{y}_{k+1} \leftarrow \mathbf{z}$
  Set first D bits of $\mathbf{x}_{i+1}$ to $\mathbf{z}$
  for $l \in \{j, \ldots, k\}$ **do** set label bits
    $x_{l,D+i} \leftarrow 1$
return $(\mathbf{x}_1, \ldots, \mathbf{x}_T), (\mathbf{y}_1, \ldots, \mathbf{y}_T)$

Bibliography

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

[2] ALLAN, M., AND WILLIAMS, C. K. I. Harmonising Chorales by Probabilistic Inference. In: *Advances in Neural Information Processing Systems, 17*. 2005, 25–32.

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

[4] BALDUZZI, D., AND GHIFARY, M. Strongly-Typed Recurrent Neural Networks. In: *International Conference on Machine Learning - ICML 2014*. 2016, 9. arXiv: 1602.02218.

[5] BARONE, A. V. M. Low-rank passthrough neural networks. *Arxiv* (Mar. 2016), 16. arXiv: 1603.03116.

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

[7] 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.

[8] 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.

[9] CHAN, W., ET AL. Listen, attend and spell. *arXiv preprint* (2015), 1–16. arXiv: 1508.01211.

[10] 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.

[11] CHO, K., ET AL. On the Properties of Neural Machine Translation: EncoderDecoder Approaches. *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation* (Sept. 2014), 103–111. arXiv: arXiv:1409.1259v2.

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

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

[14] CHUNG, J., ET AL. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv* (Dec. 2014), 1–9. arXiv: 1412.3555v1.

[15] 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.

[16] 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.

[17] COOIJMANS, T., ET AL. Recurrent Batch Normalization. *arXiv preprint* (2016), 1–10. arXiv: 1603.09025.

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

[19] DHINGRA, B., ET AL. Gated-Attention Readers for Text Comprehension. *arXiV* (June 2016). arXiv: 1606.01549.

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

[21] DZMITRY BAHDANA, ET AL. Neural Machine Translation By Jointly Learning To Align and Translate. *International Conference On Learning Representations* (2015), 1–15. arXiv: 1409.0473.

[22] 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.

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

[24] GAL, Y. A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. *arXiv preprint* (2015). arXiv: 1512.05287.

[25] GAO, Y., AND GLOWACKA, D. Deep Gate Recurrent Neural Network. *arXiv preprint* 1 (2016), 7. arXiv: 1604.02910.

[26] 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.

[27] GOODFELLOW, I. J., ET AL. Maxout Networks. *Proceedings of the 30th International Conference on Machine Learning (ICML)* 28 (2013), 1319–1327. arXiv: 1302.4389.

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

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

[30] 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.

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

[32] GREFF, K., ET AL. LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems* (2016). arXiv: 1503.04069.

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

[34] 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.

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

[36] HE, K., ET AL. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv preprint* (2015). arXiv: 1502.01852.

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

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

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

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

[41] 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.

[42] 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.

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

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

[45] HUANG, G., LIU, Z., AND WEINBERGER, K. Q. Densely Connected Convolutional Networks. *arXiv preprint* (2016), 1–12. arXiv: 1608.06993.

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

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

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

[49] KARPATHY, A., JOHNSON, J., AND FEI-FEI, L. Visualizing and Understanding Recurrent Networks. *ICLR* (June 2016), 1–13. arXiv: arXiv:1506.02078v1.

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

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

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

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

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

[55] 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.

[56] LECUN, Y., ET AL. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86, 11 (1998), 2278–2324.

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

[58] LUONG, M.-T., ET AL. Multi-task Sequence to Sequence Learning. In: *International Conference On Learning Representations*. Nov. 2016. arXiv: arXiv:1511.06114v4.

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

[60] MARCUS, M. P., SANTORINI, B., AND MARCINKIEWICZ, M. A. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics* 19, 2 (1993), 313–330.

[61] 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.

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

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

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

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

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

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

[68] 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.

[69] NEYSHABUR, B., SALAKHUTDINOV, R., AND SREBRO, N. Path-SGD: Path-Normalized Optimization in Deep Neural Networks. *arXiv* (2015), 1–12. arXiv: 1506.02617.

[70] NEYSHABUR, B., ET AL. Path-Normalized Optimization of Recurrent Neural Networks with ReLU Activations. *arXiv preprint* (May 2016). arXiv: 1605.07154.

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

[72] OORD, A. van den, ET AL. WaveNet: A Generative Model for Raw Audio (2016), 1–15. arXiv: 1609.03499.

[73] OORD, A. van den, ET AL. Conditional Image Generation with PixelCNN Decoders. *arXiv* (2016). arXiv: 1606.05328.

[74] 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.

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

[76] 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.

[77] PASCANU, R., ET AL. How to Construct Deep Recurrent Neural Networks. *CoRR* abs/1312.6 (2013), 1–10. arXiv: 1312.6026.

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

[79] POLINER, G. E., AND ELLIS, D. P. W. A discriminative model for polyphonic piano transcription. *EURASIP Journal on Advances in Signal Processing* (2007).

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

[81] 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.

[82] SALIMANS, T. Weight Normalization : A Simple Reparameterization to Accelerate Training of Deep Neural Networks. *arXiv* (2016). arXiv: arXiv:1602.07868v1.

[83] 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.

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

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

[86] 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.

[87] SRIVASTAVA, N., ET AL. Dropout : A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research (JMLR)* 15 (2014), 1929–1958. arXiv: 1102.4807.

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

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

[90] 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].

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

[92] 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.

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

[94] 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.

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

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

[97] 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.

[98] 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.

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

[100] 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.

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

[102] 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.

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

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

[105] ZAREMBA, W., SUTSKEVER, I., AND VINYALS, O. Recurrent Neural Network Regularization. *arXiv preprint* (2014). arXiv: 1409.2329.

[106] ZHANG, S., ET AL. Architectural Complexity Measures of Recurrent Neural Networks. In: *ICML*. 2016, 19. arXiv: 1602.08210.

[107] ZILLY, J. G., ET AL. Recurrent Highway Networks. *arXiv preprint* (2016), 11. arXiv: 1607.03474.