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

Recurrent Tensor Networks

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

Recurrent Neural Networks (RNNs) are a class of machine learning techniques for handling sequential data. They have achieved great practical success but prominent architectures are unwieldy and poorly understood. We reveal a *tensor* structure inherent to the most successful of these networks. Understanding and analysing this structure provides valuable insights and shows a natural way to extend neural networks with multi-linear algebra and tensor decompositions.

We propose two types of Recurrent Tensor Networks that utilise these observations: the Generalised Multiplicative RNN and the Tensor Gate Unit. The former unifies a number of existing architectures into a single framework by drawing out the implicit tensor structure. The latter draws from prominent gated RNNs to construct a novel architecture which is simple yet remarkably effective. We demonstrate their performance across a range of tasks exhibiting remarkable results while also illustrating a useful way to trade off model capacity with generalisation ability.

Acknowledgments

Firstly I would like to thank my supervisors David and Marcus for a constant stream of ideas and making it possible to pull this project together into a coherent whole. Also Alex Telfar, for valuable discussions which were sometimes even on topic and Harry Ross, for occasionally entering the firing line when I felt the urge to try and explain something.

I would also thank the School of Engineering and Computer Science for essential computational resources as well as everyone working in Cotton 240 for being relaxed, friendly and not wiping my notes off the whiteboard.

I am grateful to Jack Branthwaite for all the coffee, the impact of which can not be understated. Also Timothy Barraclough for giving me plenty of motivation and my indoor cricket team the Caged Sharks for giving me an opportunity to run around in circles for a few hours a week.

Contents

Figures

Chapter 1

Introduction

Recurrent Neural Networks (RNNs) are powerful machine learning models for learning mappings between pairs of sequences. Such tasks are highly challenging and include language modelling, machine translation, speech recognition and audio classification. They have achieved remarkable success at immense scale [100], but the most successful architectures are complicated and poorly understood while simple alternatives tend to perform badly [6]. Understanding how RNNs solve complex tasks and designing new, simpler architectures that are well-understood from the outset is an important goal.

RNNs make computations at each step of the input sequence based on the current input value and a fixed size vector of hidden states. At each step these states are updated and passed forward which enables the network to model complex temporal dynamics using the hidden states as a working memory. The most successful architectures have two key ingredients: additive state updates and multiplicative gating. The former allows running integration of new information at each time step while the latter provides the ability to selectively ignore irrelevancies.

Multiplicative gating involves computing element-wise products of vectors. We can generalise this by 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 the properties of the bilinear product in detail. 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 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 a very large tensor. 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, 36], fulfils our criteria both theoretically and experimentally.

Equipped with this tensor decomposition we 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, 101]. The TGU includes an additive state update and standard multiplicative gates as well as the bilinear product. This architecture is based on the observation that the gate is the most important part of such an architecture and that if it is sufficiently expressive, other components of the network can be simplified. This results in a unique, clearly defined architecture with full expressive power and a minimum of extraneous components.

These architectures are then evaluated empirically. We achieve excellent results on synthetic

tasks including successfully training a TGU to capture time dependencies up to 10,000 steps, more than 10 times the longest previously reported. To evaluate the ability of the model to carefully write arbitrary patterns to its memory we propose a novel form of *variable binding* as no satisfactory task was present in the literature.

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. The building block derives from the perceptron [80] and the networks are often termed *multi-layer perceptrons* (MLPs). MLPs solve the key limitations of the earlier perceptron [68], but it was not until 1986 when a reliable and general purpose procedure was proposed to train them [81].

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 direction information flows from the input 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 preceding layer. When an input is presented to the network, each neuron computes an activation by applying a non-linear function to a weighted sum of its inputs.

Computationally and notationally it is efficient to express many neurons at once using vectors and matrices. 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 of the layer is given by

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

where $f : \mathbb{R}^n \mapsto \mathbb{R}^n$ is the non-linear function applied element-wise. Common examples of f include the logistic sigmoid $\sigma(x) = 1/(1 + e^{-x})$ or a linear rectifier $\rho(x) = \max(0, x)$ [69]. We include a bias term $\mathbf{b}$ which is also learned and enhances 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 a hidden represenation $\mathbf{h}$ as before and then with another set of weights transform the hidden units to get a final output.

Although such neural networks are universal function approximators [46] it is often beneficial in practice to increase the depth as this can allow the network to express more complex relationships with only a moderate increase in learned parameters [94]. Depth is increased by recursively applying the above transform with fresh weights and biases. Omitting the biases for clarity this gives us the following network with $l - 1$ hidden layers:

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

Adding too much 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 [42, 97] and more recently residual or skip connections which change the structure of the graph to allow better flow of information [37, 47].

2.1.1 Training

Neural networks are typically trained in a supervised manner. 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 feasible $\mathbf{x}$ values and $\mathcal{Y}$ the set of feasible $\mathbf{y}$ values. We wish to learn a mapping which will transform each $\mathbf{x}_i$ into $\mathbf{y}_i$.

To evaluate such mapping 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 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:

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

In general this problem is non-convex and contains many local minima and saddle points [19, 51] although in practice, local methods work remarkably well [27].

The weights are typically learned using gradient descent – in its simplest form, each weight is updated repeatedly in a direction which should reduce the loss:

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

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 [96].

2.2 Recurrent Neural Networks

The Recurrent Neural Networks (RNNs) considered here extend feed-forward networks to address problems where we wish to map a sequence of input vectors $(\mathbf{x}_1, \mathbf{x}_2, \dots \mathbf{x}_T)$ to a sequence of output vectors $(\mathbf{y}_1, \mathbf{y}_2, \dots \mathbf{y}_T)$. They have been applied successfuly to a wide range of tasks which including statistical language modelling [66], machine translation [10, 100], speech recognition [31], music classification [12], image generation [34], automatic captioning [98, 102] and more with incredible success.

[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 except 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.

2.2.1 Classical Formulation

An RNN maintains 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:

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

$f(\cdot)$ is again some element-wise non-linearity, often the hyperbolic tangent: $\tanh(x) = (1 - e^{-2x})/(1 + e^{-2x})$. Final outputs are computed from these states in the same manner as feed-forward networks. This remains the same for the extended architectures considered below, the 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" [99], 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 incurred at each time step. This captures all common cases including sequence classification in which 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 while the term in parentheses measures how much the hidden state at time i affects the hidden state at time j.

We can 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.

Vanishing gradients were diagnosed as early as 1994 where and shown to be unavoidable if the network needs to both store information and be robust to noise [6]. However, this has not precluded further research into alleviating the main symptoms; for a thorough treatment see [76]. In the non-linear case, this remains a serious issue. While exploding gradients are often mitigated by using a *saturating* non-linearity where the gradient tends to zero as the hidden states grow, this only exacerbates the vanishing problem.

Vanishing gradients in RNNs make it much harder to learn to store information for long time periods. This causes RNNs struggle to solve simple tasks if the solution requires remembering an input for many time steps.

Butterfly Effect

A second issue is shared with many iterated non-linear dynamical systems: the *butterfly effect*. This is that seemingly negligible changes in initial conditions can lead to catastrophic changes after many iterations [58]. In RNNs this manifests as near-discontinuity of the loss surface which can make gradient descent struggle. Some partial solutions exist such as clipping the norm of gradients [76], using a regulariser to encourage gradual changes in hidden state [55] as well as more sophisticated optimisation techniques [62].

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 key shared component of these architectures is an *additive* state update. While the vanilla RNN attempts to learn an recurrent function $\mathbf{h}_t = \mathcal{F}(\mathbf{x}_t, \mathbf{h}_{t-1})$, these gated architectures instead learn something like a *residual* mapping $\mathbf{h}_t = \mathcal{F}(\mathbf{x}_t, \mathbf{h}_{t-1}) + \mathbf{h}_{t-1}$. By itself, this

[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 $\nabla_{\mathrm{vec}(\mathbf{W})}\mathcal{L}$, 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.

would assist greatly with vanishing gradients [49, 45]. This is equivalent to adding a skip connection, allowing the state to skip a time step and is directly analogous to the connections now commonly used to address the vanishing gradient problem in very deep feed-forward networks [37, 21, 91].

2.3.1 LSTM

The LSTM was proposed precisely to address 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 [45, 29]:

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

where $\tau(\cdot)$ refers to the tanh, $\sigma(\cdot)$ the sigmoid and $\odot$ is used to denote element-wise multiplication between vectors. The key elements are $\mathbf{i}_t$, $\mathbf{o}_t$ and $\mathbf{f}_t$, termed the *input*, *output* and *forget* gates respectively. These are computed in the same fashion as the activations of a vanilla neural network but use a sigmoid activation function which varies smoothly between zero and one. Combining this with element-wise multiplication has the eponymous gating effect, attenuating various components.

The forget and input gates together control the acceptance or rejection of new information by modulating the degree to which the new candidate state $\mathbf{g}_t$ is accepted into the hidden state $\mathbf{c}_t$. The output gate simply allows the network to prevent its hidden state from being exposed.

2.3.2 GRU

A closely related architecture proposed much more recently is the GRU [10]:

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

This simpler than the LSTM and makes some interesting alterations. 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 components. Half of the model's parameters are dedicated towards computing state updates. The mechanism of their computation places significant emphasis on using temporal context to do so – the *reset* gate $\mathbf{r}_t$ provides the model with the ability to ignore parts of its state but is itself a function of the previous state.

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 [45], and which have become popular benchmarks, 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, 59], no specific empirical validation of this ability is regularly carried out.

2.3.3 Alternative Architectures

While the LSTM and the GRU are the most popular, many other solutions have been proposed. In this section we outline some key lines of thought which help gain understanding of the problems identified with the above models.

An early attempt to solve the vanishing gradient problem involves feeding inputs to different units at different rates. The idea is to force the network to attend to longer time-scales by passing some information at a slower rate inducing a hierarchical structure [40]. This idea has been revisited recently in the form of the Clockwork RNN [54] 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 take a standard LSTM and adjust the temporal connectivity pattern with further multiplicative gates.

These approaches have had some success, although it is not clear how adding extra multiplicative gates could alleviate the vanishing gradient problem. Also many of these approaches add significant extra structures to the network to force it to behave in a manner which it was already fundamentally capable of. That this is necessary suggests more fundamental changes to the underlying model are in order.

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 struggle to learn to store information for very long time periods. This is may be due to the somewhat 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 by ensuring all operations on the hidden state apply element-wise, 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 [56], which simply initialise the recurrent weight matrix to the identity (presaged by the nearly diagonal initialisation in [67]), perform remarkably well on pathological tasks. The importance of the initialisation of the recurrent weights matrix was further emphasised in [39] 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.

2.3.4 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 these works. The first is to decouple the memory from the network so that it can store and interact with arbitrary

quantities of information. The second is the hope that a novel method of storing or addressing information will encourage the networks to solve more complex tasks than is typically feasible with hidden states alone.

A notable effort to decouple RNNs from their memory is the Neural Turing Machine proposed in [30]. This uses a neural network (which may or may not be recurrent) to attend to a large matrix of memory. Reading and writing is done via a address mechanism mostly based on the softmax.

Similarly 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 [32]. A similar approach is Joulin and Mikolov's Stack Augmented RNN [48]. These allow for unbounded, efficient memory at the cost of no random access. 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, exact storage. This is an interesting direction as it enables something akin to a learnable Von Neumann architecture [30].

While these approaches can be highly successful, very few of the tasks they are evaluated on are tasks 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 adding and removing information from a running representation. Hence it is worth revisiting the basic architecture before 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 introducing something close to a Holographic Reduced Representation [78]. This is a very encouraging angle – to re-think how the RNN uses its memory at a basic level is an efficient and sensible way to address the issues. Unfortunately simply adjusting the network to use such a reduced representation was insufficient for good performance and a number of complicating factors had to be incorporated such as keeping many redundant copies of memory which diminishes the elegance of the solution.

In general, the task of an RNN is to learn a mapping from one sequence to another, one step at a time using an intermediate memory (the hidden states). Intuitively a solution must consist of writing to and reading from memory, as new inputs arrive. In an RNN, reading from the memory is straightforward as the entire memory is used to produce the output at each time step. Writing can be separated into two operations: accumulating and forgetting. 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, 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 and require constant prediction should be better suited to networks with an ability to forget information that is no longer relevant.

The solutions in this section sidestep difficulties current RNN architectures have with achieving fully general memory semantics when limited to these operations by allowing additional operations on auxiliary data structures. Aside from capacity they do not fundamentally change the set of available solutions but rather impose structures to guide the network into particular modes of behaviour.

2.4 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 have a long history, introduced by Hinton as early as 1981 [41]. Sigaud et al. separate modern

uses into two classes based on the intention [84]. The first class seeks to use the gate in a manner akin to a transistor in a digital circuit, looking 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, often before any non-linearity. In general, the latter class strictly generalises the former. Two excellent reviews can be found in [63, 84] and rather than repeat in detail the history we attempt to bring out the most salient examples.

Of particular note is the Gated Boltzmann Machine [64] which generalises 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 unit being 1 given a pair of inputs $\mathbf{x}$ and $\mathbf{y}$ is defined as

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

where $\sigma(\cdot)$ is the sigmoid discussed earlier [64]. This architecture naturally extends the operation of a feed-forward network to deal with two inputs. Unfortunately the size of the tensor $\mathcal{W}$ is a serious limitation. This work outlines three key points: multiplicative interactions are a natural way of handling multiple distinct inputs, multiplicative interactions are properly parameterised by tensors and tensors can be prohibitively large.

The Multiplicative RNN [90, 89] (MRNN) picks up this idea, by observing that RNNs naturally have two inputs – the previous state and the current input. They apply the factorisation proposed in the Conditional Restricted Boltzmann Machine [93, 65] to manage the size of the tensor. 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 [62], an approximate second order optimisation method.

Closely related is the recently proposed Multiplicative Integration RNN (MI-RNN) which adjusts the building block of eq. (2.1) to use element-wise multiplication instead of addition [101]. 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 such that the final model required four weight matrices and five bias vectors per unit.

Whether explicitly acknowledged or not, both the MRNN and MI-RNN use a *tensor decomposition* to manage the parameters of a bilinear multiplicative relationship. Where they differ is the precise form of the decomposition and how they apply biases to enhance the expressive power. We find that investigating tensor products and decompositions in general allows us to generalise both of these models into a single more flexible family.

This style of gating is also used to modulate the flow of information between layers of deep architectures, being employed in Gated-Attention Readers to combine representations of questions and potential answers [20]. It is also used in PixelCNN, WaveNet and Highway Networks [73, 72, 88], allowing the model to choose how particular features propagate upwards through layers.

It is important to realise that all of these multiplicative structures, including the LSTM and GRU have this inherent tensor structure. In chapter 3 we investigate with more rigour the types of tensor products which generalise these multiplicative interactions.

Chapter 3

Three-way Tensors

In order to represent functions of two arguments with neural networks, three-way tensors are unavoidable. They have been used, directly or indirectly in a number of highly successful architectures, often without regard for the underlying structure and interpretation of the operation. In this chapter we discuss the vector-tensor-vector bilinear product seen in eq. (2.4) and used implicitly throughout successful multiplicative architectures.

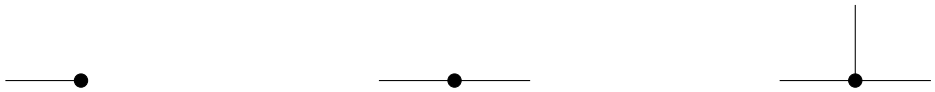
This product admits a surprising number of interpretations including theorem 1 – that it allows us to operate on a pair-wise exclusive-or of features. This is a powerful result which shows that admitting tensors into neural networks can solve the exclusive-or problem without any hidden units. Further, we demonstrate the expressive power of the tensor by showing that a polynomially sized two layer 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 tensors is the storage requirements. We discuss a well-known method for expressing the tensor as a product of much smaller matrices and finally conduct experiments to show that learning the coefficients of a decomposed tensor can be done in situ with gradient descent.

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, n is often referred to as the number of dimensions or modes of the tensor. A matrix is then a two-way tensor and a vector a one-way tensor although we will use their usual names for clarity. Notationally, we follow [53] 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.

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 with 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



(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.

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

$$y_i = \sum_j^n A_{ij} x_j = \langle \mathbf{A}_{i\cdot}, \mathbf{x} \rangle \tag{3.1}$$

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 $\mathbf{A}$ and 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, which can 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 take their inner product with the given vector. If the tensor has n dimensions, the result will have $n - 1$. A three tensor is in this way reduced to a matrix. Kolda and Bader introduce the operator $\bar{\times}_i$ for this, where i represents the index to vary [53]. For the bilinear forms we are concerned with, this leads to the following notation:

$$\mathbf{z} = \mathcal{W} \,\bar{\times}_1\, \mathbf{x} \,\bar{\times}_2\, \mathbf{y}, \implies z_j = \sum_i \sum_k W_{ijk} x_i y_k. \tag{3.2}$$

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

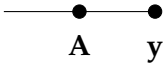
$$\mathbf{z} = \mathbf{x}^\mathsf{T} \mathcal{W} \mathbf{y}. \tag{3.3}$$

This loses none of the precision of the more verbose notation provided we make clear that we intend $\mathbf{x}$ to operate along the first dimension of the tensor and $\mathbf{y}$ the third. With either notation, for a tensor $\mathcal{W} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, we must have $\mathbf{x} \in \mathbb{R}^{n_1}$, $\mathbf{y} \in \mathbb{R}^{n_3}$ and the result $\mathbf{z} \in \mathbb{R}^{n_2}$.

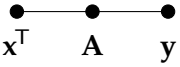
An intuitive way to illustrate these ideas is using Tensor Network Diagrams [16, 74]. In these diagrams, each object is represented as a circle, with each free 'arm' representing an index used to address elements. A scalar has no free arms, a vector one, a matrix two and so on. Figure 3.1 provides examples for these simple objects. These diagrams represent a contractive product by joining the respective arms. As an example, a matrix-vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$ has such a contraction by eq. (3.1), shown in figure 3.2a – it is clear that there is only a single free arm, so the result is a vector as it should be. Figure 3.2 shows some examples of these kinds of products.

3.2 Interpreting Bilinear Products

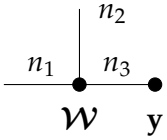
There are several ways to describe the operation performed by the bilinear products we are concerned which correspond to different interpretations of the results. The following interpretations provide insight both into what is actually being calculated and how the product might be applicable in a neural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



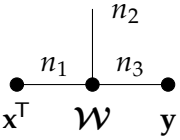
(a) Matrix-vector product $\mathbf{Ay}$, one free arm indicates the result is a vector.



(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.

3.2.1 Stacked bilinear forms

If we consider the expression for a single element of $\mathbf{z}$ in eq. 3.2, we can re-write this in terms of the slices of $\mathcal{W}$ as $z_j = \mathbf{x}^\mathsf{T}\mathcal{W}_{.j.}\mathbf{y}$. This reveals the motivation behind the notation in eq. (3.3) and shows 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 dot product is positive the two vectors are pointing in a similar direction and if it is negative they are in opposing directions. If it is exactly zero, they must be orthogonal. The dot product therefore provides us with a measure 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, 92].

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 has this form. As a matrix-vector multiplication consists of taking the dot product of the vector with each row or column of the matrix 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 think of the rows of the matrix $\mathbf{U}$ as containing patterns to look for in the $\mathbf{b}$ vector and the columns to contain patterns to look 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 based on different interpretations of the underlying feature spaces. 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

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

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 of patterns, which we then multiply by the remaining input vector.

This interpretation is clear from the expression of the product in eq. (3.2). Simply by inserting parentheses we observe that we are first generating a matrix dependent on the first input and multiplying the second input by that matrix.

This intuition of choosing a matrix is suggested in [89] in the context of language modelling with RNNs. The suggestion is that allowing the current input character to choose the hidden-to-hidden weights matrix should allow better use of the context.

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 [95] 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 arises from the observation that the bilinear product is a linear operation on pair-wise products of inputs. This interpretation is essential for understanding the expressive power of the operation as it gives rise to an obvious exclusive-or behaviour. A way to approach this interpretation arises from a method of implementing the bilinear product in terms of simple matrix operations.

To discuss this we need to introduce the *matricisation* of a tensor. This operation is a way of *unfolding* or *flattening* a tensor into a large 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 the 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 awkward – for the general case see [53].

It is easiest to show with an example. Consider the three-way tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{2\times3\times4}$ with slices

$$\mathbf{X}_{1..} = \begin{bmatrix} a & d & g & j \\ b & e & h & k \\ c & f & i & l \end{bmatrix}, \qquad \mathbf{X}_{2..} = \begin{bmatrix} m & p & s & v \\ n & q & t & w \\ o & r & u & x \end{bmatrix}. \tag{3.4}$$

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 these slices

while fixing the third amounts to choosing a column of the slice. The result is:

$$\text{mat}_2(\boldsymbol{\mathcal{X}}) = \begin{bmatrix} a & d & g & j & m & p & s & v \\ b & e & h & k & n & q & t & w \\ c & f & i & l & o & r & u & x \end{bmatrix}. \tag{3.5}$$

We also require the related vec operator from linear algebra. This flattens a matrix into a vector by stacking its columns. For some matrix $\mathbf{A}$ with n columns:

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

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:*

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

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

The indices into $\text{mat}_2(\boldsymbol{\mathcal{W}})$ arise 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, appending the generated columns to a matrix as we go. Only when i has covered 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 eq. (3.4), fixing the second index to 2 gives a 4×2 transposed slice:

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

Vectorising this matrix takes each column and stacks them, giving

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

which is precisely row two of eq. (3.5).

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:*

$$\mathbf{z} = \text{mat}_2(\boldsymbol{\mathcal{W}}) \text{vec}\left[\mathbf{y}\mathbf{x}^{\mathsf{T}}\right] \tag{3.6}$$

Proof. To prove this we can compare the expressions for a single element of the result. An element z_j from eq. (3.6) is formed as the inner product of the j-th row of the flattened tensor and the vectorised outer product of the inputs. By lemma 3.2.1:

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

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 pair-wise conjunctions of the inputs.*

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

Implications of theorem 1:

Higher level features. 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.

XOR. The case where both inputs are the same, $\mathbf{z} = \mathbf{x}^\mathsf{T} \boldsymbol{\mathcal{W}} \mathbf{x}$, 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 [68] – a neural network to solve the problem requires either a hidden layer or an additional input feature such as 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.

Apparent depth. 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 [46], recent results have suggested that increased depth allow exponential increases in expressivity for a fixed number of parameters [22, 94]. We claim that allowing tensor products has the effect of allowing similar increases without additional hidden layers.

The final claim requires 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 reasonably bounded 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* $\mathbf{x}^\mathsf{T}\boldsymbol{\mathcal{W}}\mathbf{x}$ *is capable of exactly computing the squared Euclidean norm* $||\mathbf{x}||_2^2 = \sum_i x_i^2$*.*

Proof. Recalling lemma 3.2.2, $\mathbf{x}^\mathsf{T}\boldsymbol{\mathcal{W}}\mathbf{x} = \text{mat}_2(\boldsymbol{\mathcal{W}})\,\text{vec}(\mathbf{x}\mathbf{x}^\mathsf{T})$. 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 need to define $\text{mat}_2(\boldsymbol{\mathcal{W}})$ such that it picks out the correct elements and sums them. If $\mathbf{x} \in \mathbb{R}^d$ then $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{d\times 1\times d}$ 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 to zero apart from those with indices of the form $1 + id$ for $i \in \{1, \ldots, d\}$ which we set to one. The matrix product will then simply pick out the elements formerly on the diagonal and sum them 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]. The proof is highly technical so we only describe the alterations required to allow a tensor layer. They construct a three-layer network in the following manner: for an input $\mathbf{x}$, the first two layers learn to approximate the map $\mathbf{x} \mapsto ||\mathbf{x}||_2^2 = \sum_i x_i^2$, the final layer learns a univariate function of the squared norm. Finally it is shown this can be done with layers of polynomial width in the input dimension.

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 may 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. □

We have shown a single tensor layer can do the work of two feed-forward layers which provides a strong motivation to explore architectures incorporating these tensor semantics.

[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.

3.3 Tensor Decompositions

Bilinear products show great promise. Unfortunately such a tensor is often very large; an $n \times n \times n$ tensor would require $\Theta(n^3)$ space to store explicitly and a similar number of operations to compute a bilinear product. Managing the size leads to the idea of using 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 balance between the number of parameters required and the expressive power of the model.

Here we outline and analyse the single most important decomposition. We note that three-way tensors are something of a special case in which many methods of tensor decomposition become very closely related. The CP-decomposition discussed here is in many ways the simplest method yet it retains some very appealing properties. It is also worth observing that many of the more complex decompositions were devised to solve the difficulty of finding a CP-decomposition for a given tensor [53, 35]. This issue does not affect us as we aim to learn the decomposed tensor in place to model an unknown relation. We are therefore well placed to prefer the simplicity of the CP.

The key competition is the Tensor-Train decomposition [75], which has an elegant conceptual interpretation. Unfortunately in the three-way case it saves relatively little space and proves hard to train compared to the CP. A full discussion of the differences is deferred to appendix A as the outcome is clear. We also point the interested reader to Kolda and Bader's excellent review of the history and applications of a wide range of tensor decompositions for a full background [53].

3.3.1 CANDECOMP/PARAFAC

This method of decomposing a tensor dates as far back as 1927 [44, 43] but has been rediscovered repeatedly in a range of contexts [53]. 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). [36] 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. Some interesting results on the uniqueness of the decompositions can be found in [53], 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 $\otimes$ denotes the tensor product. This product expands the dimensions; for the first two vectors it is the outer product $\mathbf{a}\mathbf{b}^\mathsf{T}$ which generates a matrix consisting of the product of each pair of elements in $\mathbf{a}$ and $\mathbf{b}$. With three such products we proceed analogously, except now our structure must contain the product of each possible triple of elements. Hence we need three indices to address each entry, so it is 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 [53] 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 $\boldsymbol{\mathcal{X}} = [\mathbf{A}, \mathbf{B}, \mathbf{C}]_{CP}$.

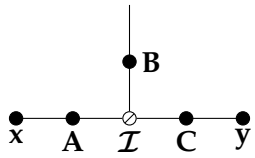


Figure 3.3: Bilinear product in the CP-decomposition.

Bilinear Product

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

Proposition 3.3.1.

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

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

Proof. This follows from straightforward rearranging. Firstly

$$z_j = \sum_i^{n_1}\sum_k^{n_3} W_{ijk}x_i y_k = \sum_i^{n_1}\sum_k^{n_3}\sum_r^{R} A_{ri}B_{rj}C_{rk}x_i y_k = \sum_r^{R} B_{rj}\left(\sum_i^{n_1} A_{ri}x_i \cdot \sum_k^{n_3} C_{rk}y_k\right). \tag{3.7}$$

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 (3.7) 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 matrix multiplication:

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

□

This 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 diagonality is represented diagrammatically with the symbol $\oslash$, 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 $\mathbf{z} = \mathbf{x}^\mathsf{T}\mathcal{W}\mathbf{y}$. Then

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

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 extracted from the inputs.

Bilinear products with the CP decomposition also retain the ability to express element-wise multiplication. Recalling the tensor $\boldsymbol{\mathcal{I}} \in \mathbb{R}^{N \times N \times N}$ used earlier to represent element-wise multiplication, it is straightforward to model with a CP decomposition: $\boldsymbol{\mathcal{I}} = [\mathbf{I}, \mathbf{I}, \mathbf{I}]_{CP}$ 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 to verify the expressive power of the decomposed tensor. We justify theoretical results by showing a tensor layer is capable of solving the exclusive-or problem without hidden units and then apply it to a real world task to illustrate the benefits of the representation.

3.4.1 Gradients

Before training with gradient descent, we first need to derive the gradients. We consider each parameter matrix individually. 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 of partial derivatives. Many authors avoid this by vectorising the matrix [60], 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 partial derivative.

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 $\mathbf{C}$ are very similar:

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

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

$$\frac{\partial z_j}{\partial B_{lm}} = \sum_i^{n_1} \sum_k^{n_3} A_{li} C_{lk} x_i y_{k\cdot} = \langle \mathbf{A}_{l\cdot}, \mathbf{x} \rangle \cdot \langle \mathbf{C}_{l\cdot}, \mathbf{y} \rangle$$

A point to make about these gradients is the prevalence of multiplicative interactions. This has been noted to occasionally cause 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 uses this as motivation for using second-order information during optimisation of RNNs containing similar structures [89]. We prefer to counsel patience – after a single step, the gradients will become perfectly aligned to the scale of the parameters.

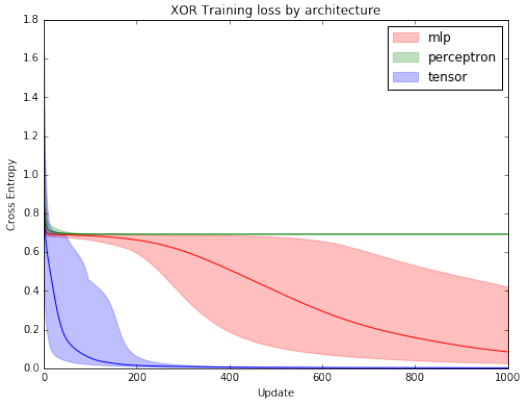


Figure 3.4: Exclusive-or results. Bold line is the mean of 50 runs, shading indicates the minimum and maximum across all runs.

3.4.2 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 standard 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.

Experiment Setup

Truth values are represented as binary ones and zeros. 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 (single layer feed-forward network), 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 $-\log 0.5 = 0.6931$, which corresponds to an output of 0.5 for all inputs (or guessing uniformly at random). As expected the simple 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 corollary 1.1 as this is precisely the kind of relationship we would expect the tensor product to represent naturally.

3.4.3 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 [95]. 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 to

naturally capture interactions in the joint space of the two factors. Our aim for reproducing a similar experiment is to verify that we can learn in place tensor decomposition that represents an unknown, complex relationship.

Problem Formulation

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

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

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 jointly with the tensor $\mathcal{W}$. This is equivalent to representing each style and content label individually 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 multiplying it by a matrix. We refer to such a matrix as an *embedding matrix* as it embeds a sparse encoding into a smaller, densely populated space. It is clear that if we were to attempt to learn the model without these embeddings it would fail completely to generalise. A bilinear product with two such one-of-n 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 and incorporating an embedding matrix further forces the tensor to separately model different modes of interaction.

Data

One dataset explored in [95] 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 or 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 then 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 ensure that using a tensor decomposition to model non-trivial interactions was a feasible goal.

We use a model with embeddings of size 25 and a rank 25 CP-decomposition. This model has 78,350 parameters, while the explicit $25 \times 1024 \times 25$ tensor would have 641,550, an eight fold difference in size. We train the model using ADAM [52], a variant of stochastic gradient descent which dynamically rescales the step size, 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 (we used 26) 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 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 $\lambda ||\mathbf{X}||_F^2 = \lambda \sum_i \sum_j X_{ij}^2$ for each matrix in the model.

Results

Figure 3.5 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 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 3.6. 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 3.7 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 3.5: Learned style and content representations.

(a) Actual images from the dataset.

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

Figure 3.6: Example of images generated from unseen style and content pairs, three letters from three different fonts.

(a) From the data

(b) Generated

Figure 3.7: A difficult example of style and content generalisation.

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. We term this family of architectures *Recurrent Tensor Networks* to emphasise their explicit incorporation of a decomposed tensor. There are two key principles: simplicity and modularity. We want to design networks that have no extraneous components so 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 led 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. 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 size $n_1 + 1 \times n_2 \times n_3 + 1$. If we then perform a bilinear product and separate back out the extra components, we find that as well as the expected bias vector we have two bias matrices, one per input. For a three-way tensor $\boldsymbol{\mathcal{W}}$,

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

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 $\boldsymbol{\mathcal{W}} = [A, B, C]_{CP}$, then appending ones to the inputs results in

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

This is quite different to eq. (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. Further, separating the matrices ensures the tensor product can at least behave in the same manner as a Vanilla RNN block.

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, 101] and Vanilla RNN. To do this, we simply replace the various linear operations with a bilinear form with appropriate biases:

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

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:

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

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):

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

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 in the TGU we incorporate several key ideas from the LSTM and the GRU. This section explains in detail the decisions taken and the analysis behind them.

4.2.1 Solving Vanishing Gradients

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

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

Recalling eq. (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 eq. (4.2):

$$\nabla_{\mathbf{h}_{k-1}}\mathbf{h}_k = \mathbf{I} + \nabla_{\mathbf{h}_{k-1}}\mathbf{z}_k. \tag{4.3}$$

Adding the identity matrix adds one to each eigenvalue of the gradient, which ensures that at least.

While the gradients may not vanish, they are likely to explode. Pascanu et al. show that a necessary condition for exploding gradients is that the largest eigenvalue of this gradient is greater than one [76]. For this additive update all we require for exploding gradients is then for $\nabla_{\mathbf{h}_{k-1}}\mathbf{z}_k$ to have at least one positive eigenvalue. This is highly likely from random initialisation, so this architecture is going to be challenging to train. 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 [33, 49], 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.

Both of these gates apply element-wise. Although as multiplicative structures we could generalise them with a tensor product, the element-wise nature is essential for keeping the gradients intact. Applying a gate will replace the identity matrix in eq. (4.3) with some other matrix. If the gating is done element-wise, then the new matrix will remain diagonal, meaning it will still act purely additively on the eigenvalues of the gradient. If we allow a tensor product, then we could be adding any arbitrary matrix into the gradient, which will exactly re-introduce the pathologies we seek to avoid.

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 and z_t will refer to the candidate update computed by the rest of the network. In general these elements are 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

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

The forget gate allows the network to wholly replace the state with a new value.

The convex gate makes the following modification:

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

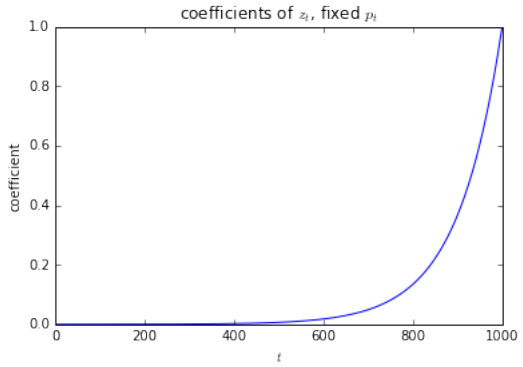
While a simple adjustment, one gate signal is now able to control the acceptance or rejection of new information.

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.

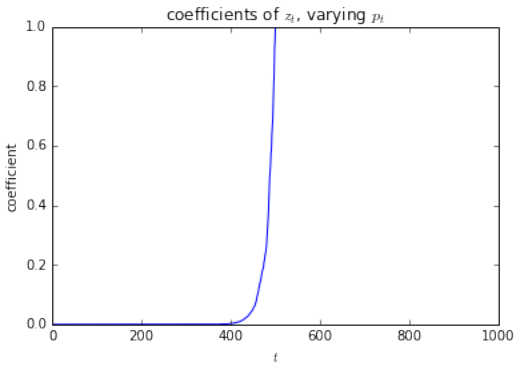
Forget Gate

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

[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.

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 ignore 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. A second method of remembering a single candidate state is by reproducing the same candidate state at each time and setting the forget gate always 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 (which correspond to using a very unusual non-linear tensor decomposition). 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 eq. 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 induce a strong decay, the network will be forced to update primarily on local information and 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 start higher [49].

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

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

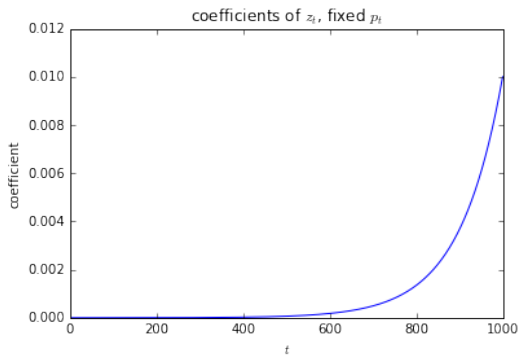
ensuring we define $\prod_{j=i+1}^{t} p_j = 1$ if $i = t$. This is very close to what we had above, but the $1 - p_i$ term in the coefficients has a significant impact. Firstly, we can prove the following statement which shows that the sum over states is a convex sum.

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

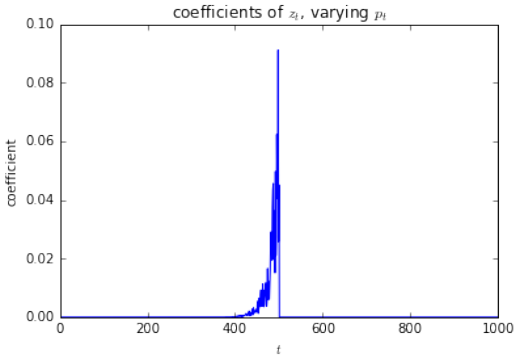
$$\sum_{i=0}^{t} \left(\prod_{j=i+1}^{t} p_j \right) (1 - p_i) = 1.$$

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 allows a much wider range of shapes.

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

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

□

This means that the scale of the states depends only on the scale of the candidate activations. Intuitively this should help address the danger of exploding inherent in the additive update. 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 forget gate.

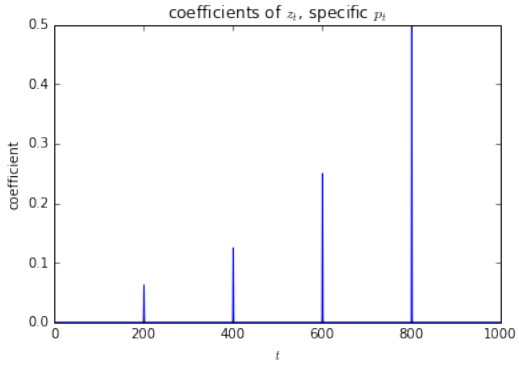


Figure 4.3: Setting specific p_i to 0.5 picks out a set of past candidates, albeit with an exponential decrease in their weighting.

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

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

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

$$\frac{\partial h_t}{\partial z_i} = \left(\prod_{k=i+1}^{t} \frac{\partial h_k}{\partial h_{k-1}} \right) \frac{\partial h_i}{\partial z_i} = \left(\prod_{k=i+1}^{t} p_k \right) (1 - p_i). \tag{4.5}$$

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 (4.4) 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 means 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, eq. (4.5) shows that proposition 4.2.1 also applies to the gradients. This suggests that as long as the scale of the candidate updates is 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 value of 0.5, but this would correspond to an exponential decay of information. Unlike the forget gate simply increasing the mean value (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 where successfully employing a forget gate requires a more complex method of producing candidate states. The convex gate avoids the issue of requiring two components to learn complementary behaviours by allowing a single component to carry out the job. This is 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, the mechanism to compute its values will have to be capable of modelling complex interactions. The bilinear tensor product is therefore 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, especially as keeping the gate activation bounded in $[0, 1]$ will ensure the gradients will not explode due to the additive connection.

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. Aside from Strongly Typed RNNs [4] and some Associative LSTM variants [18] this is usually performed via a binary function of the current input and the previous hidden state. Intuitively this needs to be the case with the standard LSTM style forget gate, but as we have decided against that it is worth questioning the necessity of the previous hidden state in this calculation.

Removing this recurrent dependence makes each role in the network clearly defined. As each input arrives, 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 best determined empirically.

What remains is to decide 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 [28, 38]. It is interesting that for many tasks a linear candidate performed best – clearly the expressive power of the tensor combined with the non-linearity of 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 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 improve 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.

Initialisation of the architectures was not explored in great detail. For most networks we find initialising the square matrices to be orthonormal and sampling others from a small uniform distribution is a sufficiently robust procedure. A brief explanation of how random orthonormal matrices are sampled, as well as a visual comparison of some procedures can be found in appendix E.

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. We therefore 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 are unable to resolve the time dependencies required.

5.1.1 Addition

Task

The first task is designed to test the networks' ability to store information for long time periods. It is one of the tasks originally proposed to benchmark the LSTM [45], although we use the slightly different formulation found more recently in [56]. The problem is a common benchmark for RNNs and has featured in a number of recent works ([3, 39, 5, 70] for example).

The inputs 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 while figure C.1 shows an example. The 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 mini-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 (160 trainable parameters, compared to the LSTM's 328). Updates were applied using ADAM [52] 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 [56]) we found most successful architectures converge within 1000 on most sequences.

While previously reported results for this task stop at $T = 750$, our most successful architectures still converge rapidly to a solution at this length. 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 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 and the rank of the tensor decomposition was increased proportionally to 16.

The models tested were the LSTM, GRU, IRNN [56] and our TGU. 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 prediction.

Results

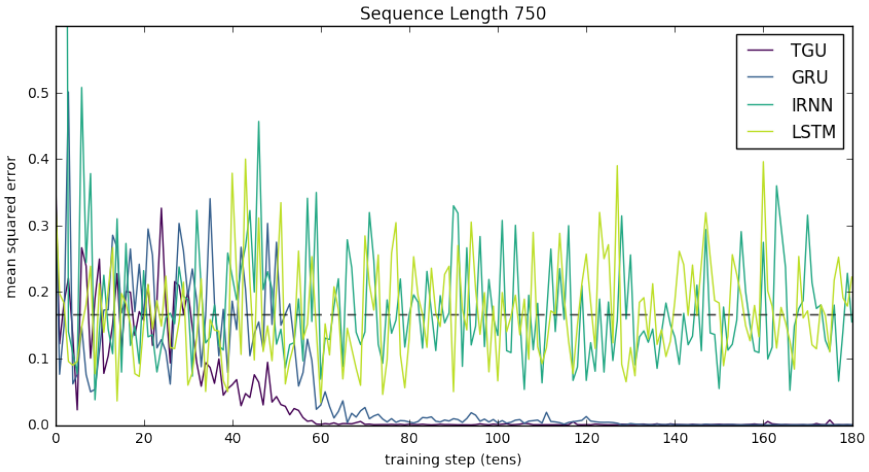
The baseline for this task is a mean squared error of 0.1767 which corresponds to predicting 1 for every input sequence. Figure 5.1 shows that we were unable to find a solution with the IRNN or LSTM within the time allotted, while the GRU and TGU consistently did so very rapidly – in less than 1,000 iterations – several orders of magnitude faster than previously published results [56, 3, 39]. Results for shorter sequences can be found in figure C.2 in the appendix.

The figures also show that this task can be frustrating. Due to the online nature, the loss curves are very noisy, so it is challenging to determine whether a model is making progress. In accordance with a number of others' results [56, 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. We have refrained from smoothing the loss curves to make clear this abrupt shift in behaviour when models begin to converge.

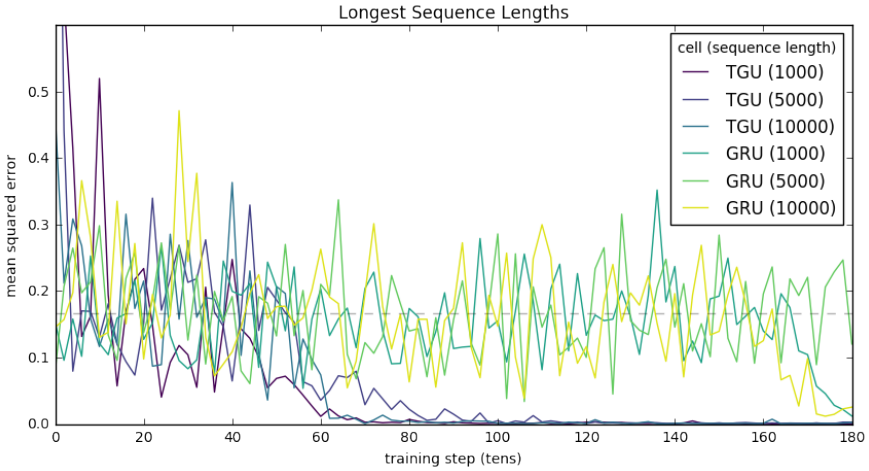
For sequences up 750 steps long there is little to separate the GRU and the TGU. As they share a gating mechanism, it is probable that this is a key factor in their success. As discussed in chapter 4 this convex gating mechanism is capable of naturally representing exactly the behaviour required to solve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

These results reveal a key weakness in the LSTM. While others have shown it is capable of solving the task eventually [39], it was unable to do so in a comparable amount of time. This adds weight to the hypothesis that the co-operation required between components in the LSTM is challenging to learn by gradient descent.

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 to solve. Results for the TGU and GRU on long sequences are shown in figure 5.1b. The GRU begins to beat the baseline on two of the tasks inside 2,000 updates.



(a) Sequence length 750. GRU and TGU both converge within 1000 steps.



(b) Very long sequences. TGU consistently converges in less than 1000 steps, while GRU shows no signs until 1600 if at all.

Figure 5.1: Results for addition task

More remarkably, the TGU is unaffected by the extreme length of the sequences still achieving a solution inside 1,000 steps.

The TGU performed much better on this task with a ReLU layer producing candidate updates, which is contrary to some of our later findings. A possible reason 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 simply summing the candidate activations equally will solve the task.

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 take the RNN and unroll it through all 10,000 time steps it is equivalent to a feed-forward network of extraordinary depth with shared weights at each layer.[1] In order to train successfully we must back-propagate gradients through 7,500 steps on average. While the shared weights undoubtedly assist it is worth observing that this is still more than 5 times the depth of the

[1] As the loss 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.

deepest feed-forward networks reported [47].

5.1.2 Variable Binding

This task tests the architectures' ability to store arbitrary patterns in memory and recall them in an organised manner. We term this "variable binding" because to solve the task the network needs to associate an arbitrary value with a specific label. The task is a natural sub-task of many large scale problems to which RNNs have already been successfully applied. An example is translation: in an end-to-end translation system, the network must, for example, detect 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 tested this ability. Close alternatives include the copy task in [45] and the variants considered in [30], but these are concerned with recalling the order in which symbols from a fixed vocabulary appear. The "variable assignment" task used to evaluate the Associative LSTM [18] has a similar aim in mind but again assesses recall of temporal patterns which does not require learning the kind of delayed one-to-one mapping we want to investigate.

Task

We propose the following synthetic task to test the ability of the network to learn this sort of mapping: 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 consists of T vectors of size D, containing the appropriate pattern when each of the label bits switches off, with zeros elsewhere. Section C.1.2 in the appendix includes some examples of this while algorithm 2 in the appendix shows 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 reason for the success of the TGU with linear candidate updates, as the correct output can be reconstructed by applying the inverse linear transformation, which may be easier to learn than the inverse of a highly non-linear 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 robust for all models. Models were trained with a mini-batch size of 32 using ADAM with a base learning rate of 0.01. We ceased 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 to remember leading to nine different experiments.

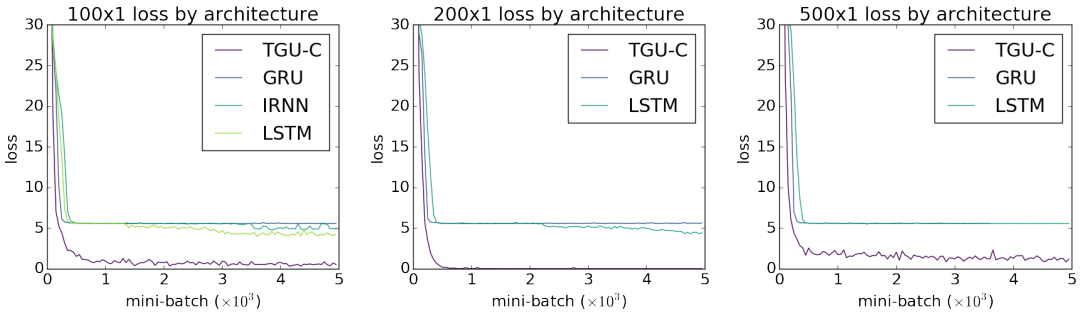
The baseline behaviour, which all architectures achieved rapidly, is to wait until any of the label bits turns off and output a pattern with mean 0.5. This achieves a total loss on sequences with N eight bit patterns of $-8N \log 0.5$. Examples of a networks exhibiting this behaviour can be found in figure C.5 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 it is equivalent to generalisation performance.

We set the number of hidden units to ten times the number of 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 fair to ensure they all have the same memory capacity. As the patterns are eight bits, having ten hidden units per pattern ensures it is possible for the networks to store the inputs directly and still have some units to track the label bits. As the patterns are binary, it should be straightforward to learn a compressed representation, so these networks are very much overspecified.

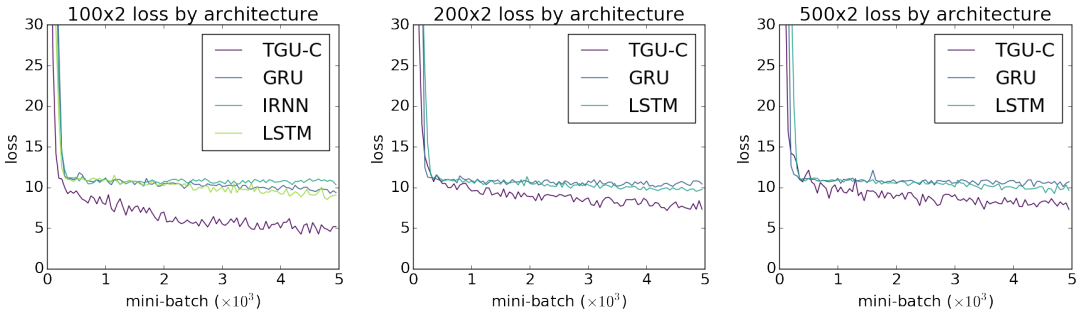
Results

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 found linear activations gave the most consistent results. We also tested the Vanilla RNN, which failed to beat the baseline in any tests and several TGU variants – for clarity we only present results for architectures which showed some promise. We were unable to reliably train IRNNs on sequences longer than 100; despite heavy clipping the gradients consistently exploded. We show the median of five runs as there were occasional outliers.



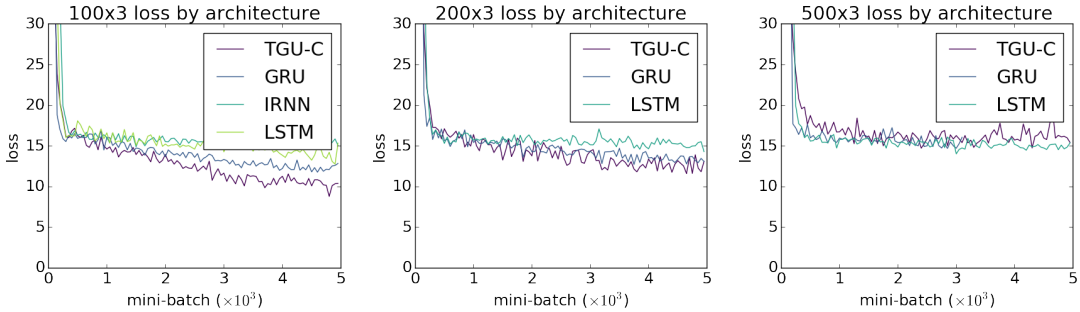
(a) Sequence length 100. (b) Sequence length 200. (c) Sequence length 500.

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



(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.

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

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 a solution although the TGU is the only architecture to make notable progress.

Increasing the number of items to remember, even just to two, makes the task considerably harder. The task highlights an issue with the manner in which RNNs interact with their memory – they are forced to attend to it all at once. In a TGU where the candidate state does not depend on the previous hidden states this requires learning a redundant mapping to embed the input into the state-space as 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 different sets of hidden units are used to store patterns for each label, the candidate update must propose to update all of them with the new input. It is up to the gate to select which set of units to update. Learning such a redundant mapping is considerably more challenging than simply learning the single reversible transformation required to solve the simplest form of the task.

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 no clear result and when the sequence length is 500 no models were capable of surpassing the baseline even when allowed to train for significantly longer than shown in figure 5.4.

The results presented here 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 significant difference to the manner in which the network attends to its state. This change is clearly beneficial for this task which is highly encouraging.

5.1.3 Sequential MNIST

Two recently proposed tasks for RNNs which have gathered significant popularity involve classifying the MNIST handwritten image dataset [57]. To turn two dimensional images into sequences 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 [56].

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, most of the information in the scan-line version is focused in the middle. In contrast, taking a random permutation of the indices is very likely to induce very long time dependencies in addition to removing 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 [104, 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 models which fail miserably in real world applications.

Proposed Architectures: CP+ and CP-Δ

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:

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

This architecture is likely to struggle – the gradients are almost guaranteed to explode during back-propagation by the argument in section 4.2.1. Further, it can never remove information from its states as the state update 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 more flexible model allows *subtractive* forgetting:

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

We name 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 perhaps less pronounced. In practice we find this 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*. 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 [82] but we achieved better results by dividing each matrix in the decomposition by its Frobenius norm.[2]

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

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

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 [56]	82.0
*u*RNN [3]	91.4
*s*TANH-RNN [104]	94.0
BN-LSTM [17]	**95.4**

Table 5.1: Test accuracy for permuted sequential MNIST.

found using grid-search. We train using ADAM with a mini-batch size of 100. For nearly all models, severe gradient clipping [77] is required to train successfully. The best clipping threshold was also found using grid search. Following [56] 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 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.

As expected, figure 5.5b figure 5.5b shows the CP+ 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 so most of the work is being done by a single 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.

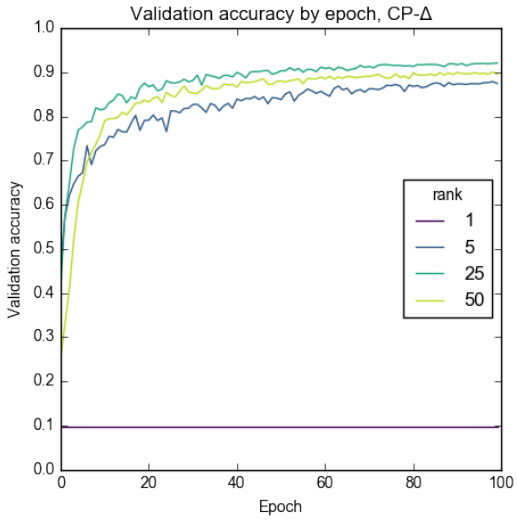
5.2 Real-world Data

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:

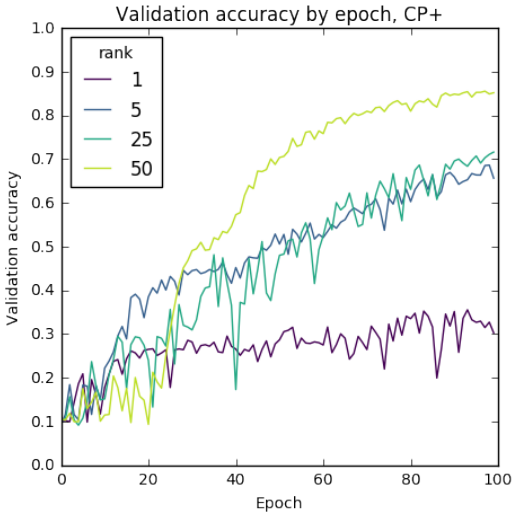
$$p(\mathbf{x}_{t+1}|\mathbf{x}_1, \ldots, \mathbf{x}_t).$$

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.

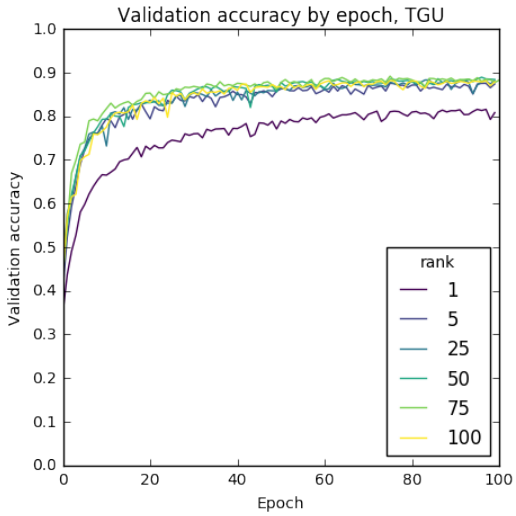
Therefore the target sequence is the input sequence shifted across by one. If we prepend a special *start-of-sequence* symbol to the data sequence and append an *end-of-sequence* symbol, then the inputs to the network are simply all but the *end-of-sequence* and the outputs are all but the *start-of-sequence*.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



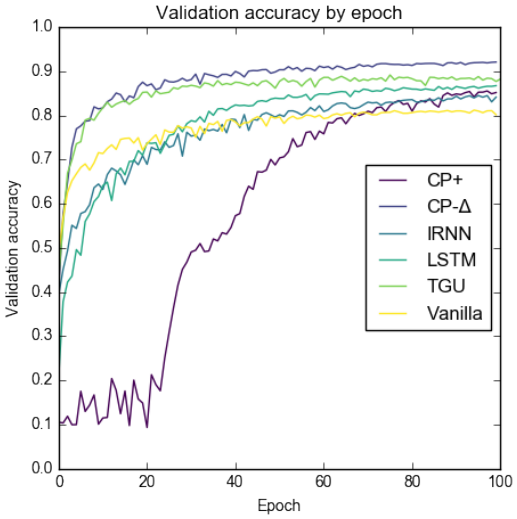
(a) CP-Δ by rank



(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.

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. While our architectures are promising, the resources required to compete at the largest scale are prohibitory (for example, [33] used 15 years of single CPU equivalent time). Rather, we simply 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 used, so it is interesting to see if the benefits of our architectures persist.

5.2.1 Polyphonic Music

This task, introduced by [7], consists of modelling four datasets of polyphonic music from a score. This is challenging – 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 present at the same time) and long term information (such as which key the piece is in). Modelling both these dependencies is challenging but reflects the difficulties inherent in 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 playable by a single 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 is 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].

JSB consists of 382 chorales harmonised by J. S. Bach. This dataset has the lowest polyphony, as it is written in four part harmony. It also has the smallest range of notes, only 54. 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.

Experiment Setup

The data is converted from MIDI files (which contain timestamped 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 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 makes sense to limit the model to an appropriate range for those instruments. Due to this we can not be directly compare results although ours are very similar, with nearly all of our models 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.

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

[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*
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
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

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

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 to either 1 or a multiple of the number of hidden states, chosen from $\{1/2, 1, 2\}$. Table B.2 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 mini-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. 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 perform very well. 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 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 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 Penn Treebank

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

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: Per-word perplexity of best early-stopped models on the Penn Treebank test set.

use the Penn Treebank [61] 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 [103, 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 [103] by replacing all but the most common 10,000 words with an 'unknown' symbol. We train using ADAM with a mini-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. This is because we want to know if having control over the number of parameters in the tensor decomposition has a significant regularisation effect in isolation from the size of the memory.

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 was indistinguishable from the Vanilla RNN on this data, so we omit it from our results.

Results

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 so 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 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.

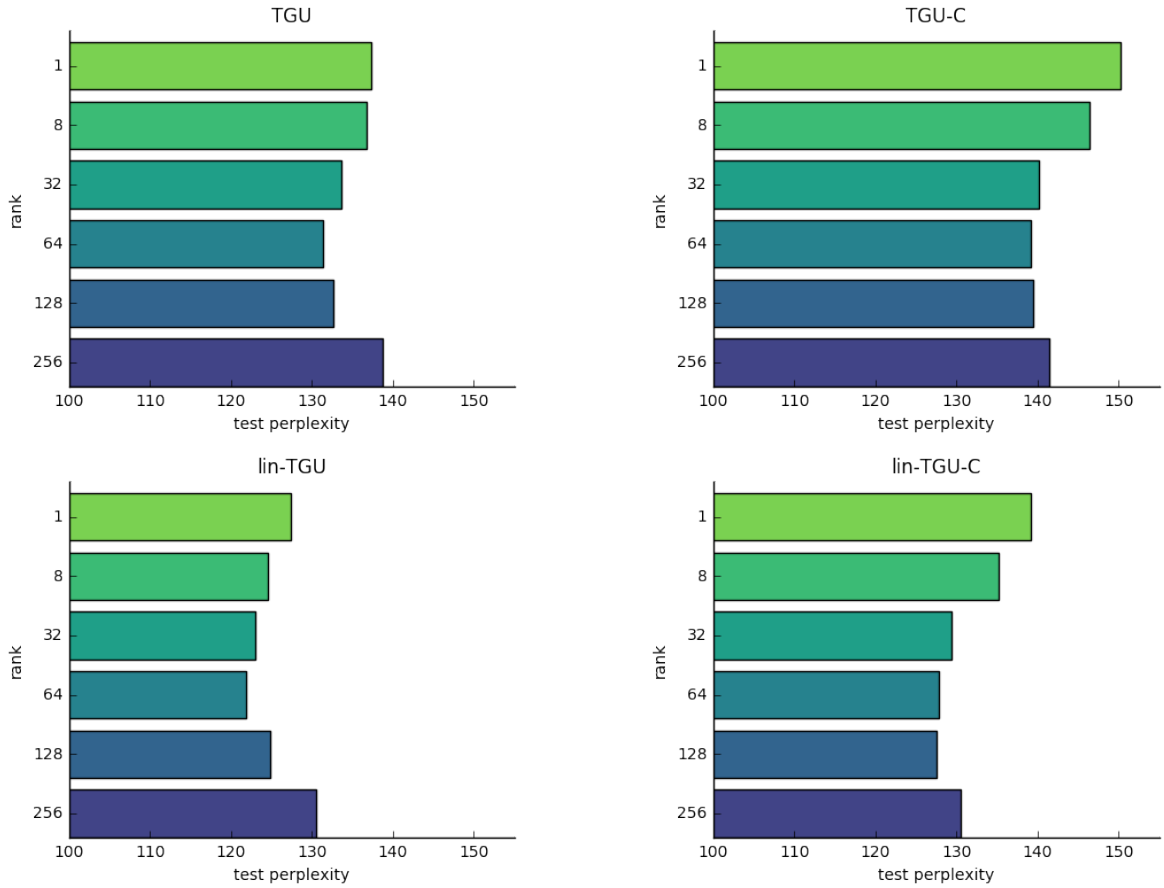


Figure 5.6: Test perplexity of best early stopped models for each type of TGU for a range of ranks on the Penn Treebank. Networks had 128 hidden units.

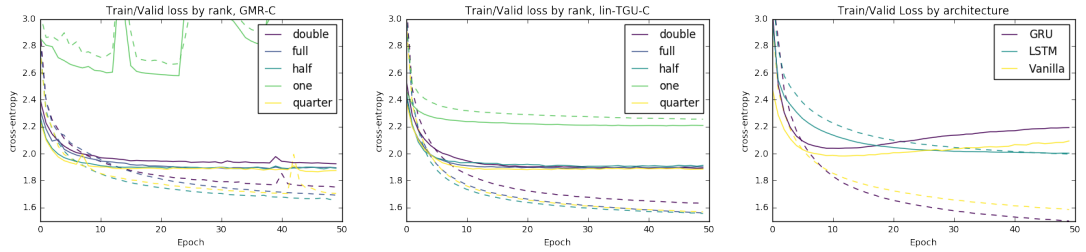
As data for this test we use the novel War and Peace by Leo Tolstoy to build character level language models. This is mostly in English and in order to succeed the models must learn to model long term temporal dependencies such as opening and closing quotation marks, correct spelling and grammar as well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

Experiment Setup

Experiments are set up following [50]. 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 mini-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



(a) GMR-C by relative rank.

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

(c) Other architectures exhibit 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.

hidden units, although it proved incapable of making use of 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 again. 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. The GMR reported the best results on the test set – we attribute this to the fact that its only component is a tensor product so altering the rank has a much greater effect.

Perhaps more unexpectedly, even the tensor model with an excessive rank of twice the number of hidden units still overfits less than the LSTM or the Vanilla RNN. This suggests that the tensor units work particularly well with the small embeddings and dropout, techniques primarily applied to allow the other architectures to report validation cross entropies in a reasonable range at the end of training.

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.

Chapter 6

Conclusions

RNNs are highly successful models for sequence learning, the most successful of which all incorporate some form of multiplicative gating. Multiplicative interactions are naturally parameterised using bilinear tensor products; understanding these is therefore essential to designing successful RNNs. We find such tensor products to be highly expressive and flexible building blocks for neural networks, naturally capturing complex relationships between two inputs. We therefore propose an RNN architecture, the Generalised Multiplicative RNN (GMR), which uses a bilinear product at its core.

We also show the benefits of RNNs with an additive state update: it naturally pushes the network to the edge between vanishing and exploding gradients. Managing such a structure with a multiplicative gate is a key reason why RNNs such as the LSTM and GRU are successful. Comparing the two gating mechanisms employed by the LSTM and GRU, we find that the *convex* gate from the GRU has the appealing property of being able to manage both the acceptance and rejection of potential state updates with a single signal.

The proposed Tensor Gate Unit (TGU) takes full advantage of this convex gate. It controls access to its memory with a bilinear tensor unit which increases the range of allowable access patterns considerably. Imbuing the gate with the extra expressivity of a tensor product means we are able to produce candidate updates in a purely feed-forward manner which provides an appealing separation of roles.

Empirically, we find the TGU and GMR demonstrate excellent performance. On a synthetic addition task which contains long time dependencies the TGU exhibits none of the difficulties encountered by other models. In fact, we are unable to find a sequence too long for the TGU – we achieve successful results learning time dependencies more than 10 times longer than previously reported. This amounts to training a feed-forward network with shared weights at each layer that is up to 5 times deeper than the deepest reported.

To investigate the ability of the TGU to use its memory carefully, we look for a task which requires writing and retrieving arbitrary patterns. As a satisfactory task was not present in the literature, we propose a novel task to test RNNs in this way. Again the TGU shows performance clearly beyond the LSTM and the GRU.

We also address a popular synthetic task: classifying permuted MNIST digits. While the TGU performs creditably, we observe that this dataset is somewhat unnatural. To show this we design two further architectures, the CP+ and the CP-Δ which perform remarkably well on this task despite their clear shortcomings. We suggest this task does not require selective access to memory to solve. As the CP+ and CP-Δ have constrained processes in this regard, the fact that they can perform well reinforces this hypothesis.

Further results on a number of polyphonic music datasets indicate that the two recurrent tensor networks exhibit excellent performance. We observe that models with lower rank tensor decompositions often generalise better than more complex models. To investigate

whether this effect can be harnessed to control model complexity we carry out two further experiments. On the Penn Treebank and the novel War and Peace we find that controlling the rank of the tensor indeed allows for control over the balance between expressive power and generalisation performance.

We have demonstrated a new way of thinking about the building blocks of recurrent neural networks. By investigating thoroughly the implicit tensor structure common to successful architectures we find a powerful way to incorporate tensors into neural networks in a manner which naturally extends the basic structures to handle two inputs. Incorporating a flexible tensor decomposition into recurrent neural networks gives us a class of Recurrent Tensor Networks with a bilinear product at their core. Two such networks are proposed which demonstrate excellent performance due to both the architectural advancements in the TGU and the ability to tune the rank of the tensor decomposition to balance model complexity with generalisation performance.

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):

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

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 $n_1 \times r_1$ matrix, $\boldsymbol{\mathcal{B}}$ a $r_1 \times n_2 \times r_1$ tensor and $\mathbf{C}$ an $r_2 \times n_3$ matrix.

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

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

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

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

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 $\boldsymbol{\mathcal{B}}$ 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 $\boldsymbol{\mathcal{B}}$ behave similarly to the CP-decomposition:

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

This is very similar to the CP-decomposition. 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.

A.4.3 Experiments

In this section we detail experiments comparing the two decompositions.

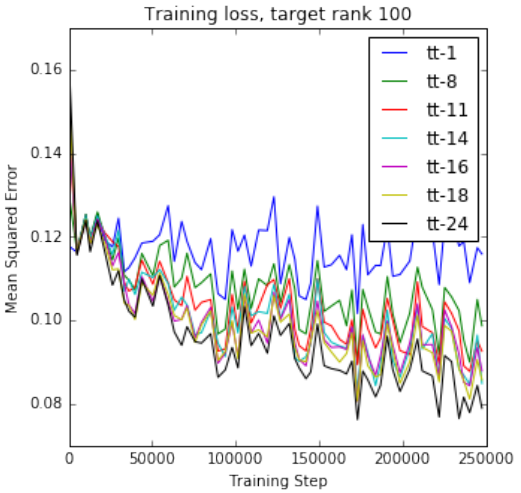
Random Bilinear Products

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. This 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.

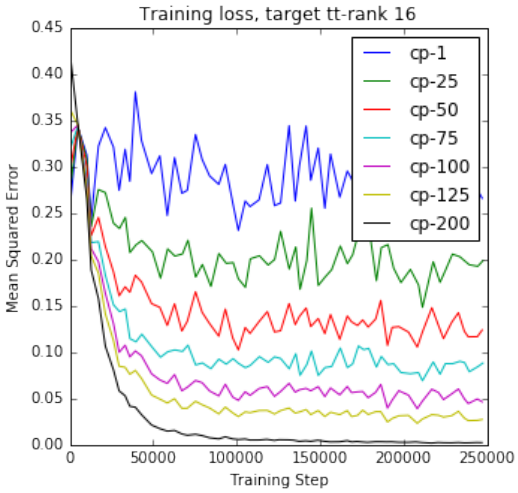
(a) Attempting to approximate by directly learning factor matrices representing a CP-decomposition of various ranks



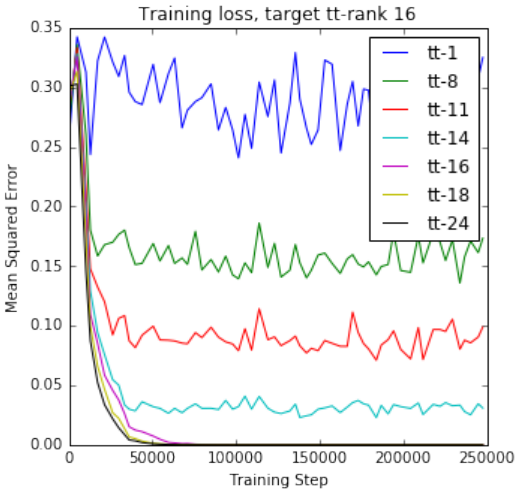
(b) Attempting to approximate by directly learning a TT-decomposition of various ranks

Figure A.2: Results for learning direct approximations of a CP-rank 100 tensor.

Results. 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 A.2. The



(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 A.3: Results for learning direct approximations of a tt-rank 16 tensor.

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 A.3. As should be expected, the TT-decomposition performed better. In particular a TT-decomposition matching the rank of the target tensor reached a very low error extraordinarily quickly, performing much better than the CP-decomposition under the same conditions.

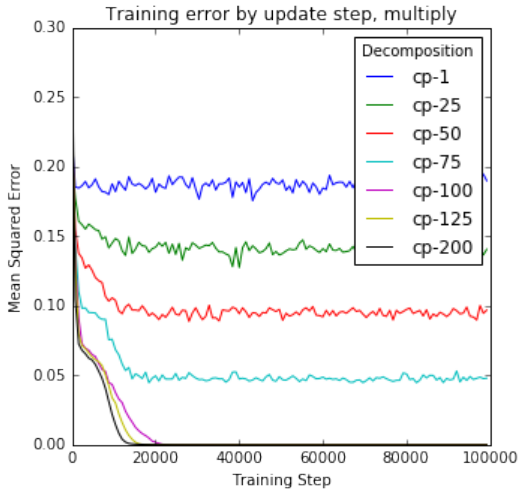
These results imply both decompositions are potentially useful. The CP-decomposition learns reasonable approximations even when it does not necessarily reflect the structure of the underlying problem. The TT-decomposition struggles in this situation, but when the problem is structured more favourably it takes greater advantage.

Learning to multiply

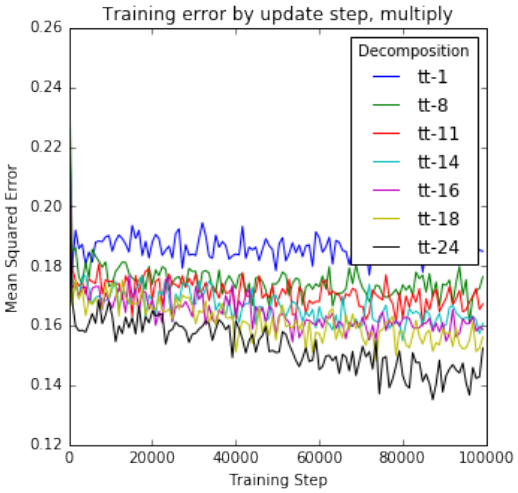
Goal. Learning bilinear functions provides a way to linearly approximate functions of two variables. A motivating example of such a function is element-wise multiplication, also known as the Hadamard product. In this section we investigate briefly the ability of the two decompositions to model such a 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 A.4.3 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



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



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

Figure A.4: Training error for various rank decompositions on the element-wise multiplication task.

$\{0, 1\}$ 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 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.

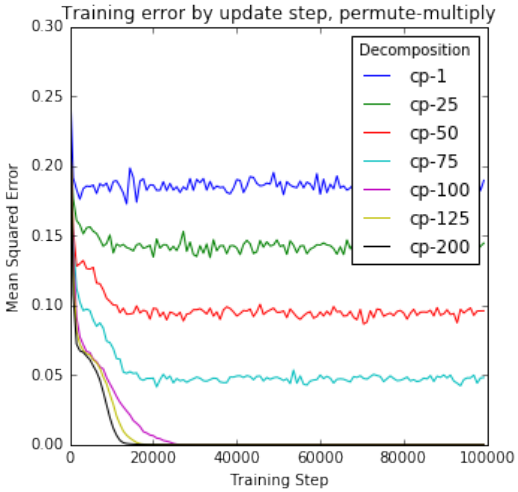
For a more realistic test we introduce a random structure to the input vectors. This is done by creating smaller vectors and expanding them to the appropriate size by copying the elements to random positions. The mapping of positions is fixed for the duration of a training run. We then experiment with varying the size of these smaller vectors and hence varying the degree of correlation between elements while keeping the rank of the decomposition being trained fixed. Here we have fewer preconceptions regarding the relative performance of the decompositions – by introducing structure and reducing the complexity of the target relationship we would hope to see both decompositions making good progress.

Results. The results are shown in figure A.4 and 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.

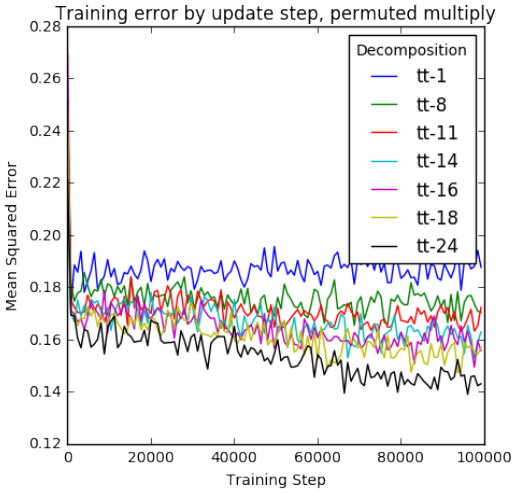
When there is significant structure to the inputs, the TT decomposition is notably worse. One hypothesis is that the CP decomposition simply represents higher rank tensors with the same number of parameters and that this is is what allows to better represent more complex structures. This is shown in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

A.5 Conclusion

In the three-way case, it is clear the the CP-decomposition is preferable. Theoretically it should be able to maintain its expressive power even at low-rank (and therefore with few parameters). Empirically this indeed seems to be the case.

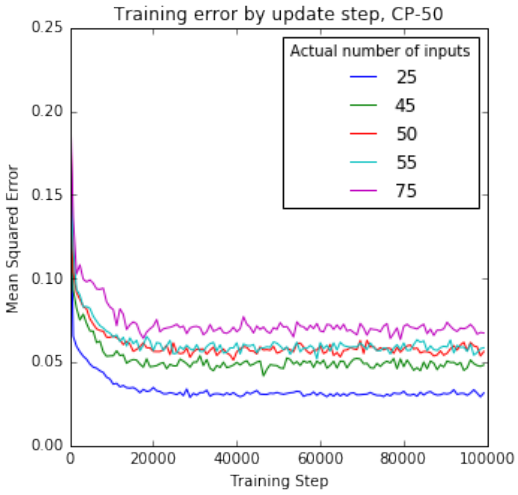


(a) Learning curves for CP-decompositions learning permuted element-wise multiplication

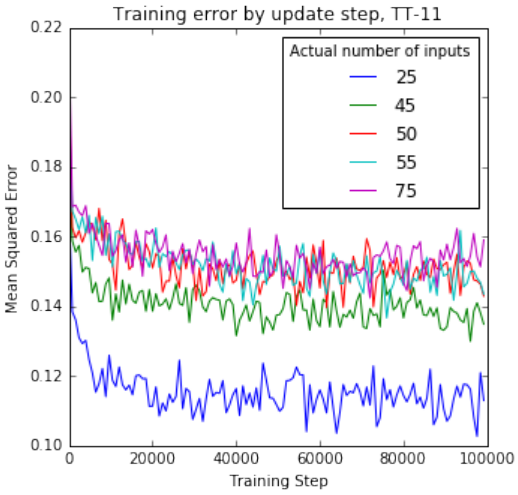


(b) Learning curves for TT-decompositions learning permuted element-wise multiplication

Figure A.5: Training error for various rank decompositions on the permuted element-wise multiplication task.



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



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

Figure A.6: Training error for two decompositions learning element-wise multiplication with varying amounts of structure.

Appendix B

Additional Tables

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.

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.

B.1 Real Data

B.1.1 Polyphonic Music

The sizes of the models used are reported in table B.2.

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 B.2: 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.

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 Synthetic Tasks

C.1.1 Addition

Task

Figure C.1 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

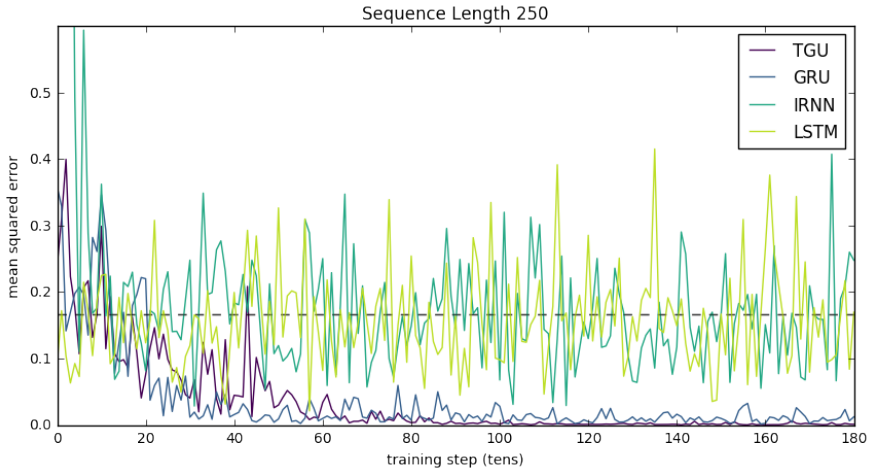
Figure C.2 shows results for the addition task on the smaller of the sequences attempted. Results are very similar to those at sequence length 750.

C.1.2 Variable Binding

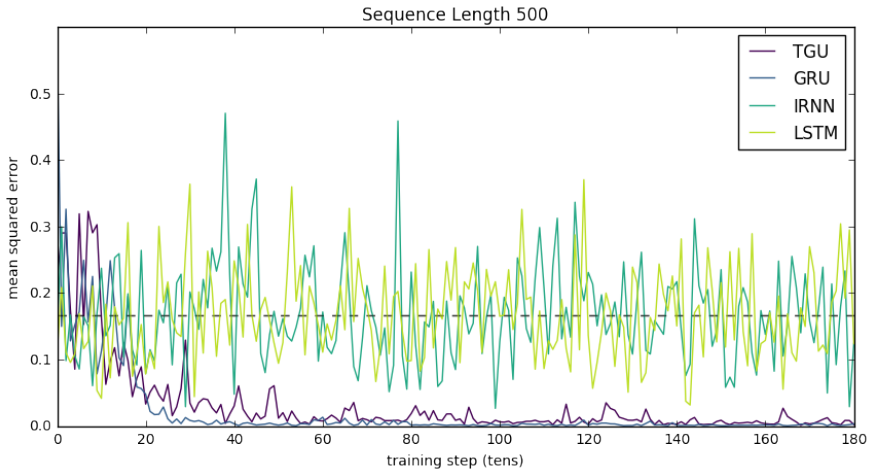
Task

Figures C.3, C.4 and C.5 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.

Figure C.1: 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.



(a) Sequence length 250.



(b) Sequence length 500.

Figure C.2: Addition results, smaller sequences.

Figure C.3: 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.4: Outputs from TGU models that that achieved good results.

Figure C.5: 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)**.** $\boldsymbol{\mathcal{H}} \in \mathbb{R}^{N\times N\times N}$ *such that*

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

implies

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

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

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

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

Appendix E

Initialization

RNNs can be sensitive to initialisation, especially with regard to learning long time dependencies [56]. It has been suggested that a good initialisation should have eigenvalues on or inside the complex unit circle [105, 67] which would suggest initialising to an orthogonal or orthonormal matrix as per. [39].

Figure E.1 shows some simplified simulations to illustrate the effect of different methods of initialisation in conjunction 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. This makes it quite clear whether the initialisation preserves the variance of the original input by observing the patterns it makes. Significant periodicities are good as they indicate the network has plenty of temporal information to latch onto during training. If most of the state is uniform grey it would suggest that the state vanishes or explodes.

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.

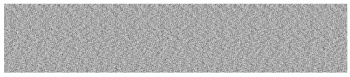
(b) Forget gate transition, normal initialisation.

(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] DAUPHIN, Y., ET AL. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv* (June 2014), 1–14. arXiv: 1406.2572.

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

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

[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., VINYALS, O., AND SAXE, A. M. Qualitatively Characterizing Neural Network Optimization Problems. In: *International Conference On Learning Representations*. 2015. arXiv: arXiv:1412.6544v5.

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

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

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

[31] 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.

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

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

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

[35] HACKBUSCH, W., AND KUHN, S. A new scheme for the tensor representation. *Journal of Fourier Analysis and Applications* 15, 5 (2009), 706–722.

[36] 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.

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

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

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

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

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

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

[43] 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.

[44] 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.

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

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

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

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

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

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

[51] KAWAGUCHI, K. Deep Learning without Poor Local Minima. In: *NIPS*. 2016. arXiv: 1605.07110.

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

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

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

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

[56] 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.

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

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

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

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

[61] 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.

[62] 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.

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

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

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

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

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

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

[69] 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.

[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] 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.

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

[93] 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.

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

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

[96] 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.

[97] 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.

[98] 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.

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

[100] 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.

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

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

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

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

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