Strongly-Typed Recurrent Neural Networks

David Balduzzi[1] DBALDUZZI@GMAIL.COM
Muhammad Ghifary[1,2] MGHIFARY@GMAIL.COM
[1]Victoria University of Wellington, New Zealand
[2]Weta Digital, New Zealand

Abstract

Recurrent neural networks are increasing popular models for sequential learning. Unfortunately, although the most effective RNN architectures are perhaps excessively complicated, extensive searches have not found simpler alternatives. This paper imports ideas from physics and functional programming into RNN design to provide guiding principles. From physics, we introduce type constraints, analogous to the constraints that forbids adding meters to seconds. From functional programming, we require that strongly-typed architectures factorize into stateless learnware and state-dependent firmware, reducing the impact of side-effects. The features learned by strongly-typed nets have a simple semantic interpretation via dynamic average-pooling on one-dimensional convolutions. We also show that strongly-typed gradients are better behaved than in classical architectures, and characterize the representational power of strongly-typed nets. Finally, experiments show that, despite being more constrained, strongly-typed architectures achieve lower training and comparable generalization error to classical architectures.

1. Introduction

Recurrent neural networks (RNNs) are models that learn nonlinear relationships between sequences of inputs and outputs. Applications include speech recognition (Graves et al., 2013), image generation (Gregor et al., 2015), machine translation (Sutskever et al., 2014) and image captioning (Vinyals et al., 2015; Karpathy & Fei-Fei, 2015). Training RNNs is difficult due to exploding and vanishing gradients (Hochreiter, 1991; Bengio et al., 1994; Pascanu et al., 2013). Researchers have therefore developed gradient-stabilizing architectures such as Long Short-Term Memories or LSTMs (Hochreiter & Schmidhuber, 1997) and Gated Recurrent Units or GRUs (Cho et al., 2014).

Unfortunately, LSTMs and GRUs are complicated and contain many components whose roles are not well understood. Extensive searches (Bayer et al., 2009; Jozefowicz et al., 2015; Greff et al., 2015) have not yielded significant improvements. This paper takes a fresh approach inspired by dimensional analysis and functional programming.

Intuition from dimensional analysis. Nodes in neural networks are devices that, by computing dot products, measure the similarity of their inputs to representations encoded in weight matrices. Ideally, the representation learned by a net should "carve nature at its joints". An exemplar is the *system of measurement* that has been carved out of nature by physicists. It prescribes units for expressing the readouts of standardized measuring devices (e.g. kelvin for thermometers and seconds for clocks) and rules for combining them.

A fundamental rule is the *principle of dimensional homogeneity*: it is only meaningful to add quantities expressed in the same units (Bridgman, 1922; Hart, 1995). For example adding seconds to volts is inadmissible. In this paper, we propose to take the measurements performed by neural networks as seriously as physicists take their measurements, and apply the principle of dimensional homogeneity to the representations learned by neural nets, see section 2.

Intuition from functional programming. Whereas feedforward nets learn to approximate functions, recurrent nets learn to approximate programs – suggesting lessons from language design are relevant to RNN design. Language researchers stress the benefits of constraints: eliminating `GOTO` (Dijkstra, 1968); introducing type-systems that prescribe the interfaces between parts of computer programs and guarantee their consistency (Pierce, 2002); and working with stateless (pure) functions.

For our purposes, types correspond to units as above. Let us therefore discuss the role of states. The reason for recur-

Proceedings of the 33^{rd} International Conference on Machine Learning, New York, NY, USA, 2016. JMLR: W&CP volume 48.

rent connections is precisely to introduce state-dependence. Unfortunately, state-dependent functions have side-effects – unintended knock-on effects such as exploding gradients.

State-dependence without side-effects is not possible. The architectures proposed below encapsulate states in *firmware* (which has no learned parameters) so that the *learnware* (which encapsulates the parameters) is stateless. It follows that the learned features and gradients in strongly-typed architectures are better behaved and more interpretable than their classical counterparts, see section 3.

Strictly speaking, the ideas from physics (to do with units) and functional programming (to do with states) are independent. However, we found that they complemented each other. We refer to architectures as strongly-typed when they both (i) preserve the type structure of their features and (ii) separate learned parameters from state-dependence.

Overview. The core of the paper is section 2, which introduces strongly-typed linear algebra. As partial motivation, we show how types are implicit in principal component analysis and feedforward networks. A careful analysis of the update equations in vanilla RNNs identifies a flaw in classical RNN designs that leads to incoherent features. Fixing the problem requires new update equations that preserve the type-structure of the features.

Section 3 presents strongly-typed analogs of standard RNN architectures. It turns out that small tweaks to the standard update rules yield simpler features and gradients, theorem 1 and corollary 2. Finally, theorem 3 shows that, despite their more constrained architecture, strongly-typed RNNs have similar representational power to classical RNNs. Experiments in section 4 show that strongly-typed RNNs have comparable generalization performance and, surprisingly, lower training error than classical architectures (suggesting greater representational power). The flipside is that regularization appears to be more important for strongly-typed architectures, see experiments.

Related work. The analogy between neural networks and functional programming was proposed in (Olah, 2015), which also argued that representations should be interpreted as types. This paper extends Olah's proposal. Prior work on typed-linear algebra (Macedo & Oliveira, 2013) is neither intended for nor suited to applications in machine learning. Many familiar RNN architectures already incorporate forms of *weak-typing*, see section 3.1.

2. Strongly-Typed Features

A variety of type systems have been developed for mathematical logic and language design (Reynolds, 1974; Girard, 1989; Pierce, 2002). We introduce a type-system based on linear algebra that is suited to deep learning. Informally, a *type* is a vector space with an orthogonal basis. A more precise definition along with rules for manipulating types is provided below. Section 2.2 provides examples; section 2.3 uses types to identify a design flaw in classical RNNs.

2.1. Strongly-Typed Quasi-Linear Algebra

Quasi-linear algebra is linear algebra supplemented with nonlinear functions that act coordinatewise.

Definition 1. *Dot-products are denoted by* $\langle \mathbf{w}, \mathbf{x} \rangle$ *or* $\mathbf{w}^\intercal \mathbf{x}$. *A* ***type*** $\mathcal{T} = \big(V, \langle \bullet, \bullet \rangle, \{\mathbf{t}_i\}_{i=1}^d\big)$ *is a* d*-dimensional vector space equipped with an inner product and an orthogonal basis such that* $\langle \mathbf{t}_i, \mathbf{t}_j \rangle = \mathbf{1}_{[i=j]}$.

Given type $\mathcal{T}$, we can represent vectors in $\mathbf{v} \in V$ as real-valued d-tuples via

$$\mathbf{v}_\mathcal{T} \leftrightarrow (v_1, \ldots, v_d) \in \mathbb{R}^d \quad \text{where } v_i := \langle \mathbf{v}, \mathbf{t}_i \rangle.$$

Definition 2. *The following operations are* admissible*:*

T1. ***Unary operations on a type:*** $\mathcal{T} \to \mathcal{T}$
Given a function $f : \mathbb{R} \to \mathbb{R}$ *(e.g. scalar multiplication, sigmoid* σ*, tanh* τ *or relu* ρ*), define*

$$f(\mathbf{v}) := \big(f(v_1), \ldots, f(v_d)\big) \in \mathcal{T}.$$

T2. ***Binary operations on a type:*** $\mathcal{T} \times \mathcal{T} \to \mathcal{T}$
Given $\mathbf{v}, \mathbf{w} \in \mathcal{T}$ *and an elementary binary operation* $\mathsf{bin} \in \{+, -, \max, \min, \pi_1, \pi_2\}$[1]*, define*

$$\mathsf{bin}(\mathbf{v}, \mathbf{w}) := \big(\mathsf{bin}(v_1, w_1), \ldots, \mathsf{bin}(v_d, w_d)\big).$$

Binary operations on two different types (e.g. adding vectors expressed in different orthogonal bases) are not *admissible.*

T3. ***Transformations between types:*** $\mathcal{T}_1 \to \mathcal{T}_2$
A type-transform is a linear map $\mathbf{P} : V_1 \to V_2$ *such that* $\mathbf{P}(\mathbf{t}_i^{(1)}) = \mathbf{t}_i^{(2)}$ *for* $i = \{1, \ldots, \min(d_1, d_2)\}$. *Type-transformations are orthogonal matrices.*

T4. ***Diagonalization:*** $\mathcal{T}_1 \to (\mathcal{T}_2 \to \mathcal{T}_2)$
Suppose that $\mathbf{v} \in \mathcal{T}_1$ *and* $\mathbf{w} \in \mathcal{T}_2$ *have the same dimension. Define*

$$\mathbf{v}_{\mathcal{T}_1} \odot \mathbf{w}_{\mathcal{T}_2} := (v_1 \cdot w_1, \ldots, v_d \cdot w_d) \in \mathcal{T}_2,$$

where $v_i := \langle \mathbf{v}, \mathbf{t}_i^{(1)} \rangle$ *and* $w_i := \langle \mathbf{w}, \mathbf{t}_i^{(2)} \rangle$. *Diagonalization converts type* $\mathcal{T}_1$ *into a new type,* $\mathcal{T}_2 \to \mathcal{T}_2$*, that acts on* $\mathcal{T}_2$ *by coordinatewise scalar multiplication.*

Definition 1 is inspired by how physicists have carved the world into an orthogonal basis of meters, amps, volts etc. The analogy is not perfect: e.g. $f(x) = x^2$ maps meters to square-meters, whereas types are invariant to coordinatewise operations. Types are looser than physical units.

[1] Note: π_i is projection onto the i^{th} coordinate.

2.2. Motivating examples

We build intuition by recasting PCA and feedforward neural nets from a type perspective.

Principal component analysis (PCA). Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ denote n datapoints $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}\} \subset \mathbb{R}^d$. PCA factorizes $\mathbf{X}^\intercal\mathbf{X} = \mathbf{P}^\intercal\mathbf{D}\mathbf{P}$ where $\mathbf{P}$ is a $(d \times d)$-orthogonal matrix and $\mathbf{D} = \mathrm{diag}(\mathbf{d})$ contains the eigenvalues of $\mathbf{X}^\intercal\mathbf{X}$. A common application of PCA is dimensionality reduction. From a type perspective, this consists in:

$$\mathcal{T}_{\{\mathbf{e}_k\}} \xrightarrow[(i)]{\mathbf{P}} \mathcal{T}_{\{\mathbf{p}_k\}} \xrightarrow[(ii)]{\texttt{Proj}} \mathcal{T}_{\{\mathbf{p}_k\}} \xrightarrow[(iii)]{\mathbf{P}^\intercal} \mathcal{T}_{\{\mathbf{e}_k\}},$$

(i) transforming the standard orthogonal basis $\{\mathbf{e}_k\}_{k=1}^d$ of $\mathbb{R}^d$ into the *latent type* given by the rows of $\mathbf{P}$; (ii) projecting onto a subtype (subset of coordinates in the latent type); and (iii) applying the inverse to recover the original type.

Feedforward nets. The basic feedforward architecture is stacked layers computing $\mathbf{h} = f(\mathbf{W} \cdot \mathbf{x})$ where $f(\bullet)$ is a nonlinearity applied coordinatewise. We present two descriptions of the computation.

The standard description is in terms of dot-products. Rows of $\mathbf{W}$ correspond to features, and matrix multiplication is a collection of dot-products that measure the similarity between the input $\mathbf{x}$ and the row-features:

$$\mathbf{W}\mathbf{x} = \begin{pmatrix} \cdots & \mathbf{w}_1 & \cdots \\ & \vdots & \\ \cdots & \mathbf{w}_d & \cdots \end{pmatrix} \mathbf{x} = \begin{pmatrix} \langle \mathbf{w}_1, \mathbf{x} \rangle \\ \vdots \\ \langle \mathbf{w}_d, \mathbf{x} \rangle \end{pmatrix}.$$

Types provide a finer-grained description. Factorize $\mathbf{W} = \mathbf{P}\mathbf{D}\mathbf{Q}^\intercal$ by *singular value decomposition* into $\mathbf{D} = \mathrm{diag}(\mathbf{d})$ and orthogonal matrices $\mathbf{P}$ and $\mathbf{Q}$. The layer-computation can be rewritten as $\mathbf{h} = f(\mathbf{P}\mathbf{D}\mathbf{Q}^\intercal\mathbf{x})$. From a type-perspective, the layer thus:

$$\mathcal{T}_{\mathbf{x}} \xrightarrow[(i)]{\mathbf{Q}^\intercal} \mathcal{T}_{\text{latent}} \xrightarrow[(ii)]{\mathrm{diag}(\mathbf{d}) \odot \bullet} \mathcal{T}_{\text{latent}} \xrightarrow[(iii)]{\mathbf{P}} \mathcal{T}_{\mathbf{h}} \xrightarrow[(iv)]{f(\bullet)} \mathcal{T}_{\mathbf{h}},$$

(i) transforms $\mathbf{x}$ to a latent type; (ii) applies coordinatewise scalar multiplication to the latent type; (iii) transforms the result to the output type; and (iv) applies a coordinatewise nonlinearity. Feedforward nets learn interleaved sequences of type transforms and unary, type-preserving operations.

2.3. Incoherent features in classical RNNs

There is a subtle inconsistency in classical RNN designs that leads to incoherent features. Consider the updates:

$$\text{vanilla RNN:} \quad \mathbf{h}_t = \sigma(\mathbf{V} \cdot \mathbf{h}_{t-1} + \mathbf{W} \cdot \mathbf{x}_t + \mathbf{b}). \quad (1)$$

We drop the nonlinearity, since the inconsistency is already visible in the linear case. Letting $\mathbf{z}_t := \mathbf{W}\mathbf{x}_t$ and unfolding Eq. (1) over time obtains

$$\mathbf{h}_t = \sum_{s=1}^{t} \mathbf{V}^{t-s} \cdot \mathbf{z}_s. \quad (2)$$

The inconsistency can be seen via dot-products and via types. From the dot-product perspective, observe that multiplying an input by a matrix squared yields

$$\mathbf{V}^2\mathbf{z} = \begin{pmatrix} \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_d \end{pmatrix} \begin{pmatrix} \vdots & & \vdots \\ \mathbf{c}_1 & \cdots & \mathbf{c}_d \\ \vdots & & \vdots \end{pmatrix} \mathbf{z} = \begin{pmatrix} \left\langle (\mathbf{v}_1^\intercal \mathbf{c}_i)_{i=1}^d, \mathbf{z} \right\rangle \\ \vdots \\ \left\langle (\mathbf{v}_d^\intercal \mathbf{c}_i)_{i=1}^d, \mathbf{z} \right\rangle \end{pmatrix},$$

where $\mathbf{v}_i$ refers to rows of $\mathbf{V}$ and $\mathbf{c}_i$ to columns. Each coordinate of $\mathbf{V}^2\mathbf{z}$ is computed by measuring the similarity of a row of $\mathbf{V}$ to all of its columns, and then measuring the similarity of the result to $\mathbf{z}$. In short, features are tangled and uninterpretable.

From a type perspective, apply an SVD to $\mathbf{V} = \mathbf{P}\mathbf{D}\mathbf{Q}^\intercal$ and observe that $\mathbf{V}^2 = \mathbf{P}\mathbf{D}\mathbf{Q}^\intercal\mathbf{P}\mathbf{D}\mathbf{Q}^\intercal$. Each multiplication by $\mathbf{P}$ or $\mathbf{Q}^\intercal$ transforms the input to a new type, obtaining

$$\underbrace{\mathcal{T}_{\mathbf{h}} \xrightarrow{\mathbf{D}\mathbf{Q}^\intercal} \mathcal{T}_{\text{lat}_1} \xrightarrow{\mathbf{P}} \mathcal{T}_{\text{lat}_2}}_{\mathbf{V}} \underbrace{\xrightarrow{\mathbf{D}\mathbf{Q}^\intercal} \mathcal{T}_{\text{lat}_3} \xrightarrow{\mathbf{P}} \mathcal{T}_{\text{lat}_4}}_{\mathbf{V}}.$$

Thus $\mathbf{V}$ sends $\mathbf{z} \mapsto \mathcal{T}_{\text{lat}_2}$ whereas $\mathbf{V}^2$ sends $\mathbf{z} \mapsto \mathcal{T}_{\text{lat}_4}$. Adding terms involving $\mathbf{V}$ and $\mathbf{V}^2$, as in Eq. (2), entails adding vectors expressed in different orthogonal bases – which is analogous to adding joules to volts. The same problem applies to LSTMs and GRUs.

Two recent papers provide empirical evidence that recurrent (horizontal) connections are problematic even after gradients are stabilized: (Zaremba et al., 2015) find that Dropout performs better when restricted to vertical connections and (Laurent et al., 2015) find that Batch Normalization fails unless restricted to vertical connections (Ioffe & Szegedy, 2015). More precisely, (Laurent et al., 2015) find that Batch Normalization improves training but not test error when restricted to vertical connections; it fails completely when also applied to horizontal connections.

Code using `GOTO` can be perfectly correct, and RNNs with type mismatches can achieve outstanding performance. Nevertheless, both lead to spaghetti-like information/gradient flows that are hard to reason about.

Type-preserving transforms. One way to resolve the type inconsistency, which we do not pursue in this paper, is to use symmetric weight matrices so that $\mathbf{V} = \mathbf{P}\mathbf{D}\mathbf{P}^\intercal$ where $\mathbf{P}$ is orthogonal and $\mathbf{D} = \mathrm{diag}(\mathbf{d})$. From the dot-product perspective,

$$\mathbf{h}_t = \sum_{s=1}^{t} \mathbf{P}\mathbf{D}^{t-s}\mathbf{P}^\intercal\mathbf{z}_s,$$

which has the simple interpretation that $\mathbf{z}$ is amplified (or dampened) by $\mathbf{D}$ in the latent type provided by $\mathbf{P}$. From the type-perspective, multiplication by $\mathbf{V}^k$ is type-preserving

$$\underbrace{\mathcal{T}_{\mathbf{h}} \xrightarrow{\mathbf{P}^\intercal} \mathcal{T}_{\text{lat}_1} \xrightarrow{\mathbf{d}^k \odot \bullet} \mathcal{T}_{\text{lat}_1} \xrightarrow{\mathbf{P}} \mathcal{T}_{\mathbf{h}}}_{\mathbf{V}^k}$$

so addition is always performed in the same basis.

A familiar example of type-preserving transforms is autoencoders – under the constraint that the decoder $\mathbf{W}^\intercal$ is the transpose of the encoder $\mathbf{W}$. Finally, (Moczulski et al., 2015) propose to accelerate matrix computations in feed-forward nets by interleaving diagonal matrices, $\mathbf{A}$ and $\mathbf{D}$, with the orthogonal discrete cosine transform, $\mathbf{C}$. The resulting transform, $\mathbf{ACDC}^\intercal$, is type-preserving.

3. Recurrent Neural Networks

We present three strongly-typed RNNs that purposefully mimic classical RNNs as closely as possible. Perhaps surprisingly, the tweaks introduced below have deep structural implications, yielding architectures that are significantly easier to reason about, see sections 3.3 and 3.4.

3.1. Weakly-Typed RNNs

We first pause to note that many classical architectures are *weakly-typed*. That is, they introduce constraints or restrictions on off-diagonal operations on recurrent states.

The memory cell $\mathbf{c}$ in LSTMs is only updated coordinate-wise and is therefore well-behaved type-theoretically – although the overall architecture is not type consistent. The gating operation $\mathbf{z}_t \odot \mathbf{h}_{t-1}$ in GRUs *reduces* type-inconsistencies by discouraging (i.e. zeroing out) unnecessary recurrent information flows.

SCRNs, or Structurally Constrained Recurrent Networks (Mikolov et al., 2015), add a type-consistent state layer:

$$\mathbf{s}_t = \alpha \cdot \mathbf{s}_{t-1} + (1-\alpha) \cdot \mathbf{W}_s \mathbf{x}_t, \quad \text{where } \alpha \text{ is a scalar.}$$

In MUT1, the best performing architecture in (Jozefowicz et al., 2015), the behavior of $\mathbf{z}$ and $\mathbf{h}$ is well-typed, although the gating by $\mathbf{r}$ is not. Finally, I-RNNs initialize their recurrent connections as the identity matrix (Le et al., 2015). In other words, the key idea is a type-consistent initialization.

3.2. Strongly-Typed RNNs

The vanilla strongly-typed RNN is

$$\begin{aligned} & \mathbf{z}_t = \mathbf{W}\mathbf{x}_t && (3) \\ \text{T-RNN} \qquad & \mathbf{f}_t = \sigma(\mathbf{V}\mathbf{x}_t + \mathbf{b}) && (4) \\ & \mathbf{h}_t = \mathbf{f}_t \odot \mathbf{h}_{t-1} + (1-\mathbf{f}_t) \odot \mathbf{z}_t && (5) \end{aligned}$$

The T-RNN has similar parameters to a vanilla RNN, Eq (1), although their roles have changed. A nonlinearity for $\mathbf{z}_t$ is not necessary because: (i) gradients do not explode, corollary 2, so no squashing is needed; and (ii) coordinatewise multiplication by $\mathbf{f}_t$ introduces a nonlinearity. Whereas relus are binary gates (0 if $\mathbf{z}_t < 0$, 1 else); the forget gate $\mathbf{f}_t$ is a *continuous* multiplicative gate on $\mathbf{z}_t$.

Replacing the horizontal connection $\mathbf{V}\mathbf{h}_{t-1}$ with a vertically controlled gate, Eq. (4), stabilizes the type-structure across time steps. Line for line, the type structure is:

$$\begin{array}{ccccc} \mathcal{T}_{\mathbf{x}} & \xrightarrow{(3)} & \mathcal{T}_{\mathbf{h}} & & \\ \mathcal{T}_{\mathbf{x}} & \xrightarrow{(4)} & \mathcal{T}_{\mathbf{f}} & \xrightarrow{\text{diag}} & (\mathcal{T}_{\mathbf{h}} \to \mathcal{T}_{\mathbf{h}}) \\ \underbrace{(\mathcal{T}_{\mathbf{h}} \to \mathcal{T}_{\mathbf{h}})}_{\mathbf{f}_t} \times \underbrace{\mathcal{T}_{\mathbf{h}}}_{\mathbf{z}_t} & \xrightarrow[\mathbf{h}_{t-1}]{(5)} & \underbrace{\mathcal{T}_{\mathbf{h}}}_{\mathbf{h}_t} & & \end{array}$$

We refer to lines (3) and (4) as *learnware* since they have parameters $(\mathbf{W}, \mathbf{V}, \mathbf{b})$. Line (5) is *firmware* since it has no parameters. The firmware depends on the previous state $\mathbf{h}_{t-1}$ unlike the learnware which is stateless. See section 3.4 for more on learnware and firmware.

Strongly-typed LSTMs differ from LSTMs in two respects: (i) $\mathbf{x}_{t-1}$ is substituted for $\mathbf{h}_{t-1}$ in the first three equations so that the type structure is coherent; and (ii) the nonlinearities in $\mathbf{z}_t$ and $\mathbf{h}_t$ are removed as for the T-RNN.

$$\begin{aligned} & \mathbf{z}_t = \tau(\mathbf{V}_z \mathbf{h}_{t-1} + \mathbf{W}_z \mathbf{x}_t + \mathbf{b}_z) \\ & \mathbf{f}_t = \sigma(\mathbf{V}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t + \mathbf{b}_f) \\ \text{LSTM} \qquad & \mathbf{o}_t = \tau(\mathbf{V}_o \mathbf{h}_{t-1} + \mathbf{W}_o \mathbf{x}_t + \mathbf{b}_o) \\ & \mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + (1-\mathbf{f}_t) \odot \mathbf{z}_t \\ & \mathbf{h}_t = \tau(\mathbf{c}_t) \odot \mathbf{o}_t \end{aligned}$$

$$\begin{aligned} & \mathbf{z}_t = \mathbf{V}_z \mathbf{x}_{t-1} + \mathbf{W}_z \mathbf{x}_t + \mathbf{b}_z \\ & \mathbf{f}_t = \sigma(\mathbf{V}_f \mathbf{x}_{t-1} + \mathbf{W}_f \mathbf{x}_t + \mathbf{b}_f) \\ \text{T-LSTM} \qquad & \mathbf{o}_t = \tau(\mathbf{V}_o \mathbf{x}_{t-1} + \mathbf{W}_o \mathbf{x}_t + \mathbf{b}_o) \\ & \mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + (1-\mathbf{f}_t) \odot \mathbf{z}_t \\ & \mathbf{h}_t = \mathbf{c}_t \odot \mathbf{o}_t \end{aligned}$$

We drop the input gate from the updates for simplicity; see (Greff et al., 2015). The type structure is

$$\begin{array}{ccccc} \mathcal{T}_{\mathbf{x}} & \longrightarrow & \mathcal{T}_{\mathbf{c}} & & \\ \mathcal{T}_{\mathbf{x}} & \longrightarrow & \mathcal{T}_{\mathbf{f}} & \xrightarrow{\text{diag}} & (\mathcal{T}_{\mathbf{c}} \to \mathcal{T}_{\mathbf{c}}) \\ \mathcal{T}_{\mathbf{x}} & \longrightarrow & \mathcal{T}_{\mathbf{h}} & & \\ (\mathcal{T}_{\mathbf{c}} \to \mathcal{T}_{\mathbf{c}}) \times \mathcal{T}_{\mathbf{c}} & \xrightarrow[\mathbf{c}_{t-1}]{} & \mathcal{T}_{\mathbf{c}} & \xrightarrow{\text{diag}} & (\mathcal{T}_{\mathbf{h}} \to \mathcal{T}_{\mathbf{h}}) \\ (\mathcal{T}_{\mathbf{h}} \to \mathcal{T}_{\mathbf{h}}) \times \mathcal{T}_{\mathbf{h}} & \longrightarrow & \mathcal{T}_{\mathbf{h}} & & \end{array}$$

Strongly-typed GRUs adapt GRUs similarly to how LSTMs were modified. In addition, the reset gate $\mathbf{z}_t$ is repurposed; it is no longer needed for weak-typing.

$$\begin{aligned}
&\mathbf{z}_t = \sigma(\mathbf{V}_z\mathbf{h}_{t-1} + \mathbf{W}_z\mathbf{x}_t + \mathbf{b}_z)\\
\text{GRU}\qquad &\mathbf{f}_t = \sigma(\mathbf{V}_f\mathbf{h}_{t-1} + \mathbf{W}_f\mathbf{x}_t + \mathbf{b}_f)\\
&\mathbf{o}_t = \tau\big(\mathbf{V}_o(\mathbf{z}_t \odot \mathbf{h}_{t-1}) + \mathbf{W}_o\mathbf{x}_t + \mathbf{b}_o\big)\\
&\mathbf{h}_t = \mathbf{f}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{f}_t) \odot \mathbf{o}_t
\end{aligned}$$

$$\begin{aligned}
&\mathbf{z}_t = \mathbf{V}_z\mathbf{x}_{t-1} + \mathbf{W}_z\mathbf{x}_t + \mathbf{b}_z\\
\text{T-GRU}\qquad &\mathbf{f}_t = \sigma(\mathbf{V}_f\mathbf{x}_{t-1} + \mathbf{W}_f\mathbf{x}_t + \mathbf{b}_f)\\
&\mathbf{o}_t = \tau(\mathbf{V}_o\mathbf{x}_{t-1} + \mathbf{W}_o\mathbf{x}_t + \mathbf{b}_o)\\
&\mathbf{h}_t = \mathbf{f}_t \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \mathbf{o}_t
\end{aligned}$$

The type structure is

$$\begin{array}{ccccc}
\mathcal{T}_{\mathbf{x}} & \longrightarrow & \mathcal{T}_{\mathbf{h}} & & \\
\mathcal{T}_{\mathbf{x}} & \longrightarrow & \mathcal{T}_{\mathbf{f}} & \xrightarrow{\text{diag}} & (\mathcal{T}_{\mathbf{h}} \to \mathcal{T}_{\mathbf{h}})\\
\mathcal{T}_{\mathbf{x}} & \longrightarrow & \mathcal{T}_{\mathbf{o}} & \xrightarrow{\text{diag}} & (\mathcal{T}_{\mathbf{h}} \to \mathcal{T}_{\mathbf{h}})\\
(\mathcal{T}_{\mathbf{h}} \to \mathcal{T}_{\mathbf{h}}) & \times & (\mathcal{T}_{\mathbf{h}} \to \mathcal{T}_{\mathbf{h}}) \times \mathcal{T}_{\mathbf{h}} & \xrightarrow[\mathbf{h}_{t-1}]{} & \mathcal{T}_{\mathbf{h}}
\end{array}$$

3.3. Feature Semantics

The output of a vanilla RNN expands as the uninterpretable

$$\mathbf{h}_t = \sigma(\mathbf{V}\sigma(\mathbf{V}\sigma(\cdots) + \mathbf{W}\mathbf{x}_{t-1} + \mathbf{b}) + \mathbf{W}\mathbf{x}_t + \mathbf{b}),$$

with even less interpretable gradient. Similar considerations hold for LSTMs and GRUs. Fortunately, the situation is more amenable for strongly-typed architectures. In fact, their semantics are related to *average-pooled convolutions*.

Convolutions. Applying a one-dimensional convolution to input sequence $\mathbf{x}[t]$ yields output sequence

$$\mathbf{z}[t] = (\mathbf{W} * \mathbf{x})[t] = \sum_s \mathbf{W}[s] \cdot \mathbf{x}[t-s]$$

Given weights f_s associated with $\mathbf{W}[s]$, average-pooling yields $\mathbf{h}_t = \sum_{s=1}^{t} f_s \cdot \mathbf{z}[s]$. A special case is when the convolution applies the same matrix to every input:

$$\mathbf{W}[s] = \begin{cases} \mathbf{W} & \text{if } s = 0\\ 0 & \text{else.}\end{cases}$$

The average-pooled convolution is then a weighted average of the features extracted from the input sequence.

Dynamic temporal convolutions. We now show that strongly-typed RNNs are one-dimensional temporal convolutions with *dynamic* average-pooling. Informally, strongly-typed RNNs transform input sequences into a weighted average of features extracted from the sequence

$$\mathbf{x}_{1:t} \mapsto \mathop{\mathbb{E}}_{\mathbb{P}_{\mathbf{x}_{1:t}}}\Big[\mathbf{W} * \mathbf{x}\Big] = \sum_{s=1}^{t} \mathbb{P}_{\mathbf{x}_{1:t}}(s) \cdot (\mathbf{W} \cdot \mathbf{x}_s) =: \mathbf{h}[t]$$

where the weights depends on the sequence. In detail:

Theorem 1 (feature semantics via dynamic convolutions)**.** *Strongly-typed features are computed explicitly as follows.*

- T-RNN. *The output is* $\mathbf{h}_t = \mathbb{E}_{s\sim\mathbb{P}_{\mathbf{x}_{1:t}}}\big[\mathbf{W}\mathbf{x}_s\big]$ *where*

$$\mathbb{P}_{\mathbf{x}_{1:t}}(s) = \begin{cases} 1 - \mathbf{f}_t & \text{if } s = t\\ \mathbf{f}_t \odot \mathbb{P}_{\mathbf{x}_{1:t-1}}(s) & \text{else.}\end{cases}$$

- T-LSTM. *Let* $\mathbf{U}_\bullet := [\mathbf{V}_\bullet; \mathbf{W}_\bullet; \mathbf{b}_\bullet]$ *and* $\tilde{\mathbf{x}}_t := [\mathbf{x}_{t-1}; \mathbf{x}_t; 1]$ *denote the vertical concatenation of the weight matrices and input vectors respectively. Then,*

$$\mathbf{h}_t = \tau\big(\mathbf{U}_o\tilde{\mathbf{x}}_t\big) \odot \mathop{\mathbb{E}}_{s\sim\mathbb{P}_{\mathbf{x}_{1:t}}}\big[\mathbf{U}_z\tilde{\mathbf{x}}_s\big]$$

  where $\mathbb{P}_{\mathbf{x}_{1:t}}$ *is defined as above.*

- T-GRU. *Using the notation above,*

$$\mathbf{h}_t = \sum_{s=1}^{t} \mathbf{F}_s \odot \Big(\tau\big(\mathbf{U}_o\tilde{\mathbf{x}}_s\big) \odot \mathbf{U}_z\tilde{\mathbf{x}}_s\Big)$$

  where

$$\mathbf{F}_s = \begin{cases} 1 & \text{if } s = t\\ \mathbf{f}_s \odot \mathbf{F}_{s+1} & \text{else.}\end{cases}$$

Proof. Direct computation. □

In summary, T-RNNs compute a dynamic distribution over time steps, and then compute the expected feedforward features over that distribution. T-LSTMs store expectations in private memory cells that are reweighted by the output gate when publicly broadcast. Finally, T-GRUs drop the requirement that the average is an expectation, and also incorporate the output gate into the memory updates.

Strongly-typed gradients are straightforward to compute and interpret:

Corollary 2 (gradient semantics)**.** *The strongly-typed gradients are*

- *T-RNN:*

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}} = \mathop{\mathbb{E}}_{s\sim\mathbb{P}_{\mathbf{x}_{1:t}}}\Big[\frac{\partial}{\partial \mathbf{W}}(\mathbf{z}_s)\Big]$$
$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{V}} = \mathop{\mathbb{E}}_{s\sim\mathbb{P}_{\mathbf{x}_{1:t}}}\Big[\mathbf{z}_s \odot \frac{\partial}{\partial \mathbf{V}}\big(\log \mathbb{P}_{\mathbf{x}_{1:t}}(s)\big)\Big]$$

  and similarly for $\frac{\partial}{\partial \mathbf{b}}$.

- *T-LSTM:*

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{U}_o} = \frac{\partial}{\partial \mathbf{U}_o}(\mathbf{o}_t) \odot \mathop{\mathbb{E}}_{s\sim\mathbb{P}_{\mathbf{x}_{1:t}}}\Big[\mathbf{z}_s\Big]$$
$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{U}_z} = \mathbf{o}_t \odot \mathop{\mathbb{E}}_{s\sim\mathbb{P}_{\mathbf{x}_{1:t}}}\Big[\frac{\partial}{\partial \mathbf{U}_z}(\mathbf{z}_s)\Big]$$
$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{U}_f} = \mathbf{o}_t \odot \mathop{\mathbb{E}}_{s\sim\mathbb{P}_{\mathbf{x}_{1:t}}}\Big[\mathbf{z}_s \odot \frac{\partial}{\partial \mathbf{U}_f}\big(\log \mathbb{P}_{\mathbf{x}_{1:t}}(s)\big)\Big]$$

- *T-GRU:*

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{U}_o} = \sum_{s=1}^{t} \mathbf{F}_s \odot \frac{\partial}{\partial \mathbf{U}_o}(\mathbf{o}_s) \odot \mathbf{z}_s$$
$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{U}_z} = \sum_{s=1}^{t} \mathbf{F}_s \odot \mathbf{o}_s \odot \frac{\partial}{\partial \mathbf{U}_z}(\mathbf{z}_s)$$
$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{U}_f} = \sum_{s=1}^{t} \frac{\partial}{\partial \mathbf{U}_f}\big(\mathbf{F}_s\big) \odot \mathbf{o}_s \odot \mathbf{z}_s$$

It follows immediately that gradients will not explode for T-RNNs or LSTMs. Empirically we find they also behave well for T-GRUs.

3.4. Feature Algebra

A vanilla RNN can approximate any continuous state update $\mathbf{h}_t = g(\mathbf{x}_t, \mathbf{h}_{t-1})$ since $\text{span}\{s(\mathbf{w}^\intercal \mathbf{x}) \,|\, \mathbf{w} \in \mathbb{R}^d\}$ is dense in continuous functions $\mathcal{C}(\mathbb{R}^d)$ on $\mathbb{R}^d$ if s is a non-polynomial nonlinear function (Leshno et al., 1993). It follows that vanilla RNNs can approximate any recursively computable partial function (Siegelmann & Sontag, 1995).

Strongly-typed RNNs are more constrained. We show the constraints reflect a coherent design-philosophy and are less severe than appears.

The learnware / firmware distinction. Strongly-typed architectures factorize into stateless *learnware* and state-dependent *firmware*. For example, T-LSTMs and T-GRUs factorize[2] as

$$(\mathbf{f}_t, \mathbf{z}_t, \mathbf{o}_t) = \text{T-LSTM}^{\text{learn}}_{\{\mathbf{V}_\bullet, \mathbf{W}_\bullet, \mathbf{b}_\bullet\}}(\mathbf{x}_{t-1}, \mathbf{x}_t)$$
$$(\mathbf{h}_t, \mathbf{c}_t) = \text{T-LSTM}^{\text{firm}}(\mathbf{f}_t, \mathbf{z}_t, \mathbf{o}_t; \underbrace{\mathbf{c}_{t-1}}_{\text{state}})$$
$$(\mathbf{f}_t, \mathbf{z}_t, \mathbf{o}_t) = \text{T-GRU}^{\text{learn}}_{\{\mathbf{V}_\bullet, \mathbf{W}_\bullet, \mathbf{b}_\bullet\}}(\mathbf{x}_{t-1}, \mathbf{x}_t)$$
$$\mathbf{h}_t = \text{T-GRU}^{\text{firm}}(\mathbf{f}_t, \mathbf{z}_t, \mathbf{o}_t; \underbrace{\mathbf{h}_{t-1}}_{\text{state}}).$$

[2] A superficially similar factorization holds for GRUs and LSTMs. However, their learnware is *state-dependent*, since $(\mathbf{f}_t, \mathbf{z}_t, \mathbf{o}_t)$ depend on $\mathbf{h}_{t-1}$.

Firmware decomposes coordinatewise, which prevents side-effects from interacting: e.g. for T-GRUs

$$\text{T-GRU}^{\text{firm}}(\mathbf{f}, \mathbf{z}, \mathbf{o}; \mathbf{h}) = \Big(\varphi(f^{(i)}, z^{(i)}, o^{(i)}; h^{(i)})\Big)_{i=1}^{d}$$
$$\text{where } \varphi(f, z, o; h) = fh + zo$$

and similarly for T-LSTMs. Learnware is stateless; it has no side-effects and does not decompose coordinatewise. Evidence that side-effects are a problem for LSTMs can be found in (Zaremba et al., 2015) and (Laurent et al., 2015), which show that Dropout and Batch Normalization respectively need to be restricted to vertical connections.

In short, under strong-typing the learnware carves out features which the firmware uses to perform *coordinatewise* state updates $h^i_t = g(h^i_{t-1}, z^i_{t-1})$. Vanilla RNNs allow *arbitrary* state updates $\mathbf{h}_t = g(\mathbf{h}_{t-1}, \mathbf{x}_t)$. LSTMs and GRUs restrict state updates, but allow arbitrary functions of the state. Translated from a continuous to discrete setting, the distinction between strongly-typed and classical architectures is analogous to working with binary logic gates (AND, OR) on variables $\mathbf{z}_t$ learned by the vertical connections – versus working directly with n-ary boolean operations.

Representational power. Motivated by the above, we show that a minimal strongly-typed architecture can span the space of continuous *binary* functions on features.

Theorem 3 (approximating binary functions)**.** *The strongly-typed* minimal *RNN with updates*

$$\textit{T-MR:} \quad \mathbf{h}_t = \rho(\mathbf{b} \odot \mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t + \mathbf{c})$$

and parameters $(\mathbf{b}, \mathbf{c}, \mathbf{W})$ *can approximate any set of continuous binary functions on features.*

Proof sketch. Let $z = \mathbf{w}^\intercal \mathbf{x}$ be a feature of interest. Combining (Leshno et al., 1993) with the observation that $a\rho(bh + z + c) = \rho(abh + az + ac)$ for $a > 0$ implies that $\overline{\text{span}}\{\rho(b \cdot h_{t-1} + z_t) \,|\, b, c \in \mathbb{R}\} = \mathcal{C}(\mathbb{R}^2)$. As many weighted copies az of z as necessary are obtained by adding rows to $\mathbf{W}$ that are scalar multiples of $\mathbf{w}$.

Any set of binary functions on any collection of features can thus be approximated. Finally, vertical connections can approximate any set of features (Leshno et al., 1993). □

4. Experiments

We investigated the empirical performance of strongly-typed recurrent nets for sequence learning. The performance was evaluated on character-level and word-level text generation. We conducted a set of proof-of-concept experiments. The goal is not to compete with previous work or

Table 1. The **(train, test) cross-entropy loss** of RNNs and T-RNNs on WP dataset.

Model	vanilla RNN			T-RNN		
Layers	1	2	3	1	2	3
64 (no dropout)	(1.365, 1.435)	(1.347, 1.417)	(1.353, 1.423)	(1.371, 1.452)	(**1.323**, 1.409)	(1.342, 1.423)
256	(1.215, 1.274)	(1.242, **1.254**)	(1.257, 1.273)	(1.300, 1.398)	(1.251, 1.276)	(1.233, 1.266)

Table 2. The **(train, test) cross-entropy loss** of LSTMs and T-LSTMs on WP dataset.

Model	LSTM			T-LSTM		
Layers	1	2	3	1	2	3
64 (no dropout)	(1.496, 1.560)	(1.485, 1.557)	(1.500, 1.563)	(1.462, 1.511)	(**1.367**, 1.432)	(1.369, 1.434)
256	(1.237, 1.251)	(1.098, 1.193)	(1.185, 1.213)	(1.254, 1.273)	(1.045, **1.189**)	(1.167, 1.198)

Table 3. The **(train, test) cross-entropy loss** of GRUs and T-GRUs on WP dataset.

Model	GRU			T-GRU		
Layers	1	2	3	1	2	3
64 (no dropout)	(1.349, 1.435)	(1.432, 1.503)	(1.445, 1.559)	(1.518 ,1.569)	(**1.337**, 1.422)	(1.377, 1.436)
256	(1.083, 1.226)	(1.163, 1.214)	(1.219, 1.227)	(1.142, 1.296)	(1.208, 1.240)	(1.216, **1.212**)

to find the best performing model under a specific hyper-parameter setting. Rather, we investigate how the two classes of architectures perform over a range of settings.

4.1. Character-level Text Generation

The first task is to generate text from a sequence of characters by predicting the next character in a sequence. We used Leo Tolstoy's *War and Peace* (WP) which consists of 3,258,246 characters of English text, split into train/val/test sets with 80/10/10 ratios. The characters are encoded into K-dimensional *one-hot* vectors, where K is the size of the vocabulary. We follow the experimental setting proposed in (Karpathy et al., 2015). Results are reported for two configurations: "64" and "256", which are models with the same number of parameters as a 1-layer LSTM with 64 and 256 cells per layer respectively. Dropout regularization was only applied to the "256" models. The dropout rate was taken from $\{0.1, 0.2\}$ based on validation performance. Tables 2 and 3 summarize the performance in terms of cross-entropy loss $H(\mathbf{y}, \mathbf{p}) = \sum_{i=1}^{K} y_i \log p_i$.

We observe that the training error of strongly-typed models is typically lower than that of the standard models for ≥ 2 layers. The test error of the two architectures are comparable. However, our results (for both classical and typed models) fail to match those reported in (Karpathy et al., 2015), where a more extensive parameter search was performed.

4.2. Word-level Text Generation

The second task was to generate word-level text by predicting the next word from a sequence. We used the Penn Treebank (PTB) dataset (Marcus et al., 1993), which consists of 929K training words, 73K validation words, and 82K test words, with vocabulary size of 10K words. The PTB dataset is publicly available on web.[3]

Table 4. Perplexity on the Penn Treebank dataset.

Model	Train	Validation	Test
small, no dropout			
vanilla RNN	416.50	442.31	432.01
T-RNN	58.66	172.47	169.33
LSTM	36.72	122.47	**117.25**
T-LSTM	**28.15**	215.71	200.39
GRU	31.14	179.47	173. 27
T-GRU	28.57	207.94	195.82
medium, with dropout			
LSTM (Zaremba et al., 2015)	**48.45**	86.16	82.70
LSTM (3-layer)	71.76	98.22	97.87
T-LSTM	50.21	87.36	82.71
T-LSTM (3-layer)	51.45	85.98	**81.52**
GRU	65.80	97.24	93.44
T-GRU	55.31	121.39	113.85

We followed the experimental setting in (Zaremba et al., 2015) and compared the performance of "small" and "medium" models. The parameter size of "small" models is equivalent to that of 2 layers of 200-cell LSTMs, while the parameter size of "medium" models is the same as that of 2 layers of 650-cell LSTMs. For the "medium" models, we selected the dropout rate from $\{0.4, 0.5, 0.6\}$ according to

[3] `http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz`

validation performance. Single run performance, measured via *perplexity*, i.e., $\exp(H(\mathbf{y}, \mathbf{p}))$, are reported in Table 4.

Perplexity. For the "small" models, we found that the training perplexity of strongly-typed models is consistently lower than their classical counterparts, in line with the result for War & Peace. Test error was significantly worse for the strongly-typed architectures. A possible explanation for both observations is that strongly-typed architectures require more extensive regularization.

An intriguing result is that the T-RNN performs in the same ballpark as LSTMs, with perplexity within a factor of two. By contrast, the vanilla RNN fails to achieve competitive performance. This suggests there may be strongly-typed architectures of intermediate complexity between RNNs and LSTMs with comparable performance to LSTMs.

The dropout-regularized "medium" T-LSTM matches the LSTM performance reported in (Zaremba et al., 2015). The 3-layer T-LSTM obtains slightly better performance. The results were obtained with almost identical parameters to Zaremba et al (the learning rate decay was altered), suggesting that T-LSTMs are viable alternatives to LSTMs for sequence learning tasks when properly regularized. Strongly-typed GRUs did not match the performance of GRUs, possibly due to insufficient regularization.

Gradients. We investigated the effect of removing gradient clipping on medium-sized LSTM and T-LSTM. T-LSTM gradients are well-behaved without clipping, although test performance is not competitive. In contrast, LSTM gradients explode without clipping and the architecture is unusable. It is possible that carefully initialized T-LSTMs may be competitive without clipping. We defer the question to future work.

Runtime. Since strongly-typed RNNs have fewer nonlinearities than standard RNNs, we expect that they should have lower computational complexity. Training on the PTB dataset on an NVIDIA GTX 980 GPU, we found that T-LSTM is on average $\sim 1.6\times$ faster than LSTM. Similarly, the T-GRU trains on average $\sim 1.4\times$ faster than GRU.

5. Conclusions

RNNs are increasingly important tools for speech recognition, natural language processing and other sequential learning problems. The complicated structure of LSTMs and GRUs has led to searches for simpler alternatives with limited success (Bayer et al., 2009; Greff et al., 2015; Jozefowicz et al., 2015; Le et al., 2015; Mikolov et al., 2015). This paper introduces strong-typing as a tool to guide the search for alternate architectures. In particular, we suggest searching for update equations that learn well-behaved features, rather than update equations that "appear simple". We draw on two disparate intuitions that turn out to be surprisingly compatible: (i) that neural networks are analogous to measuring devices (Balduzzi, 2012) and (ii) that training an RNN is analogous to writing code.

The main contribution is a new definition of type that is closely related to singular value decomposition – and is thus well-suited to deep learning. It turns out that classical RNNs are badly behaved from a type-perspective, which motivates modifying the architectures. Section 3 tweaked LSTMs and GRUs to make them well-behaved from a typing and functional programming perspective, yielding features and gradients that are easier to reason about than classical architectures.

Strong-typing has implications for the depth of RNNs. It was pointed out in (Pascanu et al., 2014) that unfolding horizontal connections over time implies the concept of depth is not straightforward in classical RNNs. By contrast, depth has the same meaning in strongly-typed architectures as in feedforward nets, since vertical connections learn features and horizontal connections act coordinatewise.

Experiments in section 4 show that strongly-typed RNNs achieve comparable generalization performance to classical architectures when regularized with dropout and have consistently lower training error. It is important to emphasize that the experiments are not conclusive. Firstly, we did not deviate far from settings optimized for classical RNNs when training strongly-typed RNNs. Secondly, the architectures were chosen to be as close as possible to classical RNNs. A more thorough exploration of the space of strongly-typed nets may yield better results.

Towards machine reasoning. A definition of machine reasoning, adapted from (Bottou, 2014), is "algebraically manipulating features to answer a question". Hard-won experience in physics (Chang, 2004), software engineering (Dijkstra, 1968), and other fields has led to the conclusion that well-chosen constraints are crucial to effective reasoning. Indeed, neural Turing machines (Graves et al., 2014) are harder to train than more constrained architectures such as neural queues and deques (Grefenstette et al., 2015).

Strongly-typed features have a consistent semantics, theorem 1, unlike features in classical RNNs which are rotated across time steps – and are consequently difficult to reason about. We hypothesize that strong-typing will provide a solid foundation for algebraic operations on learned features. Strong-typing may then provide a useful organizing principle in future machine reasoning systems.

Acknowledgements. We thank Tony Butler-Yeoman, Marcus Frean, Theofanis Karaletsos, JP Lewis and Brian McWilliams for useful comments and discussions.

References

Balduzzi, D. On the information-theoretic structure of distributed measurements. *Elect. Proc. in Theor. Comp. Sci.*, 88:28–42, 2012.

Bayer, Justin, Wierstra, Daan, Togelius, Julian, and Schmidhuber, Juergen. Evolving memory cell structures for sequence learning. In *ICANN*, 2009.

Bengio, Yoshua, Simard, P, and Frasconi, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neur. Net.*, 5(2):157–166, 1994.

Bottou, Léon. From machine learning to machine reasoning: An essay. *Machine Learning*, 94:133–149, 2014.

Bridgman, P W. *Dimensional analysis*. Yale University Press, 1922.

Chang, Hasok. *Inventing Temperature: Measurement and Scientific Progress*. Oxford University Press, 2004.

Cho, Kyunghyun, van Merriënboer, Bart, Gulcehre, Caglar, Bahdanau, Dzmitry, Bougares, Fethi, Schwenk, Holger, and Bengio, Yoshua. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *EMNLP*, 2014.

Dijkstra, Edsger W. Go To Statement Considered Harmful. *Comm. ACM*, 11(3):147–148, 1968.

Girard, Jean-Yves. *Proofs and Types*. Cambridge University Press, 1989.

Graves, Alex, Mohamed, A, and Hinton, GE. Speech recognition with deep recurrent neural networks. In *ICASSP*, 2013.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural Turing Machines. In *arXiv:1410.5401*, 2014.

Grefenstette, Edward, Hermann, Karl Moritz, Suleyman, Mustafa, and Blunsom, Phil. Learning to Transduce with Unbounded Memory. In *Adv in Neural Information Processing Systems (NIPS)*, 2015.

Greff, Klaus, Srivastava, Rupesh Kumar, Koutník, Jan, Steunebrink, Bas R, and Schmidhuber, Juergen. LSTM: A Search Space Odyssey. In *arXiv:1503.04069*, 2015.

Gregor, Karol, Danihelka, Ivo, Graves, Alex, Rezende, Danilo Jimenez, and Wierstra, Daan. DRAW: A Recurrent Neural Network For Image Generation. In *ICML*, 2015.

Hart, George W. *Multidimensional Analysis: Algebras and Systems for Science and Engineering*. Springer, 1995.

Hochreiter, S and Schmidhuber, J. Long Short-Term Memory. *Neural Comp*, 9:1735–1780, 1997.

Hochreiter, Sepp. Untersuchungen zu dynamischen neuronalen Netzen. Master's thesis, Technische Universität München, 1991.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *arXiv:1502.03167*, 2015.

Jozefowicz, Rafal, Zaremba, Wojciech, and Sutskever, Ilya. An Empirical Exploration of Recurrent Network Architectures. In *ICML*, 2015.

Karpathy, Andrej and Fei-Fei, Li. Deep Visual-Semantic Alignments for Generating Image Descriptions. In *CVPR*, 2015.

Karpathy, Andrej, Johnson, Justin, and Fei-Fei, Li. Visualizing and understanding recurrent neural networks. In *arXiv:1506.02078*, 2015.

Laurent, C, Pereyra, G, Brakel, P, Zhang, Y, and Bengio, Yoshua. Batch Normalized Recurrent Neural Networks. In *arXiv:1510.01378*, 2015.

Le, Quoc, Jaitly, Navdeep, and Hinton, Geoffrey. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. In *arXiv:1504.00941*, 2015.

Leshno, Moshe, Lin, Vladimir Ya., Pinkus, Allan, and Schocken, Shimon. Multilayer Feedforward Networks With a Nonpolynomial Activation Function Can Approximate Any Function. *Neural Networks*, 6:861–867, 1993.

Macedo, Hugo Daniel and Oliveira, José Nuno. Typing linear algebra: A biproduct-oriented approach. *Science of Computer Programming*, 78(11):2160 – 2191, 2013.

Marcus, Mitchell P., Marcinkiewics, Mary Ann, and Santorini, Beatrice. Building a large annotated corpus of english: The penn treebank. *Comp. Linguistics*, 19(2):313–330, 1993.

Mikolov, Tomas, Joulin, Armand, Chopra, Sumit, Mathieu, Michael, and Ranzato, Marc'Aurelio. Learning Longer Memory in Recurrent Neural Networks. In *ICLR*, 2015.

Moczulski, Marin, Denil, Misha, Appleyard, Jeremy, and de Freitas, Nando. ACDC: A Structured Efficient Linear Layer. In *arXiv:1511.05946*, 2015.

Olah, Christopher. Neural Networks, Types, and Functional Programming, 2015. URL `http://colah.github.io/posts/2015-09-NN-Types-FP/`.

Pascanu, Razvan, Mikolov, Tomas, and Bengio, Yoshua. On the difficulty of training recurrent neural networks. In *ICML*, 2013.

Pascanu, Razvan, Gulcehre, Caglar, Cho, Kyunghyun, and Bengio, Yoshua. How to Construct Deep Recurrent Networks. In *ICLR*, 2014.

Pierce, Benjamin C. *Types and Programming Languages*. MIT Press, 2002.

Reynolds, J C. Towards a theory of type structure. In *Paris colloquium on programming*, volume 19 of *LNCS*. Springer, 1974.

Siegelmann, Hava and Sontag, Eduardo. On the Computational Power of Neural Nets. *Journal of Computer and System Sciences*, 50:132–150, 1995.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc. Sequence to Sequence Learning with Neural Networks. In *Adv in Neural Information Processing Systems (NIPS)*, 2014.

Vinyals, Oriol, Toshev, Alexander, Bengio, Samy, and Erhan, Dumitru. Show and tell: A neural image caption generator. In *CVPR*, 2015.

Zaremba, Wojciech, Sutskever, Ilya, and Vinyals, Oriol. Recurrent Neural Network Regularization. In *arXiv:1409.2329*, 2015.