Approximating Stacked and Bidirectional Recurrent Architectures with the Delayed Recurrent Neural Network

Javier S. Turek [1] **Shailee Jain** [2] **Vy A. Vo** [1] **Mihai Capotă** [1] **Alexander G. Huth** [2 3] **Theodore L. Willke** [1]

Abstract

Recent work has shown that topological enhancements to recurrent neural networks (RNNs) can increase their expressiveness and representational capacity. Two popular enhancements are stacked RNNs, which increases the capacity for learning non-linear functions, and bidirectional processing, which exploits acausal information in a sequence. In this work, we explore the delayed-RNN, which is a single-layer RNN that has a delay between the input and output. We prove that a weight-constrained version of the delayed-RNN is equivalent to a stacked-RNN. We also show that the delay gives rise to partial acausality, much like bidirectional networks. Synthetic experiments confirm that the delayed-RNN can mimic bidirectional networks, solving some acausal tasks similarly, and outperforming them in others. Moreover, we show similar performance to bidirectional networks in a real-world natural language processing task. These results suggest that delayed-RNNs can approximate topologies including stacked RNNs, bidirectional RNNs, and stacked bidirectional RNNs – but with equivalent or faster runtimes for the delayed-RNNs.

1. Introduction

Recurrent neural networks (RNN) have successfully been used for sequential tasks like language modeling (Sutskever et al., 2011), machine translation (Sutskever et al., 2014), and speech recognition (Amodei et al., 2016). They approximate complex, non-linear temporal relationships by maintaining and updating an internal state for every input element. However, they face several challenges while modeling long-term dependencies, motivating work on variant architectures.

Firstly, due to the long credit assignment paths in RNNs, the gradients might vanish or explode (Bengio et al., 1994). This has led to gated variants like the Long Short-term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) that can retain information over long timescales. Secondly, it is well known that deeper networks can more efficiently approximate a broader range of functions (Bengio et al., 2007; Bianchini & Scarselli, 2014). While RNNs are deep *in time*, they are limited in the number of non-linearities applied to recent inputs.

To increase depth, there has been extensive work on *stacking* RNNs into multiple layers (Schmidhuber, 1992; Bengio, 2009). In vanilla stacked RNNs, each layer applies a non-linearity and passes information to the next layer, while also maintaining a recurrent connection to itself. To effectively propagate gradients across the hierarchy, skip or shortcut connections can be used (Raiko et al., 2012; Graves, 2013; Campos et al., 2018). Alternatives like recurrent highway networks (Zilly et al., 2017) introduce non-linearities between timesteps through “micro-ticks" (Graves, 2016). Pascanu et al. (2014) increase depth by adding feedforward layers between state-to-state transitions. Gated feedback networks (Chung et al., 2015) allow for layer-to-layer interactions between adjacent timesteps. All these variants thus introduce topological modifications to retain information over longer timescales and model hierarchical temporal dependencies.

Another development is the bidirectional RNN (Bi-RNN) (Schuster & Paliwal, 1997; Graves & Schmidhuber, 2005). While RNNs are inherently causal, Bi-RNNs model acausal interactions by processing sequences in both forward and backward directions. They achieve state-of-the-art performance on parts-of-speech tagging (Plank et al., 2016) and sentiment analysis (Baziotis et al., 2017), demonstrating that some natural language processing (NLP) tasks benefit greatly from combining past and future inputs.

The successes of these RNN architectural variants seem to derive from two common properties: depth and acausality.

[1]Intel Labs, Hillsboro, Oregon, USA [2]Department of Computer Science, The University of Texas at Austin, Austin, Texas, USA [3]Department of Neuroscience, The University of Texas at Austin, Austin, Texas, USA. Correspondence to: Javier S. Turek <javier.turek@intel.com>.

Proceedings of the 37^{th} International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020. Copyright 2020 by the author(s).

In this paper we investigate the **delayed-recurrent neural network (d-RNN)**, an extremely simple variant that adds both depth and acausality to the RNN. The d-RNN is a single-layer RNN that imposes depth in time by delaying the output of the model. We analyze the d-RNN and prove that when it is constrained with sparse weights, the model is equivalent to a stacked RNN. Further, noting that the delay introduces acausal processing, we use a d-RNN to approximate bidirectional recurrent networks. We show empirically that a d-RNN has the capability to solve some tasks similarly to stacked and bidirectional RNNs, and outperform them in others. Additionally, we show that even if the d-RNN approximation carries some error, this model can provide much faster runtimes than alternatives.

2. Background

Given a sequential input $\{\mathbf{x}_t\}_{t=1\dots T}, \mathbf{x}_t \in \mathbb{R}^q$, a single-layer RNN is defined by:

$$\hat{\mathbf{h}}_t = f\left(\hat{\mathbf{W}}_{\mathbf{x}}\mathbf{x}_t + \hat{\mathbf{W}}_{\mathbf{h}}\hat{\mathbf{h}}_{t-1} + \hat{\mathbf{b}}_{\mathbf{h}}\right), \tag{1}$$

$$\hat{\mathbf{y}}_t = g\left(\hat{\mathbf{W}}_{\mathbf{o}}\hat{\mathbf{h}}_t + \hat{\mathbf{b}}_{\mathbf{o}}\right), \tag{2}$$

where $f(\cdot)$ and $g(\cdot)$ are element-wise activation function such as $\tanh$ and $\mathrm{softmax}$, $\hat{\mathbf{h}}_t \in \mathbb{R}^n$ is the hidden state at timestep t with n units, and $\hat{\mathbf{y}}_t \in \mathbb{R}^m$ is the network output. Learned parameters include input weights $\hat{\mathbf{W}}_{\mathbf{x}}$, recurrent weights $\hat{\mathbf{W}}_{\mathbf{h}}$, bias term $\hat{\mathbf{b}}_{\mathbf{h}}$, output weights $\hat{\mathbf{W}}_{\mathbf{o}}$, and bias term $\hat{\mathbf{b}}_{\mathbf{o}}$. The initial hidden state is denoted $\hat{\mathbf{h}}_0$.

Stacked recurrent units are typically used to provide depth in RNNs (Schmidhuber, 1992; Bengio, 2009). Based on Eq. (1) and (2), a stacked RNN with k layers is given by:

$$\mathbf{h}_t^{(1)} = f\left(\mathbf{W}_{\mathbf{x}}^{(1)}\mathbf{x}_t + \mathbf{W}_{\mathbf{h}}^{(1)}\mathbf{h}_{t-1}^{(1)} + \mathbf{b}_{\mathbf{h}}^{(1)}\right),\ i = 1 \tag{3}$$

$$\mathbf{h}_t^{(i)} = f\left(\mathbf{W}_{\mathbf{x}}^{(i)}\mathbf{h}_t^{(i-1)} + \mathbf{W}_{\mathbf{h}}^{(i)}\mathbf{h}_{t-1}^{(i)} + \mathbf{b}_{\mathbf{h}}^{(i)}\right),\ i = 2\dots k \tag{4}$$

$$\mathbf{y}_t = g\left(\mathbf{W}_{\mathbf{o}}\mathbf{h}_t^{(k)} + \mathbf{b}_{\mathbf{o}}\right), \tag{5}$$

where the activation function and parameterization follow the single-layer RNN. Separate weights and bias terms for each layer i are given by $\mathbf{W}_{\mathbf{x}}^{(i)}$, $\mathbf{W}_{\mathbf{h}}^{(i)}$, and $\mathbf{b}_{\mathbf{h}}^{(i)}$. The hidden state for this layer at timestep t is $\mathbf{h}_t^{(i)}$. The stacked RNN has initial hidden state vectors $\mathbf{h}_0^{(1)} \dots \mathbf{h}_0^{(k)}$ corresponding to the k layers. The hat operator is used for vectors and matrices in the single-layer RNN, while those without are for the stacked RNN.

3. Delayed-Recurrent Neural Network

One way to increase depth in RNNs is to stack recurrent layers, as suggested above. An alternative is to consider time as a means to increase depth within a single-layer RNN. However, single-layer RNNs are limited in the number of non-linearities applied to recent inputs: there is a single non-linearity between the most recent input $\mathbf{x}_t$ and its respective output $\hat{\mathbf{y}}_t$. Previous efforts (Pascanu et al., 2014; Graves, 2016; Zilly et al., 2017) overcame this limitation by incorporating intermediate non-linearities between input elements. These solutions add computational steps between elements in the sequence, greatly increasing runtime complexity. In this work, we explore the delayed-recurrent neural network (d-RNN), in which effective depth is increased by introducing a "delay" between the input and output.

Formally, we define a d-RNN to be a single-layer recurrent neural network as in Equations (1) and (2), such that for any input $\mathbf{x}_t$ the respective output is obtained in $\hat{\mathbf{y}}_{t+d}$, i.e., d timesteps later (Figure 1). We refer to d as the "delay" of the network. The initial hidden state, $\hat{\mathbf{h}}_0$, for a d-RNN is initialized in the same manner as an RNN.

Delaying the output requires special considerations on the data that differ slightly from an RNN. Input sequences need to have $T + d$ elements instead of T. Depending on the task being solved, this can be achieved by adding a "null" input element (e.g., the zero vector), or including d additional elements in the input sequence. When doing a forward pass over the d-RNN for inference, outputs from $t = 1$ to d are discarded as we expect the output for $\mathbf{x}_1$ to be at $\hat{\mathbf{y}}_{1+d}$. The output sequence goes from $\hat{\mathbf{y}}_{1+d}$ to $\hat{\mathbf{y}}_{T+d}$, and has T elements.

Training loss is computed by comparing $\mathbf{z}_t$, the expected output for input $\mathbf{x}_t$, with $\hat{\mathbf{y}}_{t+d}$. Thus, gradients are back-propagated only from delayed outputs $\hat{\mathbf{y}}_{1+d}, \dots, \hat{\mathbf{y}}_{T+d}$. In this way, any modified recurrent cell, such as an LSTM or GRU, can be trained with delayed output to obtain a delayed version of the architecture, e.g., d-LSTM or d-GRU.

3.1. Complexity

Consider an RNN with n units, where input elements have dimension q, and output elements have dimension m. Computing one timestep of this RNN requires three matrix-vector multiplications with complexity $\mathcal{O}\left(nq + nm + n^2\right)$. Applying the non-linear functions $f(\cdot)$ and $g(\cdot)$ requires $\mathcal{O}(m + n)$. Hence, each step of this RNN has runtime complexity of $\mathcal{O}\left(nq + nm + n^2\right)$. For a sequence of length T, the overall computational effort is $\mathcal{O}\left(T(nq + nm + n^2)\right)$. For a d-RNN, the number of timesteps is increased by the delay d, giving total runtime complexity of $\mathcal{O}\left((T + d)(nq + nm + n^2)\right)$.

While the d-RNN incurs some cost, it is cheaper than alternative methods such as micro-steps (Graves, 2016; Zilly et al., 2017), where additional timesteps are inserted between each pair of elements in both the input and output sequences.

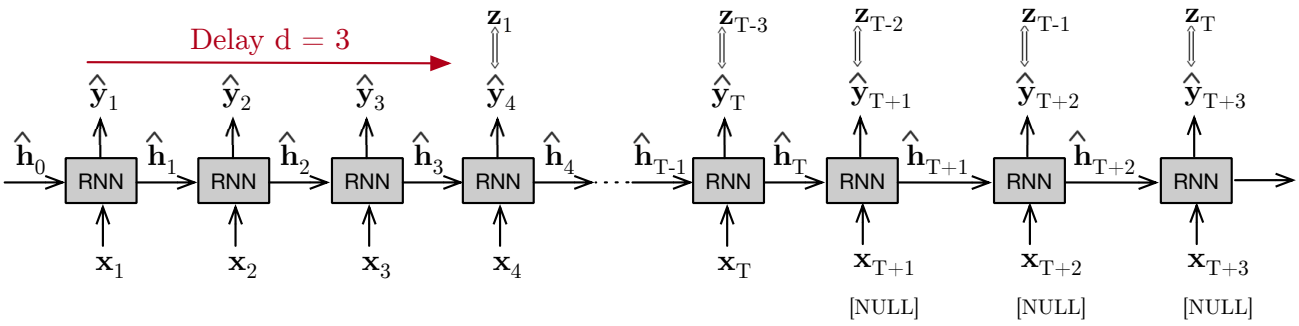


Figure 1. A delayed-recurrent neural network (d-RNN) processing a sequence of T elements. The output is delayed by $d = 3$ timesteps. The first output element is in $\hat{\mathbf{y}}_3$ and the last in $\hat{\mathbf{y}}_{T+3}$. The input sequence has $d = 3$ additional elements, such as '[NULL]' symbols. During training, the outputs are compared with the T elements of the labeled sequence $\{\mathbf{z}_j\}_{j=1}^{T}$.

The runtime complexity for each micro-step is similar to an RNN step, leading the micro-step model complexity to grow with the number of micro-steps d proportionally to $\mathcal{O}(dT)$. In contrast, the d-RNN model complexity only grows proportionally to $\mathcal{O}(d+T)$.

3.2. Stacked RNNs are d-RNNs

The mathematical structure of a stacked RNN is similar to a single-layer RNN with the addition of between-layer connections that add depth. Here we show that any stacked RNN can be flattened into a single-layer d-RNN that produces the exact sequence of hidden states and outputs. We exchange the depth from the between-layer connections with temporal depth applied through a delay in the output. To illustrate this, we rewrite the parameters of a single-layer RNN using the weights and bias terms of a k-layer stacked RNN from Equations (3)-(5):

$$\hat{\mathbf{W}}_{\mathbf{h}} = \begin{bmatrix} \mathbf{W}_{\mathbf{h}}^{(1)} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{W}_{\mathbf{x}}^{(2)} & \mathbf{W}_{\mathbf{h}}^{(2)} & & \\ \mathbf{0} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \mathbf{W}_{\mathbf{x}}^{(i)} \; \mathbf{W}_{\mathbf{h}}^{(i)} & \ddots \\ & & \ddots & \ddots \; \mathbf{0} \\ \mathbf{0} & \cdots & \mathbf{0} & \mathbf{W}_{\mathbf{x}}^{(k)} \; \mathbf{W}_{\mathbf{h}}^{(k)} \end{bmatrix}, \quad (6)$$

$$\hat{\mathbf{b}}_{\mathbf{h}} = \begin{bmatrix} \mathbf{b}_{\mathbf{h}}^{(1)} \\ \vdots \\ \mathbf{b}_{\mathbf{h}}^{(k)} \end{bmatrix}, \qquad \hat{\mathbf{W}}_{\mathbf{x}} = \begin{bmatrix} \mathbf{W}_{\mathbf{x}}^{(1)} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix}, \quad (7)$$

$$\hat{\mathbf{W}}_{\mathbf{o}} = \begin{bmatrix} \mathbf{0} \cdots \mathbf{0} \; \mathbf{W}_{\mathbf{o}} \end{bmatrix}, \quad \hat{\mathbf{b}}_{\mathbf{o}} = \mathbf{b}_{\mathbf{o}}, \quad (8)$$

where $\hat{\mathbf{W}}_{\mathbf{x}} \in \mathbb{R}^{kn \times q}$ are the input weights, $\hat{\mathbf{W}}_{\mathbf{h}} \in \mathbb{R}^{kn \times kn}$ the recurrent weights, $\hat{\mathbf{b}}_{\mathbf{h}} \in \mathbb{R}^{kn}$ the biases, $\hat{\mathbf{W}}_{\mathbf{o}} \in \mathbb{R}^{m \times kn}$ the output weights, and $\hat{\mathbf{b}}_{\mathbf{o}} \in \mathbb{R}^{m}$ the output biases.

One can see from Eq. (6)-(8) that each layer in the stacked RNN is converted into a group of units in the single-layer RNN. The block bidiagonal structure of the recurrent weight matrix $\hat{\mathbf{W}}_{\mathbf{h}}$ makes the hidden state act as a buffer, where each group of units only receives input from itself and the previous group. Information processed through this buffering mechanism eventually arrives at the output after $k-1$ timesteps. In fact, the obtained model is a d-RNN with delay $d = k-1$ and sparsely constrained weights. Note that the d-RNN performs the same computations as the stacked version by trading depth *in layers* for depth *in time*.

Next, we define the following notation: for a vector $\mathbf{v} \in \mathbb{R}^{kn}$ with k blocks, the subvector $\mathbf{v}^{\{i\}} \in \mathbb{R}^{n}$ refers to its ith block following the partition from Equations (6)-(8). For example, the subvector $\hat{\mathbf{h}}_{t+i-1}^{\{i\}}$ is the hidden state at timestep $t+i-1$ for the block i. Namely, this is the recurrent input to the block i when processing timestep $t+i$. We now prove that a d-RNN parameterized by Eq. (6)-(8) is exactly equivalent to the stacked RNN in Eqs. (3)-(5). The proof can be extended to more complex recurrent cells. We include a proof for LSTMs in the supplementary material.

Theorem 1. *Given an input sequence $\{\mathbf{x}_t\}_{t=1\ldots T}$ and a stacked RNN with k layers defined by Equations (3)-(5) with activation functions $f(\cdot)$ and $g(\cdot)$, and initial states $\{\mathbf{h}_0^{(i)}\}_{i=1\ldots k}$, the d-RNN with delay $d = k-1$, defined by Equations (6)-(8) and initialized with $\hat{\mathbf{h}}_0$ such that $\hat{\mathbf{h}}_{i-1}^{\{i\}} = \mathbf{h}_0^{(i)}$, $\forall i = 1 \ldots k$, produces the same output sequence but delayed by $k-1$ timesteps, i.e., $\hat{\mathbf{y}}_{t+k-1} = \mathbf{y}_t$ for all $t = 1 \ldots T$. Further, the sequence of hidden states at each layer i are equivalent with delay $i-1$, i.e., $\hat{\mathbf{h}}_{t+i-1}^{\{i\}} = \mathbf{h}_t^{(i)}$ for all $1 \leq i \leq k$ and $t \geq 1$.*

Proof. See Section 1 of the supplementary material. ■

Theorem 1 makes an assumption that $\hat{\mathbf{h}}_0$ in the d-RNN

can be initialized such that it achieves $\hat{\mathbf{h}}_{i-1}^{\{i\}} = \mathbf{h}_0^{(i)}$ for all blocks. Lemma 1 below implies that initialization for the d-RNN with constrained weights can always be computed from the stacked RNN. The intuition behind it is that we can compute recursively from $\hat{\mathbf{h}}_{i-1}^{\{i\}} = \mathbf{h}_0^{(i)}$ to $\hat{\mathbf{h}}_0^{\{i\}}$ for block i, while inverting the activation function. All commonly used activation functions are surjective, thus it is enough to know the right-inverse of the activation function $f(\cdot)$ (see proof of Lemma). For example, when $f(\cdot)$ is the ReLU, the right-inverse is the identity function $r(d) = d$.

Lemma 1. *Let $f : \mathbb{R} \to D$ be a surjective activation function that maps elements in $\mathbb{R}$ to elements in interval D. Also, let $\mathbf{h}_0^{(i)} \in D^n$ for $i = 1 \dots k$ be the hidden state initialization for a stacked RNN with k layers as defined in (3)-(4). Then, there exists an initial hidden state vector $\hat{\mathbf{h}}_0 \in \mathbb{R}^{kn}$ for a single-layer network in Equations (6)-(7) such that $\hat{\mathbf{h}}_{i-1}^{\{i\}} = \mathbf{h}_0^{(i)}$ $\forall i = 1 \dots k$.*

Proof. See Section 2 of the supplementary material. ■

From this theorem we see that k-layer stacked RNNs can be perfectly expressed as a single-layer d-RNN. In this case, the d-RNN has a specific sparsity structure in its weight matrices that is not present in the generic RNN or d-RNN. As the stacked RNN and the d-RNN with sparsely constrained weights models are equivalent, there is no difference in favor of which one is used in practice, and their runtime complexities are the same[1]. Moreover, they are interchangeable using the weight matrix definitions in Equations (6)-(8).

3.2.1. Relation to Other Topologies

Suppose one takes a weight constrained d-RNN and adds non-zero elements to regions not populated by weights in Eq. (6). These non-zero weights do not correspond to existing connections in the stacked RNN. So what do they correspond to?

To explore this question we illustrate a 4-layer stacked RNN in Figure 2 (a). Here, solid arrows show the standard stacked RNN connections. The d-RNN weight matrices $\hat{\mathbf{W}}_\mathbf{h}$, $\hat{\mathbf{W}}_\mathbf{x}$, and $\hat{\mathbf{W}}_\mathbf{o}$ are shown in Figure 2 (b), where the color of each block matches the corresponding arrow in Figure 2 (a). Blocks on the main diagonal of $\hat{\mathbf{W}}_\mathbf{h}$ connect groups of units to themselves recurrently, while blocks on the subdiagonal correspond to connections between layers in the stacked RNN. More generally, block (i, j) in $\hat{\mathbf{W}}_\mathbf{h}$ corresponds to a connection from $\mathbf{h}_t^{(j)}$ to $\mathbf{h}_{t+j-i+1}^{(i)}$ in the stacked RNN. Thus, blocks in the lower triangle (i.e. $i > j+1$) correspond to connections that point backwards in time, and from a lower layer to a higher layer. For example, the orange block $(3, 1)$ in Figure 2 (b) (and the dashed orange lines in Figure 2 (a)) connects layer 1 at time t to layer 3 at time $t-1$. Conversely, blocks in the upper triangle (i.e. $j > i$) point forward in time and from a higher layer to a lower layer. For example, the red block $(3, 4)$ in Figure 2 (b) (and the dashed red lines in Figure 2 (a)) connects layer 4 at time t to layer 3 at time $t+2$.

Thus we see that adding weights to empty regions in the weight constrained d-RNN can mimic the behavior of many stacked recurrent architectures that have previously been proposed. Among others, it can approximate the IndRNN (Li et al., 2018), td-RNN (Zhang et al., 2016), skip-connections (Graves, 2013), all-to-all layer networks (Chung et al., 2015), clockwork RNN (Koutnik et al., 2014), and hierarchical models (Chung et al., 2017; Ke et al., 2018). Simply removing the constraints on $\hat{\mathbf{W}}_\mathbf{h}$ during training will enable a d-RNN to learn the necessary stacked architecture. However, unlike an ordinary RNN, this requires the output to be delayed based on the desired stacking depth. Further, while the single-layer network has the same total number of units as the corresponding stacked RNN, relaxing constraints on $\hat{\mathbf{W}}_\mathbf{h}$ would mean that the single-layer would have many more parameters.

We can also see that when the delay $d \to T$, the sequence length, the d-RNN becomes similar to a sequence-to-sequence RNN (Sutskever et al., 2014), as it reads the entire input sequence before outputting the first element of the output sequence. Yet these two architectures are not exactly equivalent, as the d-RNN would receive null token inputs during the output phase instead of the previous predicted output as in sequence-to-sequence models. Still, this suggests that the d-RNN can interpolate between a normal RNN (with $d = 0$) and a sequence-to-sequence RNN (with $d = T$).

3.3. Approximating Bidirectional RNNs

We previously showed how a d-RNN can be made equivalent to a stacked RNN by constraining its weight matrices. Without these constraints, the d-RNN has the ability to peek at "future" inputs: it computes the delayed output for time t at $\hat{\mathbf{y}}_{t+d}$ using also the inputs $\mathbf{x}_{t+1}, \dots, \mathbf{x}_{t+d}$ that are beyond timestep t. A similar idea was used in the past as a baseline for bidirectional recurrent neural networks (Bi-RNNs) (Schuster & Paliwal, 1997; Graves & Schmidhuber, 2005). These papers showed that Bi-RNNs were superior to d-RNNs for relatively simple problems, but it is not clear that this comparison holds true for problems that require more non-linear solutions. If a recurrent network can compute the output for time t by exploiting future input elements, what conditions are necessary to approximate its Bi-RNN counterpart? Moreover, can the d-RNN obtain the same results? And, given these conditions, is there a benefit to

[1] Their runtime complexities are the same as we can always obtain a version with reduced computational effort for one model by executing the other and translating the result.

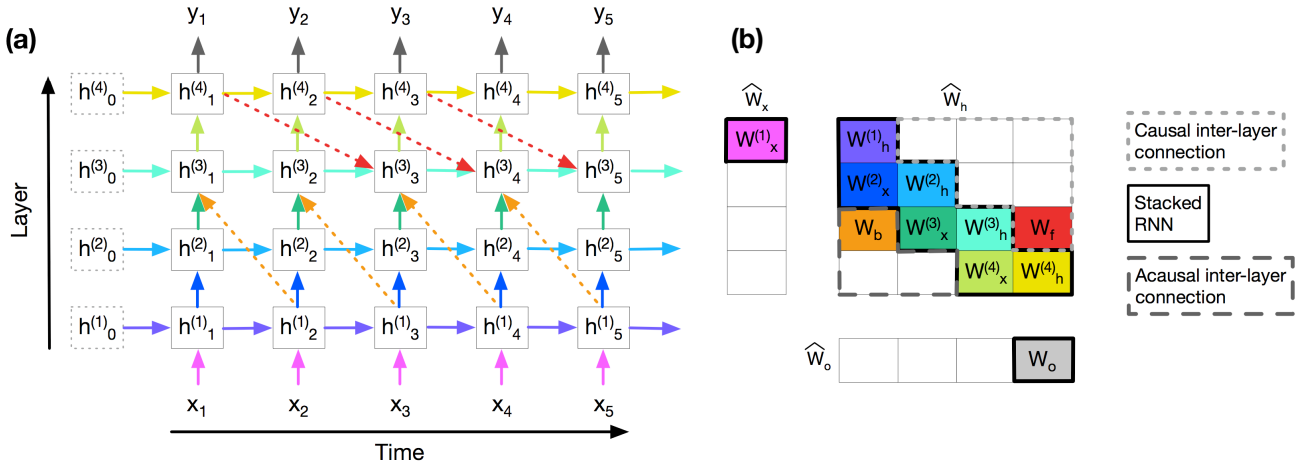


Figure 2. A stacked RNN is equivalent to a single-layer d-RNN under the given sparse weight constraints. The d-RNN produces the same representations as the stacked network. (a) Stacked RNN with $k = 4$ layers where connections show the different weight parameters. (b) Weights of the d-RNN that are equivalent to connections in the stacked RNN.

using the d-RNN instead of the Bi-RNN?

Figure 3 shows the number of non-linear transformations that each network can apply to any input element before computing the output at timestep t_0. The generic RNN processes only past inputs ($t \leq t_0$), and the number of non-linearities decreases for inputs closer to timestep t_0. The Bi-RNN has identical behavior for causal inputs but is augmented symmetrically for acausal inputs. In contrast, the d-RNN has similar behavior for the causal inputs but with a higher number of non-linearities. This trend continues for the first d acausal inputs with a decreasing number of non-linearities until the number reaches zero at $t = t_0 + d + 1$. In order for a d-RNN to have at least as many non-linearities as a Bi-RNN for every element in a sequence, it would need a delay that is twice the sequence length. However, a d-RNN could beat a Bi-RNN when the non-linear influence of nearby acausal inputs on the learned function is larger than elements farther in the future. In these cases, stacking Bi-RNNs would be needed to achieve the same objective.

Using a d-RNN to approximate a Bi-RNN can also decrease computational cost. For a sequence of length T, a stacked Bi-RNN needs to compute both forward and backward RNNs for each layer before it can compute the next one. This synchronization requirement hinders parallelization and increases runtime. In contrast, the forward-pass for the d-RNN takes $T + d$ steps, but does not suffer from synchronization. Thus in highly parallel hardware such as CPUs and GPUs, the runtime of a k-layer stacked Bi-RNN should be at least k times slower than an RNN or d-RNN. Beyond computational costs, d-RNNs can also be used where it is critical to output values in (near) realtime applications (Guo et al., 2016; Arik et al., 2017). A d-RNN requires only the last d elements and a hidden state to compute a new value, whereas bidirectional architectures need to process an entire backward pass of the sequence.

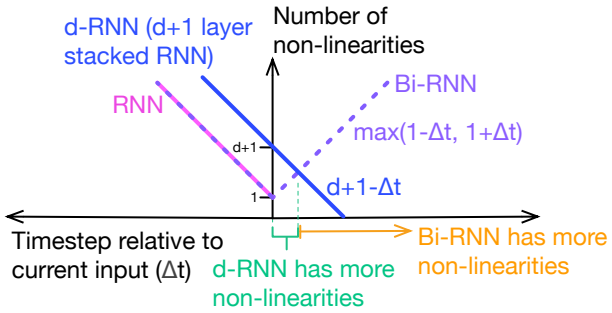


Figure 3. Number of non-linearities that can be applied to past and future sequence elements with respect to current input (Δt=0). The d-RNN only sees d steps into the future.

4. Experiments

We test the capabilities of the d-RNN in four experiments designed to shed more light on the relationships between d-RNNs, RNNs, Bi-RNNs, and stacked networks. For this purpose, the RNN implementation we use is a LSTM network, which avoids vanishing gradients and retains more information over long periods. The delayed LSTM networks are denoted as d-LSTMs. To train each d-LSTM, the input sequences are padded at the end with zero-vectors and

loss is computed by ignoring the first "delay" timesteps, as explained in Section 3. All models are trained using the Adam optimization algorithm (Kingma & Ba, 2015) with learning rate 0.001, $\beta_1 = 0.9$, and $\beta_2 = 0.999$. During training, the gradients are clipped (Pascanu et al., 2013) at 1.0 to avoid explosions. Experiments were implemented using PyTorch 1.1.0 (Paszke et al., 2017), and code can be found at `https://github.com/TuKo/dRNN`.

4.1. Sequence Reversal

First, we propose a simple test to illustrate how the d-LSTM can interpolate between a regular LSTM and Bi-LSTM. In this test we require the recurrent architectures to output a sequence in reverse order while reading it, i.e. $\mathbf{y}_t = \mathbf{x}_{T-t+1}$ for $t = 1, .., T$. Solving this task perfectly is only possible when a network has acausal access to the sequence. Moreover, depending on how many acausal elements a network can access, it is possible to analytically calculate the expected maximum performance that the network can achieve. Given a sequence of length T with elements from a vocabulary $\{1, ..., V\}$, a causal network such as the regular LSTM can output the second half of the elements correctly and guess those in the first half with probability $1/V$. When a network has access to d acausal elements it can start outputting correct elements before reaching the halfway point, and can achieve an expected true positive rate (TPR) of $\frac{1}{2}\left(1 + \frac{1}{V}\right) + \left\lfloor \frac{d+1}{2} \right\rfloor \frac{1}{T}\left(1 - \frac{1}{V}\right)$. We generate data sequences of length $T = 20$ by uniformly sampling integer values between 1 and $V = 4$. The training set consists of 10,000 sequences, the validation set 2,000, and test set 2,000. Output sequences are the input sequences reversed. Values in the input sequences are fed as one-hot vector representations. All networks output via a linear layer with a softmax function that converts to a vector of V probabilities to which cross-entropy loss is applied. The LSTM and d-LSTM networks have 100 hidden units, while the Bi-LSTM has 70 in each direction in order to keep the total number of parameters constant. We use batches of 100 sequences and train for 1,000 epochs with early stopping after 10 epochs and Δ =1e-3.

Figure 4 shows accuracy on this task as a function of the applied delay. The LSTM does not use acausal information and is unable to reverse more than half of the input sequence. Conversely, the Bi-LSTM has full access to every element in the sequence, and can perfectly solve the task. For the d-LSTM network, performance increases as we increase the delay in the output, reaching the same level as the Bi-LSTM once the network has access to the entire sequence before being required to produce any output (delay 19). This experiment demonstrates that the d-LSTM can "interpolate" between LSTM and Bi-LSTM by choosing a delay that ranges between zero and the length of the input sequence.

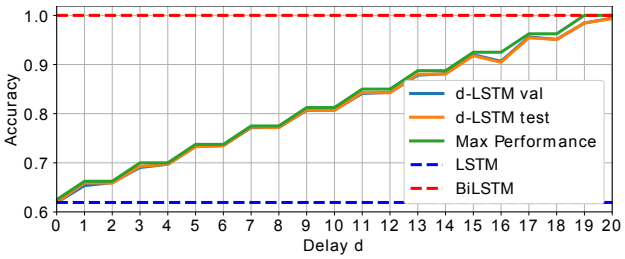


Figure 4. Comparison of different delay values for a d-LSTM network for reversing a sequence. LSTM and Bi-LSTM networks are shown for reference. The network is capable of achieving the expected statistical bound. The d-LSTM with highest delay is capable of solving the task as well as the Bi-LSTM.

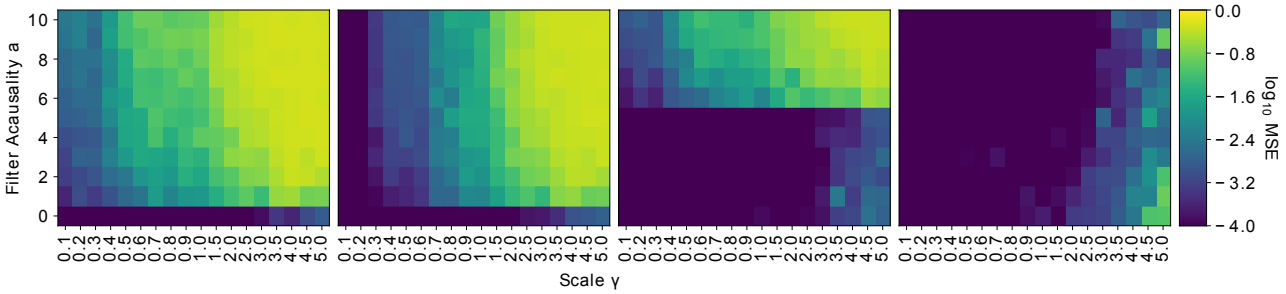
4.2. Evaluating Network Capabilities

The first experiment showed how a d-LSTM with sufficient delay can mimic a Bi-LSTM. In the next experiment we aim at comparing how well d-LSTM, LSTM, and Bi-LSTM networks approximate functions with varying degrees of non-linearity and acausality.

Drawing inspiration from (Schuster & Paliwal, 1997), we require each recurrent network to learn the function $\mathbf{y}_t = \sin(\gamma \sum_{j=-c+1}^{a} \mathbf{w}_{j+c}\mathbf{x}_{t+j})$, where $\mathbf{w}$ is a linear filter. The parameter γ scales the argument of the sine function and thus controls the degree of non-linearity in the function: for small γ the function is roughly linear, while for large γ the function is highly non-linear. Integers $a \geq 0$ (acausal) and $c \geq 0$ (causal) control the length of the causal and acausal portions of the linear filter $\mathbf{w}$ that is applied to the input $\mathbf{x}$.

We generate datasets with different combinations of $\gamma \in [0.1, \ldots, 5.0]$ and $a \in [0, \ldots, 10]$, choosing c such that $a + c = 20$. For each combination, we generate a filter $\mathbf{w}$ with 20 elements drawn uniformly in $[0.0, 1.0)$, and random input sequences with $T = 50$ elements drawn from a uniform distribution $[0.0, 1.0)$. In total, there are 10,000 generated sequences for training, 2,000 for validation, and 2,000 for testing with each set of parameter values. The output is computed following the previous formula and with zero padding for the borders. We generate 5 repetitions of each dataset with different filters $\mathbf{w}$ and inputs $\mathbf{x}$.

We train LSTM, d-LSTM with delays 5 and 10, and Bi-LSTM networks to minimize mean squared error (MSE). The LSTM and d-LSTM have 100 hidden units and the Bi-LSTM has 70 per network, matching the numbers of parameters. A linear layer after the recurrent layer outputs a single value per timestep. Models are trained in batches of 100 sequences for 1,000 epochs. Training is stopped if the validation MSE falls below 1e-5. Training is repeated five times for each (γ, a) value.



(a) LSTM (b) Bi-LSTM (c) d-LSTM (d=5) (d) d-LSTM (d=10)

Figure 5. Error maps for the sine function experiment with different degrees of non-linearity (horizontal axis) and amounts of acausality of the filter (vertical axis). Tested architectures: (a) LSTM, (b) Bi-LSTM, (c) d-LSTM with delay=5, and (d) d-LSTM with delay=10. Dark blue regions depict perfect filtering (low error), transitioning to yellow regions with high error.

Figure 5 shows the average test MSE for each model as a function of γ (degree of input non-linearity) and a (acausality). LSTM performance (Fig. 5 (a)) is poor everywhere except where the filter is purely causal. Surprisingly, the network performs quite well even when the amount of non-linearity (γ) is quite high. The reason for this seems to be that temporal depth enables the LSTM to approximate this function well. Bi-LSTM performance (Fig. 5 (b)) follows a similar trend for the causal case ($a = 0$) as the forward LSTM, but also has good performance for acausal filters ($a > 0$) when the function is nearly linear (γ is small). As the non-linearity of the function increases, however, Bi-LSTM performance suffers. This occurs because the Bi-LSTM needs to approximate a highly non-linear function with a linear combination of its forward and backward outputs, which cannot be done with small error. Improving performance would require stacked Bi-LSTM layers.

In contrast, d-LSTM networks have excellent performance for both non-linear and acausal functions. The d-LSTM with delay 5 (Fig. 5 (c)) shows a clear switch in performance from acausality $a = 5$ to 6. This perfectly matches the limit of acausal elements that the network has access to. For the d-LSTM with delay 10 (Fig. 5 (d)), the network performs well for acausality values a up to 10.

An interesting outcome of this experiment is the better performance observed for the d-LSTM over the Bi-LSTM. This shows that the d-LSTM can be a better fit than a Bi-LSTM for the right task. Furthermore, the d-LSTM network seems to approximate the functionality of a stacked Bi-LSTM by approximating highly non-linear functions. In practice, this could be a great benefit for applications where there is no need to treat the whole sequence. Moreover, this could be impossible in other cases, such as streamed data. In such cases, the d-LSTM would shine over bidirectional architectures. On the other hand, we expect the Bi-LSTM to perform better when the acausality needs for the task are longer than the delay, i.e., $a > d$.

4.3. Masked Character-Level Language Modeling

Next we examined a language task which should benefit from acausal information, masked character-level language modeling. This task is adapted from previous work in training bidirectional language models (Devlin et al., 2019). To generate masked sequences, we randomly replace each character with a mask token ('[MASK]') with 20% probability. The task of the network is to predict the correct character when it encounters a mask token. Because each sequence contains multiple mask tokens, the network will need to fill in some mask tokens conditioned on an input sequence that already contains one or more mask tokens. This can be thought of as a signal reconstruction task: when sequential inputs are randomly degraded, how well can the network recover the true signal? Acausal information clearly helps with this reconstruction. For example, the missing letter in the sequence "hik[MASK]ng" is easier to predict than the sequence "hik[MASK]".

We used `text8`, a clean 100MB sample of English Wikipedia text (Mahoney, 2006) which consists of 27 characters (the English alphabet and spaces). The input data contained an extra 28th mask character. These 28 characters were mapped to an input embedding layer of dimension 10. The output layer was independent of the input embedding, and only consisted of the 27 non-mask characters. Following previous work (Mikolov et al., 2012), the first 90M characters formed the training set, the next 5M the validation set, and the last 5M the test set. All models were trained with a sequence length of 180 characters, in mini-batches of 128 sequences for a total of 20 epochs. Success on the task is measured by calculating bits-per-character (BPC) for the mask tokens only. We measured forward-pass runtimes on a Nvidia Titan V GPU and report average time to process a mini-batch.

The results are summarized in Table 1. As expected, the stacked Bi-LSTMs achieve the lowest BPC. However, as the number of layers increases, the inference runtime also

Table 1. Performance of different networks on the masked character-level language modeling task in bits per character (BPC); lower is better. Mean and standard deviation values are computed over 5 repetitions of training and inference runtime on the test set.

MODEL	LAYERS	DELAY	UNITS / LAYER	PARAMS.	VAL. BPC	TEST BPC	RUNTIME
LSTM	1	-	1024	4271411	2.003 ± 0.003	$\mathbf{2.075 \pm 0.002}$	$3.44ms \pm 0.09$
LSTM	2	-	594	4283641	2.015 ± 0.005	2.087 ± 0.005	$4.93ms \pm 0.13$
LSTM	5	-	343	4272372	2.091 ± 0.016	2.155 ± 0.014	$17.22ms \pm 0.62$
BI-LSTM	1	-	722	4278879	0.977 ± 0.004	1.037 ± 0.004	$4.97ms \pm 0.07$
BI-LSTM	2	-	363	4277173	0.633 ± 0.003	$\mathbf{0.677 \pm 0.002}$	$13.72ms \pm 0.31$
BI-LSTM	5	-	202	4287151	0.637 ± 0.003	0.677 ± 0.004	$29.18ms \pm 0.23$
D-LSTM	1	1	1024	4271411	1.332 ± 0.001	1.390 ± 0.001	$3.29ms \pm 0.22$
D-LSTM	1	5	1024	4271411	0.708 ± 0.005	0.755 ± 0.004	$3.39ms \pm 0.08$
D-LSTM	1	8	1024	4271411	0.662 ± 0.002	$\mathbf{0.706 \pm 0.003}$	$3.36ms \pm 0.08$
D-LSTM	1	10	1024	4271411	0.666 ± 0.004	0.709 ± 0.004	$3.56ms \pm 0.10$

increases because of the synchronization needed between layers. Notably, d-LSTMs with intermediate delays achieve a BPC that is within 5% of the Bi-LSTM with at least $4\times$ faster runtime. Since all of the d-LSTMs have a single layer, inference runtime remains constant as the delay and the capacity of these networks increases. We find similar results for other network capacities (see supplementary material).

4.4. Real-World Part-of-Speech Tagging

In the previous experiments, we show that d-LSTM is capable of approximating and even outperforming a Bi-LSTM in some cases. In practice, however, the elements in a sequence may have different forward and backward relations. This poses a challenge for delayed networks that are constrained to a specific delay. If the delay is too low, it may not be enough for some long dependencies between elements. If it is too high, the network may forget information and require higher capacity (and maybe training data). This is prevalent in several NLP tasks. Therefore we compare the performance of the d-LSTM with a Bi-LSTM on an NLP task where Bi-LSTMs achieve state-of-the-art performance, the Part-of-Speech (POS) tagging task (Ling et al., 2015; Ballesteros et al., 2015; Plank et al., 2016). The task involves processing a variable length sequence to predict a POS tag (e.g. Noun, Verb) per word, using the Universal Dependencies (UD) (Nivre et al., 2016) dataset. More details can be found in the supplementary material.

The dual Bi-LSTM architecture proposed by Plank et al. (2016) is followed to test the approximation capacity of the d-LSTMs. In this model, a word is encoded using a combination of word embeddings and character-level encoding. The encoded word is fed to a Bi-LSTM followed by a linear layer with softmax to produce POS tags. The character-level encoding is produced by first computing the embedding of each character and then feeding it to a Bi-LSTM. The last hidden state in each direction is concatenated with the word embedding to form the character-level encoding.

The character-level Bi-LSTM has 100 units in each direction and the LSTM/d-LSTMs have 200 units to generate encodings of the same size. For the word-level subnetwork, the hidden state is of size 188 for the Bi-LSTM, and 300 units for the LSTM/d-LSTM to match the number of parameters. The networks are trained for 20 epochs with cross-entropy loss. We train combinations of networks with delays 0 (LSTM), 1, 3, and 5 for the character-level subnetwork, and delays 0 through 4 for the word-level. Each network has 5 repeats with random initialization.

Results are presented in Table 2. For brevity, we include a subset of the combinations for each language (the complete table can be found in the supplementary material). For the character-level model, LSTMs without delay yield reduced performance. However, replacing only the character-level Bi-LSTM with a LSTM does not affect the performance (supplementary material). This suggests that only the word-level subnetwork benefits from acausal elements in the sentence. Interestingly, using a d-LSTM with delay 1 for the character-level network achieves a small improvement over the double-bidirectional model in English and German. Replacing the word-level Bi-LSTM with an LSTM decreases performance significantly. However, using even a d-LSTM with delay 1 improves performance to within 0.3% of the original Bi-LSTM model.

5. Conclusions

In this paper we analyze the d-RNN, a single layer RNN where the output is delayed relative to the input. We show that this simple modification to the classical RNN adds both depth in time and acausal processing. We prove that a d-RNN is a superset of stacked RNNs, which are frequently used for sequence problems: a d-RNN with output delay d and specific constraints on its weights is exactly equivalent to a stacked RNN with $d + 1$ layers. We also show that the d-RNN can approximate bidirectional RNNs and stacked bidirectional RNNs because the delay allows the model to

Table 2. Parts-of-Speech performance for German, English, and French languages. The models are composed of two subnetworks at character-level and word-level. Best bidirectional network and best forward-only network are marked in bold for each language.

LANGUAGE	CHAR-LEVEL NETWORK	WORD-LEVEL NETWORK	VALIDATION ACCURACY	TEST ACCURACY
GERMAN	LSTM	LSTM	92.05 ± 0.16	91.58 ± 0.11
	D-LSTM DELAY=1	D-LSTM DELAY=1	93.48 ± 0.31	$\mathbf{92.87 \pm 0.24}$
	D-LSTM DELAY=1	BI-LSTM	93.93 ± 0.06	$\mathbf{93.39 \pm 0.18}$
	BI-LSTM	BI-LSTM	93.88 ± 0.13	93.15 ± 0.08
ENGLISH	LSTM	LSTM	92.05 ± 0.13	92.14 ± 0.10
	D-LSTM DELAY=1	D-LSTM DELAY=1	94.57 ± 0.08	$\mathbf{94.57 \pm 0.14}$
	D-LSTM DELAY=1	BI-LSTM	94.94 ± 0.07	$\mathbf{94.95 \pm 0.06}$
	BI-LSTM	BI-LSTM	94.85 ± 0.05	94.84 ± 0.08
FRENCH	LSTM	LSTM	96.67 ± 0.07	96.10 ± 0.11
	D-LSTM WITH DELAY=1	D-LSTM WITH DELAY=1	97.49 ± 0.04	$\mathbf{97.04 \pm 0.13}$
	D-LSTM WITH DELAY=1	BI-LSTM	97.67 ± 0.07	$\mathbf{97.23 \pm 0.12}$
	BI-LSTM	BI-LSTM	97.63 ± 0.06	97.22 ± 0.11

look at future as well as past inputs. In sum, we found that d-RNNs are a simple, elegant, and computationally efficient alternative that captures many of the best features of different RNN architectures while avoiding many downsides.

Acknowledgements

We would like to thank Mariano Tepper for sharing his thoughts and useful comments. In addition, we would like to show our appreciation to the reviewers who invested their time in reviewing this work and made useful and constructive observations in these difficult times.

References

Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G., et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pp. 173–182, 2016.

Arik, S. O., Chrzanowski, M., Coates, A., Diamos, G., Gibiansky, A., Kang, Y., Li, X., Miller, J., Ng, A., Raiman, J., Sengupta, S., and Shoeybi, M. Deep voice: Real-time neural text-to-speech. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 195–204. JMLR.org, 2017. URL http://dl.acm.org/citation.cfm?id=3305381.3305402.

Ballesteros, M., Dyer, C., and Smith, N. A. Improved transition-based parsing by modeling characters instead of words with LSTMs. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 349–359, Lisbon, Portugal, September 2015. Association for Computational Linguistics. doi: 10.18653/v1/D15-1041. URL https://www.aclweb.org/anthology/D15-1041.

Baziotis, C., Pelekis, N., and Doulkeridis, C. Datastories at semeval-2017 task 4: Deep lstm with attention for message-level and topic-based sentiment analysis. In *Proceedings of the 11th international workshop on semantic evaluation (SemEval-2017)*, pp. 747–754, 2017.

Bengio, Y. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009. ISSN 1935-8237. doi: 10.1561/2200000006. URL http://dx.doi.org/10.1561/2200000006.

Bengio, Y., Simard, P., and Frasconi, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, March 1994. ISSN 1045-9227. doi: 10.1109/72.279181.

Bengio, Y., LeCun, Y., et al. Scaling learning algorithms towards ai. *Large-scale kernel machines*, 34(5):1–41, 2007.

Bianchini, M. and Scarselli, F. On the complexity of neural network classifiers: A comparison between shallow and deep architectures. *IEEE Transactions on Neural Networks and Learning Systems*, 25(8):1553–1565, Aug 2014. ISSN 2162-237X. doi: 10.1109/TNNLS.2013.2293637.

Campos, V., Jou, B., i Nieto, X. G., Torres, J., and Chang, S.-F. Skip RNN: Learning to skip state updates in recurrent neural networks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=HkwVAXyCW.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. Gated feedback recurrent neural networks. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 2067–2075. JMLR.org, 2015. URL http://dl.acm.org/citation.cfm?id=3045118.3045338.

Chung, J., Ahn, S., and Bengio, Y. Hierarchical multiscale recurrent neural networks. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=S1di0sfgl.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs]*, May 2019. URL http://arxiv.org/abs/1810.04805.

Graves, A. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

Graves, A. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Graves, A. and Schmidhuber, J. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602 – 610, 2005. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2005.06.042. URL http://www.sciencedirect.com/science/article/pii/S0893608005001206. IJCNN 2005.

Guo, T., Xu, Z., Yao, X., Chen, H., Aberer, K., and Funaya, K. Robust online time series prediction with recurrent neural networks. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 816–825, Oct 2016. doi: 10.1109/DSAA.2016.92.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Ke, N. R., Żołna, K., Sordoni, A., Lin, Z., Trischler, A., Bengio, Y., Pineau, J., Charlin, L., and Pal, C. Focused hierarchical RNNs for conditional sequence processing. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 2554–2563, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL http://proceedings.mlr.press/v80/ke18a.html.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Koutnik, J., Greff, K., Gomez, F., and Schmidhuber, J. A clockwork rnn. In Xing, E. P. and Jebara, T. (eds.), *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pp. 1863–1871, Bejing, China, June 2014. PMLR. URL http://proceedings.mlr.press/v32/koutnik14.html.

Li, S., Li, W., Cook, C., Zhu, C., and Gao, Y. Independently recurrent neural network (indrnn): Building a longer and deeper rnn. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

Ling, W., Dyer, C., Black, A. W., Trancoso, I., Fermandez, R., Amir, S., Marujo, L., and Luis, T. Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 1520–1530, Lisbon, Portugal, September 2015. Association for Computational Linguistics. doi: 10.18653/v1/D15-1176. URL https://www.aclweb.org/anthology/D15-1176.

Mahoney, M. Relationship of Wikipedia Text to Clean Text, June 2006. URL http://mattmahoney.net/dc/textdata.html.

Mikolov, T., Sutskever, I., Deoras, A., Le, H.-S., and Kombrink, S. Subword language modeling with neural networks. *Preprint*, 2012. URL http://www.fit.vutbr.cz/~imikolov/rnnlm/char.pdf.

Nivre, J., de Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajič, J., Manning, C. D., McDonald, R., Petrov, S., Pyysalo, S., Silveira, N., Tsarfaty, R., and Zeman, D. Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pp. 1659–1666, Portorož, Slovenia, May 2016. European Language Resources Association (ELRA). URL https://www.aclweb.org/anthology/L16-1262.

Pascanu, R., Mikolov, T., and Bengio, Y. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pp. III–1310–III–1318. JMLR.org, 2013. URL http://dl.acm.org/citation.cfm?id=3042817.3043083.

Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. How to construct deep recurrent neural networks. In *Proceedings of the Second International Conference on Learning Representations (ICLR 2014)*, 2014.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in PyTorch. In *NIPS Workshop on the future of gradient-based machine learning software & techniques*, 2017.

Plank, B., Søgaard, A., and Goldberg, Y. Multilingual part-of-speech tagging with bidirectional long short-term memory models and auxiliary loss. In *Proceedings of the 54th Annual Meeting of the Association for*

Computational Linguistics (Volume 2: Short Papers), pp. 412–418, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-2067. URL https://www.aclweb.org/anthology/P16-2067.

Raiko, T., Valpola, H., and Lecun, Y. Deep learning made easier by linear transformations in perceptrons. In Lawrence, N. D. and Girolami, M. (eds.), *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics*, volume 22 of *Proceedings of Machine Learning Research*, pp. 924–932, La Palma, Canary Islands, 21–23 Apr 2012. PMLR. URL http://proceedings.mlr.press/v22/raiko12.html.

Schmidhuber, J. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992.

Schuster, M. and Paliwal, K. K. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, Nov 1997. ISSN 1053-587X. doi: 10.1109/78.650093.

Sutskever, I., Martens, J., and Hinton, G. E. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1017–1024, 2011. URL https://icml.cc/2011/papers/524_icmlpaper.pdf.

Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 27*, pp. 3104–3112. Curran Associates, Inc., 2014.

Zhang, S., Wu, Y., Che, T., Lin, Z., Memisevic, R., Salakhutdinov, R. R., and Bengio, Y. Architectural complexity measures of recurrent neural networks. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 29*, pp. 1822–1830. Curran Associates, Inc., 2016.

Zilly, J. G., Srivastava, R. K., Koutník, J., and Schmidhuber, J. Recurrent highway networks. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 4189–4198, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL http://proceedings.mlr.press/v70/zilly17a.html.