## 10. Recurrent Neural Networks – Modeling Sequences and Stacks

When dealing with language data, it is very common to work with sequences, such as words (sequences of letters), sentences (sequences of words) and documents. We saw how feed-forward networks can accommodate arbitrary feature functions over sequences through the use of vector concatenation and vector addition (CBOW). In particular, the CBOW representations allows to encode arbitrary length sequences as fixed sized vectors. However, the CBOW representation is quite limited, and forces one to disregard the order of features. The convolutional networks also allow encoding a sequence into a fixed size vector. While representations derived from convolutional networks are an improvement above the CBOW representation as they offer some sensitivity to word order, their order sensitivity is restricted to mostly local patterns, and disregards the order of patterns that are far apart in the sequence.

Recurrent neural networks (RNNs) (Elman, 1990) allow representing arbitrarily sized structured inputs in a fixed-size vector, while paying attention to the structured properties of the input.

### 10.1 The RNN Abstraction

We use $\mathbf{x_{i:j}}$ to denote the sequence of vectors $\mathbf{x_i}, \ldots, \mathbf{x_j}$. The RNN abstraction takes as input an ordered list of input vectors $\mathbf{x_1}, ..., \mathbf{x_n}$ together with an initial *state vector* $\mathbf{s_0}$, and returns an ordered list of state vectors $\mathbf{s_1}, ..., \mathbf{s_n}$, as well as an ordered list of *output vectors* $\mathbf{y_1}, ..., \mathbf{y_n}$. An output vector $\mathbf{y_i}$ is a function of the corresponding state vector $\mathbf{s_i}$. The input vectors $\mathbf{x_i}$ are presented to the RNN in a sequential fashion, and the state vector $\mathbf{s_i}$ and output vector $\mathbf{y_i}$ represent the state of the RNN after observing the inputs $\mathbf{x_{1:i}}$. The output vector $\mathbf{y_i}$ is then used for further prediction. For example, a model for predicting the conditional probability of an event $e$ given the sequence $\mathbf{m_{1:i}}$ can be defined as $p(e = j|\mathbf{x_{1:i}}) = softmax(\mathbf{y_i}\mathbf{W} + \mathbf{b})[j]$. The RNN model provides a framework for conditioning on the entire history $\mathbf{x_1}, \ldots, \mathbf{x_i}$ without resorting to the Markov assumption which is traditionally used for modeling sequences. Indeed, RNN-based language models result in very good perplexity scores when compared to n-gram based models.

Mathematically, we have a recursively defined function $R$ that takes as input a state vector $\mathbf{s_i}$ and an input vector $\mathbf{x_{i+1}}$, and results in a new state vector $\mathbf{s_{i+1}}$. An additional function $O$ is used to map a state vector $\mathbf{s_i}$ to an output vector $\mathbf{y_i}$. When constructing an RNN, much like when constructing a feed-forward network, one has to specify the dimension of the inputs $\mathbf{x_i}$ as well as the dimensions of the outputs $\mathbf{y_i}$. The dimensions of the states $\mathbf{s_i}$ are a function of the output dimension.[26]

26. While RNN architectures in which the state dimension is independent of the output dimension are possible, the current popular architectures, including the Simple RNN, the LSTM and the GRU do not follow this flexibility.

$$\begin{aligned} RNN(\mathbf{s_0}, \mathbf{x_{1:n}}) =& \mathbf{s_{1:n}},\ \mathbf{y_{1:n}} \\ \mathbf{s_i} &= R(\mathbf{s_{i-1}}, \mathbf{x_i}) \\ \mathbf{y_i} &= O(\mathbf{s_i}) \end{aligned}$$

$$\mathbf{x_i} \in \mathbb{R}^{d_{in}},\ \ \mathbf{y_i} \in \mathbb{R}^{d_{out}},\ \ \mathbf{s_i} \in \mathbb{R}^{f(d_{out})}$$

The functions $R$ and $O$ are the same across the sequence positions, but the RNN keeps track of the states of computation through the state vector that is kept and being passed between invocations of $R$.

Graphically, the RNN has been traditionally presented as in Figure 5.

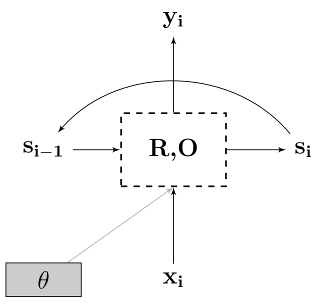

Figure 5: Graphical representation of an RNN (recursive).

This presentation follows the recursive definition, and is correct for arbitrary long sequences. However, for a finite sized input sequence (and all input sequences we deal with are finite) one can *unroll* the recursion, resulting in the structure in Figure 6.

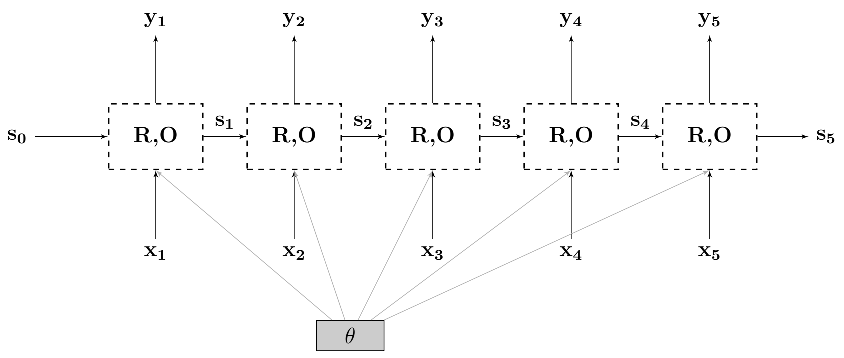

Figure 6: Graphical representation of an RNN (unrolled).

While not usually shown in the visualization, we include here the parameters $\theta$ in order to highlight the fact that the same parameters are shared across all time steps. Different instantiations of $R$ and $O$ will result in different network structures, and will exhibit different properties in terms of their running times and their ability to be trained effectively using gradient-based methods. However, they all adhere to the same abstract interface. We will provide details of concrete instantiations of $R$ and $O$ – the Simple RNN, the LSTM and the GRU – in Section 11. Before that, let's consider modeling with the RNN abstraction.

First, we note that the value of $\mathbf{s_i}$ is based on the entire input $\mathbf{x_1}, ..., \mathbf{x_i}$. For example, by expanding the recursion for $i = 4$ we get:

$$\begin{aligned}
\mathbf{s_4} =& R(\mathbf{s_3}, \mathbf{x_4}) \\
=& R(\overbrace{R(\mathbf{s_2}, \mathbf{x_3})}^{\mathbf{s_3}}, \mathbf{x_4}) \\
=& R(R(\overbrace{R(\mathbf{s_1}, \mathbf{x_2})}^{\mathbf{s_2}}, \mathbf{x_3}), \mathbf{x_4}) \\
=& R(R(R(\overbrace{R(\mathbf{s_0}, \mathbf{x_1})}^{\mathbf{s_1}}, \mathbf{x_2}), \mathbf{x_3}), \mathbf{x_4})
\end{aligned}$$

Thus, $\mathbf{s_n}$ (as well as $\mathbf{y_n}$) could be thought of as *encoding* the entire input sequence.[27] Is the encoding useful? This depends on our definition of usefulness. The job of the network training is to set the parameters of $R$ and $O$ such that the state conveys useful information for the task we are tying to solve.

## 10.2 RNN Training

Viewed as in Figure 6 it is easy to see that an unrolled RNN is just a very deep neural network (or rather, a very large *computation graph* with somewhat complex nodes), in which the same parameters are shared across many parts of the computation. To train an RNN network, then, all we need to do is to create the unrolled computation graph for a given input sequence, add a loss node to the unrolled graph, and then use the backward (backpropagation) algorithm to compute the gradients with respect to that loss. This procedure is referred to in the RNN literature as *backpropagation through time*, or BPTT (Werbos, 1990).[28] There are various ways in which the supervision signal can be applied.

**Acceptor** One option is to base the supervision signal only on the final output vector, $\mathbf{y_n}$. Viewed this way, the RNN is an *acceptor*. We observe the final state, and then decide

27. Note that, unless $R$ is specifically designed against this, it is likely that the later elements of the input sequence have stronger effect on $\mathbf{s_n}$ than earlier ones.

28. Variants of the BPTT algorithm include unrolling the RNN only for a fixed number of input symbols at each time: first unroll the RNN for inputs $\mathbf{x_{1:k}}$, resulting in $\mathbf{s_{1:k}}$. Compute a loss, and backpropagate the error through the network ($k$ steps back). Then, unroll the inputs $\mathbf{x_{k+1:2k}}$, this time using $\mathbf{s_k}$ as the initial state, and again backpropagate the error for $k$ steps, and so on. This strategy is based on the observations that for the Simple-RNN variant, the gradients after $k$ steps tend to vanish (for large enough $k$), and so omitting them is negligible. This procedure allows training of arbitrarily long sequences. For RNN variants such as the LSTM or the GRU that are designed specifically to mitigate the vanishing gradients problem, this fixed size unrolling is less motivated, yet it is still being used, for example when doing language modeling over a book without breaking it into sentences.

on an outcome.[29] For example, consider training an RNN to read the characters of a word one by one and then use the final state to predict the part-of-speech of that word (this is inspired by (Ling et al., 2015b)), an RNN that reads in a sentence and, based on the final state decides if it conveys positive or negative sentiment (this is inspired by (Wang et al., 2015b)) or an RNN that reads in a sequence of words and decides whether it is a valid noun-phrase. The loss in such cases is defined in terms of a function of $\mathbf{y_n} = O(\mathbf{s_n})$, and the error gradients will backpropagate through the rest of the sequence (see Figure 7).[30] The loss can take any familiar form – cross entropy, hinge, margin, etc.

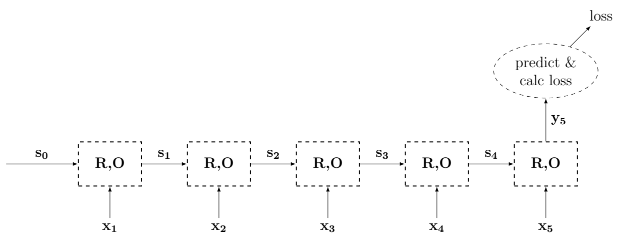

Figure 7: Acceptor RNN Training Graph.

**Encoder** Similar to the acceptor case, an encoder supervision uses only the final output vector, $\mathbf{y_n}$. However, unlike the acceptor, where a prediction is made solely on the basis of the final vector, here the final vector is treated as an encoding of the information in the sequence, and is used as additional information together with other signals. For example, an extractive document summarization system may first run over the document with an RNN, resulting in a vector $\mathbf{y_n}$ summarizing the entire document. Then, $\mathbf{y_n}$ will be used together with other features in order to select the sentences to be included in the summarization.

**Transducer** Another option is to treat the RNN as a transducer, producing an output for each input it reads in. Modeled this way, we can compute a local loss signal $L_{local}(\mathbf{\hat{y}_i}, \mathbf{y_i})$ for each of the outputs $\mathbf{\hat{y}_i}$ based on a true label $\mathbf{y_i}$. The loss for unrolled sequence will then be: $L(\hat{\mathbf{y_{1:n}}}, \mathbf{y_{1:n}}) = \sum_{i=1}^{n} L_{local}(\mathbf{\hat{y}_i}, \mathbf{y_i})$, or using another combination rather than a sum such as an average or a weighted average (see Figure 8). One example for such a transducer is a sequence tagger, in which we take $\mathbf{x_{i:n}}$ to be feature representations for the $n$ words of a sentence, and $\mathbf{y_i}$ as an input for predicting the tag assignment of word $i$ based on words $1{:}i$. A CCG super-tagger based on such an architecture provides state-of-the art CCG super-tagging results (Xu et al., 2015).

A very natural use-case of the transduction setup is for language modeling, in which the sequence of words $\mathbf{x_{1:i}}$ is used to predict a distribution over the $i+1$th word. RNN based

29. The terminology is borrowed from Finite-State Acceptors. However, the RNN has a potentially infinite number of states, making it necessary to rely on a function other than a lookup table for mapping states to decisions.
30. This kind of supervision signal may be hard to train for long sequences, especially so with the Simple-RNN, because of the vanishing gradients problem. It is also a generally hard learning task, as we do not tell the process on which parts of the input to focus.

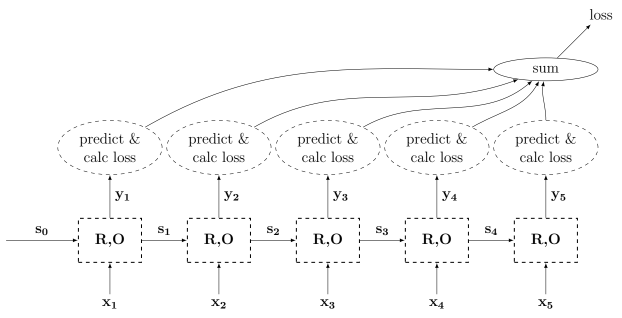

Figure 8: Transducer RNN Training Graph.

language models are shown to provide much better perplexities than traditional language models (Mikolov et al., 2010; Sundermeyer, Schlüter, & Ney, 2012; Mikolov, 2012).

Using RNNs as transducers allows us to relax the Markov assumption that is traditionally taken in language models and HMM taggers, and condition on the entire prediction history. The power of the ability to condition on arbitrarily long histories is demonstrated in generative character-level RNN models, in which a text is generated character by character, each character conditioning on the previous ones (Sutskever, Martens, & Hinton, 2011). The generated texts show sensitivity to properties that are not captured by n-gram language models, including line lengths and nested parenthesis balancing. For a good demonstration and analysis of the properties of RNN-based character level language models, see (Karpathy, Johnson, & Li, 2015).

**Encoder - Decoder** Finally, an important special case of the encoder scenario is the Encoder-Decoder framework (Cho, van Merrienboer, Bahdanau, & Bengio, 2014a; Sutskever et al., 2014). The RNN is used to encode the sequence into a vector representation $\mathbf{y_n}$, and this vector representation is then used as auxiliary input to another RNN that is used as a decoder. For example, in a machine-translation setup the first RNN encodes the source sentence into a vector representation $\mathbf{y_n}$, and then this state vector is fed into a separate (decoder) RNN that is trained to predict (using a transducer-like language modeling objective) the words of the target language sentence based on the previously predicted words as well as $\mathbf{y_n}$. The supervision happens only for the decoder RNN, but the gradients are propagated all the way back to the encoder RNN (see Figure 9).

Such an approach was shown to be surprisingly effective for Machine Translation (Sutskever et al., 2014) using LSTM RNNs. In order for this technique to work, Sutskever et al found it effective to input the source sentence in reverse, such that $\mathbf{x_n}$ corresponds to the first word of the sentence. In this way, it is easier for the second RNN to establish the relation between the first word of the source sentence to the first word of the target sentence. Another use-case of the encoder-decoder framework is for sequence transduction. Here, in order to generate tags $t_1, \ldots, t_n$, an encoder RNN is first used to encode the sentence $\mathbf{x_{1:n}}$ into fixed sized vector. This vector is then fed as the initial state vector of another (transducer) RNN, which is used together with $\mathbf{x_{1:n}}$ to predict the label $t_i$ at each position $i$. This approach

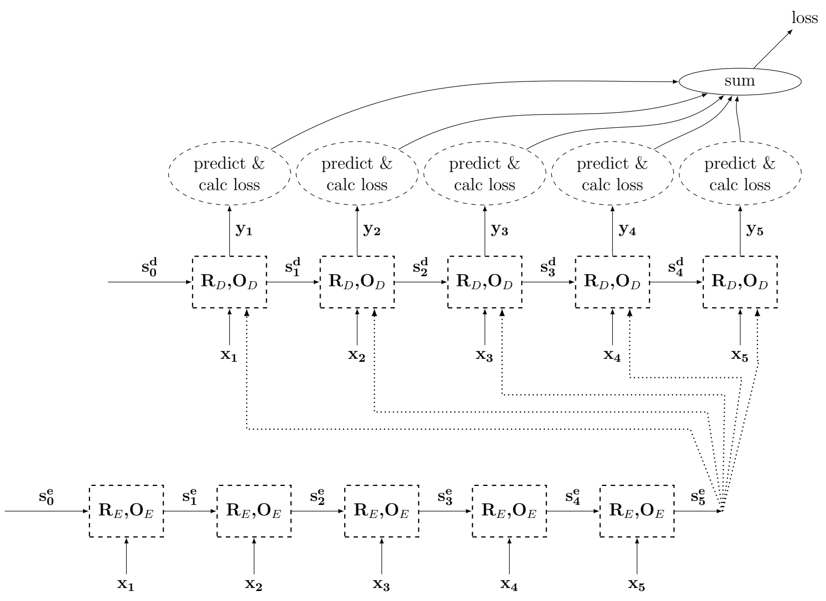

Figure 9: Encoder-Decoder RNN Training Graph.

was used in (Filippova, Alfonseca, Colmenares, Kaiser, & Vinyals, 2015) to model sentence compression by deletion.

## 10.3 Multi-layer (stacked) RNNs

RNNs can be stacked in layers, forming a grid (Hihi & Bengio, 1996). Consider $k$ RNNs, $RNN_1, \ldots, RNN_k$, where the $j$th RNN has states $\mathbf{s^j_{1:n}}$ and outputs $\mathbf{y^j_{1:n}}$. The input for the first RNN are $\mathbf{x_{1:n}}$, while the input of the $j$th RNN ($j \geq 2$) are the outputs of the RNN below it, $\mathbf{y^{j-1}_{1:n}}$. The output of the entire formation is the output of the last RNN, $\mathbf{y^k_{1:n}}$. Such layered architectures are often called *deep RNNs*. A visual representation of a 3-layer RNN is given in Figure 10.

While it is not theoretically clear what is the additional power gained by the deeper architecture, it was observed empirically that deep RNNs work better than shallower ones on some tasks. In particular, Sutskever et al (2014) report that a 4-layers deep architecture was crucial in achieving good machine-translation performance in an encoder-decoder framework. Irsoy and Cardie (2014) also report improved results from moving from a one-layer BI-RNN to an architecture with several layers. Many other works report result using layered RNN architectures, but do not explicitly compare to 1-layer RNNs.

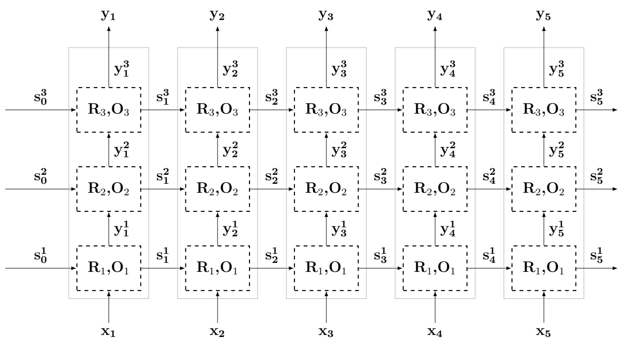
Figure 10: A 3-layer ("deep") RNN architecture.

## 10.4 BI-RNN

A useful elaboration of an RNN is a *bidirectional-RNN* (BI-RNN) (Schuster & Paliwal, 1997; Graves, 2008).[31] Consider the task of sequence tagging over a sentence $x_1, \ldots, x_n$. An RNN allows us to compute a function of the $i$th word $x_i$ based on the past – the words $x_{1:i}$ up to and including it. However, the following words $x_{i:n}$ may also be useful for prediction, as is evident by the common sliding-window approach in which the focus word is categorized based on a window of $k$ words surrounding it. Much like the RNN relaxes the Markov assumption and allows looking arbitrarily back into the past, the BI-RNN relaxes the fixed window size assumption, allowing to look arbitrarily far at both the past and the future.

Consider an input sequence $\mathbf{x_{1:n}}$. The BI-RNN works by maintaining two separate states, $\mathbf{s_i^f}$ and $\mathbf{s_i^b}$ for each input position $i$. The *forward state* $\mathbf{s_i^f}$ is based on $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_i}$, while the *backward state* $\mathbf{s_i^b}$ is based on $\mathbf{x_n}, \mathbf{x_{n-1}}, \ldots, \mathbf{x_i}$. The forward and backward states are generated by two different RNNs. The first RNN ($R^f$, $O^f$) is fed the input sequence $\mathbf{x_{1:n}}$ as is, while the second RNN ($R^b$, $O^b$) is fed the input sequence in reverse. The state representation $\mathbf{s_i}$ is then composed of both the forward and backward states.

The output at position $i$ is based on the concatenation of the two output vectors $\mathbf{y_i} = [\mathbf{y_i^f}; \mathbf{y_i^b}] = [O^f(\mathbf{s_i^f}); O^b(\mathbf{s_i^b})]$, taking into account both the past and the future. The vector $\mathbf{y_i}$ can then be used directly for prediction, or fed as part of the input to a more complex network. While the two RNNs are run independently of each other, the error gradients at position $i$ will flow both forward and backward through the two RNNs. A visual representation of the BI-RNN architecture is given in Figure 11.

The use of BI-RNNs for sequence tagging was introduced to the NLP community by Irsoy and Cardie (2014).

## 10.5 RNNs for Representing Stacks

Some algorithms in language processing, including those for transition-based parsing (Nivre, 2008), require performing feature extraction over a stack. Instead of being confined to

31. When used with a specific RNN architecture such as an LSTM, the model is called BI-LSTM.

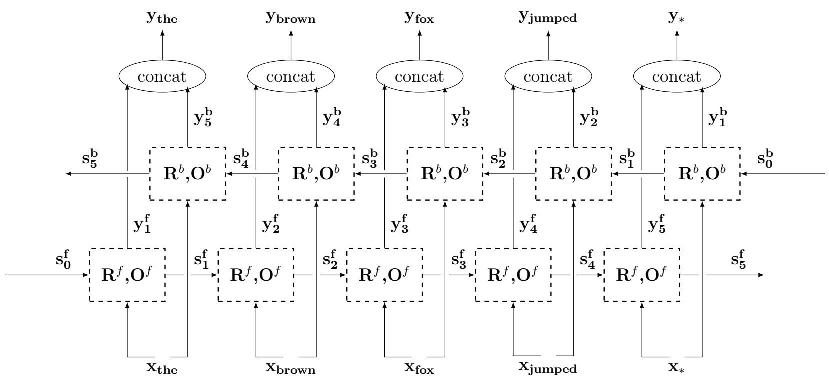

Figure 11: BI-RNN over the sentence "the brown fox jumped .".

looking at the $k$ top-most elements of the stack, the RNN framework can be used to provide a fixed-sized vector encoding of the entire stack.

The main intuition is that a stack is essentially a sequence, and so the stack state can be represented by taking the stack elements and feeding them in order into an RNN, resulting in a final encoding of the entire stack. In order to do this computation efficiently (without performing an $O(n)$ stack encoding operation each time the stack changes), the RNN state is maintained together with the stack state. If the stack was push-only, this would be trivial: whenever a new element $x$ is pushed into the stack, the corresponding vector $\mathbf{x}$ will be used together with the RNN state $\mathbf{s_i}$ in order to obtain a new state $\mathbf{s_{i+1}}$. Dealing with pop operation is more challenging, but can be solved by using the persistent-stack data-structure (Okasaki, 1999; Goldberg, Zhao, & Huang, 2013). Persistent, or immutable, data-structures keep old versions of themselves intact when modified. The persistent stack construction represents a stack as a pointer to the head of a linked list. An empty stack is the empty list. The push operation appends an element to the list, returning the new head. The pop operation then returns the parent of the head, but keeping the original list intact. From the point of view of someone who held a pointer to the previous head, the stack did not change. A subsequent push operation will add a new child to the same node. Applying this procedure throughout the lifetime of the stack results in a tree, where the root is an empty stack and each path from a node to the root represents an intermediary stack state. Figure 12 provides an example of such a tree. The same process can be applied in the computation graph construction, creating an RNN with a tree structure instead of a chain structure. Backpropagating the error from a given node will then affect all the elements that participated in the stack when the node was created, in order. Figure 13 shows the computation graph for the stack-RNN corresponding to the last state in Figure 12. This modeling approach was proposed independently by Dyer et al and Watanabe et al (Dyer et al., 2015; Watanabe & Sumita, 2015) for transition-based dependency parsing.

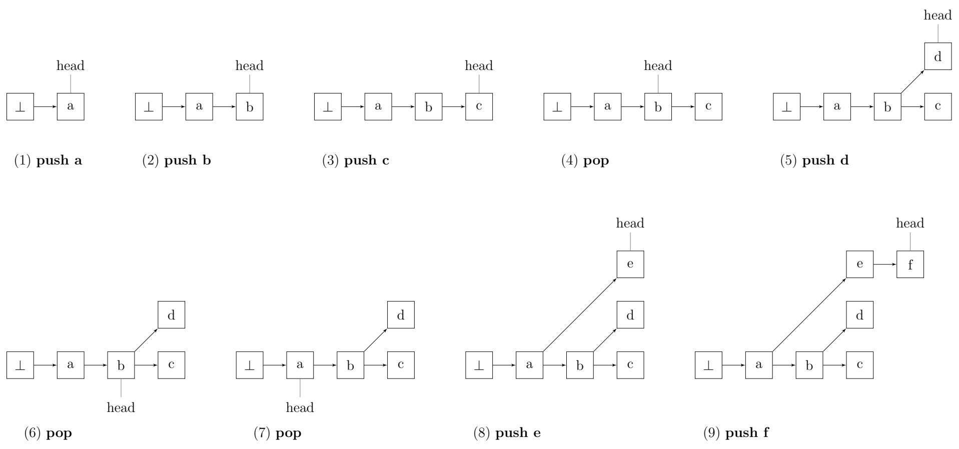

Figure 12: An immutable stack construction for the sequence of operations *push a; push b; push c; pop; push d; pop; pop; push e; push f.*

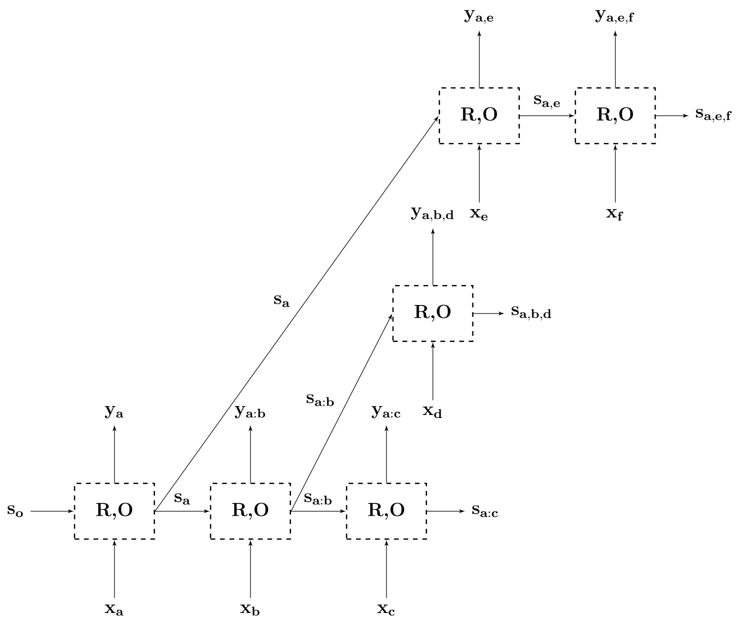

Figure 13: The stack-RNN corresponding to the final state in Figure 12.

### 10.6 A Note on Reading the Literature

Unfortunately, it is often the case that inferring the exact model form from reading its description in a research paper can be quite challenging. Many aspects of the models are not yet standardized, and different researchers use the same terms to refer to slightly different things. To list a few examples, the inputs to the RNN can be either one-hot vectors (in which case the embedding matrix is internal to the RNN) or embedded representations; The input sequence can be padded with start-of-sequence and/or end-of-sequence symbols, or not; While the output of an RNN is usually assumed to be a vector which is expected to be fed to additional layers followed by a softmax for prediction (as is the case in the presentation in this tutorial), some papers assume the softmax to be part of the RNN itself; In multi-layer RNN, the "state vector" can be either the output of the top-most layer, or a concatenation of the outputs from all layers; When using the encoder-decoder framework, conditioning on the output of the encoder can be interpreted in various different ways; and so on. On top of that, the LSTM architecture described in the next section has many small variants, which are all referred to under the common name LSTM. Some of these choices are made explicit in the papers, other require careful reading, and others still are not even mentioned, or are hidden behind ambiguous figures or phrasing.

As a reader, be aware of these issues when reading and interpret model descriptions. As a writer, be aware of these issues as well: either fully specify your model in mathematical notation, or refer to a different source in which the model is fully specified, if such a source is available. If using the default implementation from a software package without knowing the details, be explicit of that fact and specify the software package you use. In any case, don't rely solely on figures or natural language text when describing your model, as these are often ambiguous.

## 11. Concrete RNN Architectures

We now turn to present three different instantiations of the abstract *RNN* architecture discussed in the previous section, providing concrete definitions of the functions $R$ and $O$. These are the *Simple RNN* (SRNN), the *Long Short-Term Memory* (LSTM) and the *Gated Recurrent Unit* (GRU).

### 11.1 Simple RNN

The simplest RNN formulation, known as an Elman Network or Simple-RNN (S-RNN), was proposed by Elman (1990) and explored for use in language modeling by Mikolov (2012). The S-RNN takes the following form:

$$\begin{aligned}\mathbf{s_i} =& R_{SRNN}(\mathbf{s_{i-1}}, \mathbf{x_i}) = g(\mathbf{x_i}\mathbf{W^x} + \mathbf{s_{i-1}}\mathbf{W^s} + \mathbf{b})\\ \mathbf{y_i} =& O_{SRNN}(\mathbf{s_i}) = \mathbf{s_i}\end{aligned}$$

$$\mathbf{s_i}, \mathbf{y_i} \in \mathbb{R}^{d_s},\ \ \mathbf{x_i} \in \mathbb{R}^{d_x},\ \ \mathbf{W^x} \in \mathbb{R}^{d_x \times d_s},\ \ \mathbf{W^s} \in \mathbb{R}^{d_s \times d_s},\ \ \mathbf{b} \in \mathbb{R}^{d_s}$$

That is, the state at position $i$ is a linear combination of the input at position $i$ and the previous state, passed through a non-linear activation (commonly tanh or ReLU). The output at position $i$ is the same as the hidden state in that position.[32]

In spite of its simplicity, the Simple RNN provides strong results for sequence tagging (Xu et al., 2015) as well as language modeling. For comprehensive discussion on using Simple RNNs for language modeling, see the PhD thesis by Mikolov (2012).

### 11.2 LSTM

The S-RNN is hard to train effectively because of the vanishing gradients problem. Error signals (gradients) in later steps in the sequence diminish quickly in the back-propagation process, and do not reach earlier input signals, making it hard for the S-RNN to capture long-range dependencies. The Long Short-Term Memory (LSTM) architecture (Hochreiter & Schmidhuber, 1997) was designed to solve the vanishing gradients problem. The main idea behind the LSTM is to introduce as part of the state representation also "memory cells" (a vector) that can preserve gradients across time. Access to the memory cells is controlled by *gating components* – smooth mathematical functions that simulate logical gates. At each input state, a gate is used to decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forgotten. Concretely, a gate $\mathbf{g} \in [0, 1]^n$ is a vector of values in the range $[0, 1]$ that is multiplied component-wise with another vector $\mathbf{v} \in \mathbb{R}^n$, and the result is then added to another vector. The values of $\mathbf{g}$ are designed to be close to either 0 or 1, i.e. by using a sigmoid function. Indices in $\mathbf{v}$ corresponding to near-one values in $\mathbf{g}$ are allowed to pass, while those corresponding to near-zero values are blocked.

32. Some authors treat the output at position $i$ as a more complicated function of the state. In our presentation, such further transformation of the output are not considered part of the RNN, but as separate computations that are applied to the RNNs output. The distinction between the state and the output are needed for the LSTM architecture, in which not all of the state is observed outside of the RNN.

Mathematically, the LSTM architecture is defined as:[33]

$$
\begin{aligned}
\mathbf{s_j} = R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) =& [\mathbf{c_j}; \mathbf{h_j}] \\
\mathbf{c_j} =& \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\mathbf{h_j} =& \tanh(\mathbf{c_j}) \odot \mathbf{o} \\
\mathbf{i} =& \sigma(\mathbf{x_j W^{xi}} + \mathbf{h_{j-1} W^{hi}}) \\
\mathbf{f} =& \sigma(\mathbf{x_j W^{xf}} + \mathbf{h_{j-1} W^{hf}}) \\
\mathbf{o} =& \sigma(\mathbf{x_j W^{xo}} + \mathbf{h_{j-1} W^{ho}}) \\
\mathbf{g} =& \tanh(\mathbf{x_j W^{xg}} + \mathbf{h_{j-1} W^{hg}})
\end{aligned}
$$

$$\mathbf{y_j} = O_{LSTM}(\mathbf{s_j}) = \mathbf{h_j}$$

$$\mathbf{s_j} \in \mathbb{R}^{2 \cdot d_h},\ \mathbf{x_i} \in \mathbb{R}^{d_x},\ \mathbf{c_j}, \mathbf{h_j}, \mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{g} \in \mathbb{R}^{d_h},\ \mathbf{W^{x\circ}} \in \mathbb{R}^{d_x \times d_h},\ \mathbf{W^{h\circ}} \in \mathbb{R}^{d_h \times d_h},$$

The symbol $\odot$ is used to denote component-wise product. The state at time $j$ is composed of two vectors, $\mathbf{c_j}$ and $\mathbf{h_j}$, where $\mathbf{c_j}$ is the memory component and $\mathbf{h_j}$ is the output, or state, component. There are three gates, $\mathbf{i}$, $\mathbf{f}$ and $\mathbf{o}$, controlling for **i**nput, **f**orget and **o**utput. The gate values are computed based on linear combinations of the current input $\mathbf{x_j}$ and the previous state $\mathbf{h_{j-1}}$, passed through a sigmoid activation function. An update candidate $\mathbf{g}$ is computed as a linear combination of $\mathbf{x_j}$ and $\mathbf{h_{j-1}}$, passed through a tanh activation function. The memory $\mathbf{c_j}$ is then updated: the forget gate controls how much of the previous memory to keep ($\mathbf{c_{j-1}} \odot \mathbf{f}$), and the input gate controls how much of the proposed update to keep ($\mathbf{g} \odot \mathbf{i}$). Finally, the value of $\mathbf{h_j}$ (which is also the output $\mathbf{y_j}$) is determined based on the content of the memory $\mathbf{c_j}$, passed through a tanh non-linearity and controlled by the output gate. The gating mechanisms allow for gradients related to the memory part $\mathbf{c_j}$ to stay high across very long time ranges.

For further discussion on the LSTM architecture see the PhD thesis by Alex Graves (2008), as well as Chris Olah's description.[34] For an analysis of the behavior of an LSTM when used as a character-level language model, see (Karpathy et al., 2015).

LSTMs are currently the most successful type of RNN architecture, and they are responsible for many state-of-the-art sequence modeling results. The main competitor of the LSTM-RNN is the GRU, to be discussed next.

**Practical Considerations** When training LSTM networks, Jozefowicz et al (2015) strongly recommend to always initialize the bias term of the forget gate to be close to one. When applying dropout to an RNN with an LSTM, Zaremba et al (2014) found out that it is

33. There are many variants on the LSTM architecture presented here. For example, forget gates were not part of the original proposal in (Hochreiter & Schmidhuber, 1997), but are shown to be an important part of the architecture. Other variants include peephole connections and gate-tying. For an overview and comprehensive empirical comparison of various LSTM architectures see (Greff, Srivastava, Koutník, Steunebrink, & Schmidhuber, 2015).
34. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

crucial to apply dropout only on the non-recurrent connection, i.e. only to apply it between layers and not between sequence positions.

### 11.3 GRU

The LSTM architecture is very effective, but also quite complicated. The complexity of the system makes it hard to analyze, and also computationally expensive to work with. The gated recurrent unit (GRU) was recently introduced by Cho et al (2014b) as an alternative to the LSTM. It was subsequently shown by Chung et al (2014) to perform comparably to the LSTM on several (non textual) datasets.

Like the LSTM, the GRU is also based on a gating mechanism, but with substantially fewer gates and without a separate memory component.

$$\begin{aligned}\mathbf{s_j} = R_{GRU}(\mathbf{s_{j-1}}, \mathbf{x_j}) =& (\mathbf{1}-\mathbf{z}) \odot \mathbf{s_{j-1}} + \mathbf{z} \odot \mathbf{h} \\ \mathbf{z} =& \sigma(\mathbf{x_j}\mathbf{W^{xz}} + \mathbf{h_{j-1}}\mathbf{W^{hz}}) \\ \mathbf{r} =& \sigma(\mathbf{x_j}\mathbf{W^{xr}} + \mathbf{h_{j-1}}\mathbf{W^{hr}}) \\ \mathbf{h} =& \tanh(\mathbf{x_j}\mathbf{W^{xh}} + (\mathbf{h_{j-1}} \odot \mathbf{r})\mathbf{W^{hg}})\end{aligned}$$

$$\mathbf{y_j} = O_{LSTM}(\mathbf{s_j}) = \mathbf{s_j}$$

$$\mathbf{s_j} \in \mathbb{R}^{d_h},\ \mathbf{x_i} \in \mathbb{R}^{d_x},\ \mathbf{z}, \mathbf{r}, \mathbf{h} \in \mathbb{R}^{d_h},\ \mathbf{W^{x\circ}} \in \mathbb{R}^{d_x \times d_h},\ \mathbf{W^{h\circ}} \in \mathbb{R}^{d_h \times d_h},$$

One gate ($\mathbf{r}$) is used to control access to the previous state $\mathbf{s_{j-1}}$ and compute a proposed update $\mathbf{h}$. The updated state $\mathbf{s_j}$ (which also serves as the output $\mathbf{y_j}$) is then determined based on an interpolation of the previous state $\mathbf{s_{j-1}}$ and the proposal $\mathbf{h}$, where the proportions of the interpolation are controlled using the gate $\mathbf{z}$.

The GRU was shown to be effective in language modeling and machine translation. However, the jury between the GRU, the LSTM and possible alternative RNN architectures is still out, and the subject is actively researched. For an empirical exploration of the GRU and the LSTM architectures, see (Jozefowicz et al., 2015).

### 11.4 Other Variants

The gated architectures of the LSTM and the GRU help in alleviating the vanishing gradients problem of the Simple RNN, and allow these RNNs to capture dependencies that span long time ranges. Some researchers explore simpler architectures than the LSTM and the GRU for achieving similar benefits.

Mikolov et al (2014) observed that the matrix multiplication $\mathbf{s_{i-1}}\mathbf{W^s}$ coupled with the nonlinearity $g$ in the update rule $R$ of the Simple RNN causes the state vector $\mathbf{s_i}$ to undergo large changes at each time step, prohibiting it from remembering information over long time periods. They propose to split the state vector $\mathbf{s_i}$ into a slow changing component $\mathbf{c_i}$ ("context units") and a fast changing component $\mathbf{h_i}$.[35] The slow changing component $\mathbf{c_i}$ is

35. We depart from the notation in (Mikolov et al., 2014) and reuse the symbols used in the LSTM description.

updated according to a linear interpolation of the input and the previous component: $\mathbf{c_i} = (1-\alpha)\mathbf{x_i}\mathbf{W^{x1}} + \alpha\mathbf{c_{i-1}}$, where $\alpha \in (0,1)$. This update allows $\mathbf{c_i}$ to accumulate the previous inputs. The fast changing component $\mathbf{h_i}$ is updated similarly to the Simple RNN update rule, but changed to take $\mathbf{c_i}$ into account as well:[36] $\mathbf{h_i} = \sigma(\mathbf{x_i}\mathbf{W^{x2}} + \mathbf{h_{i-1}}\mathbf{W^h} + \mathbf{c_i}\mathbf{W^c})$. Finally, the output $\mathbf{y_i}$ is the concatenation of the slow and the fast changing parts of the state: $\mathbf{y_i} = [\mathbf{c_i}; \mathbf{h_i}]$. Mikolov et al demonstrate that this architecture provides competitive perplexities to the much more complex LSTM on language modeling tasks.

The approach of Mikolov et al can be interpreted as constraining the block of the matrix $\mathbf{W^s}$ in the S-RNN corresponding to $\mathbf{c_i}$ to be a multiply of the identity matrix (see Mikolov et al (2014) for the details). Le et al (Le, Jaitly, & Hinton, 2015) propose an even simpler approach: set the activation function of the S-RNN to a ReLU, and initialize the biases $\mathbf{b}$ as zeroes and the matrix $\mathbf{W^s}$ as the identify matrix. This causes an untrained RNN to copy the previous state to the current state, add the effect of the current input $\mathbf{x_i}$ and set the negative values to zero. After setting this initial bias towards state copying, the training procedure allows $\mathbf{W^s}$ to change freely. Le et al demonstrate that this simple modification makes the S-RNN comparable to an LSTM with the same number of parameters on several tasks, including language modeling.

36. The update rule diverges from the S-RNN update rule also by fixing the non-linearity to be a sigmoid function, and by not using a bias term. However, these changes are not discussed as central to the proposal.