CHAPTER 15

Concrete Recurrent Neural Network Architectures

After describing the RNN abstraction, we are now in place to discuss specific instantiations of it. Recall that we are interested in a recursive function $\boldsymbol{s_i} = R(\boldsymbol{x_i}, \boldsymbol{s_{i-1}})$ such that $\boldsymbol{s_i}$ encodes the sequence $\boldsymbol{x_{1:n}}$. We will present several concrete instantiations of the abstract RNN architecture, providing concrete definitions of the functions R and O. These include the *Simple RNN* (S-RNN), the *Long Short-Term Memory* (LSTM) and the *Gated Recurrent Unit* (GRU).

15.1 CBOW AS AN RNN

On particularly simple choice of R is the addition function:

$$\begin{aligned} \boldsymbol{s_i} &= R_{\text{CBOW}}(\boldsymbol{x_i}, \boldsymbol{s_{i-1}}) = \boldsymbol{s_{i-1}} + \boldsymbol{x_i} \\ \boldsymbol{y_i} &= O_{\text{CBOW}}(\boldsymbol{s_i}) = \boldsymbol{s_i} \end{aligned} \tag{15.1}$$

$$\boldsymbol{s_i}, \boldsymbol{y_i} \in \mathbb{R}^{d_s}, \quad \boldsymbol{x_i} \in \mathbb{R}^{d_s}.$$

Following the definition in Equation (15.1), we get the continuous-bag-of-words model: the state resulting from inputs $\boldsymbol{x_{1:n}}$ is the sum of these inputs. While simple, this instantiation of the RNN ignores the sequential nature of the data. The Elman RNN, described next, adds dependence on the sequential ordering of the elements.[1]

15.2 SIMPLE RNN

The simplest RNN formulation that is sensitive to the ordering of elements in the sequence is known as an Elman Network or Simple-RNN (S-RNN). The S-RNN was proposed by Elman [1990] and explored for use in language modeling by Mikolov [2012]. The S-RNN takes the following form:

$$\begin{aligned} \boldsymbol{s_i} &= R_{\text{SRNN}}(\boldsymbol{x_i}, \boldsymbol{s_{i-1}}) = g(\boldsymbol{s_{i-1}} \boldsymbol{W^s} + \boldsymbol{x_i} \boldsymbol{W^x} + \boldsymbol{b}) \\ \boldsymbol{y_i} &= O_{\text{SRNN}}(\boldsymbol{s_i}) = \boldsymbol{s_i} \end{aligned} \tag{15.2}$$

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

[1]The view of the CBOW representation as an RNN is not a common one in the literature. However, we find it to be a good stepping stone into the Elman RNN definition. It is also useful to have the simple CBOW encoder in the same framework as the RNNs as it can also serve the role of an encoder in a conditioned generation network such as those described in Chapter 17.

That is, the state $\boldsymbol{s_{i-1}}$ and the input $\boldsymbol{x_i}$ are each linearly transformed, the results are added (together with a bias term) and then passed through a nonlinear activation function g (commonly tanh or ReLU). The output at position i is the same as the hidden state in that position.[2]

An equivalent way of writing Equation (15.2) is Equation (15.3), both are used in the literature:

$$\begin{aligned} \boldsymbol{s_i} &= R_{\text{SRNN}}(\boldsymbol{x_i}, \boldsymbol{s_{i-1}}) = g([\boldsymbol{s_{i-1}}; \boldsymbol{x_i}]\boldsymbol{W} + \boldsymbol{b}) \\ \boldsymbol{y_i} &= O_{\text{SRNN}}(\boldsymbol{s_i}) = \boldsymbol{s_i} \end{aligned} \tag{15.3}$$

$$\boldsymbol{s_i}, \boldsymbol{y_i} \in \mathbb{R}^{d_s}, \;\; \boldsymbol{x_i} \in \mathbb{R}^{d_x}, \;\; \boldsymbol{W} \in \mathbb{R}^{(d_x+d_s)\times d_s}, \;\; \boldsymbol{b} \in \mathbb{R}^{d_s}.$$

The S-RNN is only slightly more complex than the CBOW, with the major difference being the nonlinear activation function g. However, this difference is a crucial one, as adding the linear transformation followed by the nonlinearity makes the network sensitive to the order of the inputs. Indeed, 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 Ph.D. thesis by Mikolov [2012].

15.3 GATED ARCHITECTURES

The S-RNN is hard to train effectively because of the vanishing gradients problem [Pascanu et al., 2012]. 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. Gating-based architectures, such as the LSTM [Hochreiter and Schmidhuber, 1997] and the GRU [Cho et al., 2014b] are designed to solve this deficiency.

Consider the RNN as a general purpose computing device, where the state $\boldsymbol{s_i}$ represents a finite memory. Each application of the function R reads in an input $\boldsymbol{x_{i+1}}$, reads in the current memory $\boldsymbol{s_i}$, operates on them in some way, and writes the result into memory, resulting in a new memory state $\boldsymbol{s_{i+1}}$. Viewed this way, an apparent problem with the S-RNN architecture is that the memory access is not controlled. At each step of the computation, the entire memory state is read, and the entire memory state is written.

How does one provide more controlled memory access? Consider a binary vector $\boldsymbol{g} \in \{0,1\}^n$. Such a vector can act as a *gate* for controlling access to n-dimensional vectors, using the hadamard-product operation $\boldsymbol{x} \odot \boldsymbol{g}$:[3] Consider a memory $\boldsymbol{s} \in \mathbb{R}^d$, an input $\boldsymbol{x} \in \mathbb{R}^d$ and a gate $\boldsymbol{g} \in 0, 1^d$. The computation $\boldsymbol{s'} \leftarrow \boldsymbol{g} \odot \boldsymbol{x} + (\boldsymbol{1} - \boldsymbol{g}) \odot (\boldsymbol{s})$ "reads" the entries in $\boldsymbol{x}$ that correspond to the 1 values in $\boldsymbol{g}$, and writes them to the new memory $\boldsymbol{s'}$. Then, locations that weren't

[2]Some authors treat the output at position i as a more complicated function of the state, e.g., a linear transformation, or an MLP. 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.

[3]The hadamard-product is a fancy name for element-wise multiplication of two vectors: the hadamard product $\boldsymbol{x} = \boldsymbol{u} \odot \boldsymbol{v}$ results in $\boldsymbol{x}_{[i]} = \boldsymbol{u}_{[i]} \cdot \boldsymbol{v}_{[i]}$.

read to are copied from the memory $\boldsymbol{s}$ to the new memory $\boldsymbol{s}'$ through the use of the gate $(1 - \boldsymbol{g})$. Figure 15.1 shows this process for updating the memory with positions 2 and 5 from the input.

$$\underset{\mathbf{s}'}{\begin{bmatrix} 8 \\ 11 \\ 3 \\ 7 \\ 5 \\ 15 \end{bmatrix}} \leftarrow \underset{\mathbf{g}}{\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}} \odot \underset{\mathbf{x}}{\begin{bmatrix} 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{bmatrix}} + \underset{(\mathbf{1}-\mathbf{g})}{\begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}} \odot \underset{\mathbf{s}}{\begin{bmatrix} 8 \\ 9 \\ 3 \\ 7 \\ 5 \\ 8 \end{bmatrix}}$$

Figure 15.1: Using binary gate vector $\boldsymbol{g}$ to control access to memory $\boldsymbol{s}'$.

The gating mechanism described above can serve as a building block in our RNN: gate vectors can be used to control access to the memory state $\boldsymbol{s}_i$. However, we are still missing two important (and related) components: the gates should not be static, but be controlled by the current memory state and the input, and their behavior should be learned. This introduced an obstacle, as learning in our framework entails being differentiable (because of the backpropagation algorithm) and the binary 0-1 values used in the gates are not differentiable.[4]

A solution to the above problem is to approximate the hard gating mechanism with a soft—but differentiable—gating mechanism. To achieve these *differentiable gates*, we replace the requirement that $\boldsymbol{g} \in \{0, 1\}^n$ and allow arbitrary real numbers, $\boldsymbol{g}' \in \mathbb{R}^n$, which are then pass through a sigmoid function $\sigma(\boldsymbol{g}')$. This bounds the value in the range $(0, 1)$, with most values near the borders. When using the gate $\sigma(\boldsymbol{g}') \odot \boldsymbol{x}$, indices in $\boldsymbol{x}$ corresponding to near-one values in $\sigma(\boldsymbol{g}')$ are allowed to pass, while those corresponding to near-zero values are blocked. The gate values can then be conditioned on the input and the current memory, and trained using a gradient-based method to perform a desired behavior.

This controllable gating mechanism is the basis of the LSTM and the GRU architectures, to be defined next: at each time step, differentiable gating mechanisms decide which parts of the inputs will be written to memory, and which parts of memory will be overwritten (forgotten). This rather abstract description will be made concrete in the next sections.

15.3.1 LSTM

The Long Short-Term Memory (LSTM) architecture [Hochreiter and Schmidhuber, 1997] was designed to solve the vanishing gradients problem, and is the first to introduce the gating mechanism. The LSTM architecture explicitly splits the state vector $\boldsymbol{s}_i$ into two halves, where one half

[4]It is in principle possible to learn also models with non-differentiable components such as binary gates using reinforcement-learning techniques. However, as the time of this writing such techniques are brittle to train. Reinforcement learning techniques are beyond the scope of this book.

is treated as "memory cells" and the other is working memory. The memory cells are designed to preserve the memory, and also the error gradients, across time, and are controlled through *differentiable 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. Mathematically, the LSTM architecture is defined as:[5]

$$\begin{aligned}
\boldsymbol{s_j} = R_{\text{LSTM}}(\boldsymbol{s_{j-1}}, \boldsymbol{x_j}) &= [\boldsymbol{c_j}; \boldsymbol{h_j}] \\
\boldsymbol{c_j} &= \boldsymbol{f} \odot \boldsymbol{c_{j-1}} + \boldsymbol{i} \odot \boldsymbol{z} \\
\boldsymbol{h_j} &= \boldsymbol{o} \odot \tanh(\boldsymbol{c_j}) \\
\boldsymbol{i} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xi}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hi}}) \\
\boldsymbol{f} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xf}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hf}}) \\
\boldsymbol{o} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xo}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{ho}}) \\
\boldsymbol{z} &= \tanh(\boldsymbol{x_j} \boldsymbol{W^{xz}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hz}})
\end{aligned} \tag{15.4}$$

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

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

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

For further discussion on the LSTM architecture see the Ph.D. thesis by Alex Graves [2008], as well as Chris Olah's description.[6] For an analysis of the behavior of an LSTM when used as a character-level language model, see Karpathy et al. [2015].

[5]There are many variants on the LSTM architecture presented here. For example, forget gates were not part of the original proposal in Hochreiter and 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 et al. [2015].

[6]http://colah.github.io/posts/2015-08-Understanding-LSTMs/

The vanishing gradients problem in Recurrent Neural Networks and its Solution Intuitively, recurrent neural networks can be thought of as very deep feed-forward networks, with shared parameters across different layers. For the Simple-RNN [Equation (15.3)], the gradients then include repeated multiplication of the matrix $\boldsymbol{W}$, making it very likely for the values to vanish or explode. The gating mechanism mitigate this problem to a large extent by getting rid of this repeated multiplication of a single matrix.

For further discussion of the exploding and vanishing gradient problem in RNNs, see Section 10.7 in Bengio et al. [2016]. For further explanation of the motivation behind the gating mechanism in the LSTM (and the GRU) and its relation to solving the vanishing gradient problem in recurrent neural networks, see Sections 4.2 and 4.3 in the detailed course notes of Cho [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.

15.3.2 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.

$$\begin{aligned}\boldsymbol{s_j} = R_{\text{GRU}}(\boldsymbol{s_{j-1}}, \boldsymbol{x_j}) =& (\boldsymbol{1} - \boldsymbol{z}) \odot \boldsymbol{s_{j-1}} + \boldsymbol{z} \odot \tilde{\boldsymbol{s_j}} \\ \boldsymbol{z} =& \sigma(\boldsymbol{x_j} \boldsymbol{W^{xz}} + \boldsymbol{s_{j-1}} \boldsymbol{W^{sz}}) \\ \boldsymbol{r} =& \sigma(\boldsymbol{x_j} \boldsymbol{W^{xr}} + \boldsymbol{s_{j-1}} \boldsymbol{W^{sr}}) \\ \tilde{\boldsymbol{s_j}} =& \tanh(\boldsymbol{x_j} \boldsymbol{W^{xs}} + (\boldsymbol{r} \odot \boldsymbol{s_{j-1}}) \boldsymbol{W^{sg}}) \end{aligned} \tag{15.5}$$

$$\boldsymbol{y_j} = O_{\text{GRU}}(\boldsymbol{s_j}) = \boldsymbol{s_j}$$

$$\boldsymbol{s_j}, \tilde{\boldsymbol{s_j}} \in \mathbb{R}^{d_s}, \quad \boldsymbol{x_i} \in \mathbb{R}^{d_x}, \quad \boldsymbol{z}, \boldsymbol{r} \in \mathbb{R}^{d_s}, \quad \boldsymbol{W^{x\circ}} \in \mathbb{R}^{d_x \times d_s}, \quad \boldsymbol{W^{s\circ}} \in \mathbb{R}^{d_s \times d_s}.$$

One gate ($\boldsymbol{r}$) is used to control access to the previous state $\boldsymbol{s_{j-1}}$ and compute a proposed update $\tilde{\boldsymbol{s_j}}$. The updated state $\boldsymbol{s_j}$ (which also serves as the output $\boldsymbol{y_j}$) is then determined based on an interpolation of the previous state $\boldsymbol{s_{j-1}}$ and the proposal $\tilde{\boldsymbol{s_j}}$, where the proportions of the interpolation are controlled using the gate $\boldsymbol{z}$.[7]

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

15.4 OTHER VARIANTS

Improvements to non-gated architectures 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 $\boldsymbol{s_{i-1}}\boldsymbol{W^s}$ coupled with the nonlinearity g in the update rule R of the Simple RNN causes the state vector $\boldsymbol{s_i}$ to undergo large changes at each time step, prohibiting it from remembering information over long time periods. They propose to split the state vector $\boldsymbol{s_i}$ into a slow changing component $\boldsymbol{c_i}$ ("context units") and a fast changing component $\boldsymbol{h_i}$.[8] The slow changing component $\boldsymbol{c_i}$ is updated according to a linear interpolation of the input and the previous component: $\boldsymbol{c_i} = (1-\alpha)\boldsymbol{x_i}\boldsymbol{W^{x1}} + \alpha\boldsymbol{c_{i-1}}$, where $\alpha \in (0,1)$. This update allows $\boldsymbol{c_i}$ to accumulate the previous inputs. The fast changing component $\boldsymbol{h_i}$ is updated similarly to the Simple RNN update rule, but changed to take $\boldsymbol{c_i}$ into account as well:[9] $\boldsymbol{h_i} = \sigma(\boldsymbol{x_i}\boldsymbol{W^{x2}} + \boldsymbol{h_{i-1}}\boldsymbol{W^h} + \boldsymbol{c_i}\boldsymbol{W^c})$. Finally, the output $\boldsymbol{y_i}$ is the concatenation of the slow and the fast changing parts of the state: $\boldsymbol{y_i} = [\boldsymbol{c_i};\boldsymbol{h_i}]$. 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 $\boldsymbol{W^s}$ in the S-RNN corresponding to $\boldsymbol{c_i}$ to be a multiple of the identity matrix (see Mikolov et al. [2014] for the details). Le et al. [2015] propose an even simpler approach: set the activation function of the S-RNN to a ReLU, and initialize the biases $\boldsymbol{b}$ as zeroes and the matrix $\boldsymbol{W^s}$ 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 $\boldsymbol{x_i}$ and set the negative values to zero. After setting this initial bias toward state copying, the training procedure allows $\boldsymbol{W^s}$ 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.

[7]The states $\boldsymbol{s}$ are often called $\boldsymbol{h}$ in the GRU literature.

[8]We depart from the notation in Mikolov et al. [2014] and reuse the symbols used in the LSTM description.

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

Beyond differential gates The gating mechanism is an example of adapting concepts from the theory of computation (memory access, logical gates) into differentiable—and hence gradient-trainable—systems. There is considerable research interest in creating neural network architectures to simulate and implement further computational mechanisms, allowing better and more fine grained control. One such example is the work on a *differentiable stack* [Grefenstette et al., 2015] in which a stack structure with push and pop operations is controlled using an end-to-end differentiable network, and the *neural turing machine* [Graves et al., 2014] which allows read and write access to content-addressable memory, again, in a differentiable system. While these efforts are yet to result in robust and general-purpose architectures that can be used in non-toy language processing applications, they are well worth keeping an eye on.

15.5 DROPOUT IN RNNS

Applying dropout to RNNs can be a bit tricky, as dropping different dimensions at different time steps harms the ability of the RNN to carry informative signals across time. This prompted Pham et al. [2013], Zaremba et al. [2014] to suggest applying dropout only on the non-recurrent connection, i.e., only to apply it between layers in deep-RNNs and not between sequence positions.

More recently, following a variational analysis of the RNN architecture, Gal [2015] suggests applying dropout to all the components of the RNN (both recurrent and non-recurrent), but crucially retain the same dropout mask across time steps. That is, the dropout masks are sampled once per sequence, and not once per time step. Figure 15.2 contrasts this form of dropout ("variational RNN") with the architecture proposed by Pham et al. [2013], Zaremba et al. [2014].

The variational RNN dropout method of Gal is the current best-practice for applying dropout in RNNs.

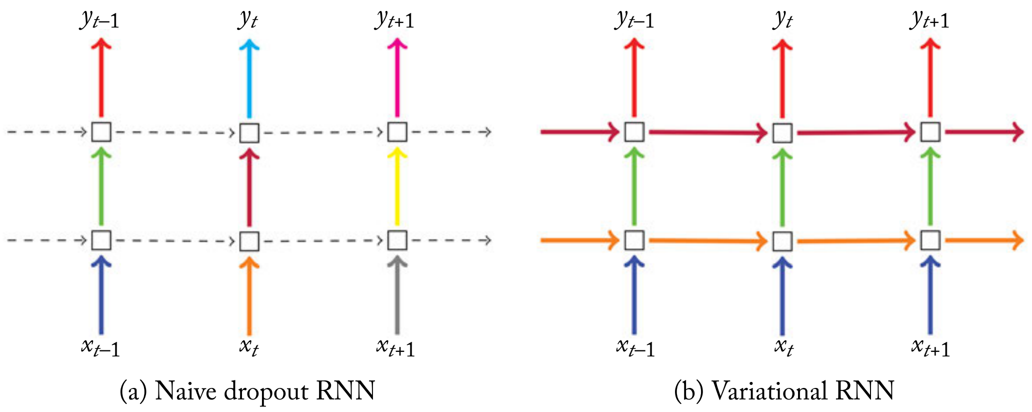

Figure 15.2: Gal's proposal for RNN dropout (b), vs. the previous suggestion by Pham et al. [2013], Zaremba et al. [2014] (a). Figure from Gal [2015], used with permission. Each square represents an RNN unit, with horizontal arrows representing time dependence (recurrent connections). Vertical arrows represent the input and output to each RNN unit. Colored connections represent dropped-out inputs, with different colors corresponding to different dropout masks. Dashed lines correspond to standard connections with no dropout. Previous techniques (naive dropout, left) use different masks at different time steps, with no dropout on the recurrent layers. Gal's proposed technique (Variational RNN, right) uses the same dropout mask at each time step, including the recurrent layers.