Chapter 10

Sequence Modeling: Recurrent and Recursive Nets

Recurrent neural networks (Rumelhart *et al.*, 1986c), or RNNs[1], are the main tool for handling sequential data, which involves variable length inputs or outputs. To go from multi-layer networks to recurrent networks, we need take advantage of one of the early ideas found in machine learning and statistical models of the 80's: *sharing parameters*[2] across different parts of a model. Parameter sharing makes it possible to extend and apply the model to examples of different forms (different lengths, here) and generalize across them. If we had separate parameters for each value of the time index, we could not generalize to sequence lengths not seen during training, nor share statistical strength across different sequence lengths and across different positions in time. Such sharing is particularly important when, like in speech, the input sequence can be stretched non-linearly, i.e., some parts (like vowels) may last longer in different examples. It means that the absolute time step at which an event occurs is meaningless: it only makes sense to consider the event in some context that somehow captures what has happened before. Compared to a multi-layer network, the weights in an RNN are shared across different instances of the artificial neurons, each associated with different time steps. This allows us to apply the network to input sequences of different lengths because the same weights are re-used at each time step. This idea is made more explicit in the early work on *time-delay neural networks* (Lang and Hinton, 1988; Waibel *et al.*, 1989), where a fully connected network is replaced by one with local connections that are shared across different temporal instances of

[1] Unfortunately, the RNN acronym is sometimes also used for denoting Recursive Neural Networks. However, since the RNN acronym has been around for much longer, we suggest keeping this acronym for recurrent neural networks.

[2] see Section 7.8 for an introduction to the concept of parameter sharing

the hidden units. Such networks are among the ancestors of *convolutional neural networks*, covered in more detail in Section 9. Recurrent nets are described in more detail below, in Section 10.2, after introducing the key idea of unfolding a computational graph. As shown in Section 10.1 below, the computational graph (a notion previously introduced in Section 6.4.3 in the case of MLPs) associated with a recurrent network is structured like a chain. Recurrent neural networks have been generalized into *recursive neural networks*, in which the unfolded structure can be more general than a chain. With recursive networks, the unfolded network has the shape of a tree. Recursive neural networks are discussed in more detail in Section 10.6. For a good textbook on RNNs, see Graves (2012).

10.1 Unfolding Flow Graphs and Sharing Parameters

A flow graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss. Please refer to Section 6.4.3 for a general introduction. In this section we explain the idea of *unfolding* a recursive or recurrent computation into a flow graph that has a repetitive structure, typically corresponding to a chain of events.

For example, consider the classical form of a dynamical system:

$$\boldsymbol{s}_t = f_\theta(\boldsymbol{s}_{t-1}) \tag{10.1}$$

where $\boldsymbol{s}_t$ is called the state of the system. The unfolded flow graph of such a system looks like in Fig. 10.1.

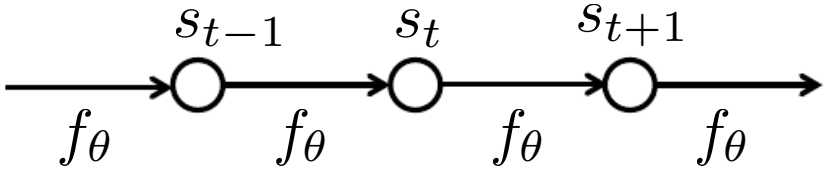

Figure 10.1: Classical dynamical system equation 10.1 illustrated as an unfolded flow graph. Each node represents the state at some time t and function f_θ maps the state at t to the state at $t+1$. The same parameters (the same function f_θ) is used for all time steps.

As another example, let us consider a dynamical system driven by an external signal $\boldsymbol{x}_t$,

$$\boldsymbol{s}_t = f_\theta(\boldsymbol{s}_{t-1}, \boldsymbol{x}_t) \tag{10.2}$$

illustrated in Fig. 10.2, where we see that the state now contains information about the whole past sequence, i.e., the above equation implicitly defines a function

$$\boldsymbol{s}_t = g_t(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_2, \boldsymbol{x}_1) \tag{10.3}$$

which maps the whole past sequence $(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_2, \boldsymbol{x}_1)$ to the current state. Equation 10.2 is actually part of the definition of a recurrent net. We can think of $\boldsymbol{s}_t$ as a kind of summary of the past sequence of inputs up to t. Note that this summary is in general necessarily lossy, since it maps an arbitrary length sequence $(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_2, \boldsymbol{x}_1)$ to a fixed length vector $\boldsymbol{s}_t$. Depending on the training criterion, this summary might selectively keep some aspects of the past sequence with more precision than other aspects. For example, if the RNN is used in statistical language modeling, typically to predict the next word given previous words, it may not be necessary to distinctly keep track of all the bits of information, only those required to predict the rest of the sentence. The most demanding situation is when we ask $\boldsymbol{s}_t$ to be rich enough to allow one to approximately recover the input sequence, as in auto-encoder frameworks (Chapter 15).

If we had to define a different function g_t for each possible sequence length (imagine a separate neural network, each with a different input size), each with its own parameters, we would not get any generalization to sequences of a size not seen in the training set. Furthermore, one would need to see a lot more training examples, because a separate model would have to be trained for each sequence length, and it would need a lot more parameters (proportionally to the size of the input sequence). It could not generalize what it learns from what happens at a position t to what could happen at a position $t' \neq t$. By instead defining the state through a recurrent formulation as in Eq. 10.2, the same parameters are used for any sequence length, allowing much better generalization properties.

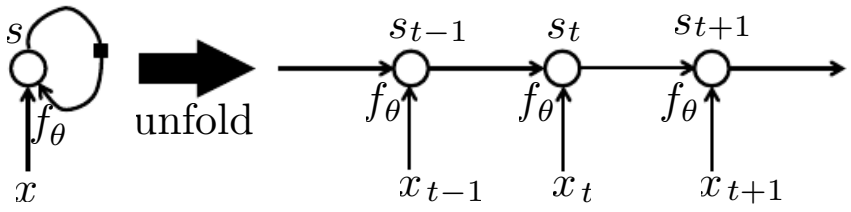

Figure 10.2: Left: input processing part of a recurrent neural network, seen as a circuit. The black square indicates a delay of 1 time step. Right: the same seen as an unfolded flow graph, where each node is now associated with one particular time instance.

Equation 10.2 can be drawn in two different ways. One is in a way that is inspired by how a physical implementation (such as a real neural network) might look like, i.e., like a circuit which operates in real time, as in the left of Fig. 10.2. The other is as a flow graph, in which the computations occurring at different time steps in the circuit are unfolded as different nodes of the flow graph, as in the right of Fig. 10.2. What we call *unfolding* is the operation that maps a circuit as in the left side of the figure to a flow graph with repeated pieces as

in the right side. Note how the unfolded graph now has a size that depends on the sequence length. The black square indicates a delay of 1 time step on the recurrent connection, from the state at time t to the state at time $t + 1$.

The other important observation to make from Fig. 10.2 is that *the same parameters (θ) are shared* over different parts of the graph, corresponding here to different time steps.

10.2 Recurrent Neural Networks

Armed with the ideas introduced in the previous section, we can design a wide variety of recurrent circuits, which are compact and simple to understand visually. As we will explain, we can automatically obtain their equivalent unfolded graph, which are useful computationally and also help focus on the idea of information flow forward in time (computing outputs and losses) and backward in time (computing gradients).

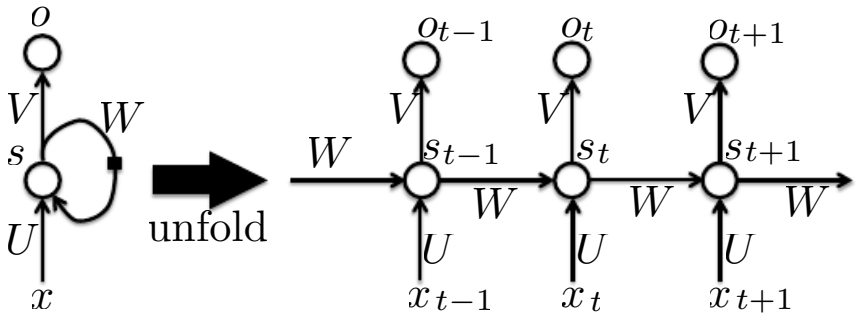

Figure 10.3: Left: ordinary recurrent network circuit with hidden-to-hidden recurrence, seen as a circuit, with weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$ for the three different kinds of connections (input-to-hidden, hidden-to-output and hidden-to-hidden, respectively). Each circle indicates a whole vector of activations. Right: the same seen as an time-unfolded flow graph, where each node is now associated with one particular time instance.

Some of the early circuit designs for recurrent neural networks are illustrated in Figs. 10.3, 10.4 and 10.6. Fig. 10.3 shows the simplest recurrent network architecture, whose equations are laid down below in Eq. 10.4. This kind of neural network has been shown to be a universal approximation machine for discrete sequences. This is loosely analogous to the universal approximator theorem for feedforward neural networks. Specifically, any function computable by a Turing machine can be computed by a recurrent network of a finite size. The output can be read from the RNN after a number of time steps that is asymptotically linear in the number of time steps used by the Turing machine and asymptotically linear in the length of the input (Siegelmann and Sontag, 1991; Siegelmann, 1995; Siegelmann and Sontag, 1995; Hyotyniemi, 1996). Note that the functions

computable by a Turing machine are discrete, so these results regard exact implementation of the function, not approximations. The outputs of the RNN must be discretized so that it can take a binary sequence in input and produce a binary sequence in output. Note also that the "input" of the Turing machine is a specification of the function to be computed, which is why only a finite-size recurrent net (Siegelmann and Sontag (1995) use 886 units) is sufficient for all problems. The RNN can simulate an unbounded stack by representing its activations and weights with rational numbers of unbounded precision.

On the other hand, the network with *output recurrence* (shown in Figure 10.4) and no hidden-to-hidden recurrence is strictly less powerful, since it requires that the output units capture the full state, which means that they represent an appropriate summary of the past as necessary to predict the future. Although this may be appropriate in some cases where the full state of the system is observed and can be provided as a target, this assumption is generally too strong. The advantage of this assumption, though, is that with the maximum likelihood criterion, all the time steps are decoupled and no back-propagation through time is necessary.

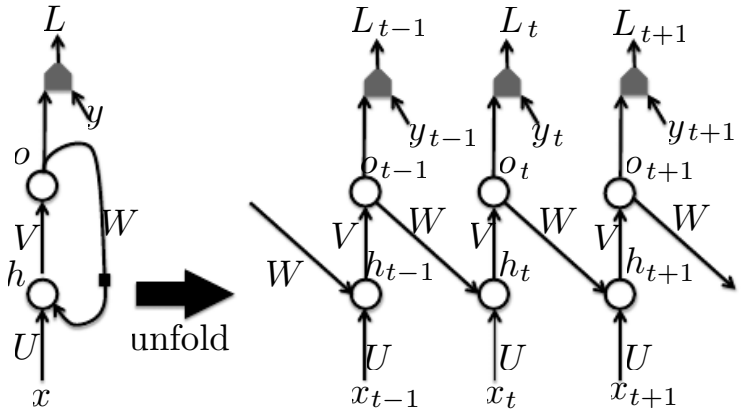

Figure 10.4: Left: RNN circuit whose recurrence is only through the output. Right: computational flow graph unfolded in time. At each t, the input is $\boldsymbol{x}_t$, the hidden layer activations $\boldsymbol{h}_t$, the output $\boldsymbol{o}_t$, the target y_t and the loss L_t. Such an RNN is less powerful (can express a smaller set of functions) than those in the family represented by Fig. 10.3 but may be easier to train because they can exploit "teacher forcing", i.e., constraining some of the units involved in the recurrent loop (here the output units) to take some target values during training. This architecture is less powerful because the only state information (carrying the information about the past) is the previous *prediction*. Unless the prediction is very high-dimensional and rich, this will usually miss important information from the past.

The recurrent network of Fig. 10.3 corresponds to the following forward propagation equations, if we assume that hyperbolic tangent non-linearities are used

in the hidden units and softmax is used in output (for classification problems):

$$\begin{aligned} \boldsymbol{a}_t &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{s}_{t-1} + \boldsymbol{U}\boldsymbol{x}_t \\ \boldsymbol{s}_t &= \tanh(\boldsymbol{a}_t) \\ \boldsymbol{o}_t &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{s}_t \\ \boldsymbol{p}_t &= \text{softmax}(\boldsymbol{o}_t) \end{aligned} \tag{10.4}$$

where the parameters are the bias vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ along with the weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$, respectively for input-to-hidden, hidden-to-output and hidden-to-hidden connections. This is an example of a recurrent network that maps an input sequence to an output sequence of the same length. The total loss for a given input/target sequence pair $(\boldsymbol{x}, \boldsymbol{y})$ would then be just the sum of the losses over all the time steps, e.g.,

$$L(\boldsymbol{x}, \boldsymbol{y}) = \sum_t L_t = \sum_t -\log p_{t,y_t} \tag{10.5}$$

where y_t is the category that should be associated with time step t in the output sequence.

The circuit in Fig. 10.4 has only its output (the prediction of the previous target) as state, as a memory of the past. This potentially limits its expressive power, but also makes it easier to train. Indeed, the "intermediate state" of the corresponding unfolded deep network is not hidden anymore: targets are available to guide this intermediate representation, which should make it easier to train. In general, the state of the RNN must be sufficiently rich to store a summary of the past sequence that is enough to properly predict the future target values. Constraining the state to be the visible variable y_t itself is therefore in general not enough to learn most tasks of interest, unless, given the sequence of inputs $\boldsymbol{x}_t$, y_t contains all the required information about the past y's that is required to predict the future y's.

Teacher forcing is the training process in which the fed back inputs are not the predicted outputs but the targets themselves, as illustrated in Fig. 10.5. The disadvantage of strict teacher forcing is that if the network is going to be later used in an *open-loop* mode, i.e., with the network outputs (or samples from the output distribution) fed back as input, then the kind of inputs that the network will have seen during training could be quite different from the kind of inputs that it will see at test time when the network is run in generative mode, potentially yielding very poor generalizations. One way to mitigate this problem is to train with both teacher-forced inputs and with free-running inputs, e.g., predicting the correct target a number of steps in the future through the unfolded recurrent output-to-input paths. In this way, the network can learn to take into account

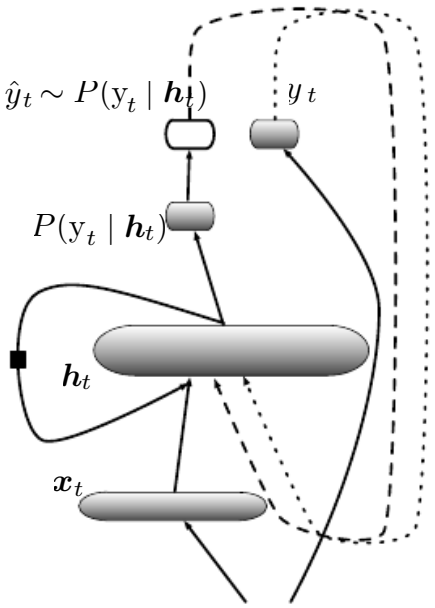

Figure 10.5: Illustration of *teacher forcing* for RNNs, which comes out naturally from the log-likelihood training objective (such as in Eq. 10.5). There are two ways in which the output variable can be fed back as input to update the next state $\boldsymbol{h}_t$: what is fed back is either the sample $\hat{y}_t$ generated from the RNN model's output distribution $P(\mathrm{y}_t \mid \boldsymbol{h}_t)$ (dashed arrow) or the actual "correct" output y_t coming from the training data (dotted arrow) $(\boldsymbol{x}_t, y_t)$. The former is what is done when one *generates a sequence* from the model. The latter is teacher forcing, done during training.

input conditions (such as those it generates itself in the free-running mode) not seen during training and how to map the state back towards one that will make the network generate proper outputs after a few steps. Another approach (Bengio *et al.*, 2015) to mitigate the gap between the generative mode of RNNs and how they are trained (with teacher forcing, i.e., maximum likelihood) randomly chooses to use generated values or actual data values as input. This approach exploits a curriculum learning strategy to gradually use more of the generated values as input.

10.2.1 Computing the Gradient in a Recurrent Neural Network

Using the generalized back-propagation algorithm (for arbitrary flow graphs) introduced in Section 6.4.3, one can obtain the so-called **Back-Propagation Through Time** (BPTT) algorithm. Once we know how to compute gradients, we can in principle apply any of the general-purpose gradient-based techniques to train an RNN. These general-purpose techniques were introduced in Section 4.3 and developed in greater depth in Chapter 8.

Let us thus work out how to compute gradients by BPTT for the RNN equations above (Eqs. 10.4 and 10.5). The nodes of our flow graph will include the

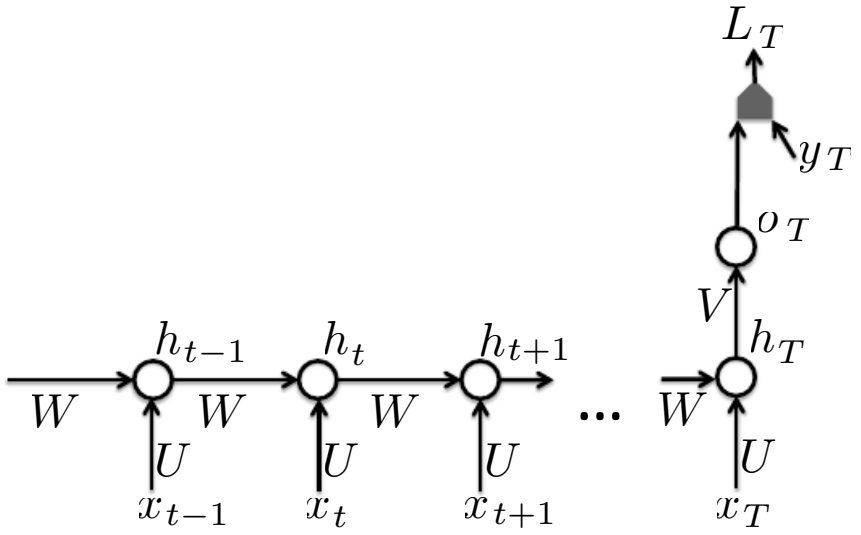

Figure 10.6: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (like in the figure) or the gradient on the output $\boldsymbol{o}_t$ can be obtained by back-propagating from further downstream modules.

parameters $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ as well as the sequence of nodes indexed by t for $\boldsymbol{x}_t$, $\boldsymbol{s}_t$, $\boldsymbol{o}_t$ and L_t. For each node a we need to compute the gradient $\nabla_{\boldsymbol{a}} L$ recursively, based on the gradient computed at nodes that follow it in the graph. We start the recursion with the nodes immediately preceding the final loss

$$\frac{\partial L}{\partial L_t} = 1$$

and the gradient $\nabla_{\boldsymbol{o}_t} L$ on the outputs at time step t, for all i, t, is as follows:

$$(\nabla_{\boldsymbol{o}_t} L)_i = \frac{\partial L}{\partial o_{ti}} = \frac{\partial L}{\partial L_t}\frac{\partial L_t}{\partial o_{ti}} = p_{t,i} - \mathbf{1}_{i,y_t}$$

and work our way backwards, starting from the end of the sequence, say T, at which point $\boldsymbol{s}_T$ only has $\boldsymbol{o}_T$ as descendent:

$$\nabla_{\boldsymbol{s}_T} L = \nabla_{\boldsymbol{o}_T} L \frac{\partial \boldsymbol{o}_T}{\partial \boldsymbol{s}_T} = \nabla_{\boldsymbol{o}_T} L \, \boldsymbol{V}.$$

Note how the above equation is vector-wise and corresponds to $\frac{\partial L}{\partial s_{Tj}} = \sum_i \frac{\partial L}{\partial o_{Ti}} V_{ij}$, scalar-wise. We can then iterate backwards in time to back-propagate gradients through time, from $t = T-1$ down to $t = 1$, noting that $\boldsymbol{s}_t$ (for $t < T$) has as descendents both $\boldsymbol{o}_t$ and $\boldsymbol{s}_{t+1}$:

$$\nabla_{\boldsymbol{s}_t} L = \nabla_{\boldsymbol{s}_{t+1}} L \frac{\partial \boldsymbol{s}_{t+1}}{\partial \boldsymbol{s}_t} + \nabla_{\boldsymbol{o}_t} L \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{s}_t} = \nabla_{\boldsymbol{s}_{t+1}} L \, \mathrm{diag}(1 - \boldsymbol{s}_{t+1}^2) \boldsymbol{W} + \nabla_{\boldsymbol{o}_t} L \, \boldsymbol{V}$$

where $\text{diag}(1 - \boldsymbol{s}_{t+1}^2)$ indicates the diagonal matrix containing the elements $1 - \boldsymbol{s}_{t+1,i}^2$, i.e., the derivative of the hyperbolic tangent associated with the hidden unit i at time $t+1$.

Once the gradients on the internal nodes of the flow graph are obtained, we can obtain the gradients on the parameter nodes, which have descendents at all the time steps:

$$\begin{aligned}
\nabla_{\boldsymbol{c}} L &= \sum_t \nabla_{\boldsymbol{o}} L \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{c}} = \sum_t \nabla_{\boldsymbol{o}_t} L \\
\nabla_{\boldsymbol{b}} L &= \sum_t \nabla_{\boldsymbol{s}_t} L \frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{b}} = \sum_t \nabla_{\boldsymbol{s}_t} L \, \text{diag}(1 - \boldsymbol{s}_t^2) \\
\nabla_{\boldsymbol{V}} L &= \sum_t \nabla_{\boldsymbol{o}} L \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{V}} = \sum_t \nabla_{\boldsymbol{o}_t} L \, \boldsymbol{s}_t^\top \\
\nabla_{\boldsymbol{W}} L &= \sum_t \nabla_{\boldsymbol{s}_t} \frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{W}} = \sum_t \nabla_{\boldsymbol{s}_t} L \, \text{diag}(1 - \boldsymbol{s}_t^2) s_{t-1}^\top
\end{aligned}$$

Note in the above (and elsewhere) that whereas $\nabla_{\boldsymbol{s}_t} L$ refers to the full influence of $\boldsymbol{s}_t$ through all paths from $\boldsymbol{s}_t$ to L, $\frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{W}}$ or $\frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{b}}$ refers to the immediate effect of the denominator on the numerator, i.e., when we consider the denominator as a parent of the numerator and only that direct dependency is accounted for. Otherwise, we would get "double counting" of derivatives.

10.2.2 Recurrent Networks as Directed Graphical Models

Up to here, we have not clearly stated what the losses L_t associated with the outputs o_t of a recurrent net should be. We have delayed this discussion because there are many possible ways to use RNNs. In this section, we describe the most common case, where the RNN models a probability distribution over a sequence of observations.

When we use a predictive log-likelihood training objective, such as Eq. 10.5, we train the RNN to estimate the conditional distribution of the next sequence element y_t given the past inputs. The conditioning input sequence $\{\boldsymbol{x}_\tau\}_{\tau<t}$ may contain the past values of y: we condition on past values of y to predict future values of y. This is a natural consequence of a probabilistic interpretation of modeling the sequence of y's, explained below. Sometimes, we include in the conditioning information not just the past y's but also other conditioning variables. Decomposing the joint probability over the sequence of y's as a series of one-step probabilistic predictions is one way to capture the full joint distribution across the whole sequence. When we do not feed past y's as inputs that condition the next step prediction, the directed graphical model contains no edges from any y_τ

in the past to the current y_t. In this case, the outputs y are conditionally independent given the sequence of $\boldsymbol{x}$'s. When we do feed the actual y values (not their prediction, but the actual observed or generated values) back into the network, the directed graphical model contains edges from all y_τ values in the past to the current y_t value.

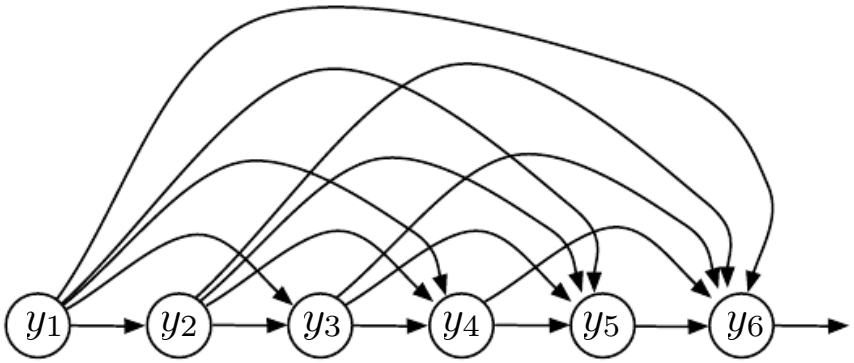

Figure 10.7: Fully connected graphical model for a sequence $y_1, y_2, \ldots, y_t, \ldots$: every past observation y_τ may influence the conditional distribution of some y_t (for $t > \tau$), given the previous values. Parametrizing the graphical model directly according to this graph (as in Eq. 10.3) might be very inefficient, with an ever growing number of inputs and parameters for each element of the sequence. RNNs obtain the same full connectivity but efficient parametrization, as illustrated in Fig. 10.8.

As a simple example, let us first consider the case where $\boldsymbol{x}_\tau$ only contains y_τ: the target output at the next time is also the next input. The RNN then defines a directed graphical model over the random variables $\mathbf{y} = (\mathrm{y}_1, \mathrm{y}_2, \ldots, \mathrm{y}_T)$. We parametrize the joint distribution of these observations using the chain rule (Eq. 3.1) for conditional probabilities:

$$P(\mathbf{y}) = P(\mathrm{y}_1, \ldots, \mathrm{y}_T) = \prod_{t=1}^{T} P(\mathrm{y}_t \mid \mathrm{y}_{t-1}, \mathrm{y}_{t-2}, \ldots, \mathrm{y}_1) \tag{10.6}$$

where the right-hand side of the bar is empty for $t = 1$, of course. Hence the negative log-likelihood of $\boldsymbol{y}$ according to such a model is

$$L = \sum_t L_t$$

where

$$L_t = -\log P(\mathrm{y}_t = y_t \mid y_{t-1}, y_{t-2}, \ldots, y_1).$$

Manipulating the structure (presence of arcs between nodes) of graphical models allow us to describe probability distributions in which some variables are conditionally independent. For example, a subset of $(\mathrm{y}_1, \ldots, \mathrm{y}_{t-1})$ might provide all

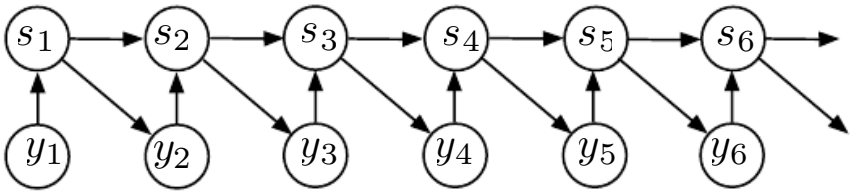

Figure 10.8: Introducing the state variable in the graphical model of the RNN, even though it is a deterministic function of its inputs, helps to see how we can obtain a very efficient parametrization, based on Eq. 10.2. Every stage in the sequence (for $\boldsymbol{s}_t$ and y_t) involves the same structure (the same number of inputs for each node) and can share the same parameters with the other stages.

the information one can get to predict y_t given $(\mathrm{y}_1, \ldots, \mathrm{y}_{t-1})$. However, in some cases, we believe that all past inputs should have an influence on the next element of the sequence, and not say, the most recent ones. In that case the graphical model would be fully connected, as in Figure 10.7. RNNs as directed graphical models have such a fully connected structure, when we ignore (i.e., marginalize over) the intermediate states. However, it is more interesting to consider the graphical model structure of RNNs when we introduce a particular kind of latent variable $\boldsymbol{s}_t$ (that is deterministic in the past observations). This results in a very efficient parametrization of the joint distribution over the observations, because rather than having the number of parameters grow exponentially with time – as would be the case for an arbitrary joint distribution – due to parameter sharing the number of parameters in the RNN in actually constant w.r.t. time. This is illustrated by Eq. 10.2 and Figure 10.8. The model is trained so that the variable $\boldsymbol{s}_t$ summarizes whatever is required from the whole previous sequence (Eq. 10.3) to predict the next step. Some operations such as predicting missing values in the middle of the sequence remain computationally inefficient despite the recurrent network parameterization. If we incorporate the $\boldsymbol{s}_t$ nodes in the graphical model, we see that it decouples that past and the future, acting as an intermediate quantity between them. It also makes the graph locally connected (each state $\boldsymbol{s}_t$ only depends the previous $\boldsymbol{s}_{t-1}$ and on the current input) whereas the original graphical model (without nodes for the state) is fully connected.

The statistical complexity of the model may then be flexibly adjusted by changing the complexity of the function producing the state variable. Crucially, the complexity of this function (as measured by the number of free parameters) is not forced to increase with sequence length. This stands in contrast to the approach of controlling statistical complexity by modifying the graph structure of a directed graphical model. A graphical model defined over discrete variables using a table to implement the conditional probability distribution $P(\mathrm{y}_t \mid \mathrm{y}_{t-1}, \mathrm{y}_{t-2}, \ldots, \mathrm{y}_1)$ would require an amount of parameters (table entries)

that grows exponentially with t. The graphical model of recurrent networks is directed and decomposes the probability distribution as a product of conditionals without explicitly cutting any arc in the graphical model. The recurrent net uses the same graphical model structure, but defines this conditional probability distribution via Eq. 10.2, using a number of parameters that is constant in t. The capacity of the model is reduced by parametrizing the transition probability in a recursive way that requires a fixed (and not exponential) number of parameters, due to a form of parameter sharing (see Section 7.8 for an introduction to the concept). Instead of reducing $P(\mathrm{y}_t \mid \mathrm{y}_{t-1}, \ldots, \mathrm{y}_1)$ to something like $P(\mathrm{y}_t \mid \mathrm{y}_{t-1}, \ldots, \mathrm{y}_{t-k})$ (assuming the k previous ones as the parents), we keep the full dependency but we parametrize the conditional efficiently in a way that does not grow with t, exploiting parameter sharing. The price recurrent networks pay for their reduced number of parameters is that *optimizing* the parameters may be difficult, as discussed below (Sections 8.2.6 and 10.7).

The parameter sharing used in recurrent networks relies on the assumption that the same parameters can be used for different time steps: the above conditional probability distribution is *stationary*, i.e., the relation between the past and the next observation does not depend on t, only on the values of the past observations. It allows one to use the same model for sequences of different lengths. In principle, it would be possible to use t as an extra input at each time step and let the learner discover any time-dependence while sharing as much as it can between different time steps. This would already be much better than using a different conditional probability distribution for each t, but the network would then have to extrapolate when faced with new values of t.

Let us consider how that stationarity assumption can be turned into an efficient recursive parametrization. When we introduce a function g_t that summarizes the past inputs and a recursive function f_θ to implement it, the decomposition of the likelihood over a sequence of vectors $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)$ is

$$P(\boldsymbol{X}) = \prod_{t=1}^{T} P(\boldsymbol{x}_t \mid g_t(\boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_1))$$

where

$$\boldsymbol{s}_t = g_t(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_2, \boldsymbol{x}_1) = f_\theta(\boldsymbol{s}_{t-1}, \boldsymbol{x}_t).$$

Note that if the self-recurrence function f_θ is *learned*, it can discard some of the information in some of the past values $\boldsymbol{x}_{t-k}$ that are not needed for predicting the future data. In fact, because the state generally has a fixed dimension smaller than the length of the sequences (times the dimension of the input), it *has to* discard some information. However, we leave it to the learning procedure to choose what information to keep and what information to throw away, so as minimize the objective function (e.g., predict future values correctly).

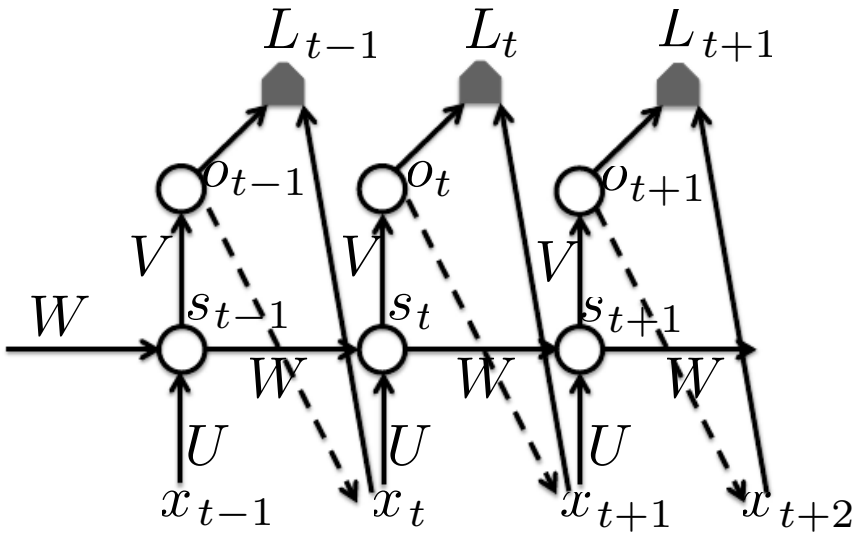

Figure 10.9: A recurrent neural network modeling $P(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$, able to generate sequences from this distribution. Each element $\boldsymbol{x}_t$ of the observed sequence serves both as input (for computing the state $\boldsymbol{s}_t$ at the current time step) and as target (for the prediction made at the previous time step). The output $\boldsymbol{o}_t$ encodes the parameters of a conditional distribution $P(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t) = P(\boldsymbol{x}_{t+1} \mid \boldsymbol{o}_t)$ for $\boldsymbol{x}_{t+1}$, given the past sequence $\boldsymbol{x}_1 \ldots, \boldsymbol{x}_t$. The loss L_t is the negative log-likelihood associated with the output prediction (or more generally, distribution parameters) $\boldsymbol{o}_t$, when the actual observed target is $\boldsymbol{x}_{t+1}$. In training mode, one measures and minimizes the sum of the losses over observed sequence(s) $\boldsymbol{x}$. In generative mode, $\boldsymbol{x}_t$ is sampled from the conditional distribution $P(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t) = P(\boldsymbol{x}_{t+1} \mid \boldsymbol{o}_t)$ (dashed arrows) and then that generated sample $\boldsymbol{x}_{t+1}$ is fed back as input for computing the next state $\boldsymbol{s}_{t+1}$, the next output $\boldsymbol{o}_{t+1}$, and generating the next sample $\boldsymbol{x}_{t+2}$, etc. The length of the sequence must also be generated (unless known in advance). This could be done by a special sigmoidal output unit that predicts a binary target (with associated cross-entropy loss) that encodes the fact that the next output is the last.

The above decompositions of the joint probability of a sequence of variables into ordered conditionals precisely corresponds to the sequence of computations performed by an RNN. The *target* to be predicted at each time step t is the next element in the sequence, while the *input* at each time step is the previous element in the sequence (with all previous inputs summarized in the state), and the *output* is interpreted as parametrizing the probability distribution of the target given the state. This is illustrated in Fig. 10.9.

If the RNN is actually going to be used to generate sequences, one must also incorporate in the output information allowing to stochastically decide when to stop generating new output elements. This can be achieved in various ways. In the case when the output is a symbol taken from a vocabulary, one can add a special symbol corresponding to the end of a sequence. When that symbol is generated, a complete sequence has been generated. The target for that special symbol occurs exactly once per sequence, as the last target after the last output element $\boldsymbol{x}_T$. In general, one may train a binomial output associated with that stopping variable, for example using a sigmoid output non-linearity and the cross entropy loss, i.e., again negative log-likelihood for the event "end of the sequence". Another kind of solution is to model the integer T itself, through any reasonable parametric distribution, and use the number of time steps left (and possibly the number of time steps since the beginning of the sequence) as extra inputs at each time step. Thus we would have decomposed $P(\boldsymbol{x}_1 \ldots, \boldsymbol{x}_T)$ into $P(T)$ and $P(\boldsymbol{x}_1 \ldots, \boldsymbol{x}_T \mid T)$. In general, one must therefore keep in mind that in order to fully generate a sequence we must not only generate the $\boldsymbol{x}_t$'s, but also the sequence length T, either implicitly through a series of continue/stop decisions (or a special "end-of-sequence" symbol), or explicitly through modeling the distribution of T itself as an integer random variable. For example, the strategy of using a special end-of-sequence symbol is used when the RNN generates an output sequence, such as a sequence of words in machine translation (Kalchbrenner and Blunsom, 2013; Cho *et al.*, 2014a; Sutskever *et al.*, 2014b; Jean *et al.*, 2014). The strategy of predicting T itself is used for example by Goodfellow *et al.* (2014d).

If we take the RNN equations of the previous section (Eq. 10.4 and 10.5), they could correspond to a generative RNN if we simply make the target $\boldsymbol{y}_t$ equal to the next input $\boldsymbol{x}_{t+1}$ (and because the outputs are the result of a softmax, it must be that the input sequence is a sequence of symbols, i.e., $\boldsymbol{x}_t$ is a symbol or bounded integer).

Other types of data can clearly be modeled in a similar way, following the discussions about the encoding of outputs and the probabilistic interpretation of losses as negative log-likelihoods, in Sections 5.6 and 6.3.2.

10.2.3 Modeling a sequence conditioned on some context by making the generative RNN conditional

In the previous section we studied how an RNN could correspond to a directed graphical model over a sequence of random variables y_t. Here we consider how one can condition this distribution on other variables. In general, as discussed in Section 6.3.2 (see especially the end of that section, in Subsection 6.3.2), when we can represent a parametric probability distribution $P(\mathbf{y} \mid \boldsymbol{\omega})$, we can make it conditional by making ω a function of the appropriate conditioning variable:

$$P(\mathbf{y} \mid \boldsymbol{\omega} = f(\mathbf{x})).$$

In the case of an RNN, this can be achieved in different ways. We review here the most common and obvious choices.

If $\mathbf{x}$ is a fixed-size vector, then we can simply make it an extra input of the RNN that generates the $\mathbf{y}$ sequence. Some common ways of providing an extra input to an RNN are:

1. as an extra input at each time step, or
2. as the initial state $\boldsymbol{s}_0$, or
3. both.

In general, one may need to add extra parameters (and parametrization) to map $\mathbf{x} = \boldsymbol{x}$ into the "extra bias" going either into only $\boldsymbol{s}_0$, into the other $\boldsymbol{s}_t$ $(t > 0)$, or into both. The first (and most commonly used) approach is illustrated in Fig. 10.10.

Speech is considered to be a sequence of phonemes or phonetic categories, roughly corresponding to one letter in latin alphabets. As an example, imagine that $\mathbf{x}$ is encoding the identity of a phoneme and the identity of a speaker. Let$\mathbf{y}$ represent an acoustic sequence corresponding to that phoneme, as pronounced by that speaker.

Consider the case where the input $\mathbf{x}$ is a sequence of the same length as the output sequence $\mathbf{y}$, and the $\mathbf{y}_t$'s are independent of each other when the past input sequence is given, i.e., $P(\mathbf{y}_t \mid \mathbf{y}_{t-1}, \ldots, \mathbf{y}_1, \mathbf{x}) = P(\mathbf{y}_t \mid \mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1)$. We therefore have a causal relationship between the $\mathbf{x}_t$'s and the predictions of the $\mathbf{y}_t$'s, in addition to the independence of the $\mathbf{y}_t$'s, given $\mathbf{x}$. Under these (pretty strong) assumptions, we can return to Fig. 10.3 and interpret the t-th output $\boldsymbol{o}_t$ as parameters for a conditional distribution for $\mathbf{y}_t$, given $\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1$.

If we want to remove the conditional independence assumption, we can do so by making the past $\mathbf{y}_t$'s inputs into the state as well. That situation is illustrated in Fig. 10.11.

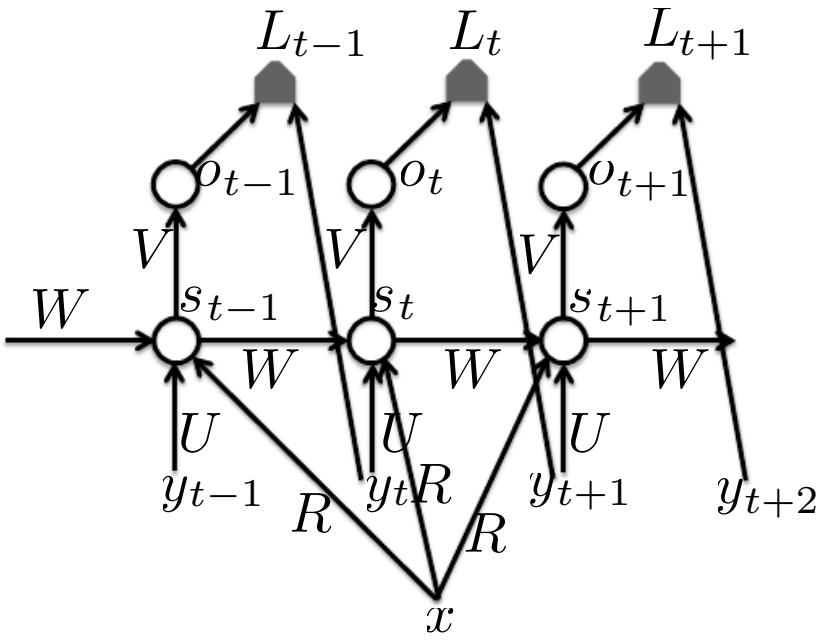

Figure 10.10: A conditional generative recurrent neural network maps a fixed-length vector $\boldsymbol{x}$ into a distribution over sequences $\mathbf{Y}$. Each element $\boldsymbol{y}_t$ of the observed output sequence serves both as input (for the current time step) and, during training, as target (for the previous time step). The generative semantics are the same as in the unconditional case (Fig. 10.9). There are only two differences. First, the state is now conditioned on the input $\boldsymbol{x}$. Second, the same parameters (weight matrix $\boldsymbol{R}$ in the figure) are used at every time step to parametrize that dependency.

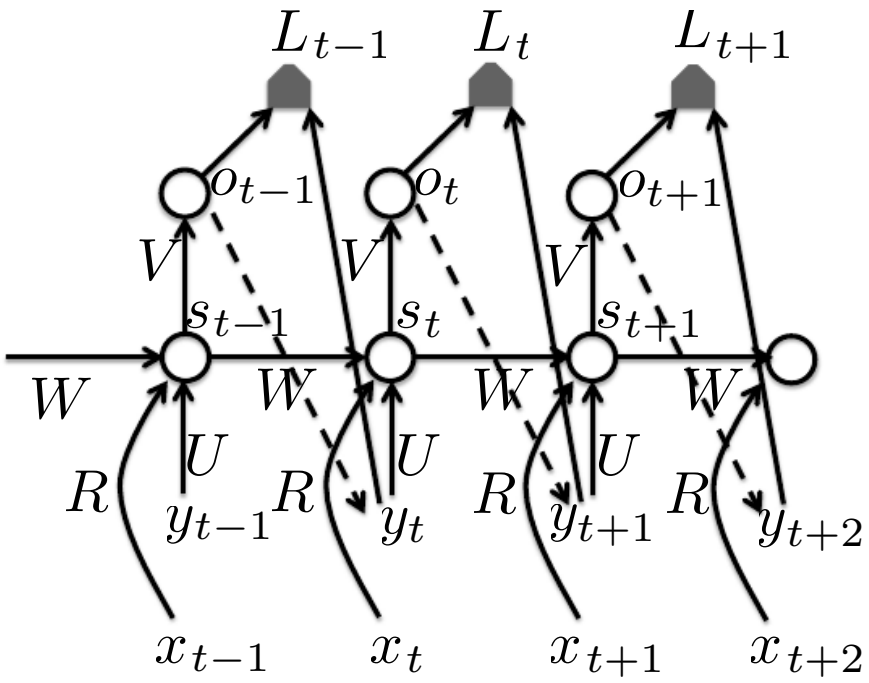

Figure 10.11: A conditional generative recurrent neural network mapping a variable-length sequence $\boldsymbol{x}$ into a distribution over sequences $\mathbf{y}$ of the same length. This architecture assumes that the predictions of $\mathbf{y}_t$'s are causally related to the $\mathbf{x}_t$'s, i.e., that we want to predict the $\mathbf{y}_t$'s only using the past $\mathbf{x}_t$'s. Note how the prediction of $\mathbf{y}_{t+1}$ is based on both the past $\mathbf{x}$'s and the past $\mathbf{y}$'s. The dashed arrows indicate that $\boldsymbol{y}_t$ can be generated by sampling from the output distribution $\boldsymbol{o}_{t-1}$. When $\boldsymbol{y}_t$ is clamped (known), it is used as a target in the loss L_{t-1} which measures the log-probability that $\boldsymbol{y}_t$ would be sampled from the distribution o_{t-1}.

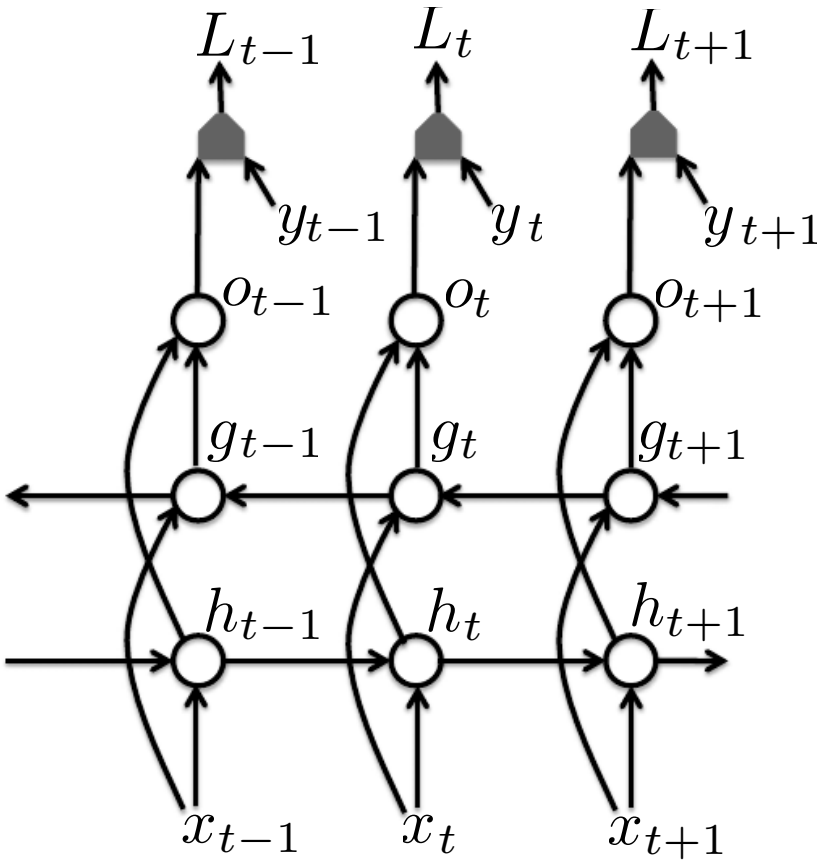

Figure 10.12: Computation of a typical bidirectional recurrent neural network, meant to learn to map input sequences $\boldsymbol{x}$ to target sequences $\boldsymbol{y}$, with loss L_t at each step t. The $\boldsymbol{h}$ recurrence propagates information forward in time (towards the right) while the $\boldsymbol{g}$ recurrence propagates information backward in time (towards the left). Thus at each point t, the output units $\boldsymbol{o}_t$ can benefit from a relevant summary of the past in its $\boldsymbol{h}_t$ input and from a relevant summary of the future in its $\boldsymbol{g}_t$ input.

10.3 Bidirectional RNNs

All of the recurrent networks we have considered up to now have a "causal" structure, meaning that the state at time t only captures information from the past, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$. However, in many applications we want to output at time t a prediction regarding an output which may depend on *the whole input sequence*. For example, in speech recognition, the correct interpretation of the current sound as a phoneme may depend on the next few phonemes because of co-articulation and potentially may even depend on the next few words because of the linguistic dependencies between nearby words: if there are two interpretations of the current word that are both acoustically plausible, we may have to look far into the future (and the past) to disambiguate them. This is also true of handwriting recognition and many other sequence-to-sequence learning tasks, described in the next section.

Bidirectional recurrent neural networks (or bidirectional RNNs) were invented to address that need (Schuster and Paliwal, 1997). They have been extremely successful (Graves, 2012) in applications where that need arises, such as handwriting recognition (Graves *et al.*, 2008; Graves and Schmidhuber, 2009), speech recognition (Graves and Schmidhuber, 2005; Graves *et al.*, 2013) and bioinformatics (Baldi *et al.*, 1999).

As the name suggests, the basic idea behind bidirectional RNNs is to combine a forward-going RNN and a backward-going RNN. Fig. 10.12 illustrates the typical bidirectional RNN, with $\boldsymbol{h}_t$ standing for the state of the forward-going RNN and $\boldsymbol{g}_t$ standing for the state of the backward-going RNN. This allows the units $\boldsymbol{o}_t$ to compute a representation that depends on *both the past and the future* but is most sensitive to the input values around time t, without having to specify a fixed-size window around t (as one would have to do with a feedforward network, a convolutional network, or a regular RNN with a fixed-size look-ahead buffer).

This idea can be naturally extended to 2-dimensional input, such as images, by having *four* RNNs, each one going in one of the four directions: up, down, left, right. At each point (i, j) of a 2-D grid, an output $o_{i,j}$ could then compute a representation that would capture mostly local information but could also depend on long-range inputs, if the RNN are able to learn to carry that information.

10.4 Encoder-Decoder Sequence-to-Sequence Architectures

We have seen in Fig. 10.6 how an RNN can map an input sequence to a fixed-size prediction. We have seen in Fig. 10.9 how an RNN can model a distribution over sequences and generate new ones from the estimated distribution. We have seen in Fig. 10.10 how one can condition on an input vector to learn to generate such

sequences. We have seen in Figures 10.11 and 10.12 how an RNN (unidirectional or bidirectional) can map an input sequence to an output sequence of the same length.

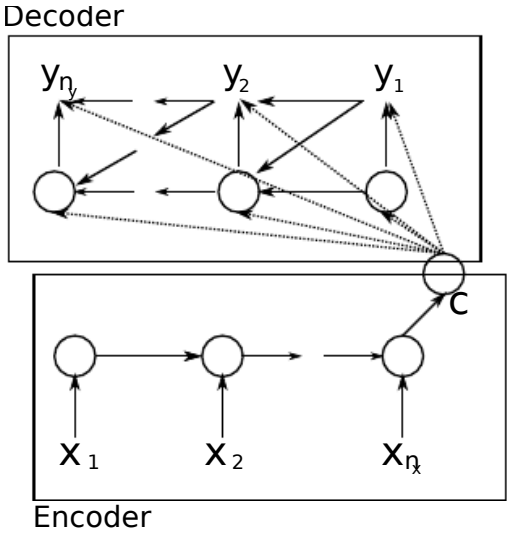

Figure 10.13: Example of encoder-decoder or sequence-to-sequence RNN architecture, for learning to generate an output sequence $(\mathbf{y}_1, \ldots, \mathbf{y}_{n_y})$ given an input sequence $(\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_{n_x})$. It is composed of an encoder RNN that reads the input sequence and a decoder RNN that generates the output sequence (or computes the probability of a given output sequence). The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable C which represents a semantic summary of the input sequence and conditions computations in the decoder RNN.

Here we discuss how an RNN can be trained to map an input sequence to an output sequence which is not necessarily of the same length. This comes up in many applications, such as speech recognition, machine translation or question answering, where the input and output sequences in the training set are generally not of the same length (although their lengths might be related).

We often call the input to the RNN the "context." We want to produce a representation of this context, C. The context C might be a vector or a sequence or sequence of vectors that summarize the input sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{nx})$.

The simplest RNN architecture for mapping a variable-length sequence to another variable-length sequence was first proposed in Cho *et al.* (2014a) and shortly after in Sutskever *et al.* (2014b). These authors respectively called this architecture, illustrated in Fig. 10.13, the encoder-decoder or sequence-to-sequence architecture. The idea is very simple: (1) an *encoder* or *reader* or *input* RNN processes the input sequence. The encoder emits the context C, usually as a simple function of its final hidden state. (2) a *decoder* or *writer* or *output* RNN is conditioned on that fixed-length vector (just like in Fig. 10.10) to generate the output sequence $\boldsymbol{Y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{ny})$, where the lengths n_x and n_y can vary from

training pair to training pair. The two RNNs are trained jointly to maximize the average of $\log P(\mathbf{Y} = \boldsymbol{Y}|\mathbf{X} = \boldsymbol{X})$ over all the training pairs $(\boldsymbol{X}, \boldsymbol{Y})$. The last state $\boldsymbol{s}_{nx}$ of the input RNN is typically used as a representation C of the input sequence that conditions the output RNN. The output RNN can be conditioned in at least two ways, which can be combined, as we have seen earlier, i.e., either by producing a starting state for the output RNN or by producing an extra input at each time step of the output RNN. In any case, one inserts an extra set of parameters to map C into a bias or initial state. Hence the two RNNs do not have to have the same hidden layer dimensionality, and sometimes it makes sense to make that mapping more complex and non-linear, e.g., an MLP could be used to map the output of the encoder RNN into an input for the decoder RNN.

One clear limitation of this architecture is when the output of the encoder RNN has a dimension that is too small to properly summarize a long sequence. This phenomenon was observed by Bahdanau *et al.* (2014) in the context of machine translation. They proposed to make C a variable length sequence rather than a fixed-size vector. Additionally, they introduced an *attention mechanism* that learns to associate elements of the sequence C to elements of the output sequence. See Sec. 12.4.6 for more details.

10.5 Deep Recurrent Networks

The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:

1. from input to hidden state,
2. from previous hidden state to next hidden state, and
3. from hidden state to output,

where the first two are actually brought together to map the input and previous state into the next state. With the RNN architecture of Fig. 10.3, each of these three blocks is associated with a single weight matrix. In other words, when the network is unfolded, each of these corresponds to a shallow transformation. By a shallow transformation, we mean a transformation that would be represented by a single layer within a deep MLP. Typically this is a transformation represented by a learned affined transformation followed by a fixed non-linearity.

Would it be advantageous to introduce depth in each of these operations? Experimental evidence (Graves *et al.*, 2013; Pascanu *et al.*, 2014a) strongly suggests so. The experimental evidence is in agreement with the idea that we need enough depth in order to perform the required mappings. See also Schmidhuber (1992); El Hihi and Bengio (1996); Jaeger (2007a) for earlier work on deep RNNs.

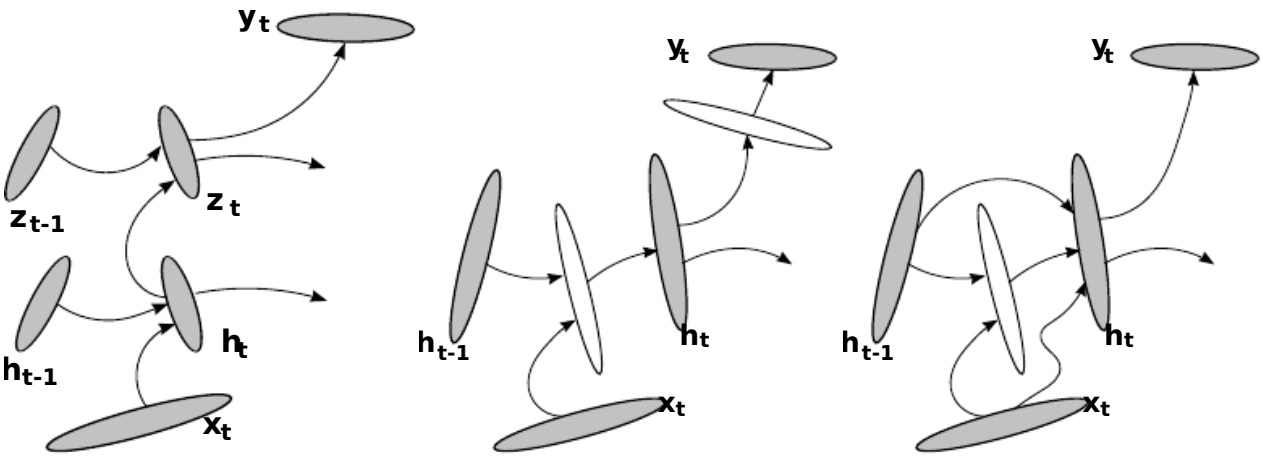

Figure 10.14: A recurrent neural network can be made deep in many ways. First, the hidden recurrent state can be broken down into groups organized hierarchically (left). Second, deeper computation (e.g., an MLP in the figure) can be introduced in the input-to-hidden, hidden-to-hidden and hidden-to-output parts (Middle). This may lengthen the shortest path linking different time steps. The path-lengthening effect can be mitigated by introduced skip connections (Right). Figures reproduced from Pascanu *et al.* (2014a) with permission.

El Hihi and Bengio (1996) first introduced the idea of decomposing the hidden state of an RNN into multiple groups of units that would operate at different time scales. Graves *et al.* (2013) were the first to show a significant benefit of decomposing the state of an RNN into groups of hidden units, with a restricted connectivity between the groups, e.g., as in Fig. 10.14 (left). Indeed, if there were no restriction at all and no pressure for some units to represent a slower time scale, then having N groups of M hidden units would be equivalent to having a single group of NM hidden units. Koutnik *et al.* (2014) showed how the multiple time scales idea from El Hihi and Bengio (1996) can be advantageous on several sequential learning tasks: each group of hidden unit is updated at a different multiple of the time step index e.g., at every time step, at every 2nd step, at every 4th step, etc.

We can also think of the lower layers in this hierarchy as playing a role in transforming the raw input into a representation that is more appropriate, at the higher levels of the hidden state. Pascanu *et al.* (2014a) go a step further and propose to have a separate MLP (possibly deep) for each of the three blocks enumerated above, as illustrated in Fig. 10.14 (middle). It makes sense to allocate enough capacity in each of these three steps, but having a deep state-to-state transition may also hurt: it makes the shortest path from an event at time t to an event at time $t' > t$ substantially longer, which make it more difficult to learn long-term dependencies (see Sections 8.2.6 and 10.7). For example if a one-hidden-layer MLP is used for the state-to-state transition, we have doubled the

length of that path, compared with the ordinary RNN of Fig. 10.3. However, as argued by Pascanu *et al.* (2014a), this can be mitigated by introducing skip connections in the hidden-to-hidden path, as illustrated in Fig. 10.14 (right).

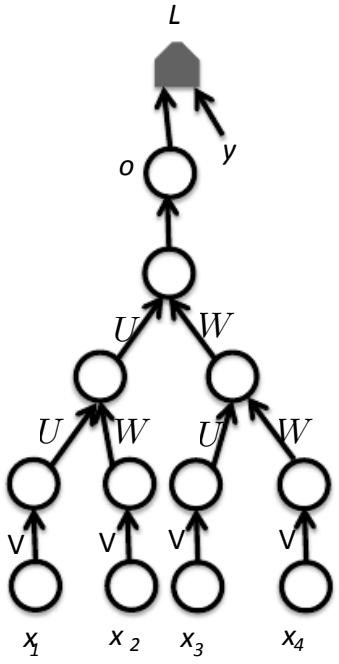

Figure 10.15: A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree. In the figure, a variable-size sequence $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$ can be mapped to a fixed-size representation (the output o), with a fixed number of parameters (e.g. the weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$). The figure illustrates a supervised learning case in which some target y is provided which is associated with the whole sequence.

10.6 Recursive Neural Networks

Recursive networks represent yet another generalization of recurrent networks, with a different kind of computational graph, which is structured as a deep tree, rather than the chain-like structure of RNNs. The typical computational graph for a recursive network is illustrated in Fig. 10.15. Recursive neural networks were introduced by Pollack (1990) and their potential use for learning to reason were nicely laid down by Bottou (2011). Recursive networks have been successfully applied in processing *data structures* as input to neural nets (Frasconi *et al.*, 1997, 1998), in natural language processing (Socher *et al.*, 2011a,c, 2013) as well as in computer vision (Socher *et al.*, 2011b).

One clear advantage of recursive nets over recurrent nets is that for a sequence of the same length N, the depth (measured as the number of compositions of non-

linear operations) can be drastically reduced from N to $O(\log N)$, which might help deal with long-term dependencies. An open question is how to best structure the tree, though. One option is to have a tree structure which does not depend on the data, e.g., a balanced binary tree. Another is to use an external method, such as a natural language parser (Socher *et al.*, 2011a, 2013). Ideally, one would like the learner itself to discover and infer the tree structure that is appropriate for any given input, as suggested in Bottou (2011).

Many variants of the recursive net idea are possible. For example, in Frasconi *et al.* (1997, 1998), the data is associated with a tree structure in the first place, and inputs and/or targets with each node of the tree. The computation performed by each node does not have to be the traditional artificial neuron computation (affine transformation of all inputs followed by a monotone non-linearity). For example, Socher *et al.* (2013) propose using tensor operations and bilinear forms, which have previously been found useful to model relationships between concepts (Weston *et al.*, 2010; Bordes *et al.*, 2012) when the concepts are represented by continuous vectors (embeddings).

10.7 The Challenge of Long-Term Dependencies

The mathematical challenge of learning long-term dependencies in recurrent networks was introduced in Section 8.2.6. The basic problem is that gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization). Even if we assume that the parameters are such that the recurrent network is stable (can store memories, with gradients not exploding), the difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions (involving the multiplication of many Jacobians) compared short-term ones. See Hochreiter (1991); Doya (1993); Bengio *et al.* (1994); Pascanu *et al.* (2013a) for a deeper treatment.

In this section we discuss various approaches that have been proposed to alleviate this difficulty with learning long-term dependencies.

10.7.1 Echo State Networks

Echo state networks are models whose weights are chosen to make the dynamics of forward propagation barely contractive.

The recurrent weights and input weights of a recurrent network are those that define the state representation captured by the model, i.e., how the state $\boldsymbol{s}_t$ (hidden units vector) at time t (Eq. 10.2) captures and summarizes information from the previous inputs $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_t$. Since learning the recurrent and input weights is difficult, one option that has been proposed (Jaeger, 2003; Lee *et al.*,

2015; Maass *et al.*, 2002; Jaeger and Haas, 2004; Jaeger, 2007b) is to *set those weights such that the recurrent hidden units do a good job of capturing the history of past inputs*, and *only learn the output weights*. This is the idea that was independently proposed for *Echo State Networks* or ESNs (Jaeger and Haas, 2004; Jaeger, 2007b) and *Liquid State Machines* (Maass *et al.*, 2002). The latter is similar, except that it uses spiking neurons (with binary outputs) instead of the continuous-valued hidden units used for ESNs. Both ESNs and liquid state machines are termed *reservoir computing* (Lukoševičius and Jaeger, 2009) to denote the fact that the hidden units form of reservoir of temporal features which may capture different aspects of the history of inputs.

One way to think about these reservoir computing recurrent networks is that they are similar to kernel machines: they map an arbitrary length sequence (the history of inputs up to time t) into a fixed-length vector (the recurrent state $\boldsymbol{s}_t$), on which a linear predictor (typically a linear regression) can be applied to solve the problem of interest. The training criterion is therefore convex in the parameters (which are just the output weights) and can actually be solved online in the linear regression case,using online updates for linear regression (Jaeger, 2003).

The important question is therefore: how do we set the input and recurrent weights so that a rich set of histories can be represented in the recurrent neural network state? The answer proposed in the reservoir computing literature is to make the dynamical system associated with the recurrent net nearly be on the edge of stability, i.e., more precisely with values around 1 for the leading singular value of the Jacobian of the state-to-state transition function. As explained in 8.2.6, an important characteristic of a recurrent network is the eigenvalue spectrum of the Jacobians $\boldsymbol{J}^{(t)} = \frac{\partial s_t}{\partial s_{t-1}}$. Of particular importance is the *spectral radius* of $\boldsymbol{J}^{(t)}$, defined to be the maximum of the absolute values of its eigenvalues (which can be complex-valued).

To understand the effect of the spectral radius, consider the simple case of back-propagation with a the Jacobian matrix $\boldsymbol{J}$ that does not change with t. This case happens, for example, when the network is purely linear. Suppose that $\boldsymbol{J}$ has an eigenvector $\boldsymbol{v}$ with corresponding eigenvalue λ. Consider what happens as we propagate a gradient vector backwards through time. If we begin with a gradient vector $\boldsymbol{g}$, then after one step of back-propagation, we will have $\boldsymbol{J}\boldsymbol{g}$, and after n steps we will have $\boldsymbol{J}^n\boldsymbol{g}$. Now consider what happens if we instead back-propagate a perturbed version of $\boldsymbol{g}$. If we begin with $\boldsymbol{g}+\delta\boldsymbol{v}$, then after one step, we will have $\boldsymbol{J}(\boldsymbol{g}+\delta\boldsymbol{v})$. After n steps, we will have $\boldsymbol{J}^n(\boldsymbol{g}+\delta\boldsymbol{v})$. From this we can see that back-propagation starting from $\boldsymbol{g}$ and back-propagation starting from $\boldsymbol{g}+\delta\boldsymbol{v}$ diverge by $\delta\boldsymbol{J}^n\boldsymbol{v}$ after n steps of back-propagation. If $\boldsymbol{v}$ is chosen to be a unit eigenvector of $\boldsymbol{J}$ with eigenvalue λ, then multiplication by the Jacobian simply

scales the difference at each step. The two executions of back-propagation are separated by a distance of $\delta|\lambda|^n$. When $\boldsymbol{v}$ corresponds to the largest value of $|\lambda|$, this perturbation achieves the widest possible separation of an initial perturbation of size δ.

When $|lambda| > 1$, the deviation size $\delta|\lambda|^n$ grows exponentially large. When $|\lambda| < 1$, the deviation size becomes exponentially small.

Of course, this example assumed that the Jacobian was the same at every time step, corresponding to a recurrent network with no non-linearity. Everything we have said about back-propagation in such a setting applies equally to forward propagation in a network with no non-linearity, where the state $\boldsymbol{s}^{(t+1)} = \boldsymbol{s}^{(t)\top}\boldsymbol{W}$.

When the spectral radius is less than one, we say that the mapping from $\boldsymbol{s}^{(t)}$ to $\boldsymbol{s}^{(t+1)}$ is *contractive*—a small changes gets *contracted*, becoming smaller after each time step. This necessarily makes the network forget information about the past when we use a finite level of precision (such as 32 bit integers) to store the state vector.

The Jacobian matrix tells us how a small change of $\boldsymbol{s}_t$ propagates one step forward, or equivalently, how the gradient on $\boldsymbol{s}_{t+1}$ propagates one step backward, during back-propagation. Note that neither $\boldsymbol{W}$ nor $\boldsymbol{J}$ need to be symmetric (although they are square and real), so they can have complex-valued eigenvalues and eigenvectors, with imaginary components corresponding to potentially oscillatory behavior (if the same Jacobian was applied iteratively). Even though $\boldsymbol{s}^{(t)}$ or a small variation of $\boldsymbol{s}^{(t)}$ of interest in back-propagation are real-valued, they can be expressed in such a complex-valued basis. What matters is what happens to the magnitude (complex absolute value) of these possibly complex-valued basis coefficients, when we multiply by the matrix by the vector. An eigenvalue with magnitude greater than one corresponds to magnification (exponential growth, if applied iteratevely) or shrinking (exponential decay, if applied iteratively).

With a non-linear map, the Jacobian is free to change at each step. The dynamics therefore become more complicated. However, it remains true that a small initial variation can turn into a large variation after a number of steps. One difference between the purely linear case and the non-linear case is that the use of a squashing non-linearity such as tanh can cause the recurrent dynamics to become bounded. Note that it is possible for back-propagation to retain unbounded dynamics even when forward propagation has bounded dynamics, for example, when a sequence of tanh units are all in the middle of their linear regime and are connected by weight matrices with spectral radius greater than 1.

The strategy proposed for reservoir computing machines (Jaeger, 2003). is to set the weights to make the Jacobians always slightly contractive. This is achieved by making the spectral radius of the weight matrix large but slightly less than 1: the maximum spectral radius of $\boldsymbol{J}$ is the spectral radius of $\boldsymbol{W}$ times the maximum

of the absolute value of the derivative of the non-linearity; when the latter is 1, such as with `tanh`, it is thus recommended to set the spectral radius of $\boldsymbol{W}$ sligthly below 1.

However, in practice, good results are often found with a spectral radius of slightly larger than 1, such as 1.2 (Sutskever, 2012; Sutskever *et al.*, 2013). Keep in mind that with hyperbolic tangent units, the maximum derivative is 1, so that in order to guarantee a Jacobian spectral radius less than 1, the weight matrix should have spectral radius less than 1 as well. However, most derivatives of the hyperbolic tangent will be less than 1, which may explain Sutskever's empirical observation.

More recently, it has been shown that the techniques used to set the weights in ESNs could be used to *initialize* the weights in a fully trainable recurrent network (e.g., trained using back-propagation through time), helping to learn long-term dependencies (Sutskever, 2012; Sutskever *et al.*, 2013). In addition to setting the spectral radius to 1.2, Sutskever sets the recurrent weight matrix to be initially sparse, with only 15 non-zero input weights per hidden unit.

Note that when some eigenvalues of the Jacobian are exactly 1 and none are larger than 1, then some information can be kept in a stable way when moving forward in time, and there are paths in the unfolded computational graph through which back-propagated gradients neither vanish nor explode. The Jacobians tell us about differential propagation. They tell us how a vector of small changes gets propagated forward (when right-multiplying that vector with the product of the Jacobian matrices) or backward (when left-multiplying that vector with the product of the Jacobian matrices). Of course, because these networks are typically not linear, the forward propagation is different from this differential forward propagation, but the differential propagation is important to understand how small changes in input or parameters can influence the later states and outputs. The next two sections show methods to keep information for a very long time by making some paths in the unfolded graph correspond to "multiplying by 1" at each step.

10.7.2 Combining Short and Long Paths in the Unfolded Flow Graph

An old idea that has been proposed to deal with the difficulty of learning long-term dependencies is to use recurrent connections with long delays (Lin *et al.*, 1996). Whereas the ordinary recurrent connections are associated with a delay of 1 (relating the state at t with the state at $t + 1$), it is possible to construct recurrent networks with longer delays (Bengio, 1991), following the idea of incorporating delays in feedforward neural networks (Lang and Hinton, 1988) in order to capture temporal structure (with Time-Delay Neural Networks, which are the

1-D predecessors of Convolutional Neural Networks, discussed in Chapter 9).

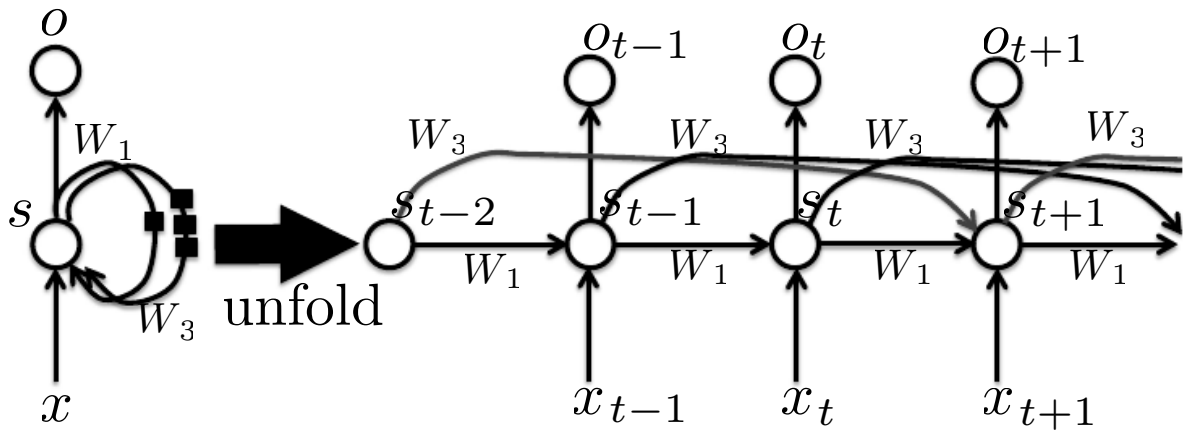

Figure 10.16: A recurrent neural networks with *delays*, in which some of the connections reach back in time to more than one time step. Left: connectivity of the recurrent net, with square boxes indicating the number of time delays associated with a connection. Right: unfolded recurrent network. In the figure there are regular recurrent connections with a delay of 1 time step (W_1) and recurrent connections with a delay of 3 time steps (W_3). The advantage of these longer-delay connections is that they allow to connect past states to future states through shorter paths (3 times shorter, here), going through these longer delay connections (in red).

As we have seen in Section 8.2.6, gradients may vanish or explode exponentially *with respect to the number of time steps*. If we have recurrent connections with a time-delay of d, then instead of the vanishing or explosion going as $O(\lambda^T)$ over T time steps (where λ is the largest eigenvalue of the Jacobians $\frac{\partial s_t}{\partial s_{t-1}}$), the unfolded recurrent network now has paths through which gradients grow as $O(\lambda^{T/d})$ because the number of effective steps is T/d. This allows the learning algorithm to capture longer dependencies although not all long-term dependencies may be well represented in this way. This idea was first explored in Lin *et al.* (1996) and is illustrated in Fig. 10.16.

10.7.3 Leaky Units and a Hierarchy of Different Time Scales

A related idea in order to obtain paths on which the product of derivatives is close to 1 is to have units with *linear* self-connections and a weight near 1 on these connections. The strength of that linear self-connection corresponds to a time scale and thus we can have different hidden units which operate at different time scales (Mozer, 1992). Depending on how close to 1 these self-connection weights are, information can travel forward and gradients backward with a different rate of "forgetting" or contraction to 0, i.e., a different *time scale*. One can view this idea as a smooth variant of the idea of having different delays in the connections

presented in the previous section. Such ideas were proposed in Mozer (1992); ElHihi and Bengio (1996), before a closely related idea discussed in the next section of *gating* these self-connections in order to let the network control at what rate each unit should be contracting. Leaky units were found to be useful (Jaeger *et al.*, 2007) in the context of echo state networks, introduced above (Sec. 10.7.1).

The idea of leaky units with a self-connection actually arises naturally when considering a *continuous-time* recurrent neural network such as

$$\dot{s}_i \tau_i = -s_i + \sigma(b_i + \boldsymbol{W}_{i,:}\boldsymbol{s} + \boldsymbol{U}_{i,:}\boldsymbol{x})$$

where σ is the neural non-linearity (e.g., sigmoid or tanh), $\tau_i > 0$ is a time constant and $\dot{s}_i$ indicates the temporal derivative of unit s_i. A related equation is

$$\dot{s}_i \tau_i = -s_i + (b_i + \boldsymbol{W}_{i,:}\sigma(\boldsymbol{s}) + \boldsymbol{U}_{i,:}\boldsymbol{x})$$

where the state vector $\boldsymbol{s}$ (with elements s_i) now represents the pre-activation of the hidden units.

When discretizing in time such equations (which changes the meaning of τ), one gets

$$\begin{aligned} s_{t+1,i} - s_{t,i} &= -\frac{s_{t,i}}{\tau_i} + \frac{1}{\tau_i}\sigma(b_i + \boldsymbol{W}_{i,:}\boldsymbol{s}_t + \boldsymbol{U}_{i,:}\boldsymbol{x}_t) \\ s_{t+1,i} &= (1 - \frac{1}{\tau_i})s_{t,i} + \frac{1}{\tau_i}\sigma(b_i + \boldsymbol{W}_{i,:}\boldsymbol{s}_t + \boldsymbol{U}_{i,:}\boldsymbol{x}_t). \end{aligned} \tag{10.7}$$

We see that the new value of the state is a convex linear combination of the old value and of the value computed based on current inputs and recurrent weights, if $1 \leq \tau_i < \infty$. When $\tau_i = 1$, there is no linear self-recurrence, only the non-linear update which we find in ordinary recurrent networks. When $\tau_i > 1$, this linear recurrence allows gradients to propagate more easily. When τ_i is large, the state changes very slowly, integrating the past values associated with the input sequence.

By associating different time scales τ_i with different units, one obtains different paths corresponding to different forgetting rates. There are two basic strategies for setting these time constants. One strategy is to manually fix them to values that remain constant, for eample by sampling their values from some distribution once at initialization time. Another strategy is to make the time constants free parameters and learn them. Having such leaky units at different time scales appears to help with long-term dependencies (Mozer, 1992; Pascanu *et al.*, 2013a).

Note that the time constant τ corresponds to a *self-weight* of $(1 - \frac{1}{\tau})$, *without any non-linearity involved in the self-recurrence*. If the recursive computation of the leaky unit is expanded, we find that it computes a weighted average of past input values, with weights that are exponentially decaying for older values, and

τ controls that exponential rate of decay. In the extreme case where $\tau \to \infty$, the weights converge to being all the same: the leaky unit just takes a simple *average* of contributions from the past. In that case, there is no associated vanishing or exploding effect. An alternative is to avoid the weight of $\frac{1}{\tau_i}$ in front of $\sigma\left(b_i + \boldsymbol{W}\boldsymbol{s}^{(t)} + \boldsymbol{U}\boldsymbol{x}^{(t)}\right)$, thus making the state *sum* all the past values when τ_i is large, instead of averaging them.

10.7.4 The Long-Short-Term-Memory Architecture and Other Gated RNNs

Whereas in the previous section we consider creating paths where derivatives neither vanish nor explode too quickly by introducing self-loops, leaky units have self-weights that are not context-dependent: they are fixed, or learned, but remain constant during a whole test sequence.

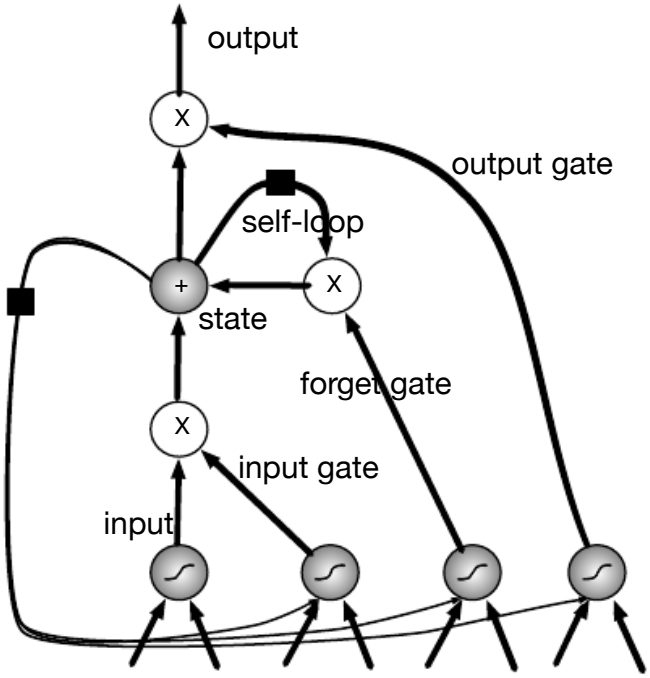

Figure 10.17: Block diagram of the LSTM recurrent network "cell". Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid non-linearity, while the input unit can have any squashing non-linearity. The state unit can also be used as extra input to the gating units. The black square indicates a delay of 1 time unit.

It is worthwhile to consider the role played by leaky units: they allow the

network to *accumulate* information (e.g. evidence for a particular feature or category) over a long duration. However, once that information gets used, it might be useful for the neural network to *forget* the old state. For example, if a sequence is made of subsequences and we want a leaky unit to accumulate evidence inside each sub-subsequence, we need a mechanism to forget the old state by setting it to zero and starting to count from fresh. Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it.

LSTM

This clever idea of conditioning the forgetting on the context is a core contribution of the Long-Short-Term-Memory (LSTM) algorithm (Hochreiter and Schmidhuber, 1997), described below. Several variants of the LSTM are found in the literature (Hochreiter and Schmidhuber, 1997; Graves, 2012; Graves *et al.*, 2013; Graves, 2013; Sutskever *et al.*, 2014a) but the principle is always to have a linear self-loop through which gradients can flow for long durations. By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically (even for fixed parameters, but based on the input sequence). The LSTM has been found extremely successful in a number of applications, such as unconstrained handwriting recognition (Graves *et al.*, 2009), speech recognition (Graves *et al.*, 2013; Graves and Jaitly, 2014), handwriting generation (Graves, 2013), machine translation (Sutskever *et al.*, 2014a), image to text conversion (captioning) (Kiros *et al.*, 2014b; Vinyals *et al.*, 2014b; Xu *et al.*, 2015b) and parsing (Vinyals *et al.*, 2014a).

The LSTM block diagram is illustrated in Fig. 10.17. The corresponding forward (state update) equations are given below, in the case of a shallow recurrent network architecture. Deeper architectures have been successfully used in Graves *et al.* (2013); Pascanu *et al.* (2014a). Instead of a unit that simply applies an element-wise non-linearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have "LSTM cells" that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit $s_{t,i}$ that has a linear self-loop similar to the leaky units described in the previous section. However, here, the self-loop weight (or the associated time constant) is controlled by a *forget gate* unit $h^f_{t,i}$ (for time step t and cell i), that sets this weight to a value between 0 and 1 via a sigmoid unit.

$$h^f_{t,i} = \text{sigmoid}(b^f_i + \sum_j U^f_{ij} x_{t,j} + \sum_j W^f_{ij} h_{t,j}), \tag{10.8}$$

where $\boldsymbol{x}_t$ is the current input vector and $\boldsymbol{h}_t$ is the current hidden layer vector,

containing the outputs of all the LSTM cells, and $\boldsymbol{b}^f$,$\boldsymbol{U}^f$, $\boldsymbol{W}^f$ are respectively biases, input weights and recurrent weights for the forget gates. The LSTM cell internal state is thus updated as follows, following the pattern of Eq. 10.7, but with a conditional self-loop weight $h^f_{t,i}$:

$$s_{t+1,i} = h^f_{t,i} s_{t,i} + h^e_{t,i} \sigma(b_i + \sum_j U_{ij} x_{t,j} + \sum_j W_{ij} h_{t,j}), \tag{10.9}$$

where $\boldsymbol{b}$, $\boldsymbol{U}$ and $\boldsymbol{W}$ respectively denote the biases, input weights and recurrent weights into the LSTM cell. The *external input gate* unit $h^e_{t,i}$ is computed similarly to the forget gate (i.e., with a sigmoid unit to obtain a gating value between 0 and 1), but with its own parameters:

$$h^e_{t,i} = \text{sigmoid}(b^e_i + \sum_j U^e_{ij} x_{t,j} + \sum_j W^e_{ij} h_{t,j}). \tag{10.10}$$

The output $h_{t+1,i}$ of the LSTM cell can also be shut off, via the *output gate* $h^o_{t,i}$, which also uses a sigmoid unit for gating:

$$\begin{aligned} h_{t+1,i} &= \tanh(s_{t+1,i}) h^o_{t,i} \\ h^o_{t,i} &= \text{sigmoid}(b^o_i + \sum_j U^o_{ij} x_{t,j} + \sum_j W^o_{ij} h_{t,j}) \end{aligned} \tag{10.11}$$

which has parameters $\boldsymbol{b}^o$, $\boldsymbol{U}^o$, $\boldsymbol{W}^o$ for its biases, input weights and recurrent weights, respectively. Among the variants, one can choose to use the cell state $s_{t,i}$ as an extra input (with its weight) into the three gates of the i-th unit, as shown in Fig. 10.17. This would require three additional parameters.

LSTM networks have been shown to learn long-term dependencies more easily than the simple recurrent architectures, first on artificial data sets designed for testing the ability to learn long-term dependencies Bengio *et al.* (1994); Hochreiter and Schmidhuber (1997); Hochreiter *et al.* (2000), then on challenging sequence processing tasks where state-of-the-art performance was obtained (Graves, 2012; Graves *et al.*, 2013; Sutskever *et al.*, 2014a). Variants and alternatives to the LSTM have been studied and used and are discussed next.

Other Gated RNNs

Which pieces of the LSTM architecture are actually necessary? What other successful architectures could be designed that allow the network to dynamically control the time scale and forgetting behavior of different units?

Some answers to these questions are given with the recent work on gated RNNs, whose units are also known as Gated Recurrent Units (GRU) (Cho *et al.*,

2014b; Chung *et al.*, 2014, 2015; Jozefowicz *et al.*, 2015b; Chrupala *et al.*, 2015), which were successfully used in reaching the MOSES state-of-the-art for English-to-French machine translation (Cho *et al.*, 2014a). The main difference with the LSTM is that a single gating unit simultaneously controls the forgetting factor and the decision to update the state unit, which is natural if we consider the continuous-time interpretation of the self-weight of the state, as in the equation for leaky units, Eq. 10.7. The update equations are the following:

$$h_{t+1,i} = h^u_{t,i} h_{t,i} + (1 - h^u_{t,i}) \sigma(b_i + \sum_j U_{ij} x_{t,j} + \sum_j W_{ij} h^r_{t,j} h_{t,j}). \tag{10.12}$$

where $\boldsymbol{g}^u$ stands for "update" gate and $\boldsymbol{g}^r$ for "reset" gate. Their value is defined as usual:

$$h^u_{t,i} = \text{sigmoid}(b^u_i + \sum_j U^u_{ij} x_{t,j} + \sum_j W^u_{ij} h_{t,j}) \tag{10.13}$$

and

$$h^r_{t,i} = \text{sigmoid}(b^r_i + \sum_j U^r_{ij} x_{t,j} + \sum_j W^r_{ij} h_{t,j}). \tag{10.14}$$

The reset and updates gates can individually "ignore" parts of the state vector. The update gates act like conditional leaky integrators that can linearly gate any dimension, thus choosing to copy it (at one extreme of the sigmoid) or completely ignore it (at the other extreme) by replacing it by the new "target state" value (towards which the leaky integrator wants to converge). The reset gates control which parts of the state get used to compute the next target state, introducing an additional non-linear effect in the relationship between past state and future state.

Many more variants around this theme can be designed. For example the reset gate (or forget gate) output could be shared across a number of hidden units. Or the product of a global gate (covering a whole group of units, e.g., a layer) and a local gate (per unit) could be used to combine global control and local control. However, several investigations over architectural variations of the LSTM and GRU found no variant that would clearly beat both of these across a wide range of tasks (Greff *et al.*, 2015; Jozefowicz *et al.*, 2015a). Greff *et al.* (2015) found that a crucial ingredient is the forget gate, while Jozefowicz *et al.* (2015a) found that adding a bias of 1 to the LSTM forget gate, a practice advocated by Gers *et al.* (2000), makes the LSTM as strong as the best of the explored architectural variants.

10.7.5 Explicit Memory

Intelligence requires knowledge and acquiring knowledge can be done via learning, which has motivated the development of large-scale deep architectures. However,

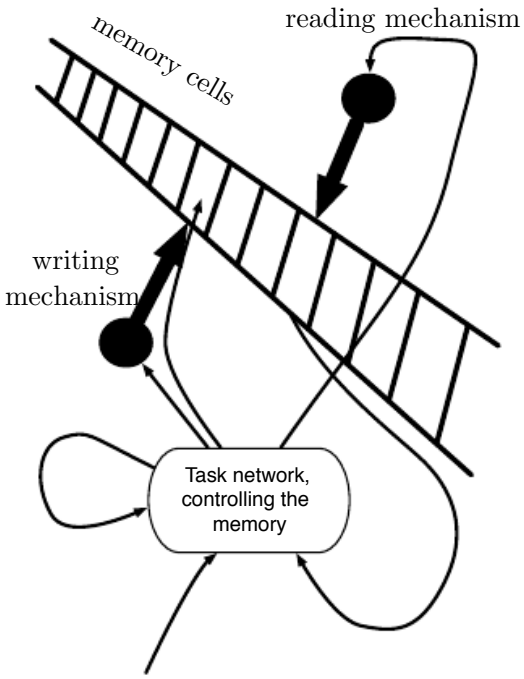

Figure 10.18: A schematic example of memory network architecture, in which we explicitly distinguish the "representation" part of the model (the "task network", here a recurrent net in the bottom) from the "memory" part of the model (the set of cells), which can store facts. From this representation, one learns to "control" the memory, decided from where to read and write (through the reading and writing mechanims, indicated by circles with arrows pointing at the reading and writing addresses).

there are different kinds of knowledge. Some knowledge can be implicit, subconscious, and difficult to verbalize—such as how to walk, or how a dog looks different from a cat. Other knowledge can be explicit, declarative, and relatively straightforward to put into words—every day commonsense knowledge, like "a cat is a kind of animal," or very specific facts that you need to know to accomplish your current goals, like "The meeting with the sales team is at 3:00 PM in room 141."

Neural networks excel at storing implicit knowledge. However, they struggle to memorize facts. Stochastic gradient descent requires many presentations of the same input before it can be stored in a neural networks parameters, and even then, that input will not be stored especially precisely. Graves *et al.* (2014b) hypothesized that this is because neural networks lack the equivalent of the *working memory* system that allows human beings to explicitly hold and manipulate pieces of information that are relevant to achieving some goal. Such explicit memory components would allow our systems not only to rapidly and "intentionally" store and retrieve specific facts but also to sequentially reason with them.

The core idea behind *memory networks* (Weston *et al.*, 2014) and *neural Turing machines* (Graves *et al.*, 2014b) is to add a set of memory cells that can be read from and written to via an addressing mechanism. Each memory cell can be thought of as an extension of the memory cells in LSTMs and GRUs. The difference is that the network outputs an internal state that chooses which cell to read from or write to, just as memory accesses in a digital computer read from or write to a specific address.

It is difficult to optimize functions that produce exact, integer addresses. To alleviate this problem, memory networks and NTMs actually read to or write from many memory cells simultaneously. To read, they take a weighted average of many cells. To write, they modify multiple cells by different amounts. The coefficients for these operations are chosen to be focused on a small number of cells, for example, by producing them via a softmax function. Using these weights with non-zero derivatives allows the functions controlling access to the memory to be optimized using gradient descent. The gradient on these coefficients indicates whether each of them should be increased or decreased, but the gradient will typically be large only for those memory addresses receiving a large coefficient.

These memory cells are typically augmented to contain a vector, rather than the single scalar stored by an LSTM or GRU memory cell. There are two reasons to increase the size of the memory cell. One reason is that we have increased the cost of accessing a memory cell. We pay the computational cost of producing a coefficient for many cells, but we expect these coefficients to cluster around a small number of cells. By reading a vector value, rather than a scalar value, we can offset some of this cost. Another reason to use vector-valued memory cells is

that they allow for *content-based addressing*, where the weight used to read to or write from a cell is a function of that cell. Vector-valued cells allow us to retrieve a complete vector-valued memory if we are able to produce a pattern that matches some but not all of its elements. This is analogous to the way that people can recall the lyrics of a song based on a few words. We can think of a content-based read instruction as saying, "Retrieve the lyrics of the song that has the chorus 'We all live on a yellow submarine." ' Content-based addressing is more useful when we make the objects to be retrieved large—if every letter of the song was stored in a separate memory cell, we would not be able to find them this way. By comparison, *location-based addressing* is not allowed to refer to the content of the memory. We can think of a location-based read instruction as saying "Retrieve the lyrics of the song in slot 347." Location-based addressing can often be a perfectly sensible mechanism even when the memory cells are small.

If the content of a cell in a memory network is copied (not forgotten) at most time steps, then the information it contains can be propagated forward in time and the gradients propagated backward in time without either vanishing or exploding. This is illustrated in Fig. 10.18, where we see that a "task neural network" is coupled with a memory. Although that task neural network could be feedforward or recurrent (the latter being the case in the figure), the overall system is a recurrent network. The task network can choose to read from or write to specific memory addresses. Explicit memory seems to allow models to learn tasks that ordinary RNNs or LSTM RNNs cannot learn. One reason for this advantage may be because information and gradients can be propagated (forward in time or backwards in time, respectively) for very long durations.

As an alternative to back-propagation through weighted averages of memory cells, we can interpret the weights as probabilities and stochastically read just one cell. A variant of the REINFORCE algorithm (Williams, 1992) can estimate a noisy gradient on the addressing weights (Zaremba and Sutskever, 2015). So far, training these stochastic architectures that make hard decisions remains harder than training deterministic algorithms that make soft decisions.

Whether it is soft (allowing back-propagation) or stochastic and hard, the mechanism for choosing an address is in its form identical to the *attention mechanism* which had been previously introduced in the context of machine translation (Bahdanau *et al.*, 2014) and discussed in Sec. 12.4.6. The idea of attention mechanisms for neural networks was introduced even earlier, in the context of handwriting generation (Graves, 2013), in which the focus attention is moving monotonically in the input sequence. In the case of machine translation and memory networks, at each step, the focus of attention can move to a completely different place, compared to the previous step.

10.7.6 Better Optimization

A central optimization difficulty with RNNs regards the learning of long-term dependencies (Hochreiter, 1991; Bengio *et al.*, 1993, 1994). This difficulty has been explained in detail in Section 8.2.6. The gist of the problem is that the composition of the non-linear recurrence with itself over many many time steps yields a highly non-linear function whose derivatives (e.g. of the state at T w.r.t. the state at $t < T$, i.e. the Jacobian matrix $\frac{\partial \boldsymbol{s}_T}{\partial \boldsymbol{s}_t}$) tend to either vanish or explode as $T-t$ increases, because it is equal to the product of the state transition Jacobian matrices $\frac{\partial \boldsymbol{s}_{t+1}}{\partial \boldsymbol{s}_t}$)

If it explodes, the parameter gradient $\nabla_{\boldsymbol{\theta}} L$ also explodes, yielding gradient-based parameter updates that are poor. A simple heuristic but practical solution to this problem is discussed in the next section (Sec. 10.7.7). However, as discussed in Bengio *et al.* (1994), if the state transition Jacobian matrix has eigenvalues that are larger than 1 in magnitude, then it can yield to "unstable" dynamics, in the sense that a bit of information cannot be stored reliably for a long time in the presence of input "noise". Indeed, the state transition Jacobian matrix eigenvalues indicate how a small change in some direction (the corresponding eigenvector) will be expanded (if the eigenvalue is greater than 1) or contracted (if it is less than 1).

An interesting idea proposed in Martens and Sutskever (2011) is that at the same time as first derivatives are becoming smaller in directions associated with long-term effects, *so may the higher derivatives.* In particular, if we use a second-order optimization method (such as the Hessian-free method of Martens and Sutskever (2011)), then we could differentially treat different directions: divide the small first derivative (gradient) by a small second derivative, while not scaling up in the directions where the second derivative is large (and hopefully, the first derivative as well). Whereas in the scalar case, if we add a large number and a small number, the small number is "lost", in the vector case, if we add a large vector with a small vector, it is still possible to recover the information about the direction of the small vector if we have access to information (such as in the second derivative matrix) that tells us how to rescale appropriately each direction.

One disadvantage of many second-order methods, including the Hessian-free method, is that they tend to be geared towards "batch" training (or fairly large minibatches) rather than "stochastic" updates (where only one example or a small minibatch of examples are examined before a parameter update is made). Although the experiments on recurrent networks applied to problems with long-term dependencies showed very encouraging results in Martens and Sutskever (2011), it was later shown that similar results could be obtained by much simpler methods (Sutskever, 2012; Sutskever *et al.*, 2013) involving better initialization, a cheap surrogate to second-order optimization (a variant on the momentum

technique, Section 8.5), and the clipping trick described below.

10.7.7 Clipping Gradients

As discussed in Section 8.2.5, strongly non-linear functions such as those computed by a recurrent net over many time steps tend to have derivatives that can be either very large or very small in magnitude. This is illustrated in Fig.s 8.2 and 8.3, in which we see that the objective function (as a function of the parameters) has a "landscape" in which one finds "cliffs": wide and rather flat regions separated by tiny regions where the objective function changes quickly, forming a kind of cliff.

The difficulty that arises is that when the parameter gradient is very large, a gradient descent parameter update could throw the parameters very far, into a region where the objective function is larger, undoing a lot of the work that had been done to reach the current solution. The gradient tells us the direction that corresponds to the steepest descent within an infinitesimal region surrounding the current parameters. Outside of this infinitesimal region, the cost function may begin to curve back upwards. The update must be chosen to be small enough to avoid traversing too much upward curvature. We typically use learning rates that decay slowly enough that consecutive steps have approximately the same learning rate. A step size that is appropriate for a relatively linear part of the landscape is often inappropriate and causes uphill motion if we enter a more curved part of the landscape on the next step.

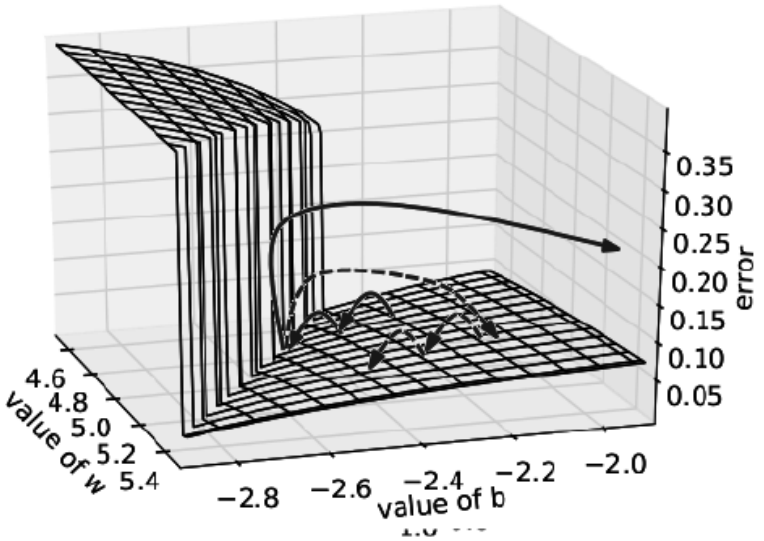

Figure 10.19: Example of the effect of gradient clipping in a recurrent network with two parameters $\boldsymbol{w}$ and $\boldsymbol{b}$. Vertical axis is the objective function to minimize. Note the cliff where the gradient explodes and from where gradient descent can get pushed very far. Clipping the gradient when its norm is above a threshold (Pascanu *et al.*, 2013a) prevents this catastrophic outcome and helps training recurrent nets with long-term dependencies to be captured.

A simple type of solution has been in used by practitioners for many years: *clipping the gradient*. There are different instances of this idea (Mikolov, 2012; Pascanu *et al.*, 2013a). One option is to clip the parameter gradient from a mini-batch *element-wise* (Mikolov, 2012) just before the parameter update. Another is to *clip the norm* $||\boldsymbol{g}||$ *of the gradient* $\boldsymbol{g}$ (Pascanu *et al.*, 2013a) just before the parameter update:

$$\begin{aligned} &\text{if } ||\boldsymbol{g}|| > v \\ &\quad \boldsymbol{g} \leftarrow \frac{\boldsymbol{g} v}{||\boldsymbol{g}||} \end{aligned} \tag{10.15}$$

where v is the norm threshold and $\boldsymbol{g}$ is used to update parameters. Because the gradient of all the parameters (including different groups of parameters, such as weights and biases) is renormalized jointly with a single scaling factor, the latter method has the advantage that it guarantees that each step is still in the gradient direction, but experiments suggest that both forms work similarly. Although the parameter update has the same direction as the true gradient, with gradient norm clipping, the parameter update vector norm is now bounded. This bounded gradient avoids performing a detrimental step when the gradient explodes. In fact, even simply taking a *random step* when the gradient magnitude is above a threshold tends to work almost as well. If the explosion is so severe that the gradient is numerically `Inf` or `Nan` (considered infinite or not-a-number), then

a random step of size v can be taken and will typically move away from the numerically unstable configuration. Clipping the gradient norm per-minibatch will not change the direction of the gradient for an individual minibatch. However, taking the average of the norm-clipped gradient from many minibatches is not equivalent to clipping the norm of the true gradient (the gradient formed from using all examples). Examples that have large gradient norm, as well as examples that appear in the same minibatch as such examples, will have their contribution to the final direction diminished. This stands in contrast to traditional minibatch gradient descent, where the true gradient direction is equal to the average over all minibatch gradients. Put another way, traditional stochastic gradient descent uses an unbiased estimate of the gradient, while gradient descent with norm clipping introduces a heuristic bias that we know empirically to be useful. With element-wise clipping, the direction of the update is not aligned with the true gradient or the minibatch gradient, but it is still a descent direction. It has also been proposed (Graves, 2013) to clip the back-propagated gradient (with respect to hidden units) but no comparison has been published between these variants; we conjecture that all these methods behave similarly.

10.7.8 Regularizing to Encourage Information Flow

Whereas clipping helps dealing with exploding gradients, it does not help with vanishing gradients. To address vanishing gradients and better capture long-term dependencies, we discussed the idea of creating paths in the computational graph of the unfolded recurrent architecture along which the product of gradients associated with arcs is near 1. One approach to achieve this is with LSTMs and other self-loops and gating mechanisms, described above in Section 10.7.4. Another idea is to regularize or constrain the parameters so as to encourage "information flow". In particular, we would like the gradient vector $\nabla_{\boldsymbol{s}_t} L$ being back-propagated to maintain its magnitude (even if there is only a loss at the end of the sequence), i.e., we want

$$\nabla_{\boldsymbol{s}_t} L \, \frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{s}_{t-1}}$$

to be as large as

$$\nabla_{\boldsymbol{s}_t} L.$$

With this objective, Pascanu *et al.* (2013a) propose the following regularizer:

$$\Omega = \sum_{t} \left(\frac{|\nabla_{\boldsymbol{s}_t} L \frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{s}_{t-1}}|}{||\nabla_{\boldsymbol{s}_t} L||} - 1 \right)^2 . \tag{10.16}$$

It looks like computing the gradient of this regularizer is difficult, but Pascanu *et al.* (2013a) propose an approximation in which we consider the back-propagated vectors $\nabla_{\boldsymbol{s}} L$ as if they were constants (for the purpose of this regularizer, i.e., no need to back-prop through them). The experiments with this regularizer suggest that, if combined with the norm clipping heuristic (which handles gradient explosion), it can considerably increase the span of the dependencies that an RNN can learn. Because it keeps the RNN dynamics on the edge of explosive gradients, the gradient clipping is particularly important: otherwise gradient explosion prevents learning to succeed.

10.7.9 Organizing the State at Multiple Time Scales

Another promising approach to handle long-term dependencies is the old idea of organizing the state of the RNN at multiple time-scales (El Hihi and Bengio, 1996), with information flowing more easily through long distances at the slower time scales. This is illustrated in Fig. 10.20.

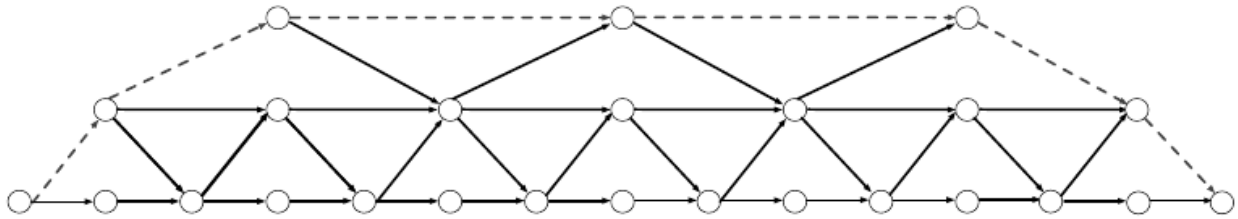

Figure 10.20: Example of a multi-scale recurrent net architecture (unfolded in time), with higher levels operating at a slower time scale. Information can flow unhampered (either forward or backward in time) over longer durations at the higher levels, thus creating long-paths (such as the dotted path) through which long-term dependencies between elements of the input/output sequence can be captured.

There are different ways in which a group of recurrent units can be forced to operate at different time scales. One option is to make the recurrent units leaky (as in Eq. 10.7), but to have different groups of units associated with different fixed time scales. This was the proposal in Mozer (1992) and has been successfully used in Pascanu *et al.* (2013a). Another option is to have explicit and discrete updates taking place at different times, with a different frequency for different groups of units, as in Fig. 10.20. This is the approach of El Hihi and Bengio (1996); Koutnik *et al.* (2014) and it also worked well on a number of benchmark datasets.