Chapter 10

Sequence Modeling: Recurrent and Recursive Nets

Recurrent neural networks, or RNNs (Rumelhart *et al.*, 1986a), are a family of neural networks for processing sequential data. Much as a convolutional network is a neural network that is specialized for processing a grid of values $\mathbf{X}$ such as an image, a recurrent neural network is a neural network that is specialized for processing a sequence of values $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$. Just as convolutional networks can readily scale to images with large width and height, and some convolutional networks can process images of variable size, recurrent networks can scale to much longer sequences than would be practical for networks without sequence-based specialization. Most recurrent networks can also process sequences of variable length.

To go from multilayer networks to recurrent networks, we need to take advantage of one of the early ideas found in machine learning and statistical models of the 1980s: sharing parameters 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 a specific piece of information can occur at multiple positions within the sequence. For example, consider the two sentences "I went to Nepal in 2009" and "In 2009, I went to Nepal." If we ask a machine learning model to read each sentence and extract the year in which the narrator went to Nepal, we would like it to recognize the year 2009 as the relevant piece of information, whether it appears in the sixth

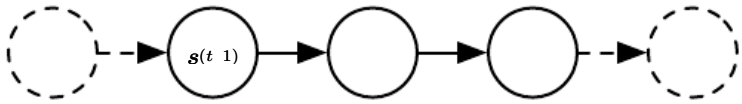
word or in the second word of the sentence. Suppose that we trained a feedforward network that processes sentences of fixed length. A traditional fully connected feedforward network would have separate parameters for each input feature, so it would need to learn all the rules of the language separately at each position in the sentence. By comparison, a recurrent neural network shares the same weights across several time steps.

A related idea is the use of convolution across a 1-D temporal sequence. This convolutional approach is the basis for time-delay neural networks (Lang and Hinton, 1988; Waibel *et al.*, 1989; Lang *et al.*, 1990). The convolution operation allows a network to share parameters across time but is shallow. The output of convolution is a sequence where each member of the output is a function of a small number of neighboring members of the input. The idea of parameter sharing manifests in the application of the same convolution kernel at each time step. Recurrent networks share parameters in a different way. Each member of the output is a function of the previous members of the output. Each member of the output is produced using the same update rule applied to the previous outputs. This recurrent formulation results in the sharing of parameters through a very deep computational graph.

For the simplicity of exposition, we refer to RNNs as operating on a sequence that contains vectors $\boldsymbol{x}^{(t)}$ with the time step index t ranging from 1 to τ. In practice, recurrent networks usually operate on minibatches of such sequences, with a different sequence length τ for each member of the minibatch. We have omitted the minibatch indices to simplify notation. Moreover, the time step index need not literally refer to the passage of time in the real world. Sometimes it refers only to the position in the sequence. RNNs may also be applied in two dimensions across spatial data such as images, and even when applied to data involving time, the network may have connections that go backward in time, provided that the entire sequence is observed before it is provided to the network.

This chapter extends the idea of a computational graph to include cycles. These cycles represent the influence of the present value of a variable on its own value at a future time step. Such computational graphs allow us to define recurrent neural networks. We then describe many different ways to construct, train, and use recurrent neural networks.

For more information on recurrent neural networks than is available in this chapter, we refer the reader to the textbook of Graves (2012).



t

f t $t+1$
$\boldsymbol{\theta}$ f

A computational 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 6.5.1 for a general introduction. In this section we explain the idea of **unfolding** a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events. Unfolding this graph results in the sharing of parameters across a deep network structure.

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

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t\ 1)}; \boldsymbol{\theta}), \tag{10.1}$$

where $\boldsymbol{s}^{(t)}$ is called the state of the system.

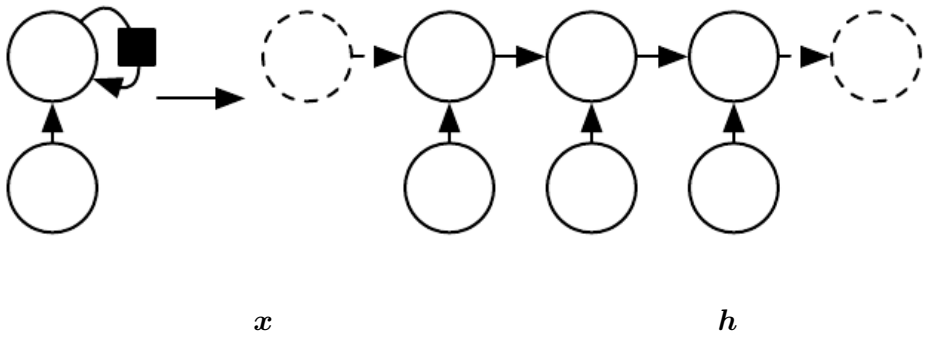
Equation 10.1 is recurrent because the definition of $\boldsymbol{s}$ at time t refers back to the same definition at time $t \quad 1$.

For a finite number of time steps τ, the graph can be unfolded by applying the definition $_{\tau} \quad 1$ times. For example, if we unfold equation 10.1 for $\tau = 3$ time steps, we obtain

$$\begin{aligned} \boldsymbol{s}^{(3)} &= f(\boldsymbol{s}^{(2)}; \boldsymbol{\theta}) && (10.2) \\ &= f(f(\boldsymbol{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta}). && (10.3) \end{aligned}$$

Unfolding the equation by repeatedly applying the definition in this way has yielded an expression that does not involve recurrence. Such an expression can now be represented by a traditional directed acyclic computational graph. The unfolded computational graph of equation 10.1 and equation 10.3 is illustrated in figure 10.1.

As another example, let us consider a dynamical system driven by an external



signal $\boldsymbol{x}^{(t)}$,

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \tag{10.4}$$

where we see that the state now contains information about the whole past sequence.

Recurrent neural networks can be built in many different ways. Much as almost any function can be considered a feedforward neural network, essentially any function involving recurrence can be considered a recurrent neural network.

Many recurrent neural networks use equation 10.5 or a similar equation to define the values of their hidden units. To indicate that the state is the hidden units of the network, we now rewrite equation 10.4 using the variable $\boldsymbol{h}$ to represent the state,

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \tag{10.5}$$

illustrated in figure 10.2; typical RNNs will add extra architectural features such as output layers that read information out of the state $\boldsymbol{h}$ to make predictions.

When the recurrent network is trained to perform a task that requires predicting the future from the past, the network typically learns to use $\boldsymbol{h}^{(t)}$ as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t. 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 $\boldsymbol{h}^{(t)}$. 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, storing all the information in the input sequence up to time t may not be necessary; storing only enough information to predict the rest of the sentence is sufficient. The most demanding situation is when we ask $\boldsymbol{h}^{(t)}$ to be rich enough to allow one to approximately recover the input sequence, as in autoencoder

frameworks (chapter 14).

Equation 10.5 can be drawn in two different ways. One way to draw the RNN is with a diagram containing one node for every component that might exist in a physical implementation of the model, such as a biological neural network. In this view, the network defines a circuit that operates in real time, with physical parts whose current state can influence their future state, as in the left of figure 10.2. Throughout this chapter, we use a black square in a circuit diagram to indicate that an interaction takes place with a delay of a single time step, from the state at time t to the state at time $t+1$. The other way to draw the RNN is as an unfolded computational graph, in which each component is represented by many different variables, with one variable per time step, representing the state of the component at that point in time. Each variable for each time step is drawn as a separate node of the computational graph, as in the right of figure 10.2. What we call unfolding is the operation that maps a circuit, as in the left side of the figure, to a computational graph with repeated pieces, as in the right side. The unfolded graph now has a size that depends on the sequence length.

We can represent the unfolded recurrence after t steps with a function $g^{(t)}$:

$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t\ 1)}, \boldsymbol{x}^{(t\ 2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) \tag{10.6}$$

$$= f(\boldsymbol{h}^{(t\ 1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}). \tag{10.7}$$

The function $g^{(t)}$ takes the whole past sequence $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t\ 1)}, \boldsymbol{x}^{(t\ 2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ as input and produces the current state, but the unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f. The unfolding process thus introduces two major advantages:

1. Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states.

2. It is possible to use the *same* transition function f with the same parameters at every time step.

These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths, rather than needing to learn a separate model $g^{(t)}$ for all possible time steps. Learning a single shared model allows generalization to sequence lengths that did not appear in the training set, and enables the model to be estimated with far fewer training examples than would be required without parameter sharing.

Both the recurrent graph and the unrolled graph have their uses. The recurrent graph is succinct. The unfolded graph provides an explicit description of which computations to perform. The unfolded graph also helps illustrate the idea of information flow forward in time (computing outputs and losses) and backward in time (computing gradients) by explicitly showing the path along which this information flows.

Armed with the graph-unrolling and parameter-sharing ideas of section 10.1, we can design a wide variety of recurrent neural networks.

Some examples of important design patterns for recurrent neural networks include the following:

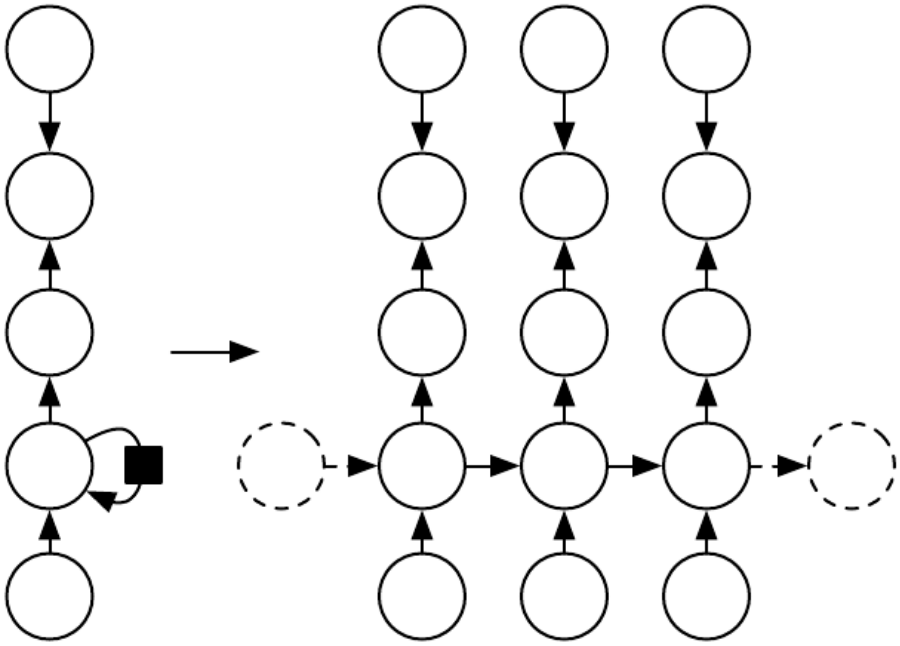
Recurrent networks that produce an output at each time step and have recurrent connections between hidden units, illustrated in figure 10.3

Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step, illustrated in figure 10.4

Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output, illustrated in figure 10.5

Figure 10.3 is a reasonably representative example that we return to throughout most of the chapter.

The recurrent neural network of figure 10.3 and equation 10.8 is universal in the sense that any function computable by a Turing machine can be computed by such 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). The functions computable by a Turing machine are discrete, so these results regard exact implementation of the function, not approximations. The RNN, when used as a Turing machine, takes a binary sequence as input, and its outputs must be discretized to provide a binary output. It is possible to compute all functions in this setting using a single specific RNN of finite size (Siegelmann and Sontag[1995] use 886 units). The "input" of the Turing machine is



$\boldsymbol{x}$ $\boldsymbol{o}$

L $\boldsymbol{o}$ $\boldsymbol{y}$

$\boldsymbol{o}$ L

$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o})$ $\boldsymbol{y}$

$\boldsymbol{U}$

$\boldsymbol{W}$

$\boldsymbol{V}$

a specification of the function to be computed, so the same network that simulates this Turing machine is sufficient for all problems. The theoretical RNN used for the proof can simulate an unbounded stack by representing its activations and weights with rational numbers of unbounded precision.

We now develop the forward propagation equations for the RNN depicted in figure 10.3. The figure does not specify the choice of activation function for the hidden units. Here we assume the hyperbolic tangent activation function. Also, the figure does not specify exactly what form the output and loss function take. Here we assume that the output is discrete, as if the RNN is used to predict words or characters. A natural way to represent discrete variables is to regard the output $\boldsymbol{o}$ as giving the unnormalized log probabilities of each possible value of the discrete variable. We can then apply the softmax operation as a post-processing step to obtain a vector $\hat{\boldsymbol{y}}$ of normalized probabilities over the output. Forward propagation begins with a specification of the initial state $\boldsymbol{h}^{(0)}$. Then, for each time step from $t = 1$ to $t = \tau$, we apply the following update equations:

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t\ 1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}, \tag{10.8}$$

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)}), \tag{10.9}$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}, \tag{10.10}$$

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}(\boldsymbol{o}^{(t)}), \tag{10.11}$$

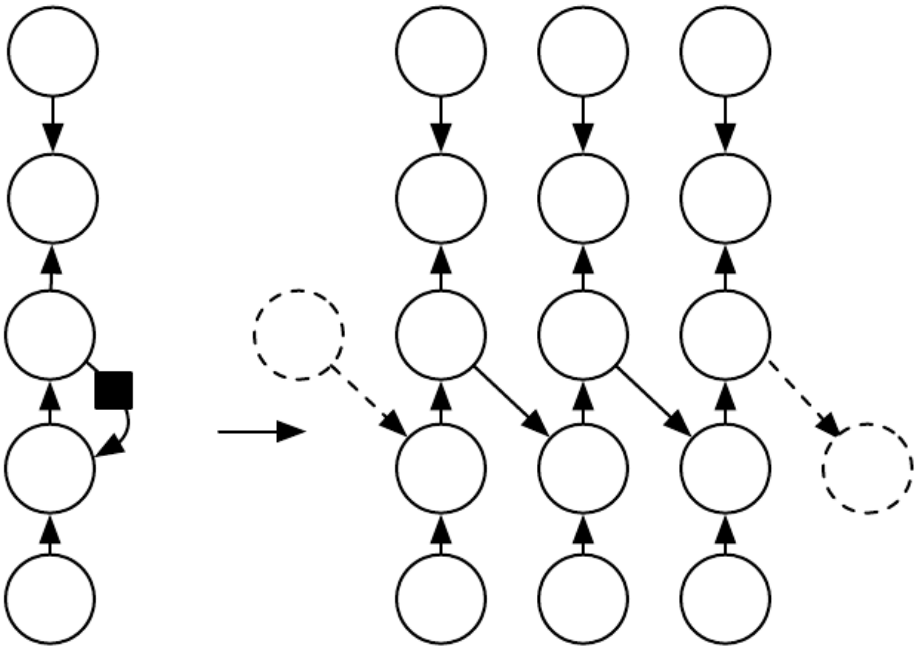
where the parameters are the bias vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ along with the weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$, respectively, for input-to-hidden, hidden-to-output and hidden-to-hidden connections. This is an example of a recurrent network that maps an input sequence to an output sequence of the same length. The total loss for a given sequence of $\boldsymbol{x}$ values paired with a sequence of $\boldsymbol{y}$ values would then be just the sum of the losses over all the time steps. For example, if $L^{(t)}$ is the negative log-likelihood of $y^{(t)}$ given $\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}$, then

$$L \quad \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)} \ , \ \boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(\tau)} \tag{10.12}$$

$$= \quad L^{(t)} \tag{10.13}$$

$$= \quad \log p_{\text{model}} \quad y^{(t)} \quad \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)} \quad , \tag{10.14}$$

where $p_{\text{model}} \quad y^{(t)} \quad \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{\boldsymbol{y}}^{(t)}$. Computing the gradient of this loss function with respect to the parameters is an expensive operation. The gradient computation involves performing a forward propagation pass moving left to right through our illustration



t $\boldsymbol{x}$

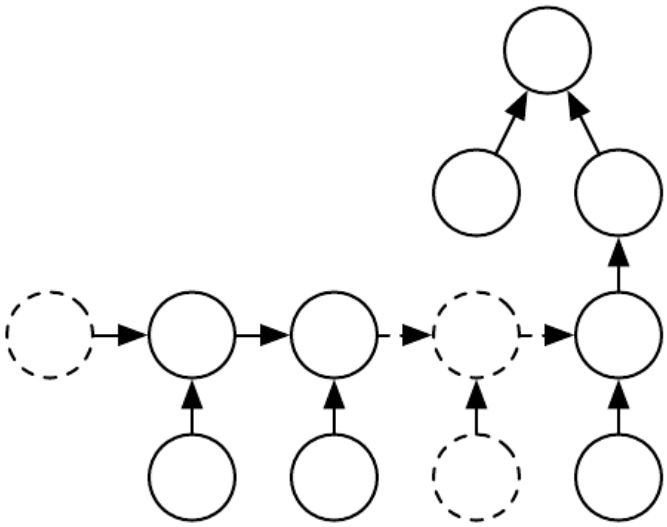
$\boldsymbol{h}$ $\boldsymbol{o}$ $\boldsymbol{y}$ L

$\boldsymbol{h}$ $\boldsymbol{h}$

$\boldsymbol{o}$ $\boldsymbol{o}$

$\boldsymbol{h}$ $\boldsymbol{h}$

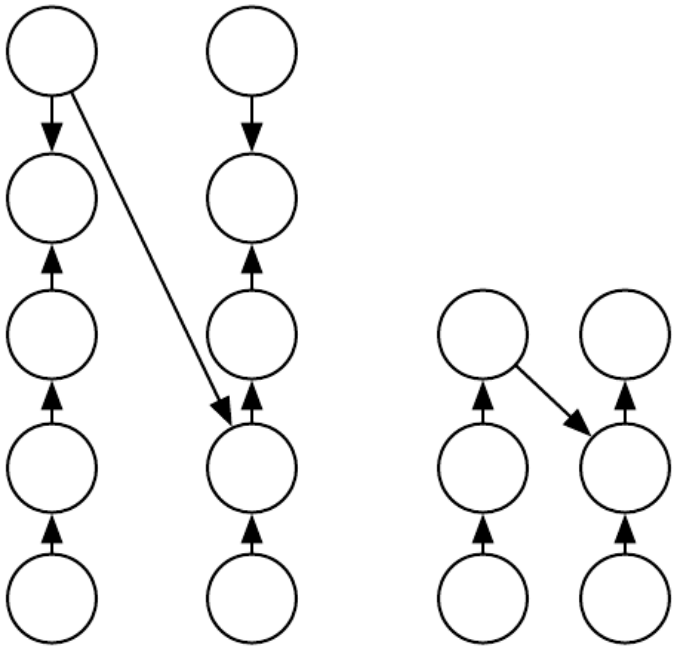
$\boldsymbol{o}$



$\boldsymbol{o}$

of the unrolled graph in figure 10.3, followed by a backward propagation pass moving right to left through the graph. The runtime is $O(\tau)$ and cannot be reduced by parallelization because the forward propagation graph is inherently sequential; each time step may be computed only after the previous one. States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also $O(\tau)$. The back-propagation algorithm applied to the unrolled graph with $O(\tau)$ cost is called **back-propagation through time** (BPTT) and is discussed further in section 10.2.2. The network with recurrence between hidden units is thus very powerful but also expensive to train. Is there an alternative?

The network with recurrent connections only from the output at one time step to the hidden units at the next time step (shown in figure 10.4) is strictly less powerful because it lacks hidden-to-hidden recurrent connections. For example, it cannot simulate a universal Turing machine. Because this network lacks hidden-to-hidden recurrence, it requires that the output units capture all the information about the past that the network will use to predict the future. Because the output units are



$\boldsymbol{y}$

$\boldsymbol{h}$

$\boldsymbol{y}$

$\boldsymbol{o}$

explicitly trained to match the training set targets, they are unlikely to capture the necessary information about the past history of the input, unless the user knows how to describe the full state of the system and provides it as part of the training set targets. The advantage of eliminating hidden-to-hidden recurrence is that, for any loss function based on comparing the prediction at time t to the training target at time t, all the time steps are decoupled. Training can thus be parallelized, with the gradient for each step t computed in isolation. There is no need to compute the output for the previous time step first, because the training set provides the ideal value of that output.

Models that have recurrent connections from their outputs leading back into the model may be trained with **teacher forcing**. Teacher forcing is a procedure that emerges from the maximum likelihood criterion, in which during training the model receives the ground truth output $y^{(t)}$ as input at time $t + 1$. We can see this by examining a sequence with two time steps. The conditional maximum

likelihood criterion is

$$\log p \quad \boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} \quad \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)} \tag{10.15}$$

$$= \log p \quad \boldsymbol{y}^{(2)} \quad \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)} \quad + \log p \quad \boldsymbol{y}^{(1)} \quad \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)} \quad . \tag{10.16}$$

In this example, we see that at time $t = 2$, the model is trained to maximize the conditional probability of $\boldsymbol{y}^{(2)}$ given *both* the $\boldsymbol{x}$ sequence so far and the previous $\boldsymbol{y}$ value from the training set. Maximum likelihood thus specifies that during training, rather than feeding the model's own output back into itself, these connections should be fed with the target values specifying what the correct output should be. This is illustrated in figure 10.6.

We originally motivated teacher forcing as allowing us to avoid back-propagation through time in models that lack hidden-to-hidden connections. Teacher forcing may still be applied to models that have hidden-to-hidden connections as long as they have connections from the output at one time step to values computed in the next time step. As soon as the hidden units become a function of earlier time steps, however, the BPTT algorithm is necessary. Some models may thus be trained with both teacher forcing and BPTT.

The disadvantage of strict teacher forcing arises if the network is going to be later used in an **open-loop** mode, with the network outputs (or samples from the output distribution) fed back as input. In this case, the kind of inputs that the network sees during training could be quite different from the kind of inputs that it will see at test time. One way to mitigate this problem is to train with both teacher-forced inputs and free-running inputs, for example by 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 input conditions (such as those it generates itself in the free-running mode) not seen during training and how to map the state back toward one that will make the network generate proper outputs after a few steps. Another approach (Bengio *et al.*, 2015b) to mitigate the gap between the inputs seen at training time and the inputs seen at test time 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.

Computing the gradient through a recurrent neural network is straightforward. One simply applies the generalized back-propagation algorithm of section 6.5.6 to the unrolled computational graph. No specialized algorithms are necessary.

Gradients obtained by back-propagation may then be used with any general-purpose gradient-based techniques to train an RNN.

To gain some intuition for how the BPTT algorithm behaves, we provide an example of how to compute gradients by BPTT for the RNN equations above (equation 10.8 and equation 10.12). The nodes of our computational graph include the parameters $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ as well as the sequence of nodes indexed by t for $\boldsymbol{x}^{(t)}$, $\boldsymbol{h}^{(t)}$, $\boldsymbol{o}^{(t)}$ and $L^{(t)}$. For each node N we need to compute the gradient

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:

$$\frac{\partial L}{\partial L^{(t)}} = 1. \tag{10.17}$$

In this derivation we assume that the outputs $\boldsymbol{o}^{(t)}$ are used as the argument to the softmax function to obtain the vector $\hat{\boldsymbol{y}}$ of probabilities over the output. We also assume that the loss is the negative log-likelihood of the true target $y^{(t)}$ given the input so far. The gradient L on the outputs at time step t, for all i, t, is as follows:

$$(\quad L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} \quad _{i,y} \quad . \tag{10.18}$$

We work our way backward, starting from the end of the sequence. At the final time step τ, $\boldsymbol{h}^{(\tau)}$ only has $\boldsymbol{o}^{(\tau)}$ as a descendent, so its gradient is simple:

$$L = \boldsymbol{V} \qquad L. \tag{10.19}$$

We can then iterate backward in time to back-propagate gradients through time, from $t = _{\tau} \quad 1$ down to $t = 1$, noting that $\boldsymbol{h}^{(t)}$ (for $t < \tau$) has as descendents both $\boldsymbol{o}^{(t)}$ and $\boldsymbol{h}^{(t+1)}$. Its gradient is thus given by

$$L = \quad \frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}} \quad (\qquad L) + \quad \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \quad (\qquad L) \tag{10.20}$$

$$= \boldsymbol{W} \quad (\qquad L) \operatorname{diag} \quad 1 \quad \boldsymbol{h}^{(t+1)} \quad ^2 + \boldsymbol{V} \quad (\qquad L), \tag{10.21}$$

where diag $\quad 1 \quad \boldsymbol{h}^{(t+1)} \quad ^2$ indicates the diagonal matrix containing the elements $1 \quad (h_i^{(t+1)})^2$. This is the Jacobian of the hyperbolic tangent associated with the hidden unit i at time $t + 1$.

Once the gradients on the internal nodes of the computational graph are obtained, we can obtain the gradients on the parameter nodes. Because the

parameters are shared across many time steps, we must take some care when denoting calculus operations involving these variables. The equations we wish to implement use the method of section 6.5.6, which computes the contribution of a single edge in the computational graph to the gradient. The f operator used in calculus, however, takes into account the contribution of $\boldsymbol{W}$ to the value of f due to *all* edges in the computational graph. To resolve this ambiguity, we introduce dummy variables $\boldsymbol{W}^{(t)}$ that are defined to be copies of $\boldsymbol{W}$ but with each $\boldsymbol{W}^{(t)}$ used only at time step t. We may then use to denote the contribution of the weights at time step t to the gradient.

Using this notation, the gradient on the remaining parameters is given by

$$L = \quad \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}} \qquad L = \quad L, \tag{10.22}$$

$$L = \quad \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}} \qquad L = \quad \text{diag} \quad 1 \qquad \boldsymbol{h}^{(t)} \quad ^2 \qquad L, \tag{10.23}$$

$$L = \quad \frac{\partial L}{\partial o_i^{(t)}} \qquad o_i^{(t)} = \quad (\qquad L) \, \boldsymbol{h}^{(t)} \quad , \tag{10.24}$$

$$L = \quad \frac{\partial L}{\partial h_i^{(t)}} \qquad h_i^{(t)} \tag{10.25}$$

$$= \quad \text{diag} \quad 1 \qquad \boldsymbol{h}^{(t)} \quad ^2 \quad (\qquad L) \, \boldsymbol{h}^{(t \ 1)} \quad , \tag{10.26}$$

$$L = \quad \frac{\partial L}{\partial h_i^{(t)}} \qquad h_i^{(t)} \tag{10.27}$$

$$= \quad \text{diag} \quad 1 \qquad \boldsymbol{h}^{(t)} \quad ^2 \quad (\qquad L) \, \boldsymbol{x}^{(t)} \quad , \tag{10.28}$$

We do not need to compute the gradient with respect to $\boldsymbol{x}^{(t)}$ for training because it does not have any parameters as ancestors in the computational graph defining the loss.

In the example recurrent network we have developed so far, the losses $L^{(t)}$ were cross-entropies between training targets $\boldsymbol{y}^{(t)}$ and outputs $\boldsymbol{o}^{(t)}$. As with a feedforward network, it is in principle possible to use almost any loss with a recurrent network.

The loss should be chosen based on the task. As with a feedforward network, we usually wish to interpret the output of the RNN as a probability distribution, and we usually use the cross-entropy associated with that distribution to define the loss. Mean squared error is the cross-entropy loss associated with an output distribution that is a unit Gaussian, for example, just as with a feedforward network.

When we use a predictive log-likelihood training objective, such as equation 10.12, we train the RNN to estimate the conditional distribution of the next sequence element $\boldsymbol{y}^{(t)}$ given the past inputs. This may mean that we maximize the log-likelihood

$$\log p(\boldsymbol{y}^{(t)} \quad \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}), \tag{10.29}$$

or, if the model includes connections from the output at one time step to the next time step,

$$\log p(\boldsymbol{y}^{(t)} \quad \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}, \boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(t\ 1)}). \tag{10.30}$$

Decomposing the joint probability over the sequence of $\boldsymbol{y}$ values 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 $\boldsymbol{y}$ values as inputs that condition the next step prediction, the directed graphical model contains no edges from any $\boldsymbol{y}^{(i)}$ in the past to the current $\boldsymbol{y}^{(t)}$. In this case, the outputs $\boldsymbol{y}$ are conditionally independent given the sequence of $\boldsymbol{x}$ values. When we do feed the actual $\boldsymbol{y}$ values (not their prediction, but the actual observed or generated values) back into the network, the directed graphical model contains edges from all $\boldsymbol{y}^{(i)}$ values in the past to the current $y^{(t)}$ value.

As a simple example, let us consider the case where the RNN models only a sequence of scalar random variables $= \mathrm{y}^{(1)}, \dots, \mathrm{y}^{(\tau)}$, with no additional inputs x. The input at time step t is simply the output at time step $t \quad 1$. The RNN then defines a directed graphical model over the y variables. We parametrize the joint distribution of these observations using the chain rule (equation 3.6) for conditional probabilities:

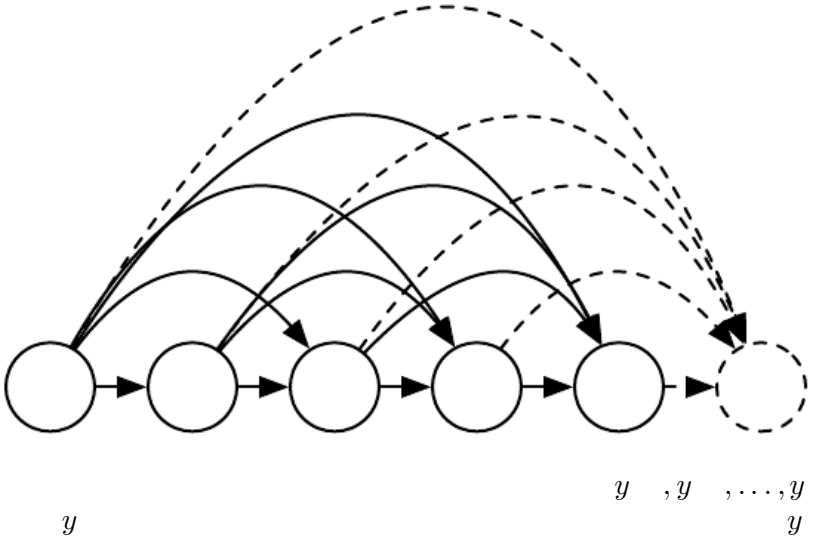
$$P(\quad) = P(\ ^{(1)}, \dots, \ ^{(\tau)}) = \prod_{t=1}^{\tau} P(\ ^{(t)} \quad ^{(t\ 1)}, \ ^{(t\ 2)}, \dots, \ ^{(1)}), \tag{10.31}$$

where the righthand side of the bar is empty for $t = 1$, of course. Hence the negative log-likelihood of a set of values $y^{(1)}, \dots, y^{(\tau)}$ according to such a model is

$$L = \sum_{t} L^{(t)}, \tag{10.32}$$

where

$$L^{(t)} = \quad \log P(\mathrm{y}^{(t)} = y^{(t)} \quad y^{(t\ 1)}, y^{(t\ 2)}, \dots, y^{(1)}). \tag{10.33}$$



$y \quad , y \quad , \ldots , y \quad , \ldots$

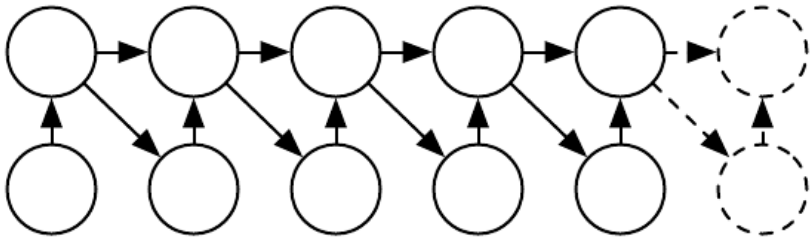
$y \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad y \qquad t > i$

The edges in a graphical model indicate which variables depend directly on other variables. Many graphical models aim to achieve statistical and computational efficiency by omitting edges that do not correspond to strong interactions. For example, it is common to make the Markov assumption that the graphical model should contain only edges from $\mathrm{y}^{(t\ k)}, \ldots, \mathrm{y}^{(t\ 1)}$ to $\mathrm{y}^{(t)}$, rather than containing edges from the entire history. In some cases, however, we believe that all past inputs should have an influence on the next element of the sequence. RNNs are useful when we believe that the distribution over $\mathrm{y}^{(t)}$ may depend on a value of $\mathrm{y}^{(i)}$ from the distant past in a way that is not captured by the effect of $\mathrm{y}^{(i)}$ on $\mathrm{y}^{(t\ 1)}$.

One way to interpret an RNN as a graphical model is to view the RNN as defining a graphical model whose structure is the complete graph, able to represent direct dependencies between any pair of y values. The graphical model over the y values with the complete graph structure is shown in figure 10.7. The complete graph interpretation of the RNN is based on ignoring the hidden units $\boldsymbol{h}^{(t)}$ by marginalizing them out of the model.

It is more interesting to consider the graphical model structure of RNNs that results from regarding the hidden units $\boldsymbol{h}^{(t)}$ as random variables. Including the

The conditional distribution over these variables given their parents is deterministic. This is perfectly legitimate, though it is somewhat rare to design a graphical model with such deterministic hidden units.



$\boldsymbol{h}$

$\boldsymbol{y}$

hidden units in the graphical model reveals that the RNN provides an efficient parametrization of the joint distribution over the observations. Suppose that we represented an arbitrary joint distribution over discrete values with a tabular representation—an array containing a separate entry for each possible assignment of values, with the value of that entry giving the probability of that assignment occurring. If y can take on k different values, the tabular representation would have $O(k^{\tau})$ parameters. By comparison, because of parameter sharing, the number of parameters in the RNN is $O(1)$ as a function of sequence length. The number of parameters in the RNN may be adjusted to control model capacity but is not forced to scale with sequence length. Equation 10.5 shows that the RNN parametrizes long-term relationships between variables efficiently, using recurrent applications of the same function f and the same parameters $\boldsymbol{\theta}$ at each time step. Figure 10.8 illustrates the graphical model interpretation. Incorporating the $\boldsymbol{h}^{(t)}$ nodes in the graphical model decouples the past and the future, acting as an intermediate quantity between them. A variable $y^{(i)}$ in the distant past may influence a variable $y^{(t)}$ via its effect on $\boldsymbol{h}$. The structure of this graph shows that the model can be efficiently parametrized by using the same conditional probability distributions at each time step, and that when the variables are all observed, the probability of the joint assignment of all variables can be evaluated efficiently.

Even with the efficient parametrization of the graphical model, some operations remain computationally challenging. For example, it is difficult to predict missing values in the middle of the sequence.

The price recurrent networks pay for their reduced number of parameters is that *optimizing* the parameters may be difficult.

The parameter sharing used in recurrent networks relies on the assumption that the same parameters can be used for different time steps. Equivalently, the

assumption is that the conditional probability distribution over the variables at time $t+1$ given the variables at time t is **stationary**, meaning that the relationship between the previous time step and the next time step does not depend on t. 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.

To complete our view of an RNN as a graphical model, we must describe how to draw samples from the model. The main operation that we need to perform is simply to sample from the conditional distribution at each time step. However, there is one additional complication. The RNN must have some mechanism for determining the length of the sequence. This can be achieved in various ways.

When the output is a symbol taken from a vocabulary, one can add a special symbol corresponding to the end of a sequence (Schmidhuber, 2012). When that symbol is generated, the sampling process stops. In the training set, we insert this symbol as an extra member of the sequence, immediately after $\boldsymbol{x}^{(\tau)}$ in each training example.

Another option is to introduce an extra Bernoulli output to the model that represents the decision to either continue generation or halt generation at each time step. This approach is more general than the approach of adding an extra symbol to the vocabulary, because it may be applied to any RNN, rather than only RNNs that output a sequence of symbols. For example, it may be applied to an RNN that emits a sequence of real numbers. The new output unit is usually a sigmoid unit trained with the cross-entropy loss. In this approach the sigmoid is trained to maximize the log-probability of the correct prediction as to whether the sequence ends or continues at each time step.

Another way to determine the sequence length τ is to add an extra output to the model that predicts the integer τ itself. The model can sample a value of τ and then sample τ steps worth of data. This approach requires adding an extra input to the recurrent update at each time step so that the recurrent update is aware of whether it is near the end of the generated sequence. This extra input can either consist of the value of τ or can consist of $\tau - t$, the number of remaining time steps. Without this extra input, the RNN might generate sequences that end abruptly, such as a sentence that ends before it is complete. This approach is based on the decomposition

$$P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}) = P(\tau) P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)} \mid \tau). \tag{10.34}$$

The strategy of predicting τ directly is used, for example, by Goodfellow *et al.* (2014d).

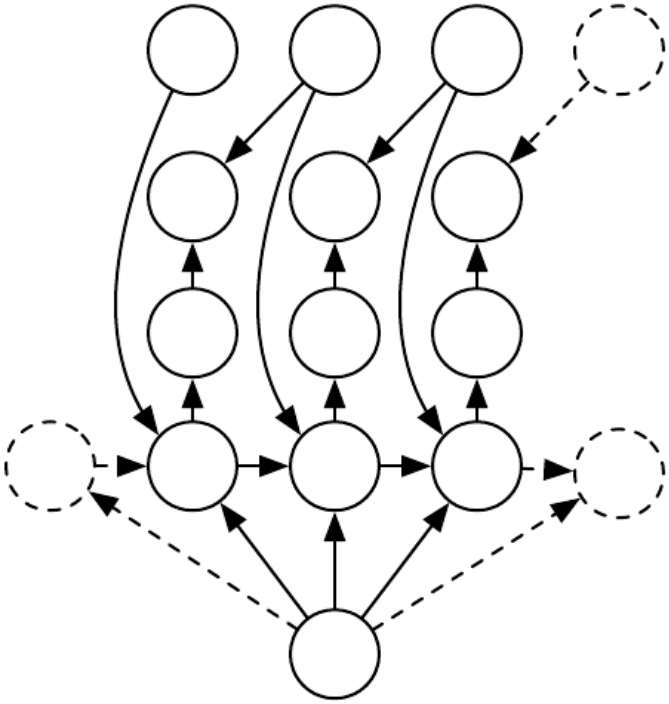
In the previous section we described how an RNN could correspond to a directed graphical model over a sequence of random variables $y^{(t)}$ with no inputs $\boldsymbol{x}$. Of course, our development of RNNs as in equation 10.8 included a sequence of inputs $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(\tau)}$. In general, RNNs allow the extension of the graphical model view to represent not only a joint distribution over the y variables but also a conditional distribution over y given $\boldsymbol{x}$. As discussed in the context of feedforward networks in section 6.2.1.1, any model representing a variable $P(\boldsymbol{y}; \boldsymbol{\theta})$ can be reinterpreted as a model representing a conditional distribution $P(\boldsymbol{y} \mid \boldsymbol{\omega})$ with $\boldsymbol{\omega} = \boldsymbol{\theta}$. We can extend such a model to represent a distribution $P(\boldsymbol{y} \mid \boldsymbol{x})$ by using the same $P(\boldsymbol{y} \mid \boldsymbol{\omega})$ as before, but making $\boldsymbol{\omega}$ a function of $\boldsymbol{x}$. In the case of an RNN, this can be achieved in different ways. We review here the most common and obvious choices.

Previously, we have discussed RNNs that take a sequence of vectors $\boldsymbol{x}^{(t)}$ for $t = 1, \ldots, \tau$ as input. Another option is to take only a single vector $\boldsymbol{x}$ as input. When $\boldsymbol{x}$ is a fixed-size vector, we can simply make it an extra input of the RNN that generates the 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 $\boldsymbol{h}^{(0)}$, or
3. both.

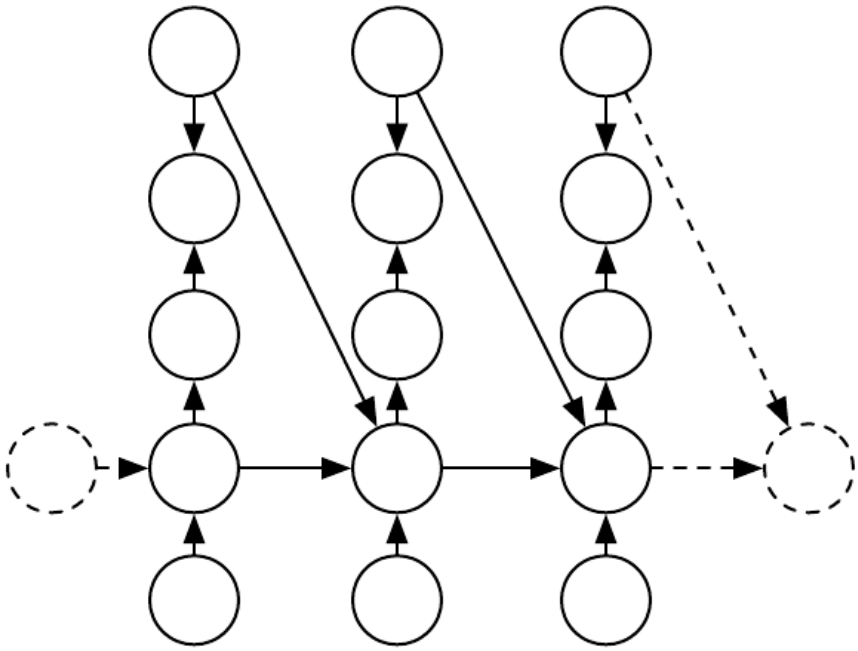
The first and most common approach is illustrated in figure 10.9. The interaction between the input $\boldsymbol{x}$ and each hidden unit vector $\boldsymbol{h}^{(t)}$ is parametrized by a newly introduced weight matrix $\boldsymbol{R}$ that was absent from the model of only the sequence of y values. The same product $\boldsymbol{x}\boldsymbol{R}$ is added as additional input to the hidden units at every time step. We can think of the choice of $\boldsymbol{x}$ as determining the value of $\boldsymbol{x}\boldsymbol{R}$ that is effectively a new bias parameter used for each of the hidden units. The weights remain independent of the input. We can think of this model as taking the parameters $\boldsymbol{\theta}$ of the nonconditional model and turning them into $\boldsymbol{\omega}$, where the bias parameters within $\boldsymbol{\omega}$ are now a function of the input.

Rather than receiving only a single vector $\boldsymbol{x}$ as input, the RNN may receive a sequence of vectors $\boldsymbol{x}^{(t)}$ as input. The RNN described in equation 10.8 corre-



$\boldsymbol{x}$

$\boldsymbol{y}$



$\boldsymbol{x}$ $\boldsymbol{y}$

$\boldsymbol{y}$

$\boldsymbol{x}$

$\boldsymbol{y}$

$\boldsymbol{x}$

sponds to a conditional distribution $P(\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(\tau)} \quad \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)})$ that makes a conditional independence assumption that this distribution factorizes as

$$P(\boldsymbol{y}^{(t)} \quad \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}). \tag{10.35}$$
$$t$$

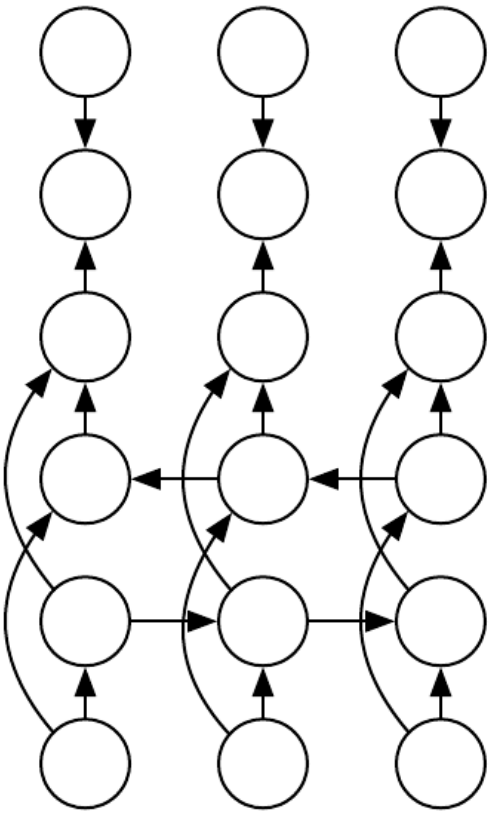
To remove the conditional independence assumption, we can add connections from the output at time t to the hidden unit at time $t+1$, as shown in figure 10.10. The model can then represent arbitrary probability distributions over the $\boldsymbol{y}$ sequence. This kind of model representing a distribution over a sequence given another sequence still has one restriction, which is that the length of both sequences must be the same. We describe how to remove this restriction in section 10.4.

All the recurrent networks we have considered up to now have a "causal" structure, meaning that the state at time t captures only information from the past, $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t\ 1)}$, and the present input $\boldsymbol{x}^{(t)}$. Some of the models we have discussed also allow information from past $\boldsymbol{y}$ values to affect the current state when the $\boldsymbol{y}$ values are available.

In many applications, however, we want to output a prediction of $\boldsymbol{y}^{(t)}$ that 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 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, bidirectional RNNs combine an RNN that moves forward through time, beginning from the start of the sequence, with another RNN that moves backward through time, beginning from the end of the sequence. Figure 10.11 illustrates the typical bidirectional RNN, with $\boldsymbol{h}^{(t)}$ standing for the state of the



$\boldsymbol{x}$ $\boldsymbol{y}$ L t
$\boldsymbol{h}$ $\boldsymbol{g}$
t $\boldsymbol{o}$ $\boldsymbol{h}$
$\boldsymbol{g}$

sub-RNN that moves forward through time and $\boldsymbol{g}^{(t)}$ standing for the state of the sub-RNN that moves backward through time. This allows the output 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).

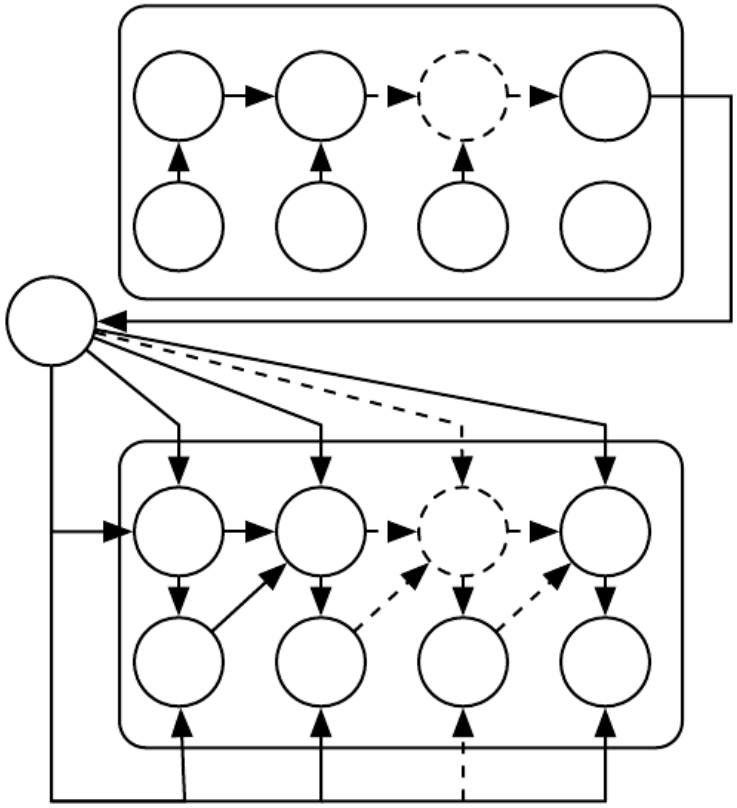
This idea can be naturally extended to two-dimensional input, such as images, by having *four* RNNs, each one going in one of the four directions: up, down, left, right. At each point (i, j) of a 2-D grid, an output $O_{i,j}$ could then compute a representation that would capture mostly local information but could also depend on long-range inputs, if the RNN is able to learn to carry that information. Compared to a convolutional network, RNNs applied to images are typically more expensive but allow for long-range lateral interactions between features in the same feature map (Visin *et al.*, 2015; Kalchbrenner *et al.*, 2015). Indeed, the forward propagation equations for such RNNs may be written in a form that shows they use a convolution that computes the bottom-up input to each layer, prior to the recurrent propagation across the feature map that incorporates the lateral interactions.

We have seen in figure 10.5 how an RNN can map an input sequence to a fixed-size vector. We have seen in figure 10.9 how an RNN can map a fixed-size vector to a sequence. We have seen in figures 10.3, 10.4, 10.10 and 10.11 how an RNN can map an input sequence to an output sequence of the same length.

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 and 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 sequence of vectors that summarize the input sequence $\boldsymbol{X} = (\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n\ )})$.

The simplest RNN architecture for mapping a variable-length sequence to another variable-length sequence was first proposed by Cho *et al.* (2014a) and shortly after by Sutskever *et al.* (2014), who independently developed that archi-



(, . . . ,)

(, , . . . ,)

C

tecture and were the first to obtain state-of-the-art translation using this approach. The former system is based on scoring proposals generated by another machine translation system, while the latter uses a standalone recurrent network to generate the translations. These authors respectively called this architecture, illustrated in figure 10.12, 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 as in figure 10.9) to generate the output sequence $\boldsymbol{Y} = (\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(n\ )})$. The innovation of this kind of architecture over those presented in earlier sections of this chapter is that the lengths n_x and n_y can vary from each other, while previous architectures constrained $n_x = n_y = \tau$. In a sequence-to-sequence architecture, the two RNNs are trained jointly to maximize the average of $\log P(\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(n\ )} \ \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n\ )})$ over all the pairs of $\boldsymbol{x}$ and $\boldsymbol{y}$ sequences in the training set. The last state $\boldsymbol{h}_n$ of the encoder RNN is typically used as a representation C of the input sequence that is provided as input to the decoder RNN.

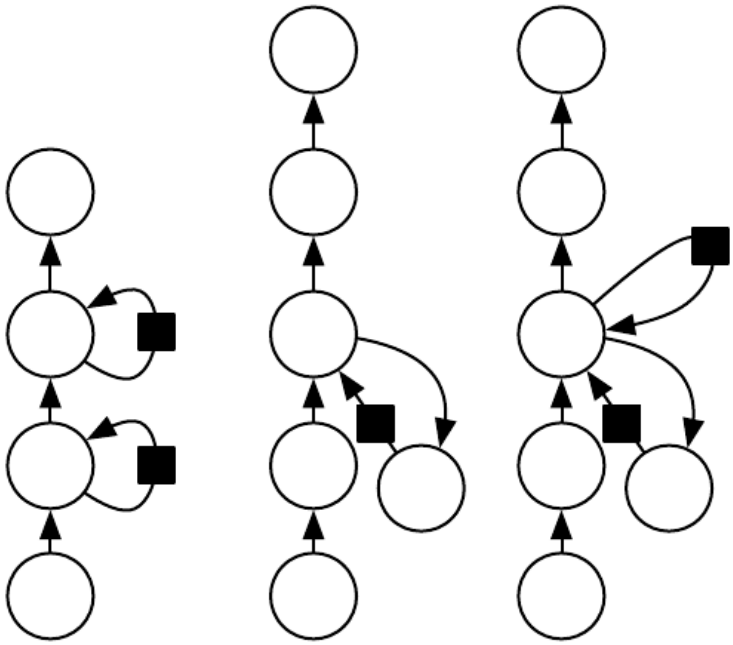
If the context C is a vector, then the decoder RNN is simply a vector-to-sequence RNN, as described in section 10.2.4. As we have seen, there are at least two ways for a vector-to-sequence RNN to receive input. The input can be provided as the initial state of the RNN, or the input can be connected to the hidden units at each time step. These two ways can also be combined.

There is no constraint that the encoder must have the same size of hidden layer as the decoder.

One clear limitation of this architecture is when the context C output by the encoder RNN has a dimension that is too small to properly summarize a long sequence. This phenomenon was observed by Bahdanau *et al.* (2015) 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 section 12.4.5.1 for more details.

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

1. from the input to the hidden state,



2. from the previous hidden state to the next hidden state, and

3. from the hidden state to the output.

With the RNN architecture of figure 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 blocks 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 affine transformation followed by a fixed nonlinearity.

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

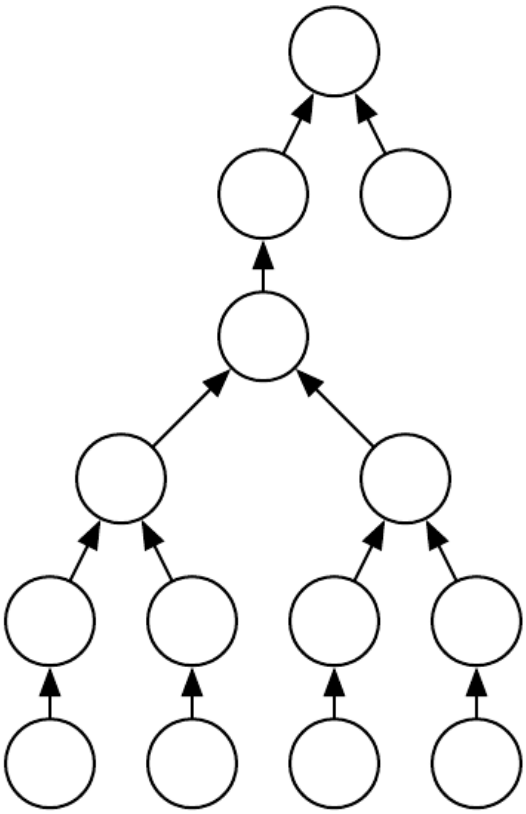
depth to perform the required mappings. See also Schmidhuber (1992), El Hihi and Bengio (1996), or Jaeger (2007a) for earlier work on deep RNNs.

Graves *et al.* (2013) were the first to show a significant benefit of decomposing the state of an RNN into multiple layers, as in figure 10.13 (left). We can think of the lower layers in the hierarchy depicted in figure 10.13a 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 figure 10.13b. Considerations of representational capacity suggest allocating enough capacity in each of these three steps, but doing so by adding depth may hurt learning by making optimization difficult. In general, it is easier to optimize shallower architectures, and adding the extra depth of figure 10.13b makes the shortest path from a variable in time step t to a variable in time step $t+1$ become longer. For example, if an MLP with a single hidden layer is used for the state-to-state transition, we have doubled the length of the shortest path between variables in any two different time steps, compared with the ordinary RNN of figure 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 figure 10.13c.

Recursive neural networks [1] 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 figure 10.14. Recursive neural networks were introduced by Pollack (1990), and their potential use for learning to reason was described by Bottou (2011). Recursive networks have been successfully applied to processing *data structures* as input to neural nets (Frasconi *et al.*, 1997, 1998), in natural language processing (Socher *et al.*, 2011a,c, 2013a), 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 τ, the depth (measured as the number of compositions of nonlinear operations) can be drastically reduced from τ to $O(\log \tau)$, which might help deal with long-term dependencies. An open question is how to best structure the tree. One option is to have a tree structure that does not depend on the data,

[1] We suggest not abbreviating "recursive neural network" as "RNN" to avoid confusion with "recurrent neural network."



$\boldsymbol{x}$, $\boldsymbol{x}$, . . . , $\boldsymbol{x}$

$\boldsymbol{o}$

$\boldsymbol{U}$ $\boldsymbol{V}$ $\boldsymbol{W}$

$\boldsymbol{y}$

such as a balanced binary tree. In some application domains, external methods can suggest the appropriate tree structure. For example, when processing natural language sentences, the tree structure for the recursive network can be fixed to the structure of the parse tree of the sentence provided by a natural language parser (Socher *et al.*, 2011a, 2013a). Ideally, one would like the learner itself to discover and infer the tree structure that is appropriate for any given input, as suggested by Bottou (2011).

Many variants of the recursive net idea are possible. For example, Frasconi *et al.* (1997) and Frasconi *et al.* (1998) associate the data with a tree structure, and associate the inputs and targets with individual nodes 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 nonlinearity). For example, Socher *et al.* (2013a) 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).

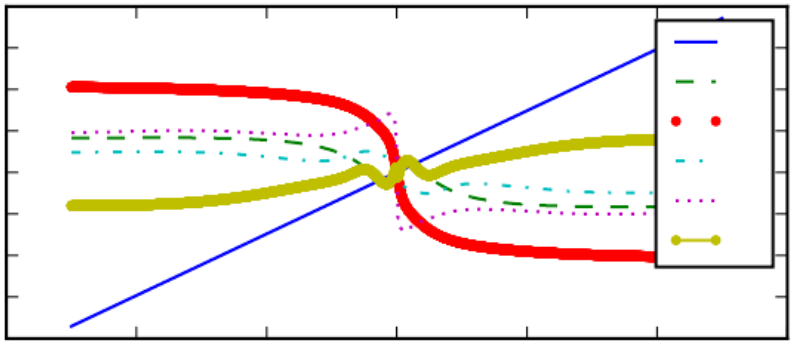
The mathematical challenge of learning long-term dependencies in recurrent networks is introduced in section 8.2.5. The basic problem is that gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization). Even if we assume 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 to short-term ones. Many other sources provide a deeper treatment (Hochreiter, 1991; Doya, 1993; Bengio *et al.*, 1994; Pascanu *et al.*, 2013). In this section, we describe the problem in more detail. The remaining sections describe approaches to overcoming the problem.

Recurrent networks involve the composition of the same function multiple times, once per time step. These compositions can result in extremely nonlinear behavior, as illustrated in figure 10.15.

In particular, the function composition employed by recurrent neural networks somewhat resembles matrix multiplication. We can think of the recurrence relation

$$\boldsymbol{h}^{(t)} = \boldsymbol{W} \quad \boldsymbol{h}^{(t\ 1)} \tag{10.36}$$

as a very simple recurrent neural network lacking a nonlinear activation function,



tanh

y

x

and lacking inputs $\boldsymbol{x}$. As described in section 8.2.5, this recurrence relation essentially describes the power method. It may be simplified to

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}^t \boldsymbol{h}^{(0)}, \tag{10.37}$$

and if $\boldsymbol{W}$ admits an eigendecomposition of the form

$$\boldsymbol{W} = \boldsymbol{Q} \boldsymbol{Q}, \tag{10.38}$$

with orthogonal $\boldsymbol{Q}$, the recurrence may be simplified further to

$$\boldsymbol{h}^{(t)} = \boldsymbol{Q} {}^t\boldsymbol{Q}\boldsymbol{h}^{(0)}. \tag{10.39}$$

The eigenvalues are raised to the power of t, causing eigenvalues with magnitude less than one to decay to zero and eigenvalues with magnitude greater than one to explode. Any component of $\boldsymbol{h}^{(0)}$ that is not aligned with the largest eigenvector will eventually be discarded.

This problem is particular to recurrent networks. In the scalar case, imagine multiplying a weight w by itself many times. The product w^t will either vanish or explode depending on the magnitude of w. If we make a nonrecurrent network that has a different weight $w^{(t)}$ at each time step, the situation is different. If the initial state is given by 1, then the state at time t is given by ${}_t w^{(t)}$. Suppose that the $w^{(t)}$ values are generated randomly, independently from one another, with zero mean and variance v. The variance of the product is $O(v^n)$. To obtain some desired variance v we may choose the individual weights with variance $v = \overline{v}$. Very deep feedforward networks with carefully chosen scaling can thus avoid the vanishing and exploding gradient problem, as argued by Sussillo (2014).

The vanishing and exploding gradient problem for RNNs was independently discovered by separate researchers (Hochreiter, 1991; Bengio *et al.*, 1993, 1994). One may hope that the problem can be avoided simply by staying in a region of parameter space where the gradients do not vanish or explode. Unfortunately, in order to store memories in a way that is robust to small perturbations, the RNN must enter a region of parameter space where gradients vanish (Bengio *et al.*, 1993, 1994). Specifically, whenever the model is able to represent long-term dependencies, the gradient of a long-term interaction has exponentially smaller magnitude than the gradient of a short-term interaction. This means not that it is impossible to learn, but that it might take a very long time to learn long-term dependencies, because the signal about these dependencies will tend to be hidden by the smallest fluctuations arising from short-term dependencies. In practice, the experiments in Bengio *et al.* (1994) show that as we increase the span of the dependencies that

need to be captured, gradient-based optimization becomes increasingly difficult, with the probability of successful training of a traditional RNN via SGD rapidly reaching 0 for sequences of only length 10 or 20.

For a deeper treatment of recurrent networks as dynamical systems, see Doya (1993), Bengio *et al.* (1994), and Siegelmann and Sontag (1995), with a review in Pascanu *et al.* (2013). The remaining sections of this chapter discuss various approaches that have been proposed to reduce the difficulty of learning long-term dependencies (in some cases allowing an RNN to learn dependencies across hundreds of steps), but the problem of learning long-term dependencies remains one of the main challenges in deep learning.

The recurrent weights mapping from $\boldsymbol{h}^{(t\ 1)}$ to $\boldsymbol{h}^{(t)}$ and the input weights mapping from $\boldsymbol{x}^{(t)}$ to $\boldsymbol{h}^{(t)}$ are some of the most difficult parameters to learn in a recurrent network. One proposed (Jaeger, 2003; Maass *et al.*, 2002; Jaeger and Haas, 2004; Jaeger, 2007b) approach to avoiding this difficulty is to set the recurrent 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 a reservoir of temporal features that 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 $\boldsymbol{h}^{(t)}$), on which a linear predictor (typically a linear regression) can be applied to solve the problem of interest. The training criterion may then be easily designed to be convex as a function of the output weights. For example, if the output consists of linear regression from the hidden units to the output targets, and the training criterion is mean squared error, then it is convex and may be solved reliably with simple learning algorithms (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 view the recurrent net as a dynamical system, and set the input and recurrent weights such that the dynamical system is near the edge of stability.

The original idea was to make the eigenvalues of the Jacobian of the state-to-state transition function be close to 1. As explained in section 8.2.5, an important characteristic of a recurrent network is the eigenvalue spectrum of the Jacobians $\boldsymbol{J}^{(t)} = \frac{\partial s}{\partial s}$. Of particular importance is the **spectral radius** of $\boldsymbol{J}^{(t)}$, defined to be the maximum of the absolute values of its eigenvalues.

To understand the effect of the spectral radius, consider the simple case of back-propagation with a 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 backward 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 λ, 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 nonlinearity. When a nonlinearity is present, the derivative of the nonlinearity will approach zero on many time steps and help prevent the explosion resulting from a large spectral radius. Indeed, the most recent work on echo state networks advocates using a spectral radius much larger than unity (Yildiz *et al.*, 2012; Jaeger, 2012).

Everything we have said about back-propagation via repeated matrix multiplication applies equally to forward propagation in a network with no nonlinearity, where the state $\boldsymbol{h}^{(t+1)} = \boldsymbol{h}^{(t)}\ \boldsymbol{W}$.

When a linear map $\boldsymbol{W}$ always shrinks $\boldsymbol{h}$ as measured by the L^2 norm, then

we say that the map is **contractive**. When the spectral radius is less than one, the mapping from $\boldsymbol{h}^{(t)}$ to $\boldsymbol{h}^{(t+1)}$ is contractive, so a small change becomes 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 $\boldsymbol{h}^{(t)}$ propagates one step forward, or equivalently, how the gradient on $\boldsymbol{h}^{(t+1)}$ 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 $\boldsymbol{h}^{(t)}$ or a small variation of $\boldsymbol{h}^{(t)}$ 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 the matrix by the vector. An eigenvalue with magnitude greater than one corresponds to magnification (exponential growth, if applied iteratively) or shrinking (exponential decay, if applied iteratively).

With a nonlinear map, the Jacobian is free to change at each step. The dynamics therefore become more complicated. It remains true, however, that a small initial variation can turn into a large variation after several steps. One difference between the purely linear case and the nonlinear case is that the use of a squashing nonlinearity 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. Nonetheless, it is rare for all the tanh units to simultaneously lie at their linear activation point.

The strategy of echo state networks is simply to fix the weights to have some spectral radius such as 3, where information is carried forward through time but does not explode because of the stabilizing effect of saturating nonlinearities like tanh.

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 (with the hidden-to-hidden recurrent weights trained using back-propagation through time), helping to learn long-term dependencies (Sutskever, 2012; Sutskever *et al.*, 2013). In this setting, an initial spectral radius of 1.2 performs well, combined with the sparse initialization scheme described in section 8.4.

One way to deal with long-term dependencies is to design a model that operates at multiple time scales, so that some parts of the model operate at fine-grained time scales and can handle small details, while other parts operate at coarse time scales and transfer information from the distant past to the present more efficiently. Various strategies for building both fine and coarse time scales are possible. These include the addition of skip connections across time, "leaky units" that integrate signals with different time constants, and the removal of some of the connections used to model fine-grained time scales.

One way to obtain coarse time scales is to add direct connections from variables in the distant past to variables in the present. The idea of using such skip connections dates back to Lin *et al.* (1996) and follows from the idea of incorporating delays in feedforward neural networks (Lang and Hinton, 1988). In an ordinary recurrent network, a recurrent connection goes from a unit at time t to a unit at time $t+1$. It is possible to construct recurrent networks with longer delays (Bengio, 1991).

As we have seen in section 8.2.5, gradients may vanish or explode exponentially *with respect to the number of time steps*. Lin *et al.* (1996) introduced recurrent connections with a time delay of d to mitigate this problem. Gradients now diminish exponentially as a function of $\frac{\tau}{d}$ rather than τ. Since there are both delayed and single step connections, gradients may still explode exponentially in τ. This allows the learning algorithm to capture longer dependencies, although not all long-term dependencies may be represented well in this way.

Another way to obtain paths on which the product of derivatives is close to one is to have units with *linear* self-connections and a weight near one on these connections.

When we accumulate a running average $\mu^{(t)}$ of some value $v^{(t)}$ by applying the update $\mu^{(t)} \quad \alpha\mu^{(t\ 1)} + (1 \quad \alpha)v^{(t)}$, the α parameter is an example of a linear self-connection from $\mu^{(t\ 1)}$ to $\mu^{(t)}$. When α is near one, the running average remembers information about the past for a long time, and when α is near zero, information about the past is rapidly discarded. Hidden units with linear self-connections can behave similarly to such running averages. Such hidden units are called **leaky**

units.

Skip connections through d time steps are a way of ensuring that a unit can always learn to be influenced by a value from d time steps earlier. The use of a linear self-connection with a weight near one is a different way of ensuring that the unit can access values from the past. The linear self-connection approach allows this effect to be adapted more smoothly and flexibly by adjusting the real valued α rather than by adjusting the integer-valued skip length.

These ideas were proposed by Mozer (1992) and by El Hihi and Bengio (1996). Leaky units were also found to be useful in the context of echo state networks (Jaeger *et al.*, 2007).

There are two basic strategies for setting the time constants used by leaky units. One strategy is to manually fix them to values that remain constant, for example, 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.*, 2013).

Another approach to handling long-term dependencies is the 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 idea differs from the skip connections through time discussed earlier because it involves actively *removing* length-one connections and replacing them with longer connections. Units modified in such a way are forced to operate on a long time scale. Skip connections through time *add* edges. Units receiving such new connections may learn to operate on a long time scale but may also choose to focus on their other, short-term connections.

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, 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.* (2013). Another option is to have explicit and discrete updates taking place at different times, with a different frequency for different groups of units. This is the approach of El Hihi and Bengio (1996) and Koutnik *et al.* (2014). It worked well on a number of benchmark datasets.

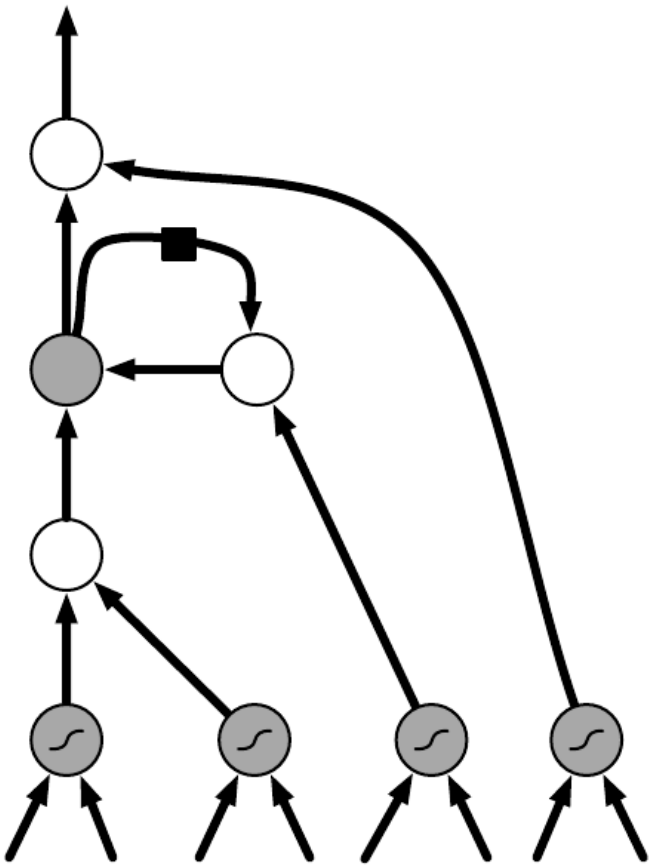
As of this writing, the most effective sequence models used in practical applications are called **gated RNNs**. These include the **long short-term memory** and networks based on the **gated recurrent unit**.

Like leaky units, gated RNNs are based on the idea of creating paths through time that have derivatives that neither vanish nor explode. Leaky units did this with connection weights that were either manually chosen constants or were parameters. Gated RNNs generalize this to connection weights that may change at each time step.

Leaky units allow the network to *accumulate* information (such as evidence for a particular feature or category) over a long duration. Once that information has been used, however, 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. Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it. This is what gated RNNs do.

The clever idea of introducing self-loops to produce paths where the gradient can flow for long durations is a core contribution of the initial **long short-term memory** (LSTM) model (Hochreiter and Schmidhuber, 1997). A crucial addition has been to make the weight on this self-loop conditioned on the context, rather than fixed (Gers *et al.*, 2000). By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically. In this case, we mean that even for an LSTM with fixed parameters, the time scale of integration can change based on the input sequence, because the time constants are output by the model itself. The LSTM has been found extremely successful in many 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.*, 2014), image captioning (Kiros *et al.*, 2014b; Vinyals *et al.*, 2014b; Xu *et al.*, 2015), and parsing (Vinyals *et al.*, 2014a).

The LSTM block diagram is illustrated in figure 10.16. The corresponding forward propagation equations are given below, for a shallow recurrent network



architecture. Deeper architectures have also been successfully used (Graves *et al.*, 2013; Pascanu *et al.*, 2014a). Instead of a unit that simply applies an element-wise nonlinearity 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 also has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit $s_i^{(t)}$, which has a linear self-loop similar to the leaky units described in the previous section. Here, however, the self-loop weight (or the associated time constant) is controlled by a **forget gate** unit $f_i^{(t)}$ (for time step t and cell i), which sets this weight to a value between 0 and 1 via a sigmoid unit:

$$f_i^{(t)} = \sigma \quad b_i^f + \quad _j U_{i,j}^f x_j^{(t)} + \quad _j W_{i,j}^f h_j^{(t \ 1)} \quad , \tag{10.40}$$

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, but with a conditional self-loop weight $f_i^{(t)}$:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t \ 1)} + g_i^{(t)} \sigma \quad b_i + \quad _j U_{i,j} x_j^{(t)} + \quad _j W_{i,j} h_j^{(t \ 1)} \quad , \tag{10.41}$$

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 $g_i^{(t)}$ is computed similarly to the forget gate (with a sigmoid unit to obtain a gating value between 0 and 1), but with its own parameters:

$$g_i^{(t)} = \sigma \quad b_i^g + \quad _j U_{i,j}^g x_j^{(t)} + \quad _j W_{i,j}^g h_j^{(t \ 1)} \quad . \tag{10.42}$$

The output $h_i^{(t)}$ of the LSTM cell can also be shut off, via the **output gate** $q_i^{(t)}$, which also uses a sigmoid unit for gating:

$$h_i^{(t)} = \tanh \quad s_i^{(t)} \quad q_i^{(t)}, \tag{10.43}$$

$$q_i^{(t)} = \sigma \quad b_i^o + \quad _j U_{i,j}^o x_j^{(t)} + \quad _j W_{i,j}^o h_j^{(t \ 1)} \quad , \tag{10.44}$$

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 $s_i^{(t)}$ as an extra input (with its weight) into the three gates of the i-th unit, as shown in figure 10.16. 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 datasets designed for testing the ability to learn long-term dependencies (Bengio *et al.*, 1994; Hochreiter and Schmidhuber, 1997; Hochreiter *et al.*, 2001), then on challenging sequence processing tasks where state-of-the-art performance was obtained (Graves, 2012; Graves *et al.*, 2013; Sutskever *et al.*, 2014). Variants and alternatives to the LSTM that have been studied and used are discussed next.

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, or GRUs (Cho *et al.*, 2014b; Chung *et al.*, 2014, 2015a; Jozefowicz *et al.*, 2015; Chrupala *et al.*, 2015). 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. The update equations are the following:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t-1)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)} \right), \tag{10.45}$$

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

$$u_i^{(t)} = \sigma \left(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)} \right) \tag{10.46}$$

and

$$r_i^{(t)} = \sigma \left(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)} \right). \tag{10.47}$$

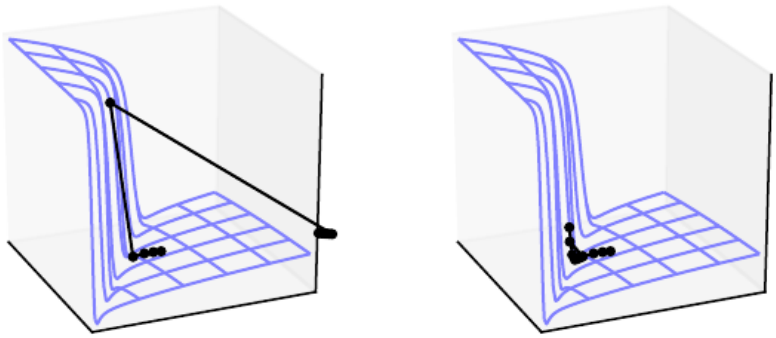
The reset and update gates can individually "ignore" parts of the state vector. The update gates act like conditional leaky integrators that can linearly gate any

dimension, thus choosing to copy it (at one extreme of the sigmoid) or completely ignore it (at the other extreme) by replacing it with the new "target state" value (toward 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 nonlinear 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 multiple hidden units. Alternately, the product of a global gate (covering a whole group of units, such as an entire layer) and a local gate (per unit) could be used to combine global control and local control. Several investigations over architectural variations of the LSTM and GRU, however, found no variant that would clearly beat both of these across a wide range of tasks (Greff *et al.*, 2015; Jozefowicz *et al.*, 2015). Greff *et al.* (2015) found that a crucial ingredient is the forget gate, while Jozefowicz *et al.* (2015) 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.

Section 8.2.5 and section 10.7 have described the vanishing and exploding gradient problems that occur when optimizing RNNs over many time steps.

An interesting idea proposed by Martens and Sutskever (2011) is that second derivatives may vanish at the same time that first derivatives vanish. Second-order optimization algorithms may roughly be understood as dividing the first derivative by the second derivative (in higher dimension, multiplying the gradient by the inverse Hessian). If the second derivative shrinks at a similar rate to the first derivative, then the ratio of first and second derivatives may remain relatively constant. Unfortunately, second-order methods have many drawbacks, including high computational cost, the need for a large minibatch, and a tendency to be attracted to saddle points. Martens and Sutskever (2011) found promising results using second-order methods. Later, Sutskever *et al.* (2013) found that simpler methods such as Nesterov momentum with careful initialization could achieve similar results. See Sutskever (2012) for more detail. Both of these approaches have largely been replaced by simply using SGD (even without momentum) applied to LSTMs. This is part of a continuing theme in machine learning that it is often much easier to design a model that is easy to optimize than it is to design a more powerful optimization algorithm.



As discussed in section 8.2.4, strongly nonlinear 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 figure 8.3 and in figure 10.17, 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 much 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 this infinitesimal region, the cost function may

begin to curve back upward. 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.

A simple type of solution has been in use by practitioners for many years: **clipping the gradient**. There are different instances of this idea (Mikolov, 2012; Pascanu *et al.*, 2013). One option is to clip the parameter gradient from a minibatch *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.*, 2013) just before the parameter update:

$$\text{if } \boldsymbol{g} > v \tag{10.48}$$

$$\boldsymbol{g} \quad \frac{\boldsymbol{g}v}{\boldsymbol{g}}, \tag{10.49}$$

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 of guaranteeing 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 or (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. Taking the average of the norm-clipped gradient from many minibatches, however, 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.

Gradient clipping helps to deal with exploding gradients, but 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 in section 10.10. Another idea is to regularize or constrain the parameters so as to encourage "information flow." In particular, we would like the gradient vector $\quad L$ being back-propagated to maintain its magnitude, even if the loss function only penalizes the output at the end of the sequence. Formally, we want

$$(\quad L) \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t \ 1)}} \tag{10.50}$$

to be as large as

$$L. \tag{10.51}$$

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

$$\Omega = \quad _t \quad \frac{(\quad L) \frac{\partial}{\partial}}{L} \quad 1 \quad ^2 . \tag{10.52}$$

Computing the gradient of this regularizer may appear difficult, but Pascanu *et al.* (2013) propose an approximation in which we consider the back-propagated vectors $\quad L$ as if they were constants (for the purpose of this regularizer, so that there is no need to back-propagate through them). The experiments with this regularizer suggest that, if combined with the norm clipping heuristic (which handles gradient explosion), the regularizer 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. Without gradient clipping, gradient explosion prevents learning from succeeding.

A key weakness of this approach is that it is not as effective as the LSTM for tasks where data is abundant, such as language modeling.

Intelligence requires knowledge, and acquiring knowledge can be done via learning, which has motivated the development of large-scale deep architectures. However, 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—everyday 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, but they struggle to memorize facts. Stochastic gradient descent requires many presentations of the same input before it can be stored in neural network 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 enables 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 need for neural networks that can process information in a sequence of steps, changing the way the input is fed into the network at each step, has long been recognized as important for the ability to reason rather than to make automatic, intuitive responses to the input (Hinton, 1990).

To resolve this difficulty, Weston *et al.* (2014) introduced **memory networks** that include a set of memory cells that can be accessed via an addressing mechanism. Memory networks originally required a supervision signal instructing them how to use their memory cells. Graves *et al.* (2014b) introduced the **neural Turing machine**, which is able to learn to read from and write arbitrary content to memory cells without explicit supervision about which actions to undertake, and allowed end-to-end training without this supervision signal, via the use of a content-based soft attention mechanism (see Bahdanau *et al.*[2015] and section 12.4.5.1). This soft addressing mechanism has become standard with other related architectures, emulating algorithmic mechanisms in a way that still allows gradient-based optimization (Sukhbaatar *et al.*, 2015; Joulin and Mikolov, 2015; Kumar *et al.*, 2015; Vinyals *et al.*, 2015a; Grefenstette *et al.*, 2015).

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

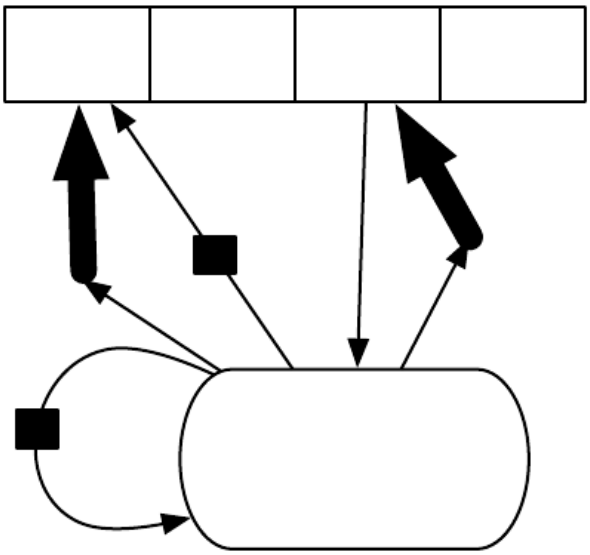
digital computer read from or write to a specific address.

It is difficult to optimize functions that produce exact integer addresses. To alleviate this problem, 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 nonzero derivatives enables 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 its elements. This is analogous to how 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 in 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 memory cell 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.

The explicit memory approach is illustrated in figure 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 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 that information and gradients can be propagated (forward in time or backward in time, respectively) for very long durations.

As an alternative to back-propagation through weighted averages of memory cells, we can interpret the memory addressing coefficients as probabilities and stochastically read just one cell (Zaremba and Sutskever, 2015). Optimizing models that make discrete decisions requires specialized optimization algorithms, described in section 20.9.1. So far, training these stochastic architectures that make discrete 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.*, 2015) and is discussed in section 12.4.5.1. The idea of attention mechanisms for neural networks was introduced even earlier, in the context of handwriting generation (Graves, 2013), with an attention mechanism that was constrained to move only forward in time through the 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.

Recurrent neural networks provide a way to extend deep learning to sequential data. They are the last major tool in our deep learning toolbox. Our discussion now moves to how to choose and use these tools and how to apply them to real-world tasks.