

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.

10.2 Recurrent Neural Networks

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

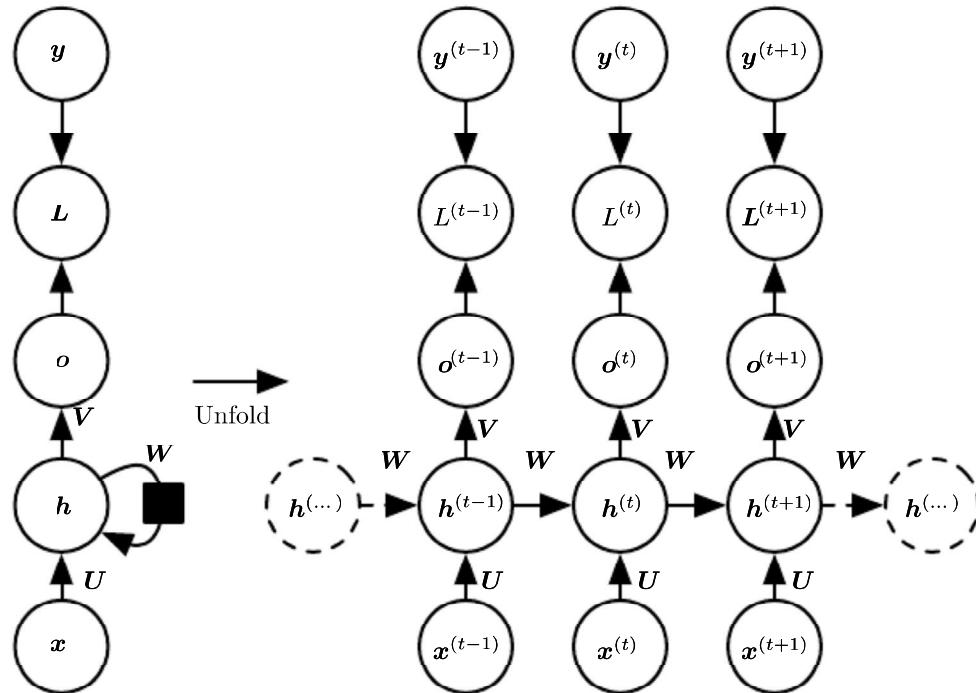


Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of \mathbf{x} values to a corresponding sequence of output \mathbf{o} values. A loss L measures how far each \mathbf{o} is from the corresponding training target \mathbf{y} . When using softmax outputs, we assume \mathbf{o} is the unnormalized log probabilities. The loss L internally computes $\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$ and compares this to the target \mathbf{y} . The RNN has input to hidden connections parametrized by a weight matrix \mathbf{U} , hidden-to-hidden recurrent connections parametrized by a weight matrix \mathbf{W} , and hidden-to-output connections parametrized by a weight matrix \mathbf{V} . Equation 10.8 defines forward propagation in this model. (*Left*)The RNN and its loss drawn with recurrent connections. (*Right*)The same seen as an time-unfolded computational graph, where each node is now associated with one particular time instance.

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; Hytyniemi, 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 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 \mathbf{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 $\hat{\mathbf{y}}$ of normalized probabilities over the output. Forward propagation begins with a specification of the initial state $\mathbf{h}^{(0)}$. Then, for each time step from

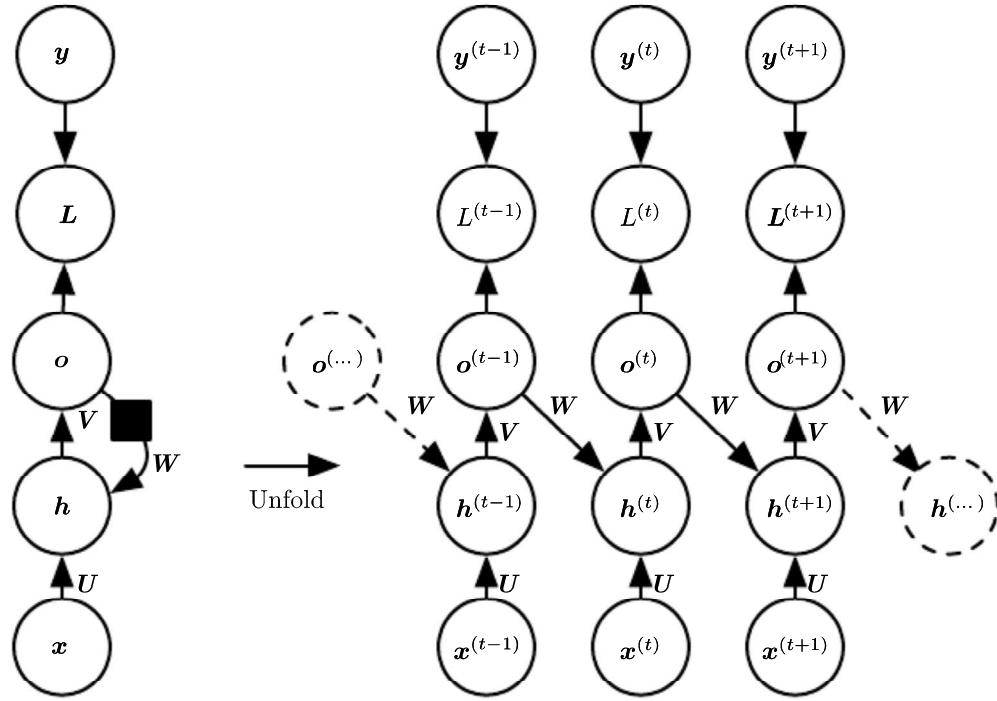


Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time step t , the input is \mathbf{x}_t , the hidden layer activations are $\mathbf{h}^{(t)}$, the outputs are $\mathbf{o}^{(t)}$, the targets are $\mathbf{y}^{(t)}$ and the loss is $L^{(t)}$. (Left) Circuit diagram. (Right) Unfolded computational graph. Such an RNN is less powerful (can express a smaller set of functions) than those in the family represented by figure 10.3. The RNN in figure 10.3 can choose to put any information it wants about the past into its hidden representation \mathbf{h} and transmit \mathbf{h} to the future. The RNN in this figure is trained to put a specific output value into \mathbf{o} , and \mathbf{o} is the only information it is allowed to send to the future. There are no direct connections from \mathbf{h} going forward. The previous \mathbf{h} is connected to the present only indirectly, via the predictions it was used to produce. Unless \mathbf{o} is very high-dimensional and rich, it will usually lack important information from the past. This makes the RNN in this figure less powerful, but it may be easier to train because each time step can be trained in isolation from the others, allowing greater parallelization during training, as described in section 10.2.1.

$t = 1$ to $t = \tau$, we apply the following update equations:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} \quad (10.8)$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)}) \quad (10.9)$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)} \quad (10.10)$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)}) \quad (10.11)$$

where the parameters are the bias vectors \mathbf{b} and \mathbf{c} along with the weight matrices \mathbf{U} , \mathbf{V} and \mathbf{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 \mathbf{x} values paired with a sequence of \mathbf{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 $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}$, then

$$L\left(\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}\}, \{\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau)}\}\right) \quad (10.12)$$

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

$$= - \sum_t \log p_{\text{model}}\left(y^{(t)} \mid \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\}\right), \quad (10.14)$$

where $p_{\text{model}}\left(y^{(t)} \mid \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\}\right)$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{\mathbf{y}}^{(t)}$. 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 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 only be computed 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** or 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?

10.2.1 Teacher Forcing and Networks with Output Recurrence

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 of the information about the past that the network will use to predict the future. Because the output units are 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.

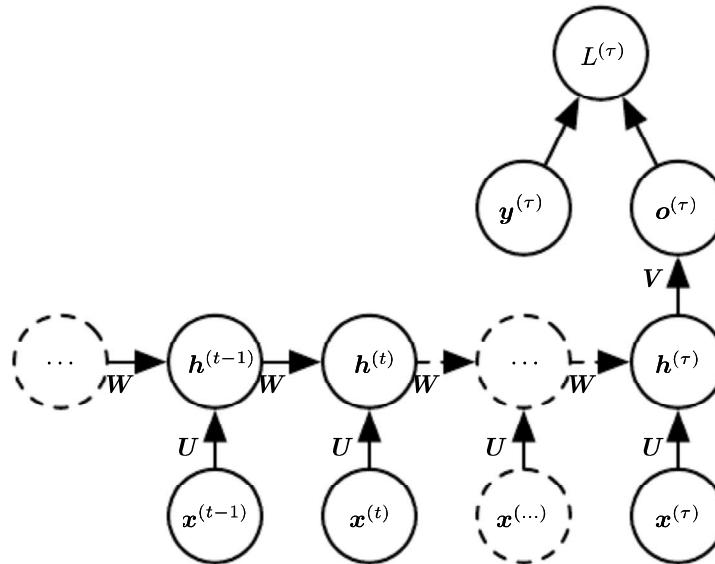


Figure 10.5: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here) or the gradient on the output $\mathbf{o}^{(\tau)}$ can be obtained by back-propagating from further downstream modules.

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

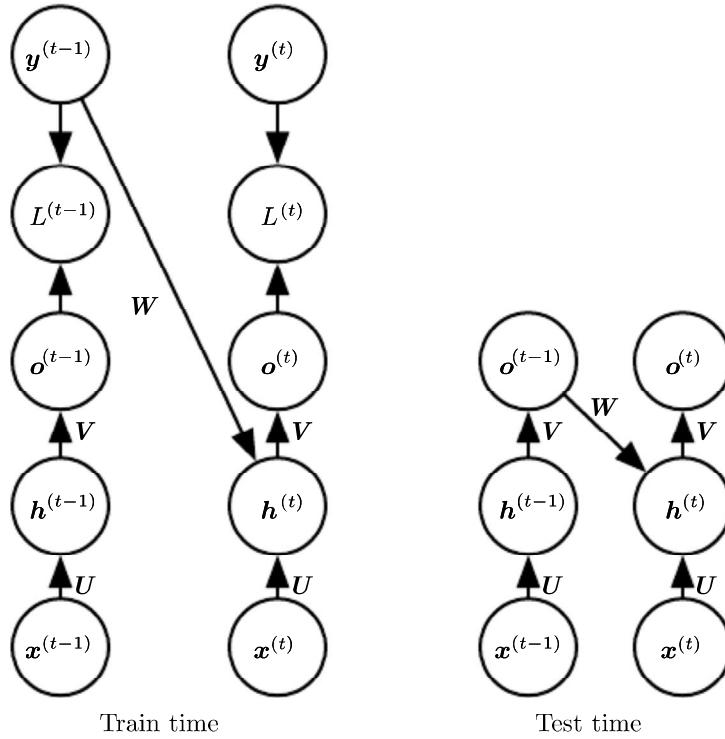


Figure 10.6: Illustration of teacher forcing. Teacher forcing is a training technique that is applicable to RNNs that have connections from their output to their hidden states at the next time step. (*Left*) At train time, we feed the *correct output* $\mathbf{y}^{(t)}$ drawn from the train set as input to $\mathbf{h}^{(t+1)}$. (*Right*) When the model is deployed, the true output is generally not known. In this case, we approximate the correct output $\mathbf{y}^{(t)}$ with the model’s output $\mathbf{o}^{(t)}$, and feed the output back into the model.

likelihood criterion is

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

$$= \log p(\mathbf{y}^{(2)} | \mathbf{y}^{(1)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) + \log p(\mathbf{y}^{(1)} | \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) \quad (10.16)$$

In this example, we see that at time $t = 2$, the model is trained to maximize the conditional probability of $\mathbf{y}^{(2)}$ given *both* the \mathbf{x} sequence so far and the previous \mathbf{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 so long as they have connections from the output at one time step to values computed in the next time step. However, as soon as the hidden units become a function of earlier time steps, 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 with 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 towards 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 train 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.

10.2.2 Computing the Gradient in a Recurrent Neural Network

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 \mathbf{U} , \mathbf{V} , \mathbf{W} , \mathbf{b} and \mathbf{c} as well as the sequence of nodes indexed by t for $\mathbf{x}^{(t)}$, $\mathbf{h}^{(t)}$, $\mathbf{o}^{(t)}$ and $L^{(t)}$. For each node \mathbf{N} we need to compute the gradient $\nabla_{\mathbf{N}} L$ 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

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

In this derivation we assume that the outputs $\mathbf{o}^{(t)}$ are used as the argument to the softmax function to obtain the vector $\hat{\mathbf{y}}$ 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 $\nabla_{\mathbf{o}^{(t)}} L$ on the outputs at time step t , for all i, t , is as follows:

$$(\nabla_{\mathbf{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}. \quad (10.18)$$

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

$$\nabla_{\mathbf{h}^{(\tau)}} L = \mathbf{V}^\top \nabla_{\mathbf{o}^{(\tau)}} L. \quad (10.19)$$

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

$$\nabla_{\mathbf{h}^{(t)}} L = \left(\frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \right)^\top (\nabla_{\mathbf{h}^{(t+1)}} L) + \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t)}} \right)^\top (\nabla_{\mathbf{o}^{(t)}} L) \quad (10.20)$$

$$= \mathbf{W}^\top (\nabla_{\mathbf{h}^{(t+1)}} L) \text{diag} \left(1 - (\mathbf{h}^{(t+1)})^2 \right) + \mathbf{V}^\top (\nabla_{\mathbf{o}^{(t)}} L) \quad (10.21)$$

where $\text{diag} \left(1 - (\mathbf{h}^{(t+1)})^2 \right)$ indicates the diagonal matrix containing the elements $1 - (h_i^{(t+1)})^2$. 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 `bprop` method of section 6.5.6, that computes the contribution of a single edge in the computational graph to the gradient. However, the $\nabla_{\mathbf{W}f}$ operator used in calculus takes into account the contribution of \mathbf{W} to the value of f due to *all* edges in the computational graph. To resolve this ambiguity, we introduce dummy variables $\mathbf{W}^{(t)}$ that are defined to be copies of \mathbf{W} but with each $\mathbf{W}^{(t)}$ used only at time step t . We may then use $\nabla_{\mathbf{W}^{(t)}}$ 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:

$$\nabla_{\mathbf{c}} L = \sum_t \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{c}} \right)^{\top} \nabla_{\mathbf{o}^{(t)}} L = \sum_t \nabla_{\mathbf{o}^{(t)}} L \quad (10.22)$$

$$\nabla_{\mathbf{b}} L = \sum_t \left(\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{b}^{(t)}} \right)^{\top} \nabla_{\mathbf{h}^{(t)}} L = \sum_t \text{diag} \left(1 - (\mathbf{h}^{(t)})^2 \right) \nabla_{\mathbf{h}^{(t)}} L \quad (10.23)$$

$$\nabla_{\mathbf{v}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}} \right) \nabla_{\mathbf{v} o_i^{(t)}} = \sum_t (\nabla_{\mathbf{o}^{(t)}} L) \mathbf{h}^{(t)}{}^{\top} \quad (10.24)$$

$$\nabla_{\mathbf{w}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\mathbf{w}^{(t)}} h_i^{(t)} \quad (10.25)$$

$$= \sum_t \text{diag} \left(1 - (\mathbf{h}^{(t)})^2 \right) (\nabla_{\mathbf{h}^{(t)}} L) \mathbf{h}^{(t-1)}{}^{\top} \quad (10.26)$$

$$\nabla_{\mathbf{u}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\mathbf{u}^{(t)}} h_i^{(t)} \quad (10.27)$$

$$= \sum_t \text{diag} \left(1 - (\mathbf{h}^{(t)})^2 \right) (\nabla_{\mathbf{h}^{(t)}} L) \mathbf{x}^{(t)}{}^{\top} \quad (10.28)$$

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

10.2.3 Recurrent Networks as Directed Graphical Models

In the example recurrent network we have developed so far, the losses $L^{(t)}$ were cross-entropies between training targets $\mathbf{y}^{(t)}$ and outputs $\mathbf{o}^{(t)}$. 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 $\mathbf{y}^{(t)}$ given the past inputs. This may mean that we maximize the log-likelihood

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

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

$$\log p(\mathbf{y}^{(t)} \mid \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}, \mathbf{y}^{(1)}, \dots, \mathbf{y}^{(t-1)}). \quad (10.30)$$

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

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

$$P(\mathbb{Y}) = P(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau)}) = \prod_{t=1}^{\tau} P(\mathbf{y}^{(t)} \mid \mathbf{y}^{(t-1)}, \mathbf{y}^{(t-2)}, \dots, \mathbf{y}^{(1)}) \quad (10.31)$$

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

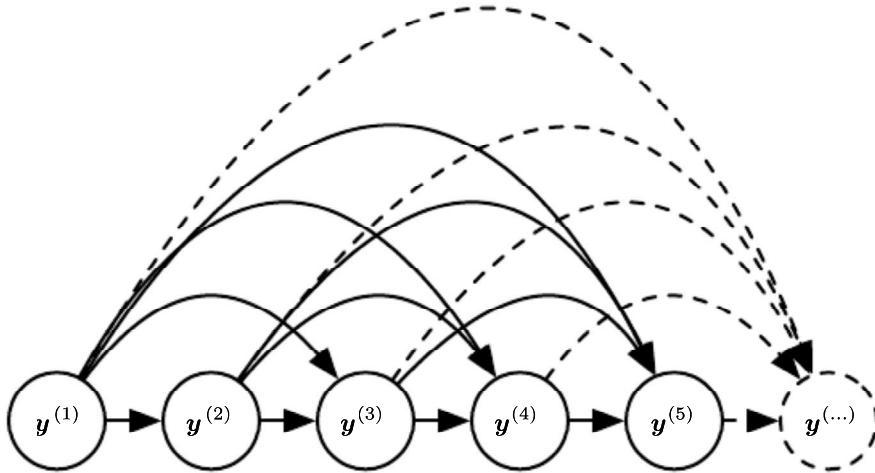


Figure 10.7: Fully connected graphical model for a sequence $y^{(1)}, y^{(2)}, \dots, y^{(t)}, \dots$: every past observation $y^{(i)}$ may influence the conditional distribution of some $y^{(t)}$ (for $t > i$), given the previous values. Parametrizing the graphical model directly according to this graph (as in equation 10.6) might be very inefficient, with an ever growing number of inputs and parameters for each element of the sequence. RNNs obtain the same full connectivity but efficient parametrization, as illustrated in figure 10.8.

is

$$L = \sum_t L^{(t)} \quad (10.32)$$

where

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

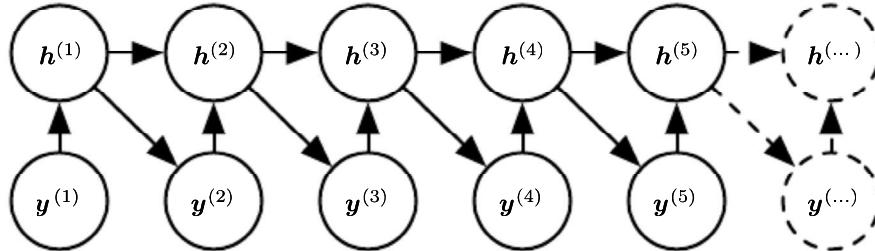


Figure 10.8: Introducing the state variable in the graphical model of the RNN, even though it is a deterministic function of its inputs, helps to see how we can obtain a very efficient parametrization, based on equation 10.5. Every stage in the sequence (for $h^{(t)}$ and $y^{(t)}$) involves the same structure (the same number of inputs for each node) and can share the same parameters with the other stages.

The 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 only contain edges from $\{y^{(t-k)}, \dots, y^{(t-1)}\}$ to $y^{(t)}$, rather than containing edges from the entire past history. However, in some cases, 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 $y^{(t)}$ may depend on a value of $y^{(i)}$ from the distant past in a way that is not captured by the effect of $y^{(i)}$ on $y^{(t-1)}$.

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 $\mathbf{h}^{(t)}$ 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 $\mathbf{h}^{(t)}$ as random variables.¹ Including the hidden units in the graphical model reveals that the RNN provides a very 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 $O(k^T)$ parameters. By comparison, due to 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 same parameters θ at each time step. Figure 10.8 illustrates the graphical model interpretation. Incorporating the $\mathbf{h}^{(t)}$ 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 \mathbf{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

¹The 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.

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.

In the case when the output is a symbol taken from a vocabulary, one can add a special symbol corresponding to the end of a sequence (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 $\mathbf{x}^{(\tau)}$ 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

$$P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}) = P(\tau)P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)} | \tau). \quad (10.34)$$

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

10.2.4 Modeling Sequences Conditioned on Context with RNNs

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 \mathbf{x} . Of course, our development of RNNs as in equation 10.8 included a sequence of inputs $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(\tau)}$. 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 \mathbf{x} . As discussed in the context of feedforward networks in section 6.2.1.1, any model representing a variable $P(\mathbf{y}; \boldsymbol{\theta})$ can be reinterpreted as a model representing a conditional distribution $P(\mathbf{y}|\boldsymbol{\omega})$ with $\boldsymbol{\omega} = \boldsymbol{\theta}$. We can extend such a model to represent a distribution $P(\mathbf{y} | \mathbf{x})$ by using the same $P(\mathbf{y} | \boldsymbol{\omega})$ as before, but making $\boldsymbol{\omega}$ a function of \mathbf{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 $\mathbf{x}^{(t)}$ for $t = 1, \dots, \tau$ as input. Another option is to take only a single vector \mathbf{x} as input. When \mathbf{x} is a fixed-size vector, we can simply make it an extra input of the RNN that generates the \mathbf{y} sequence. Some common ways of providing an extra input to an RNN are:

1. as an extra input at each time step, or
2. as the initial state $\mathbf{h}^{(0)}$, or
3. both.

The first and most common approach is illustrated in figure 10.9. The interaction between the input \mathbf{x} and each hidden unit vector $\mathbf{h}^{(t)}$ is parametrized by a newly introduced weight matrix \mathbf{R} that was absent from the model of only the sequence of y values. The same product $\mathbf{x}^\top \mathbf{R}$ is added as additional input to the hidden units at every time step. We can think of the choice of \mathbf{x} as determining the value

of $\mathbf{x}^\top \mathbf{R}$ 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 non-conditional model and turning them into $\boldsymbol{\omega}$, where the bias parameters within $\boldsymbol{\omega}$ are now a function of the input.

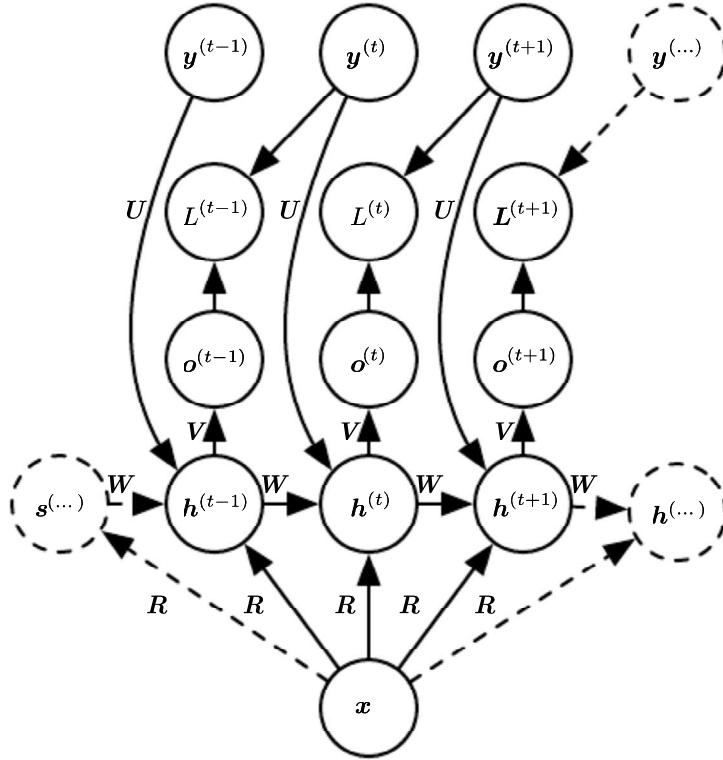


Figure 10.9: An RNN that maps a fixed-length vector \mathbf{x} into a distribution over sequences \mathbf{Y} . This RNN is appropriate for tasks such as image captioning, where a single image is used as input to a model that then produces a sequence of words describing the image. Each element $\mathbf{y}^{(t)}$ of the observed output sequence serves both as input (for the current time step) and, during training, as target (for the previous time step).

Rather than receiving only a single vector \mathbf{x} as input, the RNN may receive a sequence of vectors $\mathbf{x}^{(t)}$ as input. The RNN described in equation 10.8 corresponds to a conditional distribution $P(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)})$ that makes a conditional independence assumption that this distribution factorizes as

$$\prod_t P(\mathbf{y}^{(t)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}). \quad (10.35)$$

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 \mathbf{y} sequence. This kind of model representing a distribution over a sequence given another

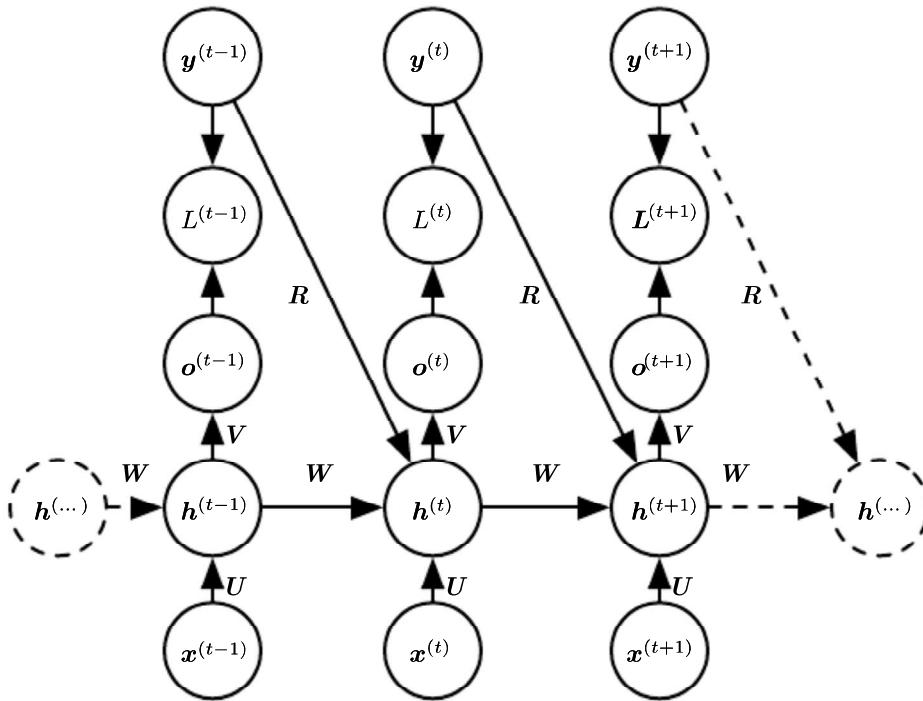


Figure 10.10: A conditional recurrent neural network mapping a variable-length sequence of \mathbf{x} values into a distribution over sequences of \mathbf{y} values of the same length. Compared to figure 10.3, this RNN contains connections from the previous output to the current state. These connections allow this RNN to model an arbitrary distribution over sequences of \mathbf{y} given sequences of \mathbf{x} of the same length. The RNN of figure 10.3 is only able to represent distributions in which the \mathbf{y} values are conditionally independent from each other given the \mathbf{x} values.

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.

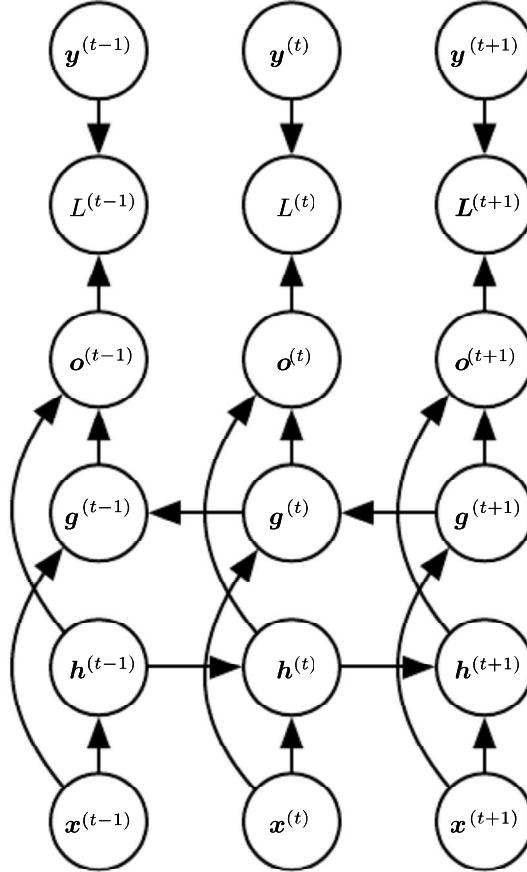


Figure 10.11: Computation of a typical bidirectional recurrent neural network, meant to learn to map input sequences \mathbf{x} to target sequences \mathbf{y} , with loss $L^{(t)}$ at each step t . The \mathbf{h} recurrence propagates information forward in time (towards the right) while the \mathbf{g} recurrence propagates information backward in time (towards the left). Thus at each point t , the output units $\mathbf{o}^{(t)}$ can benefit from a relevant summary of the past in its $\mathbf{h}^{(t)}$ input and from a relevant summary of the future in its $\mathbf{g}^{(t)}$ input.

10.3 Bidirectional RNNs

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

However, in many applications we want to output a prediction of $\mathbf{y}^{(t)}$ which may

depend on *the whole input sequence*. For example, in speech recognition, the correct interpretation of the current sound as a phoneme may depend on the next few phonemes because of co-articulation and potentially may even depend on the next few words because of the linguistic dependencies between nearby words: if there are two interpretations of the current word that are both acoustically plausible, we may have to look far into the future (and the past) to disambiguate them. This is also true of handwriting recognition and many other sequence-to-sequence learning tasks, described in the next section.

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

As the name suggests, 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 $\mathbf{h}^{(t)}$ standing for the state of the sub-RNN that moves forward through time and $\mathbf{g}^{(t)}$ standing for the state of the sub-RNN that moves backward through time. This allows the output units $\mathbf{o}^{(t)}$ to compute a representation that depends on *both the past and the future* but is most sensitive to the input values around time t , without having to specify a fixed-size window around t (as one would have to do with a feedforward network, a convolutional network, or a regular RNN with a fixed-size look-ahead buffer).

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