MODULE 5

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 $\mathsf{\mathbf{X}}$ such as an image, a recurrent neural network is a neural network that is specialized for processing a sequence of values $\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)}$. 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 multi-layer 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 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 of 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 backwards 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).

10.1 Unfolding Computational Graphs

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 $t-1$.

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

$$\boldsymbol{s}^{(3)} = f(\boldsymbol{s}^{(2)}; \boldsymbol{\theta}) \tag{10.2}$$

$$= f(f(\boldsymbol{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta}) \tag{10.3}$$

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.

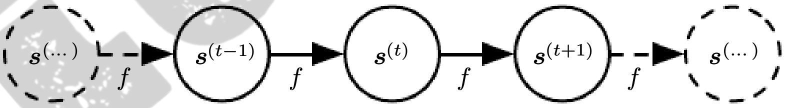

Figure 10.1: The classical dynamical system described by equation 10.1, illustrated as an unfolded computational graph. Each node represents the state at some time t and the function f maps the state at t to the state at $t+1$. The same parameters (the same value of $\boldsymbol{\theta}$ used to parametrize f) are used for all time steps.

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

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

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, it may not be necessary to store all of the information in the input sequence up to time t, but rather only enough information to predict the rest of the sentence. 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).

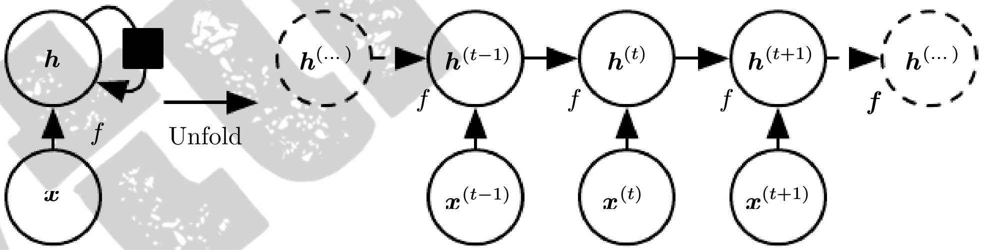

Figure 10.2: A recurrent network with no outputs. This recurrent network just processes information from the input $\boldsymbol{x}$ by incorporating it into the state $\boldsymbol{h}$ that is passed forward through time. *(Left)*Circuit diagram. The black square indicates a delay of a single time step. *(Right)*The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance.

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 $g^{(t)}$:

$$\begin{aligned}\boldsymbol{h}^{(t)} =& g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) && (10.6)\\ =& f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) && (10.7)\end{aligned}$$

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 allows 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 to 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.

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.

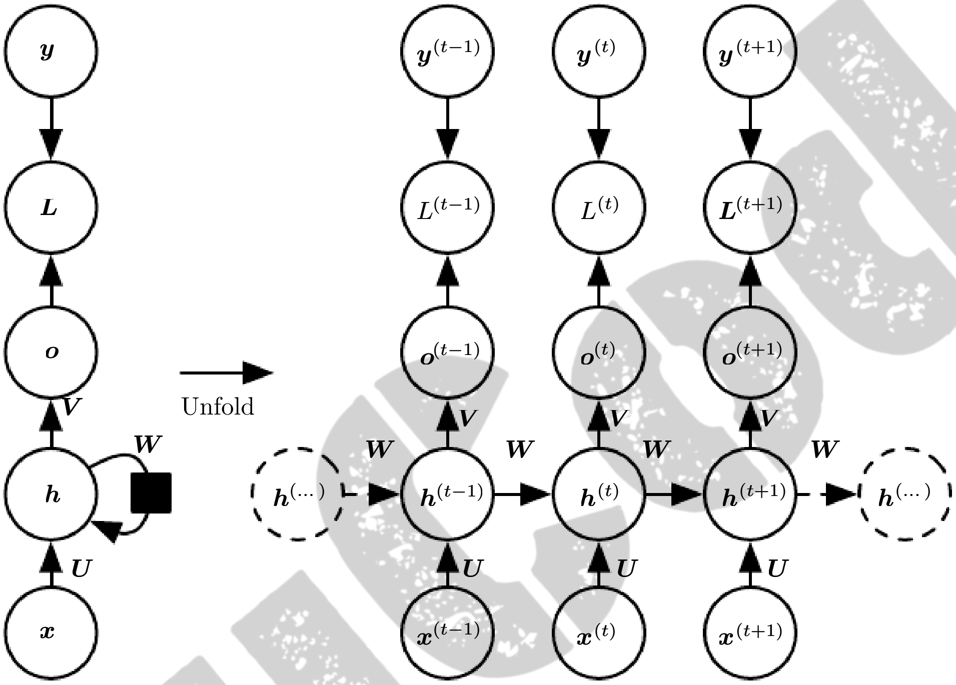

Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of $\boldsymbol{x}$ values to a corresponding sequence of output $\boldsymbol{o}$ values. A loss L measures how far each $\boldsymbol{o}$ is from the corresponding training target $\boldsymbol{y}$. When using softmax outputs, we assume $\boldsymbol{o}$ is the unnormalized log probabilities. The loss L internally computes $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o})$ and compares this to the target $\boldsymbol{y}$. The RNN has input to hidden connections parametrized by a weight matrix $\boldsymbol{U}$, hidden-to-hidden recurrent connections parametrized by a weight matrix $\boldsymbol{W}$, and hidden-to-output connections parametrized by a weight matrix $\boldsymbol{V}$. 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; 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 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

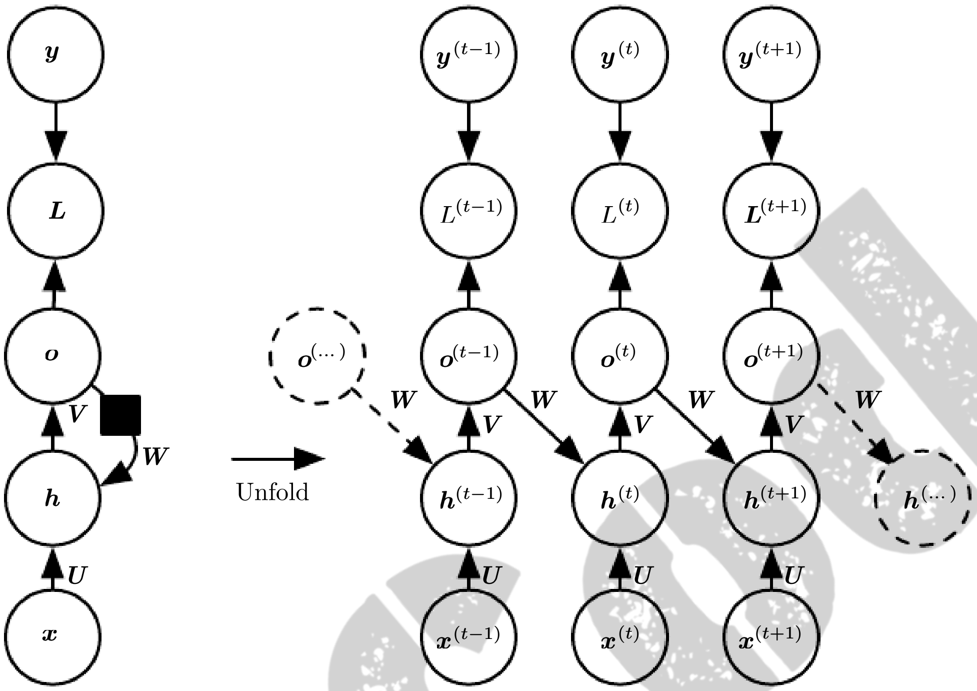

Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time step t, the input is $\boldsymbol{x}_t$, the hidden layer activations are $\boldsymbol{h}^{(t)}$, the outputs are $\boldsymbol{o}^{(t)}$, the targets are $\boldsymbol{y}^{(t)}$ and the 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 $\boldsymbol{h}$ and transmit $\boldsymbol{h}$ to the future. The RNN in this figure is trained to put a specific output value into $\boldsymbol{o}$, and $\boldsymbol{o}$ is the only information it is allowed to send to the future. There are no direct connections from $\boldsymbol{h}$ going forward. The previous $\boldsymbol{h}$ is connected to the present only indirectly, via the predictions it was used to produce. Unless $\boldsymbol{o}$ 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:

$$\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} && (10.8)\\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) && (10.9)\\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} && (10.10)\\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)}) && (10.11)
\end{aligned}$$

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 $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}$, then

$$\begin{aligned}
& L\left(\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}\}, \{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(\tau)}\}\right) && (10.12)\\
&= \sum_t L^{(t)} && (10.13)\\
&= -\sum_t \log p_{\text{model}}\left(y^{(t)} \mid \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}\right), && (10.14)
\end{aligned}$$

where $p_{\text{model}}\left(y^{(t)} \mid \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}\right)$ 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 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.

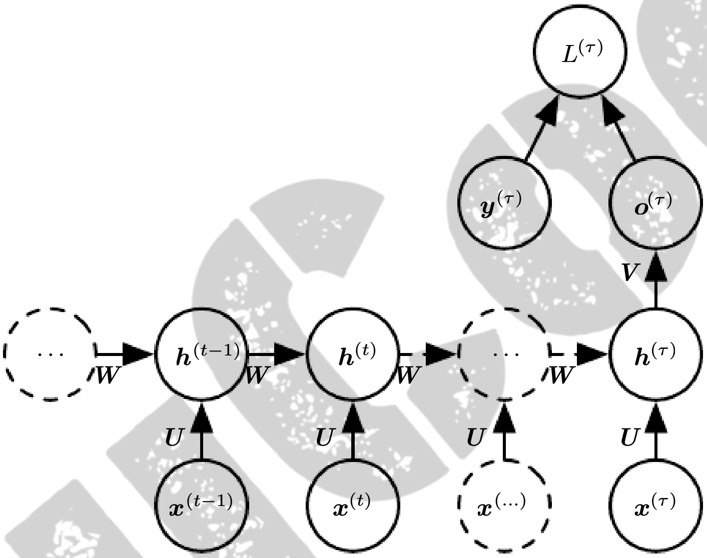

Figure 10.5: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here) or the gradient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

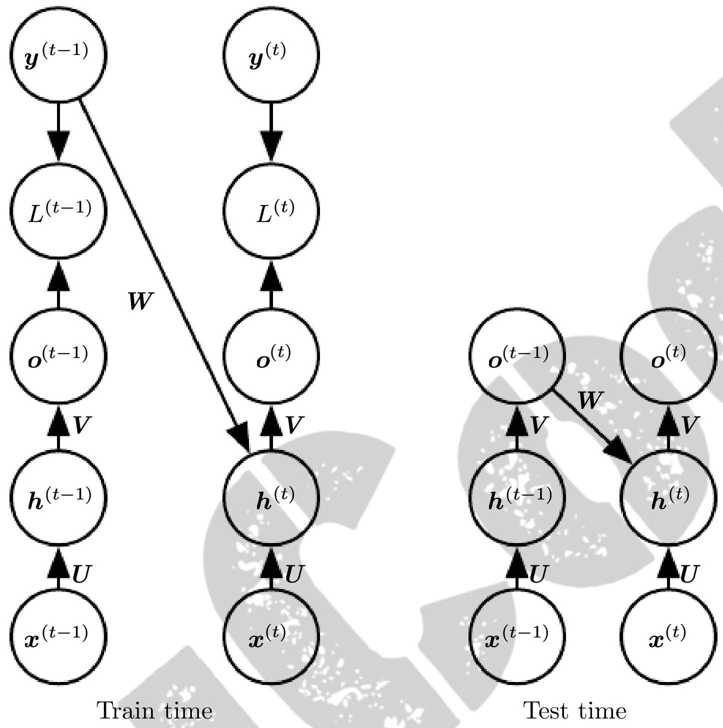

Figure 10.6: Illustration of teacher forcing. Teacher forcing is a training technique that is applicable to RNNs that have connections from their output to their hidden states at the next time step. *(Left)*At train time, we feed the *correct output* $\boldsymbol{y}^{(t)}$ drawn from the train set as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

likelihood criterion is

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

$$= \log p\left(\boldsymbol{y}^{(2)} \mid \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) + \log p\left(\boldsymbol{y}^{(1)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) \tag{10.16}$$

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 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 $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$, $\boldsymbol{b}$ 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 $\nabla_{\mathsf{N}} L$ 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

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

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 $\nabla_{\boldsymbol{o}^{(t)}} L$ on the outputs at time step t, for all i, t, is as follows:

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

We work our way backwards, 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:

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

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

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

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

where $\operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right)$ 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 $\nabla_{\boldsymbol{W}} f$ operator used in calculus 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 $\nabla_{\boldsymbol{W}^{(t)}}$ 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:

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

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

$$\nabla_{\boldsymbol{V}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla_{\boldsymbol{V}} o_i^{(t)} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{h}^{(t)^\top} \tag{10.24}$$

$$\nabla_{\boldsymbol{W}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} \tag{10.25}$$

$$= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{h}^{(t-1)^\top} \tag{10.26}$$

$$\nabla_{\boldsymbol{U}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} \tag{10.27}$$

$$= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{x}^{(t)^\top} \tag{10.28}$$

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.

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 $\boldsymbol{y}^{(t)}$ 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

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

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

$$\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}, \boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(t-1)}). \tag{10.30}$$

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 $\mathbb{Y} = \{\mathrm{y}^{(1)}, \ldots, \mathrm{y}^{(\tau)}\}$, with no additional inputs 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 y variables. We parametrize the joint distribution of these observations using the chain rule (equation 3.6) for conditional probabilities:

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

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 $\{y^{(1)}, \ldots, y^{(\tau)}\}$ according to such a model

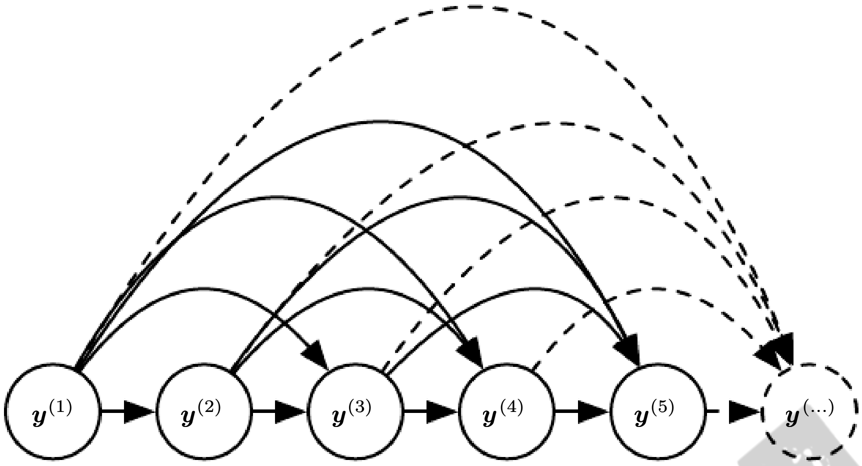

Figure 10.7: Fully connected graphical model for a sequence $y^{(1)}, y^{(2)}, \dots, y^{(t)}, \dots$: every past observation $y^{(i)}$ may influence the conditional distribution of some $y^{(t)}$ (for $t > i$), given the previous values. Parametrizing the graphical model directly according to this graph (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

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

where

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

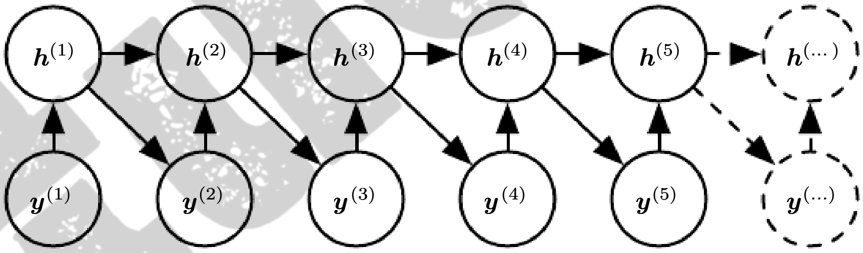

Figure 10.8: Introducing the state variable in the graphical model of the RNN, even though it is a deterministic function of its inputs, helps to see how we can obtain a very efficient parametrization, based on equation 10.5. Every stage in the sequence (for $\boldsymbol{h}^{(t)}$ and $\boldsymbol{y}^{(t)}$) 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 $\{\mathrm{y}^{(t-k)}, \ldots, \mathrm{y}^{(t-1)}\}$ to $\mathrm{y}^{(t)}$, 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 $\mathrm{y}^{(t)}$ 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.[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 $O(k^{\tau})$ 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 $\boldsymbol{\theta}$ 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

[1] 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.

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 $\boldsymbol{x}^{(\tau)}$ 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).

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 $\boldsymbol{x}$. 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 $P(\boldsymbol{y}|\boldsymbol{\omega})$ 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 $\mathbf{y}$ sequence. Some common ways of providing an extra input to an RNN are:

1. as an extra input at each time step, or
2. as the initial state $\boldsymbol{h}^{(0)}$, 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 $\boldsymbol{x}^\top \boldsymbol{R}$ 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 $\boldsymbol{x}^\top \boldsymbol{R}$ 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 non-conditional model and turning them into $\boldsymbol{\omega}$, where the bias parameters within $\boldsymbol{\omega}$ are now a function of the input.

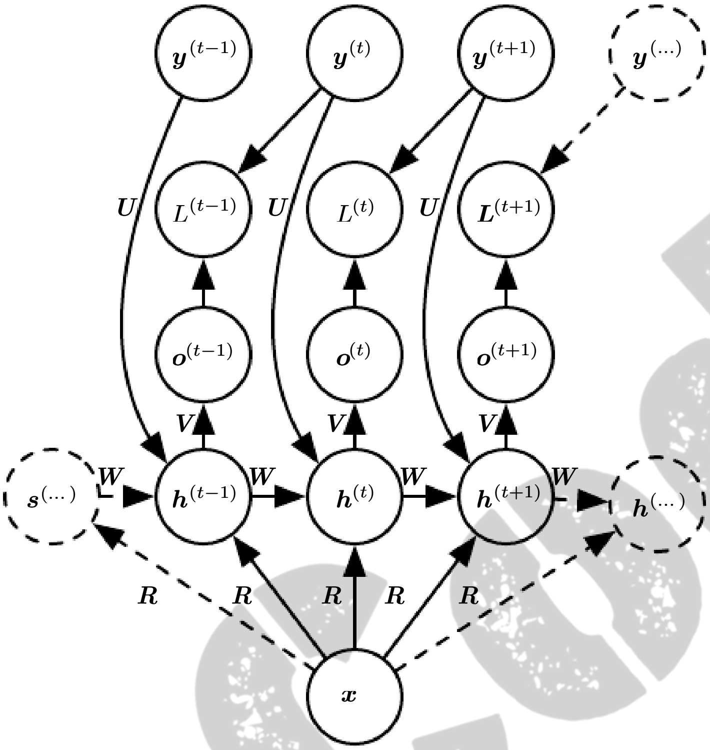

Figure 10.9: An RNN that maps a fixed-length vector $\boldsymbol{x}$ into a distribution over sequences $\mathbf{Y}$. This RNN is appropriate for tasks such as image captioning, where a single image is used as input to a model that then produces a sequence of words describing the image. Each element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

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

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

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

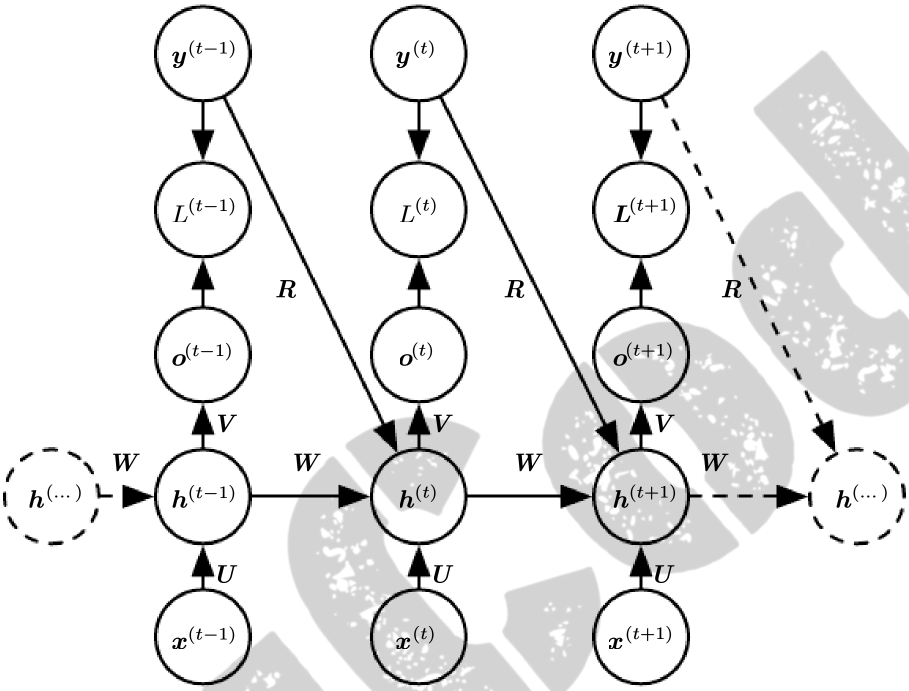

Figure 10.10: A conditional recurrent neural network mapping a variable-length sequence of $\boldsymbol{x}$ values into a distribution over sequences of $\boldsymbol{y}$ values of the same length. Compared to figure 10.3, this RNN contains connections from the previous output to the current state. These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

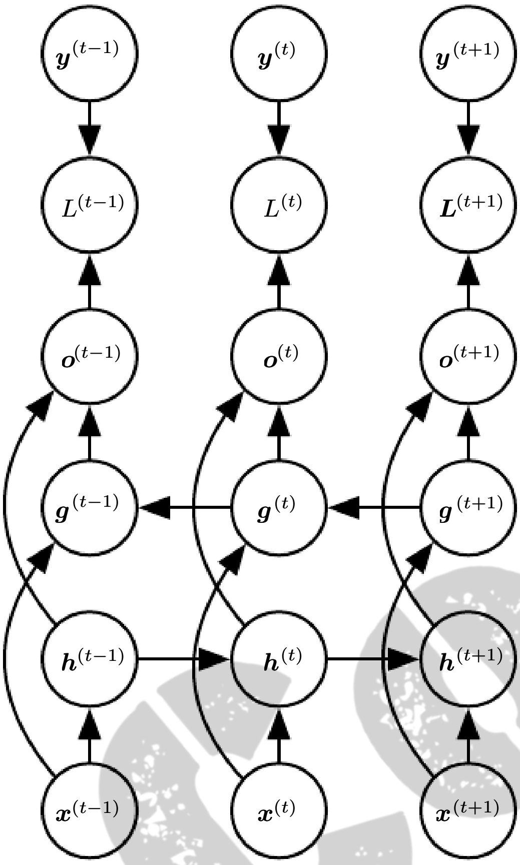

Figure 10.11: Computation of a typical bidirectional recurrent neural network, meant to learn to map input sequences $\boldsymbol{x}$ to target sequences $\boldsymbol{y}$, with loss $L^{(t)}$ at each step t. The $\boldsymbol{h}$ recurrence propagates information forward in time (towards the right) while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

10.3 Bidirectional RNNs

All of the recurrent networks we have considered up to now have a "causal" structure, meaning that the state at time t only captures information from the past, $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t-1)}$, 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.

However, in many applications we want to output a prediction of $\boldsymbol{y}^{(t)}$ 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 $\boldsymbol{h}^{(t)}$ standing for the state of the 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 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.

10.5 Deep Recurrent Networks

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 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 in order 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 to allocate 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

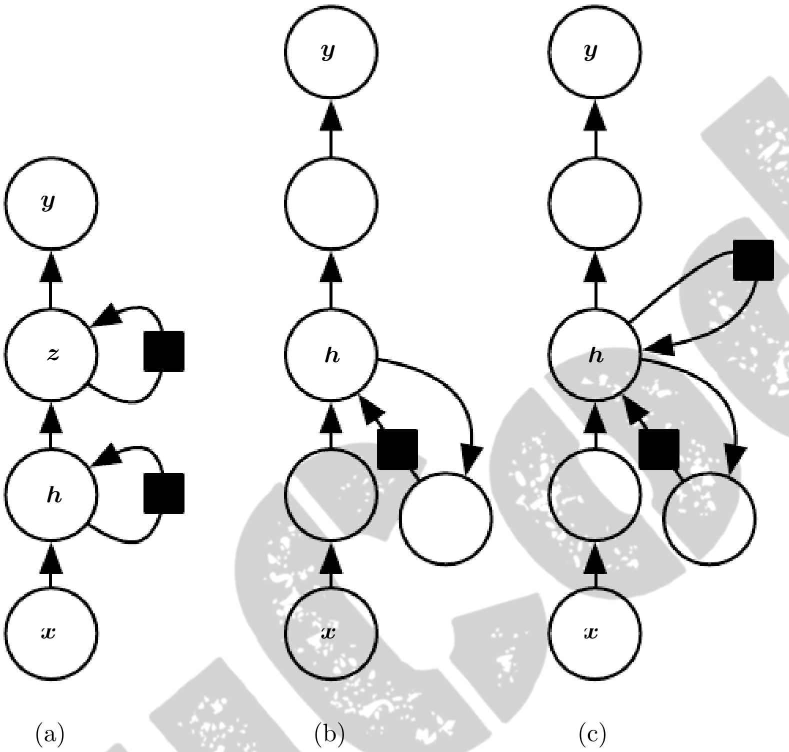

Figure 10.13: A recurrent neural network can be made deep in many ways (Pascanu *et al.*, 2014a). *(a)*The hidden recurrent state can be broken down into groups organized hierarchically. *(b)*Deeper computation (e.g., an MLP) can be introduced in the input-to-hidden, hidden-to-hidden and hidden-to-output parts. This may lengthen the shortest path linking different time steps. *(c)*The path-lengthening effect can be mitigated by introducing skip connections.

can be mitigated by introducing skip connections in the hidden-to-hidden path, as illustrated in figure 10.13c.

10.6 Recursive Neural Networks

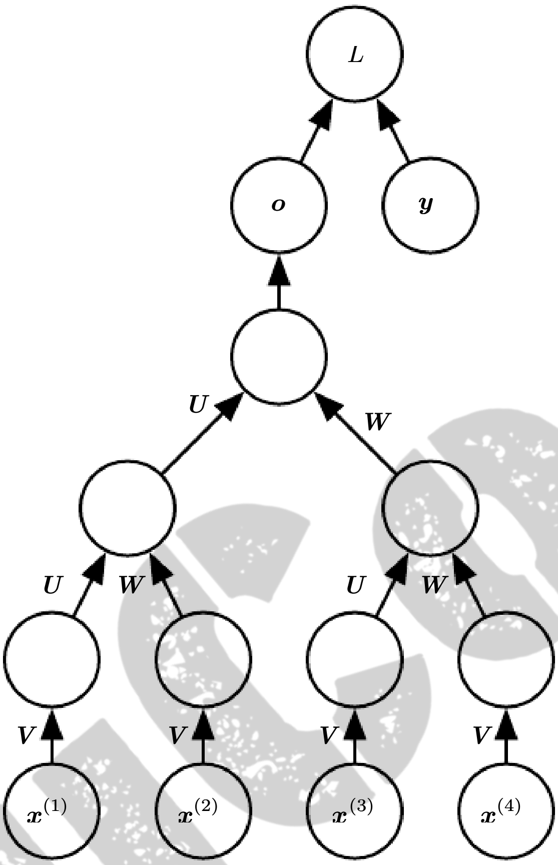

Figure 10.14: A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree. A variable-size sequence $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$ can be mapped to a fixed-size representation (the output $\boldsymbol{o}$), with a fixed set of parameters (the weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$). The figure illustrates a supervised learning case in which some target $\boldsymbol{y}$ is provided which is associated with the whole sequence.

Recursive neural networks[2] 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

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

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 which does not depend on the data, 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).

10.10 The Long Short-Term Memory and Other Gated RNNs

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.

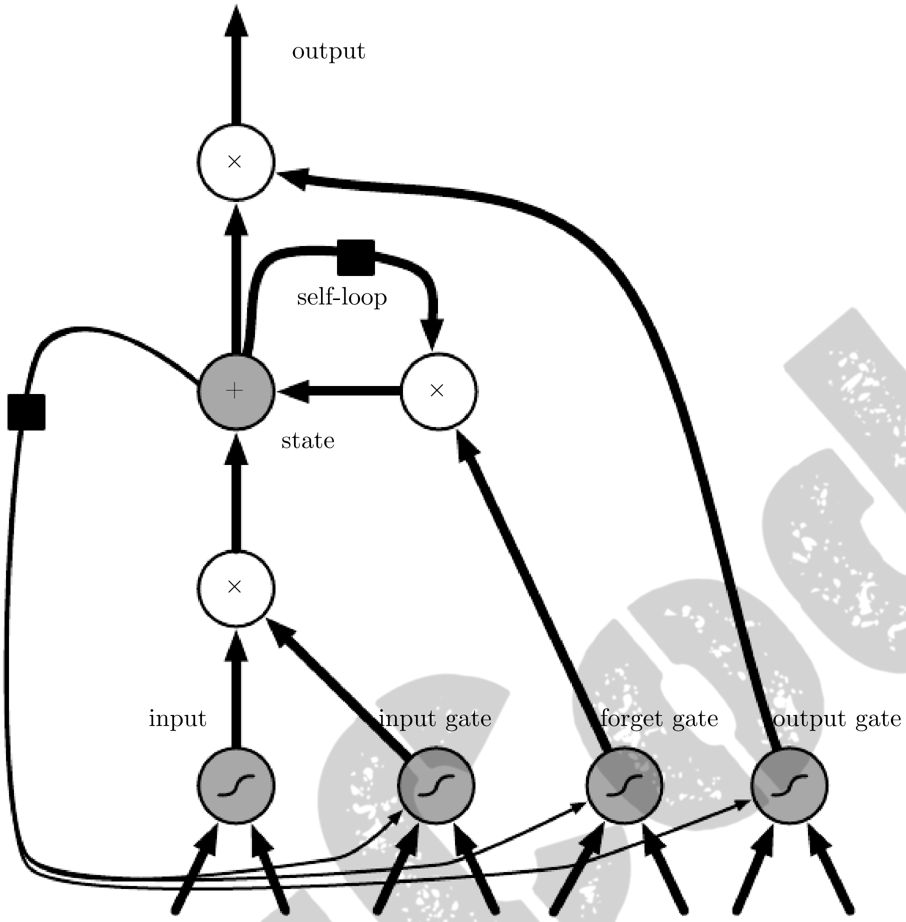

Figure 10.16: Block diagram of the LSTM recurrent network "cell." Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity. The state unit can also be used as an extra input to the gating units. The black square indicates a delay of a single time step.

Leaky units allow the network to *accumulate* information (such as evidence for a particular feature or category) over a long duration. However, once that information has been used, it might be useful for the neural network to *forget* the old state. For example, if a sequence is made of sub-sequences 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.

10.10.1 LSTM

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, in the case of 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 has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit $s_i^{(t)}$ that has a linear self-loop similar to the leaky units described in the previous section. However, here, the self-loop weight (or the associated time constant) is controlled by a **forget gate** unit $f_i^{(t)}$ (for time step t and cell i), that sets this weight to a value between 0 and 1 via a sigmoid unit:

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

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 $f_i^{(t)}$:

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

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:

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

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:

$$h_i^{(t)} = \tanh\left(s_i^{(t)} \right) q_i^{(t)} \tag{10.43}$$

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

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 data sets 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 have been studied and used and are discussed next.

10.10.2 Other Gated RNNs

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

Some answers to these questions are given with the recent work on gated RNNs, whose units are also known as gated recurrent units 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 updates gates can individually "ignore" parts of the state vector. The update gates act like conditional leaky integrators that can linearly gate any dimension, thus choosing to copy it (at one extreme of the sigmoid) or completely ignore it (at the other extreme) by replacing it by the new "target state" value (towards which the leaky integrator wants to converge). The reset gates control which parts of the state get used to compute the next target state, introducing an additional 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. However, several investigations over architectural variations of the LSTM and GRU found no variant that would clearly beat both of these across a wide range of tasks (Greff *et al.*, 2015; Jozefowicz *et al.*, 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.

Applications

In this chapter, we describe how to use deep learning to solve applications in computer vision, speech recognition, natural language processing, and other application areas of commercial interest. We begin by discussing the large scale neural network implementations required for most serious AI applications. Next, we review several specific application areas that deep learning has been used to solve. While one goal of deep learning is to design algorithms that are capable of solving a broad variety of tasks, so far some degree of specialization is needed. For example, vision tasks require processing a large number of input features (pixels) per example. Language tasks require modeling a large number of possible values (words in the vocabulary) per input feature.

12.1 Large-Scale Deep Learning

Deep learning is based on the philosophy of connectionism: while an individual biological neuron or an individual feature in a machine learning model is not intelligent, a large population of these neurons or features acting together can exhibit intelligent behavior. It truly is important to emphasize the fact that the number of neurons must be *large*. One of the key factors responsible for the improvement in neural network's accuracy and the improvement of the complexity of tasks they can solve between the 1980s and today is the dramatic increase in the size of the networks we use. As we saw in section 1.2.3, network sizes have grown exponentially for the past three decades, yet artificial neural networks are only as large as the nervous systems of insects.

Because the size of neural networks is of paramount importance, deep learning requires high performance hardware and software infrastructure.

12.1.1 Fast CPU Implementations

Traditionally, neural networks were trained using the CPU of a single machine. Today, this approach is generally considered insufficient. We now mostly use GPU computing or the CPUs of many machines networked together. Before moving to these expensive setups, researchers worked hard to demonstrate that CPUs could not manage the high computational workload required by neural networks.

A description of how to implement efficient numerical CPU code is beyond the scope of this book, but we emphasize here that careful implementation for specific CPU families can yield large improvements. For example, in 2011, the best CPUs available could run neural network workloads faster when using fixed-point arithmetic rather than floating-point arithmetic. By creating a carefully tuned fixed-point implementation, Vanhoucke *et al.* (2011) obtained a threefold speedup over a strong floating-point system. Each new model of CPU has different performance characteristics, so sometimes floating-point implementations can be faster too. The important principle is that careful specialization of numerical computation routines can yield a large payoff. Other strategies, besides choosing whether to use fixed or floating point, include optimizing data structures to avoid cache misses and using vector instructions. Many machine learning researchers neglect these implementation details, but when the performance of an implementation restricts the size of the model, the accuracy of the model suffers.

12.1.2 GPU Implementations

Most modern neural network implementations are based on graphics processing units. Graphics processing units (GPUs) are specialized hardware components that were originally developed for graphics applications. The consumer market for video gaming systems spurred development of graphics processing hardware. The performance characteristics needed for good video gaming systems turn out to be beneficial for neural networks as well.

Video game rendering requires performing many operations in parallel quickly. Models of characters and environments are specified in terms of lists of 3-D coordinates of vertices. Graphics cards must perform matrix multiplication and division on many vertices in parallel to convert these 3-D coordinates into 2-D on-screen coordinates. The graphics card must then perform many computations at each pixel in parallel to determine the color of each pixel. In both cases, the computations are fairly simple and do not involve much branching compared to the computational workload that a CPU usually encounters. For example, each vertex in the same rigid object will be multiplied by the same matrix; there is no need to evaluate an if statement per-vertex to determine which matrix to multiply by. The computations are also entirely independent of each other, and thus may be parallelized easily. The computations also involve processing massive buffers of memory, containing bitmaps describing the texture (color pattern) of each object to be rendered. Together, this results in graphics cards having been designed to have a high degree of parallelism and high memory bandwidth, at the cost of having a lower clock speed and less branching capability relative to traditional CPUs.

Neural network algorithms require the same performance characteristics as the real-time graphics algorithms described above. Neural networks usually involve large and numerous buffers of parameters, activation values, and gradient values, each of which must be completely updated during every step of training. These buffers are large enough to fall outside the cache of a traditional desktop computer so the memory bandwidth of the system often becomes the rate limiting factor. GPUs offer a compelling advantage over CPUs due to their high memory bandwidth. Neural network training algorithms typically do not involve much branching or sophisticated control, so they are appropriate for GPU hardware. Since neural networks can be divided into multiple individual “neurons” that can be processed independently from the other neurons in the same layer, neural networks easily benefit from the parallelism of GPU computing.

GPU hardware was originally so specialized that it could only be used for graphics tasks. Over time, GPU hardware became more flexible, allowing custom subroutines to be used to transform the coordinates of vertices or assign colors to pixels. In principle, there was no requirement that these pixel values actually be based on a rendering task. These GPUs could be used for scientific computing by writing the output of a computation to a buffer of pixel values. Steinkrau *et al.* (2005) implemented a two-layer fully connected neural network on a GPU and reported a threefold speedup over their CPU-based baseline. Shortly thereafter, Chellapilla *et al.* (2006) demonstrated that the same technique could be used to accelerate supervised convolutional networks.

The popularity of graphics cards for neural network training exploded after the advent of **general purpose GPUs**. These GP-GPUs could execute arbitrary code, not just rendering subroutines. NVIDIA’s CUDA programming language provided a way to write this arbitrary code in a C-like language. With their relatively convenient programming model, massive parallelism, and high memory bandwidth, GP-GPUs now offer an ideal platform for neural network programming. This platform was rapidly adopted by deep learning researchers soon after it became available (Raina *et al.*, 2009; Ciresan *et al.*, 2010).

Writing efficient code for GP-GPUs remains a difficult task best left to specialists. The techniques required to obtain good performance on GPU are very different from those used on CPU. For example, good CPU-based code is usually designed to read information from the cache as much as possible. On GPU, most writable memory locations are not cached, so it can actually be faster to compute the same value twice, rather than compute it once and read it back from memory. GPU code is also inherently multi-threaded and the different threads must be coordinated with each other carefully. For example, memory operations are faster if they can be **coalesced**. Coalesced reads or writes occur when several threads can each read or write a value that they need simultaneously, as part of a single memory transaction. Different models of GPUs are able to coalesce different kinds of read or write patterns. Typically, memory operations are easier to coalesce if among n threads, thread i accesses byte $i + j$ of memory, and j is a multiple of some power of 2. The exact specifications differ between models of GPU. Another common consideration for GPUs is making sure that each thread in a group executes the same instruction simultaneously. This means that branching can be difficult on GPU. Threads are divided into small groups called **warps**. Each thread in a warp executes the same instruction during each cycle, so if different threads within the same warp need to execute different code paths, these different code paths must be traversed sequentially rather than in parallel.

Due to the difficulty of writing high performance GPU code, researchers should structure their workflow to avoid needing to write new GPU code in order to test new models or algorithms. Typically, one can do this by building a software library of high performance operations like convolution and matrix multiplication, then specifying models in terms of calls to this library of operations. For example, the machine learning library Pylearn2 (Goodfellow *et al.*, 2013c) specifies all of its machine learning algorithms in terms of calls to Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) and cuda-convnet (Krizhevsky, 2010), which provide these high-performance operations. This factored approach can also ease support for multiple kinds of hardware. For example, the same Theano program can run on either CPU or GPU, without needing to change any of the calls to Theano itself. Other libraries like TensorFlow (Abadi *et al.*, 2015) and Torch (Collobert *et al.*, 2011b) provide similar features.

12.1.3 Large-Scale Distributed Implementations

In many cases, the computational resources available on a single machine are insufficient. We therefore want to distribute the workload of training and inference across many machines.

Distributing inference is simple, because each input example we want to process can be run by a separate machine. This is known as **data parallelism**.

It is also possible to get **model parallelism**, where multiple machines work together on a single datapoint, with each machine running a different part of the model. This is feasible for both inference and training.

Data parallelism during training is somewhat harder. We can increase the size of the minibatch used for a single SGD step, but usually we get less than linear returns in terms of optimization performance. It would be better to allow multiple machines to compute multiple gradient descent steps in parallel. Unfortunately, the standard definition of gradient descent is as a completely sequential algorithm: the gradient at step t is a function of the parameters produced by step $t-1$.

This can be solved using **asynchronous stochastic gradient descent** (Bengio *et al.*, 2001; Recht *et al.*, 2011). In this approach, several processor cores share the memory representing the parameters. Each core reads parameters without a lock, then computes a gradient, then increments the parameters without a lock. This reduces the average amount of improvement that each gradient descent step yields, because some of the cores overwrite each other's progress, but the increased rate of production of steps causes the learning process to be faster overall. Dean *et al.* (2012) pioneered the multi-machine implementation of this lock-free approach to gradient descent, where the parameters are managed by a **parameter server** rather than stored in shared memory. Distributed asynchronous gradient descent remains the primary strategy for training large deep networks and is used by most major deep learning groups in industry (Chilimbi *et al.*, 2014; Wu *et al.*, 2015). Academic deep learning researchers typically cannot afford the same scale of distributed learning systems but some research has focused on how to build distributed networks with relatively low-cost hardware available in the university setting (Coates *et al.*, 2013).

12.1.4 Model Compression

In many commercial applications, it is much more important that the time and memory cost of running inference in a machine learning model be low than that the time and memory cost of training be low. For applications that do not require personalization, it is possible to train a model once, then deploy it to be used by billions of users. In many cases, the end user is more resource-constrained than the developer. For example, one might train a speech recognition network with a powerful computer cluster, then deploy it on mobile phones.

A key strategy for reducing the cost of inference is **model compression** (Buciluă *et al.*, 2006). The basic idea of model compression is to replace the original, expensive model with a smaller model that requires less memory and runtime to store and evaluate.

Model compression is applicable when the size of the original model is driven primarily by a need to prevent overfitting. In most cases, the model with the lowest generalization error is an ensemble of several independently trained models. Evaluating all n ensemble members is expensive. Sometimes, even a single model generalizes better if it is large (for example, if it is regularized with dropout).

These large models learn some function $f(\boldsymbol{x})$, but do so using many more parameters than are necessary for the task. Their size is necessary only due to the limited number of training examples. As soon as we have fit this function $f(\boldsymbol{x})$, we can generate a training set containing infinitely many examples, simply by applying f to randomly sampled points $\boldsymbol{x}$. We then train the new, smaller, model to match $f(\boldsymbol{x})$ on these points. In order to most efficiently use the capacity of the new, small model, it is best to sample the new $\boldsymbol{x}$ points from a distribution resembling the actual test inputs that will be supplied to the model later. This can be done by corrupting training examples or by drawing points from a generative model trained on the original training set.

Alternatively, one can train the smaller model only on the original training points, but train it to copy other features of the model, such as its posterior distribution over the incorrect classes (Hinton *et al.*, 2014, 2015).

12.1.5 Dynamic Structure

One strategy for accelerating data processing systems in general is to build systems that have **dynamic structure** in the graph describing the computation needed to process an input. Data processing systems can dynamically determine which subset of many neural networks should be run on a given input. Individual neural networks can also exhibit dynamic structure internally by determining which subset of features (hidden units) to compute given information from the input. This form of dynamic structure inside neural networks is sometimes called **conditional computation** (Bengio, 2013; Bengio *et al.*, 2013b). Since many components of the architecture may be relevant only for a small amount of possible inputs, the system can run faster by computing these features only when they are needed.

Dynamic structure of computations is a basic computer science principle applied generally throughout the software engineering discipline. The simplest versions of dynamic structure applied to neural networks are based on determining which subset of some group of neural networks (or other machine learning models) should be applied to a particular input.

A venerable strategy for accelerating inference in a classifier is to use a **cascade** of classifiers. The cascade strategy may be applied when the goal is to detect the presence of a rare object (or event). To know for sure that the object is present, we must use a sophisticated classifier with high capacity, that is expensive to run. However, because the object is rare, we can usually use much less computation to reject inputs as not containing the object. In these situations, we can train a sequence of classifiers. The first classifiers in the sequence have low capacity, and are trained to have high recall. In other words, they are trained to make sure we do not wrongly reject an input when the object is present. The final classifier is trained to have high precision. At test time, we run inference by running the classifiers in a sequence, abandoning any example as soon as any one element in the cascade rejects it. Overall, this allows us to verify the presence of objects with high confidence, using a high capacity model, but does not force us to pay the cost of full inference for every example. There are two different ways that the cascade can achieve high capacity. One way is to make the later members of the cascade individually have high capacity. In this case, the system as a whole obviously has high capacity, because some of its individual members do. It is also possible to make a cascade in which every individual model has low capacity but the system as a whole has high capacity due to the combination of many small models. Viola and Jones (2001) used a cascade of boosted decision trees to implement a fast and robust face detector suitable for use in handheld digital cameras. Their classifier localizes a face using essentially a sliding window approach in which many windows are examined and rejected if they do not contain faces. Another version of cascades uses the earlier models to implement a sort of hard attention mechanism: the early members of the cascade localize an object and later members of the cascade perform further processing given the location of the object. For example, Google transcribes address numbers from Street View imagery using a two-step cascade that first locates the address number with one machine learning model and then transcribes it with another (Goodfellow *et al.*, 2014d).

Decision trees themselves are an example of dynamic structure, because each node in the tree determines which of its subtrees should be evaluated for each input. A simple way to accomplish the union of deep learning and dynamic structure is to train a decision tree in which each node uses a neural network to make the splitting decision (Guo and Gelfand, 1992), though this has typically not been done with the primary goal of accelerating inference computations.

In the same spirit, one can use a neural network, called the **gater** to select which one out of several **expert networks** will be used to compute the output, given the current input. The first version of this idea is called the **mixture of experts** (Nowlan, 1990; Jacobs *et al.*, 1991), in which the gater outputs a set of probabilities or weights (obtained via a softmax nonlinearity), one per expert, and the final output is obtained by the weighted combination of the output of the experts. In that case, the use of the gater does not offer a reduction in computational cost, but if a single expert is chosen by the gater for each example, we obtain the **hard mixture of experts** (Collobert *et al.*, 2001, 2002), which can considerably accelerate training and inference time. This strategy works well when the number of gating decisions is small because it is not combinatorial. But when we want to select different subsets of units or parameters, it is not possible to use a "soft switch" because it requires enumerating (and computing outputs for) all the gater configurations. To deal with this problem, several approaches have been explored to train combinatorial gaters. Bengio *et al.* (2013b) experiment with several estimators of the gradient on the gating probabilities, while Bacon *et al.* (2015) and Bengio *et al.* (2015a) use reinforcement learning techniques (policy gradient) to learn a form of conditional dropout on blocks of hidden units and get an actual reduction in computational cost without impacting negatively on the quality of the approximation.

Another kind of dynamic structure is a switch, where a hidden unit can receive input from different units depending on the context. This dynamic routing approach can be interpreted as an attention mechanism (Olshausen *et al.*, 1993). So far, the use of a hard switch has not proven effective on large-scale applications. Contemporary approaches instead use a weighted average over many possible inputs, and thus do not achieve all of the possible computational benefits of dynamic structure. Contemporary attention mechanisms are described in section 12.4.5.1.

One major obstacle to using dynamically structured systems is the decreased degree of parallelism that results from the system following different code branches for different inputs. This means that few operations in the network can be described as matrix multiplication or batch convolution on a minibatch of examples. We can write more specialized sub-routines that convolve each example with different kernels or multiply each row of a design matrix by a different set of columns of weights. Unfortunately, these more specialized subroutines are difficult to implement efficiently. CPU implementations will be slow due to the lack of cache coherence and GPU implementations will be slow due to the lack of coalesced memory transactions and the need to serialize warps when members of a warp take different branches. In some cases, these issues can be mitigated by partitioning the examples into groups that all take the same branch, and processing these groups of examples simultaneously. This can be an acceptable strategy for minimizing the time required to process a fixed amount of examples in an offline setting. In a real-time setting where examples must be processed continuously, partitioning the workload can result in load-balancing issues. For example, if we assign one machine to process the first step in a cascade and another machine to process the last step in a cascade, then the first will tend to be overloaded and the last will tend to be underloaded. Similar issues arise if each machine is assigned to implement different nodes of a neural decision tree.

12.1.6 Specialized Hardware Implementations of Deep Networks

Since the early days of neural networks research, hardware designers have worked on specialized hardware implementations that could speed up training and/or inference of neural network algorithms. See early and more recent reviews of specialized hardware for deep networks (Lindsey and Lindblad, 1994; Beiu *et al.*, 2003; Misra and Saha, 2010).

Different forms of specialized hardware (Graf and Jackel, 1989; Mead and Ismail, 2012; Kim *et al.*, 2009; Pham *et al.*, 2012; Chen *et al.*, 2014a,b) have been developed over the last decades, either with ASICs (application-specific integrated circuit), either with digital (based on binary representations of numbers), analog (Graf and Jackel, 1989; Mead and Ismail, 2012) (based on physical implementations of continuous values as voltages or currents) or hybrid implementations (combining digital and analog components). In recent years more flexible FPGA (field programmable gated array) implementations (where the particulars of the circuit can be written on the chip after it has been built) have been developed.

Though software implementations on general-purpose processing units (CPUs and GPUs) typically use 32 or 64 bits of precision to represent floating point numbers, it has long been known that it was possible to use less precision, at least at inference time (Holt and Baker, 1991; Holi and Hwang, 1993; Presley and Haggard, 1994; Simard and Graf, 1994; Wawrzynek *et al.*, 1996; Savich *et al.*, 2007). This has become a more pressing issue in recent years as deep learning has gained in popularity in industrial products, and as the great impact of faster hardware was demonstrated with GPUs. Another factor that motivates current research on specialized hardware for deep networks is that the rate of progress of a single CPU or GPU core has slowed down, and most recent improvements in computing speed have come from parallelization across cores (either in CPUs or GPUs). This is very different from the situation of the 1990s (the previous neural network era) where the hardware implementations of neural networks (which might take two years from inception to availability of a chip) could not keep up with the rapid progress and low prices of general-purpose CPUs. Building specialized hardware is thus a way to push the envelope further, at a time when new hardware designs are being developed for low-power devices such as phones, aiming for general-public applications of deep learning (e.g., with speech, computer vision or natural language).

Recent work on low-precision implementations of backprop-based neural nets (Vanhoucke *et al.*, 2011; Courbariaux *et al.*, 2015; Gupta *et al.*, 2015) suggests that between 8 and 16 bits of precision can suffice for using or training deep neural networks with back-propagation. What is clear is that more precision is required during training than at inference time, and that some forms of dynamic fixed point representation of numbers can be used to reduce how many bits are required per number. Traditional fixed point numbers are restricted to a fixed range (which corresponds to a given exponent in a floating point representation). Dynamic fixed point representations share that range among a set of numbers (such as all the weights in one layer). Using fixed point rather than floating point representations and using less bits per number reduces the hardware surface area, power requirements and computing time needed for performing multiplications, and multiplications are the most demanding of the operations needed to use or train a modern deep network with backprop.

12.2 Computer Vision

Computer vision has traditionally been one of the most active research areas for deep learning applications, because vision is a task that is effortless for humans and many animals but challenging for computers (Ballard *et al.*, 1983). Many of the most popular standard benchmark tasks for deep learning algorithms are forms of object recognition or optical character recognition.

Computer vision is a very broad field encompassing a wide variety of ways of processing images, and an amazing diversity of applications. Applications of computer vision range from reproducing human visual abilities, such as recognizing faces, to creating entirely new categories of visual abilities. As an example of the latter category, one recent computer vision application is to recognize sound waves from the vibrations they induce in objects visible in a video (Davis *et al.*, 2014). Most deep learning research on computer vision has not focused on such exotic applications that expand the realm of what is possible with imagery but rather a small core of AI goals aimed at replicating human abilities. Most deep learning for computer vision is used for object recognition or detection of some form, whether this means reporting which object is present in an image, annotating an image with bounding boxes around each object, transcribing a sequence of symbols from an image, or labeling each pixel in an image with the identity of the object it belongs to. Because generative modeling has been a guiding principle of deep learning research, there is also a large body of work on image synthesis using deep models. While image synthesis *ex nihilo* is usually not considered a computer vision endeavor, models capable of image synthesis are usually useful for image restoration, a computer vision task involving repairing defects in images or removing objects from images.

12.2.1 Preprocessing

Many application areas require sophisticated preprocessing because the original input comes in a form that is difficult for many deep learning architectures to represent. Computer vision usually requires relatively little of this kind of preprocessing. The images should be standardized so that their pixels all lie in the same, reasonable range, like [0,1] or [-1, 1]. Mixing images that lie in [0,1] with images that lie in [0, 255] will usually result in failure. Formatting images to have the same scale is the only kind of preprocessing that is strictly necessary. Many computer vision architectures require images of a standard size, so images must be cropped or scaled to fit that size. Even this rescaling is not always strictly necessary. Some convolutional models accept variably-sized inputs and dynamically adjust the size of their pooling regions to keep the output size constant (Waibel *et al.*, 1989). Other convolutional models have variable-sized output that automatically scales in size with the input, such as models that denoise or label each pixel in an image (Hadsell *et al.*, 2007).

Dataset augmentation may be seen as a way of preprocessing the training set only. Dataset augmentation is an excellent way to reduce the generalization error of most computer vision models. A related idea applicable at test time is to show the model many different versions of the same input (for example, the same image cropped at slightly different locations) and have the different instantiations of the model vote to determine the output. This latter idea can be interpreted as an ensemble approach, and helps to reduce generalization error.

Other kinds of preprocessing are applied to both the train and the test set with the goal of putting each example into a more canonical form in order to reduce the amount of variation that the model needs to account for. Reducing the amount of variation in the data can both reduce generalization error and reduce the size of the model needed to fit the training set. Simpler tasks may be solved by smaller models, and simpler solutions are more likely to generalize well. Preprocessing of this kind is usually designed to remove some kind of variability in the input data that is easy for a human designer to describe and that the human designer is confident has no relevance to the task. When training with large datasets and large models, this kind of preprocessing is often unnecessary, and it is best to just let the model learn which kinds of variability it should become invariant to. For example, the AlexNet system for classifying ImageNet only has one preprocessing step: subtracting the mean across training examples of each pixel (Krizhevsky *et al.*, 2012).

12.2.1.1 Contrast Normalization

One of the most obvious sources of variation that can be safely removed for many tasks is the amount of contrast in the image. Contrast simply refers to the magnitude of the difference between the bright and the dark pixels in an image. There are many ways of quantifying the contrast of an image. In the context of deep learning, contrast usually refers to the standard deviation of the pixels in an image or region of an image. Suppose we have an image represented by a tensor $\mathbf{X} \in \mathbb{R}^{r \times c \times 3}$, with $X_{i,j,1}$ being the red intensity at row i and column j, $X_{i,j,2}$ giving the green intensity and $X_{i,j,3}$ giving the blue intensity. Then the contrast of the entire image is given by

$$\sqrt{\frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}\left(X_{i,j,k}-\bar{\mathbf{X}}\right)^2} \tag{12.1}$$

where $\bar{\mathbf{X}}$ is the mean intensity of the entire image:

$$\bar{\mathbf{X}} = \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}X_{i,j,k}. \tag{12.2}$$

Global contrast normalization (GCN) aims to prevent images from having varying amounts of contrast by subtracting the mean from each image, then rescaling it so that the standard deviation across its pixels is equal to some constant s. This approach is complicated by the fact that no scaling factor can change the contrast of a zero-contrast image (one whose pixels all have equal intensity). Images with very low but non-zero contrast often have little information content. Dividing by the true standard deviation usually accomplishes nothing more than amplifying sensor noise or compression artifacts in such cases. This motivates introducing a small, positive regularization parameter λ to bias the estimate of the standard deviation. Alternately, one can constrain the denominator to be at least ϵ. Given an input image $\mathbf{X}$, GCN produces an output image $\mathbf{X}'$, defined such that

$$X'_{i,j,k} = s\frac{X_{i,j,k} - \bar{X}}{\max\left\{\epsilon, \sqrt{\lambda + \frac{1}{3rc}\sum_{i=1}^{r}\sum_{j=1}^{c}\sum_{k=1}^{3}\left(X_{i,j,k} - \bar{X}\right)^2}\right\}}. \tag{12.3}$$

Datasets consisting of large images cropped to interesting objects are unlikely to contain any images with nearly constant intensity. In these cases, it is safe to practically ignore the small denominator problem by setting $\lambda = 0$ and avoid division by 0 in extremely rare cases by setting ϵ to an extremely low value like 10^{-8}. This is the approach used by Goodfellow *et al.* (2013a) on the CIFAR-10 dataset. Small images cropped randomly are more likely to have nearly constant intensity, making aggressive regularization more useful. Coates *et al.* (2011) used $\epsilon = 0$ and $\lambda = 10$ on small, randomly selected patches drawn from CIFAR-10.

The scale parameter s can usually be set to 1, as done by Coates *et al.* (2011), or chosen to make each individual pixel have standard deviation across examples close to 1, as done by Goodfellow *et al.* (2013a).

The standard deviation in equation 12.3 is just a rescaling of the L^2 norm of the image (assuming the mean of the image has already been removed). It is preferable to define GCN in terms of standard deviation rather than L^2 norm because the standard deviation includes division by the number of pixels, so GCN based on standard deviation allows the same s to be used regardless of image size. However, the observation that the L^2 norm is proportional to the standard deviation can help build a useful intuition. One can understand GCN as mapping examples to a spherical shell. See figure 12.1 for an illustration. This can be a useful property because neural networks are often better at responding to directions in space rather than exact locations. Responding to multiple distances in the same direction requires hidden units with collinear weight vectors but different biases. Such coordination can be difficult for the learning algorithm to discover. Additionally, many shallow graphical models have problems with representing multiple separated modes along the same line. GCN avoids these problems by reducing each example to a direction rather than a direction and a distance.

Counterintuitively, there is a preprocessing operation known as **sphering** and it is not the same operation as GCN. Sphering does not refer to making the data lie on a spherical shell, but rather to rescaling the principal components to have

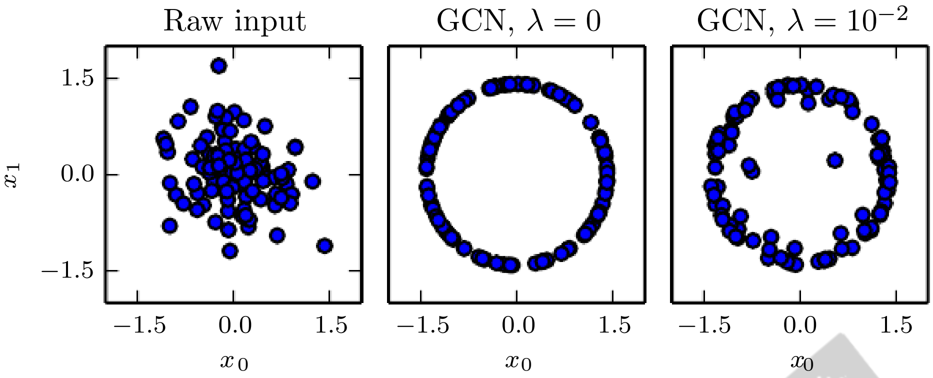

Figure 12.1: GCN maps examples onto a sphere. *(Left)*Raw input data may have any norm. *(Center)*GCN with $\lambda = 0$ maps all non-zero examples perfectly onto a sphere. Here we use $s = 1$ and $\epsilon = 10^{-8}$. Because we use GCN based on normalizing the standard deviation rather than the L^2 norm, the resulting sphere is not the unit sphere. *(Right)*Regularized GCN, with $\lambda > 0$, draws examples toward the sphere but does not completely discard the variation in their norm. We leave s and ϵ the same as before.

equal variance, so that the multivariate normal distribution used by PCA has spherical contours. Sphering is more commonly known as **whitening**.

Global contrast normalization will often fail to highlight image features we would like to stand out, such as edges and corners. If we have a scene with a large dark area and a large bright area (such as a city square with half the image in the shadow of a building) then global contrast normalization will ensure there is a large difference between the brightness of the dark area and the brightness of the light area. It will not, however, ensure that edges within the dark region stand out.

This motivates **local contrast normalization**. Local contrast normalization ensures that the contrast is normalized across each small window, rather than over the image as a whole. See figure 12.2 for a comparison of global and local contrast normalization.

Various definitions of local contrast normalization are possible. In all cases, one modifies each pixel by subtracting a mean of nearby pixels and dividing by a standard deviation of nearby pixels. In some cases, this is literally the mean and standard deviation of all pixels in a rectangular window centered on the pixel to be modified (Pinto *et al.*, 2008). In other cases, this is a weighted mean and weighted standard deviation using Gaussian weights centered on the pixel to be modified. In the case of color images, some strategies process different color

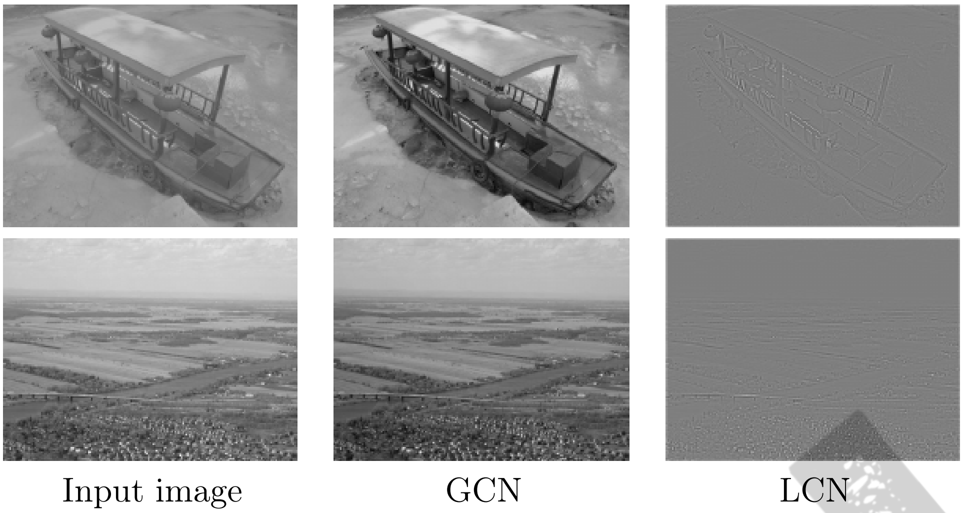

Figure 12.2: A comparison of global and local contrast normalization. Visually, the effects of global contrast normalization are subtle. It places all images on roughly the same scale, which reduces the burden on the learning algorithm to handle multiple scales. Local contrast normalization modifies the image much more, discarding all regions of constant intensity. This allows the model to focus on just the edges. Regions of fine texture, such as the houses in the second row, may lose some detail due to the bandwidth of the normalization kernel being too high.

channels separately while others combine information from different channels to normalize each pixel (Sermanet *et al.*, 2012).

Local contrast normalization can usually be implemented efficiently by using separable convolution (see section 9.8) to compute feature maps of local means and local standard deviations, then using element-wise subtraction and element-wise division on different feature maps.

Local contrast normalization is a differentiable operation and can also be used as a nonlinearity applied to the hidden layers of a network, as well as a preprocessing operation applied to the input.

As with global contrast normalization, we typically need to regularize local contrast normalization to avoid division by zero. In fact, because local contrast normalization typically acts on smaller windows, it is even more important to regularize. Smaller windows are more likely to contain values that are all nearly the same as each other, and thus more likely to have zero standard deviation.

12.2.1.2 Dataset Augmentation

As described in section 7.4, it is easy to improve the generalization of a classifier by increasing the size of the training set by adding extra copies of the training examples that have been modified with transformations that do not change the class. Object recognition is a classification task that is especially amenable to this form of dataset augmentation because the class is invariant to so many transformations and the input can be easily transformed with many geometric operations. As described before, classifiers can benefit from random translations, rotations, and in some cases, flips of the input to augment the dataset. In specialized computer vision applications, more advanced transformations are commonly used for dataset augmentation. These schemes include random perturbation of the colors in an image (Krizhevsky *et al.*, 2012) and nonlinear geometric distortions of the input (LeCun *et al.*, 1998b).

12.3 Speech Recognition

The task of speech recognition is to map an acoustic signal containing a spoken natural language utterance into the corresponding sequence of words intended by the speaker. Let $\boldsymbol{X} = (\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(T)})$ denote the sequence of acoustic input vectors (traditionally produced by splitting the audio into 20ms frames). Most speech recognition systems preprocess the input using specialized hand-designed features, but some (Jaitly and Hinton, 2011) deep learning systems learn features from raw input. Let $\boldsymbol{y} = (y_1, y_2, \ldots, y_N)$ denote the target output sequence (usually a sequence of words or characters). The **automatic speech recognition** (ASR) task consists of creating a function f^*_{ASR} that computes the most probable linguistic sequence $\boldsymbol{y}$ given the acoustic sequence $\boldsymbol{X}$:

$$f^*_{\text{ASR}}(\boldsymbol{X}) = \arg\max_{\boldsymbol{y}} P^*(\mathbf{y} \mid \mathbf{X} = \boldsymbol{X}) \tag{12.4}$$

where P^* is the true conditional distribution relating the inputs $\boldsymbol{X}$ to the targets $\boldsymbol{y}$.

Since the 1980s and until about 2009–2012, state-of-the art speech recognition systems primarily combined hidden Markov models (HMMs) and Gaussian mixture models (GMMs). GMMs modeled the association between acoustic features and phonemes (Bahl *et al.*, 1987), while HMMs modeled the sequence of phonemes. The GMM-HMM model family treats acoustic waveforms as being generated by the following process: first an HMM generates a sequence of phonemes and discrete sub-phonemic states (such as the beginning, middle, and end of each phoneme), then a GMM transforms each discrete symbol into a brief segment of audio waveform. Although GMM-HMM systems dominated ASR until recently, speech recognition was actually one of the first areas where neural networks were applied, and numerous ASR systems from the late 1980s and early 1990s used neural nets (Bourlard and Wellekens, 1989; Waibel *et al.*, 1989; Robinson and Fallside, 1991; Bengio *et al.*, 1991, 1992; Konig *et al.*, 1996). At the time, the performance of ASR based on neural nets approximately matched the performance of GMM-HMM systems. For example, Robinson and Fallside (1991) achieved 26% phoneme error rate on the TIMIT (Garofolo *et al.*, 1993) corpus (with 39 phonemes to discriminate between), which was better than or comparable to HMM-based systems. Since then, TIMIT has been a benchmark for phoneme recognition, playing a role similar to the role MNIST plays for object recognition. However, because of the complex engineering involved in software systems for speech recognition and the effort that had been invested in building these systems on the basis of GMM-HMMs, the industry did not see a compelling argument for switching to neural networks. As a consequence, until the late 2000s, both academic and industrial research in using neural nets for speech recognition mostly focused on using neural nets to learn extra features for GMM-HMM systems.

Later, with *much larger and deeper models* and much larger datasets, recognition accuracy was dramatically improved by using neural networks to replace GMMs for the task of associating acoustic features to phonemes (or sub-phonemic states). Starting in 2009, speech researchers applied a form of deep learning based on unsupervised learning to speech recognition. This approach to deep learning was based on training undirected probabilistic models called restricted Boltzmann machines (RBMs) to model the input data. RBMs will be described in part III. To solve speech recognition tasks, unsupervised pretraining was used to build deep feedforward networks whose layers were each initialized by training an RBM. These networks take spectral acoustic representations in a fixed-size input window (around a center frame) and predict the conditional probabilities of HMM states for that center frame. Training such deep networks helped to significantly improve the recognition rate on TIMIT (Mohamed *et al.*, 2009, 2012a), bringing down the phoneme error rate from about 26% to 20.7%. See Mohamed *et al.* (2012b) for an analysis of reasons for the success of these models. Extensions to the basic phone recognition pipeline included the addition of speaker-adaptive features (Mohamed *et al.*, 2011) that further reduced the error rate. This was quickly followed up by work to expand the architecture from phoneme recognition (which is what TIMIT is focused on) to large-vocabulary speech recognition (Dahl *et al.*, 2012), which involves not just recognizing phonemes but also recognizing sequences of words from a large vocabulary. Deep networks for speech recognition eventually shifted from being based on pretraining and Boltzmann machines to being based on techniques such as rectified linear units and dropout (Zeiler *et al.*, 2013; Dahl *et al.*, 2013). By that time, several of the major speech groups in industry had started exploring deep learning in collaboration with academic researchers. Hinton *et al.* (2012a) describe the breakthroughs achieved by these collaborators, which are now deployed in products such as mobile phones.

Later, as these groups explored larger and larger labeled datasets and incorporated some of the methods for initializing, training, and setting up the architecture of deep nets, they realized that the unsupervised pretraining phase was either unnecessary or did not bring any significant improvement.

These breakthroughs in recognition performance for word error rate in speech recognition were unprecedented (around 30% improvement) and were following a long period of about ten years during which error rates did not improve much with the traditional GMM-HMM technology, in spite of the continuously growing size of training sets (see figure 2.4 of Deng and Yu (2014)). This created a rapid shift in the speech recognition community towards deep learning. In a matter of roughly two years, most of the industrial products for speech recognition incorporated deep neural networks and this success spurred a new wave of research into deep learning algorithms and architectures for ASR, which is still ongoing today.

One of these innovations was the use of convolutional networks (Sainath *et al.*, 2013) that replicate weights across time and frequency, improving over the earlier time-delay neural networks that replicated weights only across time. The new two-dimensional convolutional models regard the input spectrogram not as one long vector but as an image, with one axis corresponding to time and the other to frequency of spectral components.

Another important push, still ongoing, has been towards end-to-end deep learning speech recognition systems that completely remove the HMM. The first major breakthrough in this direction came from Graves *et al.* (2013) who trained a deep LSTM RNN (see section 10.10), using MAP inference over the frame-to-phoneme alignment, as in LeCun *et al.* (1998b) and in the CTC framework (Graves *et al.*, 2006; Graves, 2012). A deep RNN (Graves *et al.*, 2013) has state variables from several layers at each time step, giving the unfolded graph two kinds of depth: ordinary depth due to a stack of layers, and depth due to time unfolding. This work brought the phoneme error rate on TIMIT to a record low of 17.7%. See Pascanu *et al.* (2014a) and Chung *et al.* (2014) for other variants of deep RNNs, applied in other settings.

Another contemporary step toward end-to-end deep learning ASR is to let the system learn how to “align” the acoustic-level information with the phonetic-level information (Chorowski *et al.*, 2014; Lu *et al.*, 2015).

12.4 Natural Language Processing

Natural language processing (NLP) is the use of human languages, such as English or French, by a computer. Computer programs typically read and emit specialized languages designed to allow efficient and unambiguous parsing by simple programs. More naturally occurring languages are often ambiguous and defy formal description. Natural language processing includes applications such as machine translation, in which the learner must read a sentence in one human language and emit an equivalent sentence in another human language. Many NLP applications are based on language models that define a probability distribution over sequences of words, characters or bytes in a natural language.

As with the other applications discussed in this chapter, very generic neural network techniques can be successfully applied to natural language processing. However, to achieve excellent performance and to scale well to large applications, some domain-specific strategies become important. To build an efficient model of natural language, we must usually use techniques that are specialized for processing sequential data. In many cases, we choose to regard natural language as a sequence of words, rather than a sequence of individual characters or bytes. Because the total number of possible words is so large, word-based language models must operate on an extremely high-dimensional and sparse discrete space. Several strategies have been developed to make models of such a space efficient, both in a computational and in a statistical sense.

12.4.1 n-grams

A **language model** defines a probability distribution over sequences of tokens in a natural language. Depending on how the model is designed, a token may be a word, a character, or even a byte. Tokens are always discrete entities. The earliest successful language models were based on models of fixed-length sequences of tokens called n-grams. An n-gram is a sequence of n tokens.

Models based on n-grams define the conditional probability of the n-th token given the preceding $n-1$ tokens. The model uses products of these conditional distributions to define the probability distribution over longer sequences:

$$P(x_1, \ldots, x_\tau) = P(x_1, \ldots, x_{n-1}) \prod_{t=n}^{\tau} P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}). \qquad (12.5)$$

This decomposition is justified by the chain rule of probability. The probability distribution over the initial sequence $P(x_1, \ldots, x_{n-1})$ may be modeled by a different model with a smaller value of n.

Training n-gram models is straightforward because the maximum likelihood estimate can be computed simply by counting how many times each possible n gram occurs in the training set. Models based on n-grams have been the core building block of statistical language modeling for many decades (Jelinek and Mercer, 1980; Katz, 1987; Chen and Goodman, 1999).

For small values of n, models have particular names: **unigram** for n=1, **bigram** for n=2, and **trigram** for n=3. These names derive from the Latin prefixes for the corresponding numbers and the Greek suffix "-gram" denoting something that is written.

Usually we train both an n-gram model and an $n-1$ gram model simultaneously. This makes it easy to compute

$$P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{P_n(x_{t-n+1}, \ldots, x_t)}{P_{n-1}(x_{t-n+1}, \ldots, x_{t-1})} \tag{12.6}$$

simply by looking up two stored probabilities. For this to exactly reproduce inference in P_n, we must omit the final character from each sequence when we train P_{n-1}.

As an example, we demonstrate how a trigram model computes the probability of the sentence "THE DOG RAN AWAY." The first words of the sentence cannot be handled by the default formula based on conditional probability because there is no context at the beginning of the sentence. Instead, we must use the marginal probability over words at the start of the sentence. We thus evaluate $P_3(\texttt{THE DOG RAN})$. Finally, the last word may be predicted using the typical case, of using the conditional distribution $P(\texttt{AWAY} \mid \texttt{DOG RAN})$. Putting this together with equation 12.6, we obtain:

$$P(\texttt{THE DOG RAN AWAY}) = P_3(\texttt{THE DOG RAN})P_3(\texttt{DOG RAN AWAY})/P_2(\texttt{DOG RAN}). \tag{12.7}$$

A fundamental limitation of maximum likelihood for n-gram models is that P_n as estimated from training set counts is very likely to be zero in many cases, even though the tuple $(x_{t-n+1}, \ldots, x_t)$ may appear in the test set. This can cause two different kinds of catastrophic outcomes. When P_{n-1} is zero, the ratio is undefined, so the model does not even produce a sensible output. When P_{n-1} is non-zero but P_n is zero, the test log-likelihood is $-\infty$. To avoid such catastrophic outcomes, most n-gram models employ some form of **smoothing**. Smoothing techniques shift probability mass from the observed tuples to unobserved ones that are similar. See Chen and Goodman (1999) for a review and empirical comparisons. One basic technique consists of adding non-zero probability mass to all of the possible next symbol values. This method can be justified as Bayesian inference with a uniform or Dirichlet prior over the count parameters. Another very popular idea is to form a mixture model containing higher-order and lower-order n-gram models, with the higher-order models providing more capacity and the lower-order models being more likely to avoid counts of zero. **Back-off methods** look-up the lower-order n-grams if the frequency of the context $x_{t-1}, \ldots, x_{t-n+1}$ is too small to use the higher-order model. More formally, they estimate the distribution over x_t by using contexts $x_{t-n+k}, \ldots, x_{t-1}$, for increasing k, until a sufficiently reliable estimate is found.

Classical n-gram models are particularly vulnerable to the curse of dimensionality. There are $|\mathbb{V}|^n$ possible n-grams and $|\mathbb{V}|$ is often very large. Even with a massive training set and modest n, most n-grams will not occur in the training set. One way to view a classical n-gram model is that it is performing nearest-neighbor lookup. In other words, it can be viewed as a local non-parametric predictor, similar to k-nearest neighbors. The statistical problems facing these extremely local predictors are described in section 5.11.2. The problem for a language model is even more severe than usual, because any two different words have the same distance from each other in one-hot vector space. It is thus difficult to leverage much information from any "neighbors"—only training examples that repeat literally the same context are useful for local generalization. To overcome these problems, a language model must be able to share knowledge between one word and other semantically similar words.

To improve the statistical efficiency of n-gram models, **class-based language models** (Brown *et al.*, 1992; Ney and Kneser, 1993; Niesler *et al.*, 1998) introduce the notion of word categories and then share statistical strength between words that are in the same category. The idea is to use a clustering algorithm to partition the set of words into clusters or classes, based on their co-occurrence frequencies with other words. The model can then use word class IDs rather than individual word IDs to represent the context on the right side of the conditioning bar. Composite models combining word-based and class-based models via mixing or back-off are also possible. Although word classes provide a way to generalize between sequences in which some word is replaced by another of the same class, much information is lost in this representation.

12.4.2 Neural Language Models

Neural language models or NLMs are a class of language model designed to overcome the curse of dimensionality problem for modeling natural language sequences by using a distributed representation of words (Bengio *et al.*, 2001). Unlike class-based n-gram models, neural language models are able to recognize that two words are similar without losing the ability to encode each word as distinct from the other. Neural language models share statistical strength between one word (and its context) and other similar words and contexts. The distributed representation the model learns for each word enables this sharing by allowing the model to treat words that have features in common similarly. For example, if the word `dog` and the word `cat` map to representations that share many attributes, then sentences that contain the word `cat` can inform the predictions that will be made by the model for sentences that contain the word `dog`, and vice-versa. Because there are many such attributes, there are many ways in which generalization can happen, transferring information from each training sentence to an exponentially large number of semantically related sentences. The curse of dimensionality requires the model to generalize to a number of sentences that is exponential in the sentence length. The model counters this curse by relating each training sentence to an exponential number of similar sentences.

We sometimes call these word representations **word embeddings**. In this interpretation, we view the raw symbols as points in a space of dimension equal to the vocabulary size. The word representations embed those points in a feature space of lower dimension. In the original space, every word is represented by a one-hot vector, so every pair of words is at Euclidean distance $\sqrt{2}$ from each other. In the embedding space, words that frequently appear in similar contexts (or any pair of words sharing some "features" learned by the model) are close to each other. This often results in words with similar meanings being neighbors. Figure 12.3 zooms in on specific areas of a learned word embedding space to show how semantically similar words map to representations that are close to each other.

Neural networks in other domains also define embeddings. For example, a hidden layer of a convolutional network provides an "image embedding." Usually NLP practitioners are much more interested in this idea of embeddings because natural language does not originally lie in a real-valued vector space. The hidden layer has provided a more qualitatively dramatic change in the way the data is represented.

The basic idea of using distributed representations to improve models for natural language processing is not restricted to neural networks. It may also be used with graphical models that have distributed representations in the form of multiple latent variables (Mnih and Hinton, 2007).

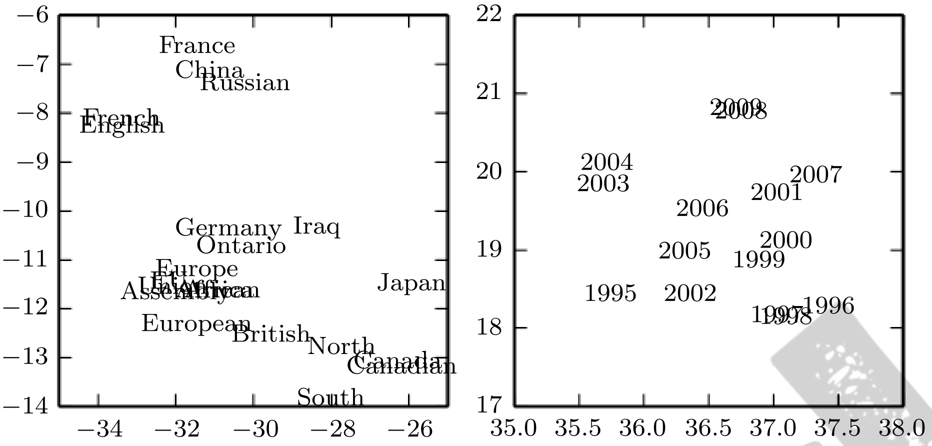

Figure 12.3: Two-dimensional visualizations of word embeddings obtained from a neural machine translation model (Bahdanau *et al.*, 2015), zooming in on specific areas where semantically related words have embedding vectors that are close to each other. Countries appear on the left and numbers on the right. Keep in mind that these embeddings are 2-D for the purpose of visualization. In real applications, embeddings typically have higher dimensionality and can simultaneously capture many kinds of similarity between words.

12.4.3 High-Dimensional Outputs

In many natural language applications, we often want our models to produce words (rather than characters) as the fundamental unit of the output. For large vocabularies, it can be very computationally expensive to represent an output distribution over the choice of a word, because the vocabulary size is large. In many applications, $\mathbb{V}$ contains hundreds of thousands of words. The naive approach to representing such a distribution is to apply an affine transformation from a hidden representation to the output space, then apply the softmax function. Suppose we have a vocabulary $\mathbb{V}$ with size $|\mathbb{V}|$. The weight matrix describing the linear component of this affine transformation is very large, because its output dimension is $|\mathbb{V}|$. This imposes a high memory cost to represent the matrix, and a high computational cost to multiply by it. Because the softmax is normalized across all $|\mathbb{V}|$ outputs, it is necessary to perform the full matrix multiplication at training time as well as test time—we cannot calculate only the dot product with the weight vector for the correct output. The high computational costs of the output layer thus arise both at training time (to compute the likelihood and its gradient) and at test time (to compute probabilities for all or selected words). For specialized loss functions, the gradient can be computed efficiently (Vincent *et al.*, 2015), but the standard cross-entropy loss applied to a traditional softmax output layer poses many difficulties.

Suppose that $\boldsymbol{h}$ is the top hidden layer used to predict the output probabilities $\hat{\boldsymbol{y}}$. If we parametrize the transformation from $\boldsymbol{h}$ to $\hat{\boldsymbol{y}}$ with learned weights $\boldsymbol{W}$ and learned biases $\boldsymbol{b}$, then the affine-softmax output layer performs the following computations:

$$a_i = b_i + \sum_j W_{ij} h_j \quad \forall i \in \{1, \ldots, |\mathbb{V}|\}, \tag{12.8}$$

$$\hat{y}_i = \frac{e^{a_i}}{\sum_{i'=1}^{|\mathbb{V}|} e^{a_{i'}}}. \tag{12.9}$$

If $\boldsymbol{h}$ contains n_h elements then the above operation is $O(|\mathbb{V}|n_h)$. With n_h in the thousands and $|\mathbb{V}|$ in the hundreds of thousands, this operation dominates the computation of most neural language models.

12.4.3.1 Use of a Short List

The first neural language models (Bengio *et al.*, 2001, 2003) dealt with the high cost of using a softmax over a large number of output words by limiting the vocabulary size to 10,000 or 20,000 words. Schwenk and Gauvain (2002) and Schwenk (2007) built upon this approach by splitting the vocabulary $\mathbb{V}$ into a **shortlist** $\mathbb{L}$ of most frequent words (handled by the neural net) and a tail $\mathbb{T} = \mathbb{V} \backslash \mathbb{L}$ of more rare words (handled by an n-gram model). To be able to combine the two predictions, the neural net also has to predict the probability that a word appearing after context C belongs to the tail list. This may be achieved by adding an extra sigmoid output unit to provide an estimate of $P(i \in \mathbb{T} \mid C)$. The extra output can then be used to achieve an estimate of the probability distribution over all words in $\mathbb{V}$ as follows:

$$\begin{aligned} P(y = i \mid C) =& 1_{i \in \mathbb{L}} P(y = i \mid C, i \in \mathbb{L})(1 - P(i \in \mathbb{T} \mid C)) \\ &+ 1_{i \in \mathbb{T}} P(y = i \mid C, i \in \mathbb{T}) P(i \in \mathbb{T} \mid C) \end{aligned} \tag{12.10}$$

where $P(y = i \mid C, i \in \mathbb{L})$ is provided by the neural language model and $P(y = i \mid C, i \in \mathbb{T})$ is provided by the n-gram model. With slight modification, this approach can also work using an extra output value in the neural language model's softmax layer, rather than a separate sigmoid unit.

An obvious disadvantage of the short list approach is that the potential generalization advantage of the neural language models is limited to the most frequent words, where, arguably, it is the least useful. This disadvantage has stimulated the exploration of alternative methods to deal with high-dimensional outputs, described below.

12.4.3.2 Hierarchical Softmax

A classical approach (Goodman, 2001) to reducing the computational burden of high-dimensional output layers over large vocabulary sets $\mathbb{V}$ is to decompose probabilities hierarchically. Instead of necessitating a number of computations proportional to $|\mathbb{V}|$ (and also proportional to the number of hidden units, n_h), the $|\mathbb{V}|$ factor can be reduced to as low as $\log |\mathbb{V}|$. Bengio (2002) and Morin and Bengio (2005) introduced this factorized approach to the context of neural language models.

One can think of this hierarchy as building categories of words, then categories of categories of words, then categories of categories of categories of words, etc. These nested categories form a tree, with words at the leaves. In a balanced tree, the tree has depth $O(\log |\mathbb{V}|)$. The probability of a choosing a word is given by the product of the probabilities of choosing the branch leading to that word at every node on a path from the root of the tree to the leaf containing the word. Figure 12.4 illustrates a simple example. Mnih and Hinton (2009) also describe how to use multiple paths to identify a single word in order to better model words that have multiple meanings. Computing the probability of a word then involves summation over all of the paths that lead to that word.

To predict the conditional probabilities required at each node of the tree, we typically use a logistic regression model at each node of the tree, and provide the same context C as input to all of these models. Because the correct output is encoded in the training set, we can use supervised learning to train the logistic regression models. This is typically done using a standard cross-entropy loss, corresponding to maximizing the log-likelihood of the correct sequence of decisions.

Because the output log-likelihood can be computed efficiently (as low as $\log |\mathbb{V}|$ rather than $|\mathbb{V}|$), its gradients may also be computed efficiently. This includes not only the gradient with respect to the output parameters but also the gradients with respect to the hidden layer activations.

It is possible but usually not practical to optimize the tree structure to minimize the expected number of computations. Tools from information theory specify how to choose the optimal binary code given the relative frequencies of the words. To do so, we could structure the tree so that the number of bits associated with a word is approximately equal to the logarithm of the frequency of that word. However, in

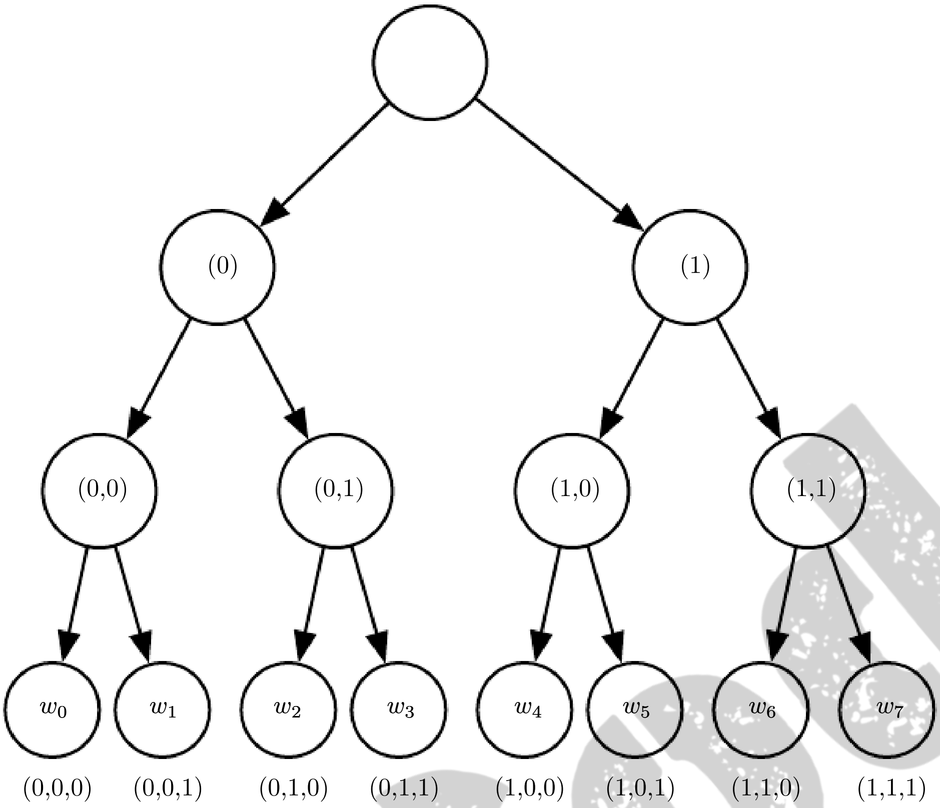

Figure 12.4: Illustration of a simple hierarchy of word categories, with 8 words $w_0, \ldots, w_7$ organized into a three level hierarchy. The leaves of the tree represent actual specific words. Internal nodes represent groups of words. Any node can be indexed by the sequence of binary decisions (0=left, 1=right) to reach the node from the root. Super-class (0) contains the classes (0, 0) and (0, 1), which respectively contain the sets of words $\{w_0, w_1\}$ and $\{w_2, w_3\}$, and similarly super-class (1) contains the classes (1,0) and (1,1), which respectively contain the words (w_4, w_5) and (w_6, w_7). If the tree is sufficiently balanced, the maximum depth (number of binary decisions) is on the order of the logarithm of the number of words $|\mathbb{V}|$: the choice of one out of $|\mathbb{V}|$ words can be obtained by doing $O(\log |\mathbb{V}|)$ operations (one for each of the nodes on the path from the root). In this example, computing the probability of a word y can be done by multiplying three probabilities, associated with the binary decisions to move left or right at each node on the path from the root to a node y. Let $b_i(y)$ be the i-th binary decision when traversing the tree towards the value y. The probability of sampling an output y decomposes into a product of conditional probabilities, using the chain rule for conditional probabilities, with each node indexed by the prefix of these bits. For example, node $(1, 0)$ corresponds to the prefix $(b_0(w_4) = 1, b_1(w_4) = 0)$, and the probability of w_4 can be decomposed as follows:

$$P(\mathrm{y} = w_4) = P(\mathrm{b}_0 = 1, \mathrm{b}_1 = 0, \mathrm{b}_2 = 0) \tag{12.11}$$

$$= P(\mathrm{b}_0 = 1)P(\mathrm{b}_1 = 0 \mid \mathrm{b}_0 = 1)P(\mathrm{b}_2 = 0 \mid \mathrm{b}_0 = 1, \mathrm{b}_1 = 0). \tag{12.12}$$

practice, the computational savings are typically not worth the effort because the computation of the output probabilities is only one part of the total computation in the neural language model. For example, suppose there are l fully connected hidden layers of width n_h. Let n_b be the weighted average of the number of bits required to identify a word, with the weighting given by the frequency of these words. In this example, the number of operations needed to compute the hidden activations grows as as $O(ln_h^2)$ while the output computations grow as $O(n_h n_b)$. As long as $n_b \leq ln_h$, we can reduce computation more by shrinking n_h than by shrinking n_b. Indeed, n_b is often small. Because the size of the vocabulary rarely exceeds a million words and $\log_2(10^6) \approx 20$, it is possible to reduce n_b to about 20, but n_h is often much larger, around 10^3 or more. Rather than carefully optimizing a tree with a branching factor of 2, one can instead define a tree with depth two and a branching factor of $\sqrt{|\mathbb{V}|}$. Such a tree corresponds to simply defining a set of mutually exclusive word classes. The simple approach based on a tree of depth two captures most of the computational benefit of the hierarchical strategy.

One question that remains somewhat open is how to best define these word classes, or how to define the word hierarchy in general. Early work used existing hierarchies (Morin and Bengio, 2005) but the hierarchy can also be learned, ideally jointly with the neural language model. Learning the hierarchy is difficult. An exact optimization of the log-likelihood appears intractable because the choice of a word hierarchy is a discrete one, not amenable to gradient-based optimization. However, one could use discrete optimization to approximately optimize the partition of words into word classes.

An important advantage of the hierarchical softmax is that it brings computational benefits both at training time and at test time, if at test time we want to compute the probability of specific words.

Of course, computing the probability of all $|\mathbb{V}|$ words will remain expensive even with the hierarchical softmax. Another important operation is selecting the most likely word in a given context. Unfortunately the tree structure does not provide an efficient and exact solution to this problem.

A disadvantage is that in practice the hierarchical softmax tends to give worse test results than sampling-based methods we will describe next. This may be due to a poor choice of word classes.

12.4.3.3 Importance Sampling

One way to speed up the training of neural language models is to avoid explicitly computing the contribution of the gradient from all of the words that do not appear in the next position. Every incorrect word should have low probability under the model. It can be computationally costly to enumerate all of these words. Instead, it is possible to sample only a subset of the words. Using the notation introduced in equation 12.8, the gradient can be written as follows:

$$\frac{\partial \log P(y \mid C)}{\partial \theta} = \frac{\partial \log \operatorname{softmax}_y(\boldsymbol{a})}{\partial \theta} \tag{12.13}$$

$$= \frac{\partial}{\partial \theta} \log \frac{e^{a_y}}{\sum_i e^{a_i}} \tag{12.14}$$

$$= \frac{\partial}{\partial \theta} (a_y - \log \sum_i e^{a_i}) \tag{12.15}$$

$$= \frac{\partial a_y}{\partial \theta} - \sum_i P(y = i \mid C) \frac{\partial a_i}{\partial \theta} \tag{12.16}$$

where $\boldsymbol{a}$ is the vector of pre-softmax activations (or scores), with one element per word. The first term is the **positive phase** term (pushing a_y up) while the second term is the **negative phase** term (pushing a_i down for all i, with weight $P(i \mid C)$. Since the negative phase term is an expectation, we can estimate it with a Monte Carlo sample. However, that would require sampling from the model itself. Sampling from the model requires computing $P(i \mid C)$ for all i in the vocabulary, which is precisely what we are trying to avoid.

Instead of sampling from the model, one can sample from another distribution, called the proposal distribution (denoted q), and use appropriate weights to correct for the bias introduced by sampling from the wrong distribution (Bengio and Sénécal, 2003; Bengio and Sénécal, 2008). This is an application of a more general technique called **importance sampling**, which will be described in more detail in section 17.2. Unfortunately, even exact importance sampling is not efficient because it requires computing weights p_i/q_i, where $p_i = P(i \mid C)$, which can only be computed if all the scores a_i are computed. The solution adopted for this application is called **biased importance sampling**, where the importance weights are normalized to sum to 1. When negative word n_i is sampled, the associated gradient is weighted by

$$w_i = \frac{p_{n_i}/q_{n_i}}{\sum_{j=1}^{N} p_{n_j}/q_{n_j}}. \tag{12.17}$$

These weights are used to give the appropriate importance to the m negative samples from q used to form the estimated negative phase contribution to the gradient:

$$\sum_{i=1}^{|\mathbb{V}|} P(i \mid C) \frac{\partial a_i}{\partial \theta} \approx \frac{1}{m} \sum_{i=1}^{m} w_i \frac{\partial a_{n_i}}{\partial \theta}. \tag{12.18}$$

A unigram or a bigram distribution works well as the proposal distribution q. It is easy to estimate the parameters of such a distribution from data. After estimating the parameters, it is also possible to sample from such a distribution very efficiently.

Importance sampling is not only useful for speeding up models with large softmax outputs. More generally, it is useful for accelerating training with large sparse output layers, where the output is a sparse vector rather than a 1-of-n choice. An example is a **bag of words**. A bag of words is a sparse vector $\boldsymbol{v}$ where v_i indicates the presence or absence of word i from the vocabulary in the document. Alternately, v_i can indicate the number of times that word i appears. Machine learning models that emit such sparse vectors can be expensive to train for a variety of reasons. Early in learning, the model may not actually choose to make the output truly sparse. Moreover, the loss function we use for training might most naturally be described in terms of comparing every element of the output to every element of the target. This means that it is not always clear that there is a computational benefit to using sparse outputs, because the model may choose to make the majority of the output non-zero and all of these non-zero values need to be compared to the corresponding training target, even if the training target is zero. Dauphin *et al.* (2011) demonstrated that such models can be accelerated using importance sampling. The efficient algorithm minimizes the loss reconstruction for the "positive words" (those that are non-zero in the target) and an equal number of "negative words." The negative words are chosen randomly, using a heuristic to sample words that are more likely to be mistaken. The bias introduced by this heuristic oversampling can then be corrected using importance weights.

In all of these cases, the computational complexity of gradient estimation for the output layer is reduced to be proportional to the number of negative samples rather than proportional to the size of the output vector.

12.4.3.4 Noise-Contrastive Estimation and Ranking Loss

Other approaches based on sampling have been proposed to reduce the computational cost of training neural language models with large vocabularies. An early example is the ranking loss proposed by Collobert and Weston (2008a), which views the output of the neural language model for each word as a score and tries to make the score of the correct word a_y be ranked high in comparison to the other scores a_i. The ranking loss proposed then is

$$L = \sum_i \max(0, 1 - a_y + a_i). \tag{12.19}$$

The gradient is zero for the i-th term if the score of the observed word, a_y, is greater than the score of the negative word a_i by a margin of 1. One issue with this criterion is that it does not provide estimated conditional probabilities, which are useful in some applications, including speech recognition and text generation (including conditional text generation tasks such as translation).

A more recently used training objective for neural language model is noise-contrastive estimation, which is introduced in section 18.6. This approach has been successfully applied to neural language models (Mnih and Teh, 2012; Mnih and Kavukcuoglu, 2013).

12.4.4 Combining Neural Language Models with n-grams

A major advantage of n-gram models over neural networks is that n-gram models achieve high model capacity (by storing the frequencies of very many tuples) while requiring very little computation to process an example (by looking up only a few tuples that match the current context). If we use hash tables or trees to access the counts, the computation used for n-grams is almost independent of capacity. In comparison, doubling a neural network's number of parameters typically also roughly doubles its computation time. Exceptions include models that avoid using all parameters on each pass. Embedding layers index only a single embedding in each pass, so we can increase the vocabulary size without increasing the computation time per example. Some other models, such as tiled convolutional networks, can add parameters while reducing the degree of parameter sharing in order to maintain the same amount of computation. However, typical neural network layers based on matrix multiplication use an amount of computation proportional to the number of parameters.

One easy way to add capacity is thus to combine both approaches in an ensemble consisting of a neural language model and an n-gram language model (Bengio *et al.*, 2001, 2003). As with any ensemble, this technique can reduce test error if the ensemble members make independent mistakes. The field of ensemble learning provides many ways of combining the ensemble members' predictions, including uniform weighting and weights chosen on a validation set. Mikolov *et al.* (2011a) extended the ensemble to include not just two models but a large array of models. It is also possible to pair a neural network with a maximum entropy model and train both jointly (Mikolov *et al.*, 2011b). This approach can be viewed as training a neural network with an extra set of inputs that are connected directly to the output, and not connected to any other part of the model. The extra inputs are indicators for the presence of particular n-grams in the input context, so these variables are very high-dimensional and very sparse. The increase in model capacity is huge—the new portion of the architecture contains up to $|sV|^n$ parameters—but the amount of added computation needed to process an input is minimal because the extra inputs are very sparse.

12.4.5 Neural Machine Translation

Machine translation is the task of reading a sentence in one natural language and emitting a sentence with the equivalent meaning in another language. Machine translation systems often involve many components. At a high level, there is often one component that proposes many candidate translations. Many of these translations will not be grammatical due to differences between the languages. For example, many languages put adjectives after nouns, so when translated to English directly they yield phrases such as “apple red.” The proposal mechanism suggests many variants of the suggested translation, ideally including “red apple.” A second component of the translation system, a language model, evaluates the proposed translations, and can score “red apple” as better than “apple red.”

The earliest use of neural networks for machine translation was to upgrade the language model of a translation system by using a neural language model (Schwenk *et al.*, 2006; Schwenk, 2010). Previously, most machine translation systems had used an n-gram model for this component. The n-gram based models used for machine translation include not just traditional back-off n-gram models (Jelinek and Mercer, 1980; Katz, 1987; Chen and Goodman, 1999) but also **maximum entropy language models** (Berger *et al.*, 1996), in which an affine-softmax layer predicts the next word given the presence of frequent n-grams in the context.

Traditional language models simply report the probability of a natural language sentence. Because machine translation involves producing an output sentence given an input sentence, it makes sense to extend the natural language model to be conditional. As described in section 6.2.1.1, it is straightforward to extend a model that defines a marginal distribution over some variable to define a conditional distribution over that variable given a context C, where C might be a single variable or a list of variables. Devlin *et al.* (2014) beat the state-of-the-art in some statistical machine translation benchmarks by using an MLP to score a phrase $\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_k$ in the target language given a phrase $\mathrm{s}_1, \mathrm{s}_2, \ldots, \mathrm{s}_n$ in the source language. The MLP estimates $P(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_k \mid \mathrm{s}_1, \mathrm{s}_2, \ldots, \mathrm{s}_n)$. The estimate formed by this MLP replaces the estimate provided by conditional n-gram models.

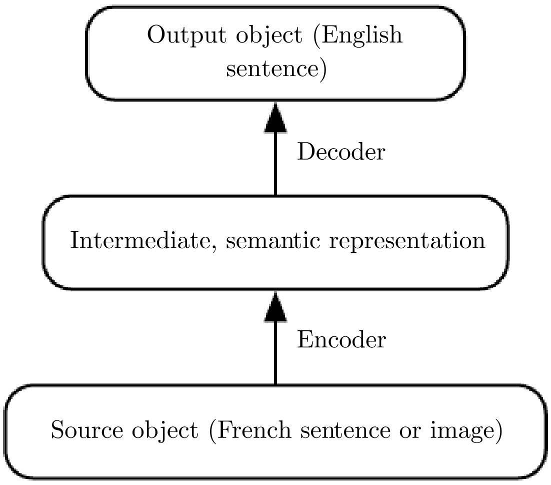
Output object (English sentence)
Decoder
Intermediate, semantic representation
Encoder
Source object (French sentence or image)

Figure 12.5: The encoder-decoder architecture to map back and forth between a surface representation (such as a sequence of words or an image) and a semantic representation. By using the output of an encoder of data from one modality (such as the encoder mapping from French sentences to hidden representations capturing the meaning of sentences) as the input to a decoder for another modality (such as the decoder mapping from hidden representations capturing the meaning of sentences to English), we can train systems that translate from one modality to another. This idea has been applied successfully not just to machine translation but also to caption generation from images.

A drawback of the MLP-based approach is that it requires the sequences to be preprocessed to be of fixed length. To make the translation more flexible, we would like to use a model that can accommodate variable length inputs and variable length outputs. An RNN provides this ability. Section 10.2.4 describes several ways of constructing an RNN that represents a conditional distribution over a sequence given some input, and section 10.4 describes how to accomplish this conditioning when the input is a sequence. In all cases, one model first reads the input sequence and emits a data structure that summarizes the input sequence. We call this summary the "context" C. The context C may be a list of vectors, or it may be a vector or tensor. The model that reads the input to produce C may be an RNN (Cho *et al.*, 2014a; Sutskever *et al.*, 2014; Jean *et al.*, 2014) or a convolutional network (Kalchbrenner and Blunsom, 2013). A second model, usually an RNN, then reads the context C and generates a sentence in the target language. This general idea of an encoder-decoder framework for machine translation is illustrated in figure 12.5.

In order to generate an entire sentence conditioned on the source sentence, the model must have a way to represent the entire source sentence. Earlier models were only able to represent individual words or phrases. From a representation learning point of view, it can be useful to learn a representation in which sentences that have the same meaning have similar representations regardless of whether they were written in the source language or the target language. This strategy was explored first using a combination of convolutions and RNNs (Kalchbrenner and Blunsom, 2013). Later work introduced the use of an RNN for scoring proposed translations (Cho *et al.*, 2014a) and for generating translated sentences (Sutskever *et al.*, 2014). Jean *et al.* (2014) scaled these models to larger vocabularies.

12.4.5.1 Using an Attention Mechanism and Aligning Pieces of Data

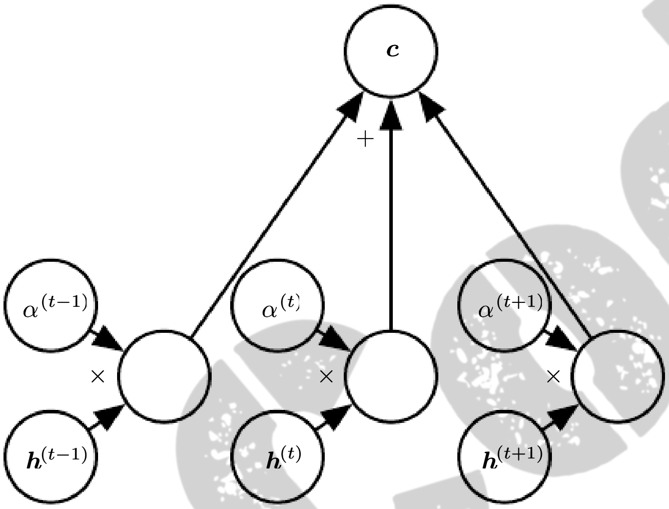

Figure 12.6: A modern attention mechanism, as introduced by Bahdanau *et al.* (2015), is essentially a weighted average. A context vector $\boldsymbol{c}$ is formed by taking a weighted average of feature vectors $\boldsymbol{h}^{(t)}$ with weights $\alpha^{(t)}$. In some applications, the feature vectors $\boldsymbol{h}$ are hidden units of a neural network, but they may also be raw input to the model. The weights $\alpha^{(t)}$ are produced by the model itself. They are usually values in the interval $[0, 1]$ and are intended to concentrate around just one $\boldsymbol{h}^{(t)}$ so that the weighted average approximates reading that one specific time step precisely. The weights $\alpha^{(t)}$ are usually produced by applying a softmax function to relevance scores emitted by another portion of the model. The attention mechanism is more expensive computationally than directly indexing the desired $\boldsymbol{h}^{(t)}$, but direct indexing cannot be trained with gradient descent. The attention mechanism based on weighted averages is a smooth, differentiable approximation that can be trained with existing optimization algorithms.

Using a fixed-size representation to capture all the semantic details of a very long sentence of say 60 words is very difficult. It can be achieved by training a sufficiently large RNN well enough and for long enough, as demonstrated by Cho *et al.* (2014a) and Sutskever *et al.* (2014). However, a more efficient approach is to read the whole sentence or paragraph (to get the context and the gist of what is being expressed), then produce the translated words one at a time, each time focusing on a different part of the input sentence in order to gather the semantic details that are required to produce the next output word. That is exactly the idea that Bahdanau *et al.* (2015) first introduced. The attention mechanism used to focus on specific parts of the input sequence at each time step is illustrated in figure 12.6.

We can think of an attention-based system as having three components:

1. A process that "*reads*" raw data (such as source words in a source sentence), and converts them into distributed representations, with one feature vector associated with each word position.

2. A list of feature vectors storing the output of the reader. This can be understood as a "*memory*" containing a sequence of facts, which can be retrieved later, not necessarily in the same order, without having to visit all of them.

3. A process that "*exploits*" the content of the memory to sequentially perform a task, at each time step having the ability put attention on the content of one memory element (or a few, with a different weight).

The third component generates the translated sentence.

When words in a sentence written in one language are aligned with corresponding words in a translated sentence in another language, it becomes possible to relate the corresponding word embeddings. Earlier work showed that one could learn a kind of translation matrix relating the word embeddings in one language with the word embeddings in another (Kočiský *et al.*, 2014), yielding lower alignment error rates than traditional approaches based on the frequency counts in the phrase table. There is even earlier work on learning cross-lingual word vectors (Klementiev *et al.*, 2012). Many extensions to this approach are possible. For example, more efficient cross-lingual alignment (Gouws *et al.*, 2014) allows training on larger datasets.

12.4.6 Historical Perspective

The idea of distributed representations for symbols was introduced by Rumelhart *et al.* (1986a) in one of the first explorations of back-propagation, with symbols corresponding to the identity of family members and the neural network capturing the relationships between family members, with training examples forming triplets such as (Colin, Mother, Victoria). The first layer of the neural network learned a representation of each family member. For example, the features for Colin might represent which family tree Colin was in, what branch of that tree he was in, what generation he was from, etc. One can think of the neural network as computing learned rules relating these attributes together in order to obtain the desired predictions. The model can then make predictions such as inferring who is the mother of Colin.

The idea of forming an embedding for a symbol was extended to the idea of an embedding for a word by Deerwester *et al.* (1990). These embeddings were learned using the SVD. Later, embeddings would be learned by neural networks.

The history of natural language processing is marked by transitions in the popularity of different ways of representing the input to the model. Following this early work on symbols or words, some of the earliest applications of neural networks to NLP (Miikkulainen and Dyer, 1991; Schmidhuber, 1996) represented the input as a sequence of characters.

Bengio *et al.* (2001) returned the focus to modeling words and introduced neural language models, which produce interpretable word embeddings. These neural models have scaled up from defining representations of a small set of symbols in the 1980s to millions of words (including proper nouns and misspellings) in modern applications. This computational scaling effort led to the invention of the techniques described above in section 12.4.3.

Initially, the use of words as the fundamental units of language models yielded improved language modeling performance (Bengio *et al.*, 2001). To this day, new techniques continually push both character-based models (Sutskever *et al.*, 2011) and word-based models forward, with recent work (Gillick *et al.*, 2015) even modeling individual bytes of Unicode characters.

The ideas behind neural language models have been extended into several natural language processing applications, such as parsing (Henderson, 2003, 2004; Collobert, 2011), part-of-speech tagging, semantic role labeling, chunking, etc, sometimes using a single multi-task learning architecture (Collobert and Weston, 2008a; Collobert *et al.*, 2011a) in which the word embeddings are shared across tasks.

Two-dimensional visualizations of embeddings became a popular tool for analyzing language models following the development of the t-SNE dimensionality reduction algorithm (van der Maaten and Hinton, 2008) and its high-profile application to visualization word embeddings by Joseph Turian in 2009.

12.5 Other Applications

In this section we cover a few other types of applications of deep learning that are different from the standard object recognition, speech recognition and natural language processing tasks discussed above. Part III of this book will expand that scope even further to tasks that remain primarily research areas.

12.5.1 Recommender Systems

One of the major families of applications of machine learning in the information technology sector is the ability to make recommendations of items to potential users or customers. Two major types of applications can be distinguished: online advertising and item recommendations (often these recommendations are still for the purpose of selling a product). Both rely on predicting the association between a user and an item, either to predict the probability of some action (the user buying the product, or some proxy for this action) or the expected gain (which may depend on the value of the product) if an ad is shown or a recommendation is made regarding that product to that user. The internet is currently financed in great part by various forms of online advertising. There are major parts of the economy that rely on online shopping. Companies including Amazon and eBay use machine learning, including deep learning, for their product recommendations. Sometimes, the items are not products that are actually for sale. Examples include selecting posts to display on social network news feeds, recommending movies to watch, recommending jokes, recommending advice from experts, matching players for video games, or matching people in dating services.

Often, this association problem is handled like a supervised learning problem: given some information about the item and about the user, predict the proxy of interest (user clicks on ad, user enters a rating, user clicks on a “like” button, user buys product, user spends some amount of money on the product, user spends time visiting a page for the product, etc). This often ends up being either a regression problem (predicting some conditional expected value) or a probabilistic classification problem (predicting the conditional probability of some discrete event).

The early work on recommender systems relied on minimal information as inputs for these predictions: the user ID and the item ID. In this context, the only way to generalize is to rely on the similarity between the patterns of values of the target variable for different users or for different items. Suppose that user 1 and user 2 both like items A, B and C. From this, we may infer that user 1 and user 2 have similar tastes. If user 1 likes item D, then this should be a strong cue that user 2 will also like D. Algorithms based on this principle come under the name of **collaborative filtering**. Both non-parametric approaches (such as nearest-neighbor methods based on the estimated similarity between patterns of preferences) and parametric methods are possible. Parametric methods often rely on learning a distributed representation (also called an embedding) for each user and for each item. Bilinear prediction of the target variable (such as a rating) is a simple parametric method that is highly successful and often found as a component of state-of-the-art systems. The prediction is obtained by the dot product between the user embedding and the item embedding (possibly corrected by constants that depend only on either the user ID or the item ID). Let $\hat{\boldsymbol{R}}$ be the matrix containing our predictions, $\boldsymbol{A}$ a matrix with user embeddings in its rows and $\boldsymbol{B}$ a matrix with item embeddings in its columns. Let $\boldsymbol{b}$ and $\boldsymbol{c}$ be vectors that contain respectively a kind of bias for each user (representing how grumpy or positive that user is in general) and for each item (representing its general popularity). The bilinear prediction is thus obtained as follows:

$$\hat{R}_{u,i} = b_u + c_i + \sum_j A_{u,j} B_{j,i}. \tag{12.20}$$

Typically one wants to minimize the squared error between predicted ratings $\hat{R}_{u,i}$ and actual ratings $R_{u,i}$. User embeddings and item embeddings can then be conveniently visualized when they are first reduced to a low dimension (two or three), or they can be used to compare users or items against each other, just like word embeddings. One way to obtain these embeddings is by performing a singular value decomposition of the matrix $\boldsymbol{R}$ of actual targets (such as ratings). This corresponds to factorizing $\boldsymbol{R} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}'$ (or a normalized variant) into the product of two factors, the lower rank matrices $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}$ and $\boldsymbol{B} = \boldsymbol{V}'$. One problem with the SVD is that it treats the missing entries in an arbitrary way, as if they corresponded to a target value of 0. Instead we would like to avoid paying any cost for the predictions made on missing entries. Fortunately, the sum of squared errors on the observed ratings can also be easily minimized by gradient-based optimization. The SVD and the bilinear prediction of equation 12.20 both performed very well in the competition for the Netflix prize (Bennett and Lanning, 2007), aiming at predicting ratings for films, based only on previous ratings by a large set of anonymous users. Many machine learning experts participated in this competition, which took place between 2006 and 2009. It raised the level of research in recommender systems using advanced machine learning and yielded improvements in recommender systems. Even though it did not win by itself, the simple bilinear prediction or SVD was a component of the ensemble models presented by most of the competitors, including the winners (Töscher *et al.*, 2009; Koren, 2009).

Beyond these bilinear models with distributed representations, one of the first uses of neural networks for collaborative filtering is based on the RBM undirected probabilistic model (Salakhutdinov *et al.*, 2007). RBMs were an important element of the ensemble of methods that won the Netflix competition (Töscher *et al.*, 2009; Koren, 2009). More advanced variants on the idea of factorizing the ratings matrix have also been explored in the neural networks community (Salakhutdinov and Mnih, 2008).

However, there is a basic limitation of collaborative filtering systems: when a new item or a new user is introduced, its lack of rating history means that there is no way to evaluate its similarity with other items or users (respectively), or the degree of association between, say, that new user and existing items. This is called the problem of cold-start recommendations. A general way of solving the cold-start recommendation problem is to introduce extra information about the individual users and items. For example, this extra information could be user profile information or features of each item. Systems that use such information are called **content-based recommender systems**. The mapping from a rich set of user features or item features to an embedding can be learned through a deep learning architecture (Huang *et al.*, 2013; Elkahky *et al.*, 2015).

Specialized deep learning architectures such as convolutional networks have also been applied to learn to extract features from rich content such as from musical audio tracks, for music recommendation (van den Oörd *et al.*, 2013). In that work, the convolutional net takes acoustic features as input and computes an embedding for the associated song. The dot product between this song embedding and the embedding for a user is then used to predict whether a user will listen to the song.

12.5.1.1 Exploration Versus Exploitation

When making recommendations to users, an issue arises that goes beyond ordinary supervised learning and into the realm of reinforcement learning. Many recommendation problems are most accurately described theoretically as **contextual bandits** (Langford and Zhang, 2008; Lu *et al.*, 2010). The issue is that when we use the recommendation system to collect data, we get a biased and incomplete view of the preferences of users: we only see the responses of users to the items they were recommended and not to the other items. In addition, in some cases we may not get any information on users for whom no recommendation has been made (for example, with ad auctions, it may be that the price proposed for an ad was below a minimum price threshold, or does not win the auction, so the ad is not shown at all). More importantly, we get no information about what outcome would have resulted from recommending any of the other items. This would be like training a classifier by picking one class $\hat{y}$ for each training example $\boldsymbol{x}$ (typically the class with the highest probability according to the model) and then only getting as feedback whether this was the correct class or not. Clearly, each example conveys less information than in the supervised case where the true label y is directly accessible, so more examples are necessary. Worse, if we are not careful, we could end up with a system that continues picking the wrong decisions even as more and more data is collected, because the correct decision initially had a very low probability: until the learner picks that correct decision, it does not learn about the correct decision. This is similar to the situation in reinforcement learning where only the reward for the selected action is observed. In general, reinforcement learning can involve a sequence of many actions and many rewards. The bandits scenario is a special case of reinforcement learning, in which the learner takes only a single action and receives a single reward. The bandit problem is easier in the sense that the learner knows which reward is associated with which action. In the general reinforcement learning scenario, a high reward or a low reward might have been caused by a recent action or by an action in the distant past. The term **contextual** bandits refers to the case where the action is taken in the context of some input variable that can inform the decision. For example, we at least know the user identity, and we want to pick an item. The mapping from context to action is also called a **policy**. The feedback loop between the learner and the data distribution (which now depends on the actions of the learner) is a central research issue in the reinforcement learning and bandits literature.

Reinforcement learning requires choosing a tradeoff between **exploration** and **exploitation**. Exploitation refers to taking actions that come from the current, best version of the learned policy—actions that we know will achieve a high reward. Exploration refers to taking actions specifically in order to obtain more training data. If we know that given context $\boldsymbol{x}$, action a gives us a reward of 1, we do not know whether that is the best possible reward. We may want to exploit our current policy and continue taking action a in order to be relatively sure of obtaining a reward of 1. However, we may also want to explore by trying action a'. We do not know what will happen if we try action a'. We hope to get a reward of 2, but we run the risk of getting a reward of 0. Either way, we at least gain some knowledge.

Exploration can be implemented in many ways, ranging from occasionally taking random actions intended to cover the entire space of possible actions, to model-based approaches that compute a choice of action based on its expected reward and the model's amount of uncertainty about that reward.

Many factors determine the extent to which we prefer exploration or exploitation. One of the most prominent factors is the time scale we are interested in. If the agent has only a short amount of time to accrue reward, then we prefer more exploitation. If the agent has a long time to accrue reward, then we begin with more exploration so that future actions can be planned more effectively with more knowledge. As time progresses and our learned policy improves, we move toward more exploitation.

Supervised learning has no tradeoff between exploration and exploitation because the supervision signal always specifies which output is correct for each input. There is no need to try out different outputs to determine if one is better than the model's current output—we always know that the label is the best output.

Another difficulty arising in the context of reinforcement learning, besides the exploration-exploitation trade-off, is the difficulty of evaluating and comparing different policies. Reinforcement learning involves interaction between the learner and the environment. This feedback loop means that it is not straightforward to evaluate the learner's performance using a fixed set of test set input values. The policy itself determines which inputs will be seen. Dudik *et al.* (2011) present techniques for evaluating contextual bandits.

12.5.2 Knowledge Representation, Reasoning and Question Answering

Deep learning approaches have been very successful in language modeling, machine translation and natural language processing due to the use of embeddings for symbols (Rumelhart *et al.*, 1986a) and words (Deerwester *et al.*, 1990; Bengio *et al.*, 2001). These embeddings represent semantic knowledge about individual words and concepts. A research frontier is to develop embeddings for phrases and for relations between words and facts. Search engines already use machine learning for this purpose but much more remains to be done to improve these more advanced representations.

12.5.2.1 Knowledge, Relations and Question Answering

One interesting research direction is determining how distributed representations can be trained to capture the **relations** between two entities. These relations allow us to formalize facts about objects and how objects interact with each other.

In mathematics, a **binary relation** is a set of ordered pairs of objects. Pairs that are in the set are said to have the relation while those who are not in the set do not. For example, we can define the relation “is less than” on the set of entities $\{1, 2, 3\}$ by defining the set of ordered pairs $\mathbb{S} = \{(1,2), (1,3), (2,3)\}$. Once this relation is defined, we can use it like a verb. Because $(1,2) \in \mathbb{S}$, we say that 1 is less than 2. Because $(2,1) \notin \mathbb{S}$, we can not say that 2 is less than 1. Of course, the entities that are related to one another need not be numbers. We could define a relation `is_a_type_of` containing tuples like (`dog`, `mammal`).

In the context of AI, we think of a relation as a sentence in a syntactically simple and highly structured language. The relation plays the role of a verb, while two arguments to the relation play the role of its subject and object. These sentences take the form of a triplet of tokens

$$(\text{subject}, \text{verb}, \text{object}) \tag{12.21}$$

with values

$$(\text{entity}_i, \text{relation}_j, \text{entity}_k). \tag{12.22}$$

We can also define an **attribute**, a concept analogous to a relation, but taking only one argument:

$$(\text{entity}_i, \text{attribute}_j). \tag{12.23}$$

For example, we could define the `has_fur` attribute, and apply it to entities like `dog`.

Many applications require representing relations and reasoning about them. How should we best do this within the context of neural networks?

Machine learning models of course require training data. We can infer relations between entities from training datasets consisting of unstructured natural language. There are also structured databases that identify relations explicitly. A common structure for these databases is the **relational database**, which stores this same kind of information, albeit not formatted as three token sentences. When a database is intended to convey commonsense knowledge about everyday life or expert knowledge about an application area to an artificial intelligence system, we call the database a **knowledge base**. Knowledge bases range from general ones like `Freebase`, `OpenCyc`, `WordNet`, or `Wikibase`,[1] etc. to more specialized knowledge bases, like `GeneOntology`.[2] Representations for entities and relations can be learned by considering each triplet in a knowledge base as a training example and maximizing a training objective that captures their joint distribution (Bordes *et al.*, 2013a).

[1] Respectively available from these web sites: `freebase.com`, `cyc.com/opencyc`, `wordnet.princeton.edu`, `wikiba.se`

[2] `geneontology.org`

In addition to training data, we also need to define a model family to train. A common approach is to extend neural language models to model entities and relations. Neural language models learn a vector that provides a distributed representation of each word. They also learn about interactions between words, such as which word is likely to come after a sequence of words, by learning functions of these vectors. We can extend this approach to entities and relations by learning an embedding vector for each relation. In fact, the parallel between modeling language and modeling knowledge encoded as relations is so close that researchers have trained representations of such entities by using *both* knowledge bases *and* natural language sentences (Bordes *et al.*, 2011, 2012; Wang *et al.*, 2014a) or combining data from multiple relational databases (Bordes *et al.*, 2013b). Many possibilities exist for the particular parametrization associated with such a model. Early work on learning about relations between entities (Paccanaro and Hinton, 2000) posited highly constrained parametric forms ("linear relational embeddings"), often using a different form of representation for the relation than for the entities. For example, Paccanaro and Hinton (2000) and Bordes *et al.* (2011) used vectors for entities and matrices for relations, with the idea that a relation acts like an operator on entities. Alternatively, relations can be considered as any other entity (Bordes *et al.*, 2012), allowing us to make statements about relations, but more flexibility is put in the machinery that combines them in order to model their joint distribution.

A practical short-term application of such models is **link prediction**: predicting missing arcs in the knowledge graph. This is a form of generalization to new facts, based on old facts. Most of the knowledge bases that currently exist have been constructed through manual labor, which tends to leave many and probably the majority of true relations absent from the knowledge base. See Wang *et al.* (2014b), Lin *et al.* (2015) and Garcia-Duran *et al.* (2015) for examples of such an application.

Evaluating the performance of a model on a link prediction task is difficult because we have only a dataset of positive examples (facts that are known to be true). If the model proposes a fact that is not in the dataset, we are unsure whether the model has made a mistake or discovered a new, previously unknown fact. The metrics are thus somewhat imprecise and are based on testing how the model ranks a held-out of set of known true positive facts compared to other facts that are less likely to be true. A common way to construct interesting examples that are probably negative (facts that are probably false) is to begin with a true fact and create corrupted versions of that fact, for example by replacing one entity in the relation with a different entity selected at random. The popular precision at 10% metric counts how many times the model ranks a "correct" fact among the top 10% of all corrupted versions of that fact.

Another application of knowledge bases and distributed representations for them is **word-sense disambiguation** (Navigli and Velardi, 2005; Bordes *et al.*, 2012), which is the task of deciding which of the senses of a word is the appropriate one, in some context.

Eventually, knowledge of relations combined with a reasoning process and understanding of natural language could allow us to build a general question answering system. A general question answering system must be able to process input information and remember important facts, organized in a way that enables it to retrieve and reason about them later. This remains a difficult open problem which can only be solved in restricted "toy" environments. Currently, the best approach to remembering and retrieving specific declarative facts is to use an explicit memory mechanism, as described in section 10.12. Memory networks were first proposed to solve a toy question answering task (Weston *et al.*, 2014). Kumar *et al.* (2015) have proposed an extension that uses GRU recurrent nets to read the input into the memory and to produce the answer given the contents of the memory.

Deep learning has been applied to many other applications besides the ones described here, and will surely be applied to even more after this writing. It would be impossible to describe anything remotely resembling a comprehensive coverage of such a topic. This survey provides a representative sample of what is possible as of this writing.

This concludes part II, which has described modern practices involving deep networks, comprising all of the most successful methods. Generally speaking, these methods involve using the gradient of a cost function to find the parameters of a model that approximates some desired function. With enough training data, this approach is extremely powerful. We now turn to part III, in which we step into the territory of research—methods that are designed to work with less training data or to perform a greater variety of tasks, where the challenges are more difficult and not as close to being solved as the situations we have described so far.