Chapter 10

Sequence Modeling: Recurrent and Recursive Nets

One of the early ideas found in machine learning and statistical models of the 80's is that of *sharing parameters*[1] across different parts of a model, allowing to *extend and apply the model* to examples of different forms and generalize across them, e.g. with examples of different lengths, in the case of sequential data. This can be found in hidden Markov models (HMMs) (Rabiner and Juang, 1986), which were the dominant technique for speech recognition for about 30 years. These models of sequences are described a bit more in Section 10.9.3 and involve parameters, such as the state-to-state transition matrix $P(\boldsymbol{s}_t \mid \boldsymbol{s}_{t-1})$, which are re-used for every time step t, i.e., the above probability depends only on the value of $\boldsymbol{s}_t$ and $\boldsymbol{s}_{t-1}$ but not on t as such. This allows one to model variable length sequences, whereas if we had specific parameters for each value of t, we could not generalize to sequence lengths not seen during training, nor share statistical strength across different sequence lengths and across different positions in time. Such sharing is particularly important when, like in speech, the input sequence can be stretched non-linearly, i.e., some parts (like vowels) may last longer in different examples. It means that the absolute time step at which an event occurs is meaningless: it only makes sense to consider the event in some context that somehow captures what has happened before. This sharing across time can also be found in a *recurrent neural network* (Rumelhart *et al.*, 1986c) or RNN [2]: the same weights are used for different instances of the artificial neurons at different time steps, allowing us to apply the network to input sequences of different lengths. This idea is made more

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

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

explicit in the early work on *time-delay neural networks* (Lang and Hinton, 1988; Waibel *et al.*, 1989), where a fully connected network is replaced by one with local connections that are shared across different temporal instances of the hidden units. Such networks are among the ancestors of *convolutional neural networks*, covered in more detail in Section 9. Recurrent nets are covered below in Section 10.2. As shown in Section 10.1 below, the flow graph (a notion introduced in Section 6.4 in the case of MLPs) associated with a recurrent network is structured like a chain, as explained next. Recurrent neural networks have been generalized into *recursive neural networks*, in which the structure can be more general, i.e., and it is typically viewed as a tree. Recursive neural networks are discussed in more detail in Section 10.6. For a good textbook on RNNs, see Graves (2012).

10.1 Unfolding Flow Graphs and Sharing Parameters

A flow graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss. Please refer to Section 6.4 for a general introduction. In this section we explain the idea of *unfolding* a recursive or recurrent computation into a flow graph that has a repetitive structure, typically corresponding to a chain of events.

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

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

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

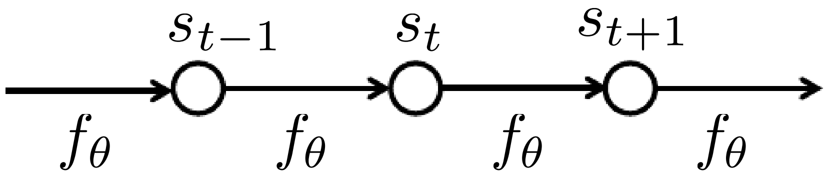

Figure 10.1: Classical dynamical system equation 10.1 illustrated as an unfolded flow graph. Each node represents the state at some time t and function f_θ maps the state at t to the state at $t+1$. The same parameters (the same function f_θ) is used for all time steps.

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

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

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

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

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

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

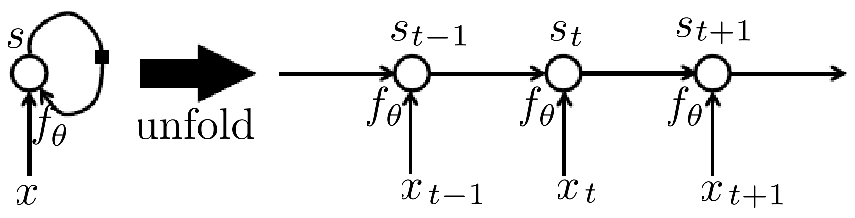

Figure 10.2: Left: input processing part of a recurrent neural network, seen as a circuit. The black square indicates a delay of 1 time step. Right: the same seen as an unfolded flow graph, where each node is now associated with one particular time instance.

Equation 10.2 can be drawn in two different ways. One is in a way that is inspired by how a physical implementation (such as a real neural network) might look like, i.e., like a circuit which operates in real time, as in the left of Figure 10.2. The other is as a flow graph, in which the computations occurring at different time steps in the circuit are unfolded as different nodes of the flow graph, as in the right of Figure 10.2. What we call *unfolding* is the operation that maps a circuit as in the left side of the figure to a flow graph with repeated pieces as in the right side. Note how the unfolded graph now has a size that depends on the sequence length. The black square indicates a delay of 1 time step on the recurrent connection, from the state at time t to the state at time $t + 1$.

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

10.2 Recurrent Neural Networks

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

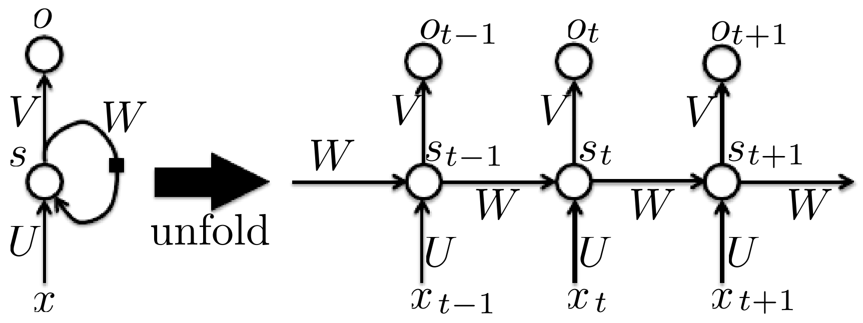

Figure 10.3: Left: vanilla recurrent network circuit with hidden-to-hidden recurrence, seen as a circuit, with weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$ for the three different kinds of connections (input-to-hidden, hidden-to-output, and hidden-to-hidden, respectively). Each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

Some of the early circuit designs for recurrent neural networks are illustrated in Figures 10.3, 10.4 and 10.6. Figure 10.3 shows the vanilla recurrent network whose equations are laid down below in Eq. 10.4, and which has been shown to be a universal approximation machine for sequences, i.e., able to implement a Turing machine (Siegelmann and Sontag, 1991; Siegelmann, 1995; Hyotyniemi, 1996). On the other hand, the network with *output recurrence* shown in TODO finish the above sentence

The vanilla recurrent network of Figure 10.3 corresponds to the following forward propagation equations, if we assume that hyperbolic tangent non-linearities

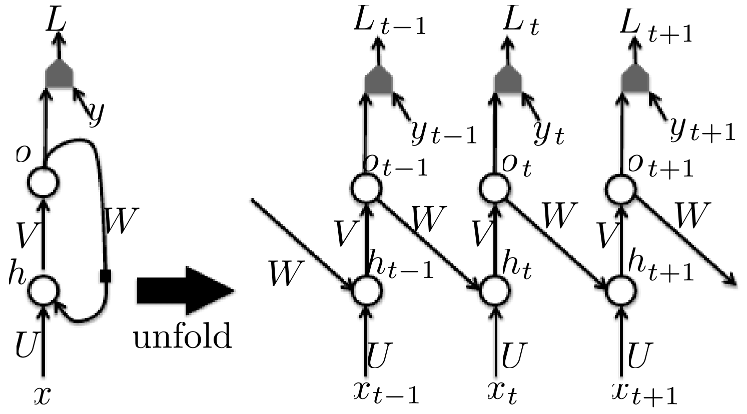

Figure 10.4: Left: RNN circuit whose recurrence is only through the output. Right: computational flow graph unfolded in time. At each t, the input is $\boldsymbol{x}_t$, the hidden layer activations $\boldsymbol{h}_t$, the output $\boldsymbol{o}_t$, the target y_t and the loss L_t. Such an RNN is less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

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

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

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

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

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

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

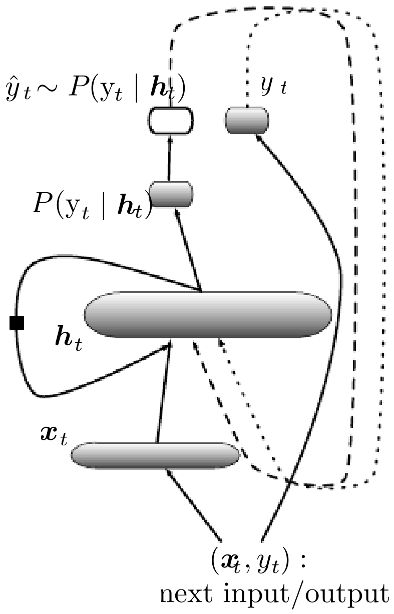
$\hat{y}_t \sim P(\mathrm{y}_t \mid \boldsymbol{h}_t)$
y_t
$P(\mathrm{y}_t \mid \boldsymbol{h}_t)$
$\boldsymbol{h}_t$
$\boldsymbol{x}_t$
$(\boldsymbol{x}_t, y_t)$:
next input/output training pair

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

Teacher forcing is the training process in which the fed back inputs are not the predicted outputs but the targets themselves, as illustrated in Figure 10.5. The disadvantage of strict teacher forcing is that if the network is going to be later used in an *open-loop* mode, i.e., with the network outputs (or samples from the output distribution) fed back as input, then the kind of inputs that the network will have seen during training could be quite different from the kind of inputs that it will see at test time when the network is run in generative mode, potentially yielding very poor generalizations. One way to mitigate this problem is to train with both teacher-forced inputs and with free-running inputs, e.g., predicting the correct target a number of steps in the future through the unfolded recurrent output-to-input paths. In this way, the network can learn to take into account input conditions (such as those it generates itself in the free-running mode) not seen during training and how to map the state back towards one that will make the network generate proper outputs after a few steps. Another approach (Bengio *et al.*, 2015) to mitigate the gap between the generative mode of RNNs and how they are trained (with teacher forcing, i.e., maximum likelihood) randomly chooses to use generated values or actual data values as input, and exploits a curriculum learning strategy to gradually use more of the generated values as input.

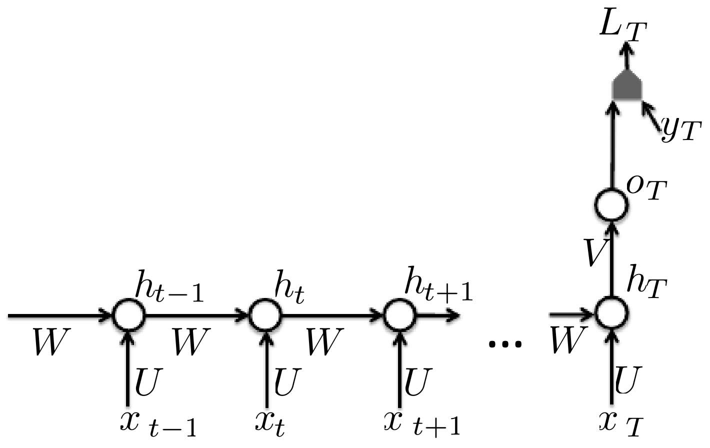

Figure 10.6: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (like in the figure) or the gradient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

10.2.1 Computing the Gradient in a Recurrent Neural Network

Using the generalized back-propagation algorithm (for arbitrary flow graphs) introduced in Section 6.4, one can obtain the so-called **Back-Propagation Through Time** (BPTT) algorithm. Once we know how to compute gradients, we can in principle apply any of the general-purpose gradient-based techniques to train an RNN. These general-purpose techniques were introduced in Section 4.3 and developed in greater depth in Chapter 8.

Let us thus work out how to compute gradients by BPTT for the RNN equations above (Eqs. 10.4 and 10.5). The nodes of our flow graph will be the sequence of $\boldsymbol{x}_t$'s, $\boldsymbol{s}_t$'s, $\boldsymbol{o}_t$'s, L_t's, and the parameters $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$, $\boldsymbol{b}$, $\boldsymbol{c}$. For each node a we need to compute the gradient $\nabla_{\boldsymbol{a}} L$ recursively, based on the gradient computed at nodes that follow it in the graph. We start the recursion with the nodes immediately preceding the final loss

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

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

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

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

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

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

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

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

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

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

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

10.2.2 Recurrent Networks as Generative Directed Acyclic Models

Up to here, we have not clearly stated what the losses L_t associated with the outputs o_t of a recurrent net should be. It is because there are many possible ways in which RNNs can be used. In this section, we consider the most common case where the RNN models a probability distribution over a sequence of observations.

When we consider a predictive log-likelihood training objective such as Eq. 10.5, we are training the RNN to estimate the conditional distribution of the next sequence element y_t given the past inputs $\boldsymbol{x}_s$ and targets y_s (for $s < t$). As we show below, this corresponds to viewing the RNN as a directed graphical model, a notion introduced in Section 3.14. In this case, the set of random variables of interest is the sequence of y_t's (given the sequence of $\boldsymbol{x}_t$'s), and we are modeling the joint probability of the y_t's given the $\boldsymbol{x}_t$'s.

To keep things simple for starters, let us assume that there is no conditioning input sequence in addition to the output sequence, i.e., that the target output at the next time step is also the next input. The random variable of interest is thus the sequence of vectors $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)$, and we parametrize the joint distribution of these vectors via

$$P(\boldsymbol{X}) = P(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T) = \prod_{t=1}^{T} P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_1) \qquad (10.6)$$

using the chain rule of conditional probabilities (Section 3.6), and where the right-hand side of the bar is empty for $t = 1$, of course. Hence the negative log-likelihood of $\boldsymbol{X}$ according to such a model is

$$L = \sum_t L_t$$

where

$$L_t = -\log P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_1).$$

In general directed graphical models, $\boldsymbol{x}_t$ can be predicted using only a subset of its predecessors $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{t-1})$. However, for RNNs, the graphical model is generally fully connected, not removing any dependency a priori. This can be achieved efficiently through the recurrent parametrization, such as in Eq. 10.2, since $\boldsymbol{s}_t$

is trained to summarize whatever is required from the whole previous sequence (Eq. 10.3).

Hence, instead of cutting statistical complexity by removing arcs in the directed graphical model for $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$, as is done in most of the work on directed graphical models, the core idea of recurrent networks is that we introduce a state variable which decouples all the past and future observations, but we make that state variable a function of the past, through the recurrence, Eq. 10.2. Consequently, the number of parameters required to parametrize $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t-2}, \ldots, \boldsymbol{x}_1)$ does not grow exponentially with t (as it would if we parametrized that probability by a straight probability table, when the data is discrete) but remains constant with t. It only grows with the dimension of the state $\boldsymbol{s}_t$. The price to be paid for that great advantage is that *optimizing* the parameters may be more difficult, as discussed below in Section 10.8. The decomposition of the likelihood thus becomes:

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

where

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

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

The above decomposition of the joint probability of a sequence of variables into ordered conditionals precisely corresponds to the sequence of computations performed by an RNN. The *target* to be predicted at each time step t is the next element in the sequence, while the *input* at each time step is the previous element in the sequence (with all previous inputs summarized in the state), and the *output* is interpreted as parametrizing the probability distribution of the target given the state. This is illustrated in Figure 10.7.

If the RNN is actually going to be used to generate sequences, one must also incorporate in the output information allowing to stochastically decide when to stop generating new output elements. This can be achieved in various ways. In the case when the output is a symbol taken from a vocabulary, one can add a special symbol corresponding to the end of a sequence. When that symbol is generated, a complete sequence has been generated. The target for that special symbol occurs exactly once per sequence, as the last target after the last output element $\boldsymbol{x}_T$. In

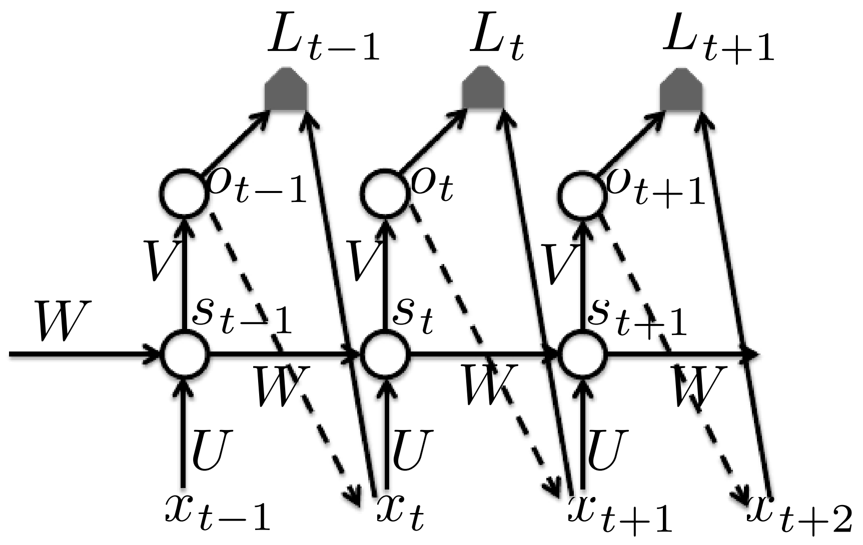

Figure 10.7: A generative recurrent neural network modeling $P(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$, able to generate sequences from this distribution. Each element $\boldsymbol{x}_t$ of the observed sequence serves both as input (for computing the state $\boldsymbol{s}_t$ at the current time step) and as target (for the prediction made at the previous time step). The output $\boldsymbol{o}_t$ encodes the parameters of a conditional distribution $P(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t) = P(\boldsymbol{x}_{t+1} \mid \boldsymbol{o}_t)$ for $\boldsymbol{x}_{t+1}$, given the past sequence $\boldsymbol{x}_1 \ldots, \boldsymbol{x}_t$. The loss L_t is the negative log-likelihood associated with the output prediction (or more generally, distribution parameters) $\boldsymbol{o}_t$, when the actual observed target is $\boldsymbol{x}_{t+1}$. In training mode, one measures and minimizes the sum of the losses over observed sequence(s) $\boldsymbol{x}$. In generative mode, $\boldsymbol{x}_t$ is sampled from the conditional distribution $P(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t) = P(\boldsymbol{x}_{t+1} \mid \boldsymbol{o}_t)$ (dashed arrows) and then that generated sample $\boldsymbol{x}_{t+1}$ is fed back as input for computing the next state $\boldsymbol{s}_{t+1}$, the next output $\boldsymbol{o}_{t+1}$, and generating the next sample $\boldsymbol{x}_{t+2}$, etc.

general, one may train a binomial output associated with that stopping variable, for example using a sigmoid output non-linearity and the cross entropy loss, i.e., again negative log-likelihood for the event "end of the sequence". Another kind of solution is to model the integer T itself, through any reasonable parametric distribution, and use the number of time steps left (and possibly the number of time steps since the beginning of the sequence) as extra inputs at each time step. Thus we would have decomposed $P(\boldsymbol{x}_1 \ldots, \boldsymbol{x}_T)$ into $P(T)$ and $P(\boldsymbol{x}_1 \ldots, \boldsymbol{x}_T \mid T)$. In general, one must therefore keep in mind that in order to fully generate a sequence we must not only generate the $\boldsymbol{x}_t$'s, but also the sequence length T, either implicitly through a series of continue/stop decisions (or a special "end-of-sequence" symbol), or explicitly through modeling the distribution of T itself as an integer random variable.

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

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

10.2.3 RNNs to Represent Conditional Probability Distributions

In general, as discussed in Section 6.3.2 (see especially the end of that section, in Subsection 6.3.2), when we can represent a parametric probability distribution $P(\mathbf{y} \mid \boldsymbol{\omega})$, we can make it conditional by making ω a function of the appropriate conditioning variable:

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

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

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

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

In general, one may need to add extra parameters (and parametrization) to map $\mathbf{x} = \boldsymbol{x}$ into the "extra bias" going either into only $\boldsymbol{s}_0$, into the other $\boldsymbol{s}_t$ $(t > 0)$, or into both. The first (and most commonly used) approach is illustrated in Figure 10.8.

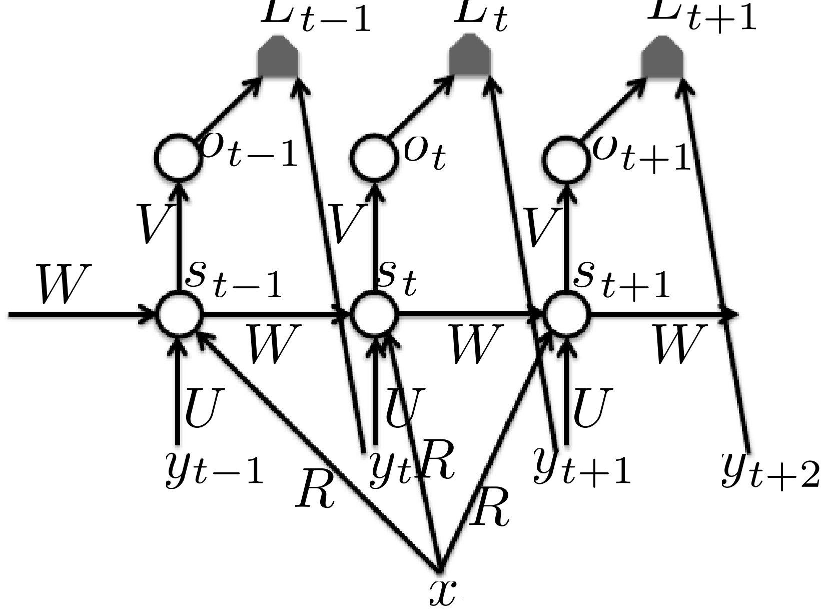

Figure 10.8: A conditional generative recurrent neural network maps a fixed-length vector $\boldsymbol{x}$ into a distribution over sequences $\mathbf{Y}$. Each element $\boldsymbol{y}_t$ of the observed output sequence serves both as input (for the current time step) and, during training, as target (for the previous time step). The generative semantics are the same as in the unconditional case (Fig. 10.7). The only difference is that the state is now conditioned on the input $\boldsymbol{x}$, and the same parameters (weight matrix $\boldsymbol{R}$ in the figure) is used at every time step to parametrize that dependency. Although this was not discussed in Fig. 10.7, in both figures one should note that the length of the sequence must also be generated (unless known in advance). This could be done by a special sigmoidal output unit that predicts a binary target (with associated cross-entropy loss) that encodes the fact that the next output is the last.

As an example, we could imagine that $\mathbf{x}$ is encoding the identity of a phoneme and the identity of a speaker, and that $\mathbf{y}$ represents an acoustic sequence corresponding to that phoneme, as pronounced by that speaker.

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

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

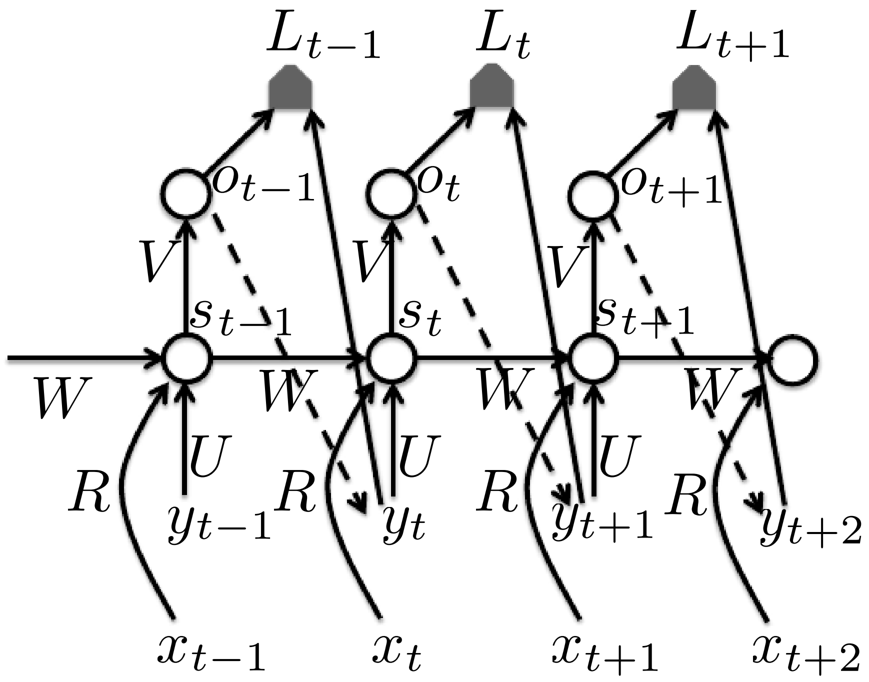

Figure 10.9: A conditional generative recurrent neural network mapping a variable-length sequence $\boldsymbol{x}$ into a distribution over sequences $\mathbf{y}$ of the same length. This architecture assumes that the predictions of $\mathbf{y}_t$'s are causally related to the $\mathbf{x}_t$'s, i.e., that we want to predict the $\mathbf{y}_t$'s only using the past $\mathbf{x}_t$'s. Note how the prediction of $\mathbf{y}_{t+1}$ is based on both the past $\mathbf{x}$'s and the past $\mathbf{y}$'s. The dashed arrows indicate that $\boldsymbol{y}_t$ can be generated by sampling from the output distribution $\boldsymbol{o}_{t-1}$. When $\boldsymbol{y}$ is clamped (known), it is used as a target in the loss L_{t-1} which measures the log-probability that $\boldsymbol{y}_t$ would be sampled from the distribution o_{t-1}.

in Fig. 10.9.

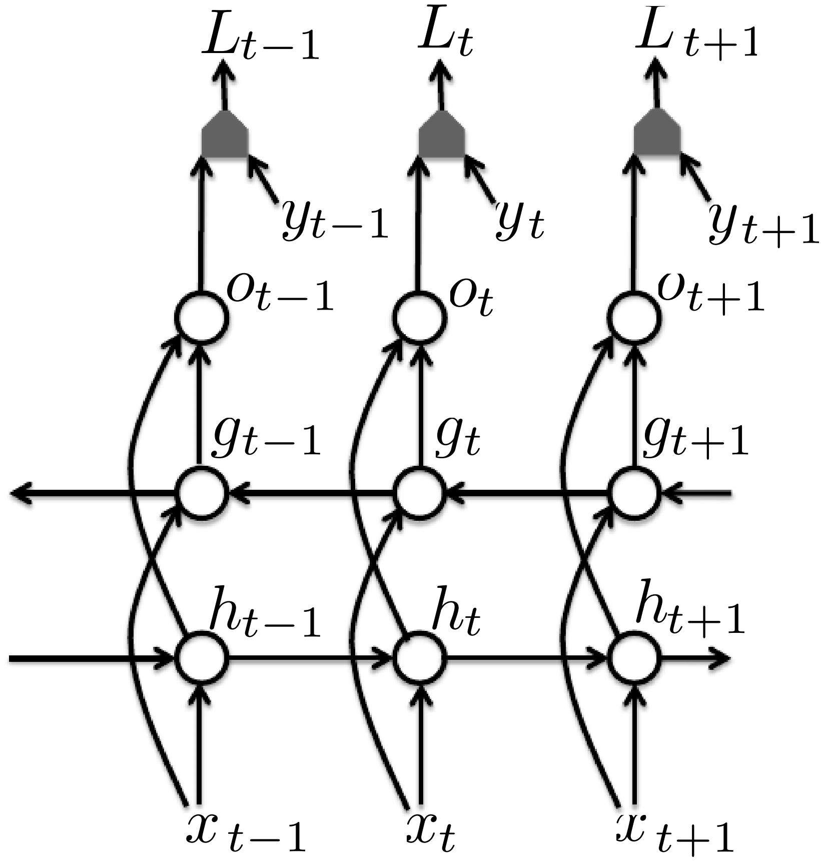

Figure 10.10: Computation of a typical bidirectional recurrent neural network, meant to learn to map input sequences $\boldsymbol{x}$ to target sequences $\boldsymbol{y}$, with loss L_t at each step t. The $\boldsymbol{h}$ recurrence propagates information forward in time (towards the right) while the $\boldsymbol{g}$ recurrence propagates information backward in time (towards the left). Thus at each point t, the output units $\boldsymbol{o}_t$ can benefit from a relevant summary of the past in its $\boldsymbol{h}_t$ input and from a relevant summary of the future in its $\boldsymbol{g}_t$ input.

10.3 Bidirectional RNNs

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

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

As the name suggests, the basic idea behind bidirectional RNNs is to combine a forward-going RNN and a backward-going RNN. Figure 10.10 illustrates the typical bidirectional RNN, with $\boldsymbol{h}_t$ standing for the state of the forward-going RNN and $\boldsymbol{g}_t$ standing for the state of the backward-going RNN. This allows the units $\boldsymbol{o}_t$ to compute a representation that depends on *both the past and the future* but is most sensitive to the input values around time t, without having to specify a fixed-size window around t (as one would have to do with a feedforward network, a convolutional network, or a regular RNN with a fixed-size look-ahead buffer).

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

10.4 Encoder-Decoder Sequence-to-Sequence Architectures

We have seen in Figure 10.6 how an RNN can map an input sequence to a fixed-size prediction. We have seen in Figure 10.7 how an RNN can model a distribution over sequences and generate new ones from the estimated distribution. We have seen in Figure 10.8 how one can condition on an input vector to learn to generate such sequences. We have seen in Figures 10.9 and 10.10 how an RNN (unidirectional or bidirectional) can map an input sequence to an output sequence of the same length.

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

The simplest RNN architecture for mapping a variable-length sequence to another variable-length sequence was first proposed in Cho *et al.* (2014) and shortly

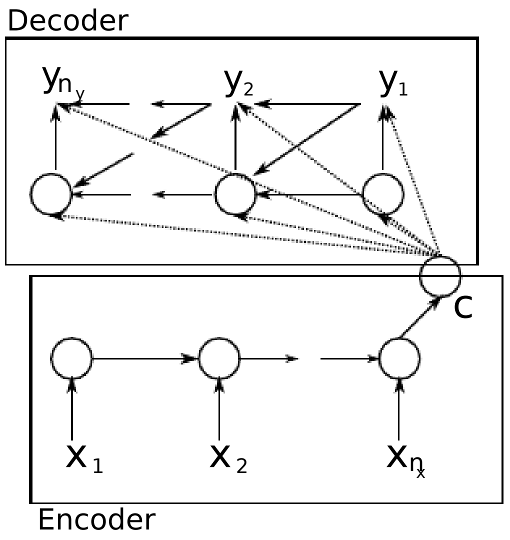

Figure 10.11: Example of encoder-decoder or sequence-to-sequence RNN architecture, for learning to generate an output sequence $(\mathbf{y}_1, \ldots, \mathbf{y}_{n_y})$ given an input sequence $(\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_{n_x})$. It is composed of an encoder RNN that reads the input sequence and a decoder RNN that generates the output sequence (or computes the probability of a given output sequence). The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable C which represents a semantic summary of the input sequence and conditions computations in the decoder RNN.

after in Sutskever *et al.* (2014b). These authors respectively called this architecture, illustrated in Figure 10.11, the encoder-decoder or sequence-to-sequence architecture. The idea is very simple: (1) an *encoder* or *reader* or *input* RNN processes the input sequence, producing from its last hidden state a representation C of the input sequence $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{n_x})$; (2) a *decoder* or *writer* or *output* RNN is conditioned on that fixed-length vector (just like in Figure 10.8) to generate the output sequence $\boldsymbol{Y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{n_y})$, where the lengths n_x and n_y can vary from training pair to training pair. The two RNNs are trained jointly to maximize the average of $\log P(\mathbf{Y} = \boldsymbol{Y} | \mathbf{X} = \boldsymbol{X})$ over all the training pairs $(\boldsymbol{X}, \boldsymbol{Y})$. The last state $\boldsymbol{s}_{n_x}$ of the input RNN is typically used as a representation C of the input sequence that conditions the output RNN. The output RNN can be conditioned in at least two ways, which can be combined, as we have seen earlier, i.e., either by producing a starting state for the output RNN or by producing an extra input at each time step of the output RNN. In any case, one inserts an extra set of parameters to map C into a bias or initial state. Hence the two RNNs do not have to have the same hidden layer dimensionality, and sometimes it makes sense to make that mapping more complex and non-linear, e.g., an MLP could be used to map the output of the encoder RNN into an input for the decoder RNN.

One clear limitation of this architecture is when the output of the encoder RNN has a dimension that is too small to properly summarize a long sequence.

This phenomenon was observed by Bahdanau *et al.* (2014) in the context of machine translation, and they proposed to make C a variable length sequence rather than a fixed-size vector, introducing a particular *attention mechanism* that learns to associate elements of the sequence C to elements of the output RNN sequence. See Section 12.4.6 for more details.

10.5 Deep Recurrent Networks

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

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

where the first two are actually brought together to map the input and previous state into the next state. With the vanilla RNN architecture (such as in 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, i.e., a single layer within a deep MLP.

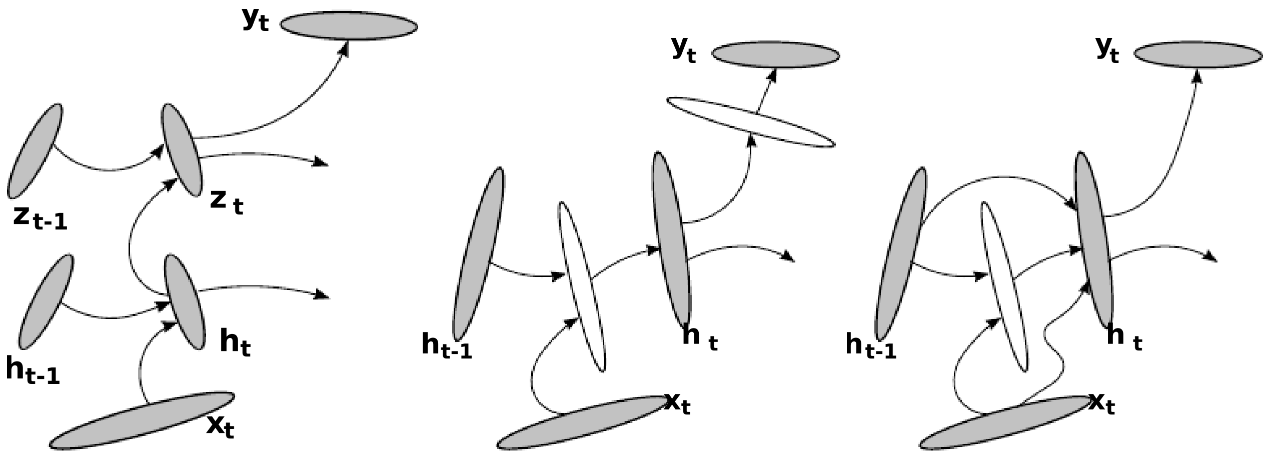

Figure 10.12: A recurrent neural network can be made deep in many ways. First, the hidden recurrent state can be broken down into groups organized hierarchically (left). Second, deeper computation (e.g., an MLP in the figure) can be introduced in the input-to-hidden, hidden-to-hidden, and hidden-to-output parts (Middle). However, this may lengthen the shortest path linking different time steps, but this can be mitigated by introduced skip connections (Right). Figures from Pascanu *et al.* (2014a) with permission.

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, and this is in agreement with the idea that we need enough depth in order to perform the required mappings. See also Schmidhuber (1992); El Hihi and Bengio (1996); Jaeger (2007a) for earlier work on deep RNNs.

El Hihi and Bengio (1996) first introduced the idea of decomposing the hidden state of an RNN into multiple groups of units that would operate at different time scales. Graves *et al.* (2013) were the first to show a significant benefit of decomposing the state of an RNN into groups of hidden units, with a restricted connectivity between the groups, e.g., as in Figure 10.12 (left). Indeed, if there were no restriction at all and no pressure for some units to represent a slower time scale, then having N groups of M hidden units would be equivalent to having a single group of NM hidden units. Koutnik *et al.* (2014) showed how the multiple time scales idea from El Hihi and Bengio (1996) can be advantageous on several sequential learning tasks: each group of hidden unit is updated at a different multiple of the time step index e.g., at every time step, at every 2nd step, at every 4th step, etc.

We can also think of the lower layers in this hierarchy as playing a role in transforming the raw input into a representation that is more appropriate, at the higher levels of the hidden state. Pascanu *et al.* (2014a) go a step further and propose to have a separate MLP (possibly deep) for each of the three blocks enumerated above, as illustrated in Figure 10.12 (middle). It makes sense to allocate enough capacity in each of these three steps, but having a deep state-to-state transition may also hurt: it makes the shortest path from an event at time t to an event at time $t' > t$ substantially longer, which make it more difficult to learn long-term dependencies (see Sections 8.2.5 and 10.8). For example if a one-hidden-layer MLP is used for the state-to-state transition, we have doubled the length of that path, compared with a vanilla RNN. However, as argued by Pascanu *et al.* (2014a), this can be mitigated by introducing skip connections in the hidden-to-hidden path, as illustrated in Figure 10.12 (right).

10.6 Recursive Neural Networks

Recursive networks represent yet another generalization of recurrent networks, with a different kind of computational graph, which is structured as a deep tree, rather than the chain-like structure of RNNs. The typical computational graph for a recursive network is illustrated in Figure 10.13. Recursive neural networks were introduced by Pollack (1990) and their potential use for learning to reason were nicely laid down by Bottou (2011). Recursive networks have been successfully applied in processing *data structures* as input to neural nets (Frasconi *et al.*, 1997, 1998), in natural language processing (Socher *et al.*, 2011a,c, 2013) as well as in computer vision (Socher *et al.*, 2011b).

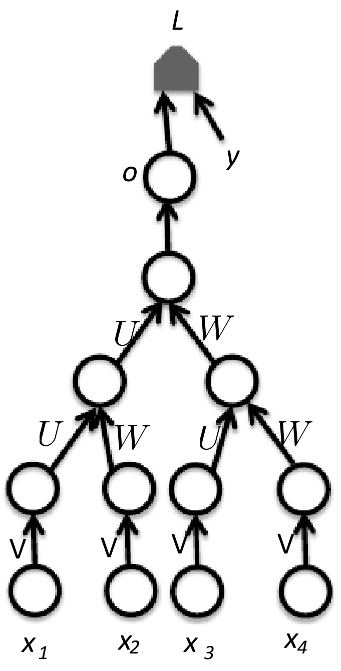

Figure 10.13: A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree. In the figure, a variable-size sequence $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$ can be mapped to a fixed-size representation (the output o), with a fixed number of parameters (e.g. the weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$). The figure illustrates a supervised learning case in which some target y is provided which is associated with the whole sequence.

One clear advantage of recursive nets over recurrent nets is that for a sequence of the same length N, depth[3] can be drastically reduced from N to $O(\log N)$, which might help deal with long-term dependencies. An open question is how to best structure the tree, though. One option is to have a tree structure which does not depend on the data, e.g., a balanced binary tree. Another is to use an external method, such as a natural language parser (Socher *et al.*, 2011a, 2013). Ideally, one would like the learner itself to discover and infer the tree structure that is appropriate for any given input, as suggested in Bottou (2011).

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

10.7 Auto-Regressive Networks

One of the basic ideas behind recurrent networks is that of directed graphical models with a twist: we decompose a probability distribution as a product of conditionals *without explicitly cutting any arc in the graphical model*[4], but instead reducing complexity by parametrizing the transition probability in a recursive way that requires a fixed (and not exponential) number of parameters, due to a form of parameter sharing (see Section 7.8 for an introduction to the concept). Instead of reducing $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1)$ to something like $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_{t-k})$ (assuming the k previous ones as the parents), we keep the full dependency but we parametrize the conditional efficiently in a way that does not grow with t, exploiting parameter sharing. When the above conditional probability distribution is in some sense stationary, i.e., the relation between the past and the next observation does not depend on t, only on the values of the past observations, then this form of parameter sharing makes a lot of sense, and for recurrent nets it allows one to use the same model for sequences of different lengths.

Auto-regressive networks are similar to recurrent networks in the sense that we also decompose a joint probability over the observed variables as a product of

[3]i.e., the number of compositions of non-linear operations

[4]i.e., every element of the sequence depends on all the previous ones; however it is possible to obtain a more compact graphical model from an RNN if one introduces deterministic nodes in the graph to represent the hidden units at different time steps, but if we restrict ourselves to non-deterministic nodes, the graphical model is fully-connected.

conditionals of the form $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1)$ but we drop the form of parameter sharing that makes these conditionals all share the same parametrization. This makes sense when the variables are *not* elements of a translation-equivariant sequence (see Section 9.2 for more on equivariance), but instead form an arbitrary tuple without any particular ordering that would correspond to a translation-equivariant form of relationship between variables at position k and variables at position k'. Such models have been called *fully-visible Bayes networks* (Frey *et al.*, 1996) and used successfully in many forms, first with logistic regression for each conditional distribution (Frey, 1998) and then with neural networks (Bengio and Bengio, 2000b; Larochelle and Murray, 2011). In some forms of auto-regressive networks, such as NADE (Larochelle and Murray, 2011), described in Section 10.7.3 below, we can re-introduce a form of parameter sharing that is different from the one found in recurrent networks, but that brings both a statistical advantage (less parameters) and a computational advantage (less computation). Although we drop the sharing over time, as we see below in Section 10.7.2, using a deep learning concept of *reuse of features*, we can *share* features that have been computed for predicting $\boldsymbol{x}_{t-k}$ with the sub-network that predicts $\boldsymbol{x}_t$.

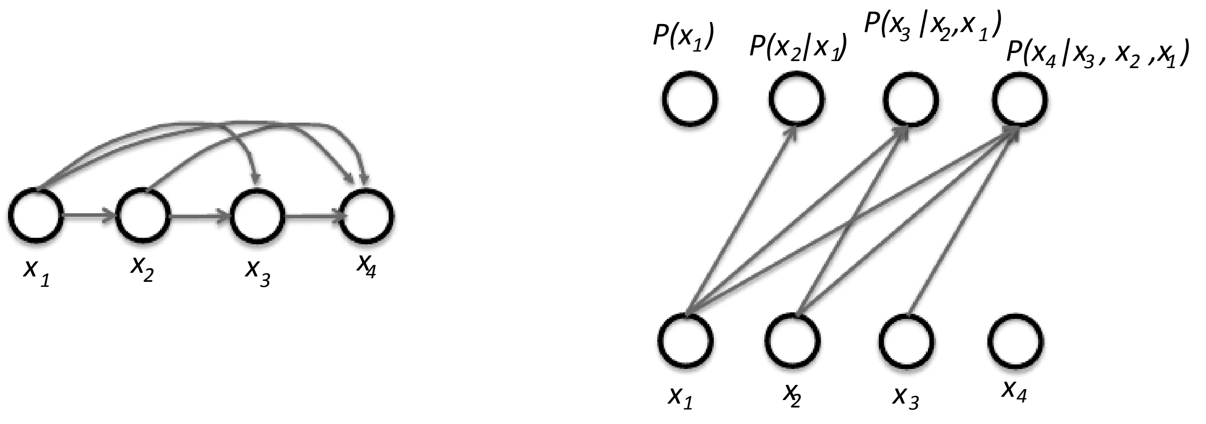

Figure 10.14: An auto-regressive network predicts the i-th variable from the $i-1$ previous ones. Left: corresponding graphical model (which is the same as that of a recurrent network). Right: corresponding computational graph, in the case of the logistic auto-regressive network, where each prediction has the form of a logistic regression, i.e., with i free parameters (for the $i-1$ weights associated with $i-1$ inputs, and an offset parameter).

10.7.1 Logistic Auto-Regressive Networks

Let us first consider the simplest auto-regressive network, without hidden units, and hence no sharing at all. Each $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1)$ is parametrized as a linear model, e.g., a logistic regression. This model was introduced by Frey (1998) and has $O(T^2)$ parameters when there are T variables to be modeled. It is illustrated in Figure 10.14, showing both the graphical model (left) and the computational graph (right).

A clear disadvantage of the logistic auto-regressive network is that one cannot easily increase its capacity in order to capture more complex data distributions. It defines a parametric family of fixed capacity, like the linear regression, the logistic regression, or the Gaussian distribution. In fact, if the variables are continuous, one gets a linear auto-regressive model, which is thus another way to formulate a Gaussian distribution, i.e., only capturing pairwise interactions between the observed variables.

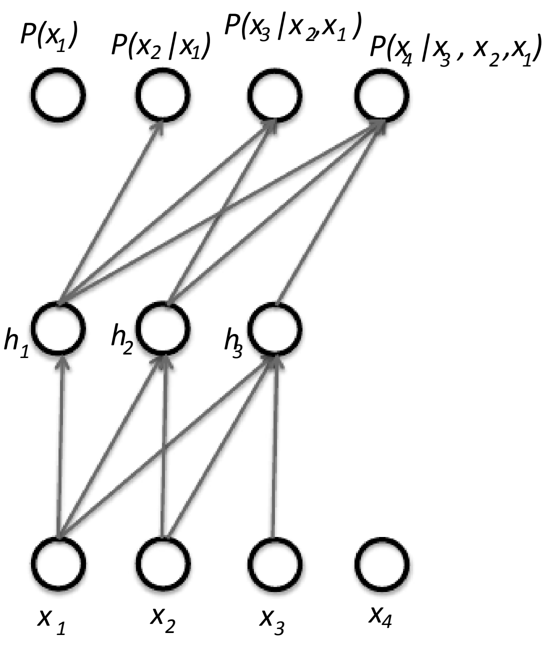

Figure 10.15: A neural auto-regressive network predicts the i-th variable $\boldsymbol{x}_i$ from the $i-1$ previous ones, but is parametrized so that features (groups of hidden units denoted h_i) that are functions of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i$ can be reused in predicting all of the subsequent variables $\boldsymbol{x}_{i+1}, \boldsymbol{x}_{i+2}, \ldots$.

10.7.2 Neural Auto-Regressive Networks

Neural Auto-Regressive Networks have the same left-to-right graphical model as logistic auto-regressive networks (Figure 10.14, left) but a different parametrization that is at once more powerful (allowing to extend the capacity as needed and approximate any joint distribution) and can improve generalization by introducing a parameter sharing and feature sharing principle common to deep learning in general. The first paper on neural auto-regressive networks by Bengio and Bengio (2000b) (see also Bengio and Bengio (2000a) for the more extensive journal version) were motivated by the objective to avoid the curse of dimensionality arising out of traditional non-parametric graphical models, sharing the same structure as Figure 10.14 (left). In the non-parametric discrete distribution models, each conditional distribution is represented by a table of probabilities, with one entry and one parameter for each possible configuration of the variables involved. By using a neural network instead, two advantages are obtained:

1. The parametrization of each $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1)$ by a neural network with $(t-1) \times k$ inputs and k outputs (if the variables are discrete and take k values, encoded one-hot) allows one to estimate the conditional probability without requiring an exponential number of parameters (and examples), yet still allowing to capture high-order dependencies between the random variables.

2. Instead of having a different neural network for the prediction of each $\boldsymbol{x}_t$, a *left-to-right* connectivity illustrated in Figure 10.15 allows one to merge all the neural networks into one. Equivalently, it means that the hidden layer features computed for predicting $\boldsymbol{x}_t$ can be reused for predicting $\boldsymbol{x}_{t+k}$ $(k > 0)$. The hidden units are thus organized in *groups* that have the particularity that all the units in the t-th group only depend on the input values $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$. In fact the parameters used to compute these hidden units are jointly optimized to help the prediction of all the variables $\boldsymbol{x}_{t+1}, \boldsymbol{x}_{t+2}, \ldots$. This is an instance of the *reuse principle* that makes *multi-task learning* and *transfer learning* successful with neural networks and deep learning in general (See Sections 7.12 and 16.2).

Each $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_1)$ can represent a conditional distribution by having outputs of the neural network predict *parameters* of the conditional distribution of $\boldsymbol{x}_t$, as discussed in Section 6.3.2. Although the original neural auto-regressive networks were initially evaluated in the context of purely discrete multivariate data (e.g., with a sigmoid output - Bernoulli case - or softmax output - multinoulli case) it is natural to extend such models to continuous variables or joint distributions involving both discrete and continuous variables, as for example with RNADE introduced below (Uria *et al.*, 2013).

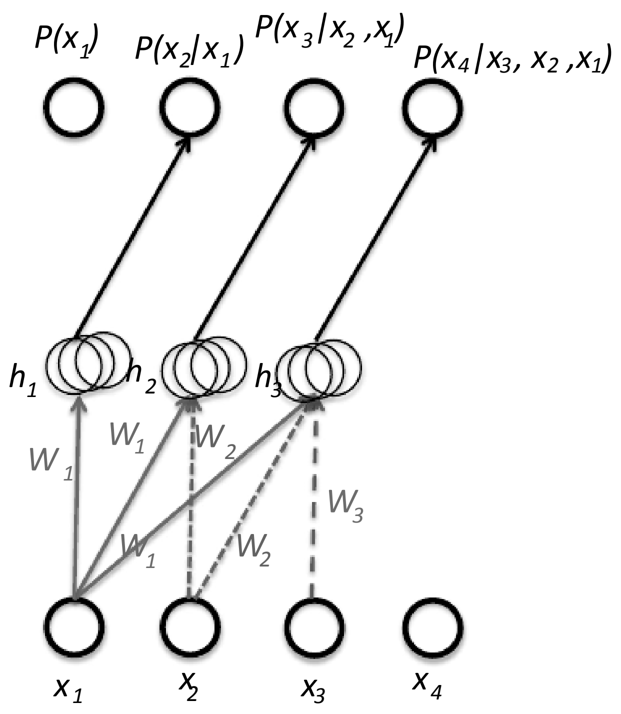

Figure 10.16: NADE (Neural Auto-regressive Density Estimator) is a neural auto-regressive network, i.e., the hidden units are organized in groups $\boldsymbol{h}_j$ so that only the inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i$ participate in computing $\boldsymbol{h}_i$ and predicting $P(\boldsymbol{x}_j \mid \boldsymbol{x}_{j-1}, \ldots, \boldsymbol{x}_1)$, for $j > i$. The particularity of NADE is the use of a particular weight sharing pattern: the same $W'_{jki} = W_{ki}$ is shared (same color and line pattern in the figure) for all the weights outgoing from $\boldsymbol{x}_i$ to the k-th unit of any group $j \geq i$. The vector $(W_{1i}, W_{2i}, \ldots)$ is denoted $\boldsymbol{W}_{:,i}$ here.

10.7.3 NADE

A very successful recent form of neural auto-regressive network was proposed by Larochelle and Murray (2011). The architecture is basically the same as for the original neural auto-regressive network of Bengio and Bengio (2000b) *except for the introduction of a weight-sharing scheme*: as illustrated in Figure 10.16. The parameters of the hidden units of different groups j are shared, i.e., the weights W'_{jki} from the i-th input $\boldsymbol{x}_i$ to the k-th element of the j-th group of hidden unit h_{jk} $(j \geq i)$ are shared:

$$W'_{jki} = W_{ki}$$

with $(W_{1i}, W_{2i}, \ldots)$ denoted $\boldsymbol{W}_{:,i}$ in Figure 10.16.

This particular sharing pattern is motivated in Larochelle and Murray (2011) by the computations performed in the mean-field inference[5] of an RBM, when only

[5] Here, unlike in Section 13.5, the inference is over some of the input variables that are missing, given the observed ones.

the first i inputs are given and one tries to infer the subsequent ones. This mean-field inference corresponds to running a recurrent network with shared weights and the first step of that inference is the same as in NADE. The only difference is that with the proposed NADE, the output weights are not forced to be simply transpose values of the input weights (they are not tied). One could imagine actually extending this procedure to not just one time step of the mean-field recurrent inference but to k steps, as in Raiko *et al.* (2014).

Although the neural auto-regressive networks and NADE were originally proposed to deal with discrete distributions, they can in principle be generalized to continuous ones by replacing the conditional discrete probability distributions (for $P(\boldsymbol{x}_j \mid \boldsymbol{x}_{j-1}, \ldots, \boldsymbol{x}_1))$ by continuous ones and following general practice to predict continuous random variables with neural networks (see Section 6.3.2) using the log-likelihood framework. A fairly generic way of parametrizing a continuous density is as a Gaussian mixture, and this avenue has been successfully evaluated for the neural auto-regressive architecture with RNADE (Uria *et al.*, 2013). One interesting point to note is that stochastic gradient descent can be numerically ill-behaved due to the interactions between the conditional means and the conditional variances (especially when the variances become small). Uria *et al.* (2013) have used a heuristic to rescale the gradient on the component means by the associated standard deviation which seems to have helped optimizing RNADE.

Another very interesting extension of the neural auto-regressive architectures gets rid of the need to choose an arbitrary *order* for the observed variables (Murray and Larochelle, 2014). In auto-regressive networks, the idea is to train the network to be able to cope with any order by randomly sampling orders and providing the information to hidden units specifying which of the inputs are observed (on the right side of the conditioning bar) and which are to be predicted and are thus considered missing (on the left side of the conditioning bar). This is nice because it allows one to use a trained auto-regressive network to *perform any inference* (i.e. predict or sample from the probability distribution over any subset of variables given any subset) extremely efficiently. Finally, since many orders are possible, the joint probability of some set of variables can be computed in many ways ($n!$ for n variables), and this can be exploited to obtain a more robust probability estimation and better log-likelihood, by simply averaging the log-probabilities predicted by different randomly chosen orders. In the same paper, the authors propose to consider deep versions of the architecture, but unfortunately that immediately makes computation as expensive as in the original neural auto-regressive neural network (Bengio and Bengio, 2000b). The first layer and the output layer can still be computed in $O(nh)$ multiply-add operations, as in the regular NADE, where h is the number of hidden units (the size of the groups h_i, in Figures 10.16 and 10.15), whereas it is $O(n^2 h)$ in Bengio and Bengio (2000b). However, for the other hidden layers, the computation is $O(n^2h^2)$ if every "previous" group at layer l participates in predicting the "next" group at layer $l + 1$, assuming n groups of h hidden units at each layer. Making the i-th group at layer $l + 1$ only depend on the i-th group, as in Murray and Larochelle (2014) at layer l reduces it to $O(nh^2)$, which is still h times worse than the regular NADE.

10.8 Facing the Challenge of Long-Term Dependencies

The mathematical challenge of learning long-term dependencies in recurrent networks was introduced in Section 8.2.5. The basic problem is that gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization). Even if we assume that the parameters are such that the recurrent network is stable (can store memories, with gradients not exploding), the difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions (involving the multiplication of many Jacobians) compared short-term ones. See Hochreiter (1991); Doya (1993); Bengio *et al.* (1994); Pascanu *et al.* (2013a) for a deeper treatment.

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

10.8.1 Echo State Networks: Choosing Weights to Make Dynamics Barely Contractive

The recurrent weights and input weights of a recurrent network are those that define the state representation captured by the model, i.e., how the state s_t (hidden units vector) at time t (Eq. 10.2) captures and summarizes information from the previous inputs $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_t$. Since learning the recurrent and input weights is difficult, one option that has been proposed (Jaeger and Haas, 2004; Jaeger, 2007b; Maass *et al.*, 2002) is to *set those weights such that the recurrent hidden units do a good job of capturing the history of past inputs*, and *only learn the output weights*. This is the idea that was independently proposed for *Echo State Networks* or ESNs (Jaeger and Haas, 2004; Jaeger, 2007b) and *Liquid State Machines* (Maass *et al.*, 2002). The latter is similar, except that it uses spiking neurons (with binary outputs) instead of the continuous-valued hidden units used for ESNs. Both ESNs and liquid state machines are termed *reservoir computing* (Lukoševičius and Jaeger, 2009) to denote the fact that the hidden units form of reservoir of temporal features which may capture different aspects of the history of inputs.

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

The important question is therefore: how do we set the input and recurrent weights so that a rich set of histories can be represented in the recurrent neural network state? The answer proposed in the reservoir computing literature is to make the dynamical system associated with the recurrent net nearly be on the edge of stability, i.e., more precisely with values around 1 for the leading singular value of the Jacobian of the state-to-state transition function. As alluded to in 8.2.5, an important characteristic of a recurrent network is the eigenvalue spectrum of the Jacobians $\boldsymbol{J}^{(t)} = \frac{\partial s_t}{\partial s_{t-1}}$, and in particular the *spectral radius* of $\boldsymbol{J}^{(t)}$, i.e., its largest eigenvalue. If it is greater than 1, the dynamics can diverge, meaning that small differences in the state value at t can yield a very large difference at T later in the sequence. To see this, consider the simpler case where the Jacobian matrix $\boldsymbol{J}$ does not change with t. If a change Δs in the state at t is aligned with an eigenvector $\boldsymbol{v}$ of $\boldsymbol{J}$ with eigenvalue $\lambda > 1$, then the small change Δs becomes $\lambda \Delta s$ after one time step, and $\lambda^N \Delta s$ after N time steps. If $\lambda > 1$ this makes the change exponentially large. More generally, Δs can be the component of the state change vector in the direction $\boldsymbol{v}$. With a non-linear map, the Jacobian keeps changing and the dynamics is more complicated but what remains is that a small initial variation can turn into a large variation after a few steps. Note that, in general, the recurrent dynamics are bounded (for example, if the hidden units use a bounded non-linearity such as the hyperbolic tangent) so that the change after N steps must also be bounded. Instead when the largest eigenvalue $\lambda < 1$, we say that the map from t to $t+1$ is *contractive*: a small change gets *contracted*, becoming smaller after each time step. This necessarily makes the network *forgetting* information about the long-term past, but it also makes its dynamics stable and easier to use.

What has been proposed to set the weights of reservoir computing machines is to make the Jacobians *slightly contractive*. This is achieved by making the spectral radius of the weight matrix large but slightly less than 1. However, in practice, good results are often found with a spectral radius of the recurrent weight matrix being slightly larger than 1, e.g., 1.2 (Sutskever, 2012; Sutskever *et al.*, 2013). Keep in mind that with hyperbolic tangent units, the maximum derivative is 1, so that in order to guarantee a Jacobian spectral radius less than 1, the weight matrix should have spectral radius less than 1 as well. However, most derivatives of the hyperbolic tangent will be less than 1, which may explain Sutskever's empirical observation.

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

Note that when some eigenvalues of the Jacobian are exactly 1, information can be kept in a stable way, and back-propagated gradients neither vanish nor explode. The next two sections show methods to make some paths in the unfolded graph correspond to "multiplying by 1" at each step, i.e., keeping information for a very long time.

10.8.2 Combining Short and Long Paths in the Unfolded Flow Graph

An old idea that has been proposed to deal with the difficulty of learning long-term dependencies is to use recurrent connections with long delays (Lin *et al.*, 1996). Whereas the ordinary recurrent connections are associated with a delay of 1 (relating the state at t with the state at $t + 1$), it is possible to construct recurrent networks with longer delays (Bengio, 1991), following the idea of incorporating delays in feedforward neural networks (Lang and Hinton, 1988) in order to capture temporal structure (with Time-Delay Neural Networks, which are the 1-D predecessors of Convolutional Neural Networks, discussed in Chapter 9).

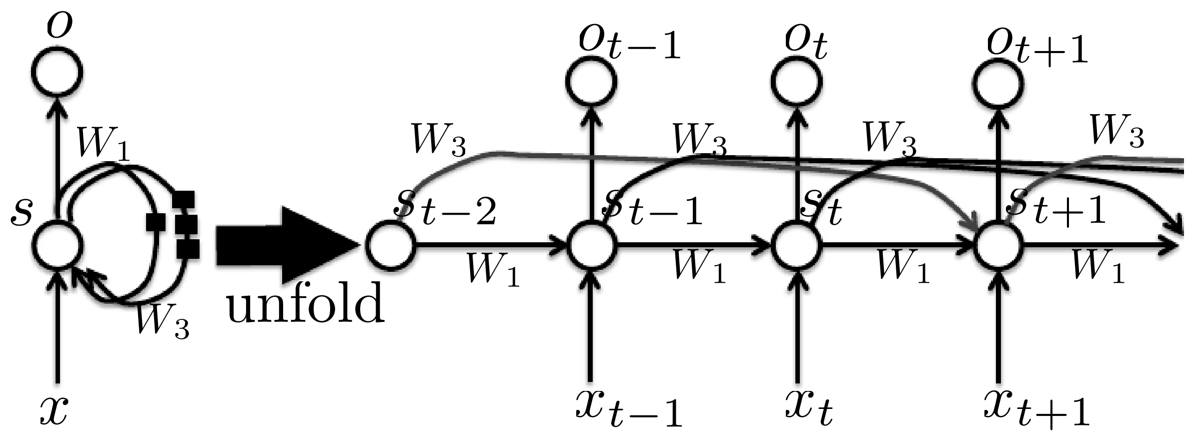

Figure 10.17: A recurrent neural networks with *delays*, in which some of the connections reach back in time to more than one time step. Left: connectivity of the recurrent net, with square boxes indicating the number of time delays associated with a connection. Right: unfolded recurrent network. In the figure there are regular recurrent connections with a delay of 1 time step (W_1) and recurrent connections with a delay of 3 time steps (W_3). The advantage of these longer-delay connections is that they allow to connect past states to future states through shorter paths (3 times shorter, here), going through these longer delay connections (in red).

As we have seen in Section 8.2.5, gradients may vanish or explode exponentially *with respect to the number of time steps.* If we have recurrent connections with a time-delay of d, then instead of the vanishing or explosion going as $O(\lambda^T)$ over T time steps (where λ is the largest eigenvalue of the Jacobians $\frac{\partial s t}{\partial s_{t-1}}$), the unfolded recurrent network now has paths through which gradients grow as $O(\lambda^{T/d})$ because the number of effective steps is T/d. This allows the learning algorithm to capture longer dependencies although not all long-term dependencies may be well represented in this way. This idea was first explored in Lin *et al.* (1996) and is illustrated in Figure 10.17.

10.8.3 Leaky Units and a Hierarchy of Different Time Scales

A related idea in order to obtain paths on which the product of derivatives is close to 1 is to have units with *linear* self-connections and a weight near 1 on these connections. The strength of that linear self-connection corresponds to a time scale and thus we can have different hidden units which operate at different time scales (Mozer, 1992). Depending on how close to 1 these self-connection weights are, information can travel forward and gradients backward with a different rate of "forgetting" or contraction to 0, i.e., a different *time scale.* One can view this idea as a smooth variant of the idea of having different delays in the connections presented in the previous section. Such ideas were proposed in Mozer (1992); ElHihi and Bengio (1996), before a closely related idea discussed in the next section of *gating* these self-connections in order to let the network control at what rate each unit should be contracting.

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

$$\dot{s}_i \tau_i = -s_i + \sigma(b_i + Ws + Ux)$$

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

$$\dot{s}_i \tau_i = -s_i + (b_i + W\sigma(s) + Ux)$$

where the state vector s (with elements s_i) now represents the pre-activation of the hidden units.

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

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

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

By associating different time scales τ_i with different units, one obtains different paths corresponding to different forgetting rates. Those time constants can be fixed manually (e.g., by sampling from a distribution of time scales) or can be learned as free parameters, and having such leaky units at different time scales appears to help with long-term dependencies (Mozer, 1992; Pascanu *et al.*, 2013a). Note that the time constant τ thus corresponds to a *self-weight* of $(1 - \frac{1}{\tau})$, but *without any non-linearity involved in the self-recurrence.*

Consider the extreme case where $\tau \to \infty$: because the leaky unit just *averages* contributions from the past, the contribution of each time step is equivalent and there is no associated vanishing or exploding effect. An alternative is to avoid the weight of $\frac{1}{\tau_i}$ in front of $\sigma(b_i + Ws_t + U\boldsymbol{x}_t)$, thus making the state *sum* all the past values when τ_i is large, instead of averaging them.

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

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

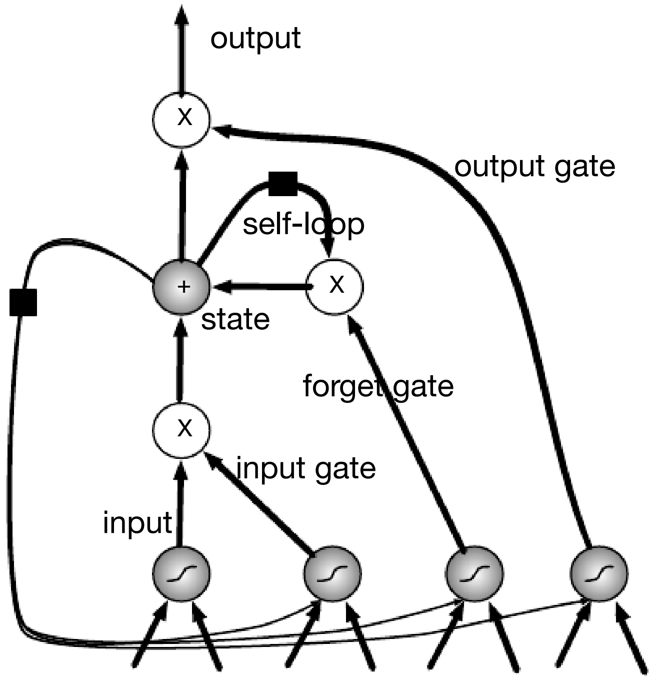

Figure 10.18: Block diagram of the LSTM recurrent network "cell". Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit, and its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid non-linearity, while the input unit can have any squashing non-linearity. The state unit can also be used as extra input to the gating units. The black square indicates a delay of 1 time unit.

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

LSTM

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

The LSTM block diagram is illustrated in Figure 10.18. The corresponding forward (state update) equations are given below, in the case of a shallow recurrent network architecture. Deeper architectures have been successfully used in Graves *et al.* (2013); Pascanu *et al.* (2014a). Instead of a unit that simply applies a squashing function on the affine transformation of inputs and recurrent units, LSTM networks have “LSTM cells”. Each cell has the same inputs and outputs as a vanilla recurrent network, but has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit s_t that has a linear self-loop similar to the leaky units described in the previous section, but where the self-loop weight (or the associated time constant) is controlled by a *forget gate* unit $h^f_{t,i}$ (for time step t and cell i), that sets this weight to a value between 0 and 1 via a sigmoid unit:

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

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

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

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

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

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

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

which has parameters $\boldsymbol{b}^o$, $\boldsymbol{U}^o$, $\boldsymbol{W}^o$ for its biases, input weights and recurrent weights, respectively. Among the variants, one can choose to use the cell state $s_{t,i}$ as an extra input (with its weight) into the three gates of the i-th unit, as shown in Figure 10.18. This would require three additional parameters.

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

Other Gated RNNs

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

Some answers to these questions are given with the recent work on gated RNNs, whose units are also known as Gated Recurrent Units (GRU) (Chung *et al.*, 2014), which were successfully used in reaching the MOSES state-of-the-art for English-to-French machine translation (Cho *et al.*, 2014). The main difference with the LSTM is that a single gating unit simultaneously controls the forgetting factor and the decision to update the state unit, which is natural if we consider the continuous-time interpretation of the self-weight of the state, as in the equation for leaky units, Eq. 10.7. The update equations are the following:

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

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

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

and

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

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

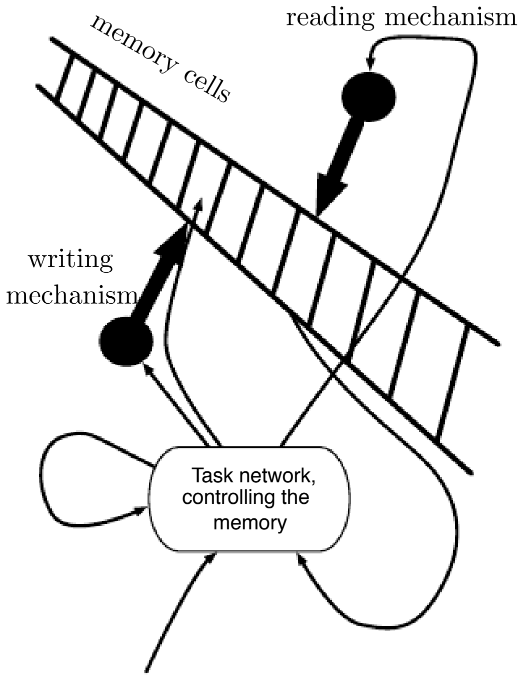

Figure 10.19: A schematic example of memory network architecture, in which we explicitly distinguish the "representation" part of the model (the "task network", here a recurrent net in the bottom) from the "memory" part of the model (the set of cells). From this representation, one learns to "control" the memory, decided from where to read and write (through the reading and writing mechanims, indicated by circles with arrows pointing at the reading and writing addresses).

10.8.5 Memory Networks

One way to extend gated RNNs (such as the LSTM and the GRU) is to make each cell store not just a single number but a whole vector, and make the different cells compete for the privilege of being updated by external inputs (writing into the memory) or read out. We can now think of the state of these recurrent units as the content of a *memory*, the input/forget gates as mechanisms to decide when and what to write into the memory, and the output gate as a mechanism to decide from which cells to read. If the content of a cell is copied (not forgotten) at most time steps, then the information it contains can be propagated forward in time and the gradients backward in time without either vanishing or exploding. This is illustrated in Figure 10.19, where we see that a neural network (recurrent in the figure) is coupled with a memory, and can choose to read from it or write to it at specific addresses. If one wants to backprop through the choice of reading or writing address, though, choosing a single discrete address is problematic, as the derivative through a discrete decision is essentially zero. To avoid this problem, it was proposed to consider multiple addresses at once, linearly weighting the reading or writing action over many (or all) the addresses, with learned weights through which we can back-propagate (Weston *et al.*, 2014; Graves *et al.*, 2014b). For example, when reading from the memory, we can take a weighted average of the contents of all the memory addresses, and those weights can be forced to sum to 1 by using a softmax. Another option is to consider these weights as probabilities for choosing to read or write at the given address, and a variant of the REINFORCE algorithm (Williams, 1992) can be used to estimate a noisy gradient on the addressing weights (Zaremba and Sutskever, 2015). These architectures with an explicit memory have been called memory networks (Weston *et al.*, 2014) or neural Turing machines (Graves *et al.*, 2014b) and seem to allow models to learn tasks that ordinary RNNs or LSTM RNNs cannot learn. One reason for this advantage may be because information and gradients can be propagated (forward in time or backwards in time, respectively) for very long durations.

10.8.6 Better Optimization

A central optimization difficulty with RNNs regards the learning of long-term dependencies (Hochreiter, 1991; Bengio *et al.*, 1993, 1994). This difficulty has been explained in detail in Section 8.2.5. The gist of the problem is that the composition of the non-linear recurrence with itself over many many time steps yields a highly non-linear function whose derivatives (e.g. of the state at T w.r.t. the state at $t < T$, i.e. the Jacobian matrix $\frac{\partial \boldsymbol{s}_T}{\partial \boldsymbol{s}_t}$) tend to either vanish or explode as $T - t$ increases, because it is equal to the product of the state transition Jacobian matrices $\frac{\partial \boldsymbol{s}_{t+1}}{\partial \boldsymbol{s}_t}$)

If it explodes, the parameter gradient $\nabla_{\boldsymbol{\theta}} L$ also explodes, yielding gradient-based parameter updates that are poor. A simple heuristic but practical solution to this problem is discussed in the next section (Sec. 10.8.7). However, as discussed in Bengio *et al.* (1994), if the state transition Jacobian matrix has eigenvalues that are larger than 1 in magnitude, then it can yield to "unstable" dynamics, in the sense that a bit of information cannot be stored reliably for a long time in the presence of input "noise". Indeed, the state transition Jacobian matrix eigenvalues indicate how a small change in some direction (the corresponding eigenvector) will be expanded (if the eigenvalue is greater than 1) or contracted (if it is less than 1).

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

One disadvantage of many second-order methods, including the Hessian-free method, is that they tend to be geared towards "batch" training (or fairly large minibatches) rather than "stochastic" updates (where only one example or a small minibatch of examples are examined before a parameter update is made). Although the experiments on recurrent networks applied to problems with long-term dependencies showed very encouraging results in Martens and Sutskever (2011), it was later shown that similar results could be obtained by much simpler methods (Sutskever, 2012; Sutskever *et al.*, 2013) involving better initialization, a cheap surrogate to second-order optimization (a variant on the momentum technique, Section 8.5), and the clipping trick described below.

10.8.7 Clipping Gradients

As discussed in Section 8.2.4, strongly non-linear functions such as those computed by a recurrent net over many time steps tend to have derivatives that can be either very large or very small in magnitude. This is illustrated in Figures 8.2 and 8.3, in which we see that the objective function (as a function of the parameters) has a "landscape" in which one finds "cliffs": wide and rather flat regions separated by tiny regions where the objective function changes quickly, forming a kind of cliff.

The difficulty that arises is that when the parameter gradient is very large, a gradient descent parameter update could throw the parameters very far, into a region where the objective function is larger, wasting a lot of the work that had been done to reach the current solution. This is because gradient descent is hinged on the assumption of *small enough steps*, and this assumption can easily be violated when the same learning rate is used for both the flatter parts and the steeper parts of the landscape.

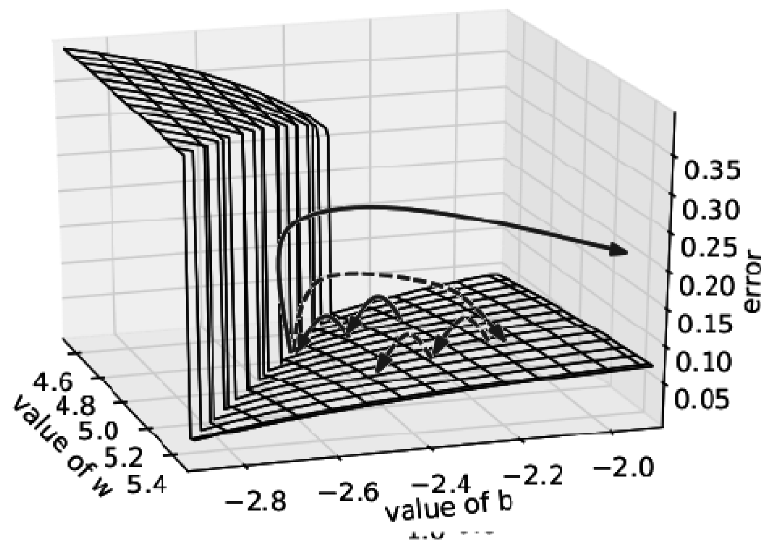

Figure 10.20: Example of the effect of gradient clipping in a recurrent network with two parameters $\boldsymbol{w}$ and $\boldsymbol{b}$. Vertical axis is the objective function to minimize. Note the cliff where the gradient explodes and from where gradient descent can get pushed very far. Clipping the gradient when its norm is above a threshold (Pascanu *et al.*, 2013a) prevents this catastrophic outcome and helps training recurrent nets with long-term dependencies to be captured.

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

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

where v is the norm threshold and $\boldsymbol{g}$ is used to update parameters. The latter method has the advantage that it guarantees that each step is still in the gradient direction, but experiments suggest that both forms work similarly. Althought the parameter update has the same direction as the true gradient, with gradient norm clipping, the parameter update vector norm is now bounded, typically avoiding to perform a detrimental step when gradient explosion occurs. In fact, even simply taking a *random step* when the gradient magnitude is above a threshold tends to work almost as well. Note that even though the per-minibatch direction does not change, the importance given to different minibatches may vary when using norm-clipping. With element-wise clipping, the direction of the update is not anymore aligned with the true gradient, but at least it is still a descent direction.

10.8.8 Regularizing to Encourage Information Flow

Whereas clipping helps dealing with exploding gradients, it does not help with vanishing gradients. To address vanishing gradients and better capture long-term dependencies, we discussed the idea of creating paths in the computational graph of the unfolded recurrent architecture along which the product of gradients associated with arcs is near 1. One approach to achieve this is with LSTMs and other self-loops and gating mechanisms, described above in Section 10.8.4. Another idea is to regularize or constrain the parameters so as to encourage "information flow". In particular, we would like the gradient vector $\nabla_{\boldsymbol{s}_t} L$ being back-propagated to maintain its magnitude (even if there is only a loss at the end of the sequence), i.e., we want

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

to be as large as

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

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

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

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

10.8.9 Organizing the State at Multiple Time Scales

Another promising approach to handle long-term dependencies is the old idea of organizing the state of the RNN at multiple time-scales (El Hihi and Bengio, 1996), with information flowing more easily through long distances at the slower time scales. This is illustrated in Figure 10.21.

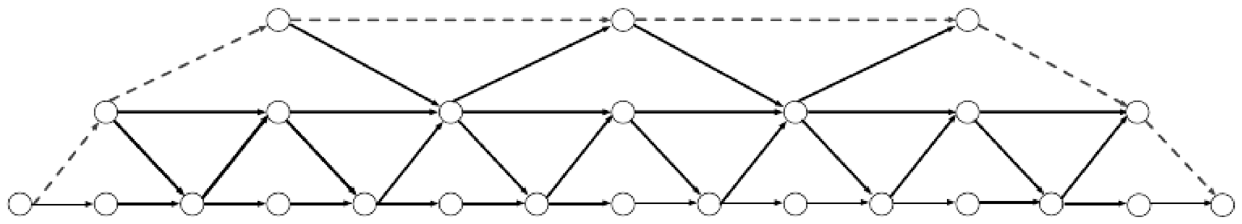

Figure 10.21: Example of a multi-scale recurrent net architecture (unfolded in time), with higher levels operating at a slower time scale. Information can flow unhampered (either forward or backward in time) over longer durations at the higher levels, thus creating long-paths (such as the dotted path) through which long-term dependencies between elements of the input/output sequence can be captured.

There are different ways in which a group of recurrent units can be forced to operate at different time scales. One option is to make the recurrent units leaky (as in Eq. 10.7), but to have different groups of units associated with different fixed time scales. This was the proposal in Mozer (1992) and has been successfully used in Pascanu *et al.* (2013a). Another option is to have explicit and discrete updates taking place at different times, with a different frequency for different groups of units, as in Figure 10.21. This is the approach of El Hihi and Bengio (1996); Koutnik *et al.* (2014) and it also worked well on a number of benchmark datasets.

10.9 Handling Temporal Dependencies with n-grams, HMMs, CRFs and Other Graphical Models

This section regards probabilistic approaches to sequential data modeling which have often been viewed as in competition with RNNs, although RNNs can be seen as a particular form of *dynamic Bayesian networks*[6], as directed graphical models with deterministic latent variables[7].

[6]Dynamic Bayesian networks or dynamic probabilistic networks are directed graphical models for sequential data, with shared parameters across time (Dean and Kanazawa, 1989; Kanazawa *et al.*, 1995; **?**)

[7]Latent variables are random variables that are not directly observed, although they can depend on some that are, and here the dependency is so strong that the latent variables are functions of observed variables

10.9.1 n-grams

n-grams are non-parametric estimators of conditional probabilities based on counting relative frequencies of occurrence, and they have been the core building block of statistical language modeling for many decades (Jelinek and Mercer, 1980; Katz, 1987; Chen and Goodman, 1999). Like RNNs, they are based on the product rule (or chain rule) decomposition of the joint probability into conditionals, Eq. 10.6, which relies on estimates $P(x_t \mid x_{t-1}, \ldots, x_1)$ to compute $P(x_1, \ldots, x_T)$. What is particular of n-grams is that

1. they estimate these conditional probabilities based only on the last $n-1$ values (to predict the next one)
2. they assume that the data is symbolic, i.e., x_t is a symbol taken from a finite alphabet $\mathbb{V}$ (for vocabulary), and
3. the conditional probability estimates are obtained from frequency counts of all the observed length-n subsequences, hence the names unigram (for n=1), bigram (for n=2), trigram (for n=3), and n-gram in general.

The maximum likelihood estimator for these conditional probabilities is simply the relative frequency of occurrence of the left hand symbol in the context of the right hand symbols, compared to all the other possible symbols in $\mathbb{V}$:

$$P(x_t \mid x_{t-1}, \ldots, x_{t-n+1}) = \frac{\#\{x_t, x_{t-1}, \ldots, x_{t-n+1}\}}{\sum_{x'_t \in \mathbb{V}} \#\{x'_t, x_{t-1}, \ldots, x_{t-n+1}\}} \tag{10.17}$$

where $\#\{a, b, c\}$ denotes the cardinality of the set of tuples (a, b, c) in the training set, and where the denominator is also a count (if border effects are handled properly).

A fundamental limitation of the above estimator is that it is very likely to be zero in many cases, even though the tuple $(x_t, x_{t-1}, \ldots, x_{t-n+1})$ may show up in the test set. In that case, the test log-likelihood would be infinitely bad ($-\infty$). To avoid that catastrophic outcome, n-grams employ some form of *smoothing*, i.e., techniques to shift probability mass from the observed tuples to unobserved ones that are similar, a central idea behind most non-parametric statistical methods. See Chen and Goodman (1999) for a review and empirical comparisons. One basic technique consists in assigning a non-zero probability mass to any of the possible next symbol values. Another very popular idea consists in backing off, or mixing (as in mixture model), the higher-order n-gram predictor with all the lower-order ones (with smaller n). 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, i.e., considering the contexts $x_{t-1}, \ldots, x_{t-n+k}$, for increasing k, until a sufficiently reliable estimate is found.

Another interesting idea that is related to neural language models (Section 12.4) is to break up the symbols into classes (by some form of clustering) and back-up to, or mix with, less precise models that only consider the classes of the words in the context (i.e. aggregating statistics from a larger portion of the training set). One can view the word classes as a very impoverished learned representation of words which help to generalize across words of the same class. What distributed representations (e.g. neural word embeddings) bring is a richer notion of similarity by which individual words keep their own identity (instead of being indistinguishable from the other words in the same class) and yet share learned attributes with other words with which they have some elements in common (but not all). This kind of richer notion of similarity makes generalization more specific and the representation not necessarily lossy, unlike with word classes.

10.9.2 Efficient Marginalization and Inference for Temporally Structured Outputs by Dynamic Programming

Many temporal modeling approaches can be cast in the following framework, which also includes hybrids of neural networks with HMMs and conditional random fields (CRFs), first introduced in Bottou *et al.* (1997); LeCun *et al.* (1998c) and later developed and applied with great success in Graves *et al.* (2006); Graves (2012) with the Connectionist Temporal Classification (CTC) approach, as well as in Do and Artières (2010) and other more recent work (Farabet *et al.*, 2013b; Deng *et al.*, 2014). These ideas have been rediscovered in a simplified form (limiting the input-output relationship to a linear one) as CRFs (Lafferty *et al.*, 2001), i.e., undirected graphical models whose parameters are linear functions of input variables. In section 10.10 we consider in more detail the neural network hybrids and the "graph transformer" generalizations of the ideas presented below.

All these approaches (with or without neural nets in the middle) concern the case where we have an input sequence (discrete or continuous-valued) $\{\boldsymbol{x}_t\}$ and a symbolic output sequence $\{y_t\}$ (typically of the same length, although shorter output sequences can be handled by introducing "empty string" values in the output). Generalizations to non-sequential output structure have been introduced more recently (e.g. to condition the Markov Random Fields sometimes used to model structural dependencies in images (Stewart *et al.*, 2007)), at the loss of exact inference (the dynamic programming methods described below).

Optionally, one also considers a latent variable sequence $\{s_t\}$ that is also discrete and inference needs to be done over $\{s_t\}$, either via marginalization (summing over all possible values of the state sequence) or maximization, i.e., picking exactly or approximately the so-called MAP sequence, the one with the largest probability, given the input. If the state variables s_t and the target variables y_t have a 1-D Markov structure to their dependency, then computing likelihood, partition function and MAP values can all be done efficiently by exploiting dynamic programming to factorize the computation. On the other hand, if the state or output sequence dependencies are captured by an RNN, then there is no finite-order Markov property and no efficient and exact inference is generally possible. However, many reasonable approximations have been used in the past, such as with variants of the beam search algorithm (Lowerre, 1976). The idea of beam search is that one maintains a set of promising candidate paths that end at some time step t. For each additional time step, one considers extensions to $t + 1$ of each of these paths and then prunes those with the worse overall cumulative score (up to $t + 1$). The beam size is the number of candidates that are kept. See Section 10.10.1 for more details on beam search.

The application of the principle of dynamic programming in these setups is the same as what is used in the Forward-Backward algorithm (detailed more around Eq. 10.21), for graphical models and HMMs (detailed more in Section 10.9.3) and the Viterbi algorithm detailed below (Eq. 10.23). For both of these algorithms, we are trying to sum (Forward-Backward algorithm) or maximize (Viterbi algorithm) over paths the probability or score associated with each path.

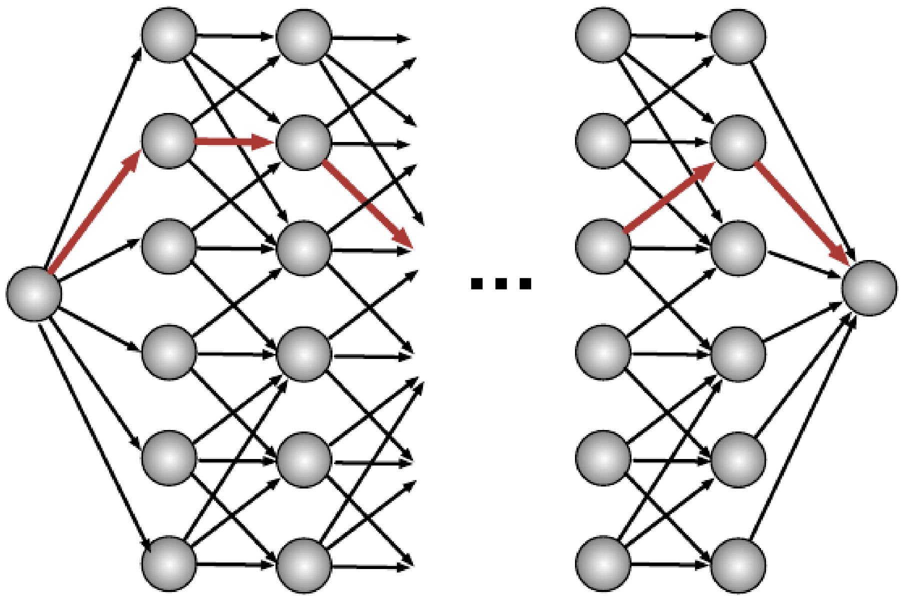

Figure 10.22: Example of a temporally structured output graph, as can be found in CRFs, HMMs, and neural net hybrids. Each node corresponds to a particular **value** of an output random variable at a particular point in the output sequence (contrast with a graphical model representation, where each node corresponds to a random variable). A path from the source node to the sink node (e.g. red bold arrows) corresponds to an interpretation of the input as a sequence of output labels. The dynamic programming recursions that are used for computing likelihood (or conditional likelihood) or performing MAP inference (finding the best path) involve sums or maximizations over sub-paths ending at one of the particular interior nodes.

Let $\mathcal{G}$ be a directed acyclic graph whose paths correspond to the sequences that can be selected (for MAP) or summed over (marginalized for computing a likelihood), as illustrated in Figure 10.22. In the above example, let z_t represent the choice variable (e.g., s_t and y_t in the above example), and each arc with score a corresponds to a particular value of z_t in its Markov context. In the language of undirected graphical models, if a is the score associated with an arc from the node for $z_{t-1} = j$ to the one for $z_t = i$, then a is minus the energy of a term of the energy function associated with the event $1_{z_{t-1}=j, z_t=i}$ and the associated information from the input $\boldsymbol{x}$ (e.g. some value of x_t).

Hidden Markov models are based on the notion of Markov chain, which is covered in much more detail in Section 14.1. A Markov chain is a sequence of random variables $\mathrm{z}_1, \dots \mathrm{z}_T$, and for our purposes the main property of a Markov chain chain of *order 1* is that the current value of z_t contains enough information about the previous values $z_1, \dots z_{t-1}$ in order to predict the distribution of the next random variable, z_{t+1}. In our context, we can make the z's conditioned on some $\boldsymbol{x}$, the order 1 Markov property then means that $P(z_t \mid z_{t-1}, z_{t-2}, \dots, z_1, \boldsymbol{x}) = P(z_t \mid z_{t-1}, \boldsymbol{x})$, where $\boldsymbol{x}$ is conditioning information (the input sequence). When we consider a path in that space, i.e. a sequence of values, we draw a graph with a node for each discrete value of z_t, and if it is possible to transition from $z_{t-1} = j$ to $z_t = i$ we draw an arc between these two nodes. Hence, the total number of nodes in the graph would be equal to the length of the sequence, T, times the number of values of z_t, n, and the number of arcs of the graph would be up to Tn^2 (if every value of z_t can follow every value of z_{t-1}, although in practice the connectivity is often much smaller because not all transitions are typically feasible). A score a is computed for each arc (which may include some component that only depends on the source or only on the destination node), as a function of the conditioning information $\boldsymbol{x}$. The inference or marginalization problems involve performing the following computations.

For the **marginalization** task, we want to compute the sum over all complete paths (e.g. from source to sink) of the product along the path of the exponentiated scores associated with the arcs on that path:

$$m(\mathcal{G}) = \sum_{\text{path} \in \mathcal{G}} \prod_{a \in \text{path}} e^a \tag{10.18}$$

where the product is over all the arcs on a path (with score a), and the sum is over all the paths associated with complete sequences (from beginning to end of a sequence). $m(\mathcal{G})$ may correspond to a likelihood, numerator or denominator of a probability. For example,

$$P(\{z_t\} \in \mathbb{Y} \mid \boldsymbol{x}) = \frac{m(\mathcal{G}_{\mathbb{Y}})}{m(\mathcal{G})} \tag{10.19}$$

where $\mathcal{G}_{\mathbb{Y}}$ is the subgraph of $\mathcal{G}$ which is restricted to sequences that are compatible with some target answer $\mathbb{Y}$.

For the **inference** task, we want to compute

$$\begin{aligned}\pi(\mathcal{G}) &= \operatorname*{arg\,max}_{\text{path}\in\mathcal{G}} \prod_{a\in\text{path}} e^{a} = \operatorname*{arg\,max}_{\text{path}\in\mathcal{G}} \sum_{a\in\text{path}} a \\ v(\mathcal{G}) &= \max_{\text{path}\in\mathcal{G}} \sum_{a\in\text{path}} a\end{aligned}$$

where $\pi(\mathcal{G})$ is the most probable path and $v(\mathcal{G})$ is its log-score or value, and again the set of paths considered includes all of those starting at the beginning and ending at the end the sequence.

The principle of dynamic programming is to recursively compute intermediate quantities that can be reused efficiently so as to avoid actually going through an exponential number of computations, e.g., though the exponential number of paths to consider in the above sums or maxima. Note how it is already at play in the underlying efficiency of back-propagation (or back-propagation through time), where gradients w.r.t. intermediate layers or time steps or nodes in a flow graph can be computed based on previously computed gradients (for later layers, time steps or nodes). Here it can be achieved by considering to restrictions of the graph to those paths that end at a node n, which we denote $\mathcal{G}^n$. $\mathcal{G}^n_Y$ indicates the additional restriction to subsequences that are compatible with the target sequence Y, i.e., with the beginning of the sequence Y.

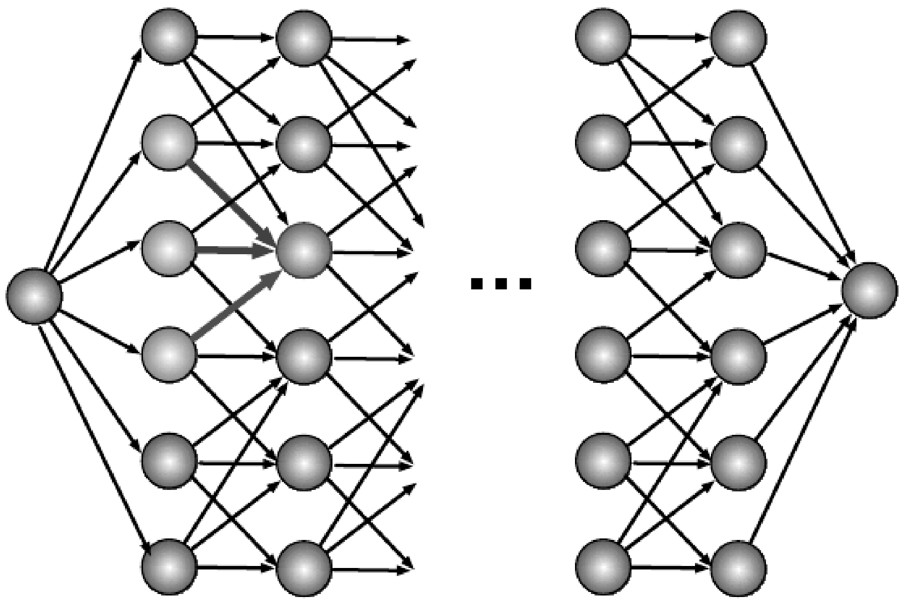

Figure 10.23: Illustration of the recursive computation taking place for inference or marginalization by dynamic programming. See Figure 10.22. These recursions involve sums or maximizations over sub-paths ending at one of the particular interior nodes (red in the figure), each time only requiring to look up previously computed values at the predecessor nodes (green).

We can thus perform marginalization efficiently as follows, and illustrated in Figure 10.23. This is a generalization of the so-called Forward-Backward algorithm for HMMs

$$m(\mathcal{G}) = \sum_{n \in \text{final}(\mathcal{G})} m(\mathcal{G}^n) \tag{10.20}$$

where final($\mathcal{G}$) is the set of final nodes in the graph $\mathcal{G}$, and we can recursively compute the node-restricted sum via

$$m(G^n) = \sum_{n' \in \text{pred}(n)} m(\mathcal{G}^{n'}) e^{a_{n',n}} \tag{10.21}$$

where pred(n) is the set of predecessors of node n in the graph and $a_{m,n}$ is the log-score associated with the arc from m to n. It is easy to see that expanding the above recursion recovers the result of Eq. 10.18.

Similarly, we can perform efficient MAP inference (also known as Viterbi decoding) as follows.

$$v(\mathcal{G}) = \max_{n \in \text{final}(\mathcal{G})} v(G^n) \tag{10.22}$$

and

$$v(\mathcal{G}^n) = \max_{m \in \text{pred}(n)} v(\mathcal{G}^m) + a_{m,n}. \tag{10.23}$$

To obtain the corresponding path, it is enough to keep track of the argmax associated with each of the above maximizations and trace back $\pi(\mathcal{G})$ starting from the nodes in final$(\mathcal{G})$. For example, the last element of $\pi(\mathcal{G})$ is

$$n^* \leftarrow \arg\max_{n \in \text{final}(\mathcal{G})} v(\mathcal{G}^n)$$

and (recursively) the argmax node before n^* along the selected path is a new n^*,

$$n^* \leftarrow \arg\max_{m \in \text{pred}(n^*)} v(\mathcal{G}^m) + a_{m,n^*},$$

etc. Keeping track of these n^* along the way gives the selected path. Proving that these recursive computations yield the desired results is straightforward and left as an exercise.

10.9.3 HMMs

Hidden Markov Models (HMMs) are probabilistic models of sequences that were introduced in the 60's (Baum and Petrie, 1966) along with the E-M algorithm (Section 19.2). They are very commonly used to model sequential structure, in particular having been since the mid 80's and until recently the technological core of speech recognition systems (Rabiner and Juang, 1986; Rabiner, 1989). Just like RNNs, HMMs are dynamic Bayes nets (Koller and Friedman, 2009), i.e., the same parameters and graphical model structure are used for every time step. Compared to RNNs, what is particular to HMMs is that the latent variable associated with each time step (called the *state*) is discrete, with a separate set of parameters associated with each state value. We consider here the most common form of HMM, in which the Markov chain is of order 1, i.e., the state s_t at time t, given the previous states, only depends on the previous state s_{t-1}:

$$P(s_t \mid s_{t-1}, s_{t-2}, \ldots, s_1) = P(s_t \mid s_{t-1}),$$

which we call the *transition or state-to-state distribution.* Generalizing to higher-order Markov chains is straightforward: for example, order-2 Markov chains can be mapped to order-1 Markov chains by considering as order-1 "states" all the pairs $(s_t = i, s_{t-1} = j)$.

Given the state value, a generative probabilistic model of the visible variable $\boldsymbol{x}_t$ is defined, that specifies how each observation $\boldsymbol{x}_t$ in a sequence $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)$ can be generated, via a model $P(\boldsymbol{x}_t \mid s_t)$. Two kinds of parameters are distinguished: those that define the transition distribution, which can be given by a matrix

$$A_{ij} = P(s_t = i \mid s_{t-1} = j),$$

and those that define the output model $P(\boldsymbol{x}_t \mid s_t)$. For example, if the data are discrete and $\boldsymbol{x}_t$ is a symbol x_t, then another matrix can be used to define the output (or emission) model:

$$B_{ki} = P(x_t = k \mid s_t = i).$$

Another common parametrization for $P(\boldsymbol{x}_t \mid s_t = i)$, in the case of continuous vector-valued $\boldsymbol{x}_t$, is the Gaussian mixture model, where we have a different mixture (with its own means, covariances and component probabilities) for each state $s_t = i$. Alternatively, the means and covariances (or just variances) can be shared across states, and only the component probabilities are state-specific.

The overall likelihood of an observed sequence is thus

$$P(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T) = \sum_{s_1, s_2, \ldots, s_T} \prod_t P(\boldsymbol{x}_t \mid s_t) P(s_t \mid s_{t-1}). \tag{10.24}$$

In the language established earlier in Section 10.9.2, we have a graph $\mathcal{G}$ with one node n per time step t and state value i, i.e., for $s_t = i$, and one arc between each node n (for $\mathbf{1}_{s_t=i}$) and its predecessors m for $\mathbf{1}_{s_{t-1}=j}$ (when the transition probability is non-zero, i.e., $P(s_t = i \mid s_{t-1} = j) \neq 0$). Following Eq. 10.24, the log-score $a_{m,n}$ for the transition between m and n would then be

$$a_{m,n} = \log P(x_t \mid s_t = i) + \log P(s_t = i \mid s_{t-1} = j).$$

As explained in Section 10.9.2, this view gives us a dynamic programming algorithm for computing the likelihood (or the conditional likelihood given some constraints on the set of allowed paths), called the forward-backward or sum-product algorithm, in time $O(kNT)$ where T is the sequence length, N is the number of states and k the average in-degree of each node.

Although the likelihood is tractable and could be maximized by a gradient-based optimization method, HMMs are typically trained by the E-M algorithm (Section 19.2), which has been shown to converge rapidly (approaching the rate of Newton-like methods) in some conditions (if we view the HMM as a big mixture, then the condition is for the final mixture components to be well-separated, i.e., have little overlap) (Xu and Jordan, 1996).

At test time, the sequence of states that maximizes the joint likelihood

$$P(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T, s_1, s_2, \ldots, s_T)$$

can also be obtained using a dynamic programming algorithm (called the Viterbi algorithm). This is a form of *inference* (see Section 13.5) that is called MAP (Maximum A Posteriori) inference because we want to find the most probable value of the unobserved state variables given the observed inputs. Using the same definitions as above (from Section 10.9.2) for the nodes and log-score of the graph $\mathcal{G}$ in which we search for the optimal path, the Viterbi algorithm corresponds to the recursion defined by Eq. 10.23.

If the HMM is structured in such a way that states have a meaning associated with labels of interest, then from the MAP sequence one can read off the associated labels. When the number of states is very large (which happens for example with large-vocabulary speech recognition based on n-gram language models), even the efficient Viterbi algorithm becomes too expensive, and approximate search must be performed. A common family of search algorithms for HMMs is the *beam search* algorithm (Lowerre, 1976) (Section 10.10.1).

More details about speech recognition are given in Section 12.3. An HMM can be used to associate a sequence of labels $(y_1, y_2, \ldots, y_N)$ with the input $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)$, where the output sequence is typically shorter than the input sequence, i.e., $N < T$. Knowledge of $(y_1, y_2, \ldots, y_N)$ constrains the set of compatible state sequences $(s_1, s_2, \ldots, s_T)$, and the generative conditional likelihood

$$P(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T \mid y_1, y_2, \ldots, y_N) = \sum_{s_1, s_2, \ldots, s_T \in \mathcal{S}(y_1, y_2, \ldots, y_N)} \prod_t P(\boldsymbol{x}_t \mid s_t) P(s_t \mid s_{t-1}). \tag{10.25}$$

can be computed using the same forward-backward technique, and its logarithm maximized during training, as discussed above.

Various discriminative alternatives to the generative likelihood of Eq. 10.25 have been proposed (Brown, 1987; Bahl *et al.*, 1987; Nadas *et al.*, 1988; Juang and Katagiri, 1992; Bengio *et al.*, 1992a; Bengio, 1993; Leprieur and Haffner, 1995; Bengio, 1999a), the simplest of which is simply $P(y_1, y_2, \ldots, y_N \mid \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)$, which is obtained from Eq. 10.25 by Bayes rule, i.e., involving a normalization over all sequences, i.e., the unconstrained likelihood of Eq. 10.24:

$$P(y_1, y_2, \ldots, y_N \mid \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T) = \frac{P(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T \mid y_1, y_2, \ldots, y_N) P(y_1, y_2, \ldots, y_N)}{P(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T)}.$$

Both the numerator and denominator can be formulated in the framework of the previous section (Eqs. 10.19-10.21), where for the numerator we merge (add) the log-scores coming from the structured output output model $P(y_1, y_2, \ldots, y_N)$ and from the input likelihood model $P(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T \mid y_1, y_2, \ldots, y_N)$. Again, each node of the graph corresponds to a state of the HMM at a particular time step t (which may or may not emit the next output symbol y_i), associated with an input vector $\boldsymbol{x}_t$. Instead of making the relationship to the input the result of a simple parametric form (Gaussian or multinomial, typically), the scores can be computed by a neural network (or any other parametrized differential function). This gives rise to discriminative hybrids of search or graphical models with neural networks, discussed below, Section 10.10.

10.9.4 CRFs

Whereas HMMs are typically trained to maximize the probability of an input sequence $\mathbf{x}$ given a target sequence $\mathbf{y}$ and correspond to a directed graphical model, Conditional Random Fields (CRFs) (Lafferty *et al.*, 2001) are *undirected* graphical models that are trained to maximize the joint probability of the target variables, given input variables, $P(\mathbf{y} \mid \mathbf{x})$. CRFs are special cases of the graph transformer model introduced in Bottou *et al.* (1997); LeCun *et al.* (1998c), where neural nets are replaced by affine transformations and there is a single graph involved. A *graph transformer* is an computational module that transforms a weighted (with a scalar on each arc) directed acyclic graph into another one. Examples of graph transformers are illustrated in Figure 10.24. In this context, graph transformers can be seen as transforming the set of weights in their input graph into a set of weights on their output graph, typically so that we can compute derivatives of the output graph weights with respect to input graph weights, i.e., we can back-propagated costs through the graph transformer. Section 10.10 below provides more discussion and examples.

Many applications of CRFs involve sequences and the discussion here will be focused on this type of application, although applications to images (e.g. for image segmentation) are also common. Compared to other graphical models, another characteristic of CRFs is that there are no latent variables. The general equation for the probability distribution modeled by a CRF is basically the same as for fully visible (not latent variable) undirected graphical models, also known as Markov Random Fields (MRFs, see Section 13.2.2), *except* that the "potentials" (terms of the energy function) are parametrized functions of the input variables, and the likelihood of interest is the posterior probability $P(\mathbf{y} \mid \mathbf{x})$.

As in many other MRFs, CRFs often have a particular connectivity structure in their graph, which allows one to perform learning or inference more efficiently. In particular, when dealing with sequences, the energy function typically only has terms that relate neighboring elements of the sequence of target variables. For example, the target variables could form a homogeneous[8] Markov chain of order k (given the input variables). A typical linear CRF example with binary outputs would have the following structure:

$$P(\mathbf{y} = \boldsymbol{y} \mid \mathbf{x}) = \frac{1}{Z} \exp\left(\sum_t y_t \left(b + \sum_j w_i x_{tj}\right) + \sum_{i=1}^{k} y_t y_{t-i}\left(u_i + \sum_j v_{ij} x_{tj}\right) \right) \tag{10.26}$$

where Z is the normalization constant, which is the sum over all $\boldsymbol{y}$ sequences of the numerator. In that case, the score marginalization framework of Section 10.9.2

[8] meaning that the same parameters are used for every time step

and coming from Bottou *et al.* (1997); LeCun *et al.* (1998c) can be applied by making terms in the above exponential correspond to scores associated with nodes t of a graph $\mathcal{G}$. If there were more than two output classes, more nodes per time step would be required but the principle would remain the same. A more general formulation for Markov chains of order d is the following:

$$P(\mathbf{y} = \boldsymbol{y} \mid \mathbf{x}) = \frac{1}{Z} \exp \quad \sum_t \sum_{d=0}^{d} f_{d'}(y_t, y_{t-1}, \ldots, y_{t-d'}, x_t) \Bigg) \tag{10.27}$$

where $f_{d'}$ computes a potential of the energy function, a parametrized function of both the past target values (up to $y_{t-d'}$) and of the current input value x_t. For example, as discussed below $f_{d'}$ could be the output of an arbitrary parametrized computation, such as a neural network.

Although Z looks intractable, because of the Markov property of the model (order 1, in the example), it is again possible to exploit dynamic programming to compute Z efficiently, as per Eqs. 10.19-10.21). Again, the idea is to compute the sub-sum for sequences of length $t \leq T$ (where T is the length of a target sequence $\boldsymbol{y}$), ending in each of the possible state values at t, e.g., $y_t = 1$ and $y_t = 0$ in the above example. For higher order Markov chains (say order d instead of 1) and a larger number of state values (say N instead of 2), the required sub-sums to keep track of are for each element in the cross-product of $d-1$ state values, i.e., N^{d-1}. For each of these elements, the new sub-sums for sequences of length $t+1$ (for each of the N values at $t+1$ and corresponding $N^{\max(0,d-2)}$ past values for the past $d-2$ time steps) can be obtained by only considering the sub-sums for the N^{d-1} joint state values for the last $d-1$ time steps before $t+1$.

Following Eq. 10.23, the same kind of decomposition can be performed to efficiently find the MAP configuration of y's given $\boldsymbol{x}$, where instead of products (sums inside the exponential) and sums (for the outer sum of these exponentials, over different paths) we respectively have sums (corresponding to adding the sums inside the exponential) and maxima (across the different competing "previous-state" choices).

10.10 Combining Neural Networks and Search

The idea of combining neural networks with HMMs or related search or alignment-based components (such as graph transformers) for speech and handwriting recognition dates from the early days of research on multi-layer neural networks (Bourlard and Wellekens, 1990; Bottou *et al.*, 1990; Bengio, 1991; Bottou, 1991; Haffner *et al.*, 1991; Bengio *et al.*, 1992a; Matan *et al.*, 1992; Bourlard and Morgan, 1993; Bengio *et al.*, 1995; Bengio and Frasconi, 1996; Baldi and Brunak, 1998) – and

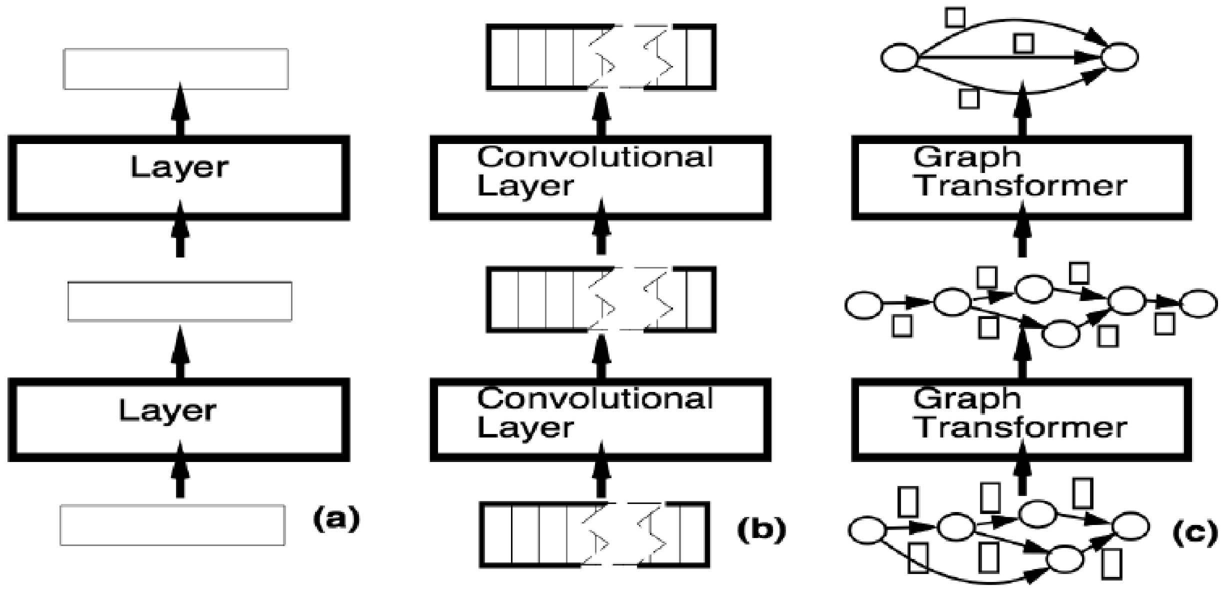

Figure 10.24: Illustration of the stacking of graph transformers (right, c) as a generalization of the stacking of convolutional layers (middle, b) or of regular feedforward layers that transform fixed-size vectors (left, a). Figure reproduced with permission from the authors of Bottou *et al.* (1997). Quoting from that paper, (c) shows that "multilayer graph transformer networks are composed of trainable modules that operate on and produce graphs whose arcs carry numerical information".

see more references in Bengio (1999b). See also 12.5 for combining recurrent and other deep learners with generative models such as CRFs, GSNs or RBMs.

The principle of efficient marginalization and inference for temporally structured outputs by exploiting dynamic programming (Sec. 10.9.2) can be applied not just when the log-scores of Eqs. 10.18 and 10.20 are parameters or linear functions of the input, but also when they are *learned non-linear functions* of the input, e.g., via a neural network transformation, as was first done in Bottou *et al.* (1997); LeCun *et al.* (1998c). These papers additionally introduced the powerful idea of *learned graph transformers*, illustrated in Figure 10.24. In this context, a graph transformer is a machine that can *map a directed acyclic graph* $\mathcal{G}_{\text{in}}$ *to another graph* $\mathcal{G}_{\text{out}}$. Both input and output graphs have paths that represent hypotheses about the observed data.

For example, in the above papers, and as illustrated in Figure 10.25, a segmentation graph transformer takes a singleton input graph (the image $\boldsymbol{x}$) and outputs a graph representing segmentation hypotheses (regarding sequences of segments that could each contain a character in the image). Such a graph transformer could be used as one layer of a *graph transformer network* for handwriting recognition or document analysis for reading amounts on checks, as illustrated respectively in Figures 10.26 and 10.27.

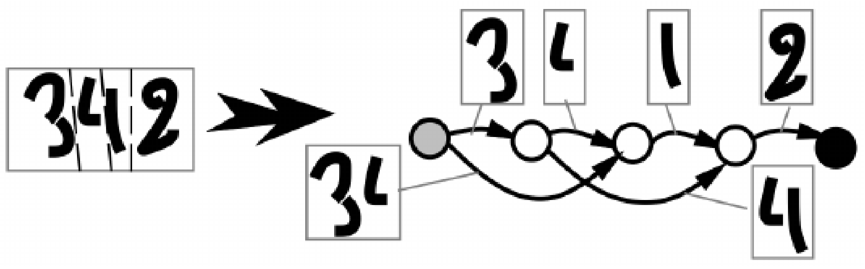

Figure 10.25: Illustration of the input and output of a simple graph transformer that maps a singleton graph corresponding to an input image to a graph representing hypothesized segmentation hypotheses. Reproduced with permission from the authors of Bottou *et al.* (1997).

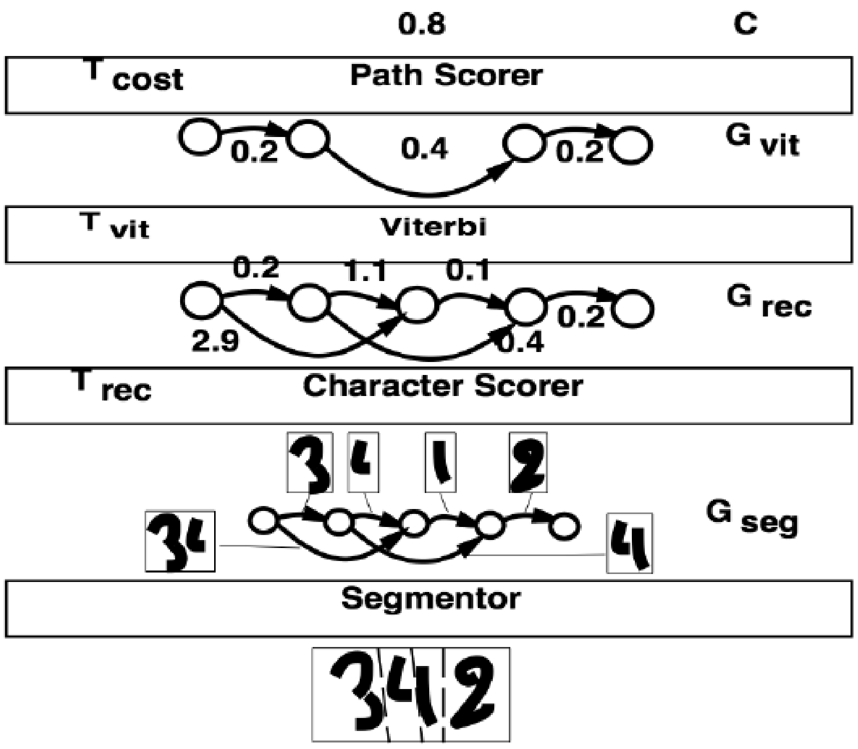

Figure 10.26: Illustration of the graph transformer network that has been used for finding the best segmentation of a handwritten word, for handwriting recognition. Reproduced with permission from Bottou *et al.* (1997).

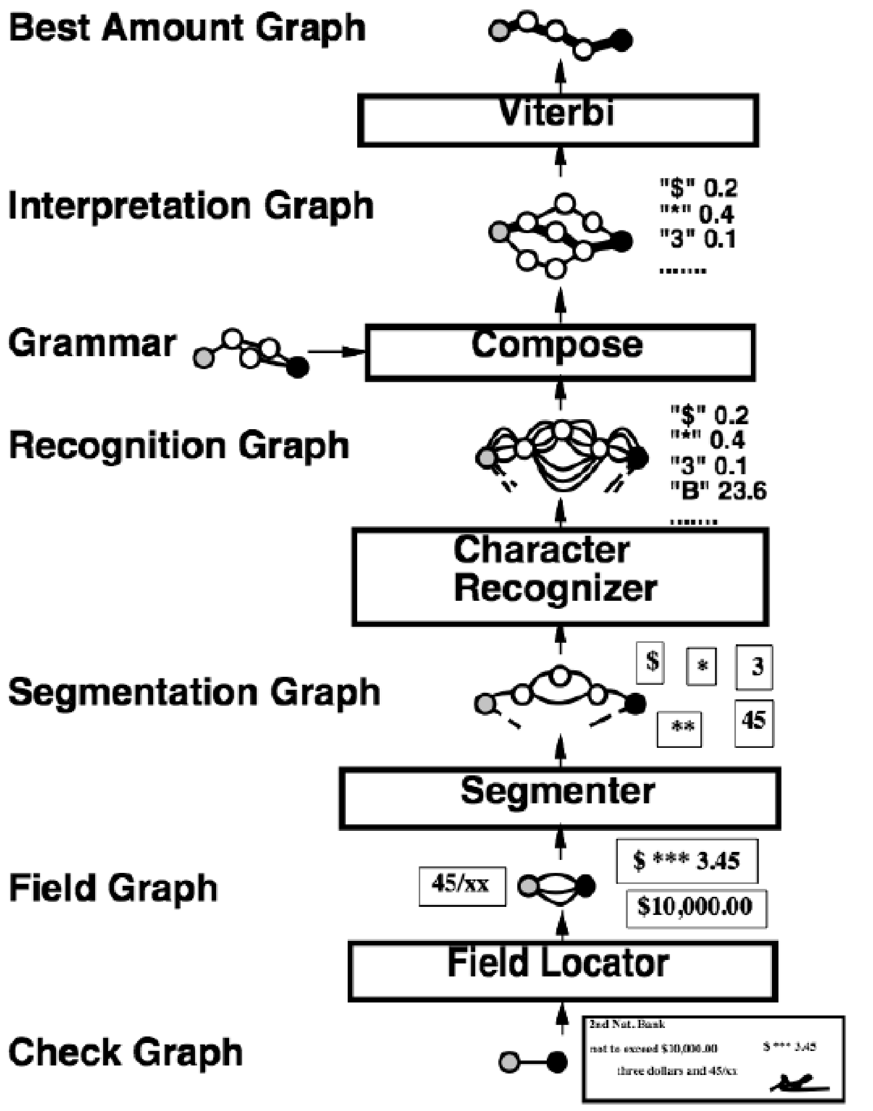

Figure 10.27: Illustration of the graph transformer network that has been used for reading amounts on checks, starting from the single graph containing the image of the graph to the recognized sequences of characters corresponding to the amount on the graph, with currency and other recognized marks. Note how the grammar graph transformer composes the grammar graph (allowed sequences of characters) and the recognition graph (with character hypotheses associated with specific input segments, on the arcs) into an interpretation graph that only contains the recognition graph paths that are compatible with the grammar. Reproduced with permission from Bottou *et al.* (1997).

For example, after the segmentation graph transformer, a recognition graph transformer could expand each node of the segmentation graph into a subgraph whose arcs correspond to different interpretations of the segment (which character is present in the segment?). Then, a dictionary graph transformer takes the recognition graph and expands it further by considering only the sequences of characters that are compatible with sequences of words in the language of interest. Finally, a language-model graph transformer expands sequences of word hypotheses so as to include multiple words in the state (context) and weigh the arcs according to the language model next-word log-probabilities.

Each of these transformations is parametrized and takes real-valued scores on the arcs of the input graph into real-valued scores on the arcs of the output graph. These transformations can be parametrized and learned by gradient-based optimization over the whole series of graph transformers.

10.10.1 Approximate Search

Unfortunately, as in the above example, when the number of nodes of the graph becomes very large (e.g., considering all previous n words to condition the log-probability of the next one, for n large), even dynamic programming (whose computation scales with the number of arcs) is too slow for practical applications such as speech recognition or machine translation. A common example is when a recurrent neural network is used to compute the arcs log-score, e.g., as in neural language models (Section 12.4). Since the prediction at step t depends on all $t-1$ previous choices, the number of states (nodes of the search graph $\mathcal{G}$) grows exponentially with the length of the sequence. In that case, one has to resort to *approximate search*.

Beam Search

In the case of sequential structures as discussed in this chapter, a common family of approximate search algorithms is the *beam search* (Lowerre, 1976).

- Break the nodes of the graph into g groups containing only "comparable nodes", e.g., the group of nodes n for which the maximum length of the paths ending at n is exactly t.

- Process these groups of nodes sequentially, keeping only at each step t a selected subset $\mathbb{S}_t$ of the nodes (the "beam"), chosen based on the subset $\mathbb{S}_{t-1}$. Each node in $\mathbb{S}_t$ is associated with a score $\hat{v}(\mathcal{G}^n)$ that represents an approximation (a lower bound) on the maximum total log-score of the path ending at the node, $v(\mathcal{G}^n)$ (defined in Eq. 10.23, Viterbi decoding).

- $\mathbb{S}_t$ is obtained by following all the arcs from the nodes in $\mathbb{S}_{t-1}$, and sorting all the resulting group t nodes n according to their estimated (lower bound) score
$$\hat{v}(\mathcal{G}^n) = \max_{n' \in \mathbb{S}_{t-1} and n' \in \text{pred}(n)} \hat{v}(\mathcal{G}^{n'}) + a_{n',n},$$
while keeping track of the argmax in order to trace back the estimated best path. Only the k nodes with the highest log-score are kept and stored in $\mathbb{S}_t$, and k is called the *beam width.*

- The estimated best final node can be read off from $\max_{n \in \mathbb{S}_T} \hat{v}(\mathcal{G}^n)$ and the estimated best path from the associated argmax choices made along the way, just like in the Viterbi algorithm.

One problem with beam search is that the beam often ends up lacking in diversity, making the approximation poor. For example, imagine that we have two "types" of solutions, but that each type has exponentially many variants (as a function of t), due, e.g., to small independent variations in ways in which the type can be expressed at each time step t. Then, even though the two types may have close best log-score up to time t, the beam could be dominated by the one that wins slightly, eliminating the other type from the search, although later time steps might reveal that the second type was actually the best one.