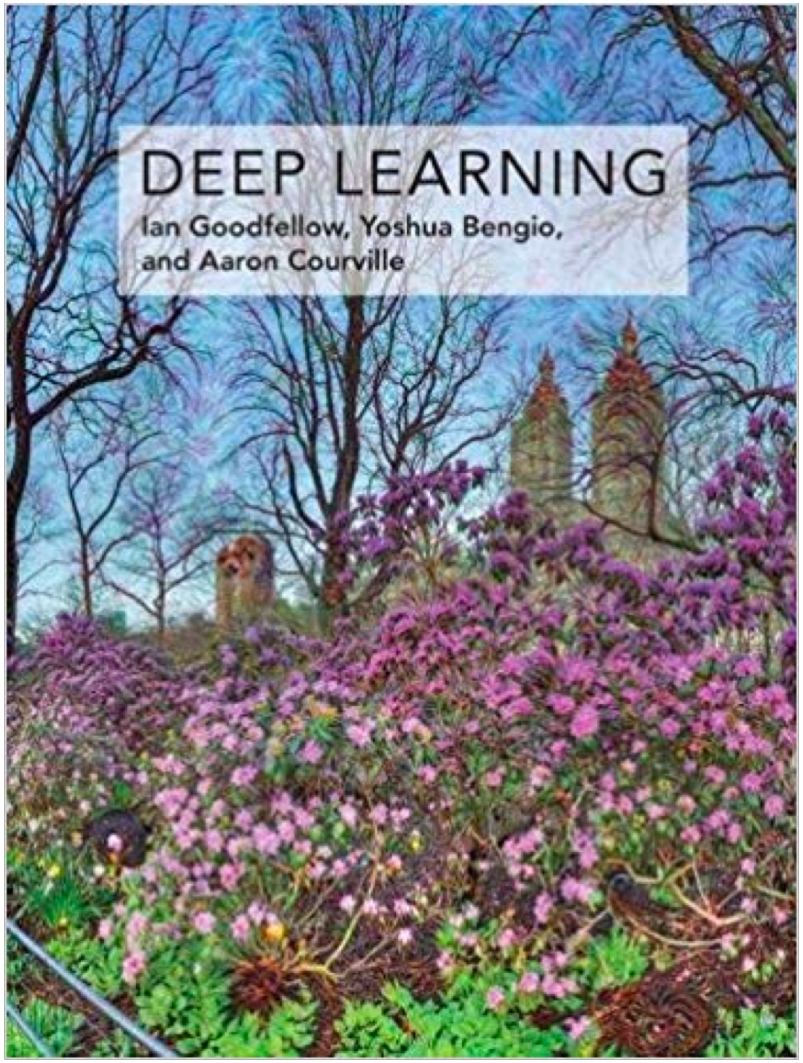
國立交通大學
National Chiao Tung University

Deep Learning
深度學習
Fall 2018

Sequence modeling: recurrent and recursive nets

(Chapter 10)

Prof. Chia-Han Lee
李佳翰 副教授

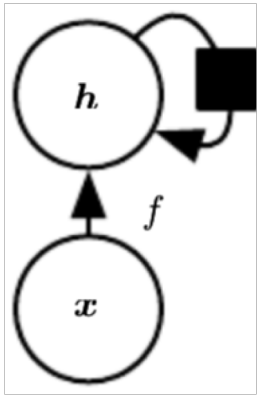


Recurrent neural networks

- A recurrent neural network, or RNN, is a neural network that is specialized for processing a sequence of values $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$.

- Recurrent networks can scale to much longer sequences than would be practical for networks without sequence-based specialization.

- Most recurrent networks can also process sequences of variable length.



Recurrent neural networks

- To go from multilayer networks to recurrent networks, we need to take advantage of the idea of sharing parameters across different parts of a model.

- Parameter sharing makes it possible to extend and apply the model to examples of different forms (different lengths, here) and generalize across them. If we had separate parameters for each value of the time index, we could not generalize to sequence lengths not seen during training, nor share statistical strength across different sequence lengths and across different positions in time.

- Such sharing is particularly important when a specific piece of information can occur at multiple positions within the sequence.

Recurrent neural networks

- For example, consider the two sentences “I went to Nepal in 2009” and “In 2009, I went to Nepal.” If we ask a machine learning model to read each sentence and extract the year in which the narrator went to Nepal, we would like it to recognize the year 2009 as the relevant piece of information, whether it appears in the sixth word or in the second word of the sentence.

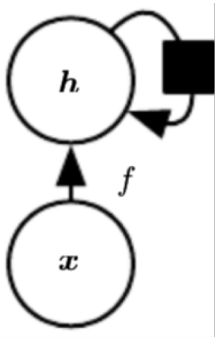
- A traditional fully connected feedforward network would need to learn all the rules of the language separately at each position in the sentence.

- By comparison, a recurrent neural network shares the same weights across several time steps.

Recurrent neural networks

- The convolution across a 1-D temporal sequence allows a network to share parameters across time but is shallow. The output of convolution is a sequence where each member of the output is a function of a small number of neighboring members of the input.

- Recurrent networks share parameters in a different way. Each member of the output is produced using the same update rule applied to the previous outputs. This recurrent formulation results in the sharing of parameters through a very deep computational graph.



Recurrent neural networks

- For the simplicity of exposition, we refer to RNNs as operating on a sequence that contains vectors $\boldsymbol{x}^{(t)}$ with the time step index t ranging from 1 to τ.

- The time step index need not literally refer to the passage of time in the real world. Sometimes it refers only to the position in the sequence.

- RNNs may also be applied in two dimensions across spatial data such as images, and even when applied to data involving time, the network may have connections that go backward in time, provided that the entire sequence is observed before it is provided to the network.

Unfolding computational graphs

- Unfolding a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events, results in the sharing of parameters across a deep network structure.

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

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

where $\boldsymbol{s}^{(t)}$ is called the state of the system. Equation 10.1 is recurrent because the definition of $\boldsymbol{s}$ at time t refers back to the same definition at time $t - 1$.

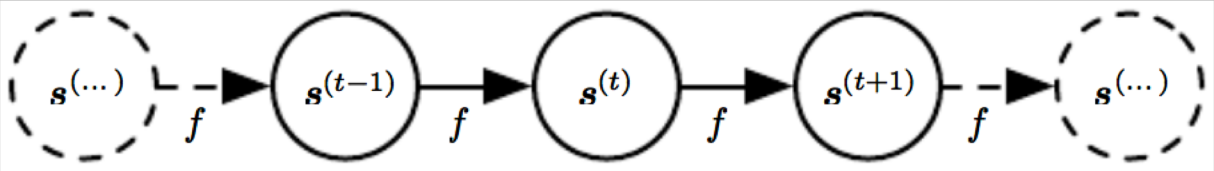
Unfolding computational graphs

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

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

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

- Unfolding the equation by repeatedly applying the definition in this way has yielded an expression that does not involve recurrence. Such an expression can now be represented by a traditional directed acyclic computational graph.



Unfolding computational graphs

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

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

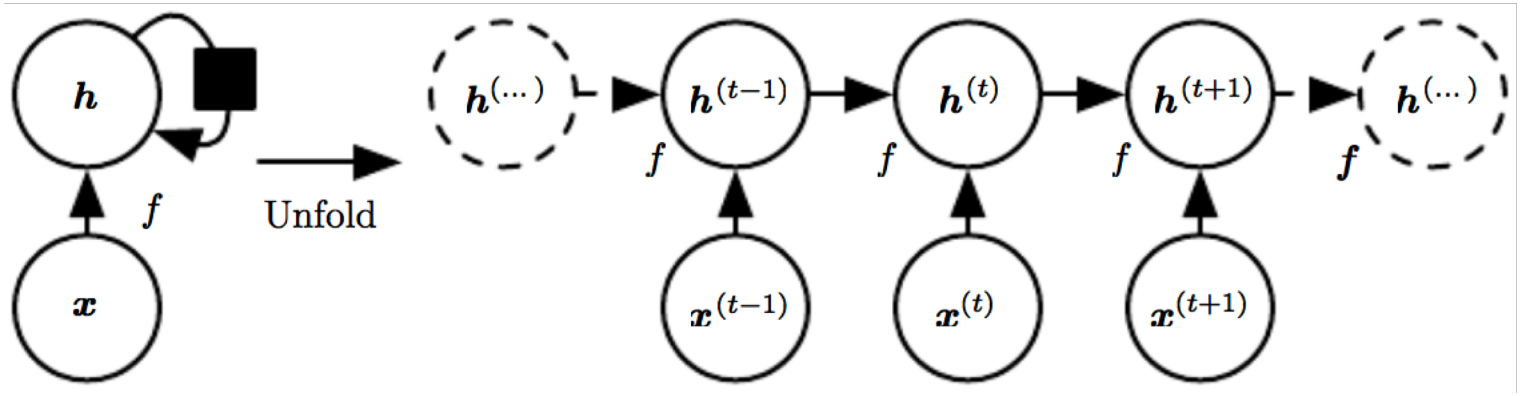
where we see that the state now contains information about the whole past sequence.

- Recurrent neural networks can be built in many different ways. Essentially any function involving recurrence can be considered a recurrent neural network.

Unfolding computational graphs

- To indicate that the state is the hidden units of the network, we now rewrite equation 10.4 using the variable h to represent the state,

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



- Typical RNNs will add extra architectural features such as output layers that read information out of the state $\boldsymbol{h}$ to make predictions.

Unfolding computational graphs

- When the recurrent network is trained to perform a task that requires predicting the future from the past, the network typically learns to use $\boldsymbol{h}^{(t)}$ as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t.

- This summary is in general necessarily lossy, since it maps an arbitrary length sequence $\left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right)$ to a fixed length vector $\boldsymbol{h}^{(t)}$.

- Depending on the training criterion, this summary might selectively keep some aspects of the past sequence with more precision than other aspects.

Unfolding computational graphs

- For example, if the RNN is used in statistical language modeling, typically to predict the next word given previous words, storing all the information in the input sequence up to time t may not be necessary; storing only enough information to predict the rest of the sentence is sufficient.

- The most demanding situation is when we ask $\boldsymbol{h}^{(t)}$ to be rich enough to allow one to approximately recover the input sequence, as in autoencoder frameworks.

Unfolding computational graphs

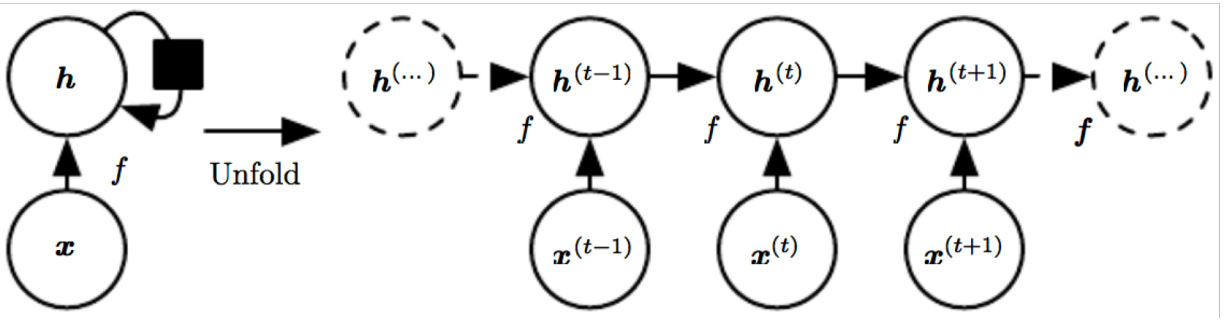
$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \tag{10.5}$$

- Equation 10.5 can be drawn in two different ways.
- One way to draw the RNN is with a diagram containing one node for every component that might exist in a physical implementation of the model.
- In this view, the network defines a circuit that operates in real time, with physical parts whose current state can influence their future state.
- We use a black square to indicate that an interaction takes place with a delay of a single time step, from the state at time t to the state at time $t + 1$.

Unfolding computational graphs

- The other way to draw the RNN is as an unfolded computational graph, in which each component is represented by many different variables, with one variable per time step, representing the state of the component at that point in time.

- Each variable for each time step is drawn as a separate node of the computational graph. What we call unfolding is the operation that maps a circuit to a computational graph with repeated pieces. The unfolded graph now has a size that depends on the sequence length.



Unfolding computational graphs

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

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

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

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

Unfolding computational graphs

The unfolding process thus has two major advantages:

1. Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another, rather than specified in terms of a variable-length history of states.
2. It is possible to use the same transition function f with the same parameters at every time step.

- These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths.

- Learning a single shared model allows generalization to sequence lengths that did not appear in the training set, and enables the model to be estimated with far fewer training examples.

Unfolding computational graphs

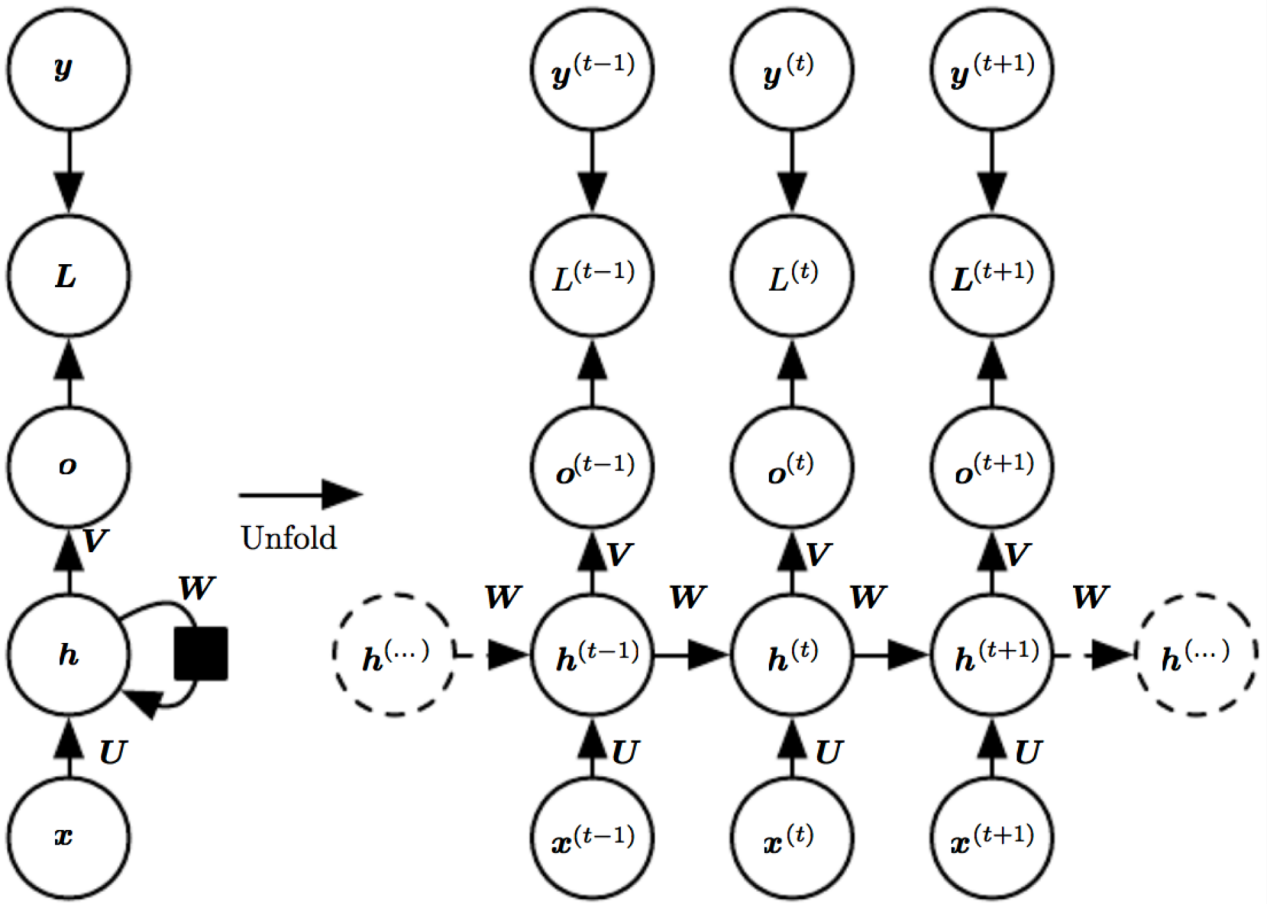
Both the recurrent graph and the unrolled graph have their uses.

- The recurrent graph is succinct.
- The unfolded graph provides an explicit description of which computations to perform. The unfolded graph also helps illustrate the idea of information flow forward in time and backward in time by explicitly showing the path along which this information flows.

- Some examples of important design patterns for recurrent neural networks are shown in the following three slides.

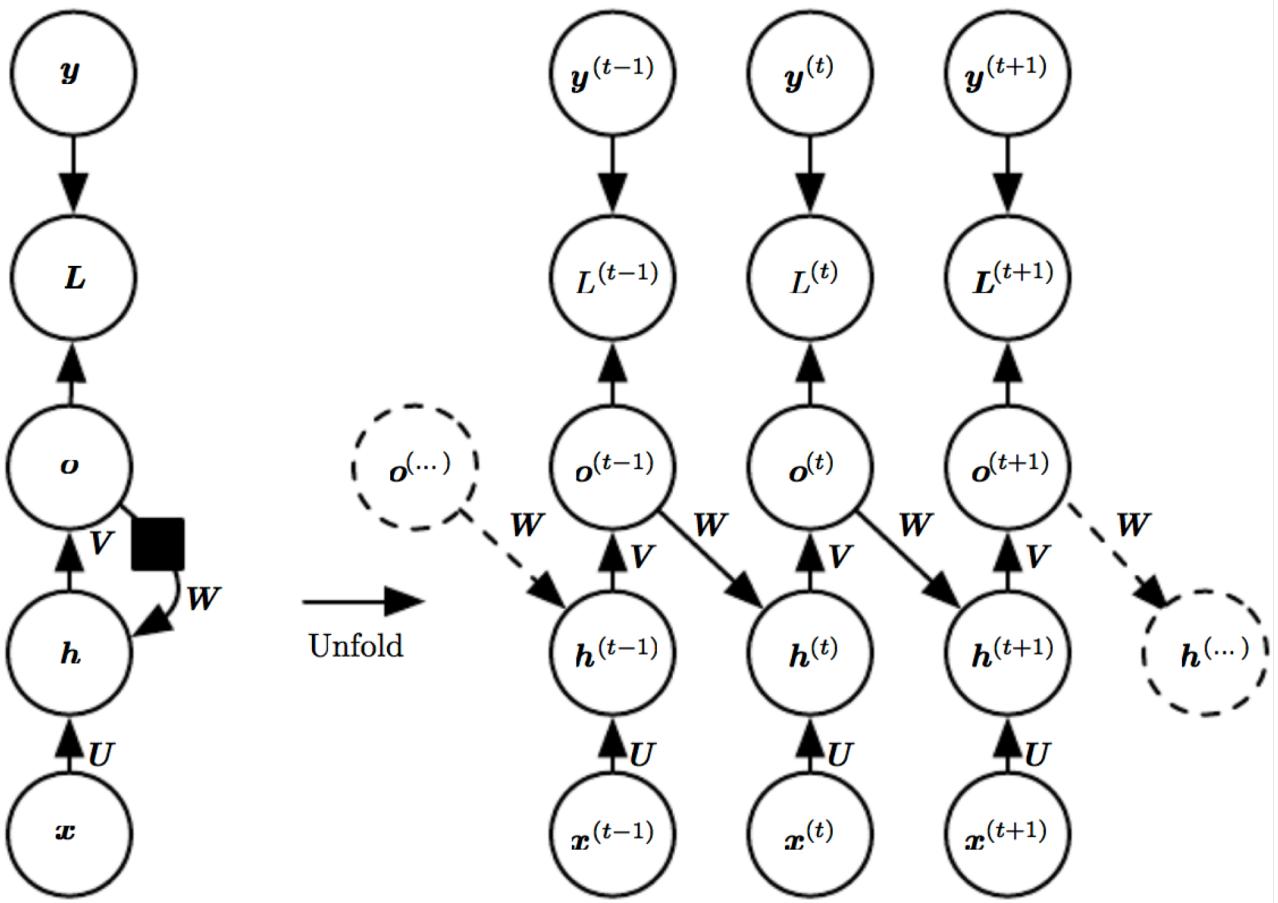
Recurrent neural networks

(Fig. 10.3) Recurrent networks that produce an output at each time step and have recurrent connections between hidden units.



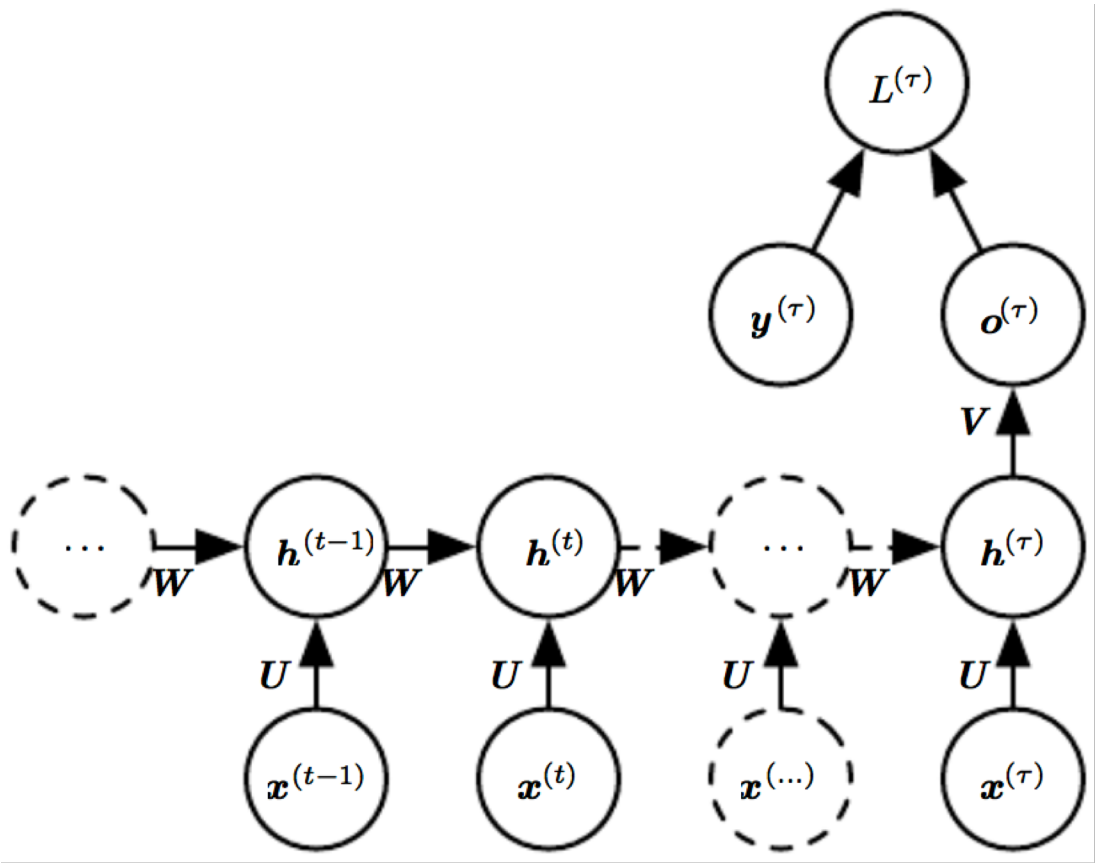
Recurrent neural networks

(Fig. 10.4) Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step.



Recurrent neural networks

(Fig. 10.5) Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output.



Recurrent neural networks

- We now develop the forward propagation equations for the RNN depicted in figure 10.3.
- We assume the hyperbolic tangent activation function.
- We assume that the output is discrete, as if the RNN is used to predict words. A natural way to represent discrete variables is to regard the output o as giving the unnormalized log probabilities of each possible value of the discrete variable. We can then apply the $\mathrm{softmax}$ operation as a post-processing step to obtain a vector $\hat{y}$ of normalized probabilities over the output.

Recurrent neural networks

- Forward propagation begins with a specification of the initial state $\boldsymbol{h}^{(0)}$. Then, for each time step from $t = 1$ to $t = \tau$, we apply the following update equations:

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

where the parameters are the bias vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ along with the weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$, respectively, for input-to-hidden, hidden-to-output and hidden-to-hidden connections.

Recurrent neural networks

- The total loss for a given sequence of x values paired with a sequence of y values would then be the sum of the losses over all the time steps.

- For example, if $L^{(t)}$ is the negative log-likelihood of $y^{(t)}$ given $\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(t)}$, then

$$L\left(\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(\tau)}\},\{\boldsymbol{y}^{(1)},\ldots,\boldsymbol{y}^{(\tau)}\}\right) \tag{10.12}$$

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

$$=-\sum_t \log p_{\text{model}}\left(y^{(t)} \mid \{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(t)}\}\right), \tag{10.14}$$

where $p_{\text{model}}\left(y^{(t)}|\{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(t)}\}\right)$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{\boldsymbol{y}}^{(t)}$.

Recurrent neural networks

- Computing the gradient of this loss function with respect to the parameters is an expensive operation. The gradient computation involves performing a forward propagation, followed by a backward propagation.

- The runtime is $O(\tau)$ and cannot be reduced by parallelization because the forward propagation graph is inherently sequential; each time step may be computed only after the previous one.

- States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also $O(\tau)$. The back-propagation algorithm applied to the unrolled graph with $O(\tau)$ cost is called back-propagation through time (BPTT).

Teacher forcing and networks with output recurrence

- The network with recurrence between hidden units is thus very powerful but also expensive to train. Is there an alternative?
- We may eliminate hidden-to-hidden recurrence.
- The network with recurrent connections only from the output at one time step to the hidden units at the next time step (shown in figure 10.4) is strictly less powerful because it lacks hidden-to-hidden recurrent connections.
- Because this network lacks hidden-to-hidden recurrence, it requires that the output units capture all the information about the past that the network will use to predict the future.

Teacher forcing and networks with output recurrence

- Because the output units are explicitly trained to match the training set targets, they are unlikely to capture the necessary information about the past history of the input, unless the user knows how to describe the full state of the system and provides it as part of the training set targets.

- The advantage of eliminating hidden-to-hidden recurrence is that, for any loss function based on comparing the prediction at time t to the training target at time t, all the time steps are decoupled. Training can thus be parallelized, with the gradient for each step t computed in isolation.

Teacher forcing and networks with output recurrence

- Models that have recurrent connections from their outputs leading back into the model may be trained with teacher forcing. Teacher forcing is a procedure that during training the model receives the ground truth output $y^{(t)}$ as input at time $t + 1$.

- We can see this by examining a sequence with two time steps. The conditional maximum likelihood criterion is

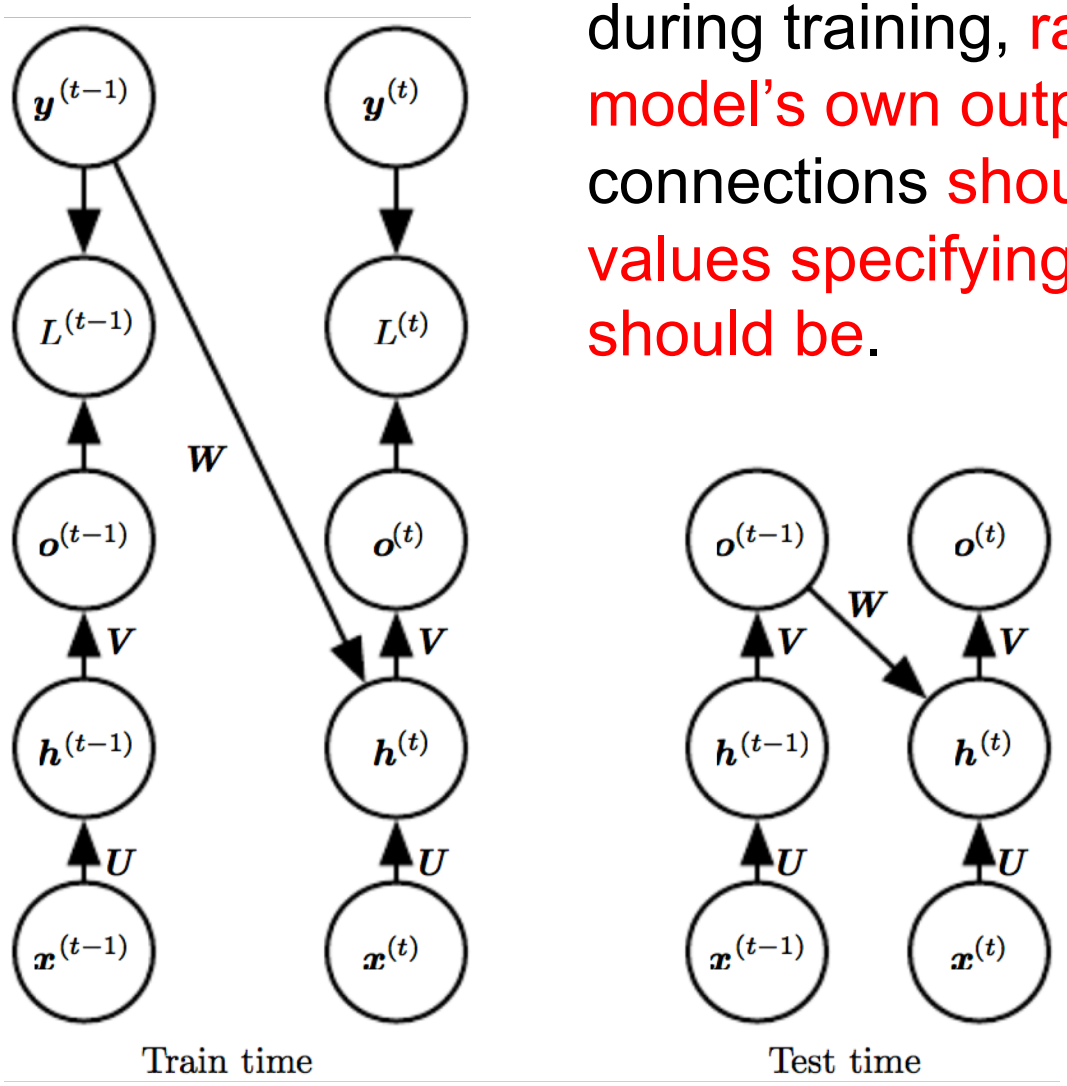
$$\log p\left(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) \tag{10.15}$$

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

In this example, we see that at time $t = 2$, the model is trained to maximize the conditional probability of $\boldsymbol{y}^{(2)}$ given both the $\boldsymbol{x}$ sequence so far and the previous $\boldsymbol{y}$ value from the training set.

Teacher forcing and networks with output recurrence

- Maximum likelihood thus specifies that during training, rather than feeding the model's own output back into itself, these connections should be fed with the target values specifying what the correct output should be.



Teacher forcing and networks with output recurrence

- We originally motivated teacher forcing as allowing us to avoid back-propagation through time in models that lack hidden-to-hidden connections.

- Teacher forcing may still be applied to models that have hidden-to-hidden connections as long as they have connections from the output at one time step to values computed in the next time step.

- However, as soon as the hidden units become a function of earlier time steps, the BPTT algorithm is necessary. Some models may thus be trained with both teacher forcing and BPTT.

Computing the gradient in a RNN

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

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

Computing the gradient in a RNN

- To gain some intuition for how the BPTT algorithm behaves, we provide an example of how to compute gradients by BPTT for the equations 10.8 and 10.12.

- The nodes of our computational graph include the parameters $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ as well as the sequence of nodes indexed by t for $\boldsymbol{x}^{(t)}$, $\boldsymbol{h}^{(t)}$, $\boldsymbol{o}^{(t)}$ and $L^{(t)}$. For each node $\mathbf{N}$ we need to compute the gradient $\nabla_{\mathbf{N}} L$ recursively, based on the gradient computed at nodes that follow it in the graph.

- We start the recursion with the nodes immediately preceding the final loss:

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

Computing the gradient in a RNN

- In this derivation we assume that the outputs $\boldsymbol{o}^{(t)}$ are used as the argument to the $\mathrm{softmax}$ function to obtain the vector $\hat{\boldsymbol{y}}$ of probabilities over the output. We also assume that the loss is the negative log-likelihood of the true target $y^{(t)}$ given the input so far.

- The gradient $\nabla_{\boldsymbol{o}^{(t)}} L$ on the outputs at time step t is

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

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

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

Computing the gradient in a RNN

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

- Its gradient is thus given by

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

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

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

Computing the gradient in a RNN

- Because the parameters are shared across many time steps, we must take some care when denoting calculus operations involving these variables.

- The equations we wish to implement use the bprop method of section 6.5.6, which computes the contribution of a single edge in the computational graph to the gradient. However, the $\nabla_{\boldsymbol{W}} f$ operator used in calculus takes into account the contribution of $\boldsymbol{W}$ to the value of f due to all edges in the computational graph.

- To resolve this, we introduce dummy variables $\boldsymbol{W}^{(t)}$ that are defined to be copies of $\boldsymbol{W}$ but with each $\boldsymbol{W}^{(t)}$ used only at time step t. We may then use $\nabla_{\boldsymbol{W}^{(t)}}$ to denote the contribution of the weights at time step t to the gradient.

Computing the gradient in a RNN

- The gradient on the remaining parameters is given by

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

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

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

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

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

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

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

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

Bidirectional RNNs

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

- In many applications, however, we want to output a prediction of $\boldsymbol{y}^{(t)}$ that may depend on the whole input sequence.

- For example, in speech recognition, the correct interpretation of the current sound as a phoneme may depend on the next few phonemes because of coarticulation and may even depend on the next few words because of the linguistic dependencies between nearby words.

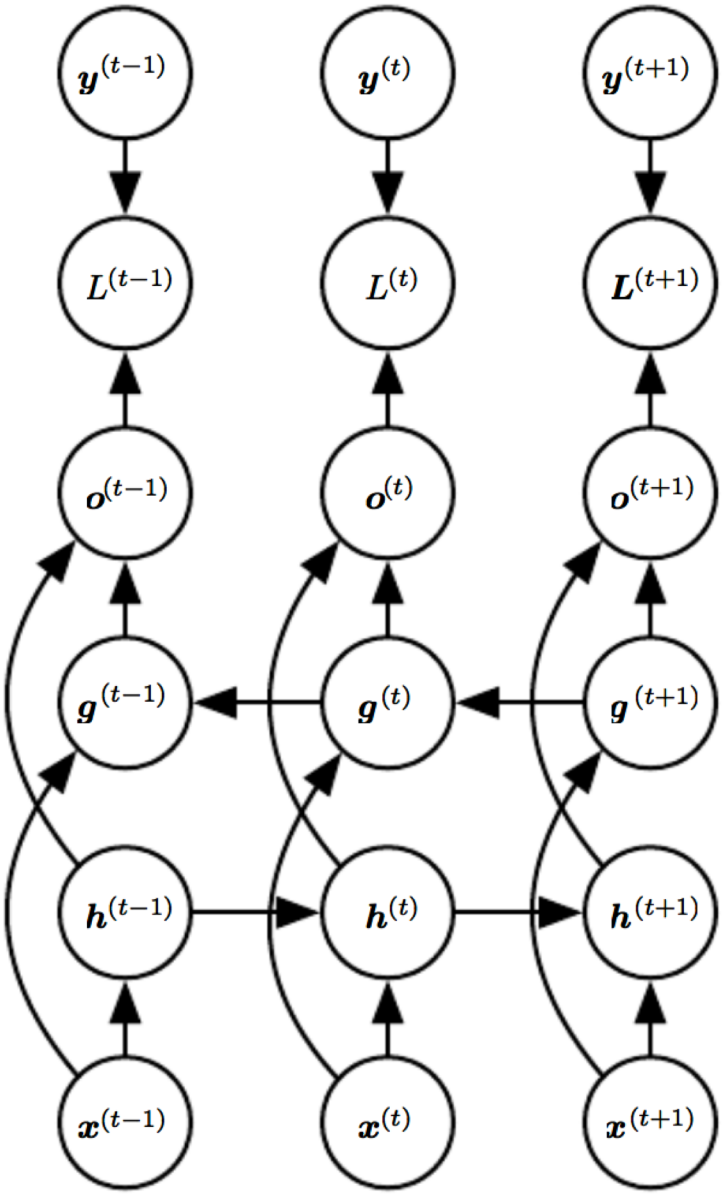
Bidirectional RNNs

- Bidirectional recurrent neural networks (or bidirectional RNNs) were invented to address that need.

- They have been extremely successful in applications where that need arises, such as handwriting recognition, speech recognition, and bioinformatics.

- As the name suggests, bidirectional RNNs combine an RNN that moves forward through time, beginning from the start of the sequence, with another RNN that moves backward through time, beginning from the end of the sequence.

Bidirectional RNNs



- $\boldsymbol{h}^{(t)}$ stands for the state of the sub-RNN that moves forward through time and $\boldsymbol{g}^{(t)}$ stands for the state of the sub-RNN that moves backward through time.
- This allows the output units $\boldsymbol{o}^{(t)}$ to compute a representation that depends on both the past and the future but is most sensitive to the input values around time t, without having to specify a fixed-size window around t.

Bidirectional RNNs

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

- Compared to a convolutional network, RNNs applied to images are typically more expensive but allow for long-range lateral interactions between features in the same feature map.

Deep recurrent networks

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

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

- With the RNN architecture of figure 10.3, each of these three blocks is associated with a single weight matrix. In other words, when the network is unfolded, each of these blocks corresponds to a shallow transformation (represented by a single layer within a deep MLP).

- Experiments strongly suggests that it is advantageous to introduce depth in each of these operations.

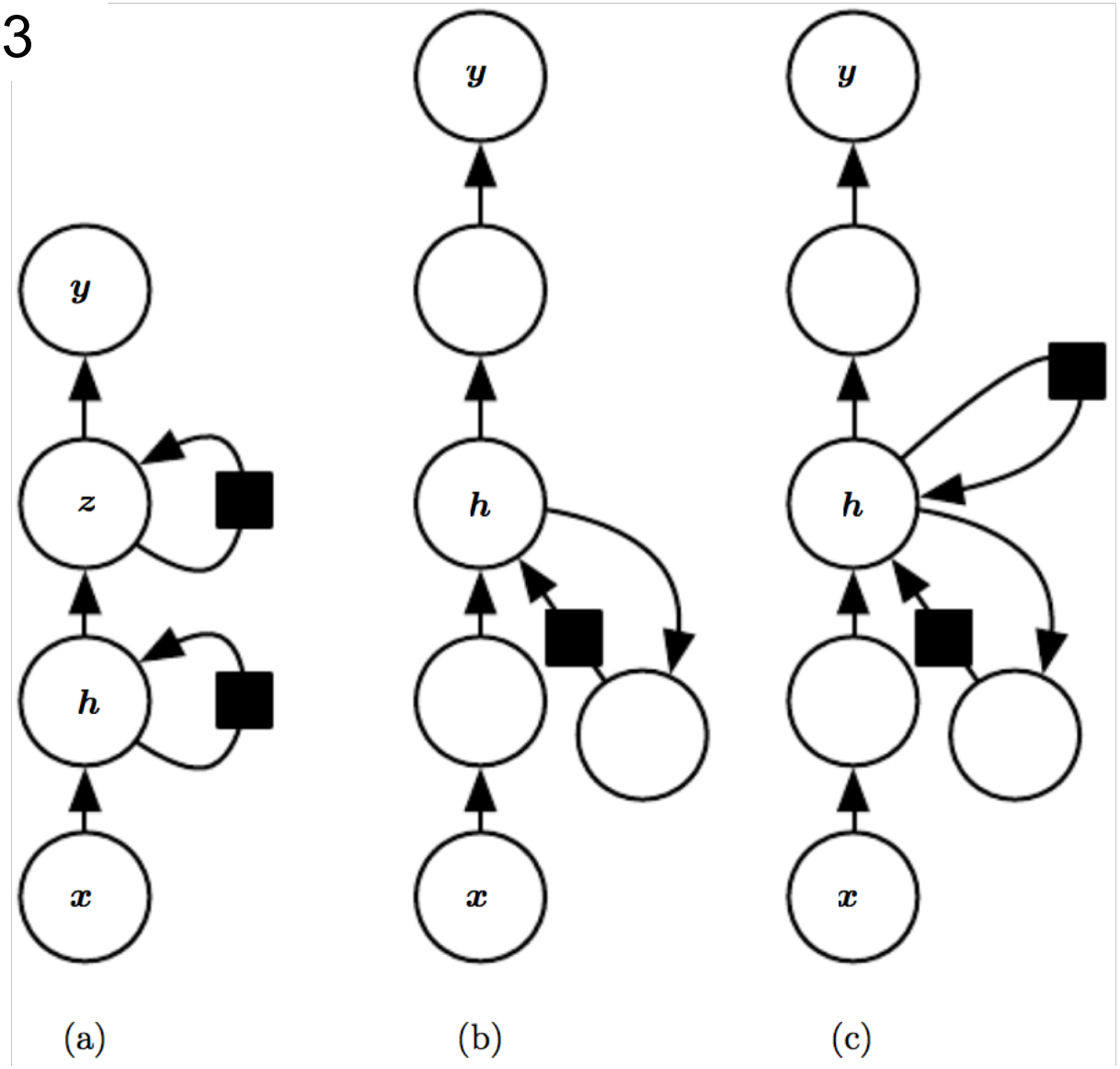
Deep recurrent networks

- There is a significant benefit of decomposing the state of an RNN into multiple layers, as in figure 10.13a. We can think of the lower layers in the 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.

- A further step can be taken by having a separate MLP (possibly deep) for each of the three blocks enumerated above, as illustrated in figure 10.13b. Considerations of representational capacity suggest allocating enough capacity in each of these three steps.

Deep recurrent networks

Fig. 10.13



Deep recurrent networks

- Adding depth may hurt learning by making optimization difficult. In general, it is easier to optimize shallower architectures, and adding the extra depth of figure 10.13b makes the shortest path from a variable in time step t to a variable in time step $t + 1$ become longer.

- For example, if an MLP with a single hidden layer is used for the state-to-state transition, we have doubled the length of the shortest path between variables in any two different time steps, compared with the ordinary RNN of figure 10.3.

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

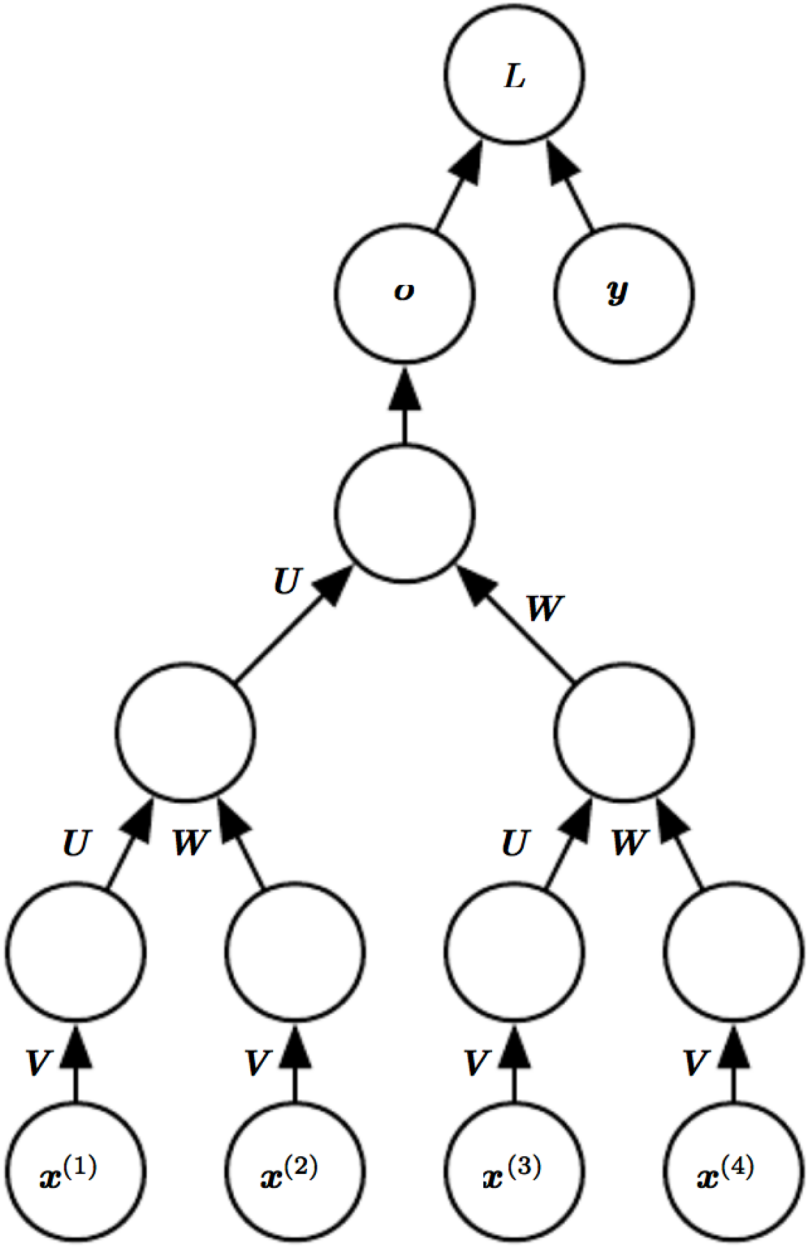
Recursive neural networks

- Recursive neural 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.

- Recursive networks have been successfully applied to processing data structures as input to neural nets, in natural language processing, as well as in computer vision.

- One clear advantage of recursive nets over recurrent nets is that for a sequence of the same length τ, the depth can be drastically reduced from τ to $O(\log \tau)$, which might help deal with long-term dependencies.

Recursive neural networks



Recursive neural networks

- An open question is how to best structure the tree.
- One option is to have a tree structure that does not depend on the data, such as a balanced binary tree.
- In some application domains, external methods can suggest the appropriate tree structure. For example, when processing natural language sentences, the tree structure for the recursive network can be fixed to the structure of the parse tree of the sentence provided by a natural language parser.
- Ideally, one would like the learner itself to discover and infer the tree structure that is appropriate for any given input.

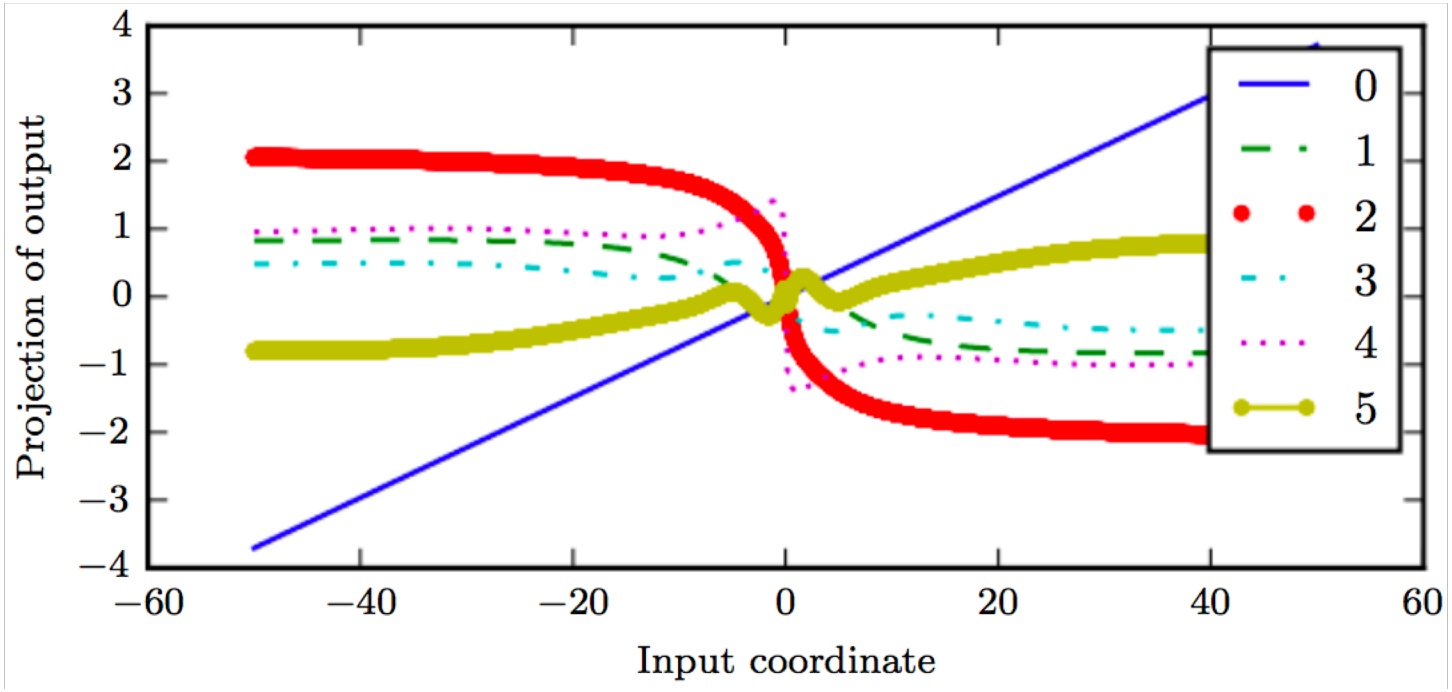
The challenge of long-term dependencies

- The basic problem of learning long-term dependencies in recurrent networks is that gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization).

- Even if we assume that the parameters are such that the recurrent network is stable (can store memories, with gradients not exploding), the difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions (involving the multiplication of many Jacobians) compared to short-term ones.

The challenge of long-term dependencies

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



The challenge of long-term dependencies

- The function composition employed by RNNs somewhat resembles matrix multiplication. The recurrence relation

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

can be thought of as a very simple recurrent neural network lacking a nonlinear activation function, and lacking inputs $\boldsymbol{x}$. It may be simplified to

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

- If $\boldsymbol{W}$ admits an eigendecomposition of the form

$$\boldsymbol{W} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\top}, \tag{10.38}$$

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

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

The eigenvalues are raised to the power of t, causing eigenvalues with magnitude less than one to decay to zero and with magnitude greater than one to explode.

The challenge of long-term dependencies

- Imagine multiplying a weight ω by itself many times. The product ω^t will either vanish or explode depending on the magnitude of ω.

- If we make a nonrecurrent network that has a different weight $\omega^{(t)}$ at each time step, the situation is different. If the initial state is given by 1, then the state at time t is given by $\prod_t \omega^{(t)}$. Suppose that the $\omega^{(t)}$ values are generated randomly, independently from one another, with zero mean and variance v. The variance of the product is $O(v^n)$. To obtain some desired variance v^* we may choose the individual weights with variance $v = \sqrt[n]{v^*}$. Very deep feedforward networks with carefully chosen scaling can thus avoid the vanishing and exploding gradient problem.

The challenge of long-term dependencies

- Whenever the model is able to represent long-term dependencies, the gradient of a long-term interaction has exponentially smaller magnitude than the gradient of a short-term interaction. This means not that it is impossible to learn, but that it might take a very long time to learn long-term dependencies, because the signal about these dependencies will tend to be hidden by the smallest fluctuations arising from short-term dependencies.

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

Echo state networks

- One proposed approach to avoiding the difficulty of learning is to set the recurrent weights such that the recurrent hidden units do a good job of capturing the history of past inputs, and only learn the output weights.

- This is the idea that was independently proposed for echo state networks, or ESNs, and liquid state machines.

- ESNs use continuous-valued hidden units while liquid state machines use spiking neurons (with binary outputs). Both ESNs and liquid state machines are termed reservoir computing.

Echo state networks

- ESNs are similar to kernel machines: they map an arbitrary length sequence (the history of inputs up to time t) into a fixed-length vector (the recurrent state $\boldsymbol{h}^{(t)}$), on which a linear predictor (typically a linear regression) can be applied to solve the problem of interest. The training criterion may then be easily designed to be convex as a function of the output weights.

- 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 is to view the recurrent net as a dynamical system, and set the input and recurrent weights such that the dynamical system is near the edge of stability.

Echo state networks

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

- To understand the effect of the spectral radius, consider the simple case of back-propagation with a Jacobian matrix $\boldsymbol{J}$ that does not change with t. This case happens, for example, when the network is purely linear. Suppose that $\boldsymbol{J}$ has an eigenvector $\boldsymbol{v}$ with corresponding eigenvalue λ.

Echo state networks

- Consider what happens as we propagate a gradient vector backward through time. If we begin with a gradient vector $\boldsymbol{g}$, then after one step of back-propagation, we will have $\boldsymbol{J}\boldsymbol{g}$, and after n steps we will have $\boldsymbol{J}^n\boldsymbol{g}$.

- Now consider what happens if we instead back-propagate a perturbed version of $\boldsymbol{g}$. If we begin with $\boldsymbol{g} + \delta\boldsymbol{v}$, then after one step, we will have $\boldsymbol{J}(\boldsymbol{g} + \delta\boldsymbol{v})$. After n steps, we will have $\boldsymbol{J}^n(\boldsymbol{g} + \delta\boldsymbol{v})$.

- Back-propagation starting from $\boldsymbol{g}$ and back-propagation starting from $\boldsymbol{g} + \delta\boldsymbol{v}$ diverge by $\delta\boldsymbol{J}^n\boldsymbol{v}$ after n steps of back-propagation. If $\boldsymbol{v}$ is chosen to be a unit eigenvector of $\boldsymbol{J}$ with eigenvalue λ, then the two executions of back-propagation are separated by a distance of $\delta|\lambda|^n$.

Echo state networks

- When $\boldsymbol{v}$ corresponds to the largest value of $|\lambda|$, this perturbation achieves the widest possible separation of an initial perturbation of size δ.

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

- This example assumed that the Jacobian was the same at every time step, corresponding to a recurrent network with no nonlinearity. When a nonlinearity is present, the derivative of the nonlinearity will approach zero on many time steps and help prevent the explosion resulting from a large spectral radius.

Echo state networks

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

- The Jacobian matrix tells us how a small change of $\boldsymbol{h}^{(t)}$ propagates one step forward, or equivalently, how the gradient on $\boldsymbol{h}^{(t+1)}$ propagates one step backward, during back-propagation.

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

Leaky units and other strategies for multiple time scales

- One way to deal with long-term dependencies is to design a model that operates at multiple time scales, so that some parts of the model operate at fine-grained time scales and can handle small details, while other parts operate at coarse time scales and transfer information from the distant past to the present more efficiently.

- Various strategies for building both fine and coarse time scales are possible. These include
  - the addition of skip connections across time,
  - "leaky units" that integrate signals with different time constants, and
  - the removal of some of the connections used to model fine-grained time scales.

Adding skip connections through time

- One way to obtain coarse time scales is to add direct connections from variables in the distant past to variables in the present.
- For recurrent connections with a time delay of d, gradients now diminish exponentially as a function of $\frac{\tau}{d}$ rather than τ.
- Since there are both delayed and single step connections, gradients may still explode exponentially in τ.

Leaky units and a spectrum of different time scales

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

- When we accumulate a running average $\mu^{(t)}$ of some value $v^{(t)}$ by applying the update $\mu^{(t)} \leftarrow \alpha\mu^{(t-1)} + (1 - \alpha)v^{(t)}$, the α parameter is an example of a linear self-connection from $\mu^{(t-1)}$ to $\mu^{(t)}$. When α is near one, the running average remembers information about the past for a long time, and when α is near zero, information about the past is rapidly discarded.

- Hidden units with linear self-connections can behave similarly to such running averages. Such hidden units are called leaky units.

Leaky units and a spectrum of different time scales

- The use of a linear self-connection with a weight near one is a different way of ensuring that the unit can access values from the past. The linear self-connection approach allows this effect to be adapted more smoothly and flexibly by adjusting the real valued α rather than by adjusting the integer-valued skip length.

Removing connections

- Another approach to handling long-term dependencies is the idea of organizing the state of the RNN at multiple time scales. This idea involves actively removing length-one connections and replacing them with longer connections. Units receiving such new connections may learn to operate on a long time scale but may also choose to focus on their other, short-term connections.

- There are different ways in which a group of recurrent units can be forced to operate at different time scales. One option is to make the recurrent units leaky, but to have different groups of units associated with different fixed time scales. Another option is to have explicit and discrete updates taking place at different times, with a different frequency for different groups of units.

The long short-term memory and other gated RNNs

- The most effective sequence models used in practical applications are called gated RNNs. These include the long short-term memory (LSTM) and networks based on the gated recurrent unit (GRU).

- Like leaky units, gated RNNs are based on the idea of creating paths through time that have derivatives that neither vanish nor explode.

- Leaky units did this with connection weights that were either manually chosen constants or were parameters.

- Gated RNNs generalize this to connection weights that may change at each time step.

The long short-term memory and other gated RNNs

- Leaky units allow the network to accumulate information (such as evidence for a particular feature or category) over a long duration. However, once that information has been used, it might be useful for the neural network to forget the old state.

- For example, if a sequence is made of subsequences and we want a leaky unit to accumulate evidence inside each sub-subsequence, we need a mechanism to forget the old state by setting it to zero.

- Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it. This is what gated RNNs do.

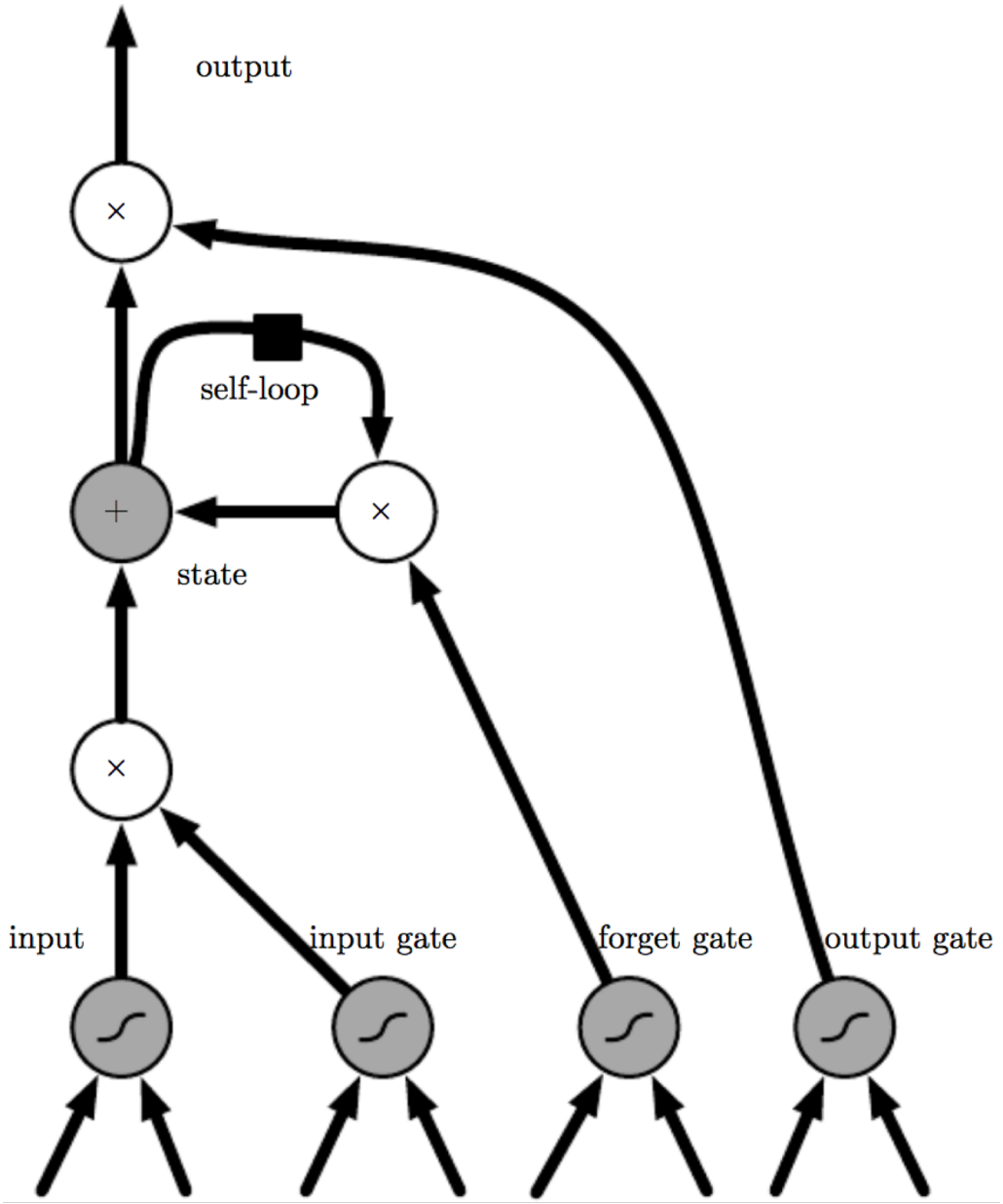
LSTM

- The LSTM model makes the weight on this self-loop conditioned on the context, rather than fixed. By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically based on the input sequence, because the time constants are output by the model.

- LSTM recurrent networks have "LSTM cells" that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN.

- Each cell has the same inputs and outputs as an ordinary recurrent network, but also has more parameters and a system of gating units that controls the flow of information.

LSTM


output
self-loop
state
input
input gate
forget gate
output gate

LSTM

- The most important component is the state unit $s_i^{(t)}$, which has a linear self-loop similar to the leaky i units.

- The self-loop weight (or the associated time constant) is controlled by a forget gate unit $f_i^{(t)}$ (for time step t and cell i), which sets this weight to a value between 0 and 1 via a sigmoid unit:

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

where $\boldsymbol{x}^{(t)}$ is the current input vector and $\boldsymbol{h}^{(t)}$ is the current hidden layer vector, containing the outputs of all the LSTM cells, and $\boldsymbol{b}^f$, $\boldsymbol{U}^f$, $\boldsymbol{W}^f$ are respectively biases, input weights, and recurrent weights for the forget gates.

LSTM

- The LSTM cell internal state is thus updated as follows, but with a conditional self-loop weight $f_i^{(t)}$:

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

where $\boldsymbol{b}$, $\boldsymbol{U}$ and $\boldsymbol{W}$ respectively denote the biases, input weights, and recurrent weights into the LSTM cell.

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

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

LSTM

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

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

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

  which has parameters $\boldsymbol{b}^o$, $\boldsymbol{U}^o$, $\boldsymbol{W}^o$ for its biases, input weights and recurrent weights, respectively.

- One can choose to use the cell state $s_i^{(t)}$ as an extra input (with its weight) into the three gates of the i-th unit. This would require three additional parameters.

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?

- In gated recurrent units, or GRUs, a single gating unit simultaneously controls the forgetting factor and the decision to update the state unit. The update equations are the following:

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

where $\boldsymbol{u}$ stands for "update" gate and $\boldsymbol{r}$ for "reset" gate.

Other gated RNNs

- The values of update gate and reset gate are defined as

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

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

- The reset and update gates can individually "ignore" parts of the state vector. The update gates act like conditional leaky integrators that can linearly gate any dimension, thus choosing to copy it or completely ignore it by replacing it with the new "target state" value.

- The reset gates control which parts of the state get used to compute the next target state, resulting in additional nonlinear effect between past state and future state.

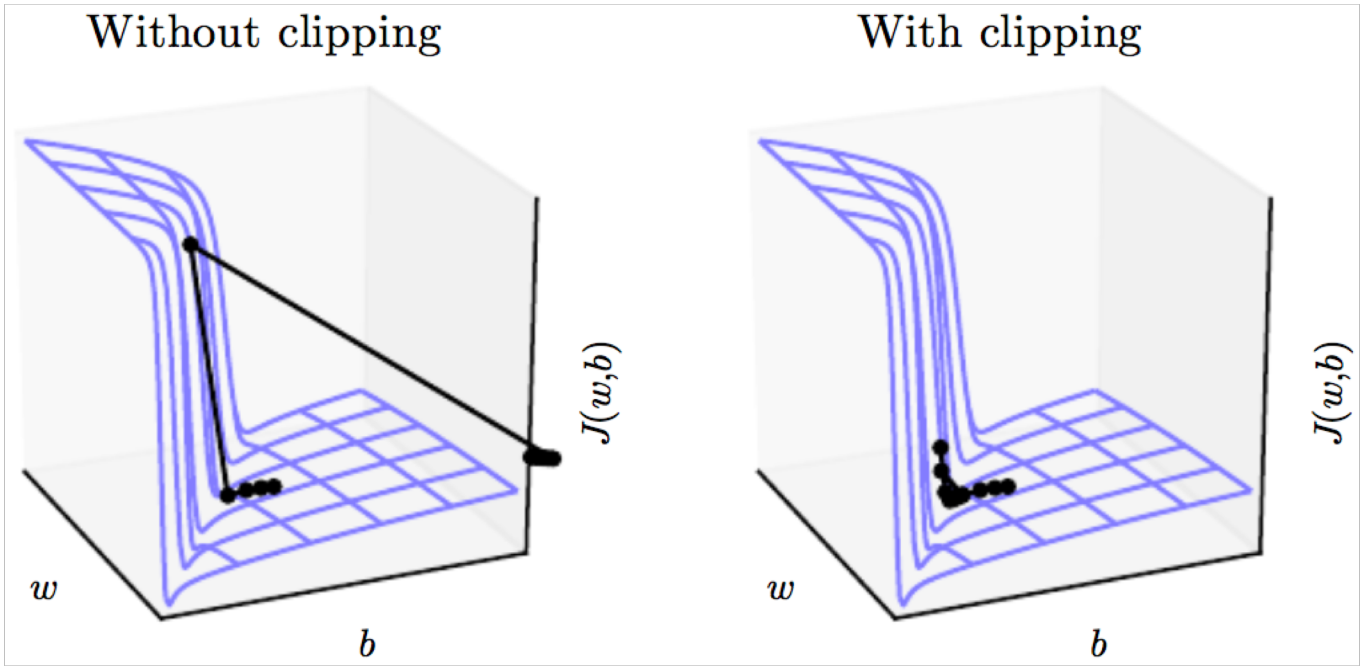
Other gated RNNs

- Many more variants around this theme can be designed. For example the reset gate (or forget gate) output could be shared across multiple hidden units. Alternately, the product of a global gate (covering a whole group of units, such as an entire layer) and a local gate (per unit) could be used to combine global control and local control.

- Several investigations over architectural variations of the LSTM and GRU, however, found no variant that would clearly beat both of these across a wide range of tasks.

- A crucial ingredient is the forget gate, while it was found that adding a bias of 1 to the LSTM forget gate makes the LSTM as strong as the best of the explored architectural variants.

Clipping gradients



Clipping gradients

- 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.

- A simple type of solution has been in use by practitioners for many years: clipping the gradient. One option is to clip the parameter gradient from a minibatch element-wise, just before the parameter update. Another is to clip the norm $||\boldsymbol{g}||$ just before the parameter update:

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

$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}v}{||\boldsymbol{g}||}, \tag{10.49}$$

where v is the norm threshold and $\boldsymbol{g}$ is used to update parameters.

Clipping gradients

- Because the gradient of all the parameters (including different groups of parameters, such as weights and biases) is renormalized jointly with a single scaling factor, clipping the norm has the advantage of guaranteeing that each step is still in the gradient direction, but experiments suggest that both forms work similarly.

- In fact, even simply taking a random step when the gradient magnitude is above a threshold tends to work almost as well. If the explosion is so severe that the gradient is numerically Inf or Nan (considered infinite or not-a-number), then a random step of size v can be taken and will typically move away from the numerically unstable configuration.

Regularizing to encourage information flow

- Gradient clipping helps to deal with exploding gradients, but it does not help with vanishing gradients.

- To address vanishing gradients and better capture long-term dependencies, we discussed the idea of creating paths in the computational graph of the unfolded recurrent architecture along which the product of gradients associated with arcs is near 1. One approach to achieve this is with LSTMs and other self-loops and gating mechanisms.

Regularizing to encourage information flow

- Another idea is to regularize or constrain the parameters so as to encourage "information flow." We would like the gradient vector $\nabla_{\boldsymbol{h}^{(t)}} L$ being back-propagated to maintain its magnitude, even if the loss function only penalizes the output at the end of the sequence.

- Formally, we want

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

to be as large as

$$\nabla_{\boldsymbol{h}^{(t)}} L. \tag{10.51}$$

- With this objective, the regularizer was proposed

$$\Omega = \sum_t \left(\frac{\left| \left| (\nabla_{\boldsymbol{h}^{(t)}} L) \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \right| \right|}{||\nabla_{\boldsymbol{h}^{(t)}} L||} - 1 \right)^2. \tag{10.52}$$

Regularizing to encourage information flow

- Computing the gradient of this regularizer may appear difficult, but it was proposed an approximation in which we consider the back-propagated vectors $\nabla_{\boldsymbol{h}^{(t)}} L$ as if they were constants.

- The experiments with this regularizer suggest that, if combined with the norm clipping heuristic, the regularizer can considerably increase the span of the dependencies that an RNN can learn. Because it keeps the RNN dynamics on the edge of explosive gradients, the gradient clipping is particularly important.

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

Explicit memory

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

- There are different kinds of knowledge. Some knowledge can be implicit, subconscious, and difficult to verbalize, such as how a dog looks different from a cat.

- Other knowledge can be explicit, declarative, and relatively straightforward to put into words—everyday commonsense knowledge, like "a cat is a kind of animal," or very specific facts that you need to know to accomplish your current goals, like "the meeting with the sales team is at 3:00 PM in room 141."

Explicit memory

- Neural networks excel at storing implicit knowledge, but they struggle to memorize facts.

- It was hypothesized that this is because neural networks lack the equivalent of the working memory system that enables human beings to explicitly hold and manipulate pieces of information that are relevant to achieving some goal.

- Such explicit memory components would allow our systems not only to rapidly and "intentionally" store and retrieve specific facts but also to sequentially reason with them.

Explicit memory

- Memory networks include a set of memory cells that can be accessed via an addressing mechanism.

- Each memory cell can be thought of as an extension of the memory cells in LSTMs and GRUs. The difference is that the network outputs an internal state that chooses which cell to read from or write to, just as memory accesses in a digital computer read from or write to a specific address.

Explicit memory

- The explicit memory approach is illustrated in figure 10.18, where we see that a “task neural network” is coupled with a memory. Although that task neural network could be feedforward or recurrent, the overall system is a recurrent network.

- The task network can choose to read from or write to specific memory addresses. Explicit memory seems to allow models to learn tasks that ordinary RNNs or LSTM RNNs cannot learn. One reason for this advantage may be that information and gradients can be propagated for very long durations.

Explicit memory