SiTE

AAiT

AAU

Course Title: Deep Learning

Credit Hour: 3

Instructor: Fantahun B. (PhD) ✉ meetfantaai@gmail.com

Office: NB #

Ch-4 Sequence Modeling: RNN

Aug-2023, AA

Sequence Modeling: RNN

Agenda:

- Overview
- Parameter sharing
- Unfolding Computational Graphs
- RNNs, Teacher Forcing, Bidirectional RNN, Deep RNN, Recursive RNN, Encoder-Decoder
- Challenges of Long-term dependencies
- LSTM, Variants of LSTM

Sequence Modeling: RNN

Objectives

After completing this chapter students will be able to:

- Explain concepts and algorithms used in sequence modeling like RNNs and LSTMs.
- Implement these algorithms to solve some real world problems

Sequence Modeling: RNN

- Recurrent neural networks or RNNs (Rumelhart et al., 1986a) are a family of neural networks for processing sequential data.
- CNN is specialized for processing a grid of values **X** such as an image,
- RNN is specialized for processing a sequence of values $x^{(1)}, \ldots, x^{(T)}$.
    - CNN can readily scale to images with large width and height, and some convolutional networks can process images of variable size,
    - RNN 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.

Sequence Modeling: RNN
Parameter sharing

- To go from multi-layer networks to recurrent networks, we need to take advantage of one of the early ideas found in machine learning and statistical models of the 1980s: sharing parameters across different parts of a model.
- *Parameter sharing* makes it possible to extend and apply the model to examples of different forms (different lengths, here) and generalize across them.
- If we had separate parameters for each value of the time index, we could not *generalize to sequence lengths not seen during training*, nor share statistical strength across different sequence lengths and across different positions in time.

Sequence Modeling: RNN
Parameter sharing

- Parameter sharing is *particularly important when a specific piece of information can occur at multiple positions within the sequence*.
- For example, consider the two sentences:
  a) "I went to Ethiopia in 2009" and
  b) "In 2009, I went to Ethiopia."
- If we ask a machine learning model to read each sentence and extract the year information, we would like it to *recognize the year 2009 as the relevant piece of information, irrespective of its appearance* in the sixth word or the second word of the sentence.
- A traditional fully connected feedforward network would have separate parameters for each input feature, so it needs to learn all of the rules of the language separately at each position in the sentence. --- *RNN shares the same weights across several time steps*.

Sequence Modeling

- A related idea is the use of 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*.
  - The idea of parameter sharing manifests in the application of the same convolution kernel at each time step.
- RNNs share parameters in a different way.
  - *Each member of the output is a function of the previous members of the output*.
  - Each member of the output is produced using the same update rule applied to the previous outputs.
  - This recurrent formulation results in the sharing of parameters through a very deep computational graph.

Sequence Modeling

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

Sequence Modeling: Unfolding Computational Graphs

- A computational graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss.
- In this section we explain the idea of *unfolding* a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events. Unfolding this graph results in the sharing of parameters across a deep network structure.
- For example, consider the classical form of a dynamical system:

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

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

- Eq. 10.1 is recurrent because the definition of s at time t refers back to the same definition at time t−1.

Sequence Modeling: Unfolding Computational Graphs

- For a finite number of time steps τ , the graph can be unfolded by applying the definition τ-1 times. Unfold Eq. 10.1 for τ = 3 time steps:

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

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

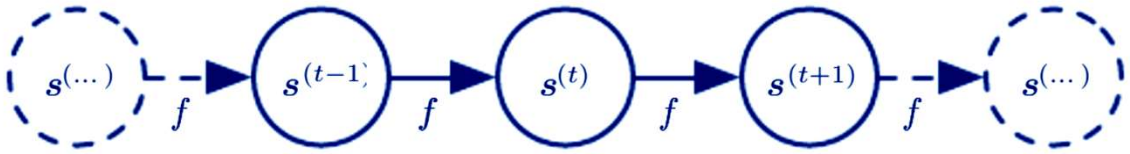


Figure 10.1: The classical dynamical system described by Eq. 10.1, illustrated as an unfolded computational graph. Each node represents the state at some time t and the function f maps the state at t to the state at t + 1. The same parameters (the same value of θ used to parametrize f) are used for all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

Sequence Modeling: Unfolding Computational Graphs

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

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

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

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

Sequence Modeling: Unfolding Computational Graphs

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

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

- Illustrated in Fig. 10.2, typical RNNs will add extra architectural features such as output layers that read information out of the state h to make predictions.
- When the recurrent network is trained to perform a task that requires predicting the future from the past, the network typically learns to use $\mathbf{h}^{(t)}$ as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t.

Sequence Modeling: Unfolding Computational Graphs

- This summary is in general necessarily lossy, since it maps an arbitrary length sequence $(x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(2)}, x^{(1)})$ to a fixed length vector $h^{(t)}$.
- Depending on the training criterion, this summary might selectively keep some aspects of the past sequence with more precision than other aspects.

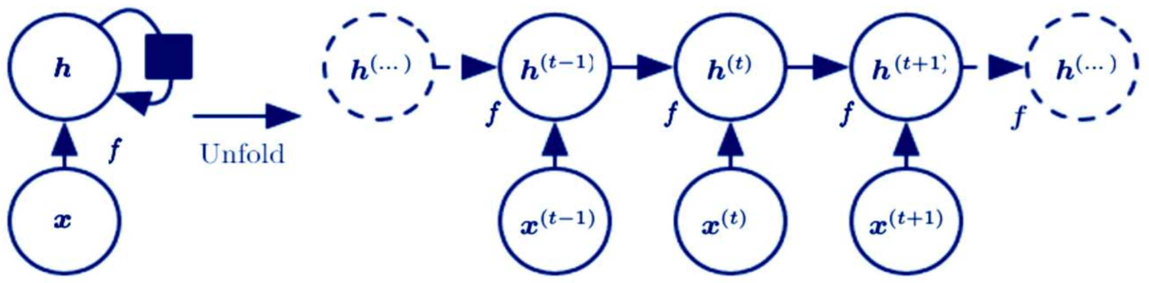


Figure 10.2: RNN with no outputs. This recurrent network just processes information from the input x by incorporating it into the state h that is passed forward through time. ***(Left)*** Circuit diagram. The black square indicates a delay of 1 time step. ***(Right)*** The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance.

Sequence Modeling: Unfolding Computational Graphs

- Eq. 10.5 can be drawn in two different ways.
  1. With a diagram containing one node for every component that might exist in a physical implementation of the model, such as a biological neural network.
     - In this view, the network defines a circuit that operates in real time, with physical parts whose current state can influence their future state, as in the left of Fig. 10.2.
  2. As an unfolded computational graph, in which each component is represented by many different variables, with one variable per time step, representing the state of the component at that point in time.
     - Each variable for each time step is drawn as a separate node of the computational graph, as in the right of Fig. 10.2.
     - Unfolding is the operation that maps a circuit as in (left side) of the figure to a computational graph with repeated pieces as in (right side).
     - The unfolded graph has a size that depends on the sequence length.

Sequence Modeling: Unfolding Computational Graphs

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

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

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

- The function $g^{(t)}$ takes the whole past sequence $(x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(2)}, x^{(1)})$ 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*.

Sequence Modeling: Unfolding Computational Graphs

- The unfolding process thus introduces two major advantages:
  1. Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states.
  2. It is possible to use the same transition function f with the same parameters at every time step.
- *These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths, rather than needing to learn a separate model g(t) for all possible time steps.*
- Learning a single, shared model allows generalization to sequence lengths that did not appear in the training set, and allows the model to be estimated with far fewer training examples than would be required without parameter sharing.

Sequence Modeling: Unfolding Computational Graphs

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

Sequence Modeling: RNNs Design Types

- Armed with the graph unrolling and parameter sharing ideas from the previous section, we can design a wide variety of recurrent neural networks.
- Some examples of important design patterns for recurrent neural networks include the following:
  1) RNNs that produce an output at each time step and have recurrent connections between *hidden units*, illustrated in Fig. 10.3.
  2) RNNs that produce an output at each time step and have recurrent connections only from the *output at one time step to the hidden units at the next time step*, illustrated in Fig. 10.4
  3) RNNs with recurrent connections between hidden units, that read an entire sequence and then produce a single output, illustrated in Fig. 10.5.
- Fig. 10.3 is a reasonably representative example that we return to throughout most of the chapter.

Sequence Modeling: RNNs Design Types

Type-1

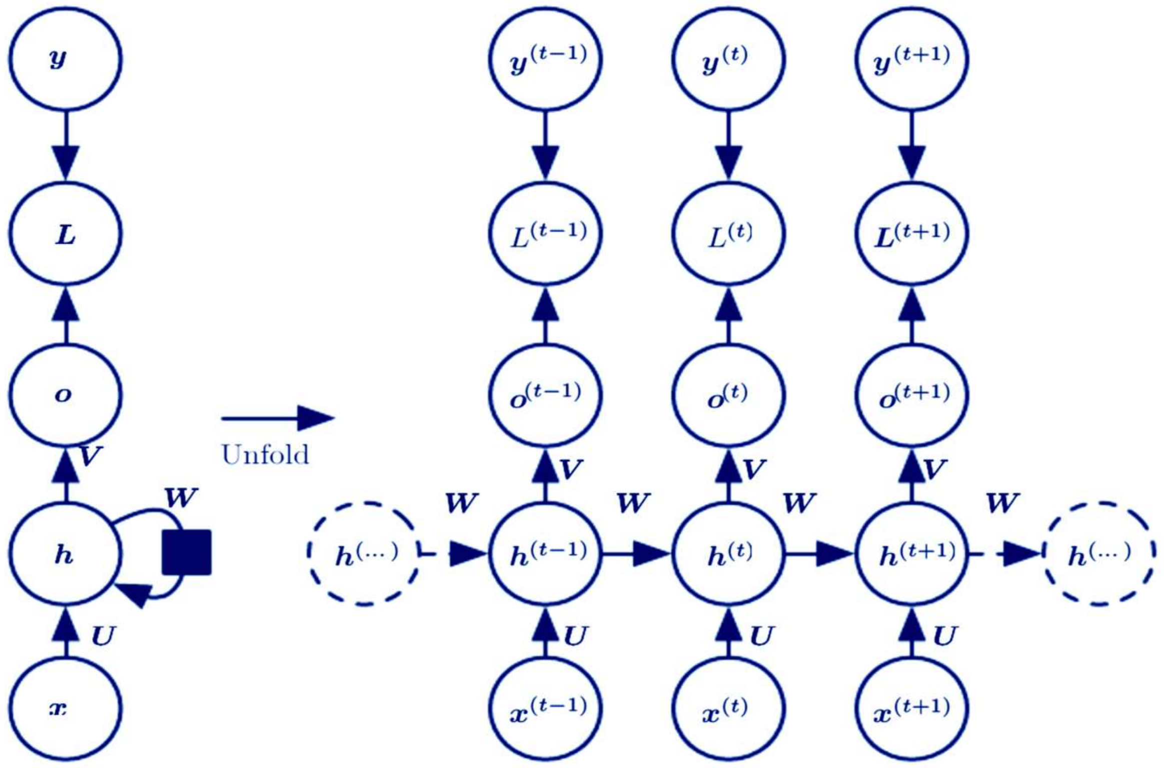


Figure10.3 : The computational graph to compute the training loss of a recurrent network that maps an input sequence of x values to a corresponding sequence of output o values. (Left) The RNN and its loss drawn with recurrent connections. (Right) The same seen as an time-unfolded computational graph, where each node is now associated with one particular time instance.

Sequence Modeling: RNNs Design Types

- The previous figure shows the computational graph to compute the training loss of a recurrent network that maps an input sequence of **x** values to a corresponding sequence of output **o** values.
- A loss **L** measures how far each **o** is from the corresponding training target **y** .
- When using softmax outputs, we assume **o** is the unnormalized log probabilities. The loss **L** internally computes $\hat{y}$ = softmax(**o**) and compares this to the target y.
- The RNN has
  - input-to-hidden connections parametrized by a weight matrix **U**,
  - hidden-to-hidden recurrent connections parametrized by a weight matrix **W**, and
  - hidden-to-output connections parametrized by a weight matrix **V**.
- Eq. 10.8 defines forward propagation in this model.

Sequence Modeling: RNNs Design Types

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

Sequence Modeling: RNNs Design Types

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

where the parameters are the bias vectors **b** and **c** along with the weight matrices **U**, **V** and **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.

Sequence Modeling: RNNs Design Types

- The total loss for a given sequence of x values paired with a sequence of y values would then be just the sum of the losses over all the time steps.
- For example, if $\mathcal{L}^{(t)}$ is the negative log-likelihood of $y^{(t)}$ given $x^{(1)}, \ldots, x^{(t)}$, 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 $p_{\text{model}}(y^{(t)} \mid \{x^{(1)}, \ldots, x^{(t)}\})$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{\mathbf{y}}^{(t)}$.

Sequence Modeling: RNNs Design Types

- Computing the gradient of this loss function with respect to the parameters is an expensive operation.
  - The gradient computation involves performing a forward propagation pass through our illustration of the unrolled graph in Fig. 10.3, followed by a backward propagation pass left through the graph.
  - The runtime is O(τ) and cannot be reduced by parallelization because the forward propagation graph is inherently sequential; (each time step may only be computed after the previous one).
  - The memory cost is also O(τ), as states computed in the forward pass must be stored until they are reused during the backward pass.
- The back-propagation algorithm applied to the unrolled graph with O(τ) cost is called back-propagation through time or BPTT.
- The network with recurrence between hidden units is thus very powerful but also expensive to train. ---- Is there an alternative? (slide #28 teacher forcing)

Sequence Modeling: RNNs Design Types

Type-2

- *Less powerful than Type-1?*
- *Why?*

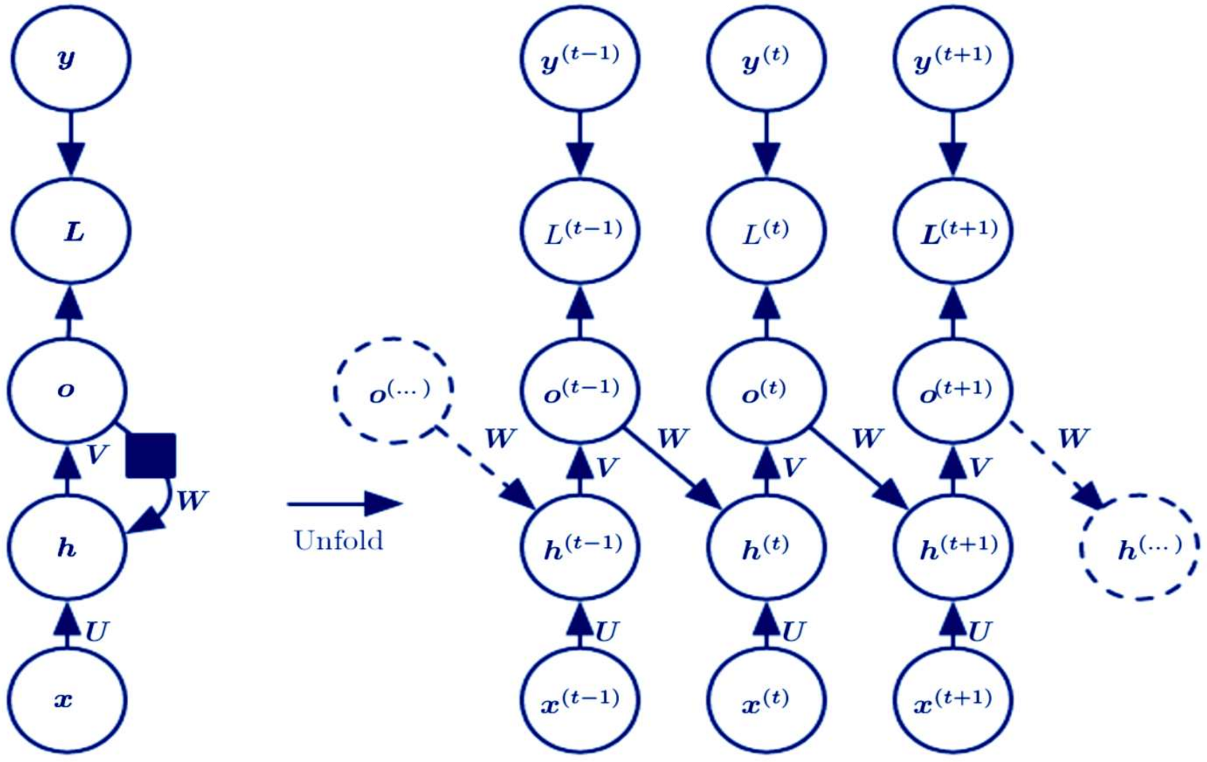


Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer. (Left) Circuit diagram. (Right) Unfolded computational graph.

Sequence Modeling: RNNs Design Types

Figure 10.4: …

- RNNs that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step, illustrated in Fig. 10.4
- At each time step t, the input is xt, the hidden layer activations are h(t), the outputs are o(t), the targets are y(t) and the loss is L(t).
- Such an RNN is less powerful (can express a smaller set of functions) than those in the family represented by Fig. 10.3.
- The RNN in Fig. 10.3 can choose to put any information it wants about the past into its hidden representation h and transmit h to the future.

Sequence Modeling: RNNs Design Types

Figure 10.4: …

- The RNN in this figure is trained to put a specific output value into o , and o is the only information it is allowed to send to the future.
- There are no direct connections from h going forward. The previous h is connected to the present only indirectly, via the predictions it was used to produce.
- Unless o is very high-dimensional and rich, it will usually lack important information from the past.
- This makes the RNN in this figure less powerful, but it may be easier to train because each time step can be trained in isolation from the others, allowing greater parallelization during training??

Sequence Modeling: RNNs Design Types

Type-3

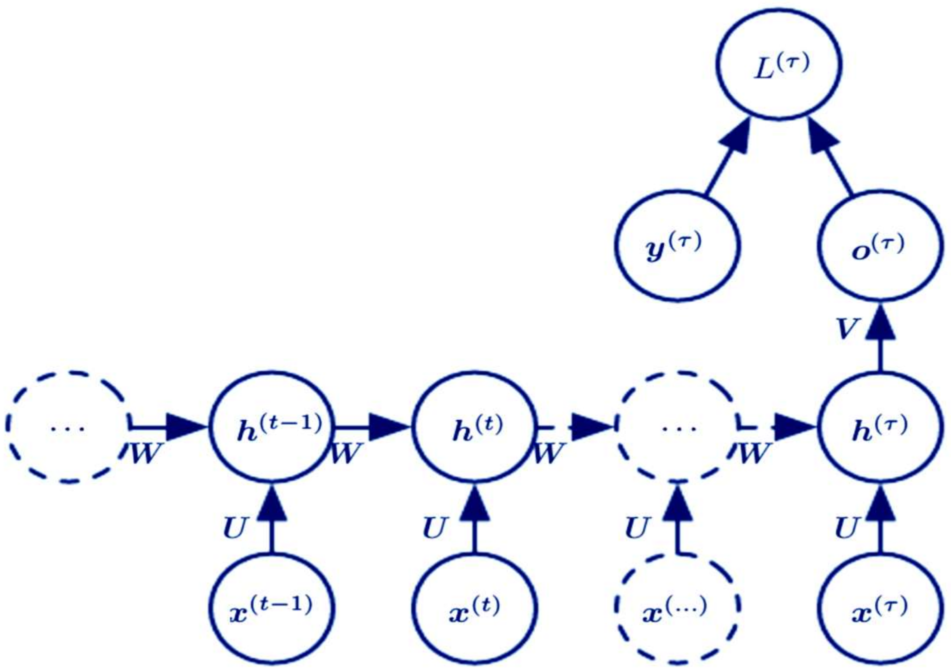


Figure 10.5: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here) or the gradient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

Sequence Modeling: RNNs …
Teacher Forcing and Networks with Output Recurrence

- The network with recurrent connections only from the output at one time step to the hidden units at the next time step (shown in Fig. 10.4) is strictly less powerful because it lacks hidden-to-hidden recurrent connections.
  - It requires that the output units capture all of the information about the past that the network will use to predict the future.
  - Because the output units are explicitly trained to match the training set targets, they are unlikely to capture the necessary information about the past history of the input, unless the user knows how to describe the full state of the system and provides it as part of the training set targets.

Sequence Modeling: RNNs …
Teacher Forcing and Networks with Output Recurrence

- On the other hand, the advantage of eliminating hidden-to-hidden recurrence is that,
  - for any loss function based on comparing the prediction at time t to the training target at time t, all the time steps are decoupled.
  - Training can thus be parallelized, with the gradient for each step t computed in isolation.
  - There is no need to compute the output for the previous time step first, because the training set provides the ideal value of that output.

- Models that have recurrent connections from their outputs leading back into the model (Fig. 10.4) may be trained with teacher forcing.

Sequence Modeling: RNNs …

Teacher Forcing and Networks with Output Recurrence

- Teacher forcing is a procedure that emerges from the maximum likelihood criterion, in which during training the model receives the ground truth output y(t) as input at time t + 1.
- We can see this by examining a sequence with two time steps. The conditional maximum likelihood criterion is:

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

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

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

Sequence Modeling: RNNs …

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.

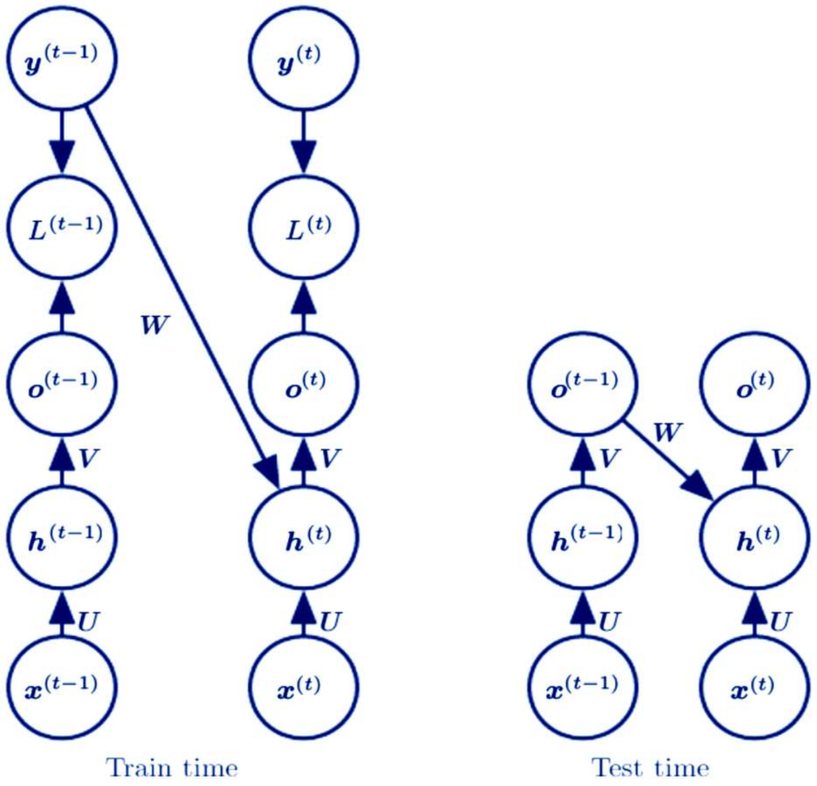


Figure 10.6: Illustration of teacher forcing. Teacher forcing is a training technique that is applicable to RNNs that have connections from their output to their hidden states at the next time step. (Left) At train time, we feed the correct output y(t) drawn from the train set as input to h(t+1) . (Right)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

Sequence Modeling: RNNs …
Teacher Forcing and Networks with Output Recurrence

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

Sequence Modeling: RNNs …

Teacher Forcing and Networks with Output Recurrence

- The disadvantage of strict teacher forcing arises if the network is going to be later used in an open-loop mode, with the network outputs (or samples from the output distribution) fed back as input.
- In this case, the kind of inputs that the network sees during training could be quite different from the kind of inputs that it will see at test time.
- One way to mitigate this problem is to train with both teacher-forced inputs and with free-running inputs, for example by predicting the correct target a number of steps in the future through the unfolded recurrent output-to-input paths.

Sequence Modeling: RNNs …

Teacher Forcing and Networks with Output Recurrence

- In this way, the network can learn to take into account input conditions (such as those it generates itself in the free-running mode) not seen during training and how to map the state back towards one that will make the network generate proper outputs after a few steps.
- Another approach (Bengio et al., 2015b) to mitigate the gap between the inputs seen at train time and the inputs seen at test time randomly chooses to use generated values or actual data values as input. This approach exploits a curriculum learning strategy to gradually use more of the generated values as input.

Sequence Modeling: Bidirectional RNNs

- All of the RNNs we have considered up to now have a "causal" structure, meaning that the state at time t only captures information from the past, $x^{(1)}, \ldots, x^{(t-1)}$, and the present input $x^{(t)}$. Some also allow information from past y values to affect the current state when the y values are available.
- However, in many applications we want to output a prediction of $y^{(t)}$ which may depend on the whole input sequence.
  - Eg. 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.

Sequence Modeling: Bidirectional RNNs

- 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).

Sequence Modeling: Bidirectional RNNs

- 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. (Fig. 10.11).
- This allows the output units $\mathbf{o}^{(t)}$ to compute a representation that depends on both the past and the future but is most sensitive to the input values around time t, without having to specify a fixed-size window around t (as one would have to do with a feedforward network, a convolutional network, or a regular RNN with a fixed-size look-ahead buffer).

Sequence Modeling: Bidirectional RNNs

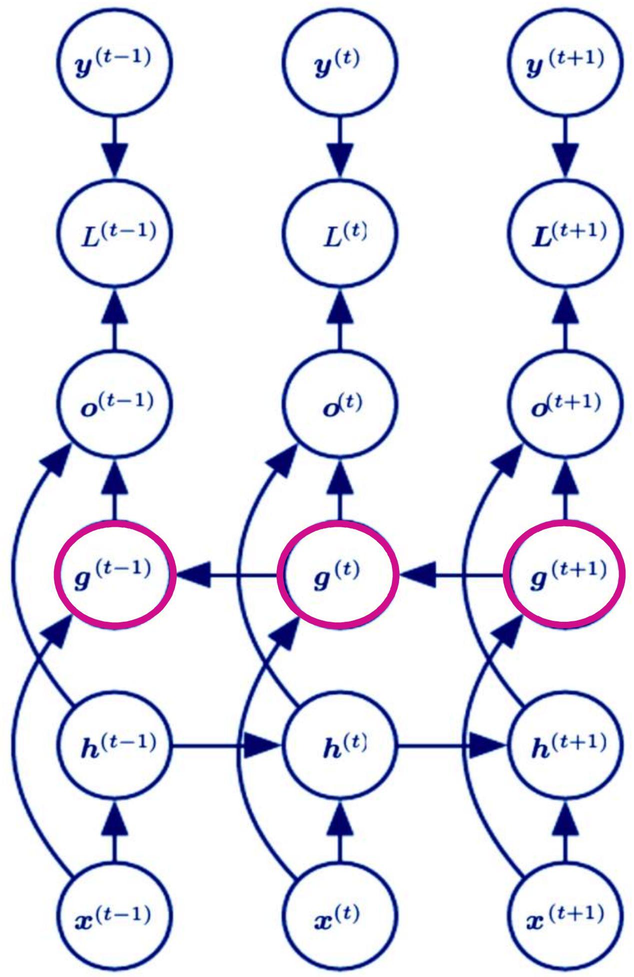


Figure 10.11: Computation of a typical bidirectional recurrent neural network, meant to learn to map input sequences x to target sequences y, with loss L(t) at each step t.

The h recurrence propagates information forward in time (towards the right) while the g recurrence propagates information backward in time (towards the left).

Thus at each point t , the output units o(t) can benefit from a relevant summary of the past in its h(t) input and from a relevant summary of the future in its g(t) input.

Sequence Modeling: Bidirectional RNNs

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

Sequence Modeling: Encoder-Decoder Sequence-to-Sequence Architectures

- Recall that, we have seen
  - Fig. 10.5 RNN can map an input sequence to a fixed-size vector.
  - Fig. 10.9 RNN can map a fixed-size vector to a sequence.
  - Fig. 10.3, Fig. 10.4, Fig. 10.10 and Fig. 10.11 RNN can map an input sequence to an output sequence of the same length.
- Here we discuss how an RNN can be trained to map an input sequence to an output sequence which is not necessarily of the same length
- This comes up in many applications, such as speech recognition, machine translation 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).

Sequence Modeling: Encoder-Decoder Sequence-to-Sequence Architectures

- We often call the *input* to the RNN the "*context*."
- We want to produce a *representation of this context*, C .
- The context C might be a vector or sequence of vectors that summarize the input sequence **X** = $(x^{(1)}, \ldots, x^{(n_x)})$.
- The simplest RNN architecture for mapping a variable-length sequence to another variable-length sequence was first proposed by Cho et al. (2014a) and shortly after by Sutskever et al. (2014), who independently developed that architecture and were the first to obtain state-of-the-art translation using this approach.
  - The former system is based on scoring proposals generated by another machine translation system, while the latter uses a standalone recurrent network to generate the translations.

Sequence Modeling: Encoder-Decoder Sequence-to-Sequence Architectures

- These authors respectively called this architecture, illustrated in Fig. 10.12, the encoder-decoder or sequence-to-sequence architecture.
- The idea is very simple:
  1) an encoder or reader or input RNN processes the input sequence. The encoder emits the context C, usually as a simple function of its final hidden state.
  2) a decoder or writer or output RNN is conditioned on that fixed-length vector (just like in Fig. 10.9) to generate the output sequence $Y = (y^{(1)}, \dots, y^{(n_y)})$.
- The innovation of this kind of architecture over those presented in earlier sections of this chapter is that the lengths n_x and n_y can vary from each other, while previous architectures constrained $n_x = n_y = \tau$.

Sequence Modeling: Encoder-Decoder Sequence-to-Sequence Architectures

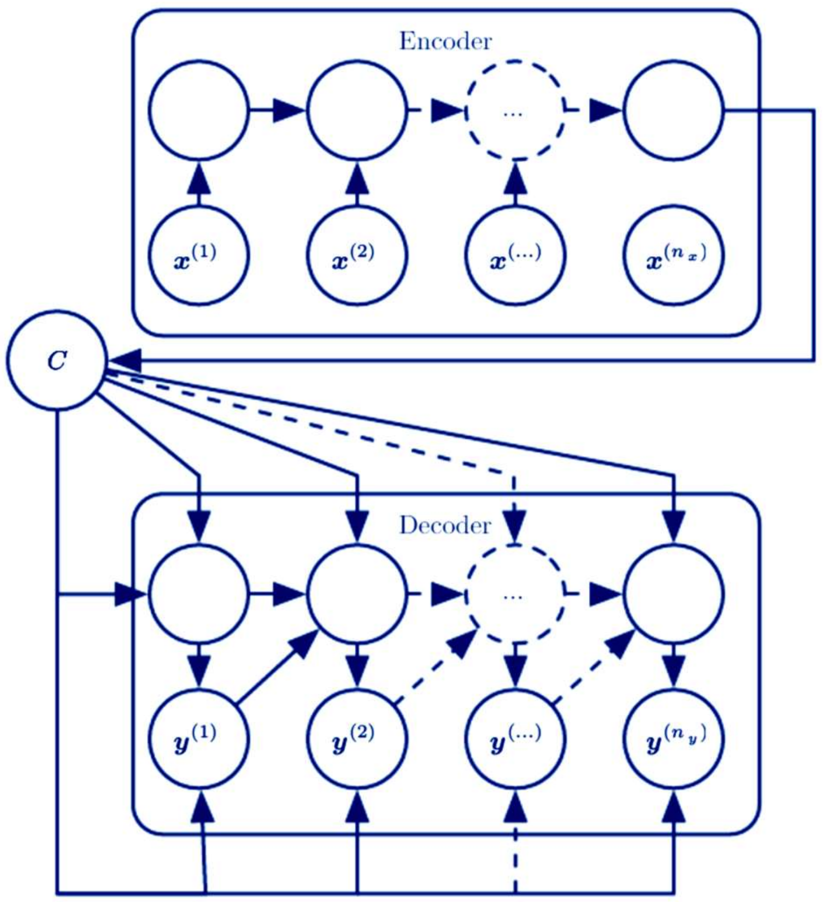


Figure 10.12: Example of an encoder-decoder or sequence-to-sequence RNN architecture, for learning to generate an output sequence (y(1), . . . , y(n y)) given an input sequence (x(1) , x(2) , . . . , x(nx)).

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 is given as input to the decoder RNN.

Sequence Modeling: Encoder-Decoder Sequence-to-Sequence Architectures

- In a sequence-to-sequence architecture, the two RNNs are trained jointly to maximize the average of

$$\log P(y^{(1)}, \ldots, y^{(n_y)} \mid x^{(1)}, \ldots, x^{(n_x)})$$

over all the pairs of x and y sequences in the training set.

- The last state h_{n_x} of the encoder RNN is typically used as a representation C of the input sequence that is provided as input to the decoder RNN.
- If the context C is a vector, then the decoder RNN is simply a vector-to sequence RNN. As we have seen, there are at least two ways for a vector-to-sequence RNN to receive input.
  1. The input can be provided as the initial state of the RNN, or
  2. the input can be connected to the hidden units at each time step.
- These two ways can also be combined.

Sequence Modeling: Encoder-Decoder Sequence-to-Sequence Architectures

- There is no constraint that the encoder must have the same size of hidden layer as the decoder.
- One clear limitation of this architecture is when the context C output by the encoder RNN has a dimension that is too small to properly summarize a long sequence.
- This phenomenon was observed by Bahdanau et al. (2015) in the context of machine translation.
  - They proposed to *make C a variable-length sequence* rather than a fixed-size vector.
  - Additionally, they *introduced an attention mechanism* that learns to associate elements of the sequence C to elements of the output sequence.

Sequence Modeling: Deep Recurrent Networks

- The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:
  1. from the input to the hidden state,
  2. from the previous hidden state to the next hidden state, and
  3. from the hidden state to the output.
- With the RNN architecture of Fig. 10.3, each of these three blocks is associated with a single weight matrix.
  - In other words, when the network is unfolded, each of these corresponds to a shallow transformation.
- By a shallow transformation, we mean a transformation that would be represented by a single layer within a deep MLP.
- Typically this is a transformation represented by a learned affine transformation followed by a fixed nonlinearity.

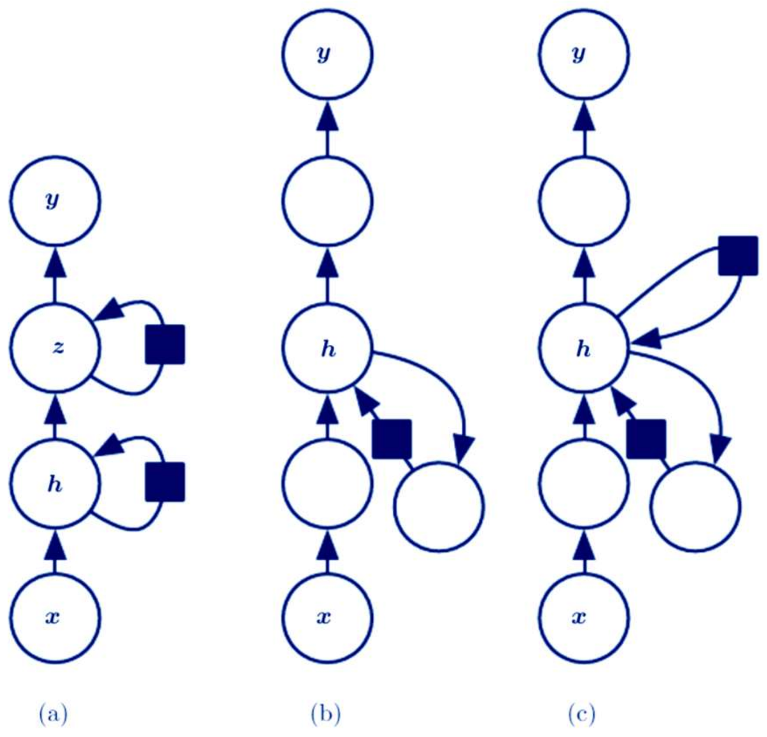
Sequence Modeling: Deep Recurrent Networks

Figure 10.13: A recurrent neural network can be made deep in many ways (Pascanu et al., 2014a). The hidden recurrent state can be broken down in to:

(a) Groups organized hierarchically.

(b) Deeper computation (e.g., an MLP) can be introduced in the input-to-hidden, hidden-to-hidden and hidden-to-output parts. This may lengthen the shortest path linking different time steps.

(c) The path-lengthening effect can be mitigated by introducing skip connections.



Sequence Modeling: Deep Recurrent Networks

- Would it be advantageous to introduce depth in each of these operations? Experimental evidence (Graves et al., 2013; Pascanu et al., 2014a) strongly suggests so.
- The experimental evidence is in agreement with the idea that we need enough depth in order to perform the required mappings. See also Schmidhuber (1992), El Hihi and Bengio (1996), or Jaeger (2007a) for earlier work on deep RNNs.
- Graves et al. (2013) were the first to show a significant benefit of decomposing the state of an RNN into multiple layers as in Fig. 10.13 *(left)*.
- We can think of the lower layers in the hierarchy depicted in Fig. 10.13a as playing a role in transforming the raw input into a representation that is more appropriate, at the higher levels of the hidden state.

Sequence Modeling: Deep Recurrent Networks

- 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 Fig. 10.13b.
- Considerations of representational capacity suggest to allocate enough capacity in each of these three steps, but doing so by adding depth may hurt learning by making optimization difficult.
- In general, it is easier to optimize shallower architectures, and adding the extra depth of Fig. 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 Fig. 10.3. However, as argued by Pascanu et al. (2014a), this can be mitigated by introducing skip connections in the hidden-to-hidden path, as illustrated in Fig. 10.13c.

Sequence Modeling: Recursive Neural Networks

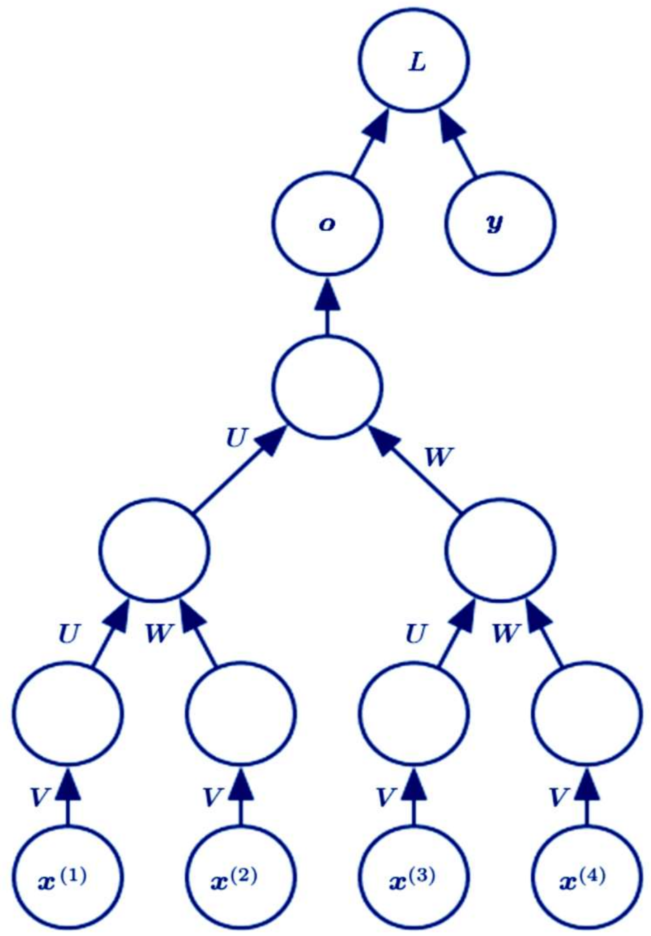
- Recursive neural networks(*don't abbreviate it as RNN*) represent yet another generalization of recurrent networks, with a different kind of computational graph, which is structured as a deep tree, rather than the chain-like structure of RNNs. The typical computational graph for a recursive network is illustrated in Fig. 10.14.
- Recursive neural networks were introduced by Pollack (1990) and their potential use for learning to reason was described by Bottou (2011).
- Recursive networks have been successfully applied to *processing data structures as input to neural nets* (Frasconi et al., 1997, 1998), in NLP (Socher et al., 2011a,c, 2013a) as well as in computer vision (Socher et al., 2011b).

Sequence Modeling: Recursive Neural Networks

Figure 10.14: A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree.

A variable-size sequence $x^{(1)}$, $x^{(2)}$, . . . , $x^{(t)}$ can be mapped to a fixed-size representation (the output o), with a fixed set of parameters (the weight matrices U , V , W).

The figure illustrates a supervised learning case in which some target y is provided which is associated with the whole sequence.



Sequence Modeling: Recursive Neural Networks

- One clear advantage of recursive nets over recurrent nets is that for a sequence of the same length τ, the depth (measured as the number of compositions of nonlinear operations) can be drastically reduced from τ to $O(\log\tau)$, which might help deal with long-term dependencies.
- An open question is how to best structure the tree.
  - One option is to have a tree structure which does not depend on the data, such as a *balanced binary tree*.
  - In some application domains, *external methods* can suggest the appropriate tree structure.
    - For example, when processing natural language sentences, the tree structure for the recursive network can be fixed to the structure of the parse tree of the sentence provided by a natural language parser (Socher et al. ,2011a, 2013a).
  - Ideally, one would like the learner itself to discover and infer the tree structure that is appropriate for any given input, as suggested by Bottou (2011).

Sequence Modeling: Recursive Neural Networks

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

Sequence Modeling: Challenge of Long-Term Dependencies

- Exploding gradients (rarely)
- Vanishing gradients (most of the time)

Sequence Modeling: Echo State Networks

Reading Assignment

- Echo State Networks

Sequence Modeling: 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, other parts operate at
  - coarse time scales and transfer information from the distant past to the present more efficiently.
- Strategies for building both fine and coarse time scales 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.

Sequence Modeling: Leaky Units and Other Strategies for Multiple 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.
- The idea of using such skip connections dates back to Lin et al. (1996) and follows from the idea of incorporating delays in feedforward neural networks (Lang and Hinton, 1988).
- In an ordinary recurrent network, a recurrent connection goes from a unit at time t to a unit at time t+1.
- It is possible to construct recurrent networks with longer delays (Bengio, 1991).

Sequence Modeling: Leaky Units and Other Strategies for Multiple Time Scales

Adding skip connections through time

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

Sequence Modeling: Leaky Units and Other Strategies for Multiple Time Scales

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

Sequence Modeling: Leaky Units and Other Strategies for Multiple Time Scales

Leaky units and a spectrum of different time scales

- Skip connections through d time steps are a way of ensuring that a unit can always learn to be influenced by a value from d time steps earlier.
- The use of a linear self-connection with a weight near one is a different way of ensuring that the unit can access values from the past.
- *The linear self-connection approach allows this effect to be adapted more smoothly and flexibly by adjusting the real-valued a rather than by adjusting the integer-valued skip length d.*
- These ideas were proposed by Mozer (1992) and by El Hihi and Bengio (1996). Leaky units were also found to be useful in the context of echo state networks (Jaeger et al., 2007).

Sequence Modeling: Leaky Units and Other Strategies for Multiple Time Scales

Leaky units and a spectrum of different time scales

- There are two basic strategies for setting the time constants used by leaky units.
  - One strategy is to manually fix them to values that remain constant, for example by sampling their values from some distribution once at initialization time.
  - Another strategy is to make the time constants free parameters and learn them.
- Having such leaky units at different time scales appears to help with long-term dependencies (Mozer, 1992; Pascanu et al., 2013a).

Sequence Modeling: Leaky Units and Other Strategies for Multiple Time Scales

Removing connections

- Another approach to handle long-term dependencies is the idea of organizing the state of the RNN at multiple time-scales (El Hihi and Bengio, 1996), with information flowing more easily through long distances at the slower time scales.
- This idea differs from the skip connections through time discussed earlier because it involves actively removing length-one connections and replacing them with longer connections.
- Units modified in such a way are forced to operate on a long time scale.
- *Skip connections through time add edges. Units receiving such new connections may learn to operate on a long time scale but may also choose to focus on their other short-term connections.*

Sequence Modeling: Leaky Units and Other Strategies for Multiple Time Scales

Removing connections

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

Sequence Modeling: LSTM and Other Gated RNNs

- As of this writing, the most effective sequence models used in practical applications are called *gated RNNs* . These include the *long short-term memory (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.

Sequence Modeling: LSTM 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 sub-sequences and we want a leaky unit to accumulate evidence inside each sub-subsequence, we need a mechanism to forget the old state by setting it to zero.
- Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it. This is what gated RNNs do.

Sequence Modeling: Long Short-Term Memory (LSTM)

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

Sequence Modeling: Long Short-Term Memory (LSTM)

- The LSTM has been found extremely successful in many applications, such as
  - unconstrained handwriting recognition (Graves et al., 2009),
  - speech recognition (Graves et al., 2013; Graves and Jaitly, 2014),
  - handwriting generation (Graves, 2013),
  - machine translation (Sutskever et al., 2014),
  - image captioning (Kiros et al., 2014b; Vinyals et al., 2014b; Xu et al., 2015) and
  - parsing (Vinyals et al., 2014a).

Sequence Modeling: Long Short-Term Memory (LSTM)

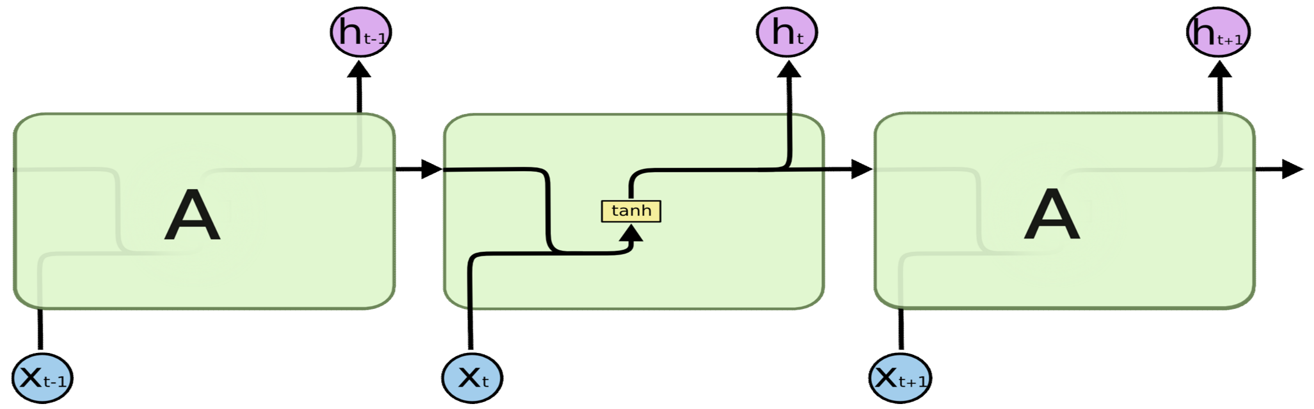
- The LSTM block diagram is illustrated in Fig. 10.16. The corresponding forward propagation equations are given below, in the case of a shallow recurrent network architecture.
- Deeper architectures have also been successfully used (Graves et al., 2013; Pascanu et al., 2014a).
- Instead of a unit that simply applies an elementwise nonlinearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have "LSTM cells" that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN.
- Each cell has the same inputs and outputs as an ordinary recurrent network, but has more parameters and a system of gating units that controls the flow of information.

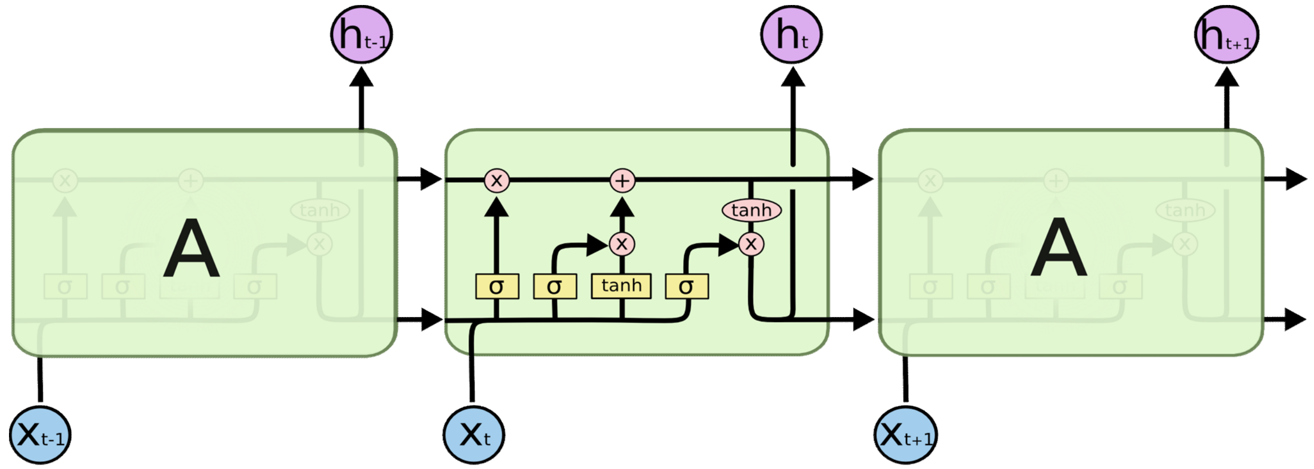
Sequence Modeling: Long Short-Term Memory (LSTM)

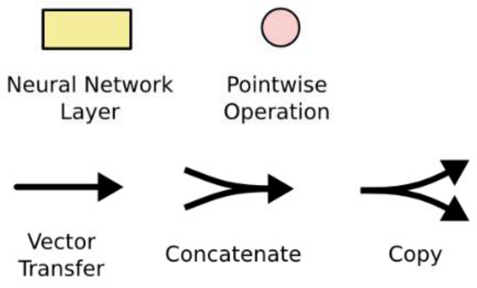
RNN

vs

LSTM







Sequence Modeling: Long Short-Term Memory (LSTM)

- The most important component is the state unit $s^{(t)}_i$ that has a linear self-loop similar to the leaky units described in the previous section.
- However, here, the self-loop weight (or the associated time constant) is controlled by a forget gate unit $f^{(t)}_i$ (for time step t and cell i), that sets this weight to a value between 0 and 1 via a sigmoid unit:

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

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

Sequence Modeling: Long Short-Term Memory (LSTM)

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

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

where b, U and W respectively denote the biases, input weights and recurrent weights into the LSTM cell.

Sequence Modeling: Long Short-Term Memory (LSTM)

- The external input gate unit $g^{(t)}_i$ 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:

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

Sequence Modeling: Long Short-Term Memory (LSTM)

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

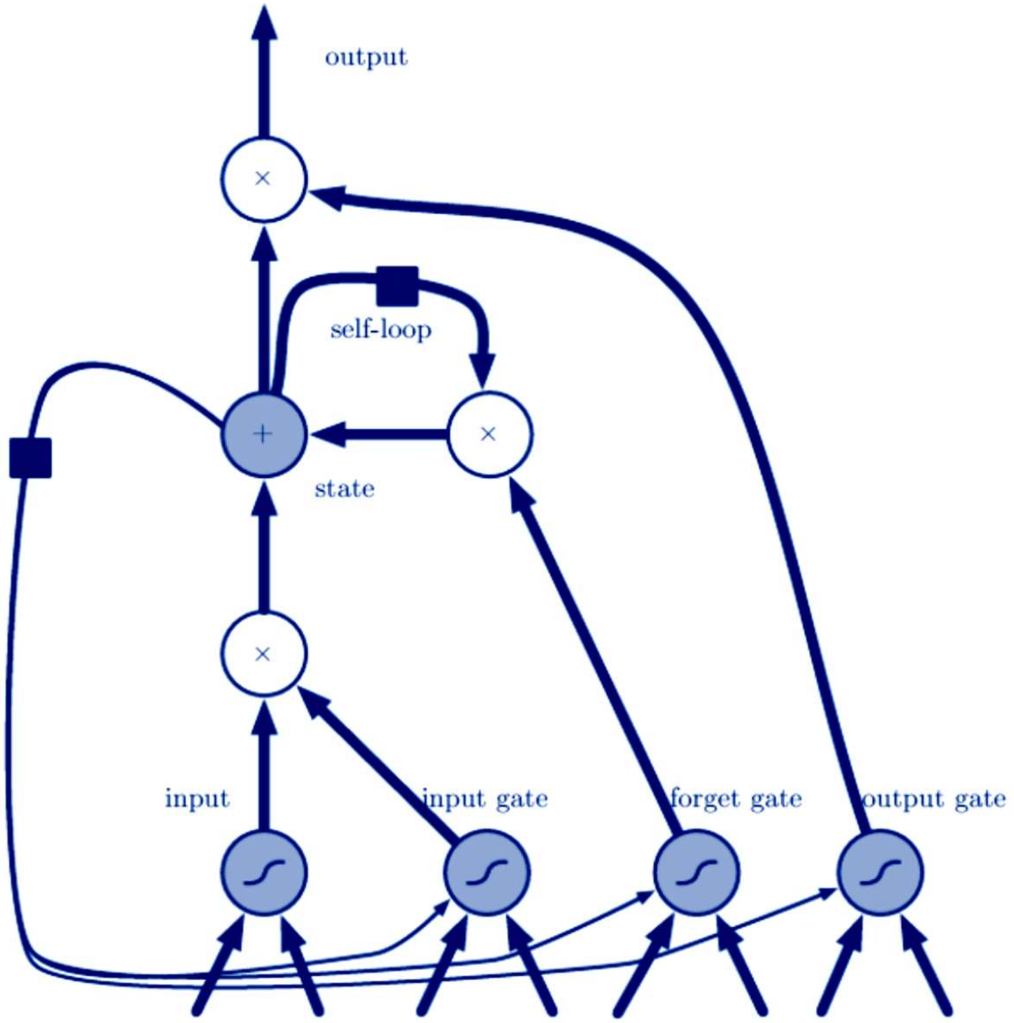
$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)} \tag{10.43}$$

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

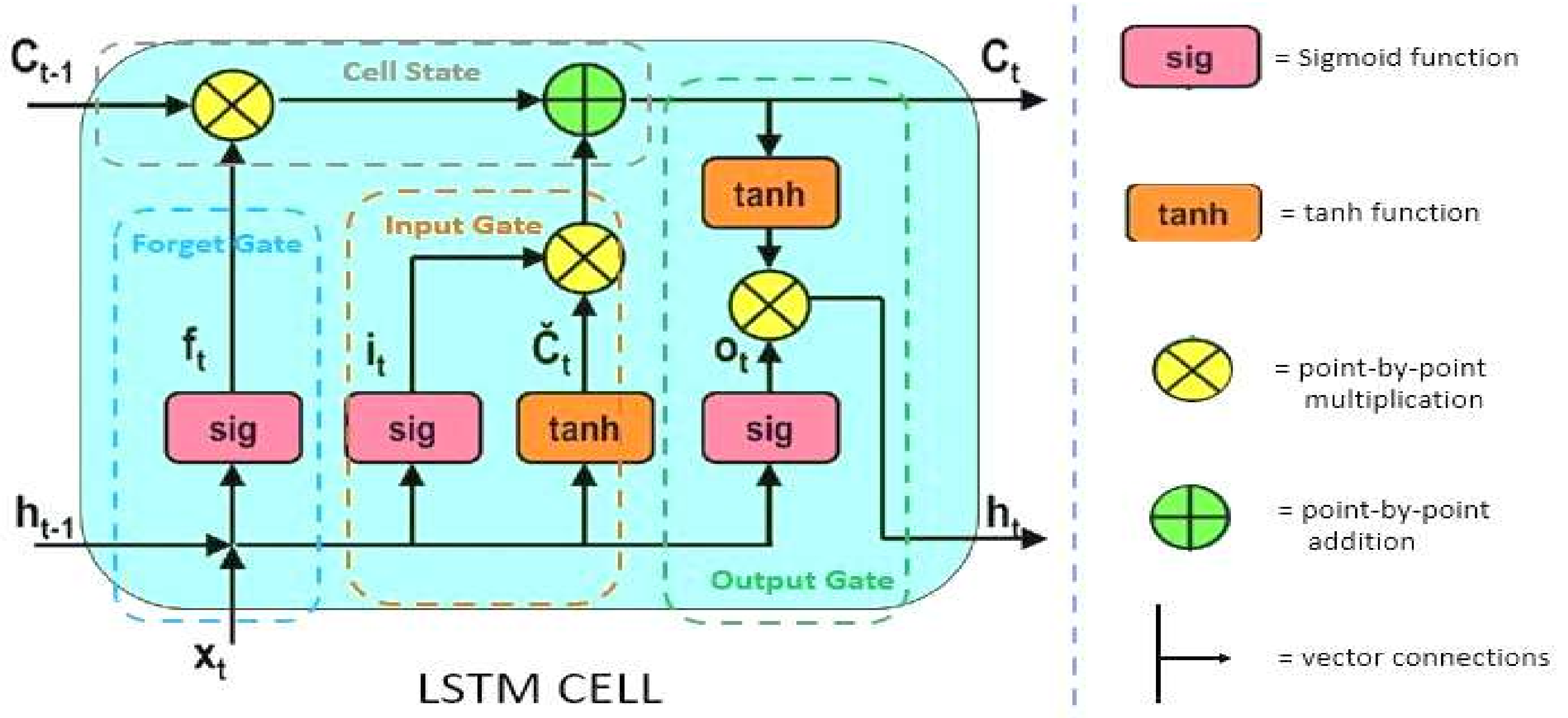
which has parameters b^o, U^o, W^o for its biases, input weights and recurrent weights, respectively. Among the variants, one can choose to use the cell state $s^{(t)}_i$ as an extra input (with its weight) into the three gates of the i-th unit, as shown in Fig. 10.16. This would require three additional parameters.

Sequence Modeling: Long Short-Term Memory (LSTM)

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



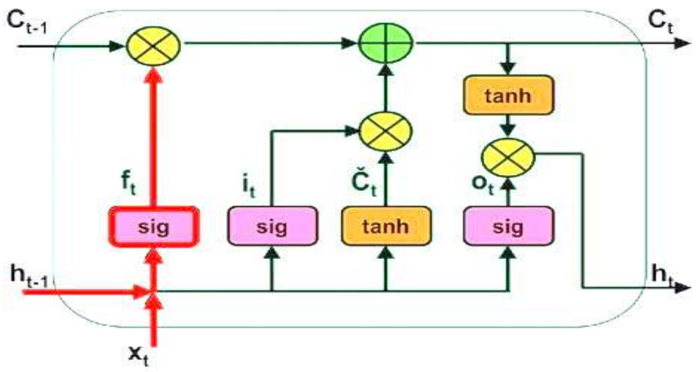
Sequence Modeling: Long Short-Term Memory (LSTM)



Sequence Modeling: Long Short-Term Memory (LSTM)

LSTM: Forget Gate

- The forget gate decides which information needs attention and which can be ignored.
- The information from the current input $X^{(t)}$ and hidden state $h^{(t-1)}$ are passed through the first sigmoid function.
- Sigmoid generates values between 0 and 1.
- It concludes whether the part of the old output is necessary.
- This value of $f^{(t)}$ will later be used by the cell for point-by-point multiplication.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

$t = timestep$

$f_t = forget\ gate\ at\ t$

$x_t = input$

$h_{t-1} = Previous\ hidden\ state$

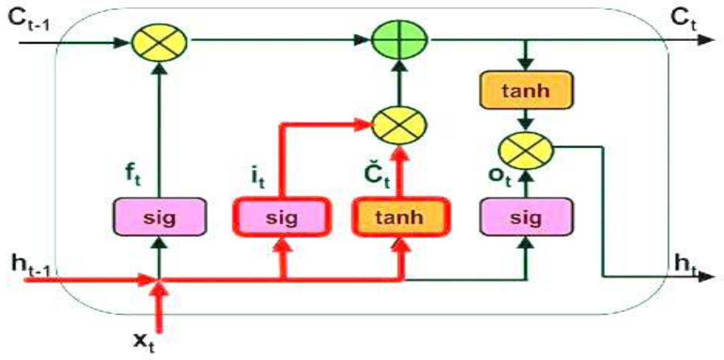
$W_f = Weight\ matrix\ between\ forget\ gate\ and\ input\ gate$

$b_t = connection\ bias\ at\ t$

Sequence Modeling: Long Short-Term Memory (LSTM)

LSTM: Input Gate

- The input gate performs the following operations to update the cell status.
  - First, the current state X^(t)^ and previously hidden state h^(t-1)^ are passed into the second sigmoid function. The values are transformed between 0 (important) and 1 (not-important).
  - Next, the same information of the hidden state and current state will be passed through the tanh function.
  - To regulate the network, the tanh operator will create a vector (C~^(t)^) with all the possible values between -1 and 1. The output values generated from the activation functions are ready for point-by-point multiplication.



$$i_t = \sigma\left(W_i\cdot[h_{t-1}, x_t] \;+\; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C\cdot[h_{t-1}, x_t] \;+\; b_C)$$

$t = timestep$

$i_t = input\ gate\ at\ t$

$W_i = Weight\ matrix\ of\ sigmoid\ operator\ between\ input\ gate\ and\ output\ gate$

$b_t = \ bias\ vector\ at\ t$

$C{\sim}_t = value\ genrated\ by\ tanh$

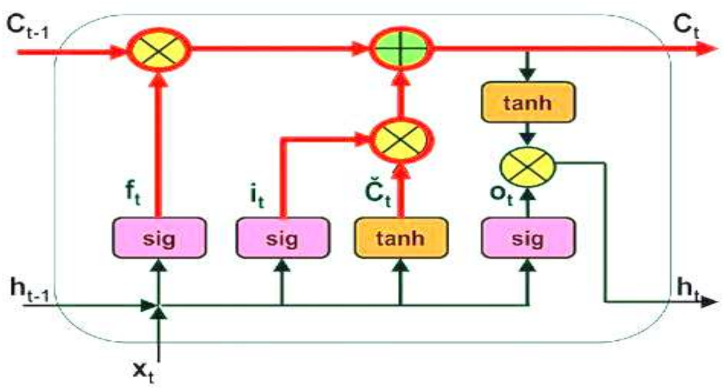
$W_c = \ Weight\ matrix\ of\ tanh\ operator\ between\ cell\ state\ information\ and\ network\ output$

$b_c = \ bias\ vector\ at\ t.w.r.t\ W_c$

Sequence Modeling: Long Short-Term Memory (LSTM)

LSTM: Cell State

- The network has enough information from the forget gate and input gate.
- The next step is to decide and store the information from the new state in the cell state.
- The previous cell state $C^{(t-1)}$ gets multiplied with forget vector $f^{(t)}$. If the outcome is 0, then values will get dropped in the cell state.
- Next, the network takes the output value of the input vector $i^{(t)}$ and performs point-by-point addition, which updates the cell state giving the network a new cell state $C^{(t)}$.



$$\boxed{C_t = f_t * C_{t-1} + i_t * \tilde{C}_t}$$

$t = timestep$

$C_t = cell\ state\ information$

$f_t = forget\ gate\ at\ t$

$i_t = input\ gate\ at\ t$

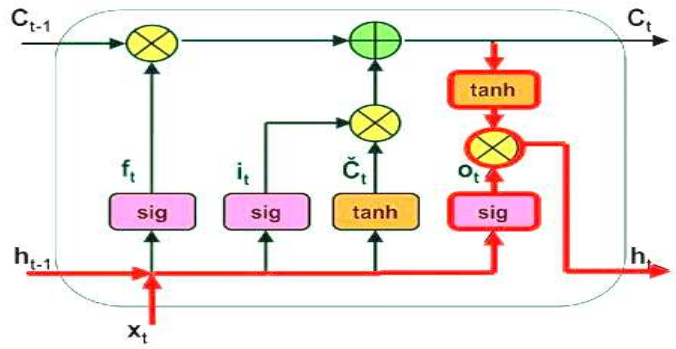
$C_{t-1} = Previous\ timestemp$

$C{\sim}_t = value\ genrated\ by\ tanh$

Sequence Modeling: Long Short-Term Memory (LSTM)

LSTM: Output Gate

- The output gate determines the value of the next hidden state.
  - First, the values of the current *state?* and previous hidden state are passed into the third sigmoid function.
  - Then the new cell state generated from the cell state is passed through the tanh function. Both these outputs are multiplied point-by-point. Based upon the final value, the network decides which information the hidden state should carry. This hidden state is used for prediction.
  - Finally, the new cell state and new hidden state are carried over to the next time step.
- In sum, the forget gate decides which relevant information from the prior steps is needed. The input gate decides what relevant information can be added from the current step, and the output gates finalize the next hidden state.



$$o_t = \sigma\left(W_o\,[h_{t-1}, x_t] + b_o\right)$$

$$h_t = o_t * \tanh(C_t)$$

$t = timestep$

$O_t = output\ gate\ at\ t$

$W_o = Weight\ matrix\ of\ output\ gate$

$b_o = bias\ vector, w.r.t\ W_o$

$h_t = LSTM\ output$

Sequence Modeling: Long Short-Term Memory (LSTM)

- LSTM networks have been shown to learn long-term dependencies more easily than the simple recurrent architectures, first on artificial data sets designed for testing the ability to learn long-term dependencies (Bengio et al., 1994; Hochreiter and Schmidhuber, 1997; Hochreiter et al., 2001), then on challenging sequence processing tasks where state-of-the-art performance was obtained (Graves, 2012; Graves et al., 2013; Sutskever et al., 2014).
- Variants and alternatives to the LSTM have been studied and used.

Sequence Modeling: LSTM and Other Gated RNNs

Reading Assignment

- Variants of LSTM like GRU

End of Ch-4