RNN & LSTM
Basics & Applications

모두의 연구소

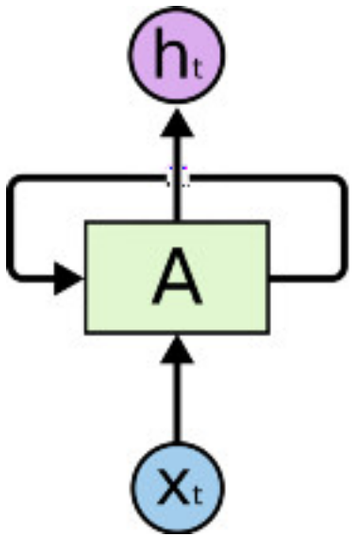
- 2018. 10. 24 -

Recurrent Neural Network

A **recurrent neural network** (RNN) is a class of artificial **neural network** where connections between nodes form a directed graph along a *sequence*.

This allows it to exhibit temporal dynamic behavior for a *time sequence.*

Recurrent Neural Network

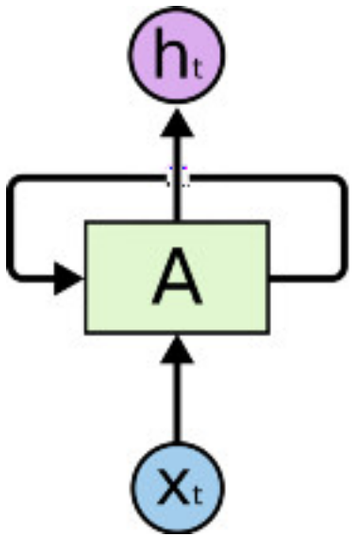


Each member of the output is produced using *the same update rule* applied to the previous outputs:

Less parameters to estimate, generalize to various lengths

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

Recurrent Neural Network

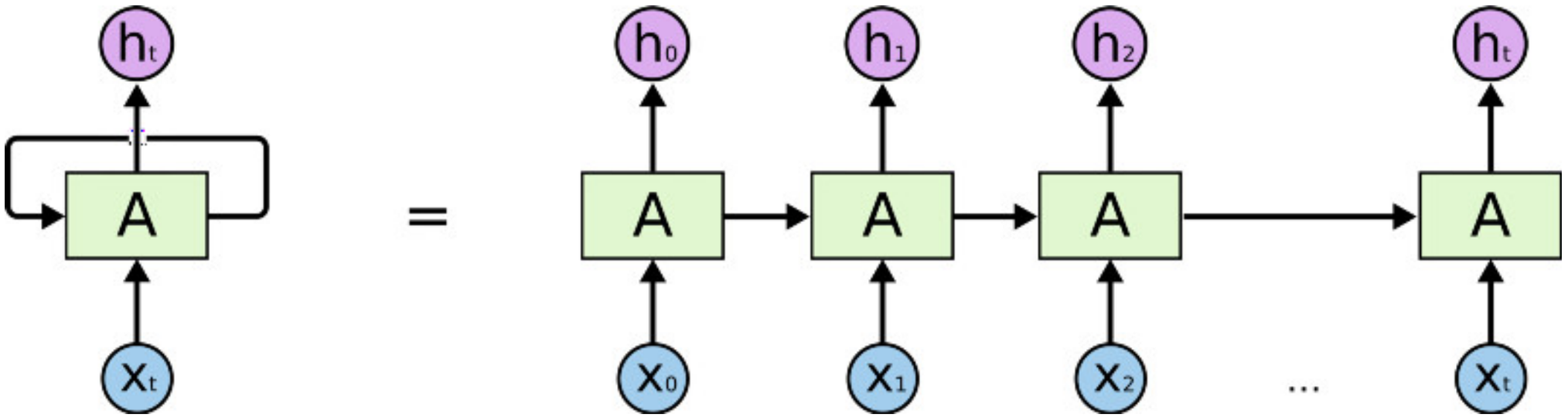


The network typically learns to use *h(t)* as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t.

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

Recurrent Neural Network

Unfolding **Computational Graphs**



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

Vanilla RNN

Recurrent Neural Network

Let's talk about Inputs and Outputs!

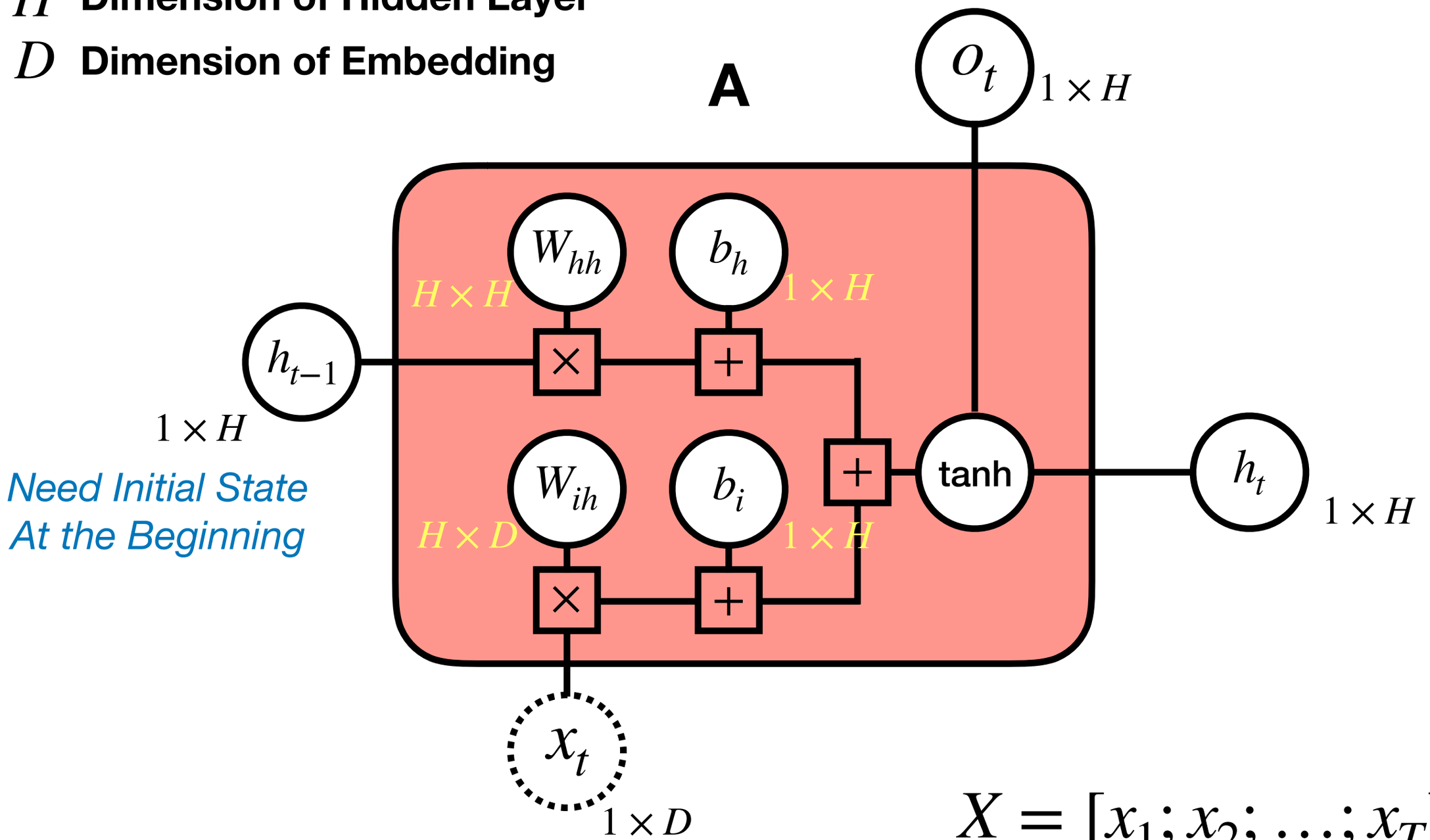
It is dependent on which framework you use.
In my case, it is GLUON
But very similar across the frameworks

Understanding network = **Know every details about input & output**

Vanilla Recurrent Neural Network

H **Dimension of Hidden Layer**

D **Dimension of Embedding**



$$X = [x_1; x_2; \ldots; x_T]$$

$T \times D$

Vanilla Recurrent Neural Network

H : **Dimension of Hidden Layer**

D : **Dimension of Embedding**

T : **Number of time steps**

```
model = rnn.RNN(num_hidden, num_layers
        , activation = 'tanh'
        , layout = 'NTC'
        , input_size = num_emb)
```

$$o_t = h_t = tanh(W_{ih}X_t + W_{hh}h_{t-1})$$

$$dim(W_{ih}) : D \times H$$

$$dim(W_{hh}) : H \times H$$

$$dim(W_o) : O \times H$$

Vanilla Recurrent Neural Network

H : Dimension of Hidden Layer

D : Dimension of Embedding

T : Number of time steps

```
model = rnn.RNN(num_hidden, num_layers
                , activation = 'tanh'
                , layout = 'NTC'
                , input_size = num_emb)
```

$$o_t = h_t = tanh(W_{ih}X_t + W_{hh}h_{t-1})$$

$$dim(W_{ih}) : D \times H$$

$$dim(W_{hh}) : H \times H$$

$$dim(W_o) : O \times H$$

Vanilla Recurrent Neural Network

H : Dimension of Hidden Layer

D : Dimension of Embedding

T : Number of time steps

```
model = rnn.RNN(num_hidden, num_layers
                , activation = 'tanh'
                , layout = 'NTC'
                , input_size = num_emb)
```

$$o_t = h_t = tanh(W_{ih}X_t + W_{hh}h_{t-1})$$

$$dim(W_{ih}) : D \times H$$

$$dim(W_{hh}) : H \times H$$

$$dim(W_o) : O \times H$$

Specified from data shape
Corresponding to C

Vanilla Recurrent Neural Network

H : Dimension of Hidden Layer

D : Dimension of Embedding

T : Number of time steps

Specified by *data shape* corresponding to *T*

```
model = rnn.RNN(num_hidden, num_layers
                , activation = 'tanh'
                , layout = 'NTC'
                , input_size = num_emb)
```

Specified by data shape Corresponding to *C*

$$o_t = h_t = tanh(W_{ih}X_t + W_{hh}h_{t-1})$$

$$dim(W_{ih}) : D \times H$$

$$dim(W_{hh}) : H \times H$$

$$dim(W_o) : O \times H$$

Vanilla Recurrent Neural Network

H : Dimension of Hidden Layer

D : Dimension of Embedding

T : Number of time steps

Specified by *data shape* corresponding to *T*

```
model = rnn.RNN(num_hidden, num_layers
                , activation = 'tanh'
                , layout = 'NTC'
                , input_size = num_emb)
```

$$o_t = h_t = tanh(W_{ih}X_t + W_{hh}h_{t-1})$$

Specified by data shape Corresponding to C

$$dim(W_{ih}) : D \times H$$

$$dim(W_{hh}) : H \times H$$

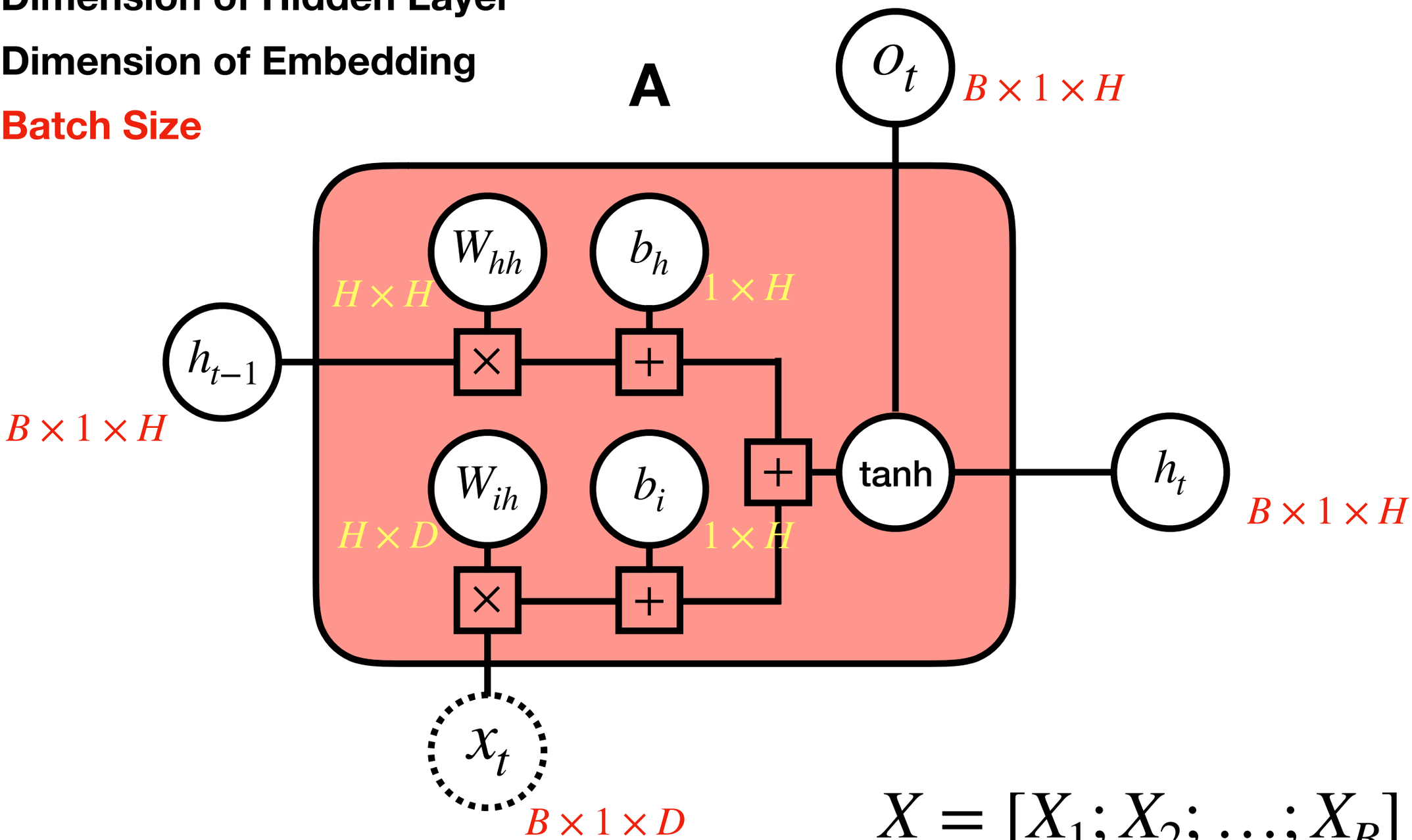
$$dim(W_o) : O \times H$$

Vanilla Recurrent Neural Network

H **Dimension of Hidden Layer**

D **Dimension of Embedding**

B **Batch Size**



$$X = [X_1; X_2; \ldots; X_B]$$

$B \times T \times D$

Vanilla Recurrent Neural Network

Understanding RNN

In [117]:

```
n_hidden_state = 10
embedding_input = 5
model = rnn.RNN(n_hidden_state, 1, layout = 'NTC', input_size = embedding_input)
model.collect_params().initialize(mx.init.Xavier(), ctx = mx.cpu())
initial_state = model.begin_state(batch_size = 16)
```

Hiddens state size

In [118]:

```
print(a.l0_h2h_weight.data().shape)
print(a.l0_h2h_bias.data().shape)
print(a.l0_i2h_weight.data().shape)
print(a.l0_i2h_bias.data().shape)
```

```
(10, 10)
(10,)
(10, 5)
(10,)
```

Vanilla Recurrent Neural Network

T=1 case

In [119]:

```
# Goes only 1 time-step
batch_size = 16
time_step = 1
dat = nd.random.normal(shape =(batch_size, time_step, embedding_input))
out, state = a(dat, initial_state)
```

In [120]:

```
out[0][0] == state[0][0][0]
```

Out[120]:

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
<NDArray 10 @cpu(0)>
```

Vanilla Recurrent Neural Network

In [124]:

```
res = nd.relu(nd.dot(dat[0][0], a.l0_i2h_weight.data(), transpose_b = True) + a.l0_i2h_bias.data() \
      + nd.dot(a.l0_h2h_weight.data(), initial_state[0][0][0]) + a.l0_h2h_bias.data())
```

In [129]:

```
print(res.asnumpy())
print(out[0][0].asnumpy())
print(state[0][0][0].asnumpy())
```

```
[0.105 0.    0.    0.    0.425 1.134 0.    0.    0.    0.18 ]
[0.105 0.    0.    0.    0.425 1.134 0.    0.    0.    0.18 ]
[0.105 0.    0.    0.    0.425 1.134 0.    0.    0.    0.18 ]
```

Vanilla Recurrent Neural Network

In [124]:

```
res = nd.relu(nd.dot(dat[0][0], a.l0_i2h_weight.data(), transpose_b = True) + a.l0_i2h_bias.data() \
      + nd.dot(a.l0_h2h_weight.data(), initial_state[0][0][0]) + a.l0_h2h_bias.data())
```

In [129]:

```
print(res.asnumpy())
print(out[0][0].asnumpy())
print(state[0][0][0].asnumpy())
```

```
[0.105 0.    0.    0.    0.425 1.134 0.    0.    0.    0.18 ]
[0.105 0.    0.    0.    0.425 1.134 0.    0.    0.    0.18 ]
[0.105 0.    0.    0.    0.425 1.134 0.    0.    0.    0.18 ]
```

Vanilla Recurrent Neural Network

T=2 case

In [130]:

```
# Goes only 1 time-step
batch_size = 16
time_step = 2
dat = nd.random.normal(shape =(batch_size, time_step, embedding_input))
out, state = a(dat, initial_state)
```

state is the same as the last time step of out

In [131]:

```
print(out[0][1].asnumpy())
print(state[0][0][0].asnumpy())
```

```
[0.    0.    0.    1.352 0.    0.    0.137 0.    0.    0.259]
[0.    0.    0.    1.352 0.    0.    0.137 0.    0.    0.259]
```

Vanilla Recurrent Neural Network

In [133]:

```
out_t1 = nd.relu(nd.dot(dat[0][0], a.l0_i2h_weight.data(), transpose_b = True) + a.l0_i2h_bias.data() \
      + nd.dot(a.l0_h2h_weight.data(), initial_state[0][0][0]) + a.l0_h2h_bias.data())
```

In [134]:

```
print(out_t1.asnumpy())
print(out[0][0].asnumpy())
```

```
[0.341 0.    0.    0.    1.19  0.    0.    0.244 0.56  0.297]
[0.341 0.    0.    0.    1.19  0.    0.    0.244 0.56  0.297]
```

Vanilla Recurrent Neural Network

In [136]:

```
out_t2 = nd.relu(nd.dot(dat[0][1], a.l0_i2h_weight.data(), transpose_b = True) + a.l0_i2h_bias.data() \
      + nd.dot(a.l0_h2h_weight.data(), out_t1) + a.l0_h2h_bias.data())
```

In [141]:

```
print(out_t2.asnumpy())
print(state[0][0][0].asnumpy())
print(out[0][1].asnumpy())
```

```
[0.    0.    0.    1.352 0.    0.    0.137 0.    0.    0.259]
[0.    0.    0.    1.352 0.    0.    0.137 0.    0.    0.259]
[0.    0.    0.    1.352 0.    0.    0.137 0.    0.    0.259]
```

Architectures

Recurrent Neural Network

Unfolding Computational Graphs

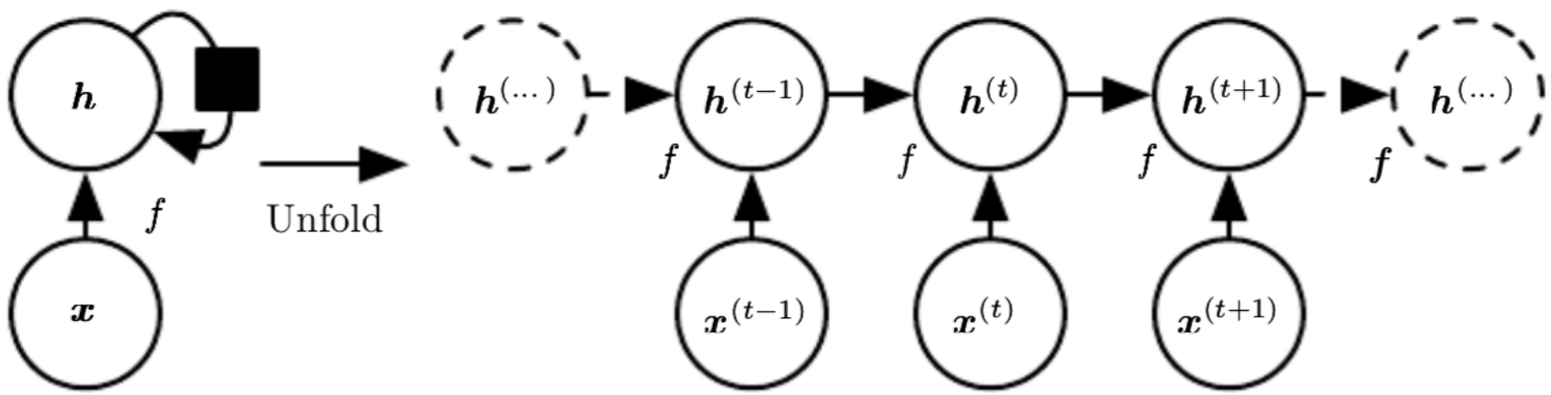


Figure 10.2: A recurrent network with no outputs. This recurrent network just processes information from the input $\boldsymbol{x}$ by incorporating it into the state $\boldsymbol{h}$ that is passed forward through time. *(Left)*Circuit diagram. The black square indicates a delay of a single time step.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

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Recurrent Neural Network

Variation 1 of RNN (hidden2hidden: Transducer)

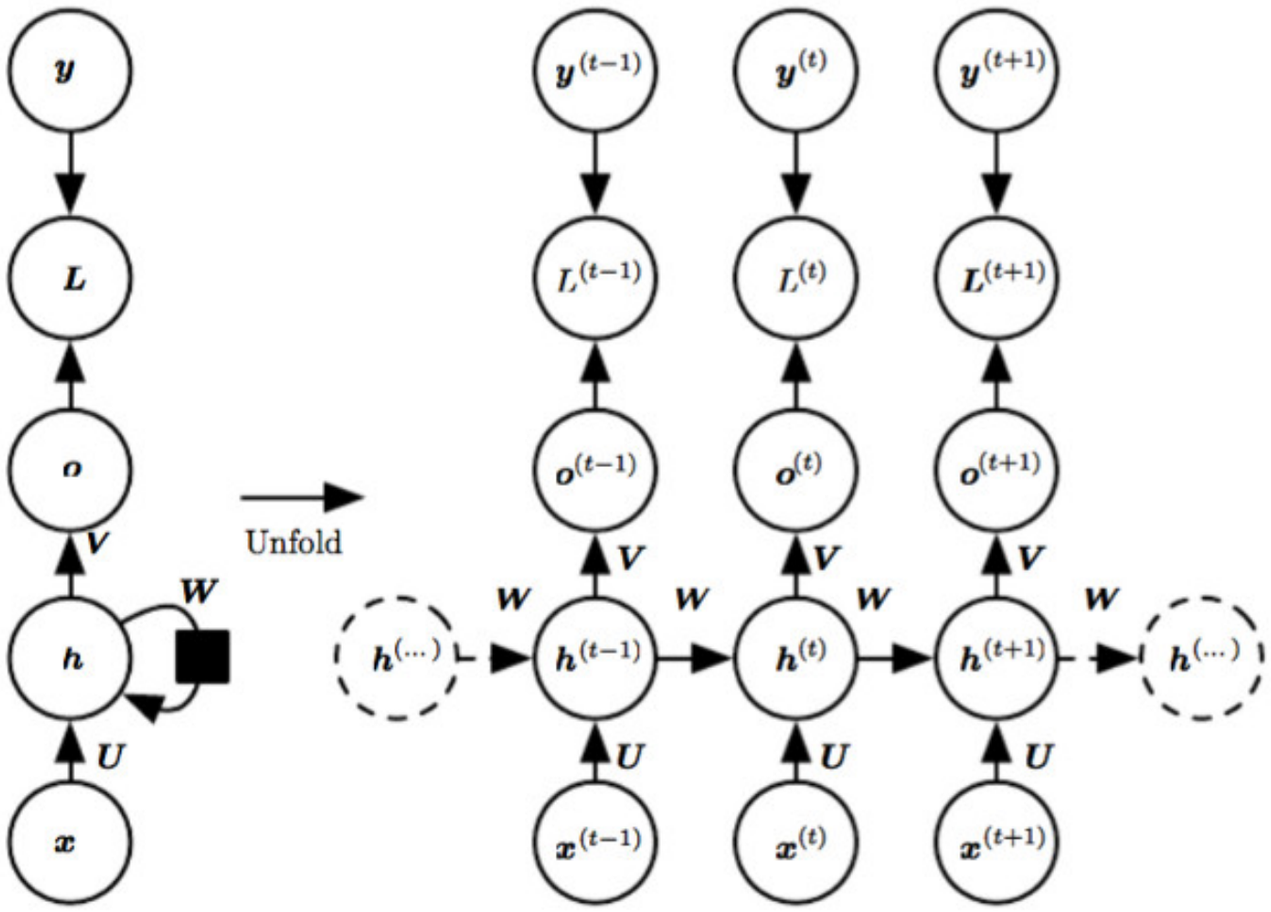


Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of $\boldsymbol{x}$ values to a corresponding sequence of output $\boldsymbol{o}$ values.

Needs to be of the same length

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Recurrent Neural Network

Variation 1 of RNN (hidden2hidden: Transducer)

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

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)}) \tag{10.9}$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \tag{10.10}$$

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}(\boldsymbol{o}^{(t)}) \tag{10.11}$$

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

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

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

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Recurrent Neural Network

Variation 1 of RNN (hidden2hidden: Transducer)

Examples in NLP:

- Sequence tagging
- Language Model
- RNN generator, Encoder-decoder

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Recurrent Neural Network

Variation 2 of RNN (output2hidden: Transducer)

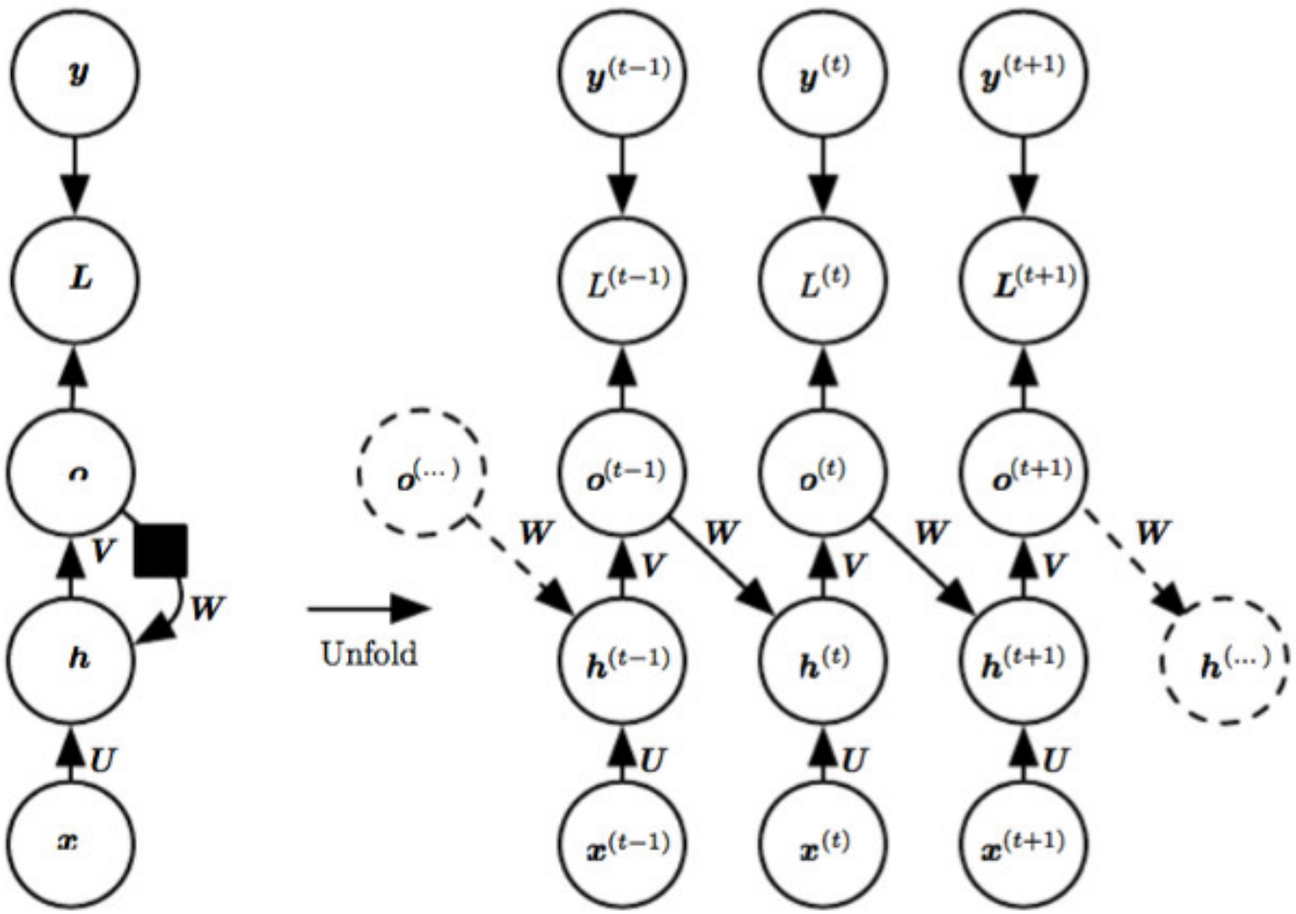


Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time step t, the input is $\boldsymbol{x}_t$, the hidden layer activations are $\boldsymbol{h}^{(t)}$, the outputs are $\boldsymbol{o}^{(t)}$, the targets are $\boldsymbol{y}^{(t)}$ and the loss is $L^{(t)}$. *(Left)* Circuit diagram.

Still needs to be of the same length

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Recurrent Neural Network

Variation 2 of RNN (output2hidden: Transducer)

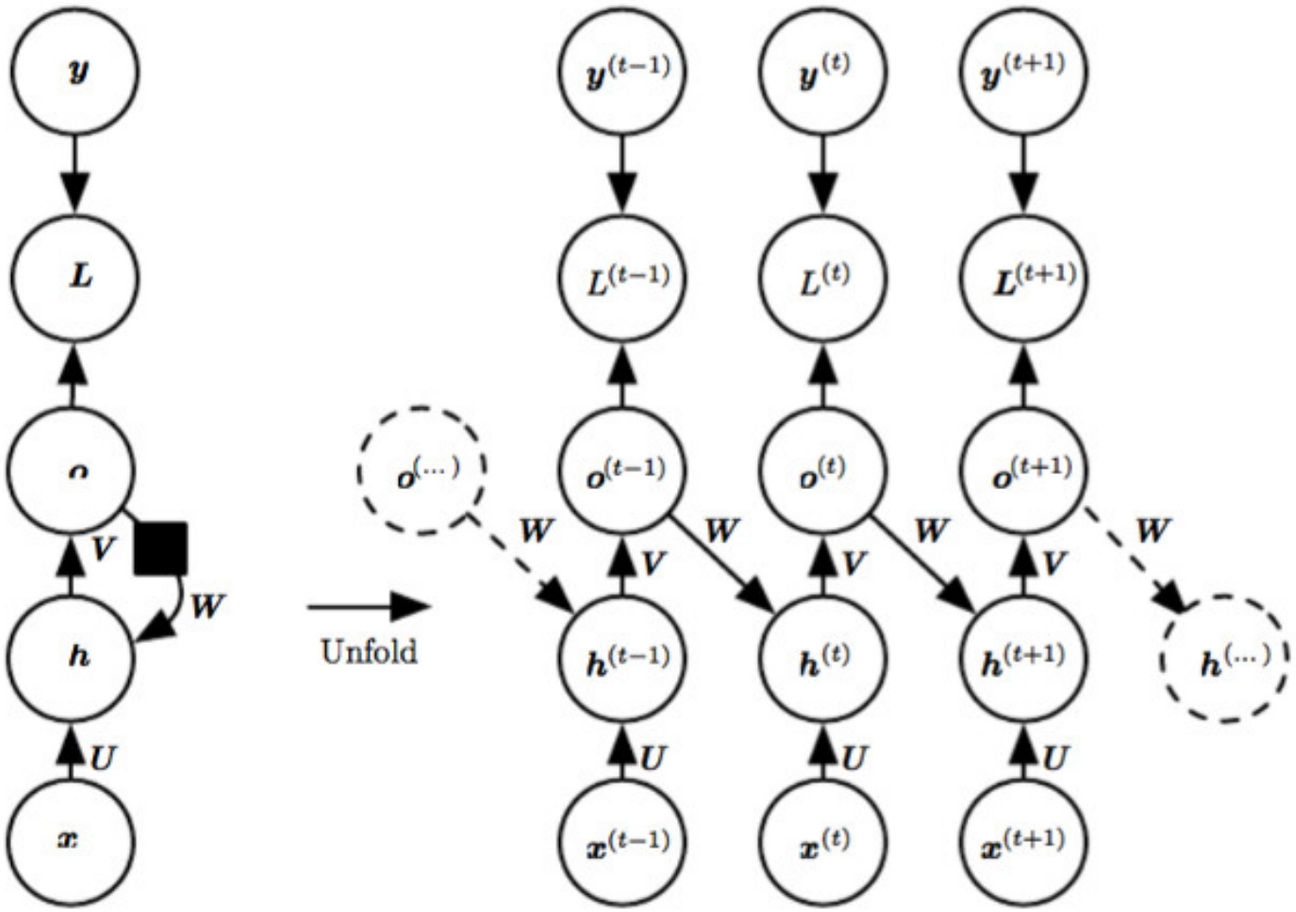


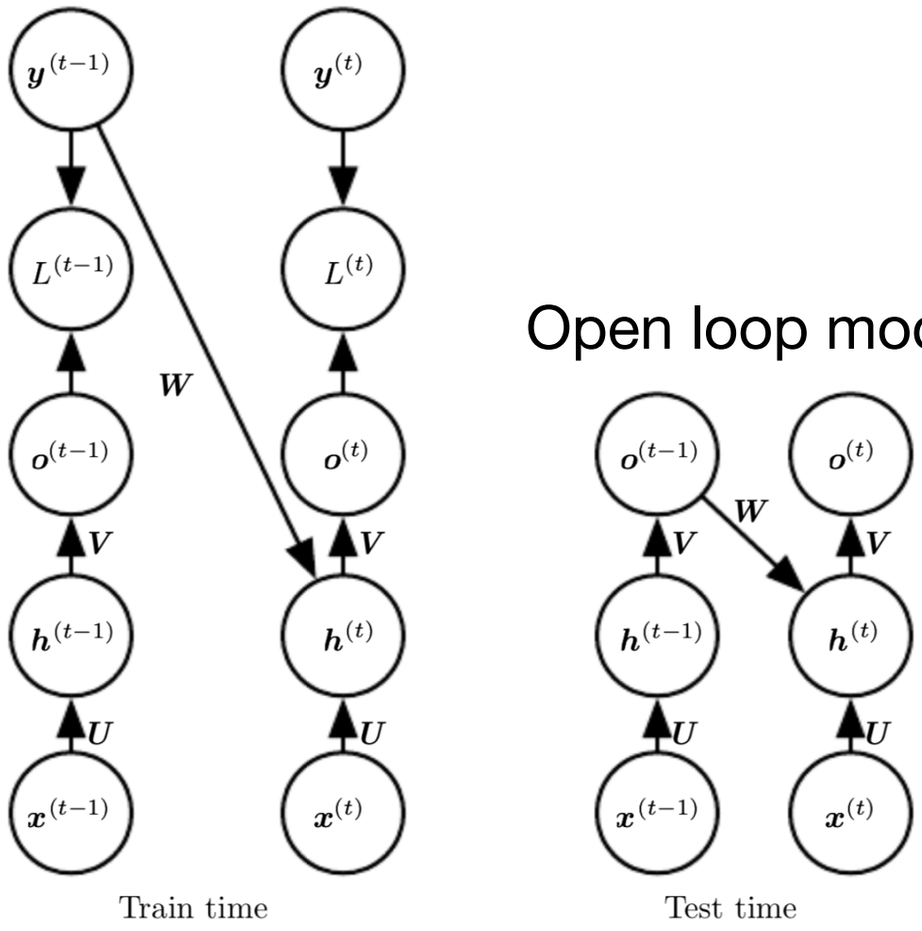
Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time step t, the input is $\boldsymbol{x}_t$, the hidden layer activations are $\boldsymbol{h}^{(t)}$, the outputs are $\boldsymbol{o}^{(t)}$, the targets are $\boldsymbol{y}^{(t)}$ and the loss is $L^{(t)}$. *(Left)* Circuit diagram.

Note: ***Teacher Forcing*** can be used in training

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Recurrent Neural Network

Variation 2 of RNN (output2hidden: Transducer)



Note: ***Teacher Forcing*** can be used in training

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

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Recurrent Neural Network

Variation 3 of RNN: Acceptor

single output

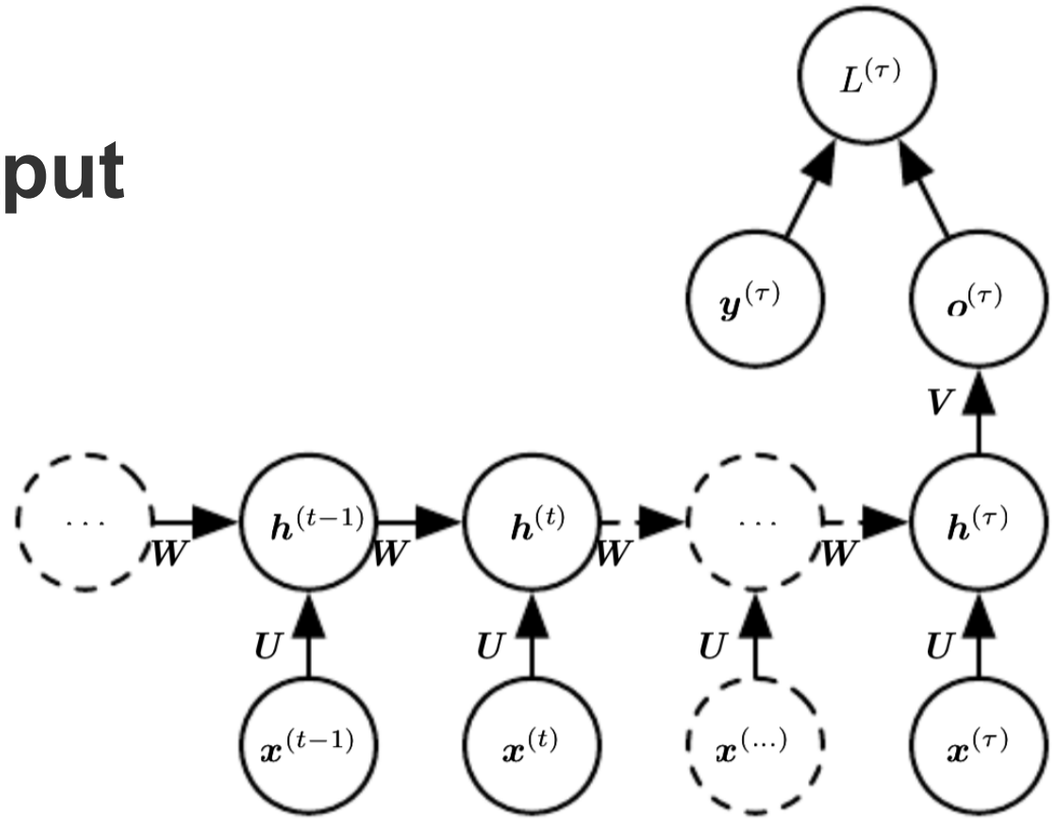


Figure 10.5: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here) or the gradient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

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Fully Connected Graphical Model

This is too much!!

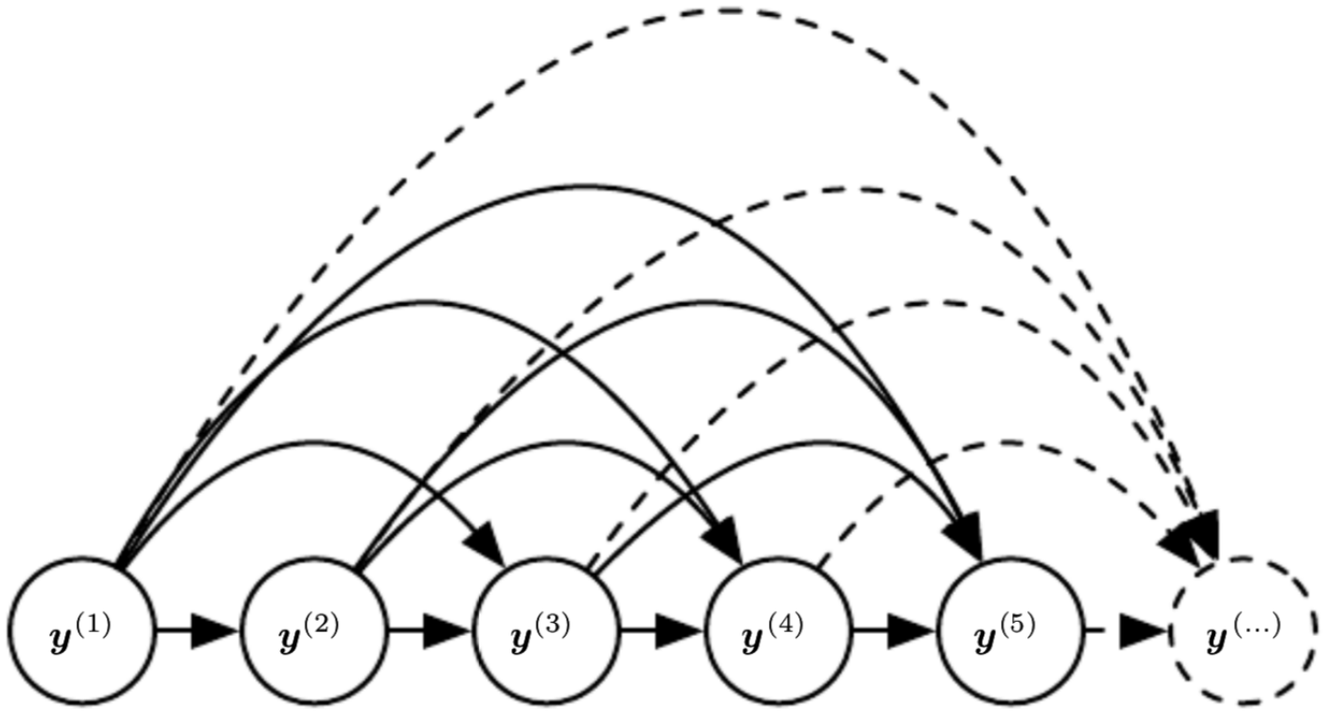


Figure 10.7: Fully connected graphical model for a sequence $y^{(1)}, y^{(2)}, \ldots, y^{(t)}, \ldots$: every past observation $y^{(i)}$ may influence the conditional distribution of some $y^{(t)}$ (for $t > i$), given the previous values. Parametrizing the graphical model directly according to this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

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

RNN - Graphical Model

Introduction to state variable!!

Think about GMM!!!

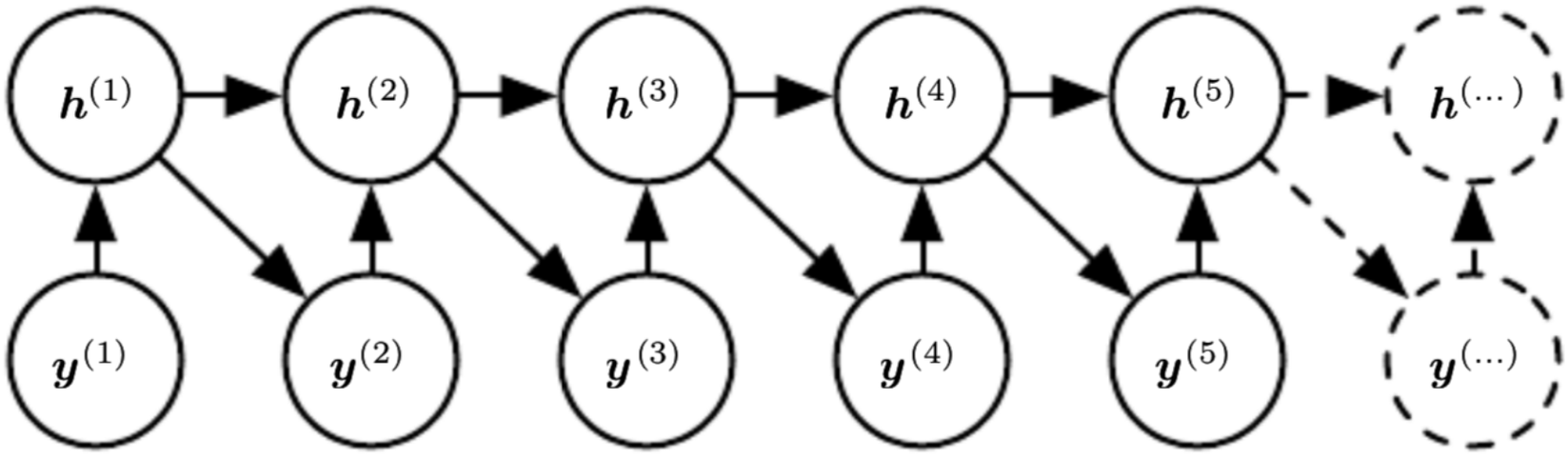


Figure 10.8: Introducing the state variable in the graphical model of the RNN, even though it is a deterministic function of its inputs, helps to see how we can obtain a very efficient parametrization, based on equation 10.5. Every stage in the sequence (for $\boldsymbol{h}^{(t)}$ and $\boldsymbol{y}^{(t)}$)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

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Requries Stationarity

The parameter sharing used in recurrent networks relies on the assumption that the same parameters can be used for different time steps. Equivalently, the assumption is that the conditional probability distribution over the variables at time $t+1$ given the variables at time t is **stationary**, meaning that the relationship between the previous time step and the next time step does not depend on t. In principle, it would be possible to use t as an extra input at each time step and let the learner discover any time-dependence while sharing as much as it can between different time steps. This would already be much better than using a different conditional probability distribution for each t, but the network would then have to extrapolate when faced with new values of t.

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Modeling Sequence Conditioned on Context with RNN

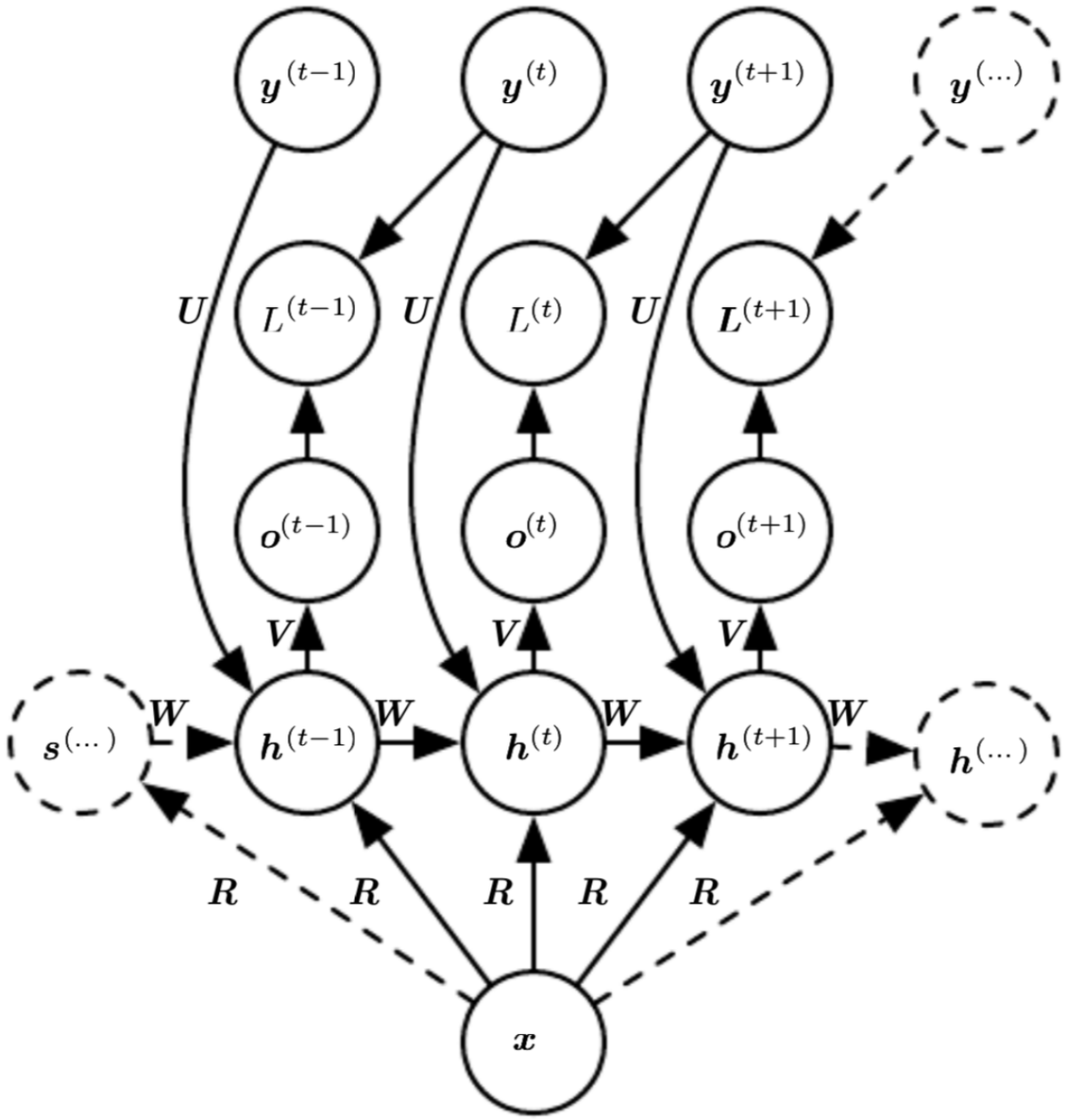


Figure 10.9: An RNN that maps a fixed-length vector $\boldsymbol{x}$ into a distribution over sequences $\mathbf{Y}$. This RNN is appropriate for tasks such as image captioning, where a single image is used as input to a model that then produces a sequence of words describing the image. Each element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

Modeling Sequence Conditioned on Context with RNN

Needs to be of the same length

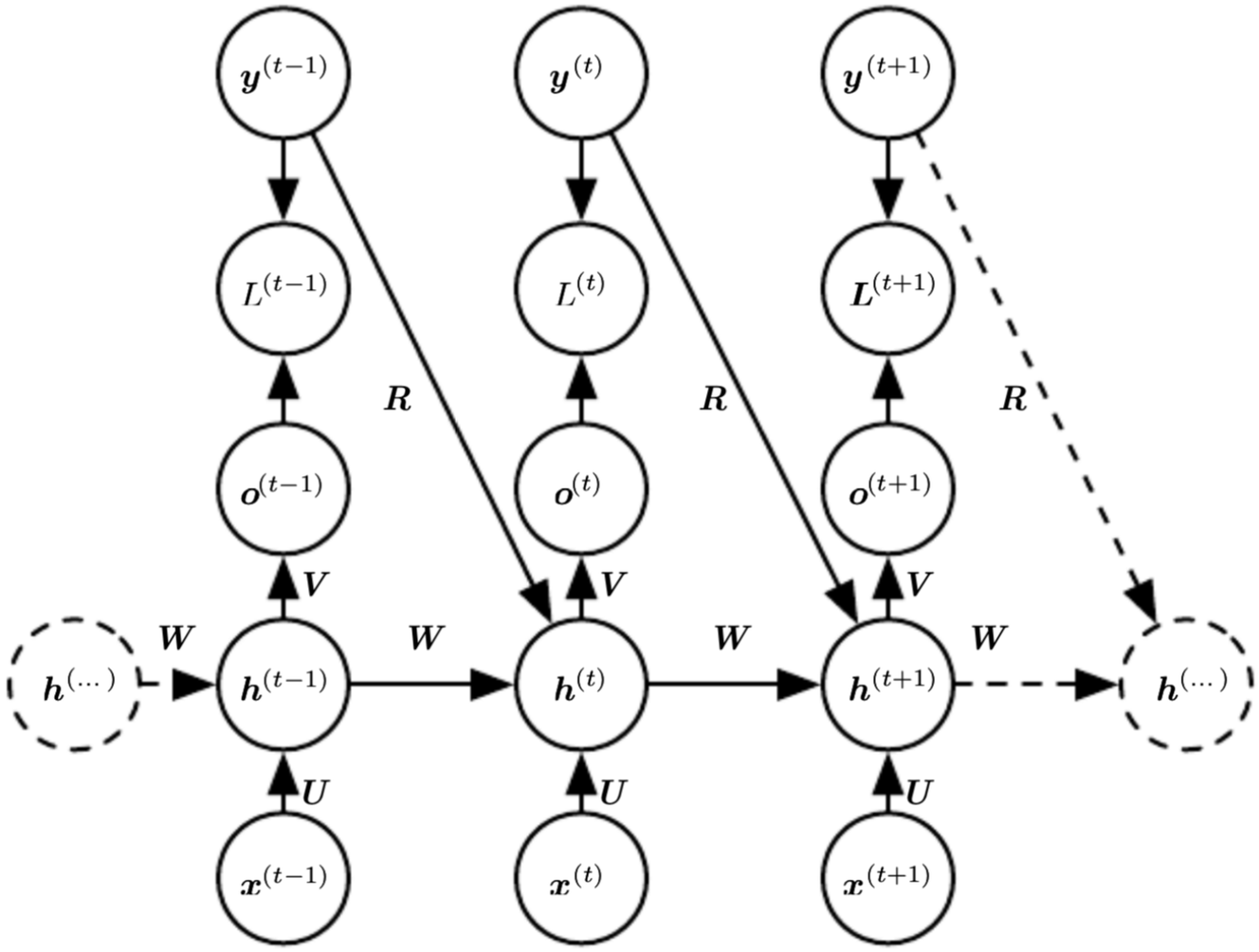


Figure 10.10: A conditional recurrent neural network mapping a variable-length sequence of $\boldsymbol{x}$ values into a distribution over sequences of $\boldsymbol{y}$ values of the same length. Compared to figure 10.3, this RNN contains connections from the previous output to the current state. These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

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Bidirectional RNN

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

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

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Bidirectional RNN

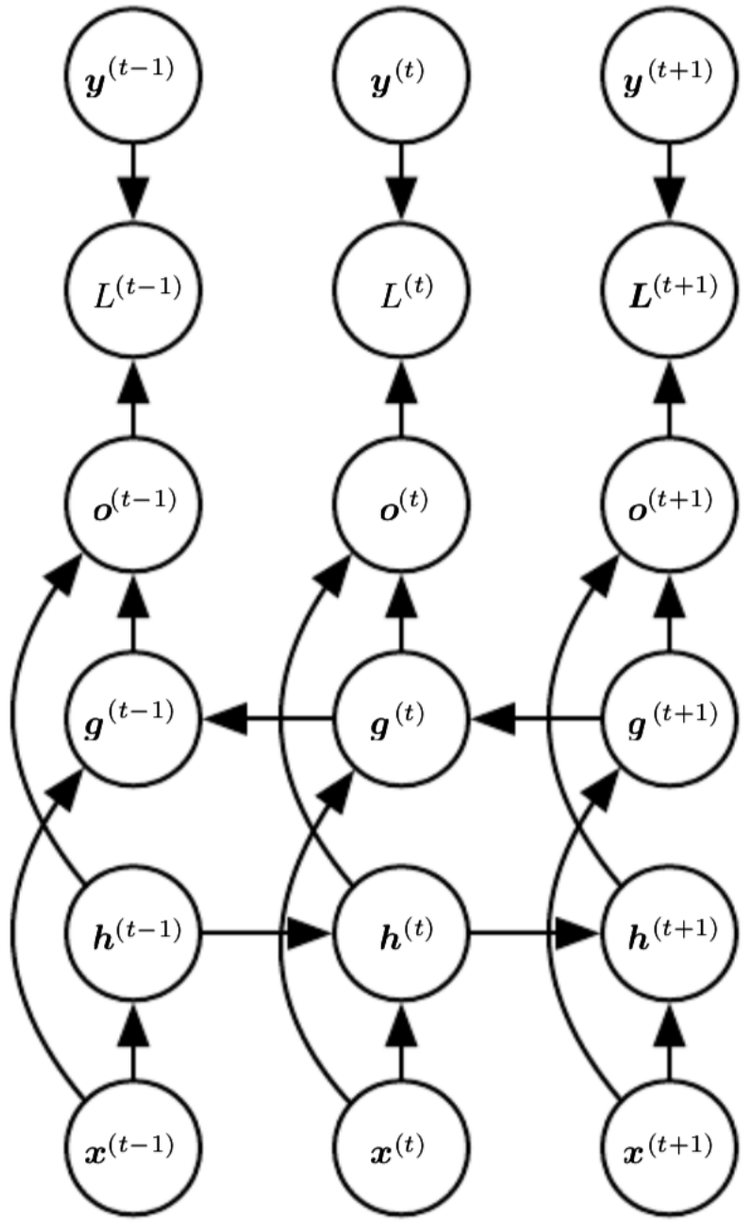


Figure 10.11: Computation of a typical bidirectional recurrent neural network, meant to learn to map input sequences $\boldsymbol{x}$ to target sequences $\boldsymbol{y}$, with loss $L^{(t)}$ at each step t. The $\boldsymbol{h}$ recurrence propagates information forward in time (towards the right) while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

Encoder-Decoder Architectures

The lengths of input and output sequences can vary from each other

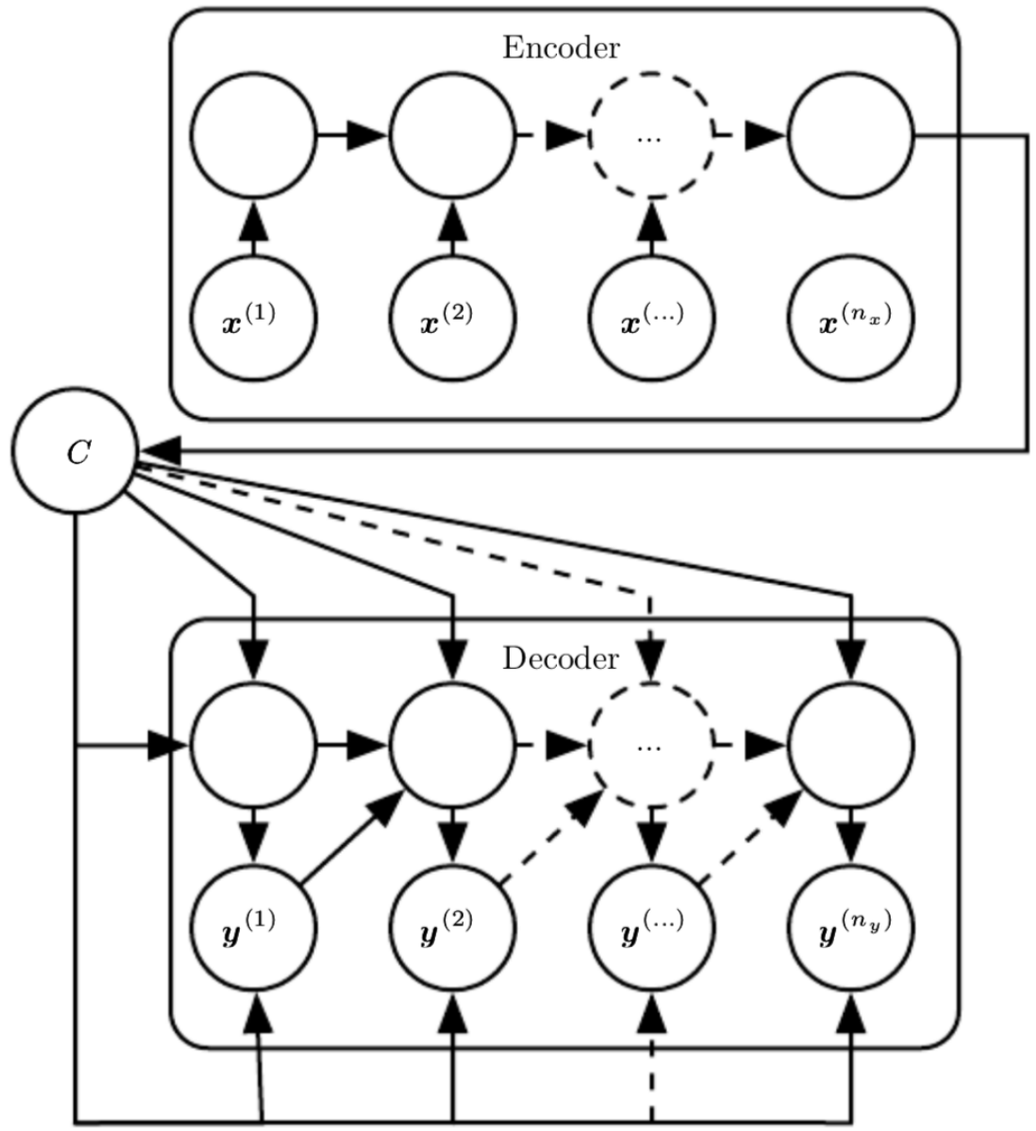


Figure 10.12: Example of an encoder-decoder or sequence-to-sequence RNN architecture, for learning to generate an output sequence $(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n_y)})$ given an input sequence $(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n_x)})$. It is composed of an encoder RNN that reads the input sequence and a decoder RNN that generates the output sequence (or computes the probability of a given output sequence). The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable C which represents a semantic summary of the input sequence and is given as input to the decoder RNN.

Encoder-Decoder Architectures

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

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Deep Recurrent Network (Stacking)

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.

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Deep Recurrent Network (Stacking)

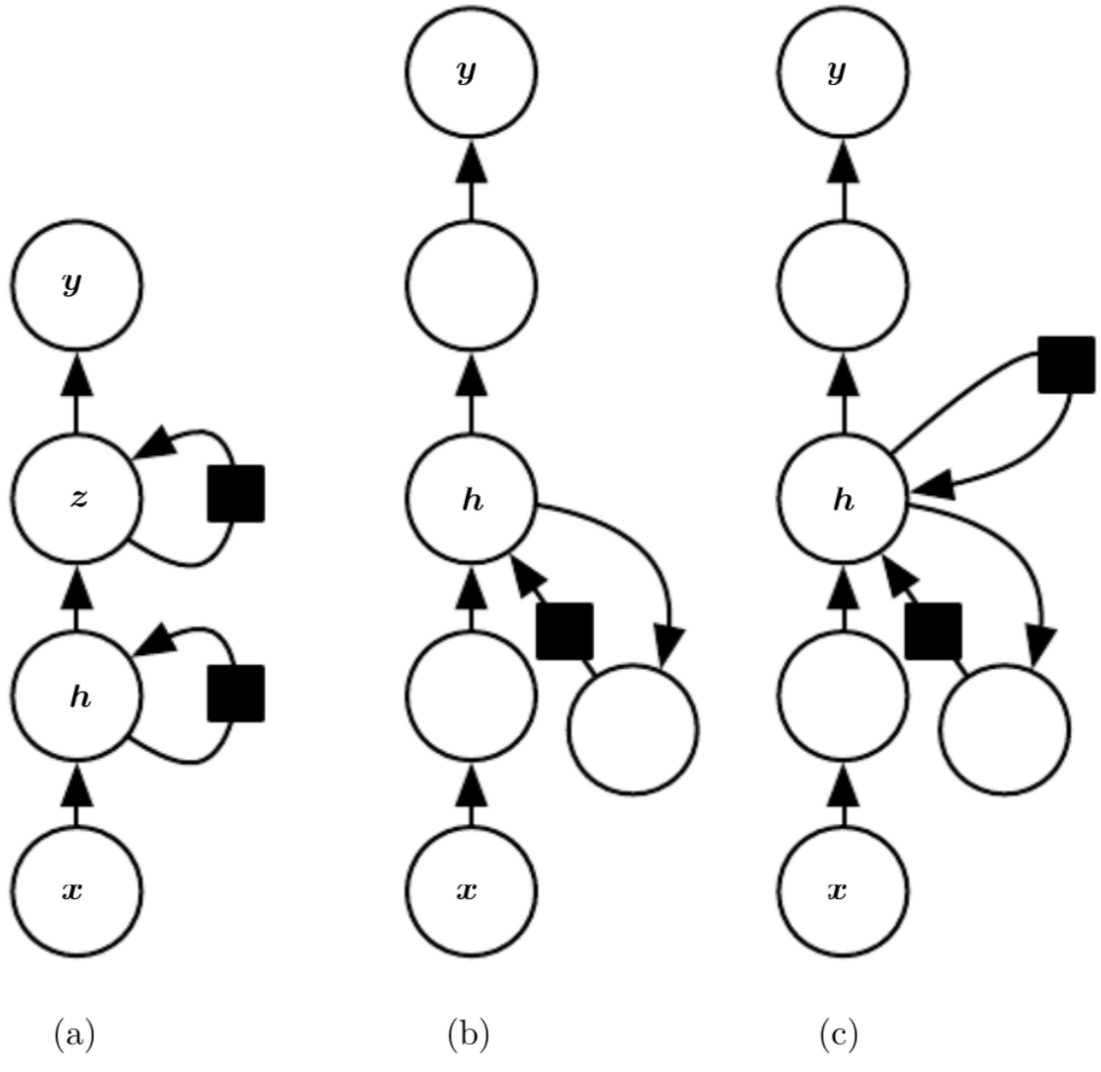


Figure 10.13: A recurrent neural network can be made deep in many ways (Pascanu *et al.*, 2014a). *(a)*The hidden recurrent state can be broken down into groups organized hierarchically. *(b)*Deeper computation (e.g., an MLP) can be introduced in the input-to-hidden, hidden-to-hidden and hidden-to-output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

Recursive Neural Network

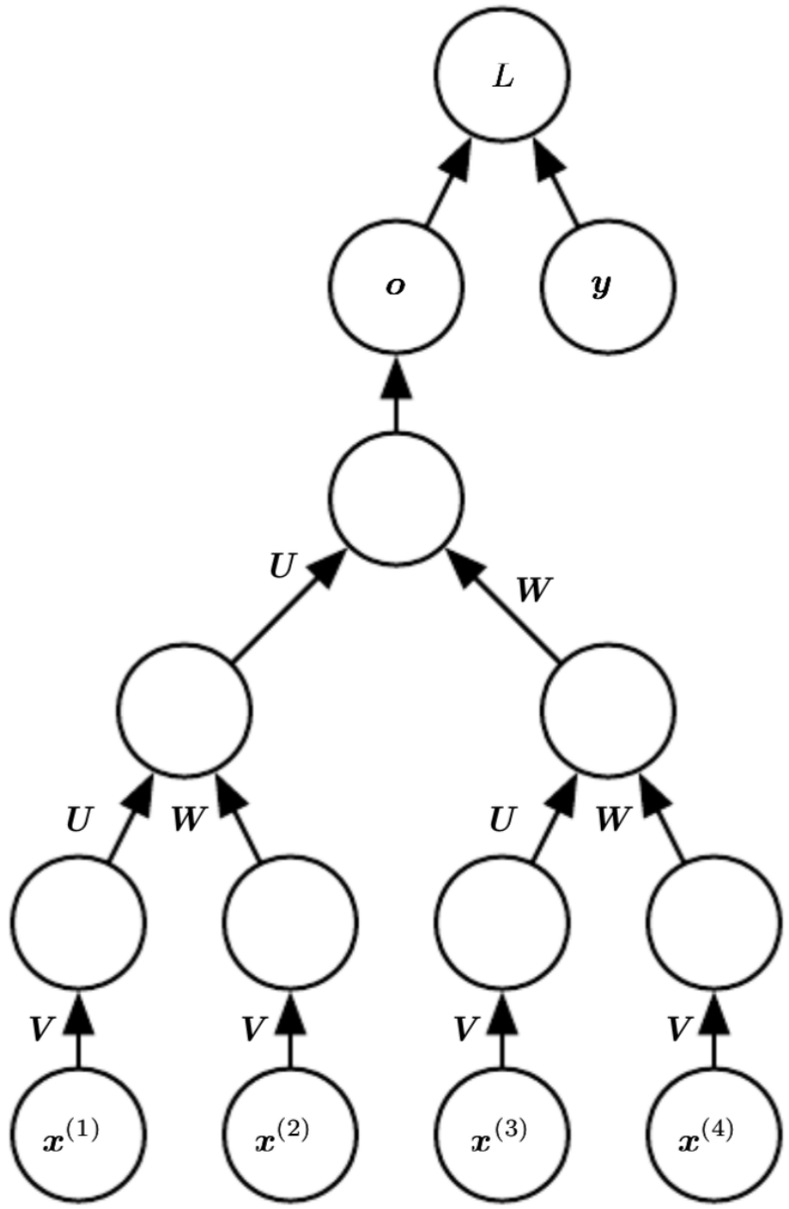


Figure 10.14: A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree. A variable-size sequence $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$ can be mapped to a fixed-size representation (the output $\boldsymbol{o}$),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

Issues in Recurrent Network

The challenge of Long-Term Dependency: Vanishing and exploding gradient problem

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

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

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

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

This means that eigenvalues with magnitude less than one to vanish to zero and eigenvalues with magnitude greater than one toexplode. The above analysis shows the essence of the vanishing and exploding gradient problem for RNNs.

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Issues in Recurrent Network

The challenge of Long-Term Dependency: Vanishing and exploding gradient problem

10.9.1 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).

As we have seen in section 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 $\frac{\tau}{d}$ 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.

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Issues in Recurrent Network

10.9.3 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.

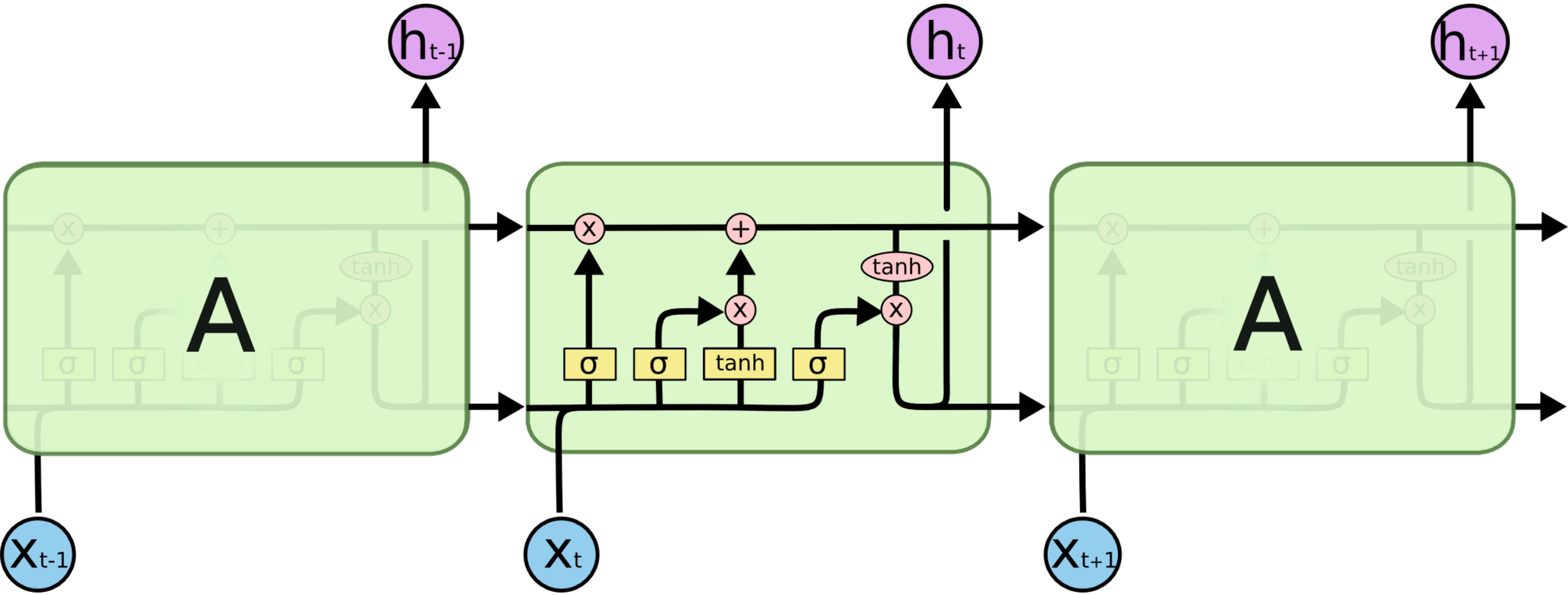
There are different ways in which a group of recurrent units can be forced to operate at different time scales. One option is to make the recurrent units leaky, but to have different groups of units associated with different fixed time scales. This was the proposal in Mozer (1992) and has been successfully used in Pascanu *et al.* (2013). 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.

Reference: Deep Learning Book: Aaron Courville, Ian Goodfellow, and Yoshua Bengio

Long Short Term Memory

Long Short Term Memory

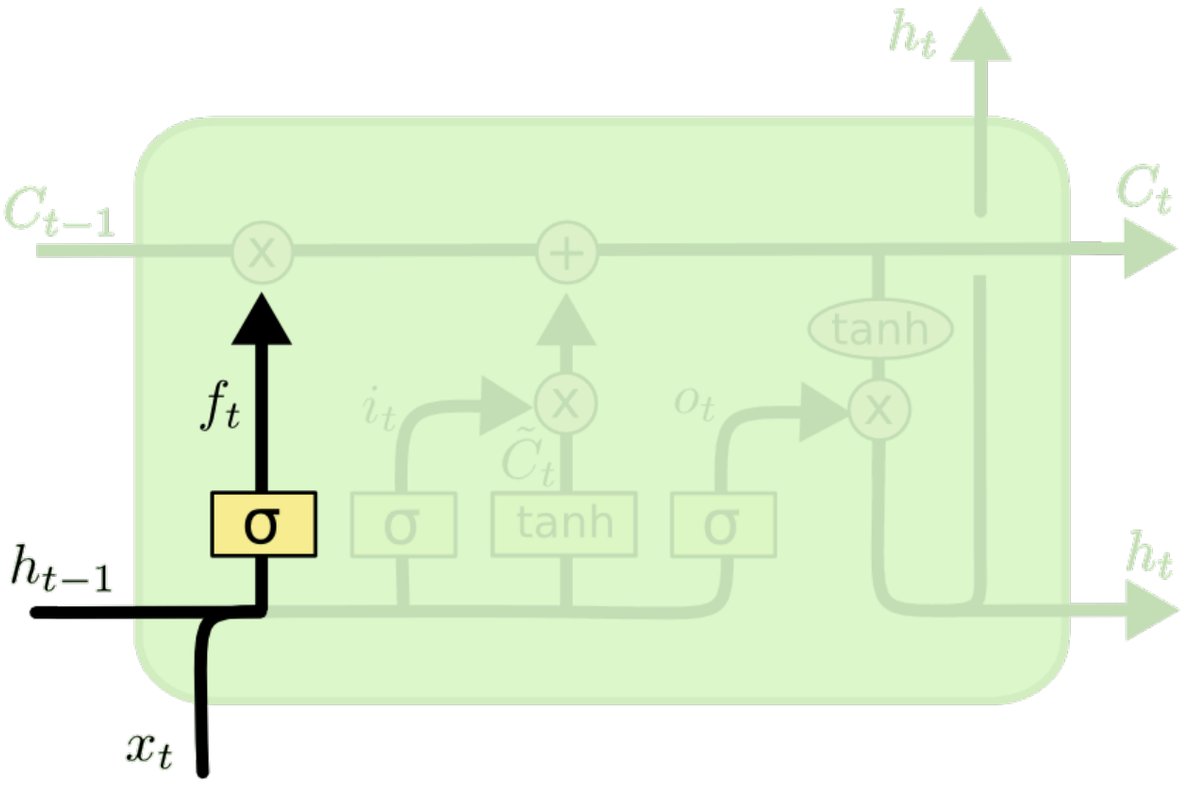
Hochreiter and Schmidhuber, 1997



http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short Term Memory

Hochreiter and Schmidhuber, 1997



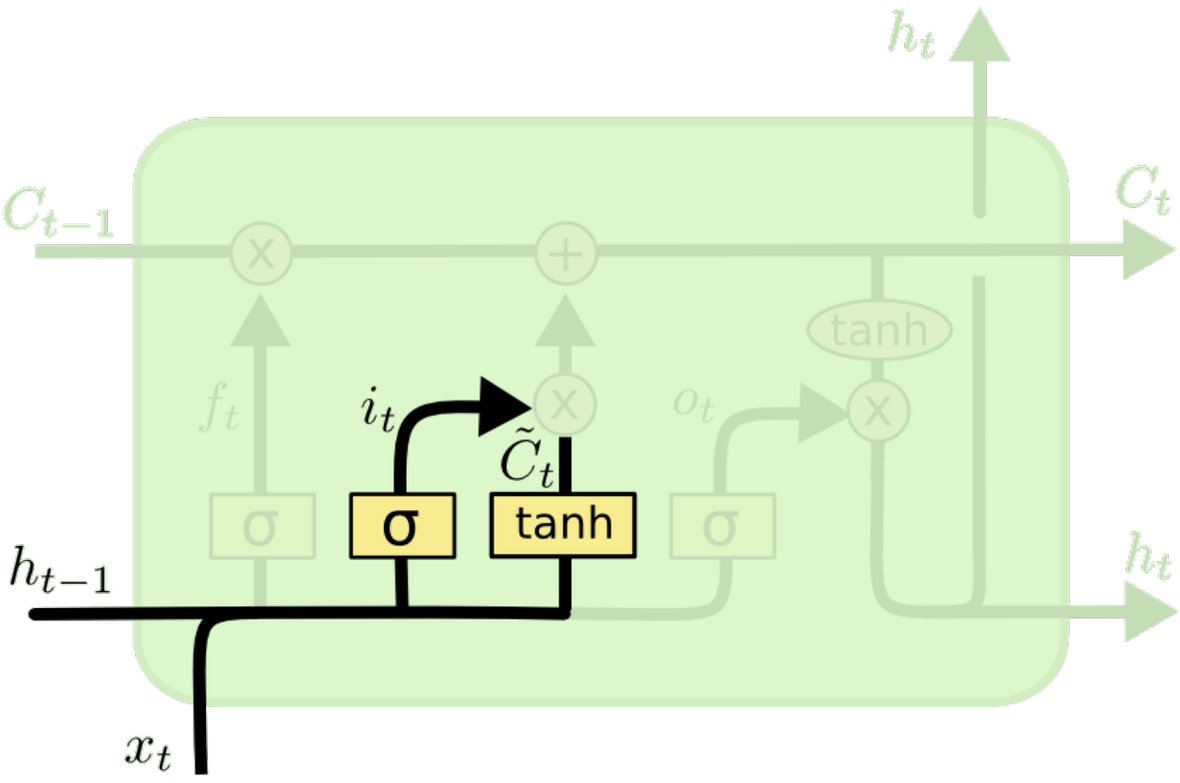
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

The forget gate f that controls how much previous "history" C(t-1) flows into the current "history"

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short Term Memory

Hochreiter and Schmidhuber, 1997



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

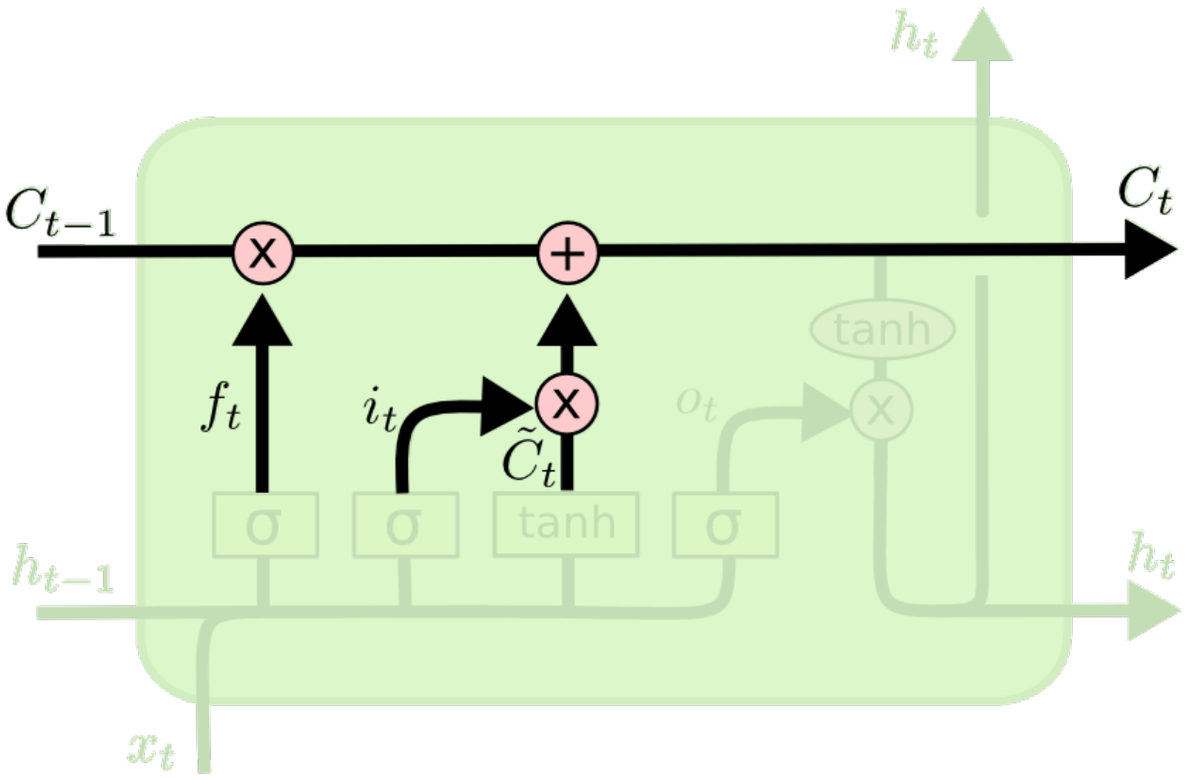
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

The external input gate g that controls how much “new stuff” flows into the new “history”

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short Term Memory

Hochreiter and Schmidhuber, 1997



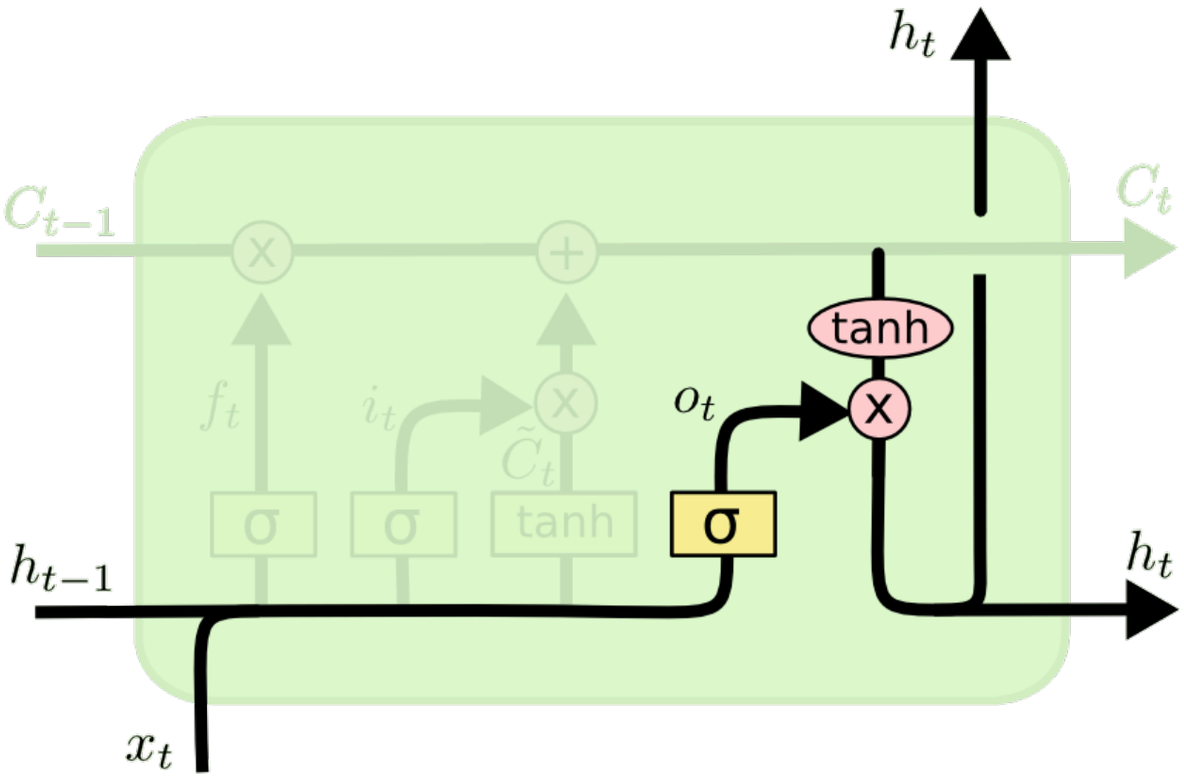
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Decide how to blend history and new stuff

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short Term Memory

Hochreiter and Schmidhuber, 1997



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

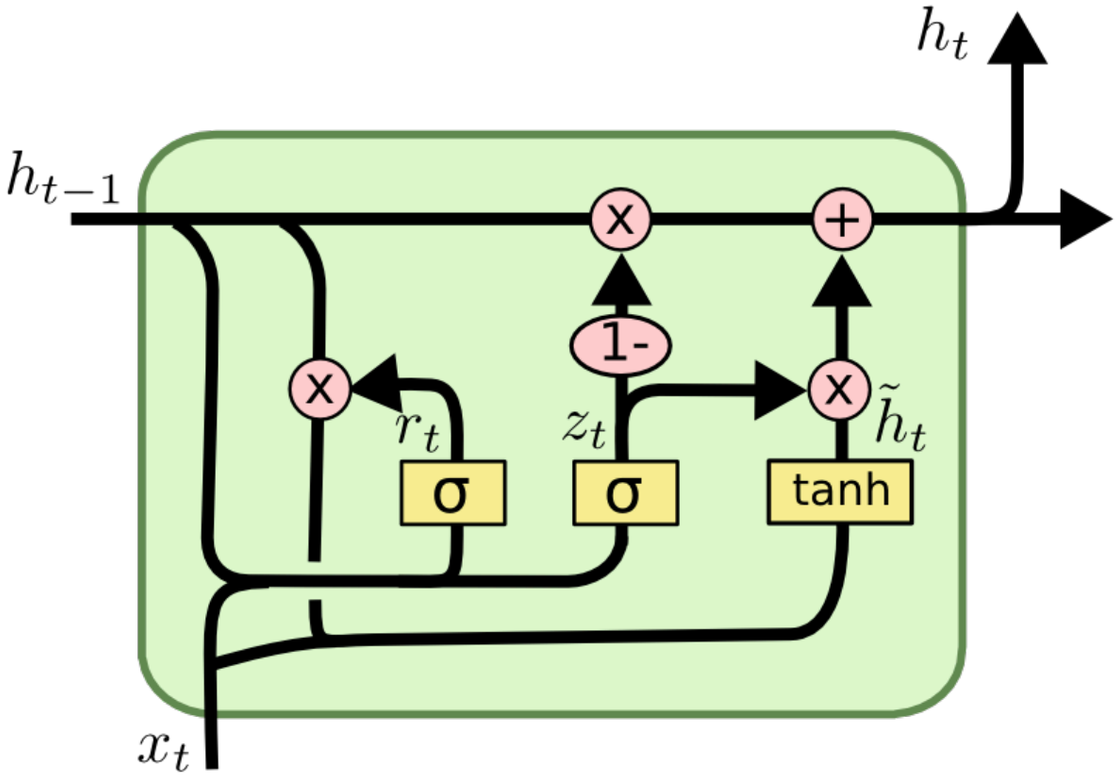
$$h_t = o_t * \tanh\left(C_t\right)$$

The output gate that controls how much current “history” C(t) follows into the current hidden states h(t)

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Gated Recurrent Unit

Hochreiter and Schmidhuber, 1997



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

reset gate

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Sentiment Analysis

-LSTM-

Data preparation

Tokenize & Build Vocal

```
nlp = spacy.load("en")

word_freq = collections.Counter()
max_len = 0
num_rec = 0
print('Count words and build vocab...')
with open('../data/umich-sentiment-train.txt', 'rb') as f:
    for line in f:
        _lab, _sen = line.decode('utf8').strip().split('\t')
        words = [token.lemma_ for token in nlp(_sen) if token.is_alpha] # Stop word제거 안한 상태
        # 제거를 위해 [token.lemma_ for token in doc if token.is_alpha and not token.is_stop]
        if len(words) > max_len:
            max_len = len(words)
        for word in words:
            word_freq[word] += 1
        num_rec += 1

# most_common output -> list
word2idx = {x[0]: i+2 for i, x in enumerate(word_freq.most_common(MAX_VOCAB - 2))}
word2idx ['PAD'] = 0
word2idx['UNK'] = 1

idx2word= {i:v for v, i in word2idx.items()}
vocab_size = len(word2idx)
```

Data preparation

Prepare Data

```
y = []
x = []
origin_txt = []
with open('../data/umich-sentiment-train.txt', 'rb') as f:
    for line in f:
        _label, _sen = line.decode('utf8').strip().split('\t')
        origin_txt.append(_sen)
        y.append(int(_label))
        words = [token.lemma_ for token in nlp(_sen) if token.is_alpha] # Stop word제거 안한 상태
        words = [x for x in words if x != '-PRON-'] # '-PRON-' 제거
        _seq = []
        for word in words:
            if word in word2idx.keys():
                _seq.append(word2idx[word])
            else:
                _seq.append(word2idx['UNK'])
        if len(_seq) < MAX_SENTENCE_LENGTH:
            _seq.extend([0] * ((MAX_SENTENCE_LENGTH) - len(_seq)))
        else:
            _seq = _seq[:MAX_SENTENCE_LENGTH]
        x.append(_seq)

pd.DataFrame(y, columns = ['yn']).reset_index().groupby('yn').count().reset_index()
```

Data Iterator

Split Data

```
tr_idx = np.random.choice(range(len(x)), int(len(x) * .8))
va_idx = [x for x in range(len(x)) if x not in tr_idx]

tr_x = [x[i] for i in tr_idx]
tr_y = [y[i] for i in tr_idx]
va_x = [x[i] for i in va_idx]
va_y = [y[i] for i in va_idx]

batch_size = 16

train_data = mx.io.NDArrayIter(data=[tr_x, tr_y], batch_size=batch_size, shuffle = False)
valid_data = mx.io.NDArrayIter(data=[va_x, va_y], batch_size=batch_size, shuffle = False)
```

Bi-directional LSTM with stacking

Sentence representation via LSTM

```
class Sentence_Representation(nn.Block):
    def __init__(self, EMB_DIM, HIDDEN_DIM, VOCAB_SIZE, dropout = .2, **kwargs):
        super(Sentence_Representation, self).__init__(**kwargs)
        self.VOCAB_SIZE = VOCAB_SIZE
        self.EMB_DIM = EMB_DIM
        self.HIDDEN_DIM = HIDDEN_DIM
        with self.name_scope():
            self.hidden = []
            self.embed = nn.Embedding(VOCAB_SIZE, EMB_DIM)
            self.lstm = rnn.LSTM(HIDDEN_DIM // 2, num_layers= 2 \
                                 , dropout = dropout, input_size = EMB_DIM \
                                 , bidirectional=True)
            self.drop = nn.Dropout(.2)

    def forward(self, x, hidden):
        embeds = self.embed(x) # batch * time step * embedding: NTC
        lstm_out, self.hidden = self.lstm(nd.transpose(embeds, (1, 0, 2)), hidden) #TNC로 변환
        _hid = [nd.transpose(x, (1, 0, 2)) for x in self.hidden]
        # Concatenate depreciated. use concat. input list of tensors
        _hidden = nd.concat(*_hid)
        return lstm_out, self.hidden

    def begin_state(self, *args, **kwargs):
        return self.lstm.begin_state(*args, **kwargs)
```

Bidirectional LSTM

Sentence representation via LSTM

```
class Sentence_Representation(nn.Block): ## Using LSTMCell : Only use the last time step
    def __init__(self, emb_dim, hidden_dim, vocab_size, dropout = .2, **kwargs):
        super(Sentence_Representation, self).__init__(**kwargs)
        self.vocab_size = vocab_size
        self.emb_dim = emb_dim
        self.hidden_dim = hidden_dim
        with self.name_scope():
            self.f_hidden = []
            self.b_hidden = []
            self.embed = nn.Embedding(self.vocab_size, self.emb_dim)
            self.drop = nn.Dropout(.2)
            self.bi_rnn = rnn.BidirectionalCell(
                rnn.LSTMCell(hidden_size = self.hidden_dim // 2),  #mx.rnn.LSTMCell doesnot work
                rnn.LSTMCell(hidden_size = self.hidden_dim // 2)
            )

    def forward(self, x, _f_hidden, _b_hidden):
        embeds = self.embed(x) # batch * time step * embedding
        h, _ = self.bi_rnn.unroll(length = embeds.shape[1] \
                                  , inputs = embeds \
                                  , layout = 'NTC' \
                                  , merge_outputs = True)
        # For understanding
        batch_size, time_step, _ = h.shape
        return x
```

Classifier

```
class SA_Classifier(nn.Block):
    def __init__(self, sen_rep, classifier, batch_size, context, **kwargs):
        super(SA_Classifier, self).__init__(**kwargs)
        self.batch_size = batch_size
        self.context = context
        with self.name_scope():
            self.sen_rep = sen_rep
            self.classifier = classifier

    def forward(self, x):
        hidden = self.sen_rep.begin_state(func = mx.nd.zeros
                                          , batch_size = self.batch_size
                                          , ctx = self.context)
        print('hidden shape = {}'.format([x.shape for x in hidden]))
        #_x, _ = self.sen_rep(x, hidden)
        _, _x = self.sen_rep(x, hidden) # Use the last hidden step
        print('x shape = {}'.format(_x[0].shape))
        x = nd.reshape(x, (-1,))
        print('xaa = {}'.format(_x[1].shape))
        x = self.classifier(x)
        return x
```

Initiatialize classifier

```
sen_rep = Sentence_Representation(emb_dim, hidden_dim, MAX_VOCAB)
sen_rep.collect_params().initialize(mx.init.Xavier(), ctx = context)

classifier = nn.Sequential()
classifier.add(nn.Dense(16, activation = 'relu'))
classifier.add(nn.Dense(8, activation = 'relu'))
classifier.add(nn.Dense(1))

emb_dim = 50 # Emb dim
hidden_dim = 30 # Hidden dim for LSTM
sa = SA_Classifier(sen_rep, classifier,  batch_size, context)
sa.collect_params().initialize(mx.init.Xavier(), ctx = context)
loss = gluon.loss.SigmoidBCELoss()
trainer = gluon.Trainer(sa.collect_params(), 'adam', {'learning_rate': 1e-3})
```

Train network

```
for i, batch in enumerate(train_data):
    _data = batch.data[0].as_in_context(context)
    _label = batch.data[1].as_in_context(context)
    L = 0
    wc = len(_data)
    log_interval_wc += wc
    epoch_wc += wc
    log_interval_sent_num += _data.shape[1]
    epoch_sent_num += _data.shape[1]
    with autograd.record():
        _out = sa(_data)
        L = L + loss(_out, _label).mean().as_in_context(context)
    L.backward()
    trainer.step(_data.shape[0])
```

Some of results

	txt	pred_sa	label
0	Anyway, thats why I love " Brokeback Mountain.	1.0	1
1	Combining the opinion / review from Gary and G...	0.0	0
2	we're gonna like watch Mission Impossible or H...	1.0	1
3	The Da Vinci Code sucked big time.	0.0	0
4	I am going to start reading the Harry Potter s...	1.0	1
5	I am going to start reading the Harry Potter s...	1.0	1
6	Always knows what I want, not guy crazy, hates...	0.0	0
7	I loved this mission impossible scenario.	0.0	1
8	I liked the first " Mission Impossible.	1.0	1
9	Brokeback Mountain is fucking horrible..	0.0	0

Addition

-Seq2Seq Model-

```
class calculator(gluon.Block):
    def __init__(self, n_hidden, in_seq_len, out_seq_len, vocab_size, **kwargs):
        super(calculator, self).__init__(**kwargs)
        self.in_seq_len = in_seq_len
        self.out_seq_len = out_seq_len
        self.n_hidden = n_hidden
        self.vocab_size = vocab_size

        with self.name_scope():
            self.encoder = rnn.LSTMCell(hidden_size = n_hidden)
            self.decoder = rnn.LSTMCell(hidden_size = n_hidden)
            self.batchnorm = nn.BatchNorm(axis = 2)
            self.dense = nn.Dense(self.vocab_size, flatten = False)

    def forward(self, inputs, outputs):
        # Since we don't use intermediate states for 'though vector', we don't need to unroll it.
        # In the later examples, we will use LSTM class rather than LSTMCell class.
        enout, (next_h, next_c) = self.encoder.unroll(inputs = inputs
                                                      , length = self.in_seq_len
                                                      , merge_outputs = True)
```

Hidden states at the last time step

```
        for i in range(self.out_seq_len):
            deout, (next_h, next_c) = self.decoder(outputs[:, i, :], [next_h, next_c],)
            if i == 0:
                deouts = deout
            else:
                deouts = nd.concat(deouts, deout, dim = 1)
            #print('i= {}, deouts= {}'.format(i, deouts.shape))

        deouts = nd.reshape(deouts, (-1, self.out_seq_len, self.n_hidden))
        deouts = self.batchnorm(deouts)
        deouts_fc = self.dense(deouts)
        return deouts_fc
```

Decode for output sequence

Fully connected networks that leads to loss

Data loader and Optimizer

```
tr_set = gluon.data.ArrayDataset(X_train, Y_train, Z_train)
tr_data_iterator = gluon.data.DataLoader(tr_set, batch_size=256, shuffle=True)

te_set =gluon.data.ArrayDataset(X_validation, Y_validation, Z_validation)
te_data_iterator = gluon.data.DataLoader(te_set, batch_size=256, shuffle=True)

ctx = mx.gpu()
model = calculator(300, 9, 6, 14)
model.collect_params().initialize(mx.init.Xavier(), ctx = ctx)

trainer = gluon.Trainer(model.collect_params(), 'rmsprop')
loss = gluon.loss.SoftmaxCrossEntropyLoss(axis = 2, sparse_label = False)
```

Training

Training part

Print test results

```
for e in range(epochs):
    train_loss = []
    for i, (x_data, y_data, z_data) in enumerate(tr_data_iterator):
        x_data = x_data.as_in_context(ctx).astype('float32')
        y_data = y_data.as_in_context(ctx).astype('float32')
        z_data = z_data.as_in_context(ctx).astype('float32')

        with autograd.record():
            z_output = model(x_data, y_data)
            loss_ = loss(z_output, z_data)
        loss_.backward()
        trainer.step(x_data.shape[0])
        curr_loss = nd.mean(loss_).asscalar()
        train_loss.append(curr_loss)

    if e % 10 == 0:
        q, y = gen_n_test(10)
        for i in range(10):
            with autograd.predict_mode():
                p = model.calculation(q[i], char_indices, indices_char).strip()
                iscorr = 1 if p == y[i] else 0
                if iscorr == 1:
                    print(colors.ok + '☑' + colors.close, end=' ')
                else:
                    print(colors.fail + '☒' + colors.close, end=' ')
                print("{} = {}({}) 1/0 {}".format(q[i], p, y[i], str(iscorr) ))
    #caculate test loss
    test_loss = calculate_loss(model, te_data_iterator, loss_obj = loss, ctx=ctx)
```

Some of results

At the beginning

```
⊠ 93+0 = 1000(93) 1/0 0
⊠ 0+43 = 449(43) 1/0 0
⊠ 864+752 = 1009(1616) 1/0 0
⊠ 175+82 = 1009(257) 1/0 0
⊠ 761+8 = 1039(769) 1/0 0
⊠ 90+421 = 1009(511) 1/0 0
⊠ 446+788 = 1039(1234) 1/0 0
⊠ 62+36 = 630(98) 1/0 0
⊠ 4+4 = 444(8) 1/0 0
⊠ 617+6 = 779(623) 1/0 0
Epoch 0. Train Loss: 1.1954819, Test Loss : 1.1388888
Epoch 1. Train Loss: 1.1225086, Test Loss : 1.1047744
Epoch 2. Train Loss: 1.0937438, Test Loss : 1.0658474
Epoch 3. Train Loss: 1.0264179, Test Loss : 0.97517955
```

Some of results

After 100 epochs

```
☑ 8+8 = 16(16) 1/0 1
☑ 38+648 = 686(686) 1/0 1
☑ 3+55 = 58(58) 1/0 1
☑ 98+598 = 696(696) 1/0 1
☑ 29+25 = 54(54) 1/0 1
☑ 75+91 = 166(166) 1/0 1
☑ 32+9 = 41(41) 1/0 1
☑ 1+570 = 571(571) 1/0 1
☑ 247+80 = 327(327) 1/0 1
☑ 897+52 = 949(949) 1/0 1
Epoch 100. Train Loss: 0.0024368812, Test Loss : 0.009117115
Epoch 101. Train Loss: 0.0019641193, Test Loss : 0.0042197886
```

Reference (not limited to)

https://www.weibo.com/ttarticle/p/show?id=2309403960559677656612

Deep Learning book, Chapter 10 Sequence Modeling: Recurrent and Recursive Nets

https://isaacchanghau.github.io/post/lstm-gru-formula/