Chapter 10

Sequence Modeling: Recurrent Neural Networks

Recurrent Neural Networks (RNN)

- A family of neural networks specialized for processing sequential data

$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(\tau)}$$

with hidden units $\boldsymbol{h}^{(t)}$ forming a dynamical system

$$\boldsymbol{h}^{(t)} = f_{\boldsymbol{\theta}}(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)})$$

- This recurrent relation, when unfolded, leads to

$$\begin{aligned}
\boldsymbol{h}^{(t)} &= f_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\boldsymbol{h}^{(t-2)}, \boldsymbol{x}^{(t-1)}), \boldsymbol{x}^{(t)}) \\
&= f_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\dots f_{\boldsymbol{\theta}}(\boldsymbol{h}^{(0)}, \boldsymbol{x}^{(1)}), \boldsymbol{x}^{(2)}), \dots, \boldsymbol{x}^{(t-1)}), \boldsymbol{x}^{(t)}) \\
&= g^{(t)}(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots, \boldsymbol{x}^{(t)})
\end{aligned}$$

- $\boldsymbol{h}^{(t)}$ summarizes the past inputs up to t with a fixed length vector

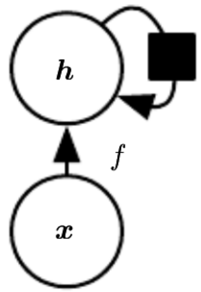
- The RNN is to learn a single shared model $f_{\boldsymbol{\theta}}$ instead of separate models $g^{(t)}$ at different time steps
- This enables generalization to any sequence length, and requires far fewer parameters
- However, it also limits the model capacity; for example, the predicted relationship between the previous time step and a current time step is independent of t
- Note that it is possible for the dynamical system not to take any input

$$\boldsymbol{h}^{(t)} = f_{\boldsymbol{\theta}}(\boldsymbol{h}^{(t-1)}),$$

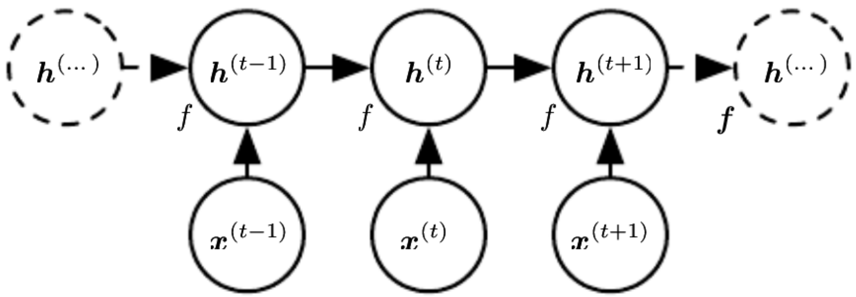
  for example, in cases where the RNN serves as a generative model in unsupervised settings
- Theoretically, when wired properly, the RNN is able to simulate procedures achievable by a Turing machine

Circuit Diagrams and Graph Unrolling

- **Circuit diagrams** – a succinct description of computations with nodes representing components that might exit in physical implementations

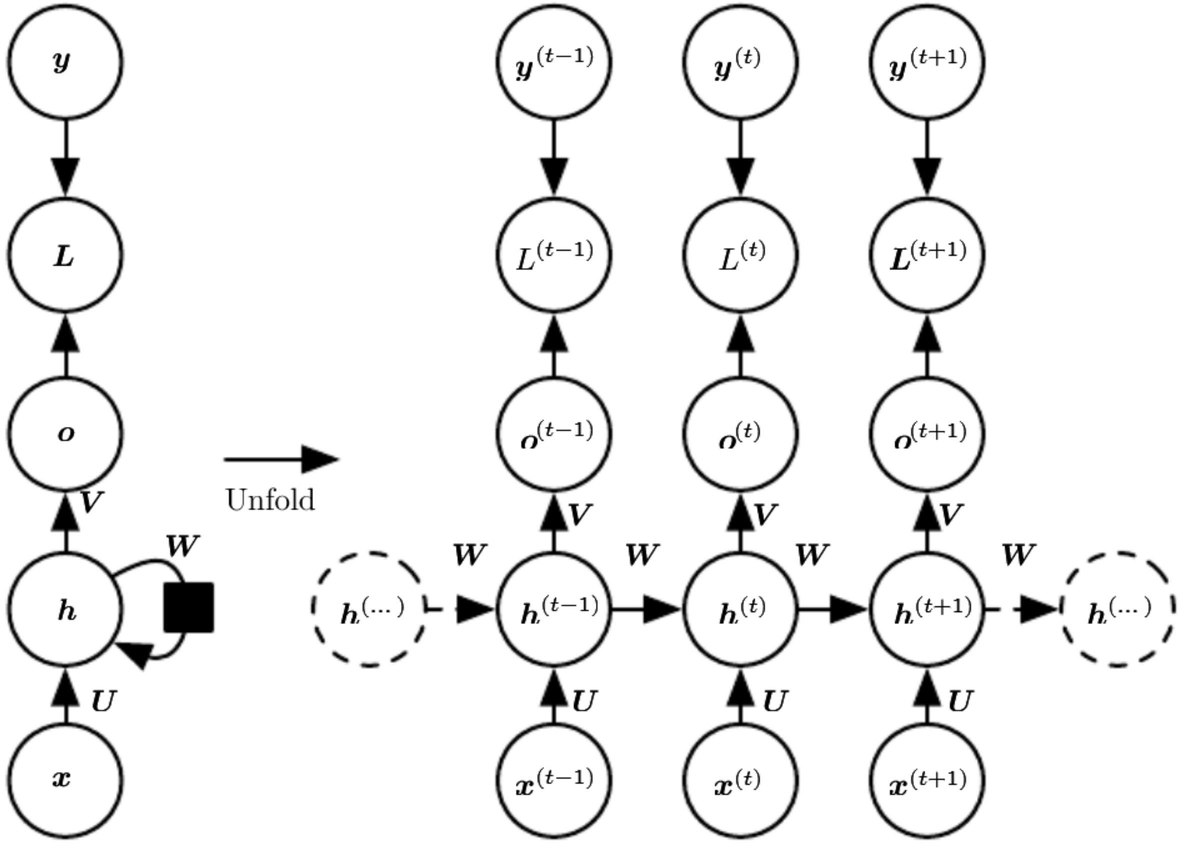


- **Unfolded computational graphs** – an explicit description of computations with nodes indicating variables at each time step



Design Pattern I

- An output at each time step with recurrent hidden unit connections



- Essentially, it implements a sequence-to-sequence mapping

$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(\tau)} \rightarrow \boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(\tau)}$$

- Forward propagation (with initial $\boldsymbol{h}^{(0)}$)

$$\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}, \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}), \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)},
\end{aligned}$$

  where $\boldsymbol{U}, \boldsymbol{W}, \boldsymbol{V}$ correspond respectively to connections for

  – Input-to-hidden units ($\boldsymbol{U}$)

  – Hidden-to-hidden units ($\boldsymbol{W}$)

  – Hidden-to-output units ($\boldsymbol{V}$)

  and $\boldsymbol{b}, \boldsymbol{c}$ are biase

- Why use tanh instead of sigmoid for activation?

- When the target $\boldsymbol{y}^{(t)}$ is a discrete (multinoulli) variable, a softmax is applied to $\boldsymbol{o}^{(t)}$ to obtain

$$\hat{\boldsymbol{y}}^{(t)} = \mathsf{softmax}(\boldsymbol{o}^{(t)}),$$

- This $\hat{\boldsymbol{y}}^{(t)}$ serves as the model prediction for the empirical conditional probability of $\boldsymbol{y}^{(t)}$ given $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$

$$p_{\mathsf{model}}(\boldsymbol{y}^{(t)}|\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}) = \prod_i \left(\hat{y}_i^{(t)}\right)^{\mathbf{1}(y_i^{(t)}=1)}$$

- During training, the total loss is the sum of losses over all time steps

$$L(\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}\}, \{\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(t)}\}) = \sum_t L^{(t)}$$

where the loss $L^{(t)}$ at time step t is given by

$$L^{(t)} = -\log p_{\mathsf{model}}(\boldsymbol{y}^{(t)}|\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)})$$

- This network architecture corresponds to a conditional distribution $p_{\text{model}}(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(\tau)} | \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(\tau)})$ that factorizes as

$$\begin{aligned} & p_{\text{model}}(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(\tau)} | \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(\tau)}) \\ = & \prod_t p_{\text{model}}(\boldsymbol{y}^{(t)} | \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}) \\ = & \prod_t p_{\text{model}}(\boldsymbol{y}^{(t)} | \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(\tau)}) \end{aligned}$$

- This suggests that $\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(\tau)}$ are modeled to be conditionally independent given $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(\tau)}$

Back-Propagation Through Time (BPTT)

- BPTT is merely back-propagation applied to unrolled graphs
- To obtain the gradient on parameter nodes, the gradient on their immediate child (downstream) nodes have to be evaluated first
- As an example, to compute $\nabla_{\boldsymbol{W}} L$, we observe that
  - The immediate child nodes of $\boldsymbol{W}$ are all $\boldsymbol{h}^{(t)}$'s, and
  - The chain rule for tensors[a] can be applied to arrive at

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

---

[a]

$$\boldsymbol{X}_{m \times n} \xrightarrow{g(\boldsymbol{X})} \boldsymbol{Y}_{s \times k} \xrightarrow{f(\boldsymbol{Y})} z_{1 \times 1}$$

$$\nabla_{\boldsymbol{X}} z = \sum_j (\frac{\partial z}{\partial Y_j}) \nabla_{\boldsymbol{X}} Y_j,$$

- To complete the evaluation, we need to know further $\nabla_{\boldsymbol{h}^{(t)}} L$, which can be evaluated using the same chain rule as

$$\begin{aligned}\nabla_{\boldsymbol{h}^{(t)}} L &= \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^T (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^T (\nabla_{\boldsymbol{o}^{(t)}} L) \\ &= \boldsymbol{W}^T \boldsymbol{H}^{(t+1)} (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \boldsymbol{V}^T (\nabla_{\boldsymbol{o}^{(t)}} L)\end{aligned}$$

where

$$\begin{aligned}\boldsymbol{H}^{(t+1)} &= \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{a}^{(t+1)}}\right)^T \\ &= \begin{bmatrix} 1-(h_1^{(t+1)})^2 & 0 & \dots & 0 \\ 0 & 1-(h_2^{(t+1)})^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1-(h_n^{(t+1)})^2 \end{bmatrix}\end{aligned}$$

$$\nabla_{\boldsymbol{o}^{(t)}} L = \hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)}$$

and

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^T (\nabla_{\boldsymbol{o}^{(\tau)}} L) = \boldsymbol{V}^T (\hat{\boldsymbol{y}}^{(\tau)} - \boldsymbol{y}^{(\tau)})$$

- In matrix form, $\nabla_{\boldsymbol{W}} L$ is given as

$$\nabla_{\boldsymbol{W}} L = \sum_t \boldsymbol{H}^{(t)} (\nabla_{\boldsymbol{h}^{(t)}} L) \boldsymbol{h}^{(t-1)T}$$

- The gradient on the remaining parameters can be obtained similarly

$$\nabla_{\boldsymbol{U}} L = \sum_t \boldsymbol{H}^{(t)} (\nabla_{\boldsymbol{h}^{(t)}} L) \boldsymbol{x}^{(t)T}$$

$$\nabla_{\boldsymbol{V}} L = \sum_t (\nabla_{\boldsymbol{o}^{(t)}} L) \boldsymbol{h}^{(t)T}$$

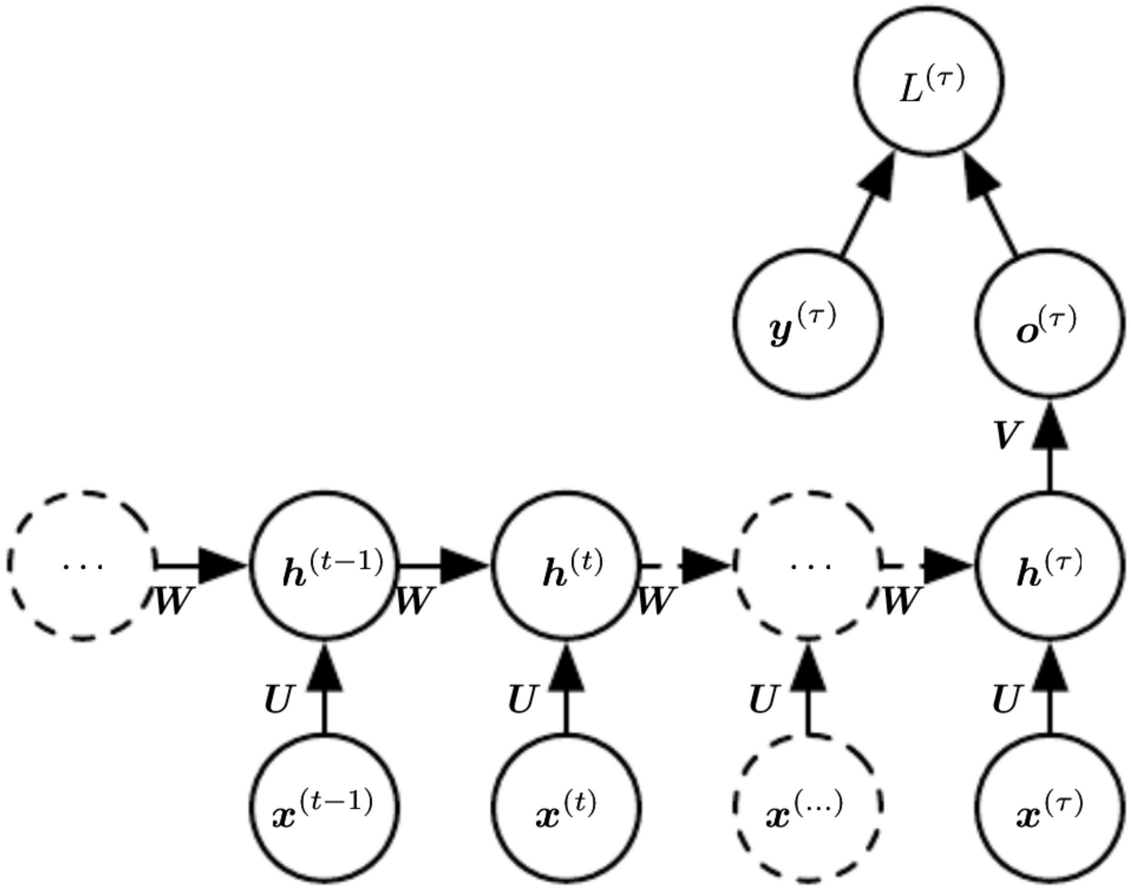
$$\nabla_{\boldsymbol{b}} L = \sum_t \boldsymbol{H}^{(t)} (\nabla_{\boldsymbol{h}^{(t)}} L)$$

$$\nabla_{\boldsymbol{c}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L$$

- The runtime and memory cost for BPTT are both $\mathcal{O}(\tau)$

Design Pattern II

- Networks with hidden unit connections that produce a single output for an entire sequence



Vanishing and Exploding Gradients

- Gradients propagate over many stages tend to either vanish or explode
- For example, the gradient $\nabla_{\boldsymbol{h}^{(t)}} L$ in pattern II is seen to follow

$$\begin{aligned}
\nabla_{\boldsymbol{h}^{(t)}} L &= \boldsymbol{W}^T \boldsymbol{H}^{(t+1)} (\nabla_{\boldsymbol{h}^{(t+1)}} L) \\
&= (\underbrace{\boldsymbol{W}^T \boldsymbol{H}^{(t+1)}}_{\boldsymbol{M}})^{(\tau-t)} (\nabla_{\boldsymbol{h}^{(\tau)}} L) \\
&= \boldsymbol{Q} \boldsymbol{\Lambda}^{(\tau-t)} \boldsymbol{Q}^T (\nabla_{\boldsymbol{h}^{(\tau)}} L)
\end{aligned}$$

  where $\boldsymbol{M}$ is assumed to be the same at every time step (as is the case without activation) and have an eigendecomposition $\boldsymbol{M} = \boldsymbol{Q} \boldsymbol{\Lambda} \boldsymbol{Q}^T$

- The term $\boldsymbol{\Lambda}^{(\tau-t)}$ causes the eigenvalues with magnitude smaller than one to decay to zero and eigenvalues with magnitude greater than one to explode (when $t \ll \tau$)

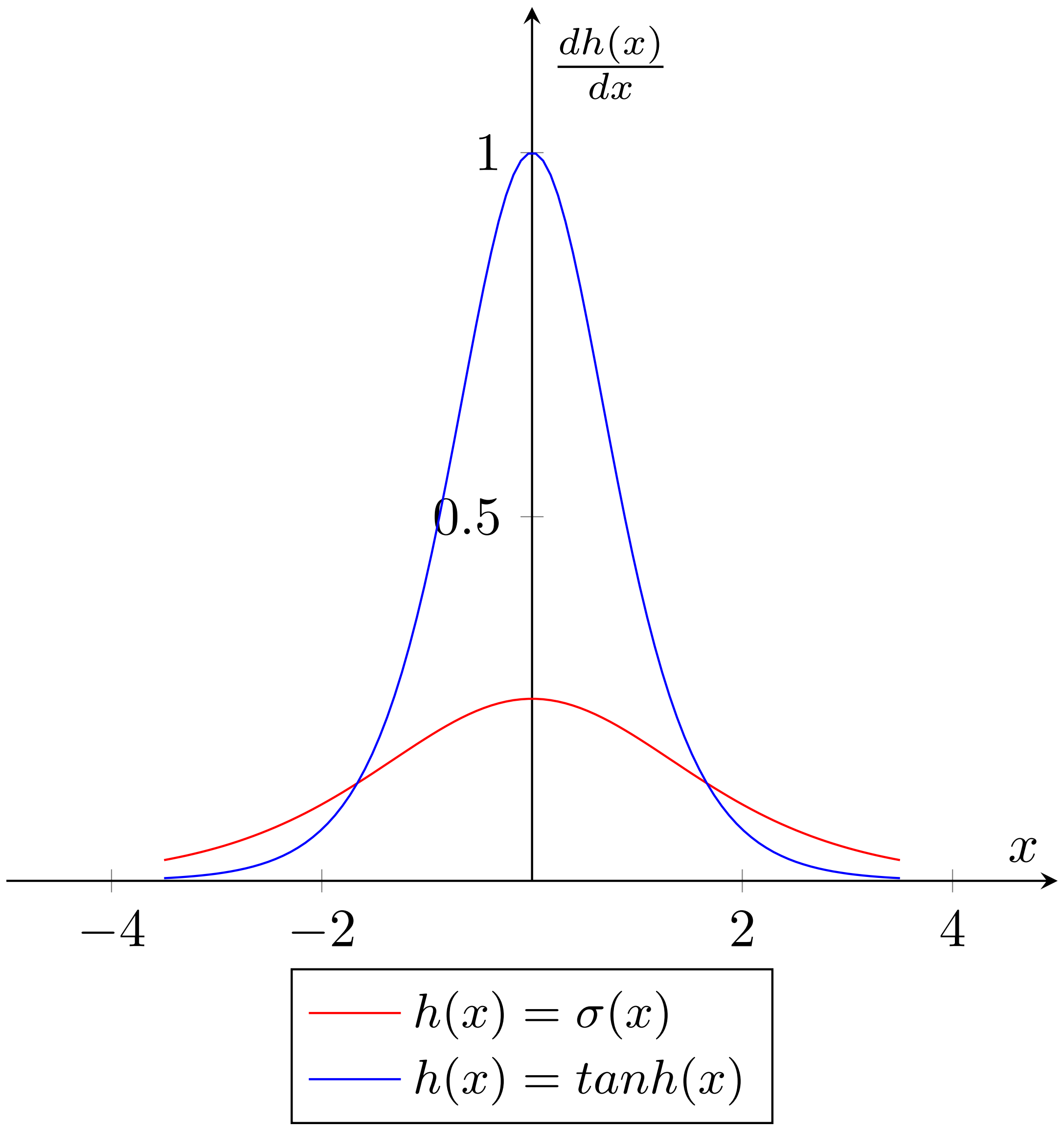
- When gradients vanish quickly, it becomes difficult to learn long-term dependencies; as an example, recall that

$$\nabla_{\boldsymbol{W}} L = \sum_t \boldsymbol{H}^{(t)} (\nabla_{\boldsymbol{h}^{(t)}} L) \boldsymbol{h}^{(t-1)T}$$
$$\nabla_{\boldsymbol{U}} L = \sum_t \boldsymbol{H}^{(t)} (\nabla_{\boldsymbol{h}^{(t)}} L) \boldsymbol{x}^{(t)T}$$

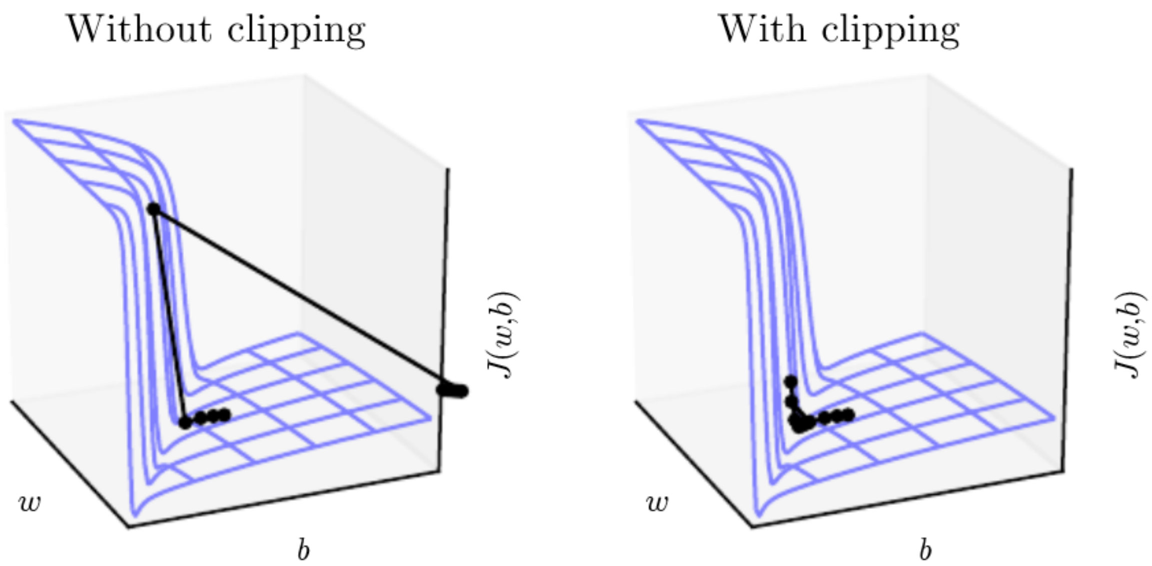
  where the distant past inputs $\boldsymbol{x}^{(t)}$ and hidden units $\boldsymbol{h}^{(t)}$ with $t \ll \tau$ are seen to contribute little to learning $\boldsymbol{W}$ and $\boldsymbol{U}$

- From the perspective of forward propagation, an input $\boldsymbol{x}^{(t)},\ t \ll \tau$ may have been attenuated significantly before it reaches the output $\boldsymbol{o}^{(\tau)}$ due to the repeated multiplication with $\boldsymbol{H}\boldsymbol{W}$

- *Sidetracking:* The reason for choosing tanh as activation has to do with $\boldsymbol{H}$, of which the diagonal entries are derivatives of $h(x)$; gradients may vanish much quicker if sigmoid is used


$\frac{dh(x)}{dx}$
1
0.5
x
−4
−2
2
4
$h(x) = \sigma(x)$
$h(x) = tanh(x)$

- When gradients explode, the gradient-based training could throw the parameters far into a region where the objective becomes larger



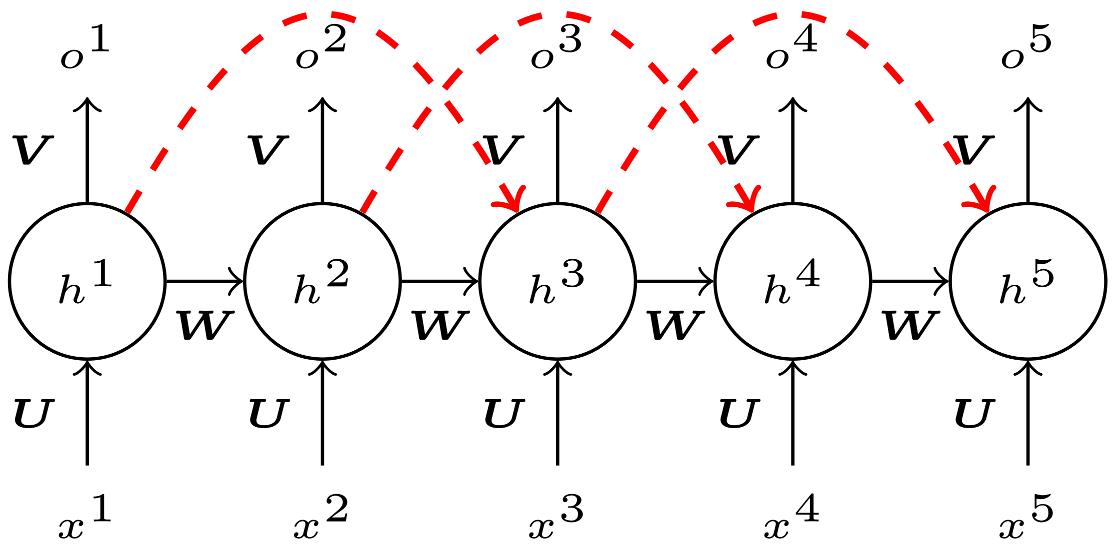
- One simple solution is to clip the gradient before the parameter update when a threshold on its norm is exceeded; that is, if $\|\boldsymbol{g}\| > v$

$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}v}{\|\boldsymbol{g}\|}$$

- Note that while the activation function tanh helps to ensure forward propagation has bounded dynamics (i.e. hidden units will not explode), it is possible for backward propagation to remain unbounded (recall that $\boldsymbol{H}$ can be $\boldsymbol{I}$)

Learning Long-Term Dependencies

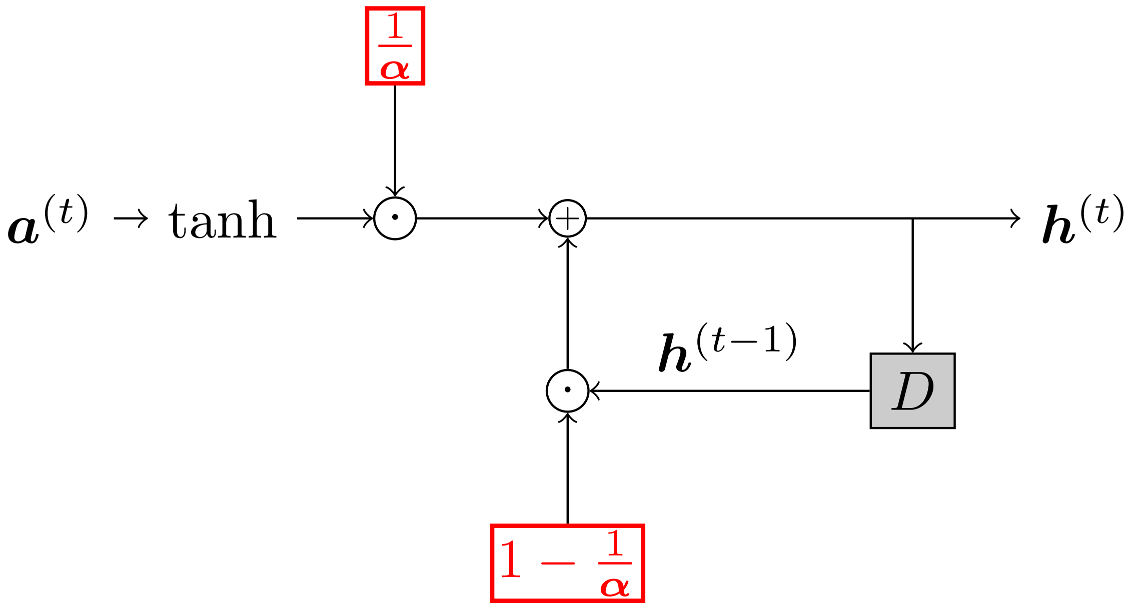
- **Echo state networks**
  - Set recurrent weights $\boldsymbol{W}$ to ensure spectral radius of Jacobian ≈ 1
  - Learn output weights $\boldsymbol{V}$ only
- **Skip connections** – connections from variables in the distant past



Gradients diminish exponentially as a function of τ/d

- **Leaky units** – introducing an integrator in the hidden unit

$$\boldsymbol{h}^{(t)} = (1 - \frac{1}{\boldsymbol{\alpha}}) \odot \boldsymbol{h}^{(t-1)} + \frac{1}{\boldsymbol{\alpha}} \odot \tanh(\boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}),\ 1 \leq \alpha_i < \infty$$



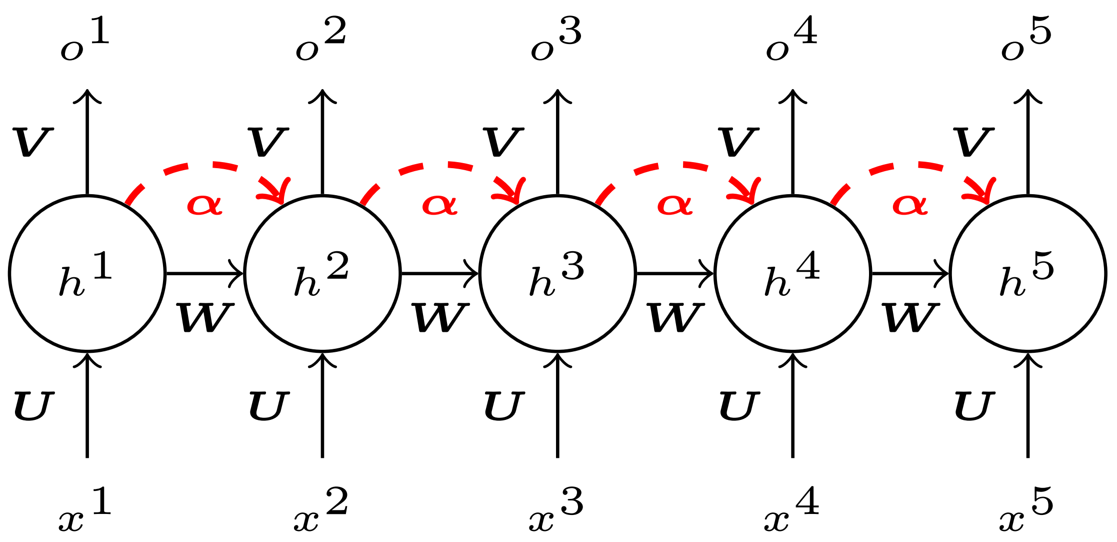
- The new content $\boldsymbol{a}^{(t)}$ is given by

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

- The conventional RNN does not have this extra inner loop

$$\boldsymbol{a}^{(t)} \to \tanh \longrightarrow \boldsymbol{h}^{(t)}$$

- $\boldsymbol{\alpha}$ determines the time scale of integration ($\alpha_i \uparrow$, scale$\uparrow$)
  - $\alpha_i = 1$: Ordinary RNN
  - $\alpha_i > 1$: $\boldsymbol{x}^{(t)}$'s flow longer; gradients propagate more easily

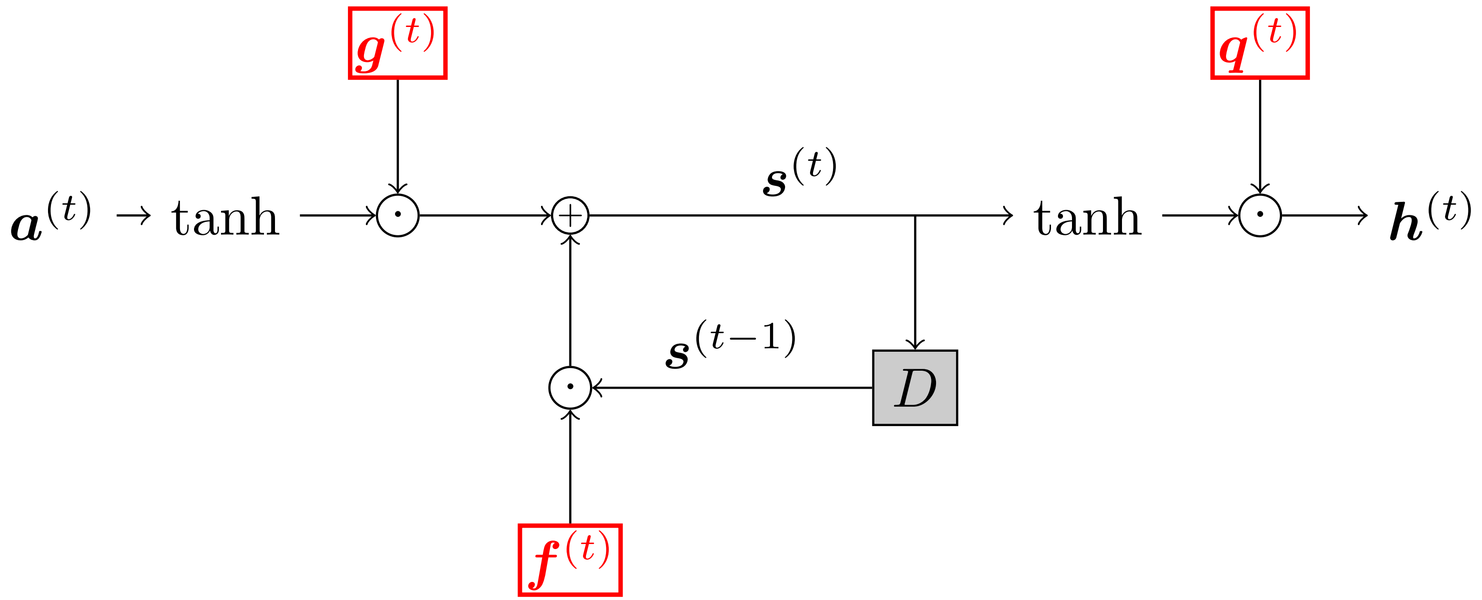


- The Jacobian $(\partial \boldsymbol{h}^{(t+1)}/\partial \boldsymbol{h}^{(t)})^T$ approximates a diagonally dominant matrix when α_i's are large enough, suggesting gradients can back propagate more easily

$$\nabla_{\boldsymbol{h}^{(t)}} L = \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}} \right)^T (\nabla_{\boldsymbol{h}^{(t+1)}} L)$$

Long Short-Term Memory (LSTM)

- To change the time scale of integration dynamically by introducing programable gates $(\boldsymbol{g}^{(t)}, \boldsymbol{f}^{(t)}, \boldsymbol{q}^{(t)})$ that are conditioned on context



- The gating context at time t refers collectively to $\{\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t-1)}\}$ and may include other inputs

– Memory state: $\boldsymbol{s}^{(t)}$

– Input gate: $\boldsymbol{g}^{(t)} = \sigma(\boldsymbol{U}^g \boldsymbol{x}^{(t)} + \boldsymbol{W}^g \boldsymbol{h}^{(t-1)})$

– Output gate: $\boldsymbol{q}^{(t)} = \sigma(\boldsymbol{U}^o \boldsymbol{x}^{(t)} + \boldsymbol{W}^o \boldsymbol{h}^{(t-1)})$

– Forget gate: $\boldsymbol{f}^{(t)} = \sigma(\boldsymbol{U}^f \boldsymbol{x}^{(t)} + \boldsymbol{W}^f \boldsymbol{h}^{(t-1)})$

– New content: $\boldsymbol{a}^{(t)} = \boldsymbol{U} \boldsymbol{x}^{(t)} + \boldsymbol{W} \boldsymbol{h}^{(t-1)}$

– Memory update: $\boldsymbol{s}^{(t)} = \boldsymbol{f}^{(t)} \odot \boldsymbol{s}^{(t-1)} + \boldsymbol{g}^{(t)} \odot \boldsymbol{a}^{(t)}$

– Hidden unit update: $\boldsymbol{h}^{(t)} = \boldsymbol{q}^{(t)} \odot \tanh(\boldsymbol{s}^{(t)})$

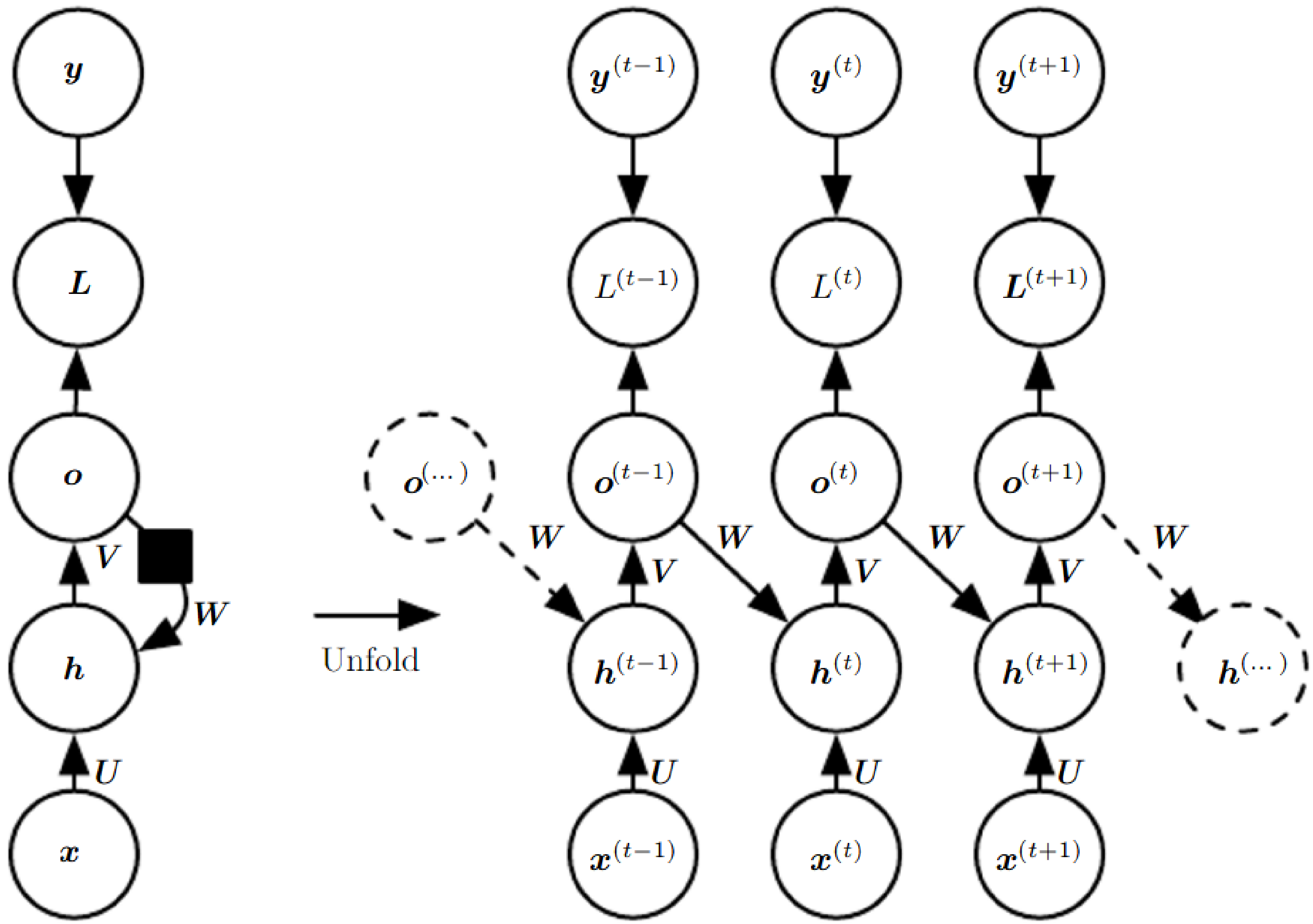
– Output unit update: $\boldsymbol{o}^{(t)} = \boldsymbol{V} \boldsymbol{h}^{(t)}$

- The design of LSTM is justified by the fact that there could be subsequences of varying statistics in a main sequence

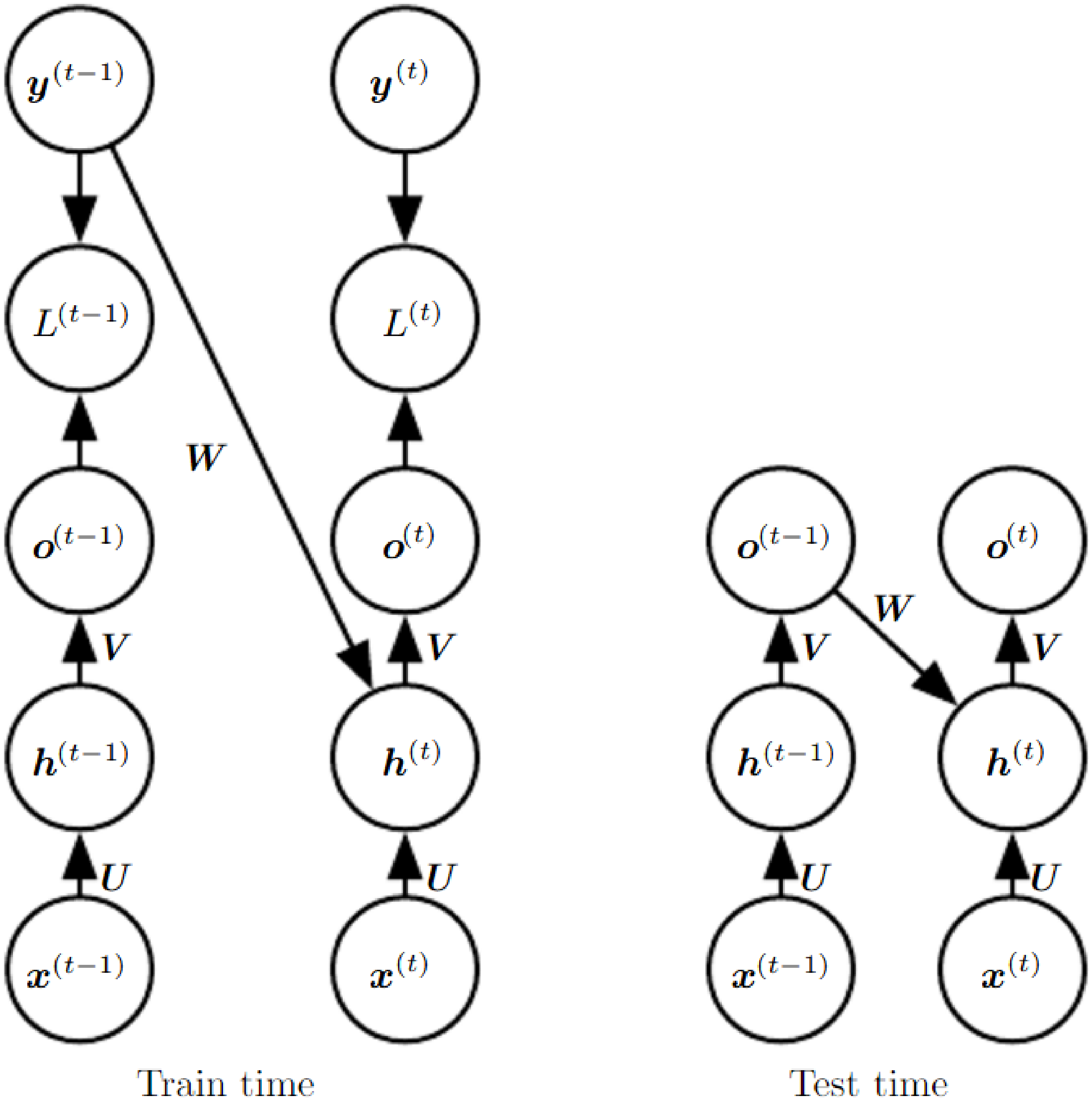
- Many variants of LSTM are available (study by yourself)

Design Pattern III

- Networks with only output recurrence

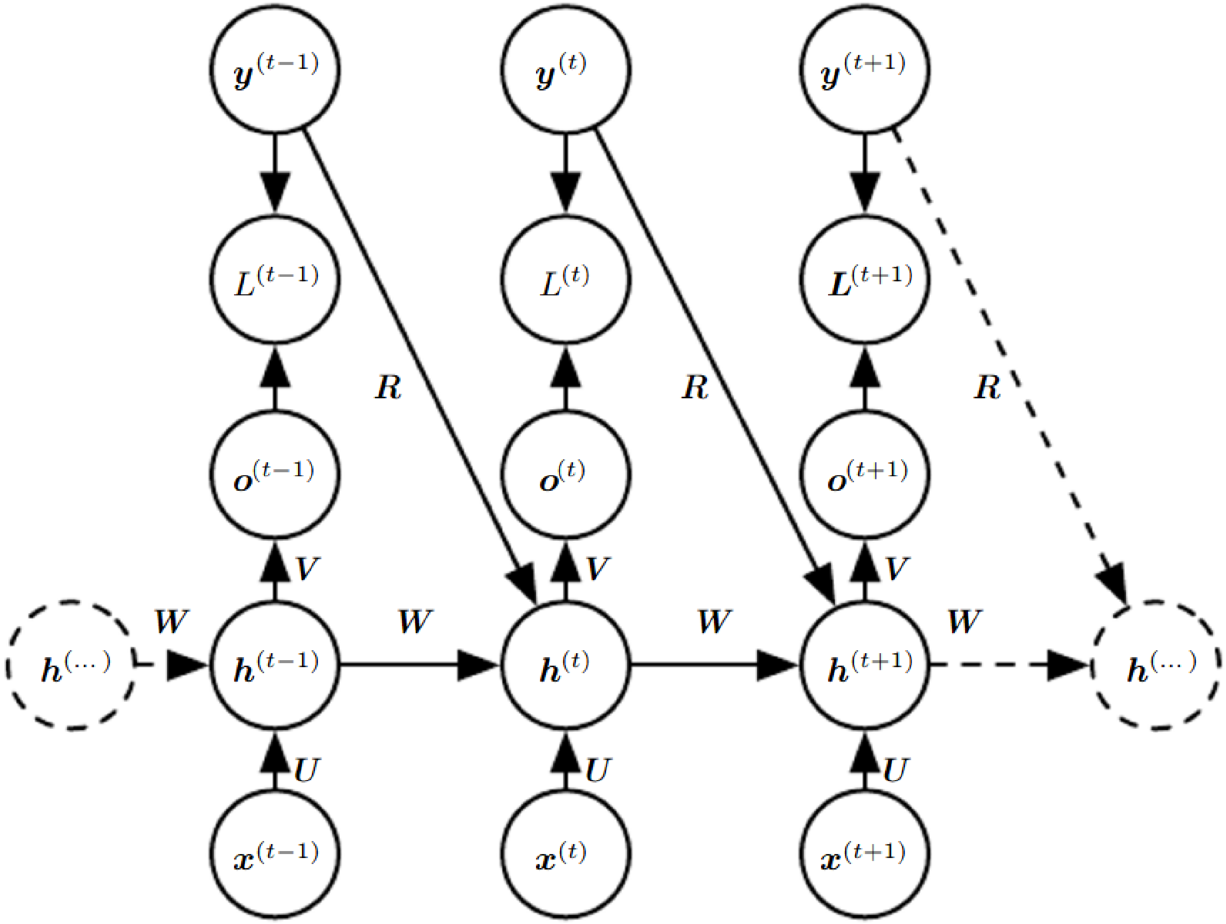


- Such networks are less powerful as the outputs $\boldsymbol{o}^{(t)}$'s are forced to approximate the targets $\boldsymbol{y}^{(t)}$'s while having to convey a good summary about the past
- They however shed lights on training networks with output recurrence
- One effective way for training this type of networks is **teacher forcing**, which feeds correct targets $\boldsymbol{y}^{(t)}$ into $\boldsymbol{h}^{(t+1)}$ during training, allowing the gradient for each time step to be computed in isolation
- **Open-loop issue:** inputs seen at training and test time are different
- To resolve this, it is common to train with both teacher-forced inputs and free-running inputs (generated by the output-to-input paths), or randomly choose between them


$y^{(t-1)}$
$y^{(t)}$
$L^{(t-1)}$
$L^{(t)}$
W
$o^{(t-1)}$
$o^{(t)}$
V
$h^{(t-1)}$
$h^{(t)}$
U
$x^{(t-1)}$
$x^{(t)}$
Train time
Test time

Design Pattern IV

- Networks with both output recurrence and hidden unit connections



- The training objective becomes to maximize

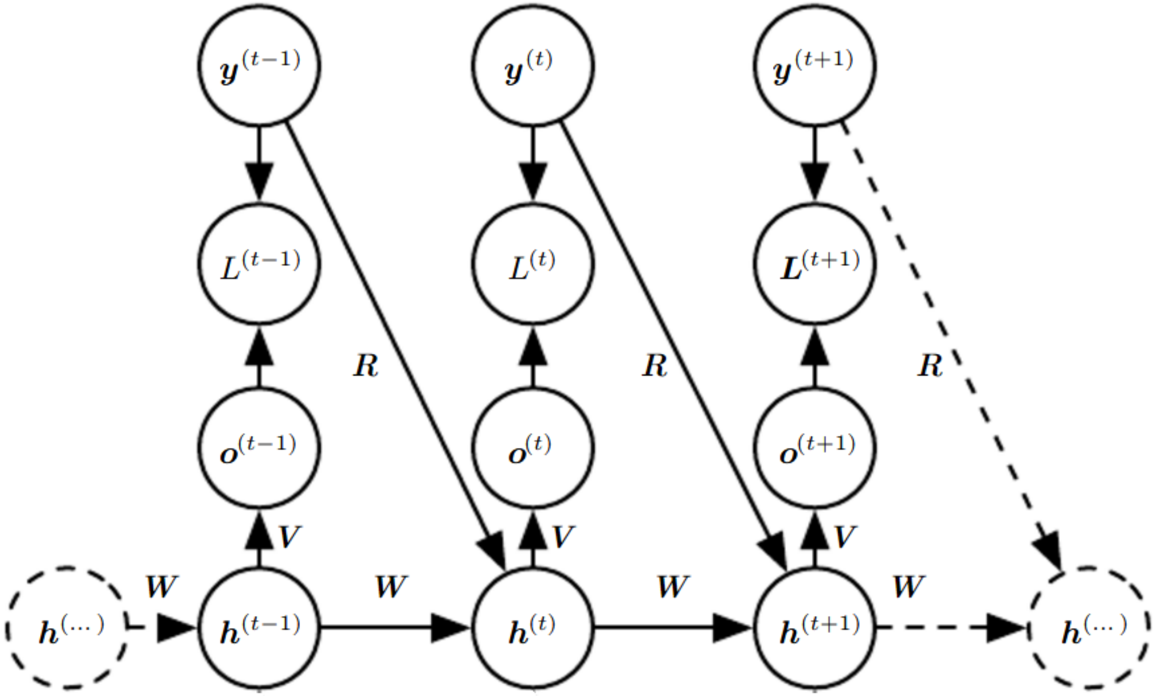
$$p_{\text{model}}(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(\tau)} | \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(\tau)})$$
$$= \prod_{t=1}^{\tau} p_{\text{model}}(\boldsymbol{y}^{(t)} | \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}, \boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(t-1)})$$

  where $\boldsymbol{y}^{(t)}$'s are modeled to be conditionally dependent (cf. Pattern I)

- This type of networks allows modeling an arbitrary distribution over the $\boldsymbol{y}$ sequence given the $\boldsymbol{x}$ sequence of the same length, whether $\boldsymbol{y}^{(t)}$'s are conditionally independent or dependent

- One may as well broadcast one single vector $\boldsymbol{x}$ as $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(\tau)}$, as is often seen in image capationing; in such tasks, $\boldsymbol{x}$ may denote the feature vector of an image and $\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(\tau)}$ are its caption

- One can also turn the network into a generative model for unsupervised learning by omitting all the $\boldsymbol{x}^{(t)}$'s

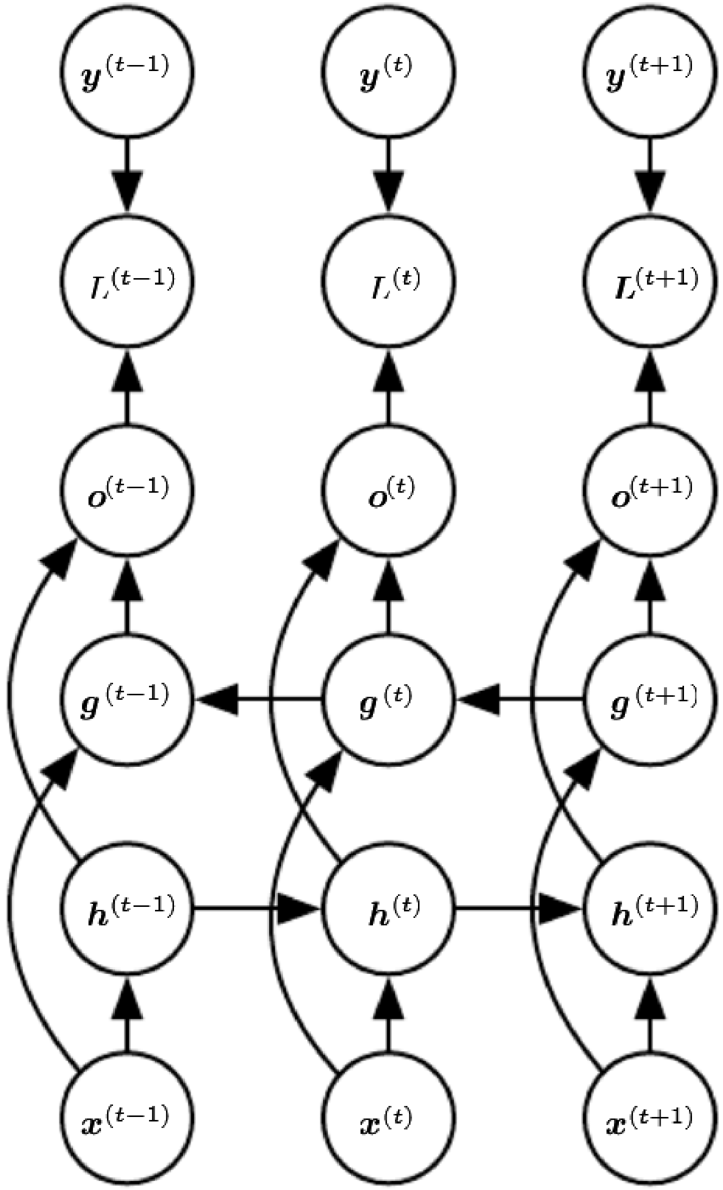


- Essentially, this corresponds to learning a model that factorizes as

$$p_{\text{model}}(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \dots, \boldsymbol{y}^{(\tau)}) = \prod_{t=1}^{\tau} p_{\text{model}}(\boldsymbol{y}^{(t)} | \boldsymbol{y}^{(t-1)}, \boldsymbol{y}^{(t-2)}, \dots, \boldsymbol{y}^{(1)})$$

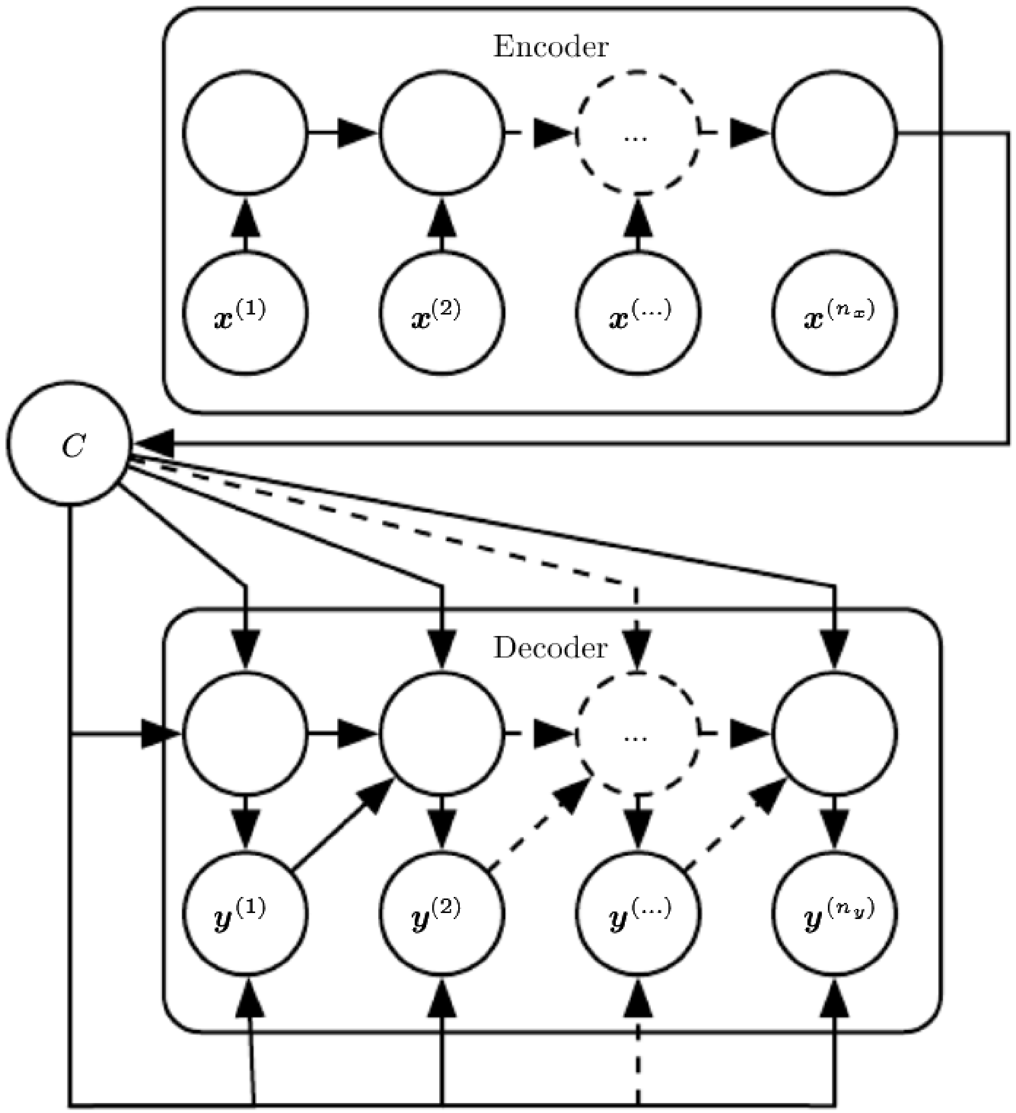
Bidirectional RNN

- RNN has a causal structure: the state $\boldsymbol{h}^{(t)}$ at time t captures only information from the past $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t-1)}$ and the present $\boldsymbol{x}^{(t)}$ inputs
- In many applications, it may be necessary to output a prediction $\boldsymbol{y}^{(t)}$ that depend on the entire sequence
- Bidirectional RNN combines two RNNs, one moving forward in time and the other moving backward, to capture information from both the past and future
- This however requires the entire sequence be buffered
- The notion can readily be extended to 2-D signals, e.g. images


$y^{(t-1)}$
$y^{(t)}$
$y^{(t+1)}$
$L^{(t-1)}$
$L^{(t)}$
$L^{(t+1)}$
$o^{(t-1)}$
$o^{(t)}$
$o^{(t+1)}$
$g^{(t-1)}$
$g^{(t)}$
$g^{(t+1)}$
$h^{(t-1)}$
$h^{(t)}$
$h^{(t+1)}$
$x^{(t-1)}$
$x^{(t)}$
$x^{(t+1)}$

Sequence-to-Sequence Networks

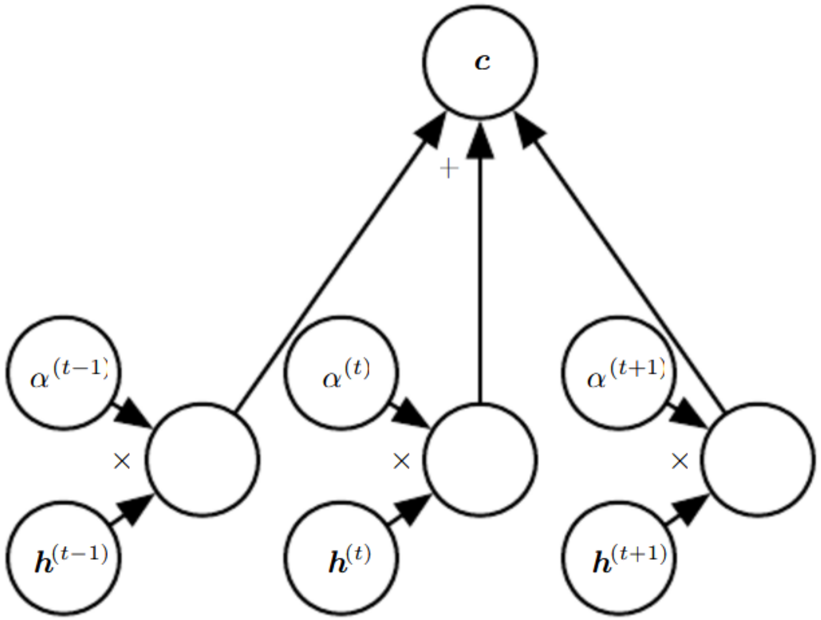
- In some applications, e.g. machine translation and question answering, it may be necessary to map an input sequence to an output sequence that is not of the same length
- The encoder network encodes the input sequence and emits a context C for the decoder network to generate the output sequence
- The context C can be a function of the last hidden unit or a summary of different hidden units by introducing an attention mechanism



Encoder
...
$x^{(1)}$
$x^{(2)}$
$x^{(\dots)}$
$x^{(n_x)}$
C
Decoder
...
$y^{(1)}$
$y^{(2)}$
$y^{(\dots)}$
$y^{(n_y)}$

Attention Mechanisms

- A weighted average of information with weights $\boldsymbol{\alpha}^{(t)}$'s (gate signals) produced by the model itself



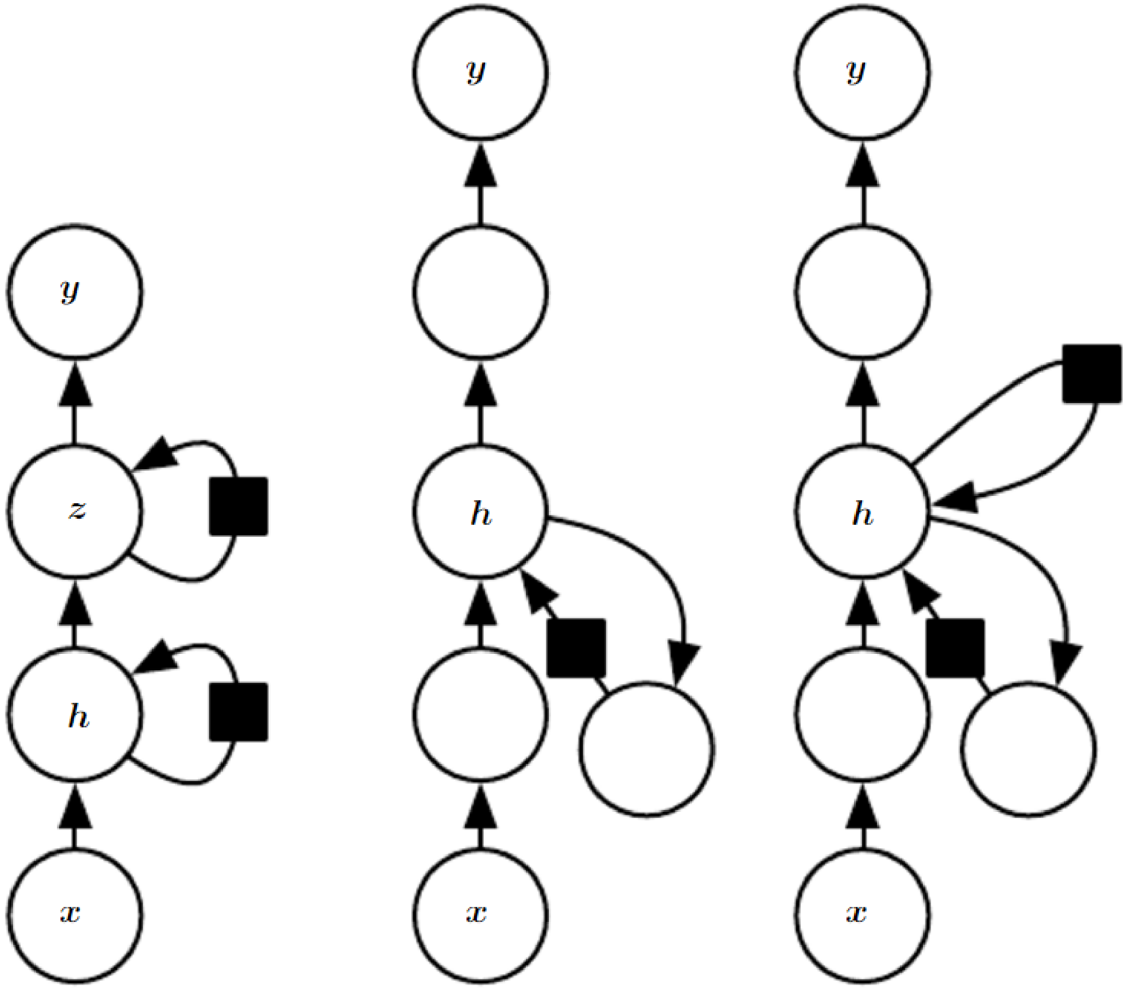
- The information to be averaged could be hidden units of a network or raw input data
- The weights are usually produced by applying a softmax function to

some relevance scores emitted somewhere in the model

- The soft weighting is more expensive than direct indexing but is differentiable, making it trainable with gradient-based algorithms

Deep Recurrent Networks

- RNN can be made deep in many ways
- Recurrent hidden units can be organized in a layered manner
- MLP can be introduced in the input-to-hidden, hidden-to-hidden, and hidden-to-output parts
- In doing so, skip connections may be necessary to mitigate the gradient vanishing and exploding problem

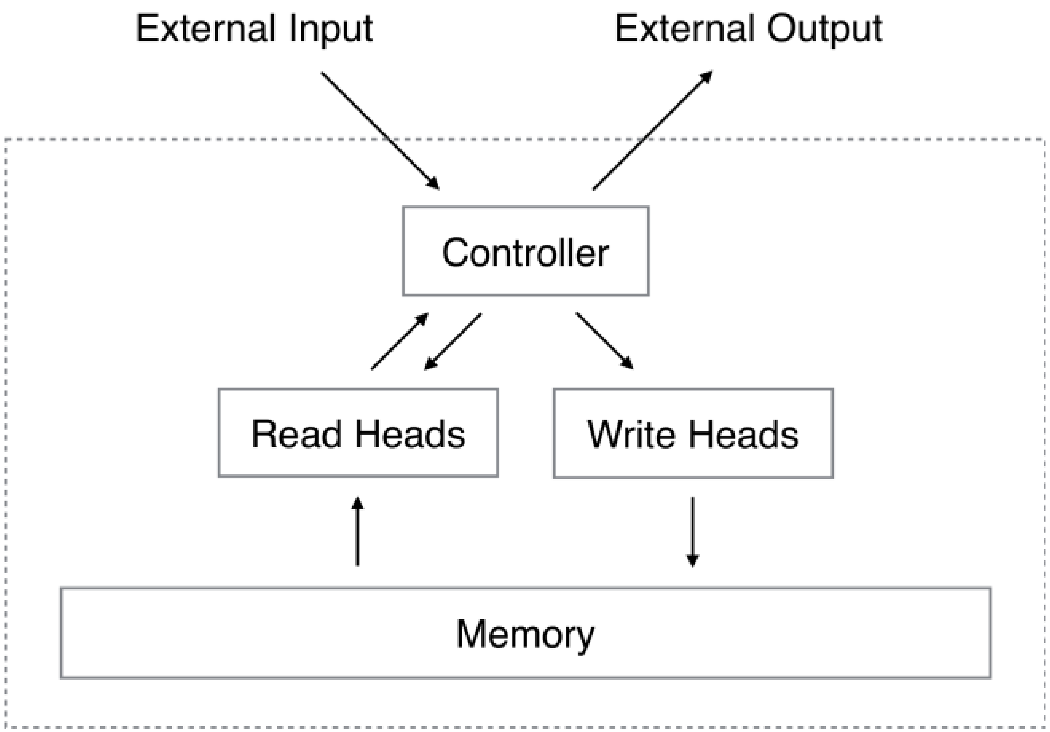


(a) (b) (c)

Neural Turing Machines (NTM)

- An end-to-end differentiable machine that combines **neural networks, attention mechanisms, and memory** to imitate a Turing Machine or Von Neumann architecture
- Von Neumann architecture
  - Arithmetic operations
  - Logical flow control (branching)
  - Memory
- NTM can learn to infer simple algorithms, e.g. copying, sorting, recall

- NTM architecture



LSTM/NN

Attention

Memory

  - Memory state: $\boldsymbol{M}_t$
  - Address: $\boldsymbol{w}_t$

$$\sum_i w_t(i) = 1, \; 0 \leq w_t(i) \leq 1$$

– Reading

$$\boldsymbol{r}_t = \sum_i w_t(i)\boldsymbol{M}_t(i)$$

– Writing

$$\boldsymbol{M}_t(i) = \sum_i \boldsymbol{M}_{t-1}(i)[\mathbf{1} - w_t(i)\boldsymbol{e}_t] + w_t(i)\boldsymbol{a}_t$$

where

$\mathbf{1}$: A vector of all 1's

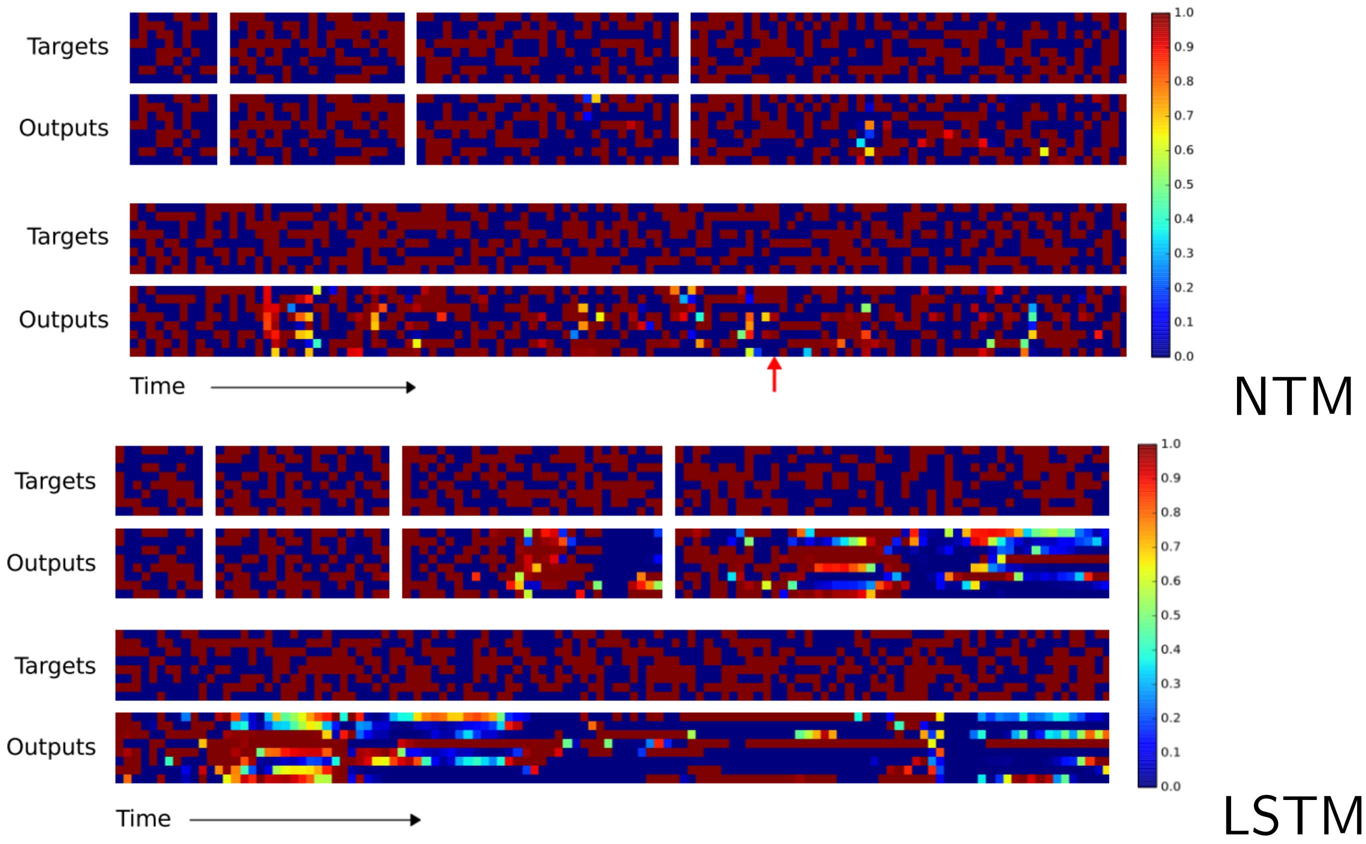
$\boldsymbol{e}_t$: Erasure vector with elements in (0,1)

$\boldsymbol{a}_t$: New content to be written

– Addressing: content- and location-based (study the paper)

– Logical flow control (study the code)

- Copying
  - Copying 8-bit random binary vectors
  - Trained on sequences of length 1-20
  - Tested on sequences of length 10, 20, 30, 50, 120



- Repeat copy, associative recall, N-grams, priority sort, etc. (study the paper)

Review

- Design patterns of RNN and their probability models
- Gradient vanishing and exploding problem
- Long short-term memory
- Back-propagation through time (BPTT)
- Training with output recurrence
- Attention mechanisms
- Neural Turing Machines