Deep Learning Crash Course

Hui Xue

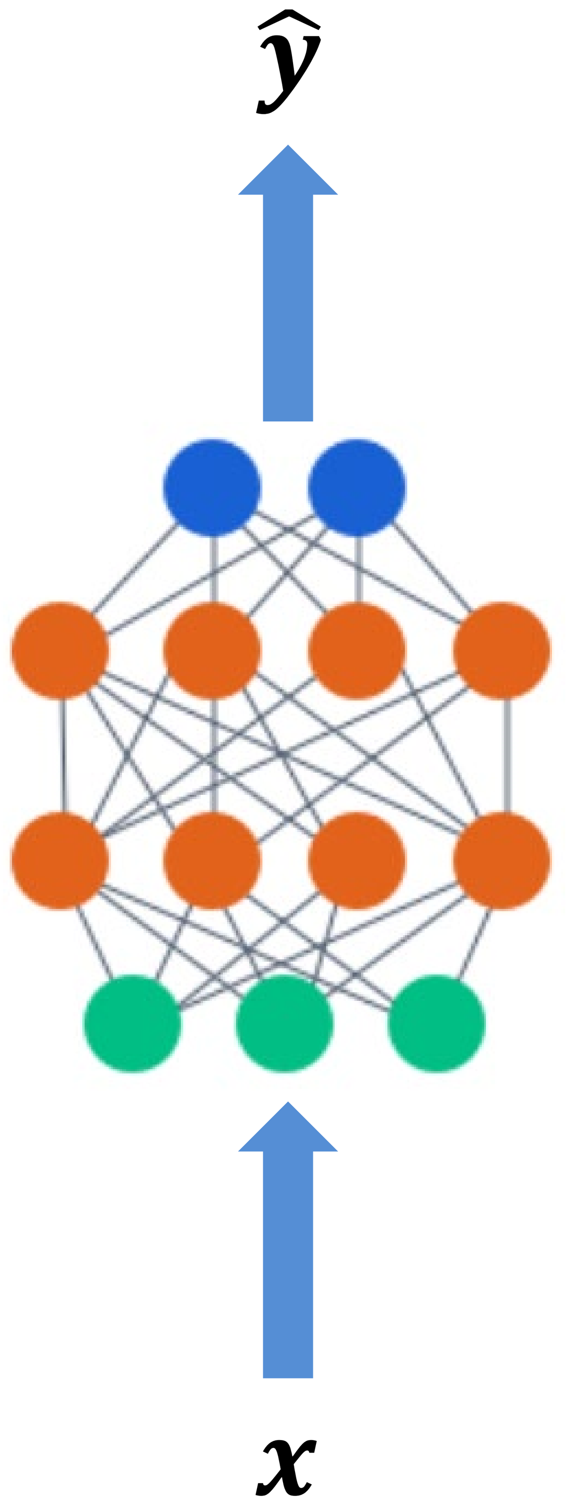
Fall 2021

www.deeplearningcrashcourse.org

Outline

- Vanilla RNN and backprop through time
- Variation of RNNs
- LSTM and GRU
- Multi-layer RNN and bidirectional RNN
- Sequence pre-processing

So far, one-to-one mapping

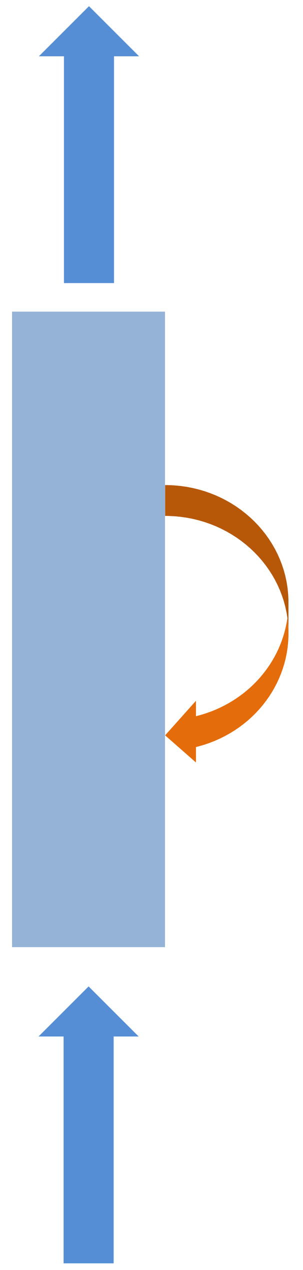


- Image
- Video with fixed length
- Tabulate feature set

…

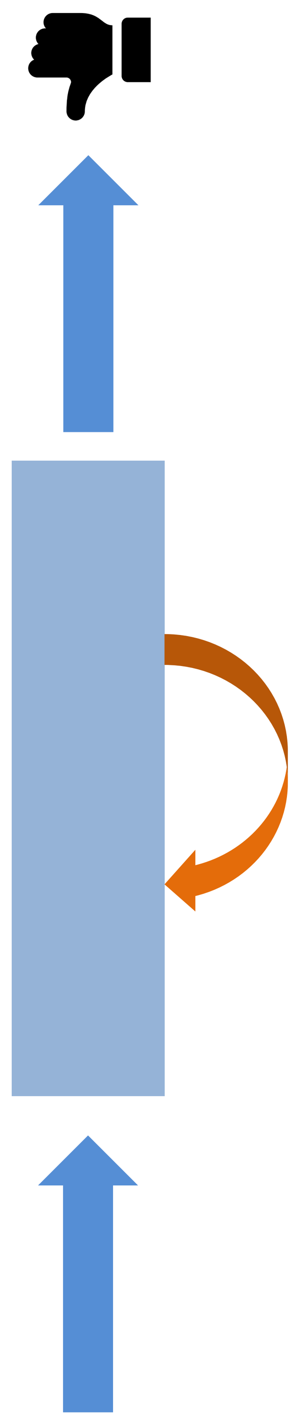
Image Captioning

a cow is standing in the middle of a street



https://cs.stanford.edu/people/karpathy/sfmltalk.pdf

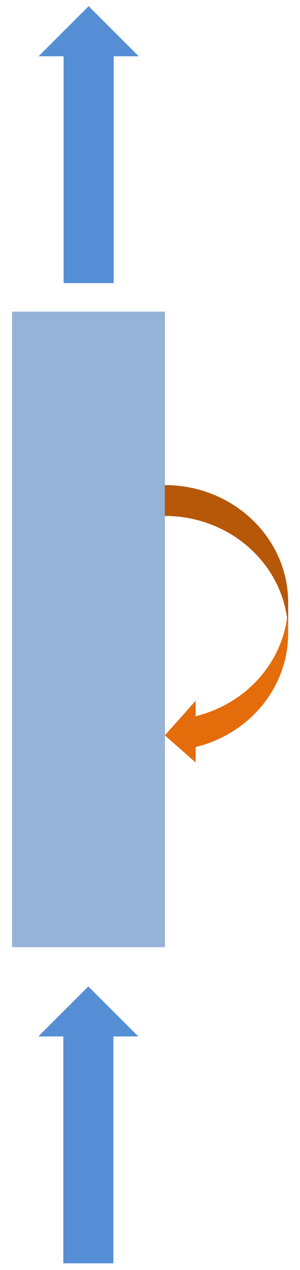
Sentimental classification

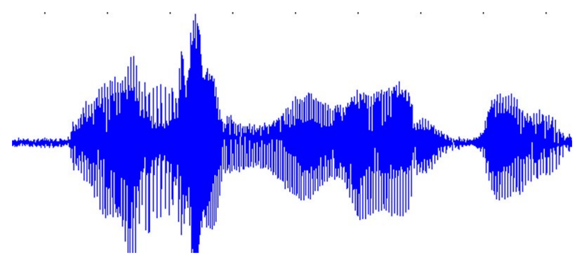


Three weeks ago, I bought this product. It worked fine at the beginning but stopped working after a week ..

Speech recognition

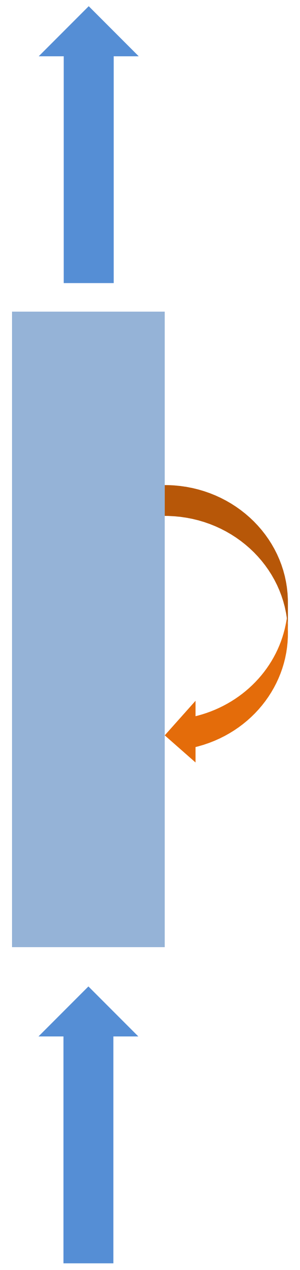
Today is nice. Let’s go to the park.





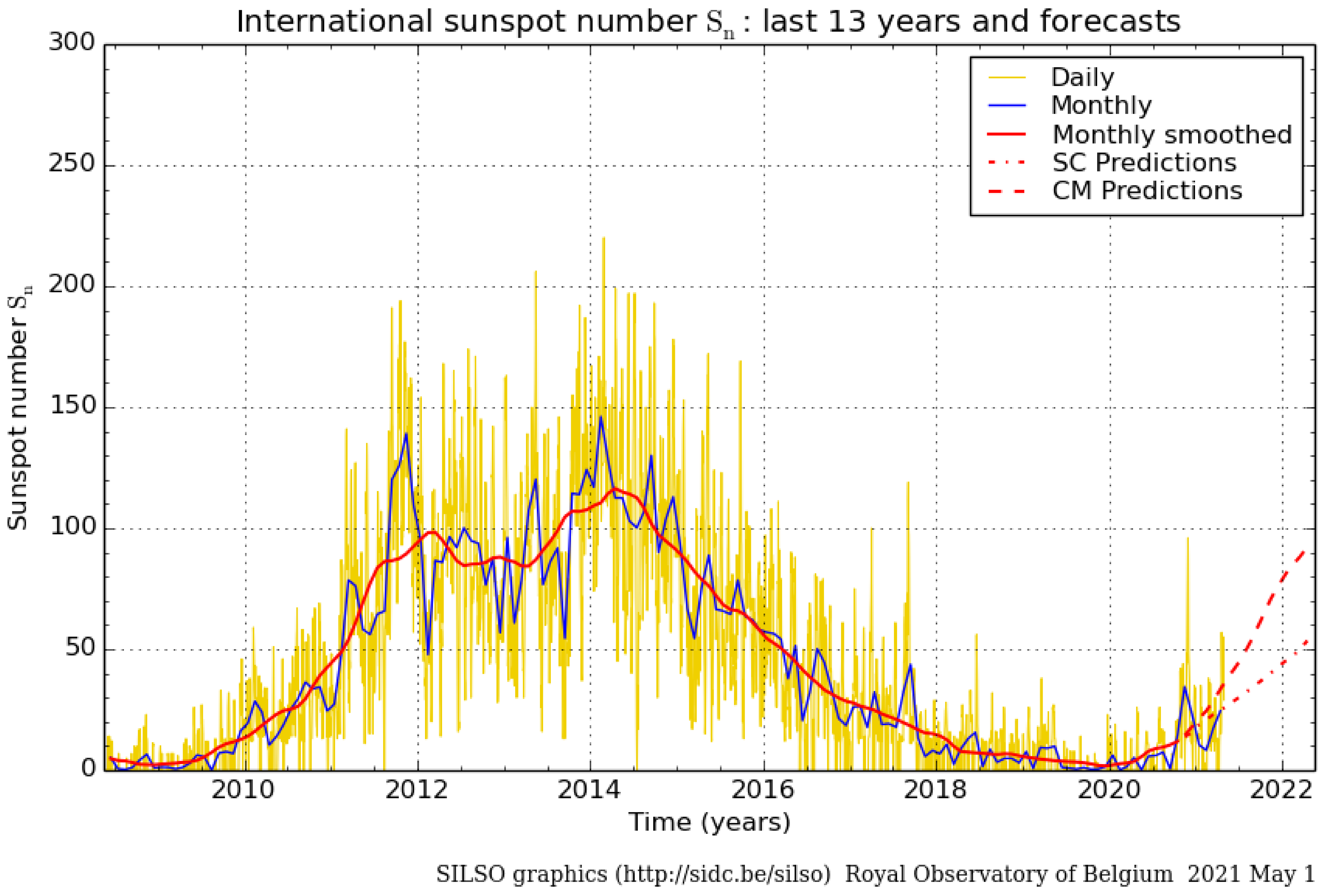
Machine translation

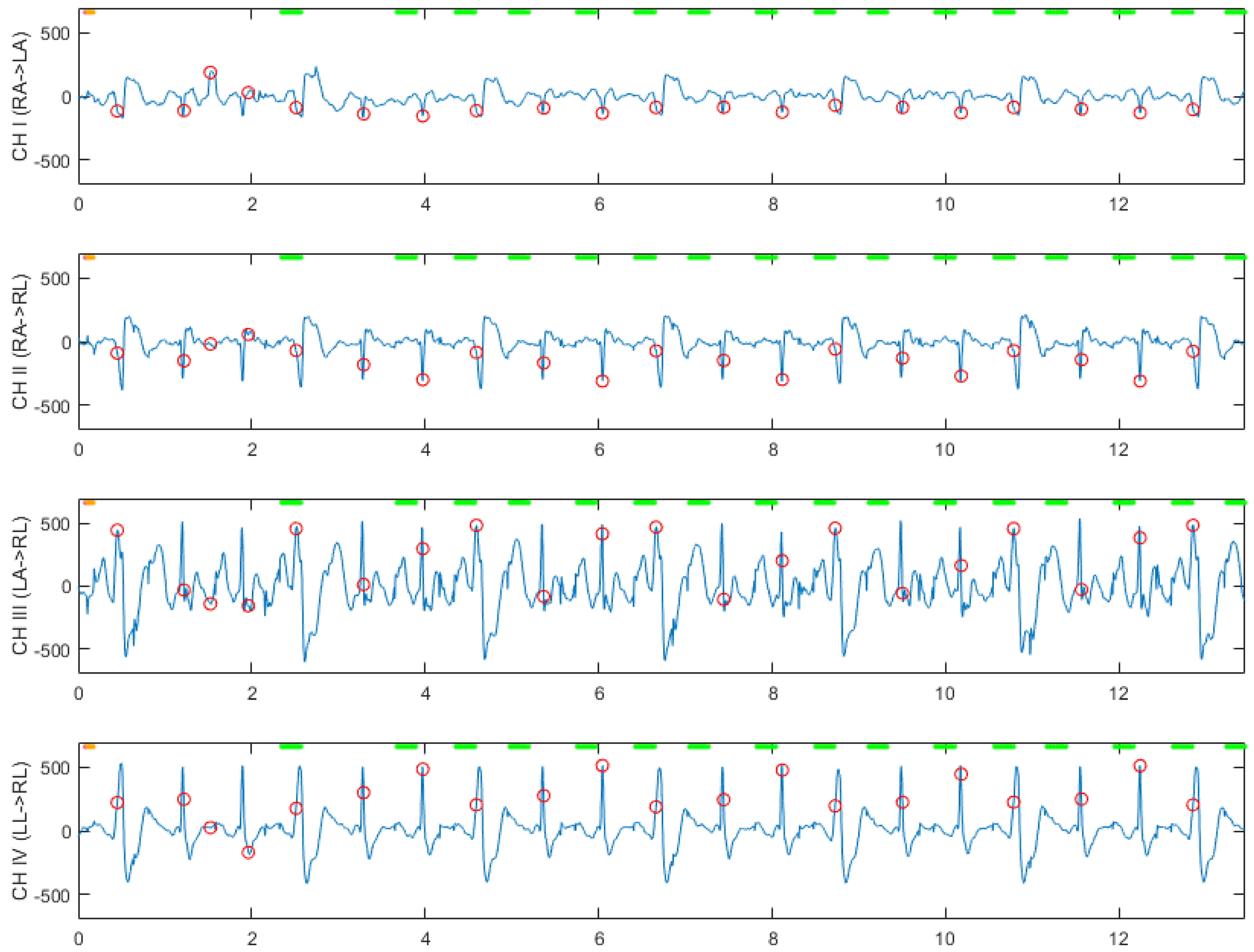
今天天气很好。我们去公园吧。



Today is nice. Let’s go to the park.

Sequence model for regression or detection





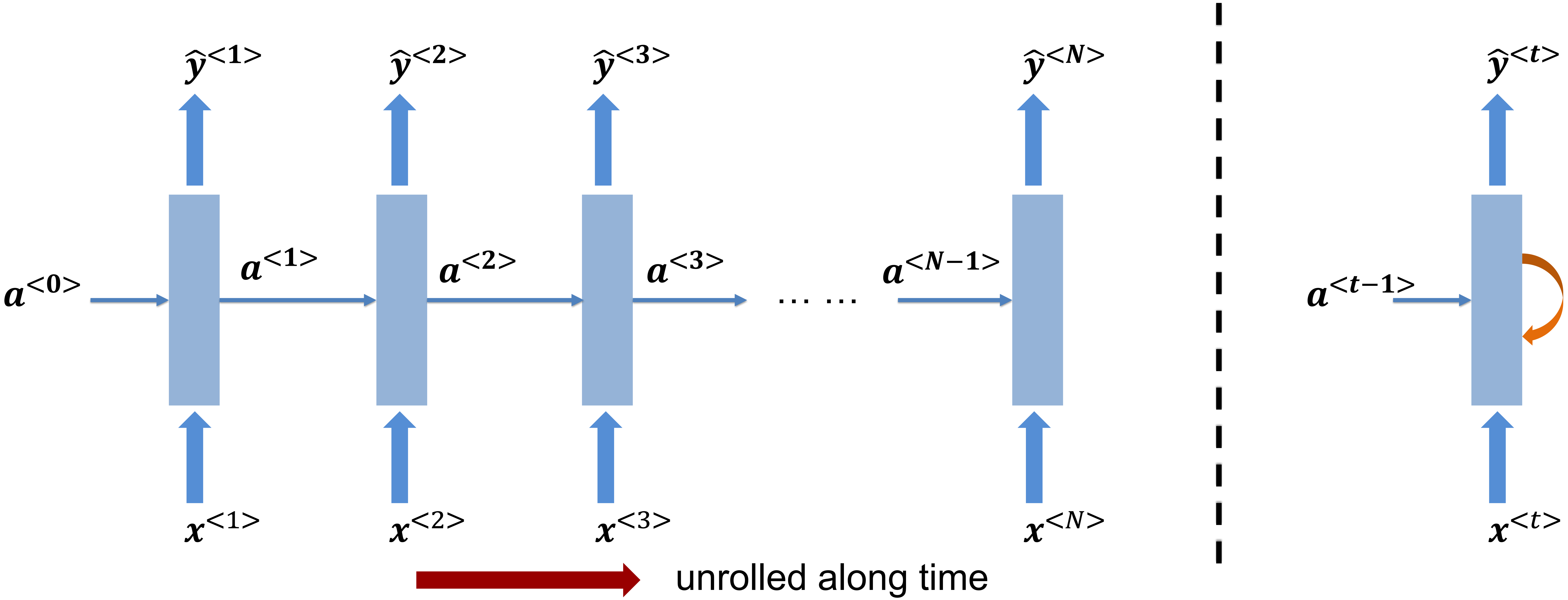
Regression to estimate the number of sunspots

Detect R-wave trigger from ECG waveform

http://www.sidc.be/silso/dayssnplot

Recurrent Neural Network

- Give a series of inputs $\boldsymbol{x}^{<t>}, t = 1, \ldots, N$, a RNN is a model to receive every input and produce output $\boldsymbol{y}^{<t>}$
- RNN has internal state $\boldsymbol{a}^{<t>}$
- All time steps share the model parameters
- $\boldsymbol{a}^{<0>}$ is often initialized as 0



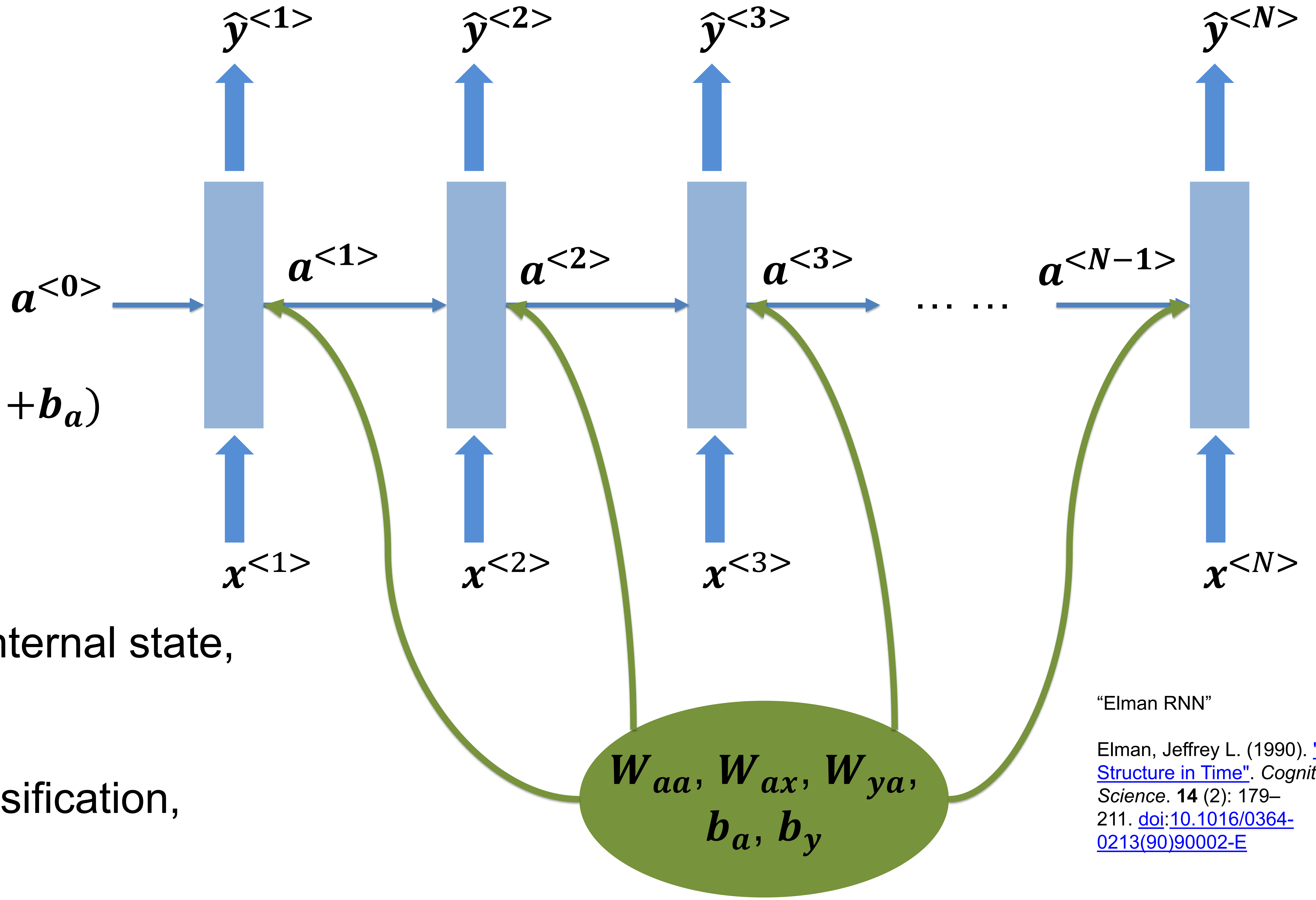
Recurrent Neural Network

- input $\boldsymbol{x}^{<t>}, t = 1, \dots, N$, output $\boldsymbol{y}^{<t>}$
- internal state $\boldsymbol{a}^{<t>}$

$$\boldsymbol{a}^{<0>} = \boldsymbol{0}$$

$$\boldsymbol{a}^{<t>} = \boldsymbol{g_a}(\boldsymbol{W_{aa}} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{ax}} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b_a})$$

$$\boldsymbol{y}^{<t>} = \boldsymbol{g_y}(\boldsymbol{W_{ya}} \cdot \boldsymbol{a}^{<t>} + \boldsymbol{b_y})$$



$\boldsymbol{g_a}$: nonlinear activation function for internal state, often tanh, ReLU

$\boldsymbol{g_y}$: for output, sigmoid for binary classification, softmax for multi-class

"Elman RNN"

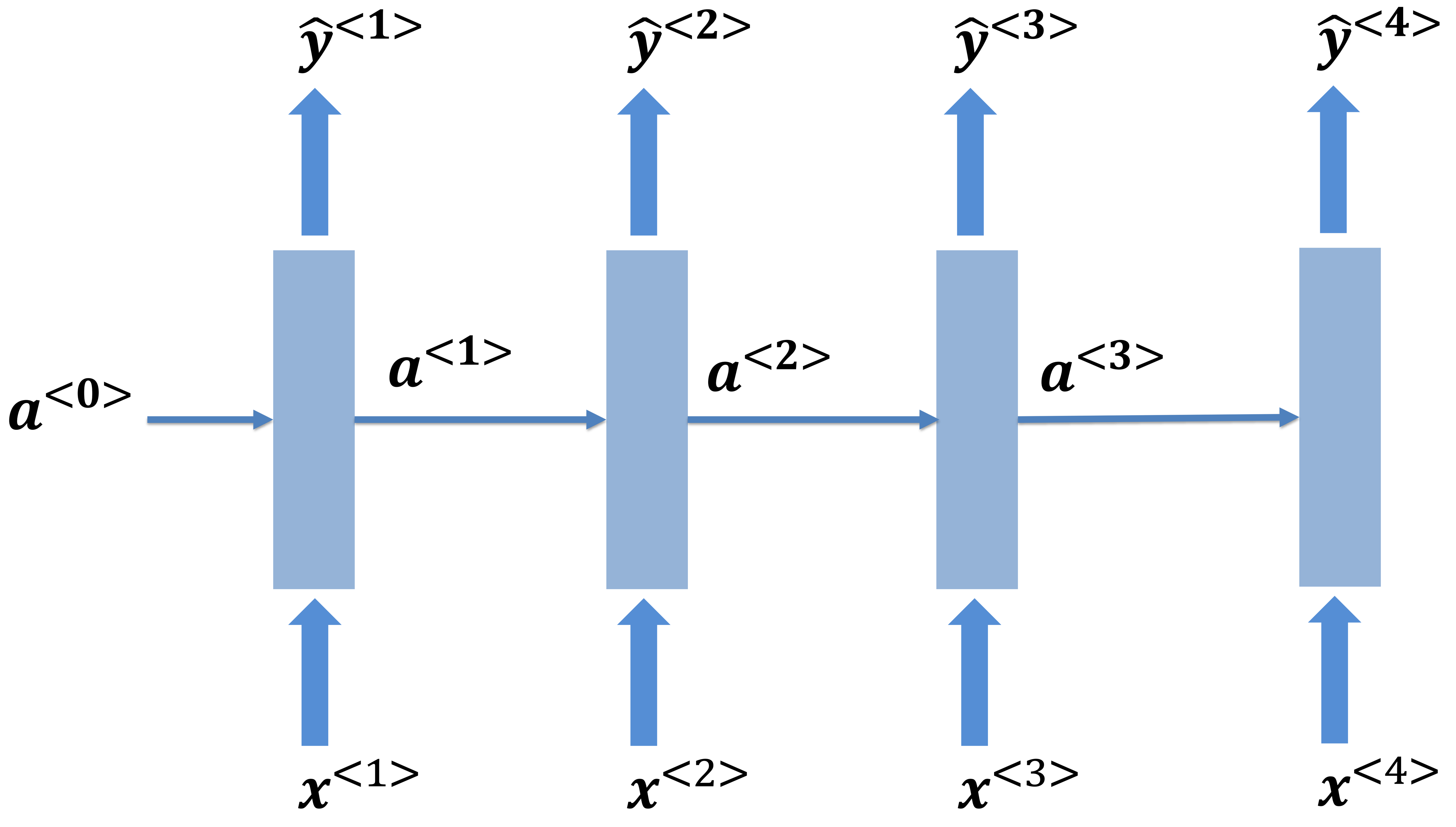
Elman, Jeffrey L. (1990). "Finding Structure in Time". *Cognitive Science*. **14** (2): 179–211. doi:10.1016/0364-0213(90)90002-E

$$\boldsymbol{Jordan\ RNN}: \boldsymbol{a}^{<t>} = \boldsymbol{g_a}(\boldsymbol{W_{aa}} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{ay}} \cdot \boldsymbol{y}^{<t-1>} + \boldsymbol{b_a})$$

Jordan, Michael I. (1997-01-01). "Serial Order: A Parallel Distributed Processing Approach". *Neural-Network Models of Cognition - Biobehavioral Foundations. Advances in Psychology*. Neural-Network Models of Cognition. **121**. pp. 471–495.

Recurrent Neural Network: Forward pass

- input $\boldsymbol{x}^{<t>}, t = 1, \ldots, N$, output $\boldsymbol{y}^{<t>}$
- internal state $\boldsymbol{a}^{<t>}$



$$\boldsymbol{a}^{<0>} = \boldsymbol{0}$$

$$\boldsymbol{a}^{<1>} = \boldsymbol{g}_a(\boldsymbol{W}_{aa} \cdot \boldsymbol{a}^{<0>} + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<1>} + \boldsymbol{b}_a)$$

$$\boldsymbol{y}^{<1>} = \boldsymbol{g}_y(\boldsymbol{W}_{ya} \cdot \boldsymbol{a}^{<1>} + \boldsymbol{b}_y)$$

$$\boldsymbol{a}^{<2>} = \boldsymbol{g}_a(\boldsymbol{W}_{aa} \cdot \boldsymbol{a}^{<1>} + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<2>} + \boldsymbol{b}_a)$$

$$\boldsymbol{y}^{<2>} = \boldsymbol{g}_y(\boldsymbol{W}_{ya} \cdot \boldsymbol{a}^{<2>} + \boldsymbol{b}_y)$$

$$\boldsymbol{a}^{<3>} = \boldsymbol{g}_a(\boldsymbol{W}_{aa} \cdot \boldsymbol{a}^{<2>} + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<3>} + \boldsymbol{b}_a)$$

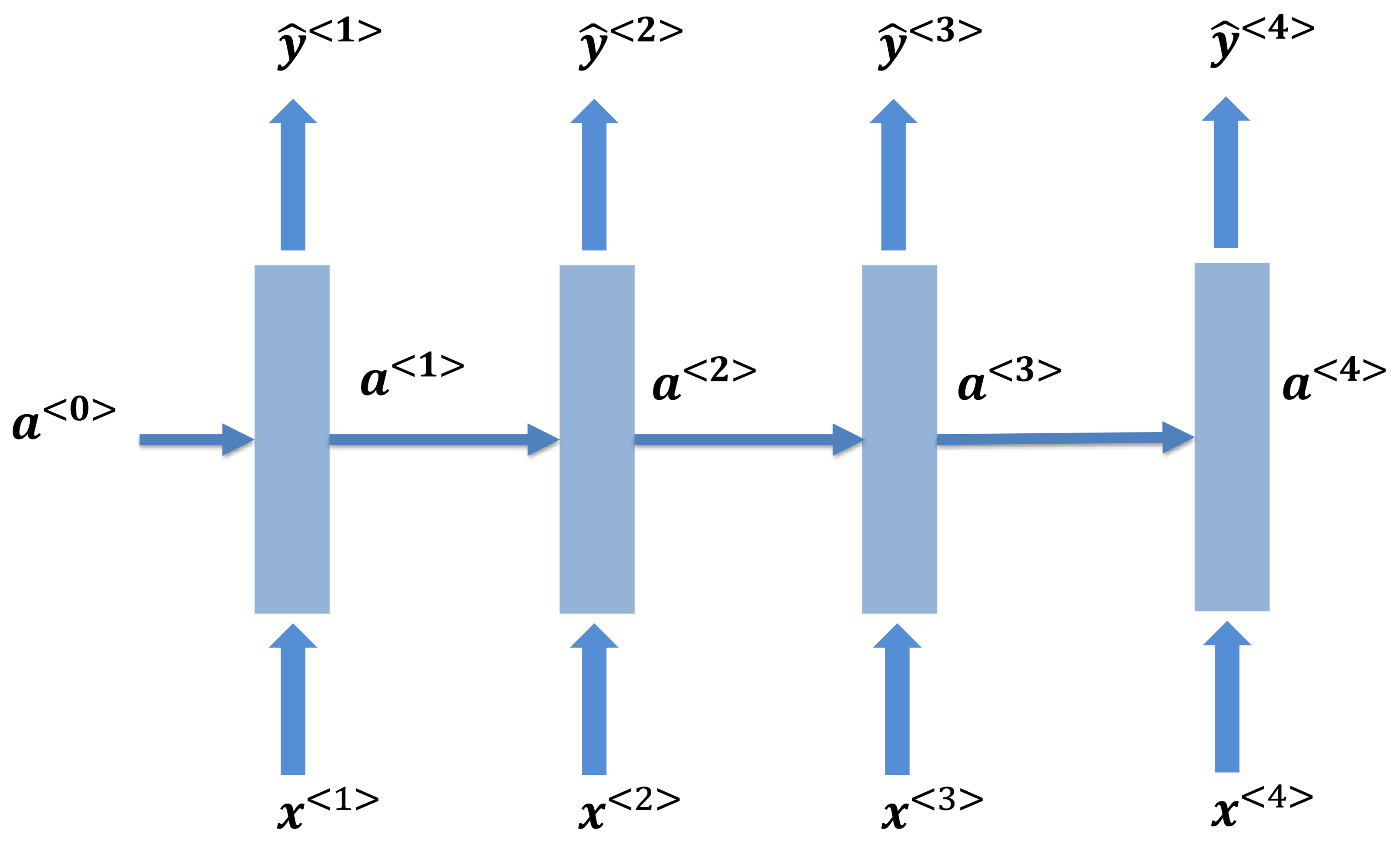
$$\boldsymbol{y}^{<3>} = \boldsymbol{g}_y(\boldsymbol{W}_{ya} \cdot \boldsymbol{a}^{<3>} + \boldsymbol{b}_y)$$

$$\boldsymbol{a}^{<4>} = \boldsymbol{g}_a(\boldsymbol{W}_{aa} \cdot \boldsymbol{a}^{<3>} + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<4>} + \boldsymbol{b}_a)$$

$$\boldsymbol{y}^{<4>} = \boldsymbol{g}_y(\boldsymbol{W}_{ya} \cdot \boldsymbol{a}^{<4>} + \boldsymbol{b}_y)$$

Recurrent Neural Network: recurrent

- input $\boldsymbol{x}^{<t>}, t = 1, \ldots, N$, output $\boldsymbol{y}^{<t>}$
- internal state $\boldsymbol{a}^{<t>}$



$$\boldsymbol{a}^{<0>} = \boldsymbol{0}$$

$$\boldsymbol{a}^{<t>} = \boldsymbol{g}_a(\boldsymbol{W}_{aa} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b}_a)$$

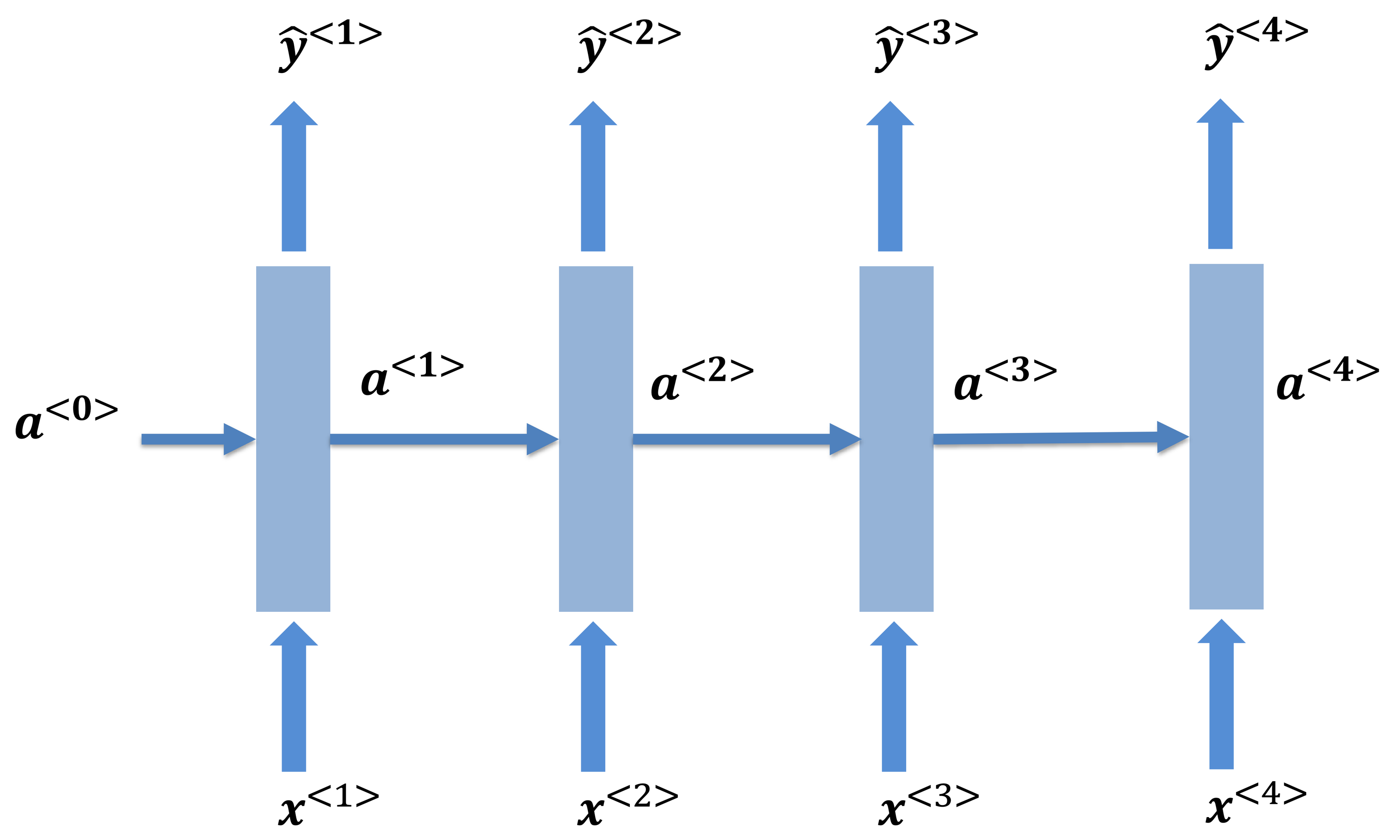
$$\begin{aligned}
\boldsymbol{a}^{<4>} &= \boldsymbol{g}_a\big(\boldsymbol{W}_{aa} \cdot \boldsymbol{a}^{<3>} + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<4>} + \boldsymbol{b}_a\big) \\
&= \boldsymbol{g}_a\big(\boldsymbol{W}_{aa} \cdot [\boldsymbol{g}_a(\boldsymbol{W}_{aa} \cdot \boldsymbol{a}^{<2>} + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<3>} + \boldsymbol{b}_a)] + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<4>} + \boldsymbol{b}_a\big) \\
&= \boldsymbol{g}_a\big(\boldsymbol{W}_{aa} \cdot [\boldsymbol{g}_a(\boldsymbol{W}_{aa} \cdot [\boldsymbol{g}_a(\boldsymbol{W}_{aa} \cdot \boldsymbol{a}^{<1>} + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<2>} + \boldsymbol{b}_a)] + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<3>} + \boldsymbol{b}_a)] + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<4>} + \boldsymbol{b}_a\big) \\
&= \boldsymbol{g}_a\big(\boldsymbol{W}_{aa} \cdot [\boldsymbol{g}_a(\boldsymbol{W}_{aa} \cdot [\boldsymbol{g}_a(\boldsymbol{W}_{aa} \cdot [\boldsymbol{g}_a(\boldsymbol{W}_{aa} \cdot \boldsymbol{a}^{<0>} + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<1>} + \boldsymbol{b}_a)] + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<2>} + \boldsymbol{b}_a)] + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<3>} + \boldsymbol{b}_a)] + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<4>} + \boldsymbol{b}_a\big)
\end{aligned}$$

$$\boldsymbol{y}^{<4>} = \boldsymbol{g}_y(\boldsymbol{W}_{ya} \cdot \boldsymbol{a}^{<4>} + \boldsymbol{b}_y)$$

- Recurrently apply the model parameters to update internal state
- $\boldsymbol{y}^{<t>}$ depends on all inputs on or before current time step

Recurrent Neural Network: loss

- input $\boldsymbol{x}^{<t>}, t = 1, \ldots, N$, output $\boldsymbol{y}^{<t>}$
- internal state $\boldsymbol{a}^{<t>}$



$$\boldsymbol{a}^{<t>} = \boldsymbol{g_a}(\boldsymbol{W_{aa}} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{ax}} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b_a})$$

$$\boldsymbol{y}^{<t>} = \boldsymbol{g_y}(\boldsymbol{W_{ya}} \cdot \boldsymbol{a}^{<t>} + \boldsymbol{b_y})$$

Assume $\boldsymbol{y}^{<t>}$ to take 0 or 1, e.g. like/dislike, binary classification

$$\ell^{<t>}(\boldsymbol{y}^{<t>}, \hat{\boldsymbol{y}}^{<t>}) = -\boldsymbol{y}^{<t>} \boldsymbol{log}(\hat{\boldsymbol{y}}^{<t>}) - (1 - \boldsymbol{y}^{<t>}) \boldsymbol{log}(1 - \hat{\boldsymbol{y}}^{<t>})$$

$$\ell = \sum_{t=1}^{N} \ell^{<t>}(\boldsymbol{y}^{<t>}, \hat{\boldsymbol{y}}^{<t>})$$

$$\frac{\partial \ell}{\partial W_{aa}} = \sum_{t=1}^{N} \frac{\partial \ell^{<t>}(\boldsymbol{y}^{<t>}, \hat{\boldsymbol{y}}^{<t>})}{W_{aa}} \qquad \frac{\partial \ell}{\partial W_{ax}} = \sum_{t=1}^{N} \frac{\partial \ell^{<t>}(\boldsymbol{y}^{<t>}, \hat{\boldsymbol{y}}^{<t>})}{W_{ax}}$$

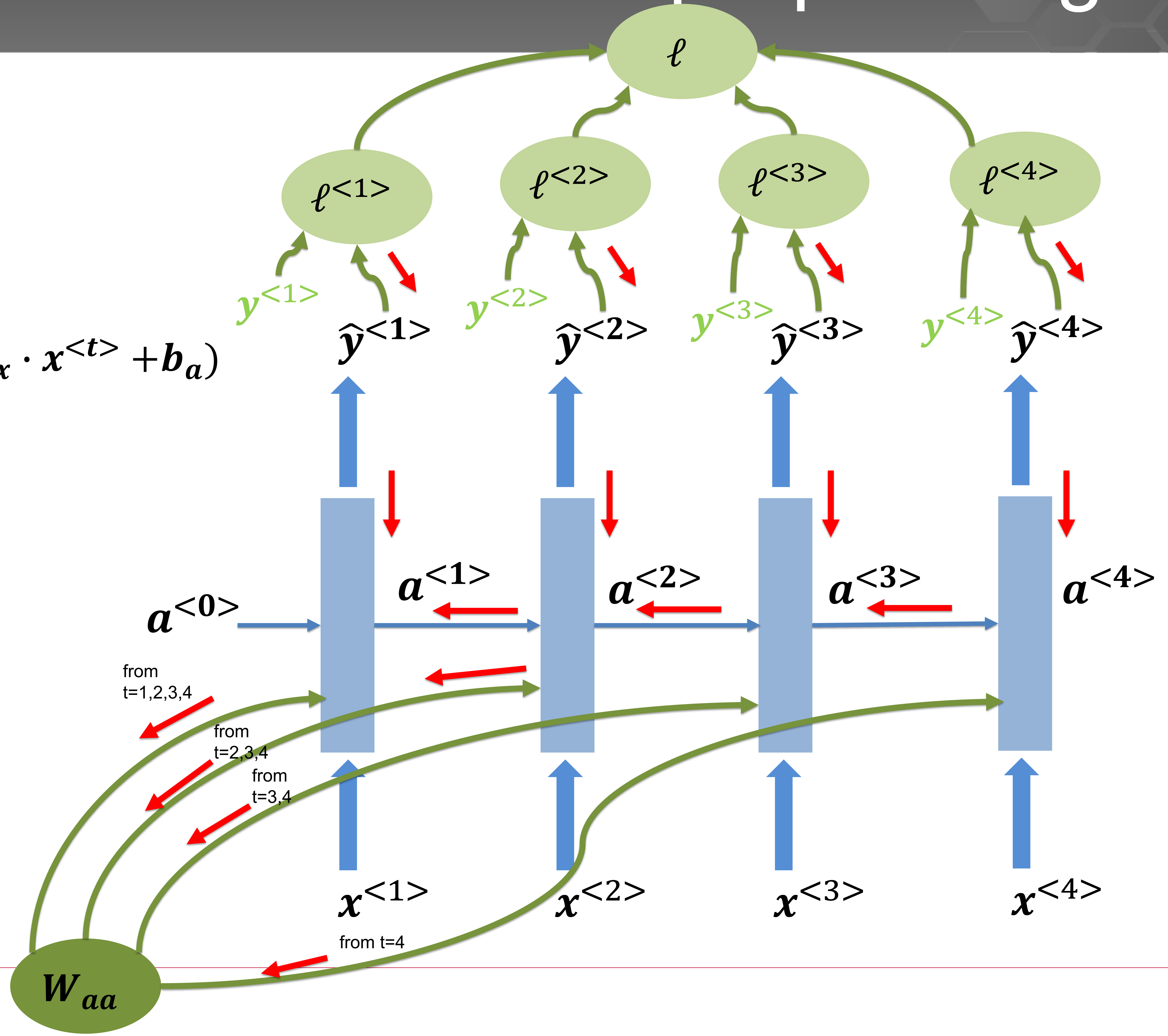
$$\frac{\partial \ell}{\partial W_{ya}} = \sum_{t=1}^{N} \frac{\partial \ell^{<t>}(\boldsymbol{y}^{<t>}, \hat{\boldsymbol{y}}^{<t>})}{W_{ya}} \qquad \ldots \ldots$$

Recurrent Neural Network: backprop through time

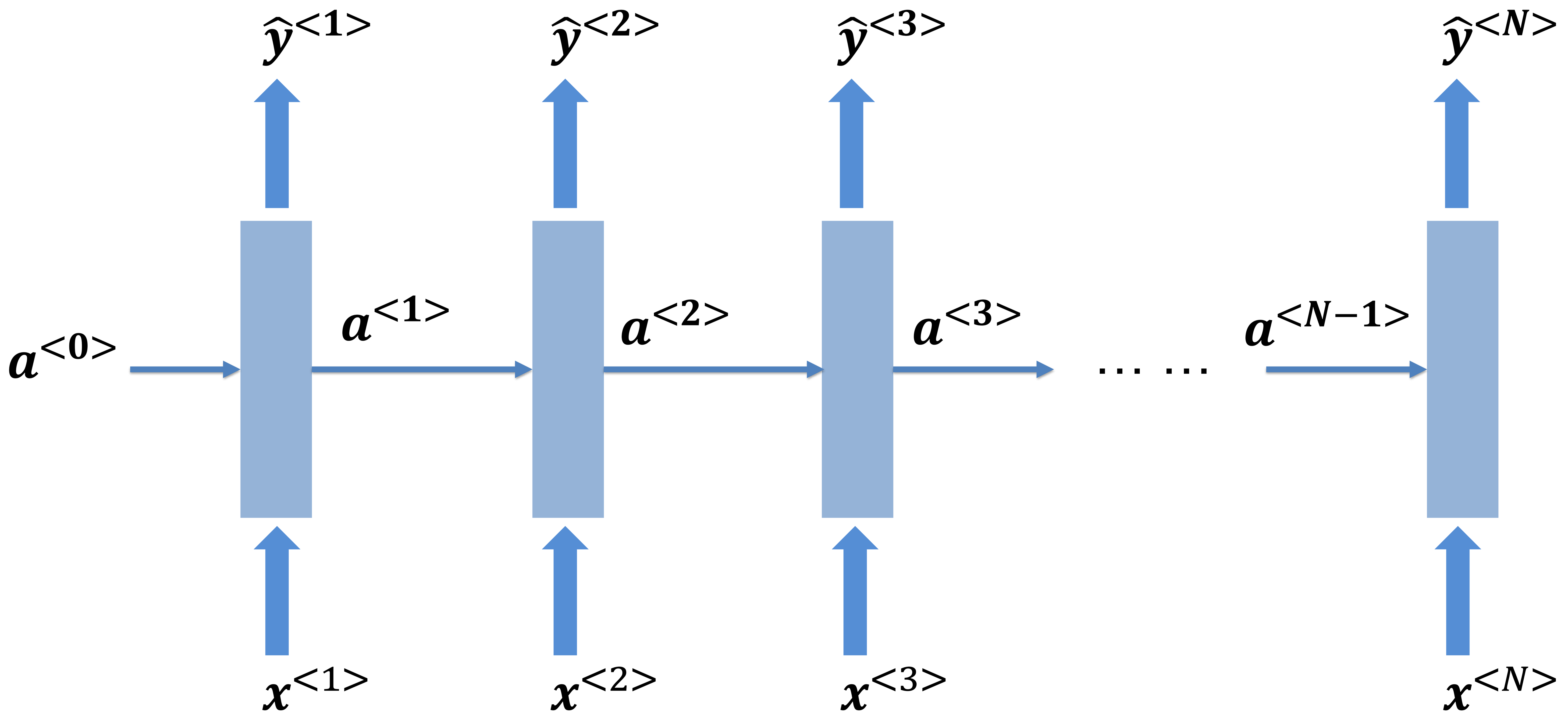
$$\frac{\partial \ell}{\partial W_{aa}} = \sum_{t=1}^{N} \frac{\partial \ell^{<t>}(y^{<t>}, \hat{y}^{<t>})}{W_{aa}}$$

$$a^{<t>} = g_a(W_{aa} \cdot a^{<t-1>} + W_{ax} \cdot x^{<t>} + b_a)$$

$$y^{<t>} = g_y(W_{ya} \cdot a^{<t>} + b_y)$$



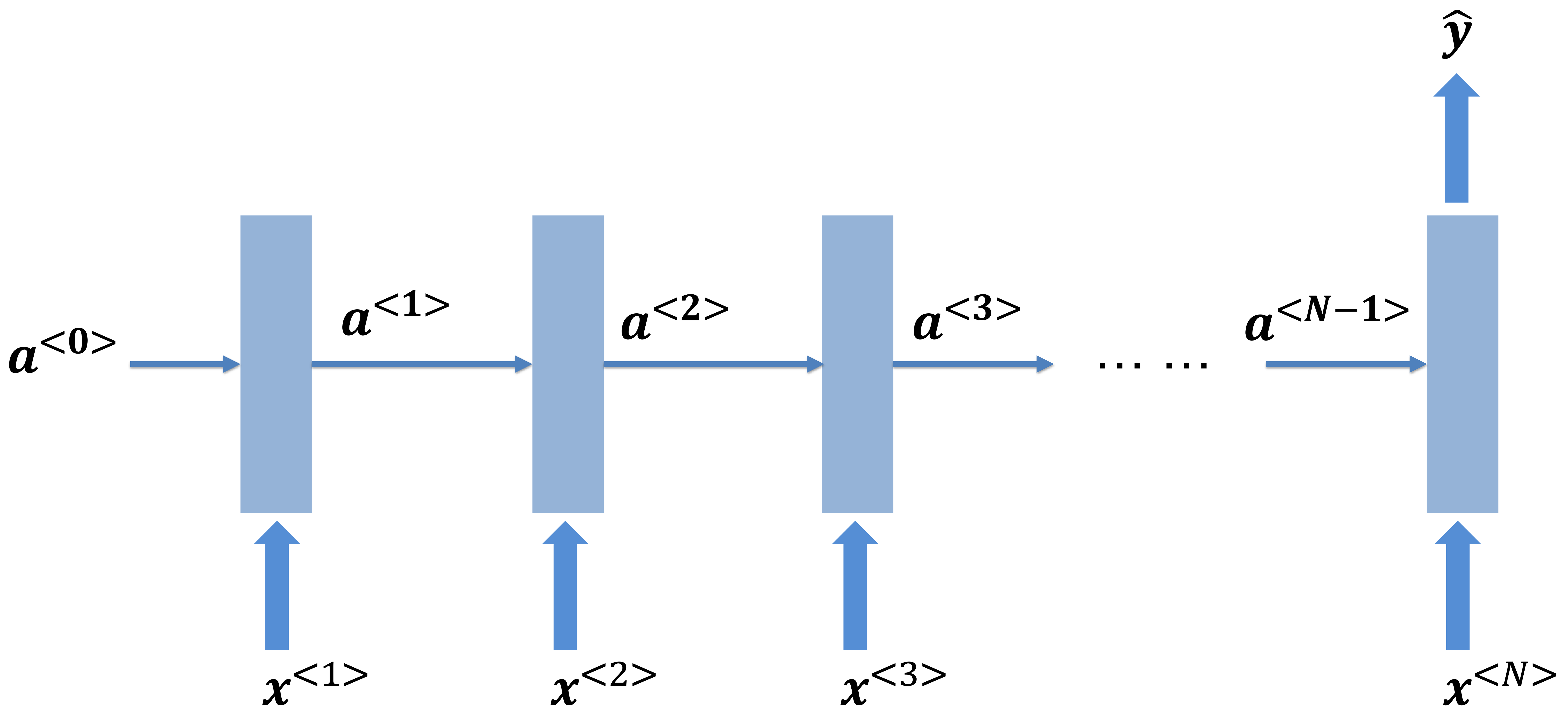
Recurrent Neural Network: variants



Many-to-many, same length

e.g. trigger word detection

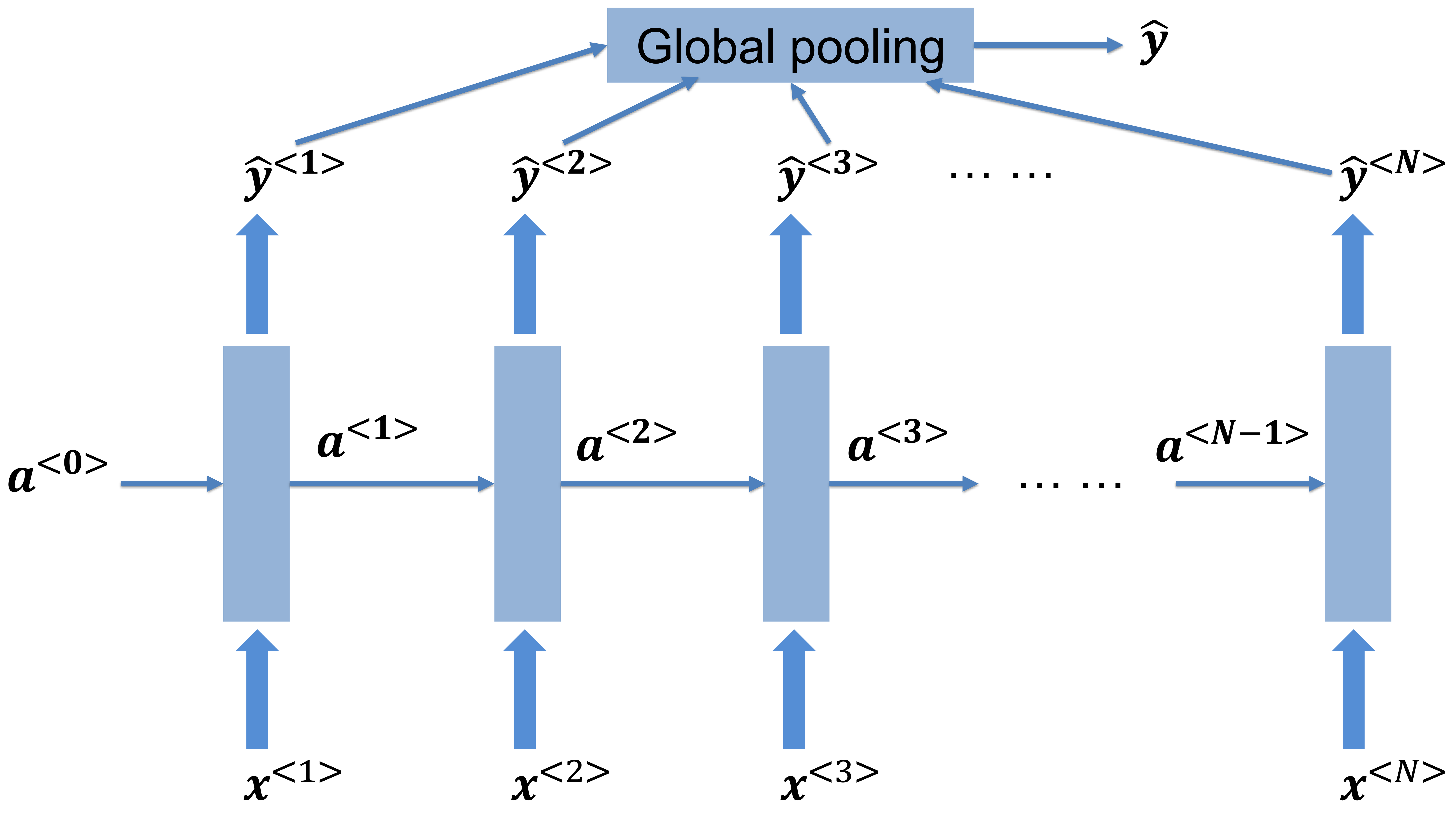
Recurrent Neural Network: variants



Many-to-one, causal

e.g. video grading system

Recurrent Neural Network: variants

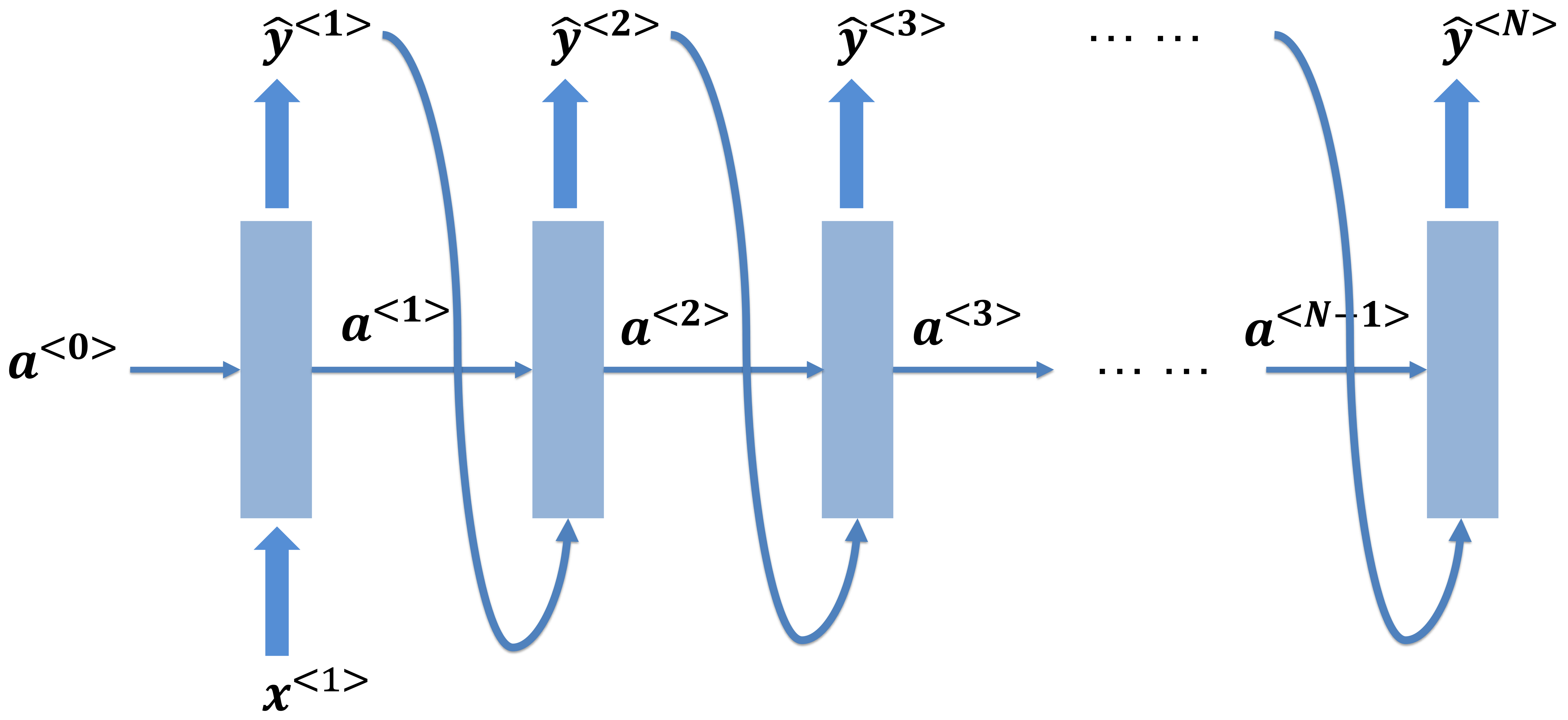


Many-to-one, non-causal

e.g. video grading system

Better balanced for all time steps

Recurrent Neural Network: variants

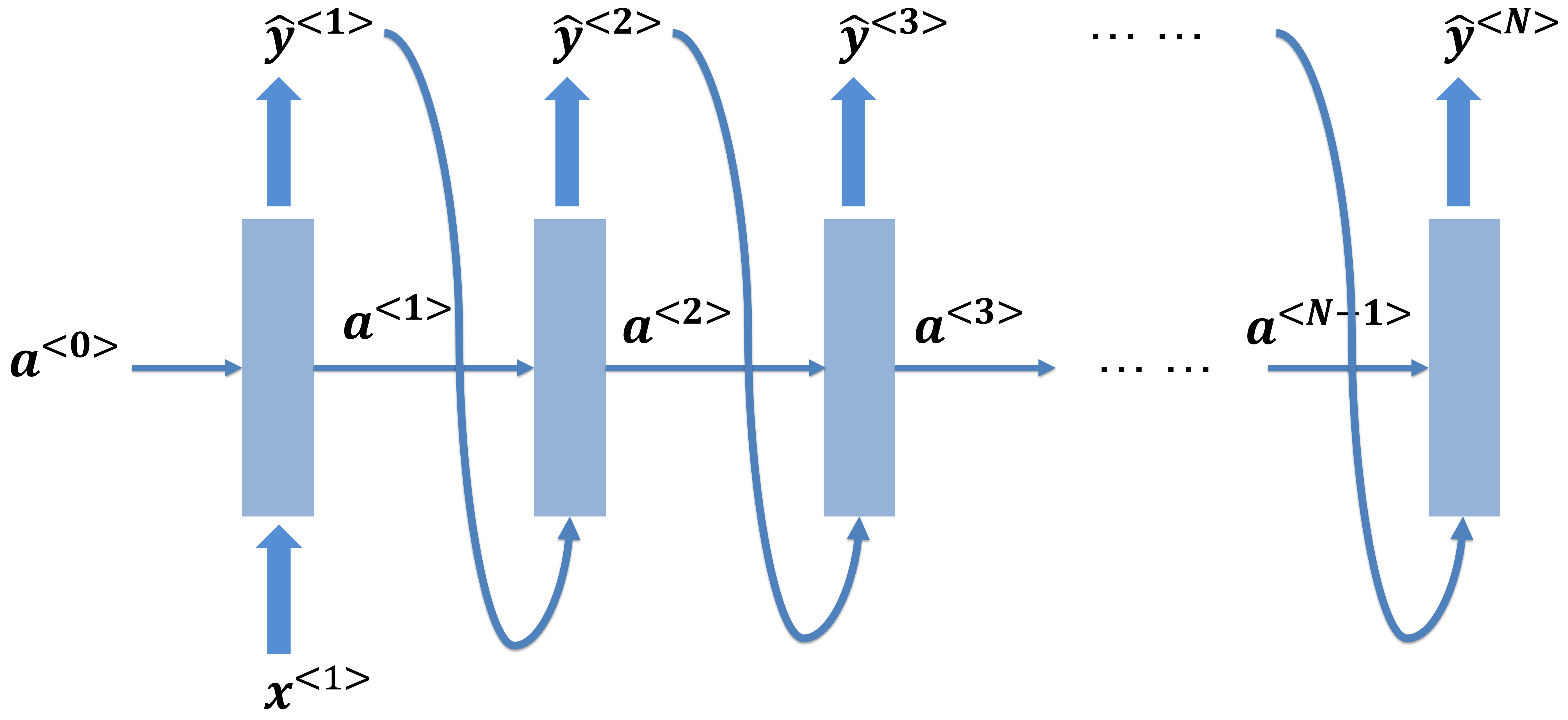


One-to-many

e.g. music or video generator, with input being a latent variable

Self-regression: Sample from distribution $\hat{y}^{<t>}$ and feed it as the next $x^{<t>}$, until <EOS> token is reached (sample from a given distribution: np.random.choice(p=prob))

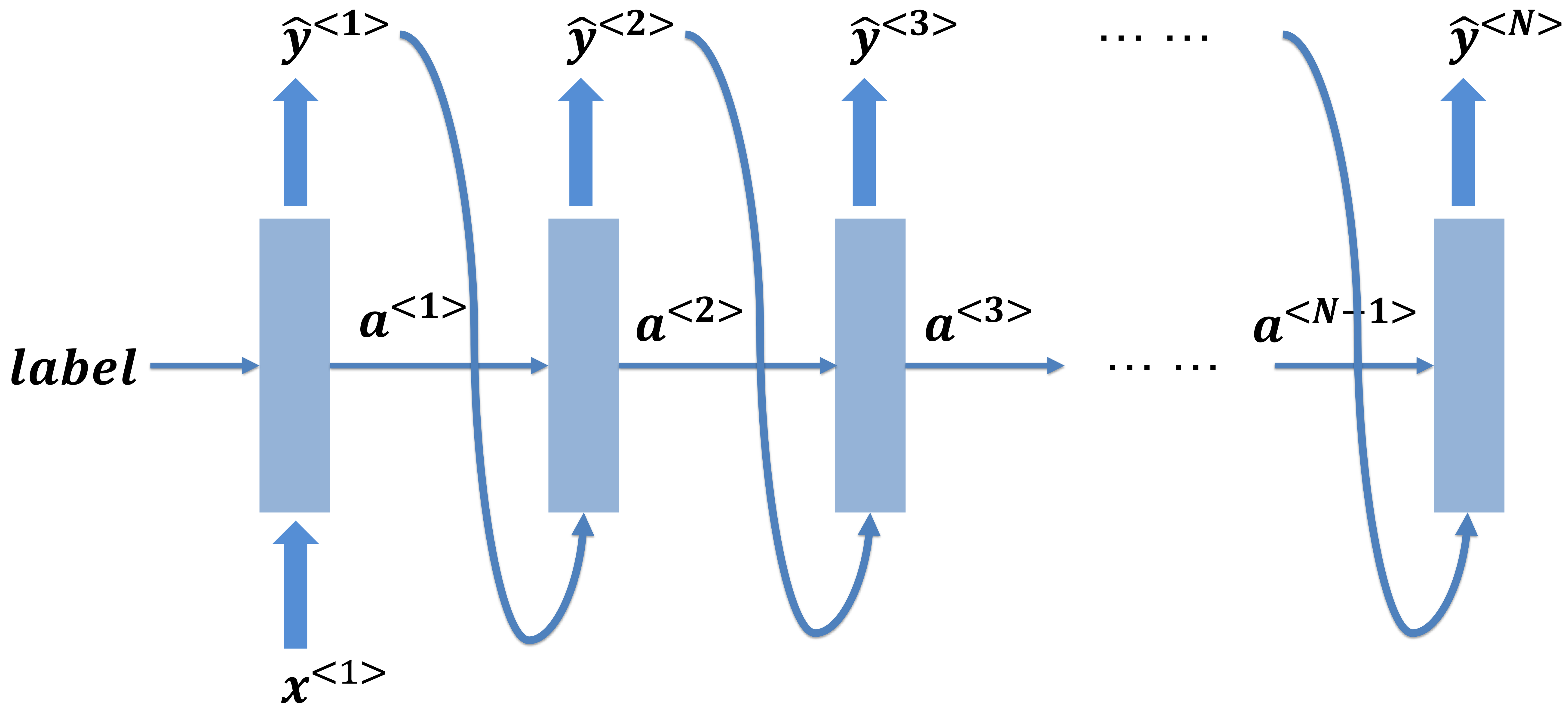
Recurrent Neural Network: variants



$$P(\hat{y}^{<1>}, \hat{y}^{<2>}, \hat{y}^{<3>}, \dots, \hat{y}^{<N>})$$
$$= P\left(\hat{y}^{<1>} \middle| x^{<1>}\right) P\left(\hat{y}^{<2>} \middle| \hat{y}^{<1>}\right) P\left(\hat{y}^{<3>} \middle| \hat{y}^{<1>}, \hat{y}^{<2>}\right) \dots P\left(\hat{y}^{<N>} \middle| \hat{y}^{<1>}, \dots, \hat{y}^{<N-1>}\right)$$

Chain rule of joint probability

Recurrent Neural Network: input label and seq



One-to-many

e.g. music, with input being a latent variable and label being its style (classical, pop, …)

Recurrent Neural Network: encoder-decoder

Many-to-many, variable length e.g. machine translation, video captioning

Recurrent Neural Network: vanishing gradient

In theory, remember all history
In reality, remember short history

$$\frac{\partial \ell^{<N>}}{\partial W_{aa}}\Big|_{t=1} = \frac{\partial \ell^{<N>}}{\partial \hat{y}^{<N>}} \frac{\partial \hat{y}^{<N>}}{\partial a^{<N>}} \frac{\partial a^{<N>}}{\partial a^{<N-1>}} \cdots \frac{\partial a^{<2>}}{\partial a^{<1>}} \frac{\partial a^{<1>}}{\partial W_{aa}}$$

$$a^{<N>} = g_a(W_{aa} \cdot a^{<N-1>} + W_{ax} \cdot x^{<N>} + b_a)$$

Recurrent Neural Network: vanishing gradient

$$\frac{\partial \ell^{<N>}}{\partial \boldsymbol{W}_{aa}}\Big|_{t=1} = \frac{\partial \ell^{<N>}}{\partial \hat{\boldsymbol{y}}^{<N>}} \frac{\partial \hat{\boldsymbol{y}}^{<N>}}{\partial \boldsymbol{a}^{<N>}} \frac{\partial \boldsymbol{a}^{<N>}}{\partial \boldsymbol{a}^{<N-1>}} \cdots \frac{\partial \boldsymbol{a}^{<2>}}{\partial \boldsymbol{a}^{<1>}} \frac{\partial \boldsymbol{a}^{<1>}}{\partial \boldsymbol{W}_{aa}}$$

$$\boldsymbol{a}^{<N>} = g_a(\boldsymbol{W}_{aa} \cdot \boldsymbol{a}^{<N-1>} + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<N>} + \boldsymbol{b}_a)$$

$$\frac{\partial \boldsymbol{a}^{<N>}}{\partial \boldsymbol{a}^{<N-1>}} = \boldsymbol{W}_{aa} \cdot g'_a\,(\boldsymbol{W}_{aa} \cdot \boldsymbol{a}^{<N-1>} + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<N>} + \boldsymbol{b}_a)$$

$$\frac{\partial \ell^{<N>}}{\partial \boldsymbol{W}_{aa}}\Big|_{t=1} = \frac{\partial \ell^{<N>}}{\partial \hat{\boldsymbol{y}}^{<N>}} \frac{\partial \hat{\boldsymbol{y}}^{<N>}}{\partial \boldsymbol{a}^{<N>}} \left[\prod_{t=2}^{N} \frac{\partial \boldsymbol{a}^{<t>}}{\partial \boldsymbol{a}^{<t-1>}}\right] \frac{\partial \boldsymbol{a}^{<1>}}{\partial \boldsymbol{W}_{aa}}$$

$$= \frac{\partial \ell^{<N>}}{\partial \hat{\boldsymbol{y}}^{<N>}} \frac{\partial \hat{\boldsymbol{y}}^{<N>}}{\partial \boldsymbol{a}^{<N>}} \boldsymbol{W}_{aa}{}^{N-1} \left[\prod_{t=2}^{N} g'_a\,(\boldsymbol{W}_{aa} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W}_{ax} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b}_a)\right] \frac{\partial \boldsymbol{a}^{<1>}}{\partial \boldsymbol{W}_{aa}}$$

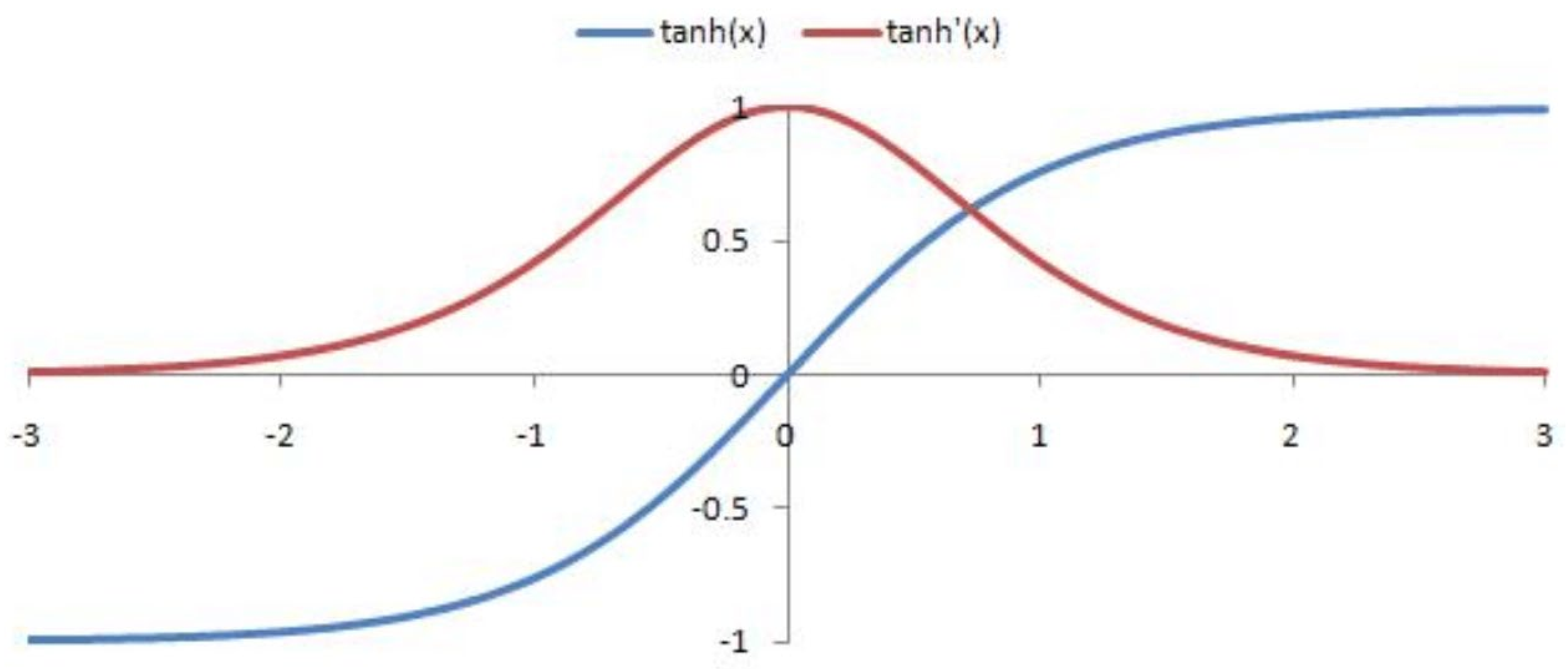
Recurrent Neural Network: vanishing gradient

$$\frac{\partial \ell^{<N>}}{\partial W_{aa}}\bigg|_{t=1} = \frac{\partial \ell^{<N>}}{\partial \hat{y}^{<N>}} \frac{\partial \hat{y}^{<N>}}{\partial a^{<N>}} {W_{aa}}^{N-1} \left[\prod_{t=2}^{N} g_a' \left(W_{aa} \cdot a^{<t-1>} + W_{ax} \cdot x^{<t>} + b_a \right) \right] \frac{\partial a^{<1>}}{\partial W_{aa}}$$

If falling into the linear regime of tanh, then minimal eigen value of W_{aa} comes into the play :

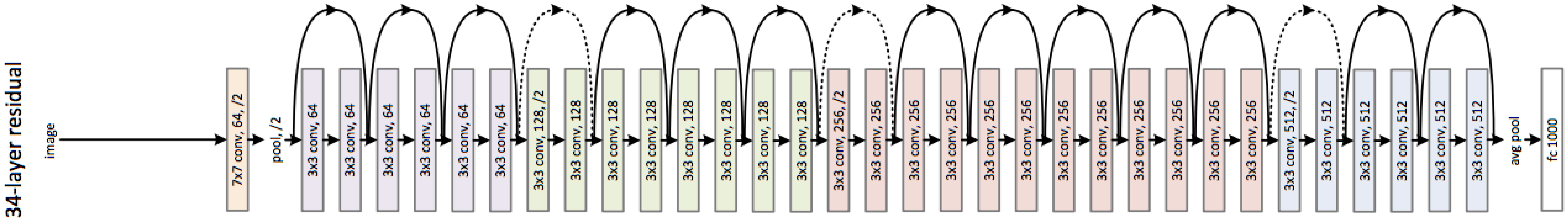
<1, vanishing gradient

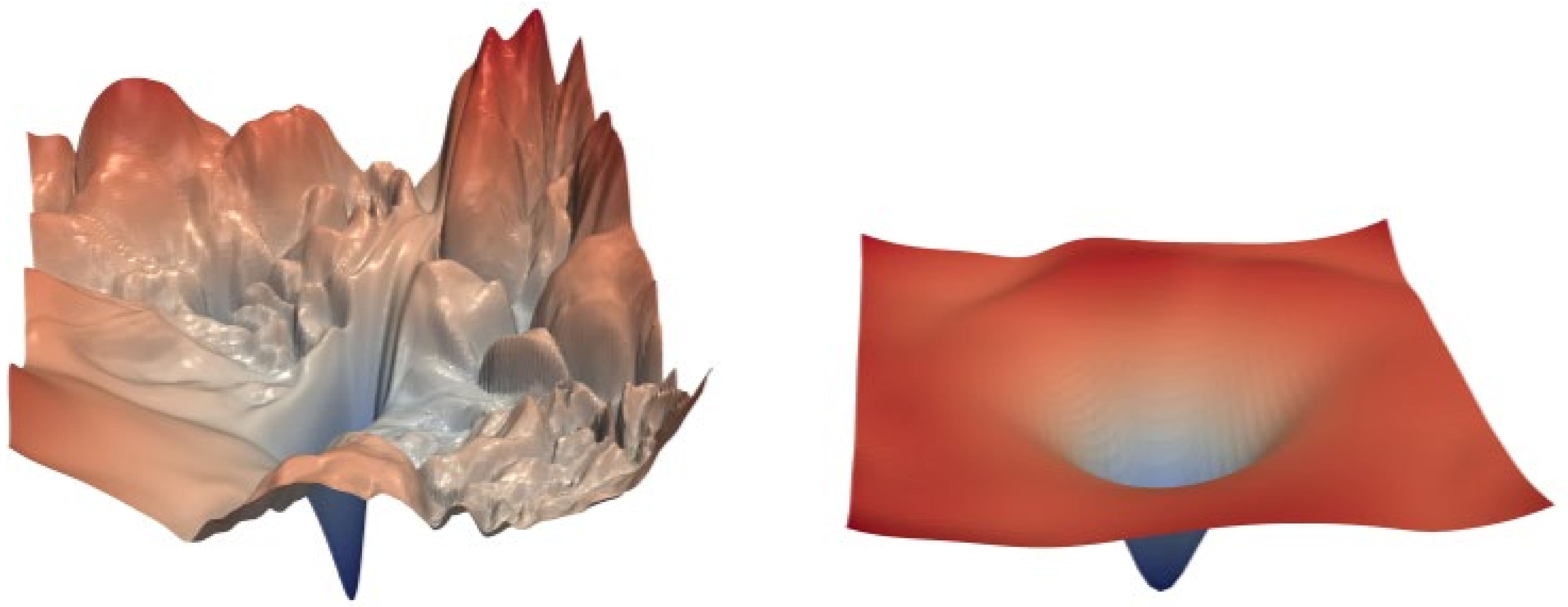
>1 exploding gradient



Often cause vanishing gradient

Fix the unstable gradient: skip connections





(a) without skip connections (b) with skip connections

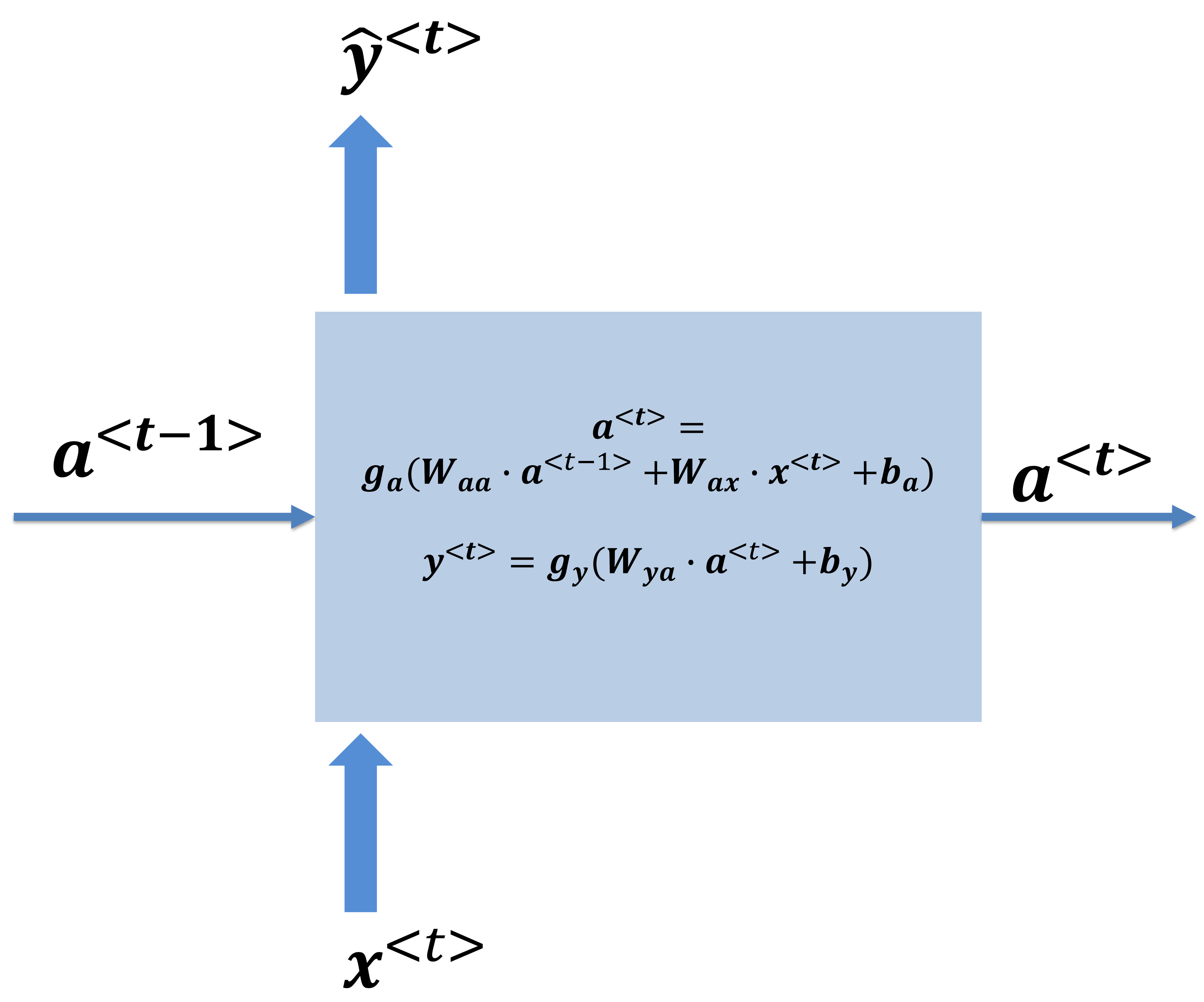
Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

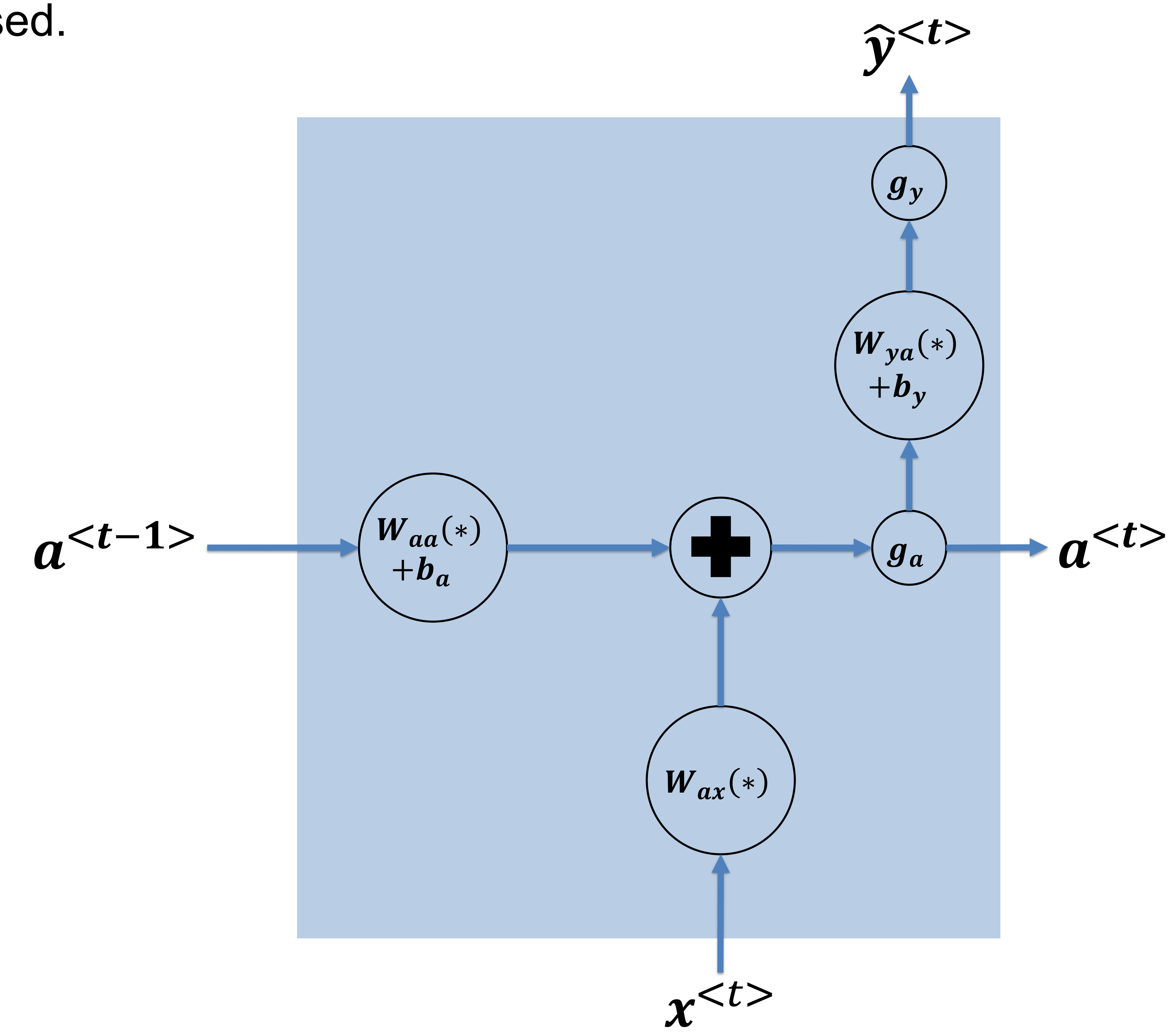
- Add skip-connections
- Allow gradient flow to pass back to earlier layers
- Network will learn to utilize these skip connections during training

Deep Residual Learning for Image Recognition. 2015. https://arxiv.org/abs/1512.03385
Visualizing the Loss Landscape of Neural Nets. https://arxiv.org/pdf/1712.09913.pdf

Invented much earlier for RNN

1995-1997: Long Short Term Memory cell (LSTM) was proposed.





LSTM can Solve Hard Long Time Lag Problems. Advances in Neural Information Processing Systems 9. 1997.

Vanilla RNN: computing cell

LSTM: introduce a memory cell

$$\boldsymbol{a}^{<t>} = \boldsymbol{g_a}(\boldsymbol{W_{aa}} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{ax}} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b_a})$$

$$\boldsymbol{y}^{<t>} = \boldsymbol{g_y}(\boldsymbol{W_{ya}} \cdot \boldsymbol{a}^{<t>} + \boldsymbol{b_y})$$

Vanilla RNN

Introduce a memory cell $\boldsymbol{c}^{<t>}$

1) Compute candidate $\tilde{\boldsymbol{c}}^{<t>}$ to update memory cell, with learnable parameters to control what to remember, and what to forget

$$\tilde{\boldsymbol{c}}^{<t>} = \boldsymbol{g_c}(\boldsymbol{W_{ca}} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{cx}} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b_c})$$

2) Add two gates to control whether to update memory cell or keep its previous states

$$\boldsymbol{c}^{<t>} = \boldsymbol{U} \circ \tilde{\boldsymbol{c}}^{<t>} + \boldsymbol{F} \circ \boldsymbol{c}^{<t-1>}$$

$\boldsymbol{F} = \boldsymbol{\sigma}(\boldsymbol{W_{fa}} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{fx}} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b_f})$: forget gate

$\boldsymbol{U} = \boldsymbol{\sigma}(\boldsymbol{W_{ua}} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{ux}} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b_u})$: update gate

LSTM: introduce a memory cell

3) Add one output gate

$$\boldsymbol{a}^{<t>} = \boldsymbol{O} \circ \boldsymbol{c}^{<t>}$$

$$\boldsymbol{O} = \boldsymbol{\sigma}(\boldsymbol{W_{oa}} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{ox}} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b_o})\text{: output gate}$$

4) Compute output

$$\boldsymbol{y}^{<t>} = \boldsymbol{g_y}(\boldsymbol{W_{ya}} \cdot \boldsymbol{a}^{<t>} + \boldsymbol{b_y})$$

or

$$\boldsymbol{y}^{<t>} = \boldsymbol{g_y}(\boldsymbol{a}^{<t>})$$

LSTM

$$\tilde{\boldsymbol{c}}^{<t>} = \boldsymbol{g_c}(\boldsymbol{W_{ca}} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{cx}} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b_c})$$

$$\boldsymbol{c}^{<t>} = \boldsymbol{U} \circ \tilde{\boldsymbol{c}}^{<t>} + \boldsymbol{F} \circ \boldsymbol{c}^{<t-1>}$$

$$\boldsymbol{F} = \boldsymbol{\sigma}(\boldsymbol{W_{fa}} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{fx}} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b_f})\text{: forget gate}$$

$$\boldsymbol{U} = \boldsymbol{\sigma}(\boldsymbol{W_{ua}} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{ux}} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b_u})\text{: update gate}$$

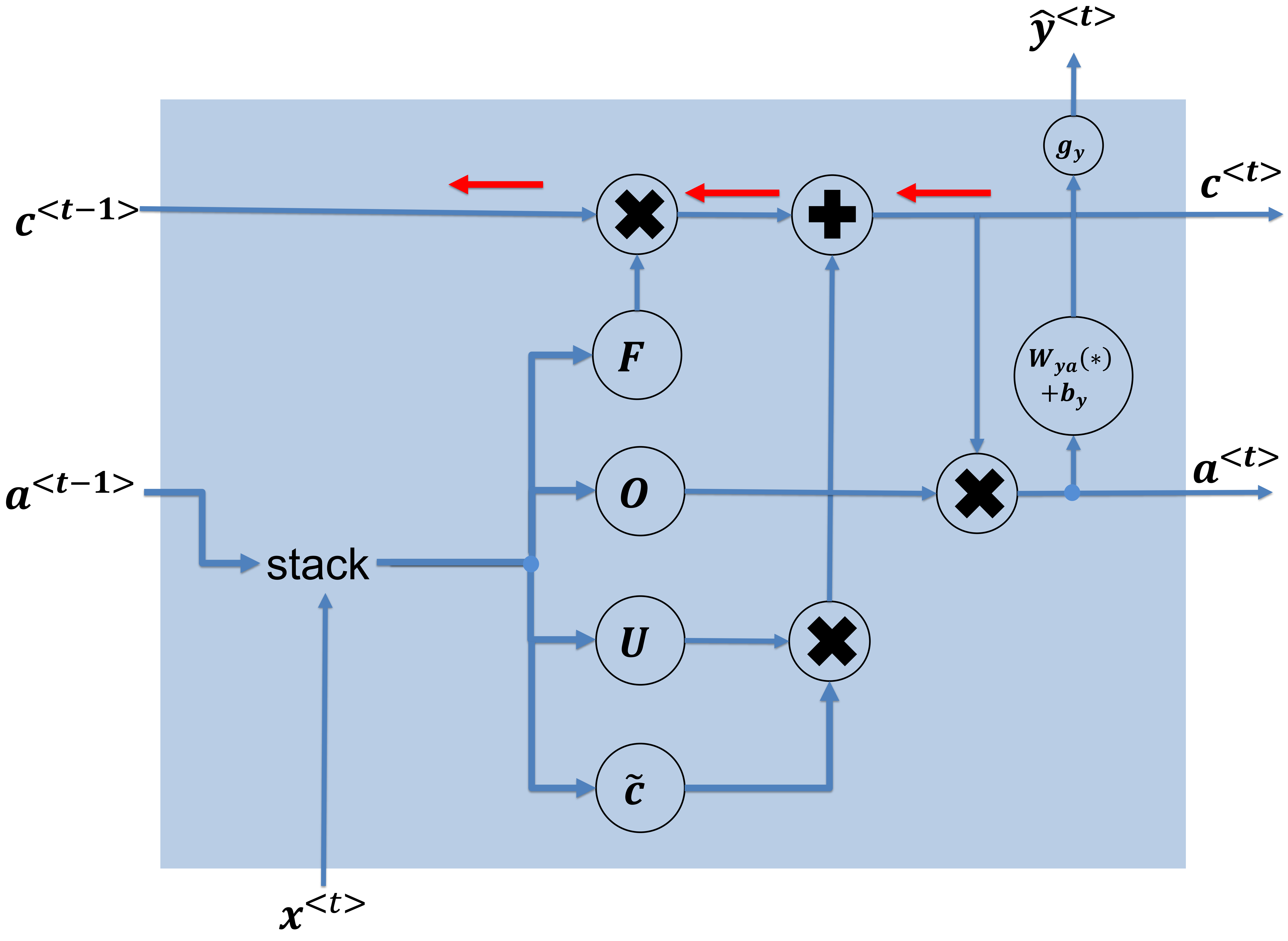
$$\boldsymbol{a}^{<t>} = \boldsymbol{O} \circ \boldsymbol{c}^{<t>}$$

$$\boldsymbol{O} = \boldsymbol{\sigma}(\boldsymbol{W_{oa}} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{ox}} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b_o})\text{: output gate}$$

$$\boldsymbol{y}^{<t>} = \boldsymbol{g_y}(\boldsymbol{W_{ya}} \cdot \boldsymbol{a}^{<t>} + \boldsymbol{b_y})$$

$\boldsymbol{F}$, $\boldsymbol{U}$, $\boldsymbol{O}$ are vectors, same size as memory cell
$\boldsymbol{U} \circ \tilde{\boldsymbol{c}}^{<t>}$, $\boldsymbol{F} \circ \boldsymbol{c}^{<t-1>}$, $\boldsymbol{O} \circ \boldsymbol{c}^{<t>}$: element-wise multiplication

LSTM: computing graph



LSTM

$$\tilde{c}^{<t>} = g_c(W_{ca} \cdot a^{<t-1>} + W_{cx} \cdot x^{<t>} + b_c)$$

$$c^{<t>} = U \circ \tilde{c}^{<t>} + F \circ c^{<t-1>}$$

$F = \sigma(W_{fa} \cdot a^{<t-1>} + W_{fx} \cdot x^{<t>} + b_f)$: forget gate

$U = \sigma(W_{ua} \cdot a^{<t-1>} + W_{ux} \cdot x^{<t>} + b_u)$: update gate

$$a^{<t>} = O \circ c^{<t>}$$

$O = \sigma(W_{oa} \cdot a^{<t-1>} + W_{ox} \cdot x^{<t>} + b_o)$: output gate

$$y^{<t>} = g_y(W_{ya} \cdot a^{<t>} + b_y)$$

LSTM can Solve Hard Long Time Lag Problems. Advances in Neural Information Processing Systems 9. 1997.

LSTM: peephole connection

Add the memory cell state to the computation of gates

$$\boldsymbol{F} = \boldsymbol{\sigma}(\boldsymbol{W}_{fa} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W}_{fx} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b}_f)\text{: forget gate}$$

$$\boldsymbol{U} = \boldsymbol{\sigma}(\boldsymbol{W}_{ua} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W}_{ux} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b}_u)\text{: update gate}$$

$$\boldsymbol{O} = \boldsymbol{\sigma}(\boldsymbol{W}_{oa} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W}_{ox} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{b}_o)\text{: output gate}$$

$$\Downarrow$$

$$\boldsymbol{F} = \boldsymbol{\sigma}(\boldsymbol{W}_{fa} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W}_{fx} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{W}_{fc} \cdot \boldsymbol{c}^{<t-1>} + \boldsymbol{b}_f)\text{: forget gate}$$

$$\boldsymbol{U} = \boldsymbol{\sigma}(\boldsymbol{W}_{ua} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W}_{ux} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{W}_{uc} \cdot \boldsymbol{c}^{<t-1>} + \boldsymbol{b}_u)\text{: update gate}$$

$$\boldsymbol{O} = \boldsymbol{\sigma}(\boldsymbol{W}_{oa} \cdot \boldsymbol{a}^{<t-1>} + \boldsymbol{W}_{ox} \cdot \boldsymbol{x}^{<t>} + \boldsymbol{W}_{oc} \cdot \boldsymbol{c}^{<t-1>} + \boldsymbol{b}_o)\text{: output gate}$$

LSTM: convolution version, convLSTM

Replace linear matrix multiplication with the convolution and flatten/reshape

$$\boldsymbol{F} = \boldsymbol{\sigma}(\boldsymbol{W_{fa}} * \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{fx}} * \boldsymbol{x}^{<t>} + \boldsymbol{W_{fc}} \cdot \boldsymbol{c}^{<t-1>} + \boldsymbol{b_f})\text{: forget gate}$$

$$\boldsymbol{U} = \boldsymbol{\sigma}(\boldsymbol{W_{ua}} * \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{ux}} * \boldsymbol{x}^{<t>} + \boldsymbol{W_{uc}} \cdot \boldsymbol{c}^{<t-1>} + \boldsymbol{b_u})\text{: update gate}$$

$$\boldsymbol{O} = \boldsymbol{\sigma}(\boldsymbol{W_{oa}} * \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{ox}} * \boldsymbol{x}^{<t>} + \boldsymbol{W_{oc}} \cdot \boldsymbol{c}^{<t-1>} + \boldsymbol{b_o})\text{: output gate}$$

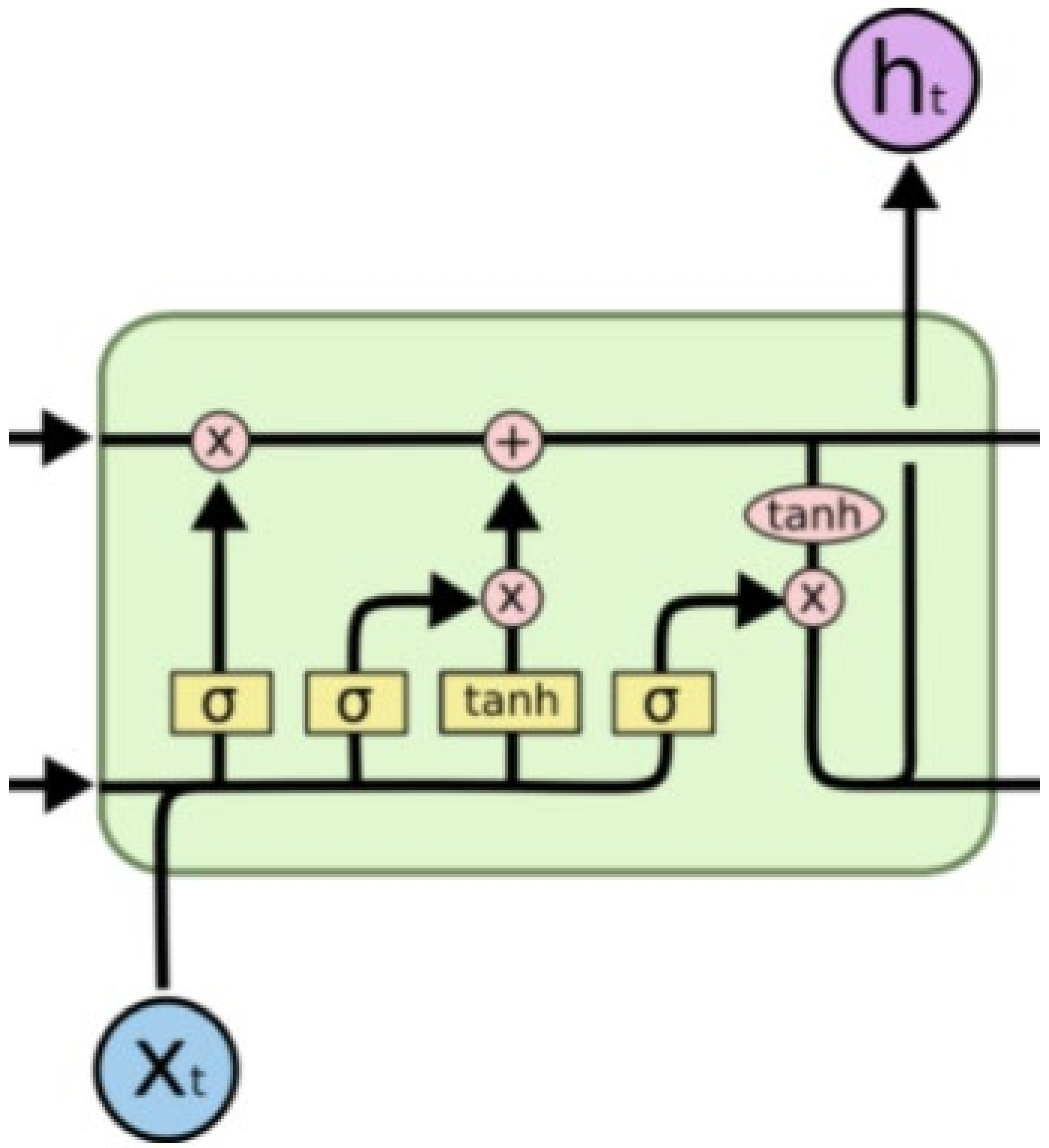
$$\tilde{\boldsymbol{c}}^{<t>} = \boldsymbol{g_c}(\boldsymbol{W_{ca}} * \boldsymbol{a}^{<t-1>} + \boldsymbol{W_{cx}} * \boldsymbol{x}^{<t>} + \boldsymbol{b_c})$$

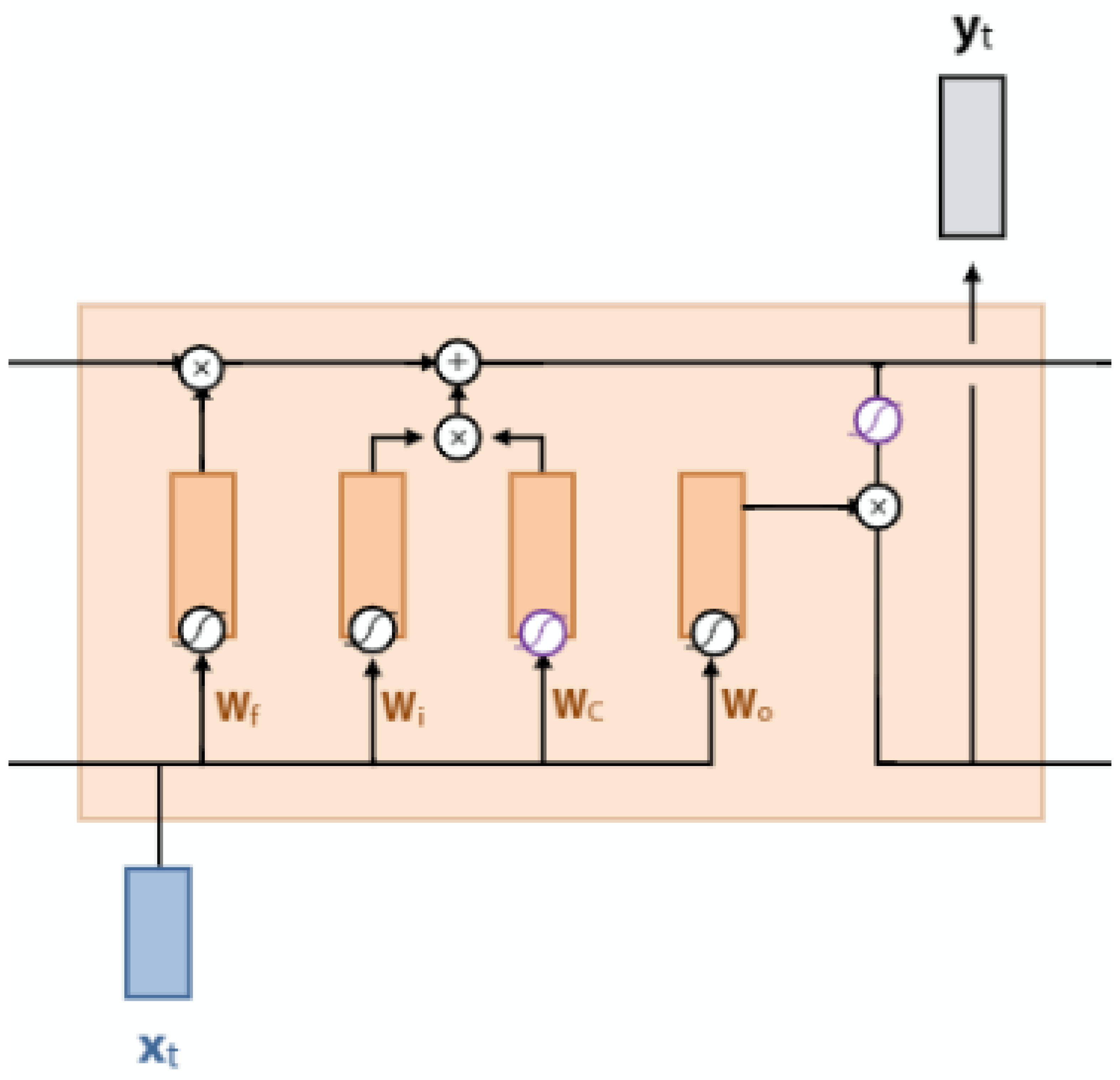
$$\boldsymbol{c}^{<t>} = \boldsymbol{U} \circ \tilde{\boldsymbol{c}}^{<t>} + \boldsymbol{F} \circ \boldsymbol{c}^{<t-1>}$$

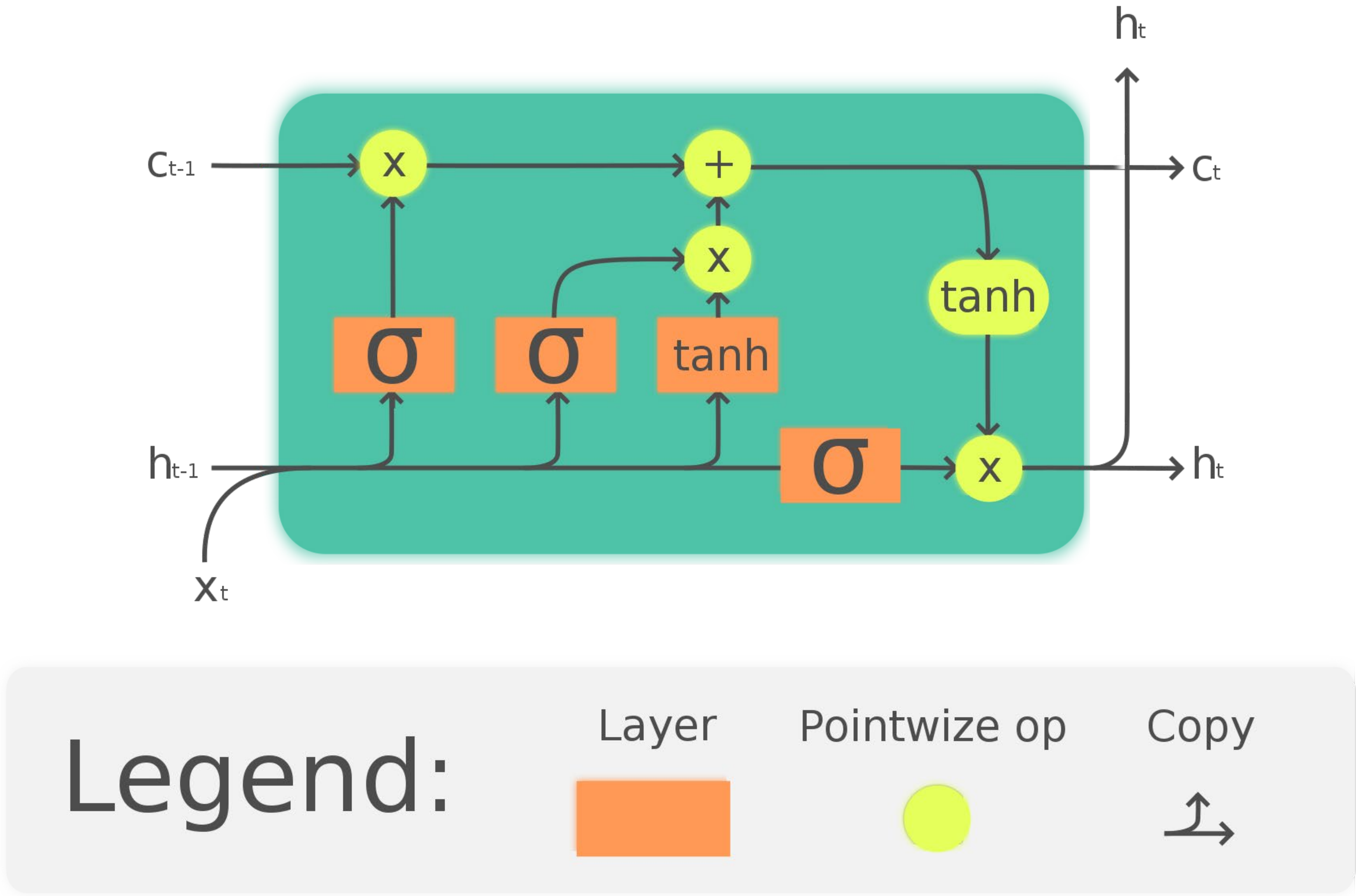
$\boldsymbol{a}^{<t>}$, $\boldsymbol{x}^{<t>}$ are images
$*$: image convolution
After convolution, results are flattened to be a vector

https://papers.nips.cc/paper/2015/file/07563a3fe3bbe7e3ba84431ad9d055af-Paper.pdf https://towardsdatascience.com/video-prediction-using-convlstm-with-pytorch-lightning-27b195fd21a2

LSTM: many faces







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

https://github.com/dlvu/dlvu.github.io/blob/main/slides/dlvu.lecture05.pdf

https://en.wikipedia.org/wiki/Long_short-term_memory#cite_note-43

GRU: Gated Recurrent Unit

GRU

$\tilde{c}^{<t>} = g_c(W_{ca} \cdot R \circ a^{<t-1>} + W_{cx} \cdot x^{<t>} + b_c)$

$c^{<t>} = U \circ \tilde{c}^{<t>} + (1 - U) \circ c^{<t-1>}$

$F = 1 - U$: forget gate

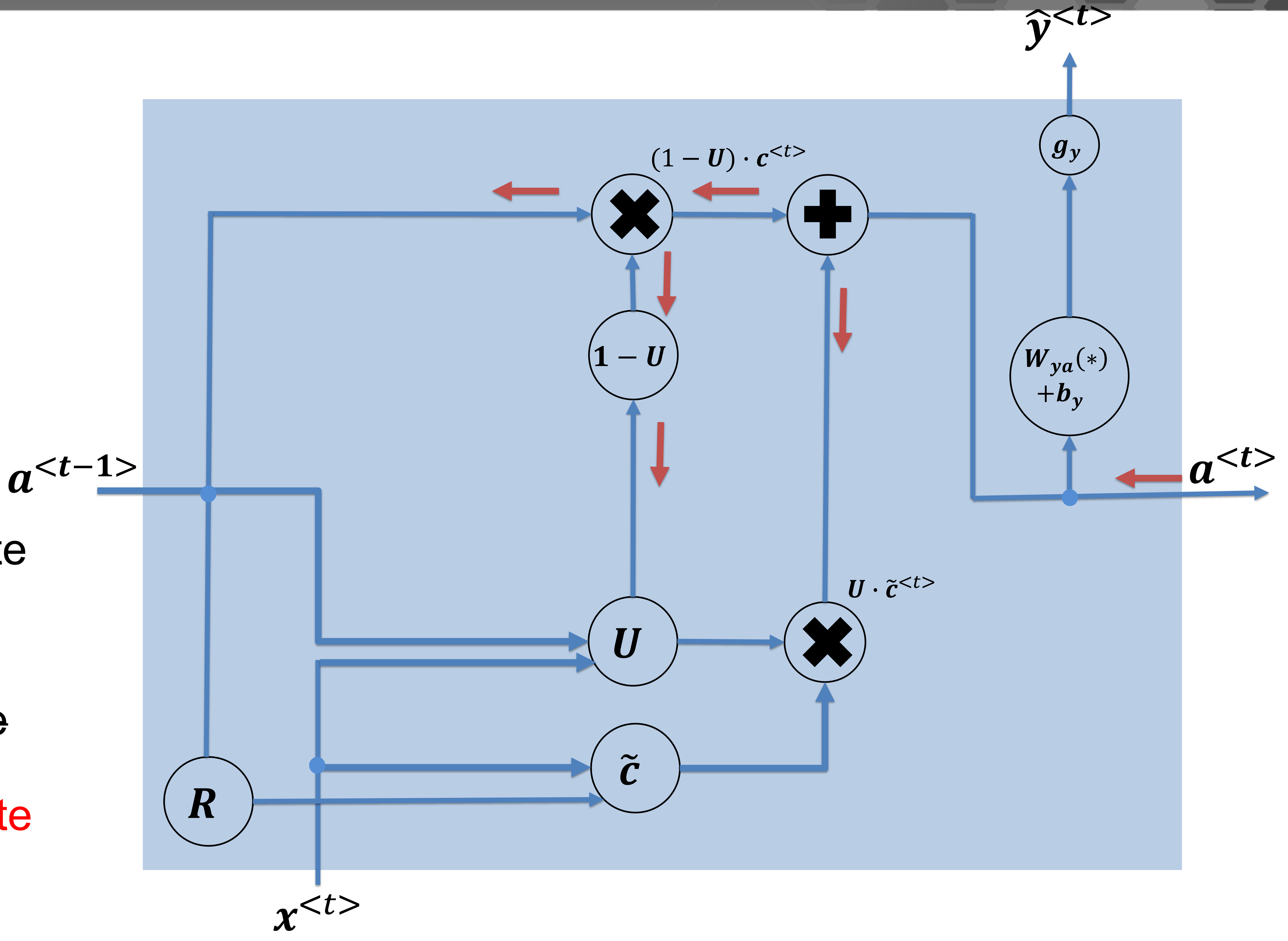
$U = \sigma(W_{ua} \cdot a^{<t-1>} + W_{ux} \cdot x^{<t>} + b_u)$: update gate

$a^{<t>} = c^{<t>}$

~~$O = \sigma(W_{oa} \cdot a^{<t-1>} + W_{ox} \cdot x^{<t>} + b_o)$: output gate~~

$R = \sigma(W_{ra} \cdot a^{<t-1>} + W_{rx} \cdot x^{<t>} + b_r)$: relevant gate

$y^{<t>} = g_y(W_{ya} \cdot a^{<t>} + b_y)$



Learning phrase representations using RNN encoder-decoder for statistical machine translation. 2014. https://arxiv.org/abs/1406.1078

Other variants

MUT1:

$$
\begin{aligned}
z &= \mathrm{sigm}(W_{\mathrm{xz}} x_t + b_{\mathrm{z}}) \\
r &= \mathrm{sigm}(W_{\mathrm{xr}} x_t + W_{\mathrm{hr}} h_t + b_{\mathrm{r}}) \\
h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + \tanh(x_t) + b_{\mathrm{h}}) \odot z \\
&+ h_t \odot (1 - z)
\end{aligned}
$$

MUT2:

$$
\begin{aligned}
z &= \mathrm{sigm}(W_{\mathrm{xz}} x_t + W_{\mathrm{hz}} h_t + b_{\mathrm{z}}) \\
r &= \mathrm{sigm}(x_t + W_{\mathrm{hr}} h_t + b_{\mathrm{r}}) \\
h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + W_{xh} x_t + b_{\mathrm{h}}) \odot z \\
&+ h_t \odot (1 - z)
\end{aligned}
$$

MUT3:

$$
\begin{aligned}
z &= \mathrm{sigm}(W_{\mathrm{xz}} x_t + W_{\mathrm{hz}} \tanh(h_t) + b_{\mathrm{z}}) \\
r &= \mathrm{sigm}(W_{\mathrm{xr}} x_t + W_{\mathrm{hr}} h_t + b_{\mathrm{r}}) \\
h_{t+1} &= \tanh(W_{\mathrm{hh}}(r \odot h_t) + W_{xh} x_t + b_{\mathrm{h}}) \odot z \\
&+ h_t \odot (1 - z)
\end{aligned}
$$

- More than ten thousand architectures
- Different sequence tasks
- “We have evaluated a variety of recurrent neural network architectures in order to find an architecture that reliably outperforms the LSTM. **Though there were architectures that outperformed the LSTM on some problems, we were unable to find an architecture that consistently beat the LSTM and the GRU in all experimental conditions**.”

An Empirical Exploration of Recurrent Network Architectures. 2015. http://proceedings.mlr.press/v37/jozefowicz15.pdf

Auto ML to find RNN cell

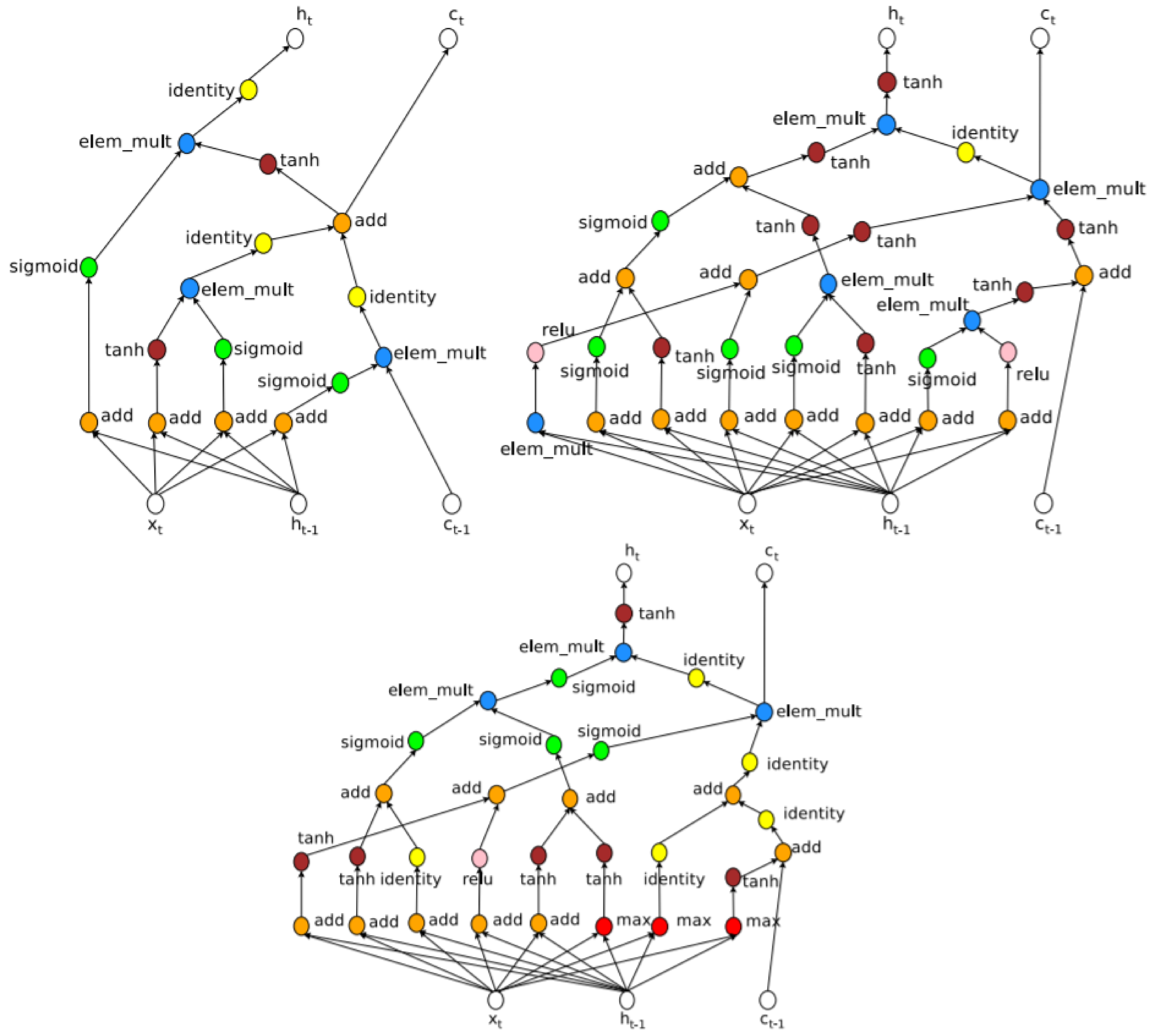
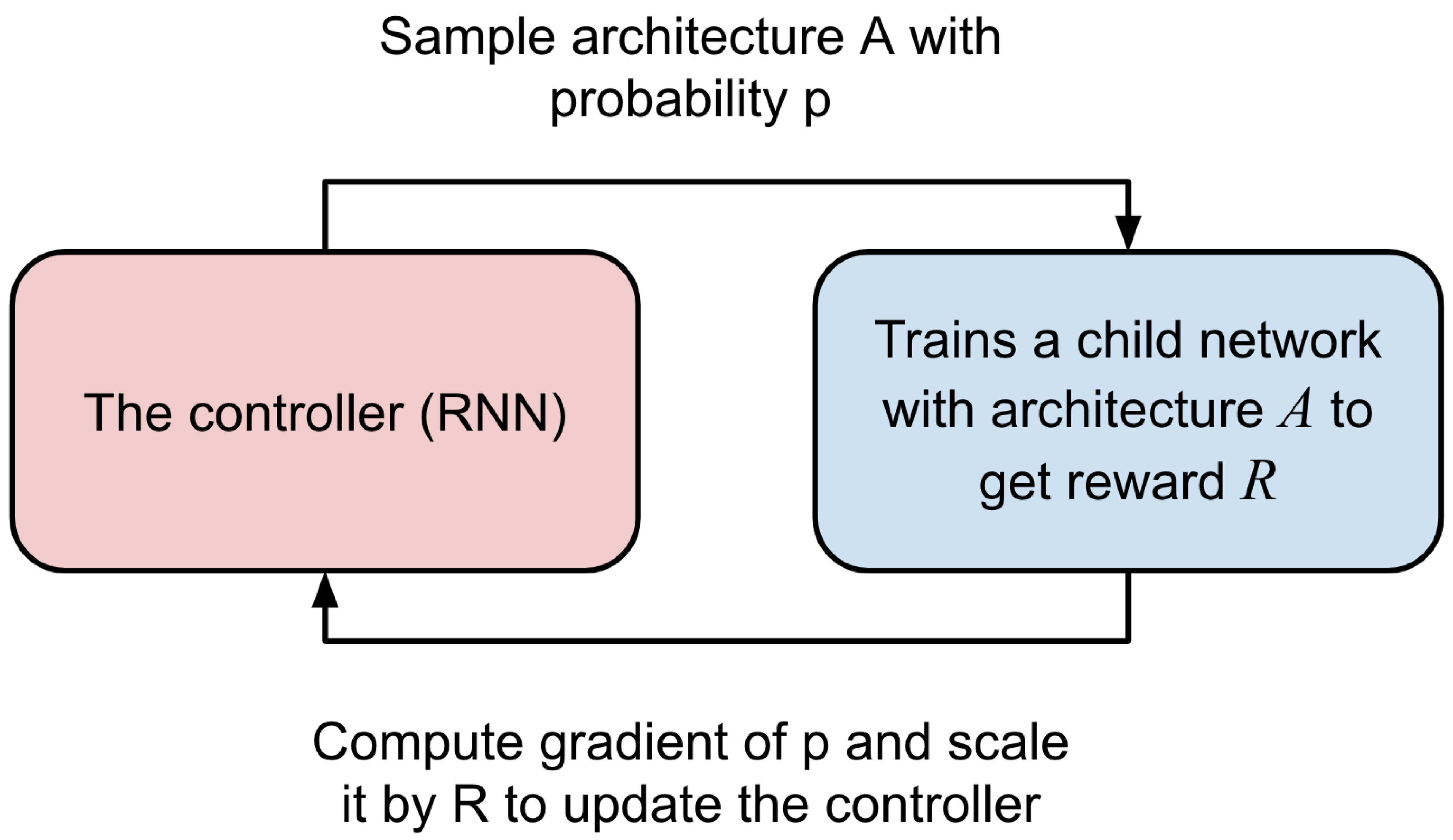


Figure 8: A comparison of the original LSTM cell vs. two good cells our model found. Top left: LSTM cell. Top right: Cell found by our model when the search space does not include max and sin. Bottom: Cell found by our model when the search space includes max and sin (the controller did not choose to use the sin function).

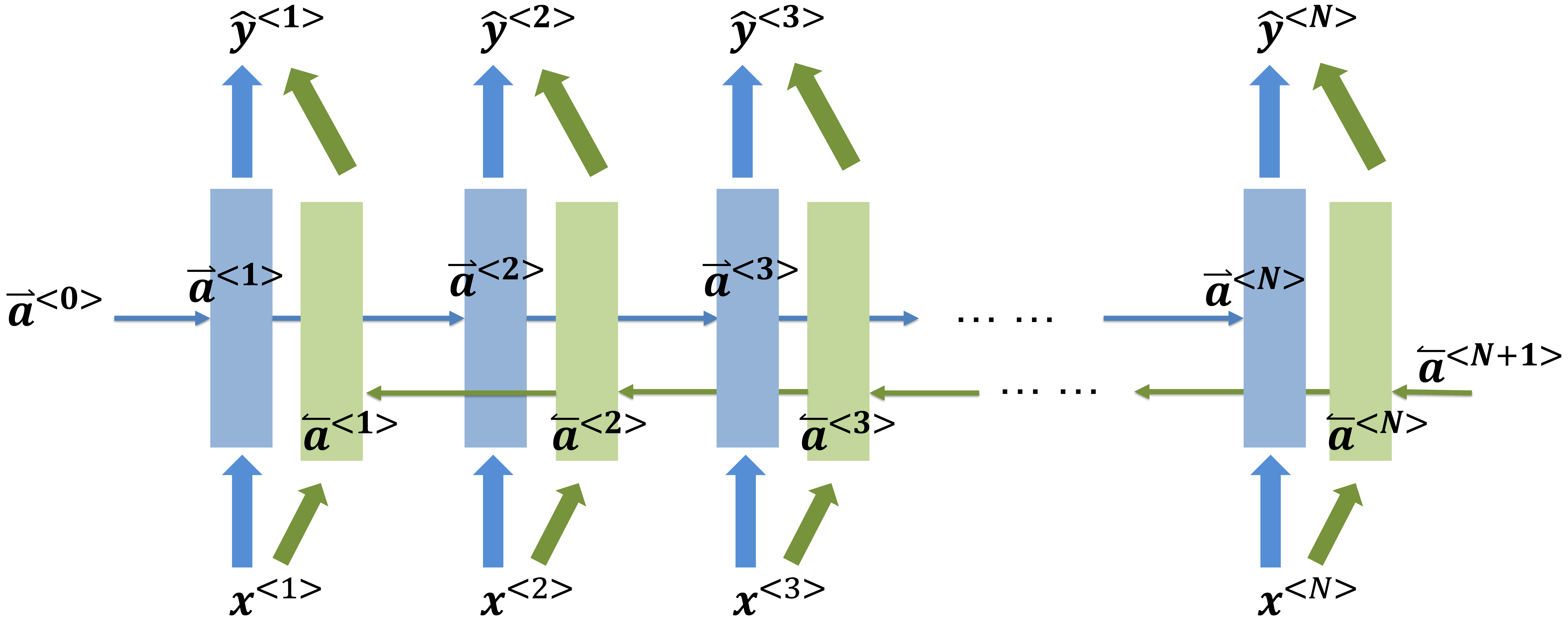
- Require large computing resource
- Can find better architecture on given tasks



https://lilianweng.github.io/lil-log/2020/08/06/neural-architecture-search.html

Neural Architecture Search with Reinforcement Learning. 2017.

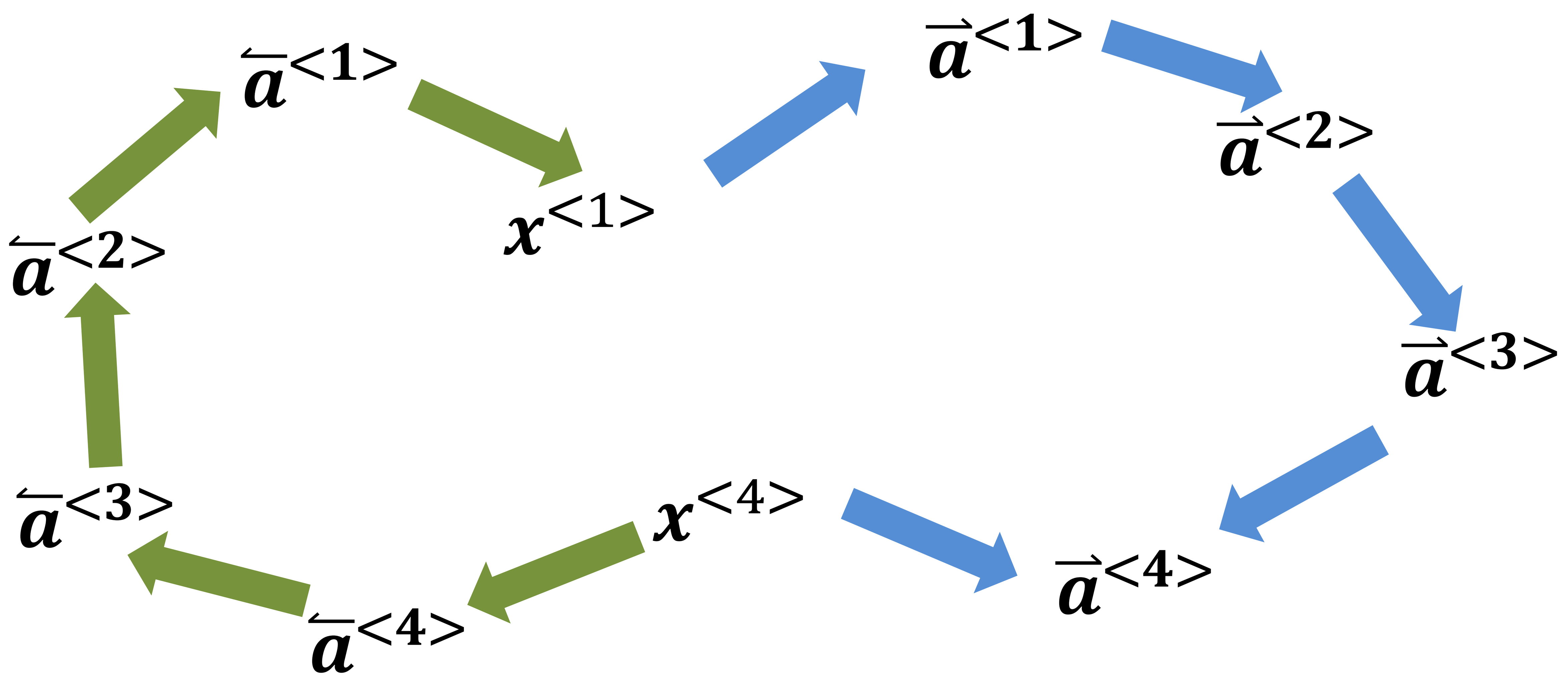
Bidirectional RNN (BRNN)



- Some applications provide the entire series, e.g. machine translation
- Or, the deployment does not require real-time processing
- Allow utilizing information from the past and the future

$$y^{<t>} = g_y(W_{ya} \cdot [\vec{a}^{<t>}; \overleftarrow{a}^{<t>}] + b_y) \quad \text{or} \quad y^{<t>} = g_y(W_{ya} \cdot (\vec{a}^{<t>} + \overleftarrow{a}^{<t>}) + b_y)$$

Bidirectional RNN (BRNN)



Two independent RNN put together

Equivalent to run two RNNs; one is fed with t=1,…,N and another is fed with t=N,N-1, …, 1

- Still acyclic graph
- Can run forward-pass and backprop
- Supported by PyTorch or Keras

CLASS `torch.nn.GRU(*args, **kwargs)` [SOURCE]

Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$\begin{aligned} r_t &= \sigma(W_{ir} x_t + b_{ir} + W_{hr} h_{(t-1)} + b_{hr}) \\ z_t &= \sigma(W_{iz} x_t + b_{iz} + W_{hz} h_{(t-1)} + b_{hz}) \\ n_t &= \tanh(W_{in} x_t + b_{in} + r_t * (W_{hn} h_{(t-1)} + b_{hn})) \\ h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)} \end{aligned}$$

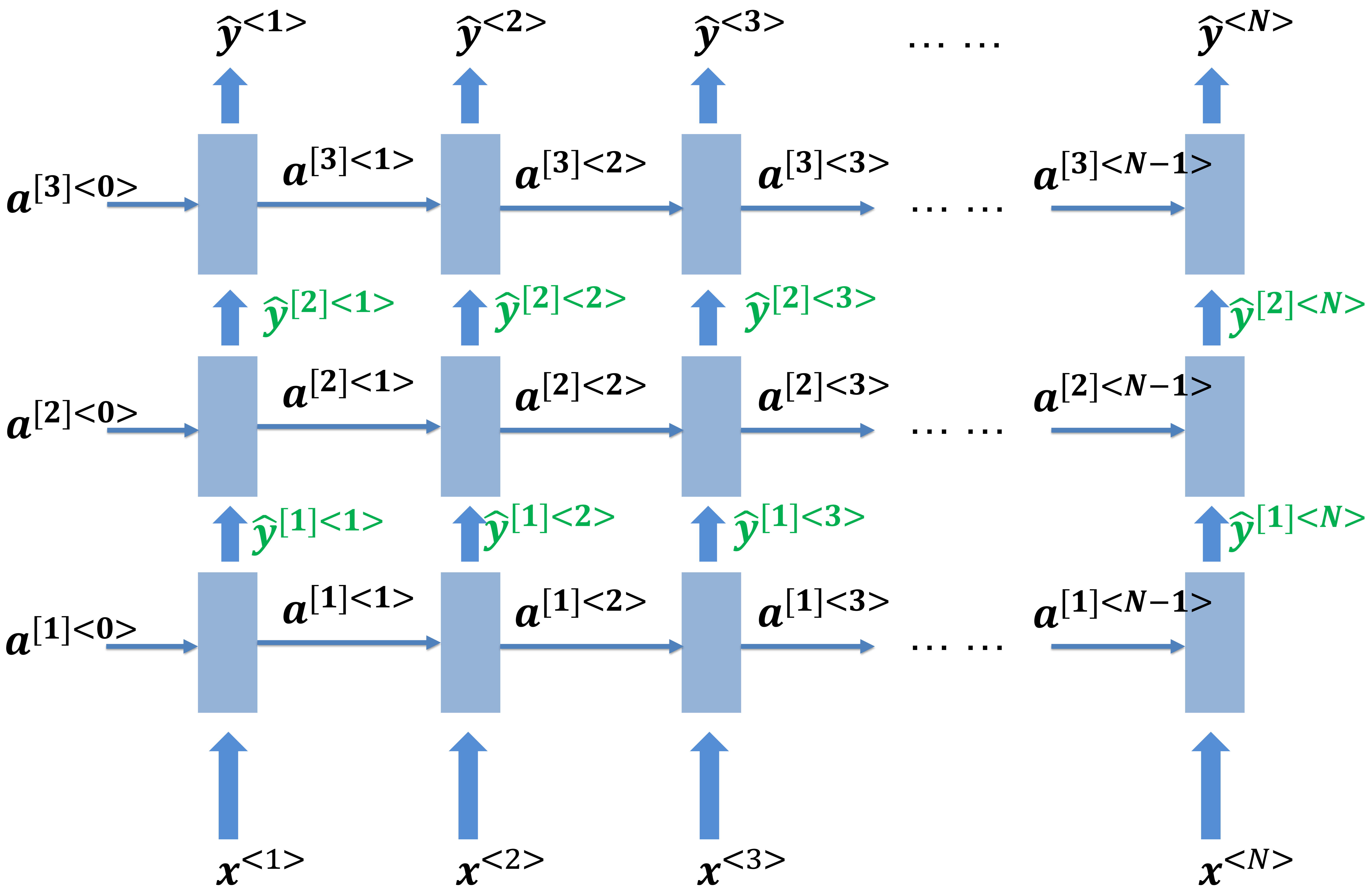
where h_t is the hidden state at time *t*, x_t is the input at time *t*, $h_{(t-1)}$ is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and r_t , z_t , n_t are the reset, update, and new gates, respectively. σ is the sigmoid function, and $*$ is the Hadamard product.

In a multilayer GRU, the input $x^{(l)}_t$ of the l -th layer ($l >= 2$) is the hidden state $h^{(l-1)}_t$ of the previous layer multiplied by dropout $\delta^{(l-1)}_t$ where each $\delta^{(l-1)}_t$ is a Bernoulli random variable which is 0 with probability `dropout`.

Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two GRUs together to form a *stacked GRU*, with the second GRU taking in outputs of the first GRU and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (batch, seq, feature). Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each GRU layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional GRU. Default: `False`

Multi-layer RNN or deep RNN



$$a^{[l]<t>} = g_a(W_{aa}^{[l]} \cdot a^{[l]<t-1>} + W_{ax}^{[l]} \cdot \hat{y}^{[l-1]<t>} + b_a^{[l]})$$

$$y^{[l]<t>} = g_y(W_{ya}^{[l]} \cdot a^{[l]<t>} + b_y^{[l]})$$

CLASS `torch.nn.GRU(*args, **kwargs)` [SOURCE]

Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$\begin{aligned} r_t &= \sigma(W_{ir} x_t + b_{ir} + W_{hr} h_{(t-1)} + b_{hr}) \\ z_t &= \sigma(W_{iz} x_t + b_{iz} + W_{hz} h_{(t-1)} + b_{hz}) \\ n_t &= \tanh(W_{in} x_t + b_{in} + r_t * (W_{hn} h_{(t-1)} + b_{hn})) \\ h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)} \end{aligned}$$

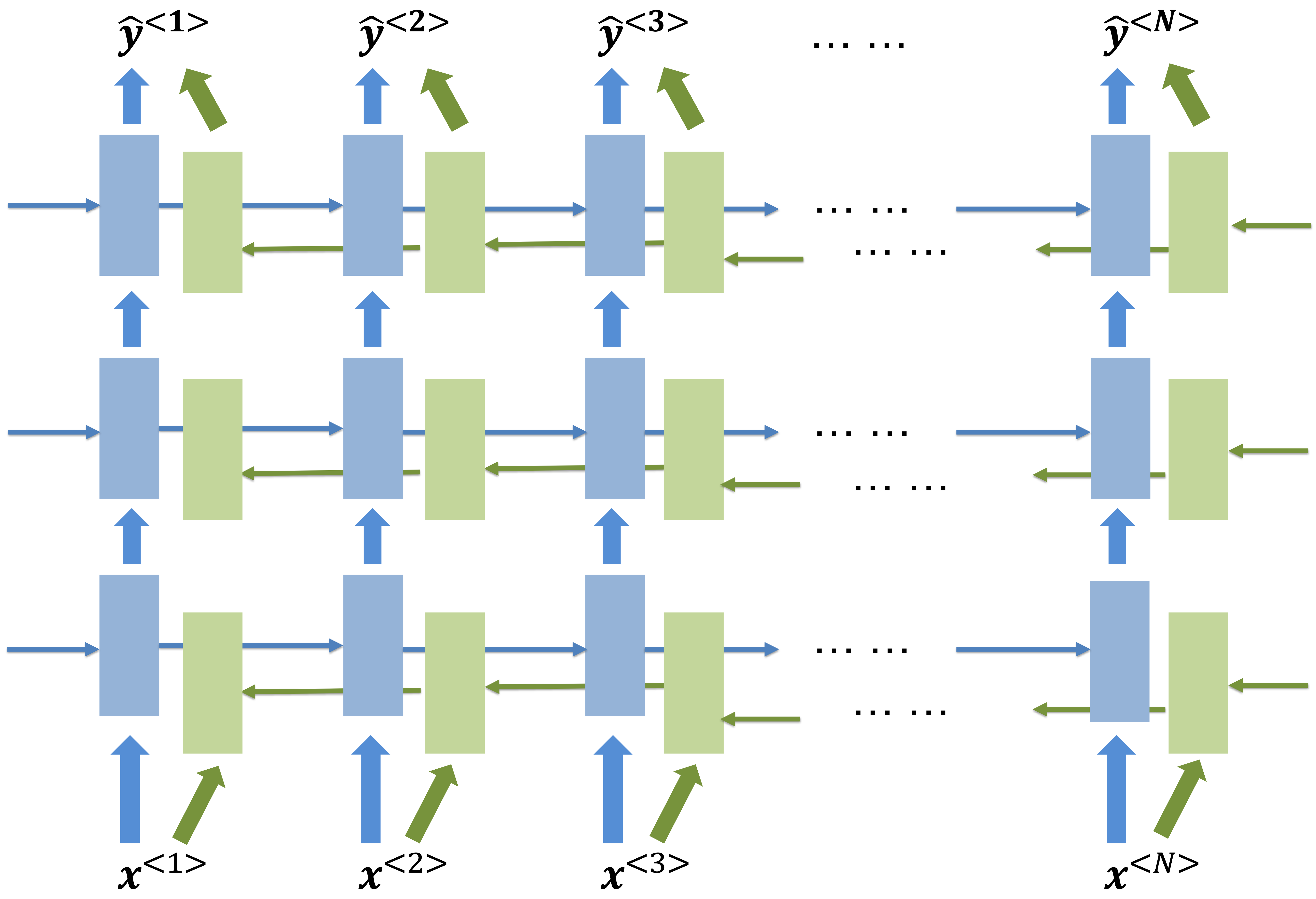
where h_t is the hidden state at time *t*, x_t is the input at time *t*, $h_{(t-1)}$ is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and r_t, z_t, n_t are the reset, update, and new gates, respectively. σ is the sigmoid function, and $*$ is the Hadamard product.

In a multilayer GRU, the input $x_t^{(l)}$ of the l -th layer ($l >= 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer multiplied by dropout $\delta_t^{(l-1)}$ where each $\delta_t^{(l-1)}$ is a Bernoulli random variable which is 0 with probability `dropout`.

Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two GRUs together to form a *stacked GRU*, with the second GRU taking in outputs of the first GRU and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (batch, seq, feature). Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each GRU layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional GRU. Default: `False`

Multi-layer RNN or deep RNN



CLASS `torch.nn.GRU(*args, **kwargs)` [SOURCE]

Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
r_t &= \sigma(W_{ir} x_t + b_{ir} + W_{hr} h_{(t-1)} + b_{hr}) \\
z_t &= \sigma(W_{iz} x_t + b_{iz} + W_{hz} h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in} x_t + b_{in} + r_t * (W_{hn} h_{(t-1)} + b_{hn})) \\
h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)}
\end{aligned}
$$

where h_t is the hidden state at time *t*, x_t is the input at time *t*, $h_{(t-1)}$ is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and r_t , z_t , n_t are the reset, update, and new gates, respectively. σ is the sigmoid function, and $*$ is the Hadamard product.

In a multilayer GRU, the input $x_t^{(l)}$ of the l -th layer ($l >= 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer multiplied by dropout $\delta_t^{(l-1)}$ where each $\delta_t^{(l-1)}$ is a Bernoulli random variable which is 0 with probability `dropout`.

Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two GRUs together to form a *stacked GRU*, with the second GRU taking in outputs of the first GRU and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (batch, seq, feature). Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each GRU layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional GRU. Default: `False`

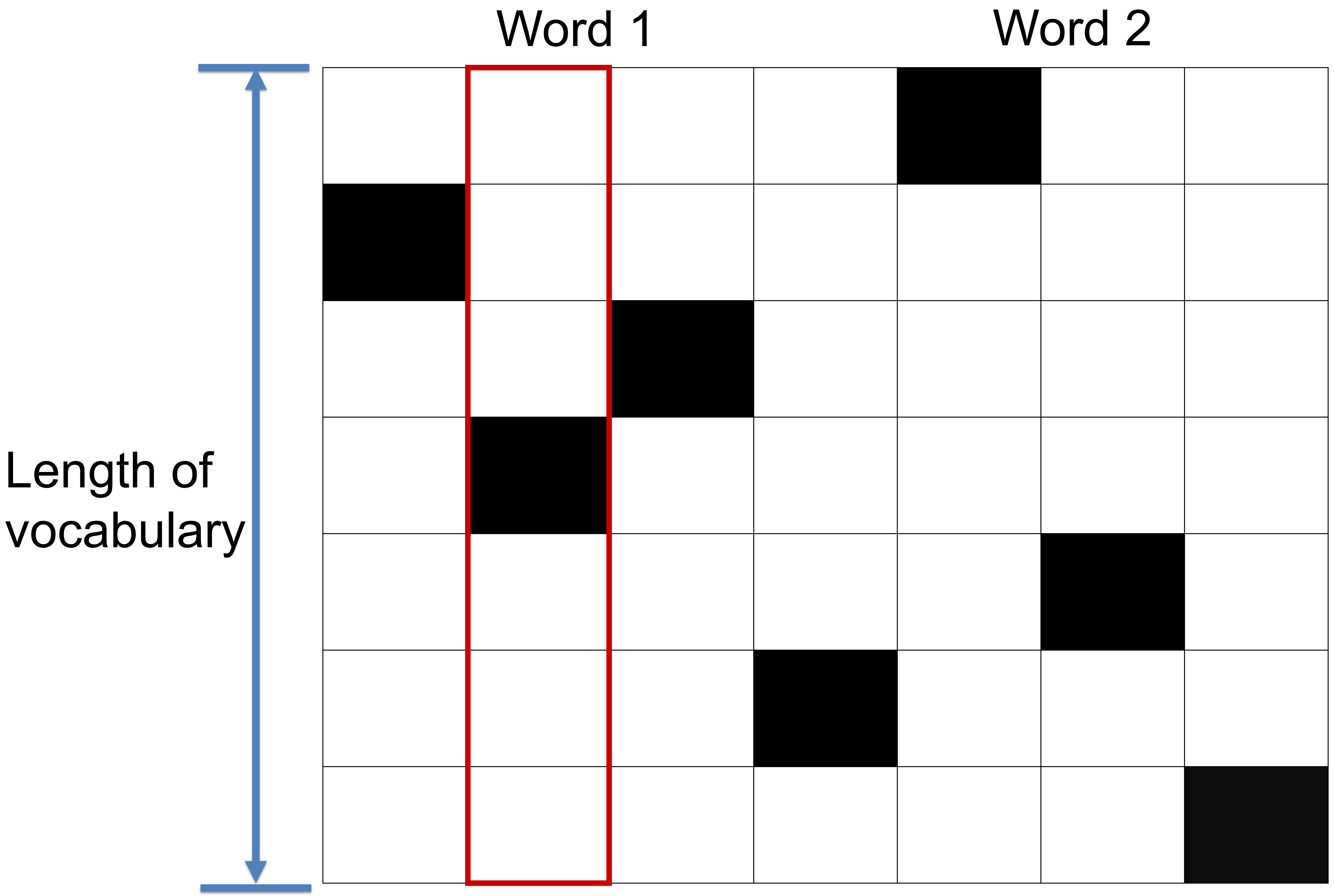
Sequence pre-processing

This is a picture showing kids are playing.

Word level vocabulary:

[
a 0
aa 1
…
air
…
is 2876
…
grass
…
this
…
playing 27890
…
zinc
]

Use one-hot encoding for words



- 10K-30K entries in English vocabulary
- Assume all words are orthogonal to each other
- Does not utilize any “pre-training”

Sequence pre-processing

Character level vocabulary:

ASCII printable characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Use one-hot encoding for characters

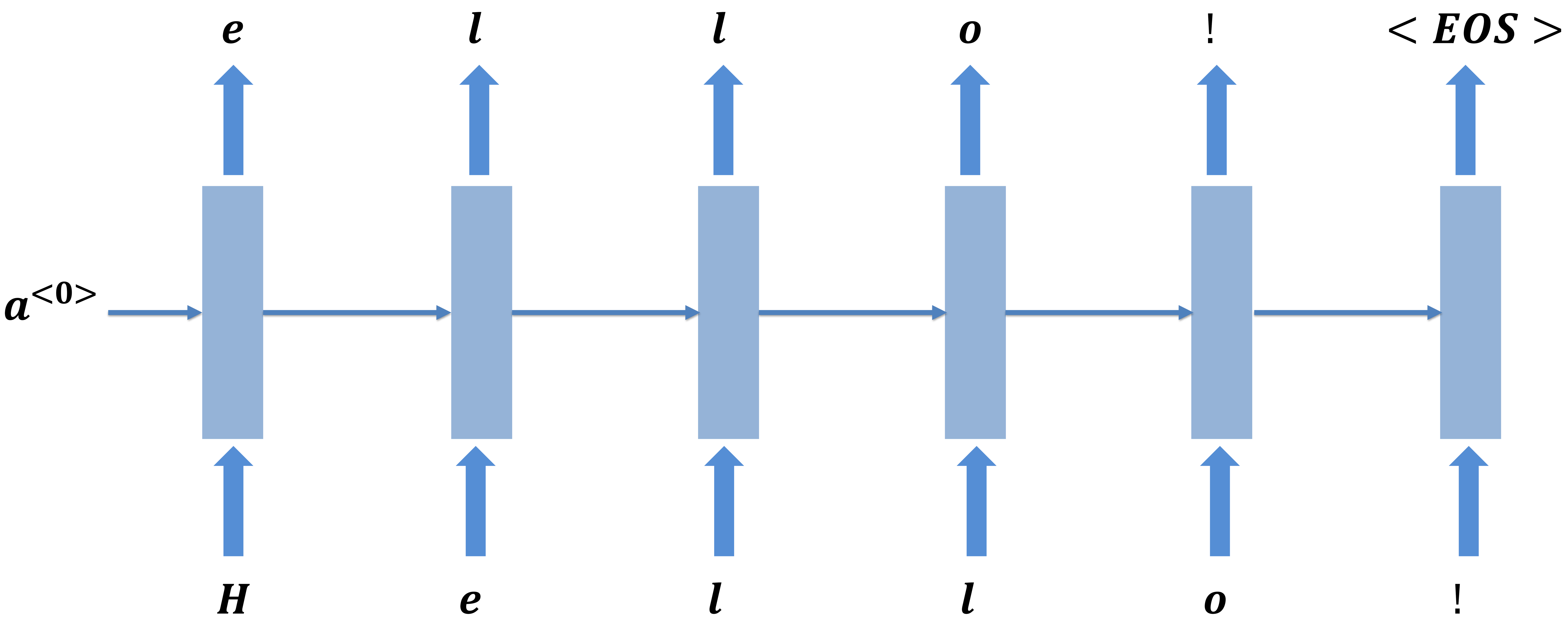
Often use compact representation:

This is a picture showing kids are playing.

T: 27	h:12	i:13	s:23	… …

Use character vocabulary

Hello!

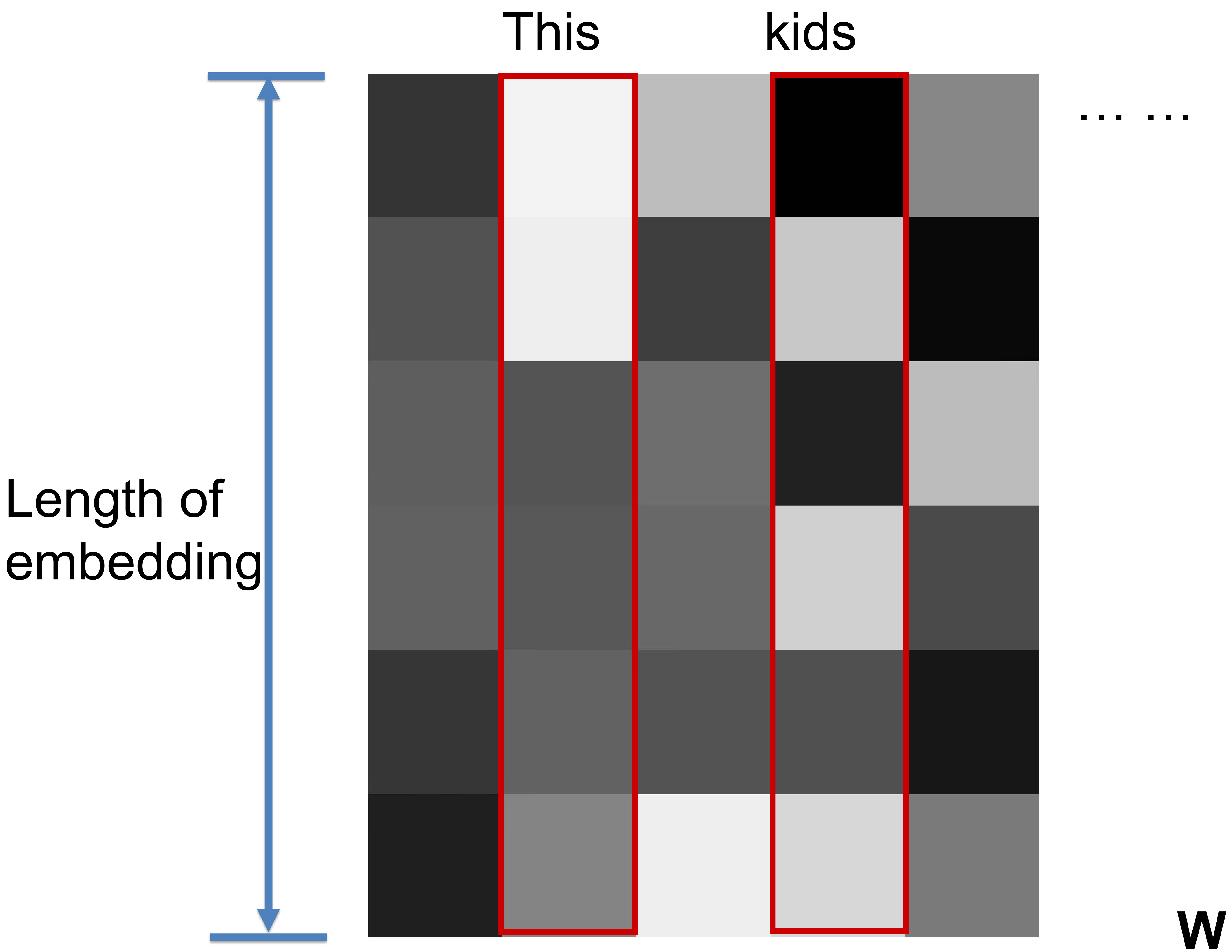


- Casual training
- Finish with <EOS> token

Word embedding

This is a picture showing kids are playing.

Use a real-value vector to represent words

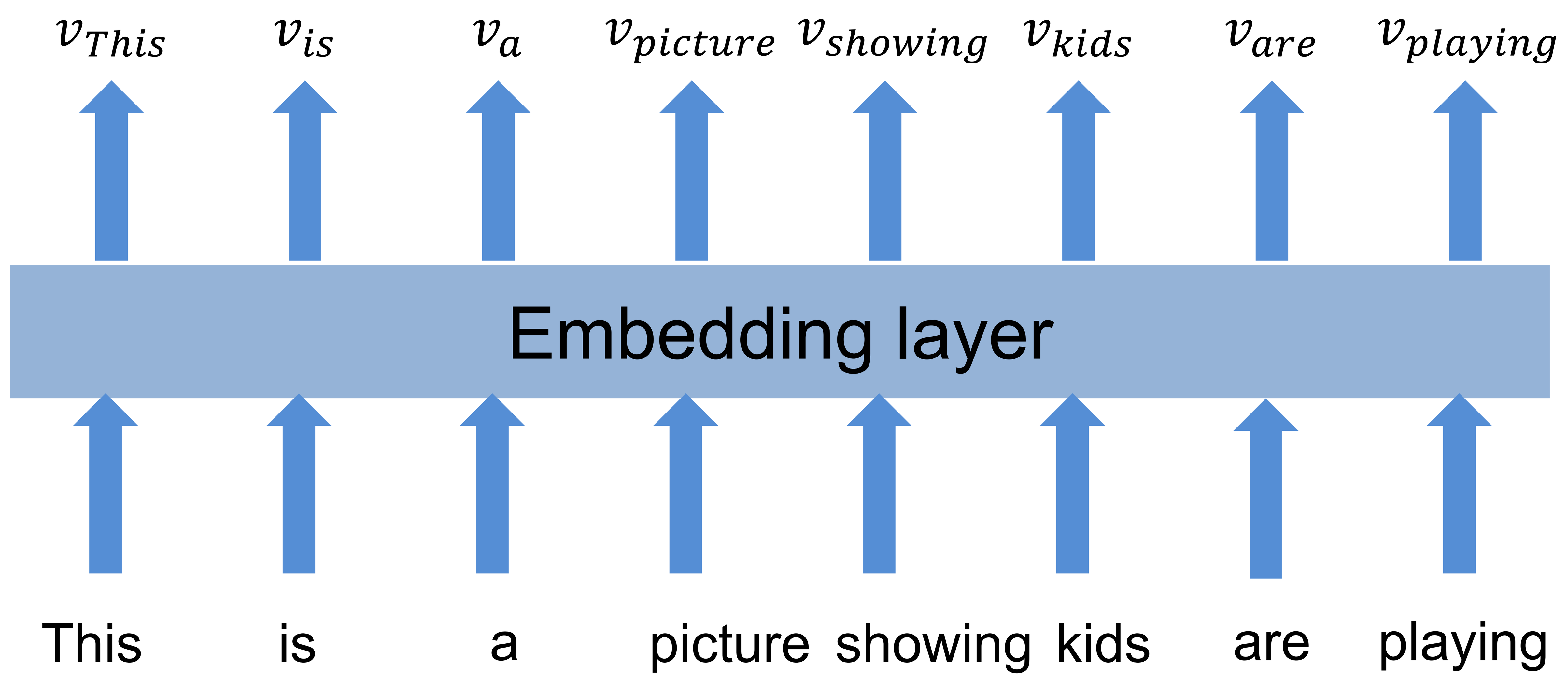


EMBEDDING

```
CLASS torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None,
      max_norm=None, norm_type=2.0, scale_grad_by_freq=False,
      sparse=False, _weight=None)
```
[SOURCE]

A simple lookup table that stores embeddings of a fixed dictionary and size.

$v = W \times one\_hot\_vector$ W: embedding_dim x num_embedding



W can be learned or pre-trained.

Word2Vec

This is a <u>picture</u> showing kids are playing.

$\underbrace{\qquad\qquad\qquad\qquad}_{x}$

Skip-grams

Self-supervised learning

X : center word

Y: neighboring word

Center	Context
<u>picture</u>	showing
<u>picture</u>	a
<u>picture</u>	is
<u>picture</u>	This

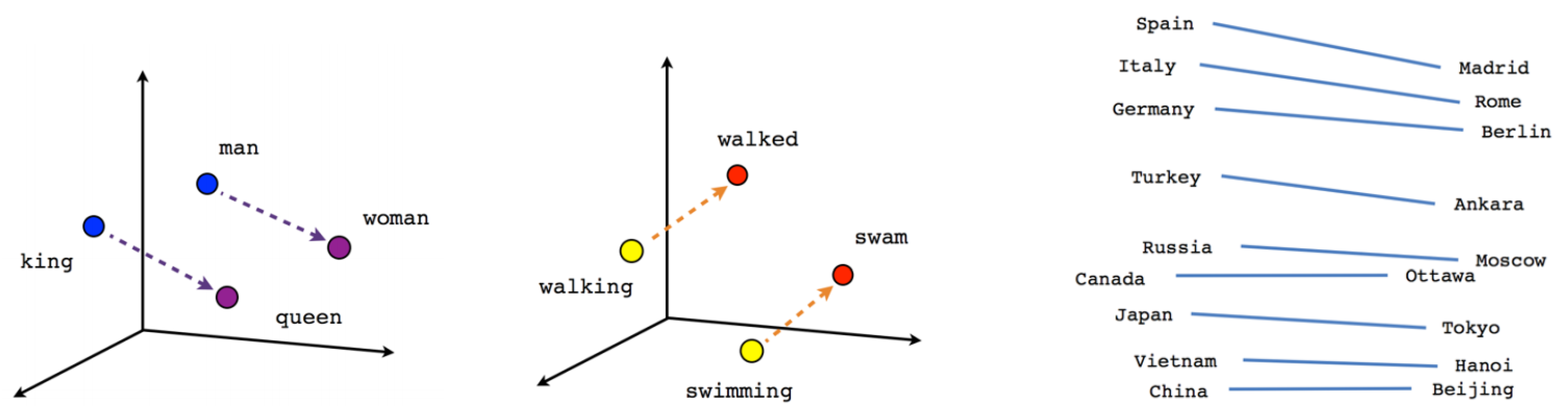
showing

$SoftMax$

$W \times (\cdot) + b$

Embedding layer

<u>picture</u>



Male-Female   Verb tense   Country-Capital

- Train on a large corpse of texts
- Map words to N-dimensional space
- Learned semantic relationship between words

$$e_{king} + (e_{woman} - e_{man}) \approx e_{queen}$$

https://www.tensorflow.org/tutorials/representation/word2vec

Efficient Estimation of Word Representations in Vector Space.2013. https://arxiv.org/abs/1301.3781

Word2Vec : carry the accumulated “bias” of the world

he (128)

she (72)

http://wordbias.umiacs.umd.edu/

Sequence pre-processing

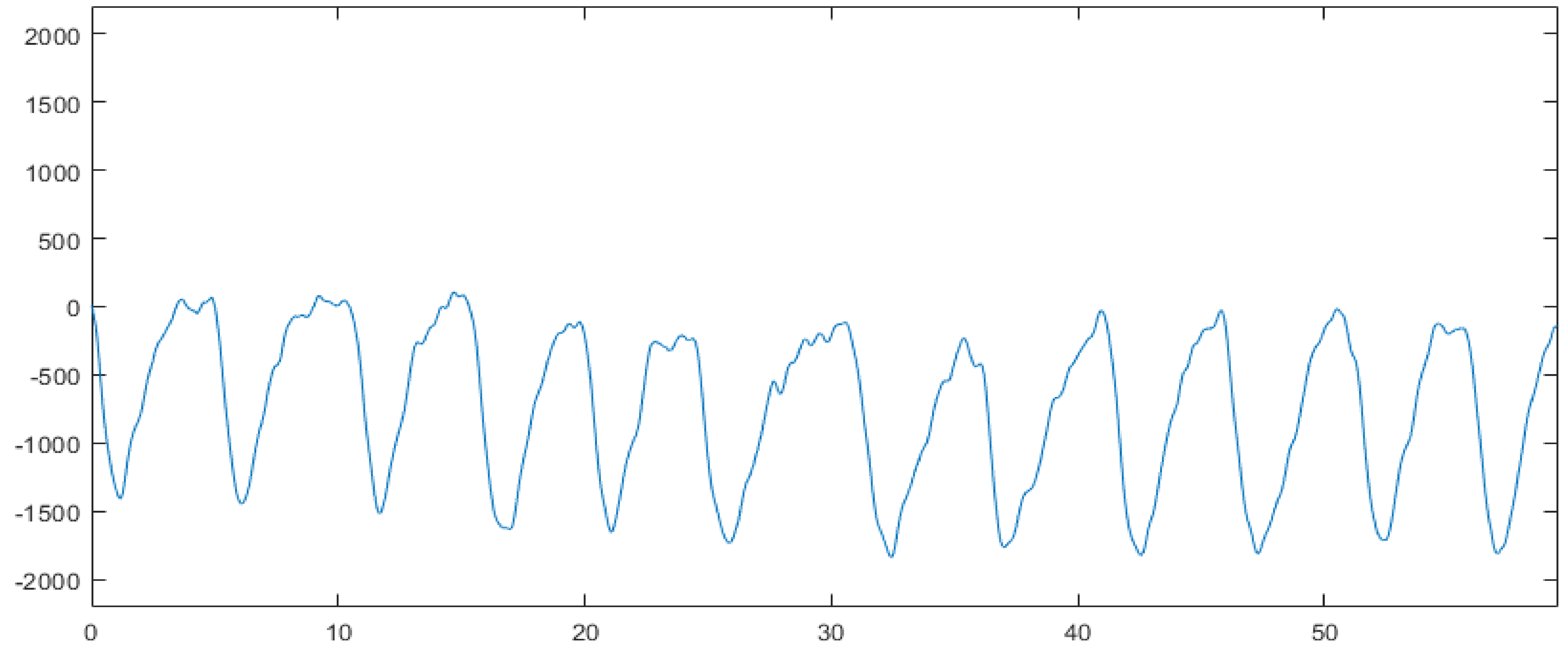
Multiple instances of sequences:

This is a picture showing kids are playing → 8x1024 matrix

Today is a good day → 5x1024 matrix

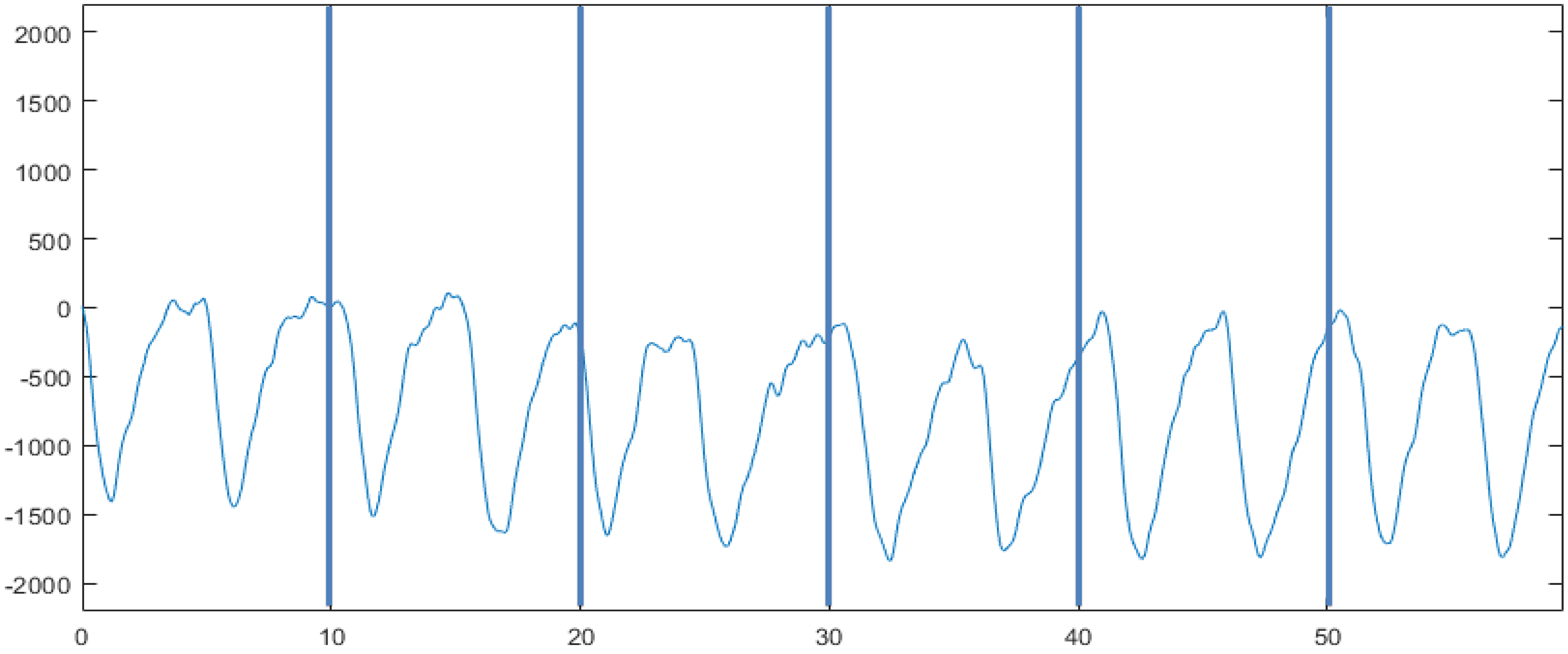
The boy is reading a book under the tree → 9x1024 matrix

One instance of long sequence:



Sequence pre-processing

One instance of long sequence:



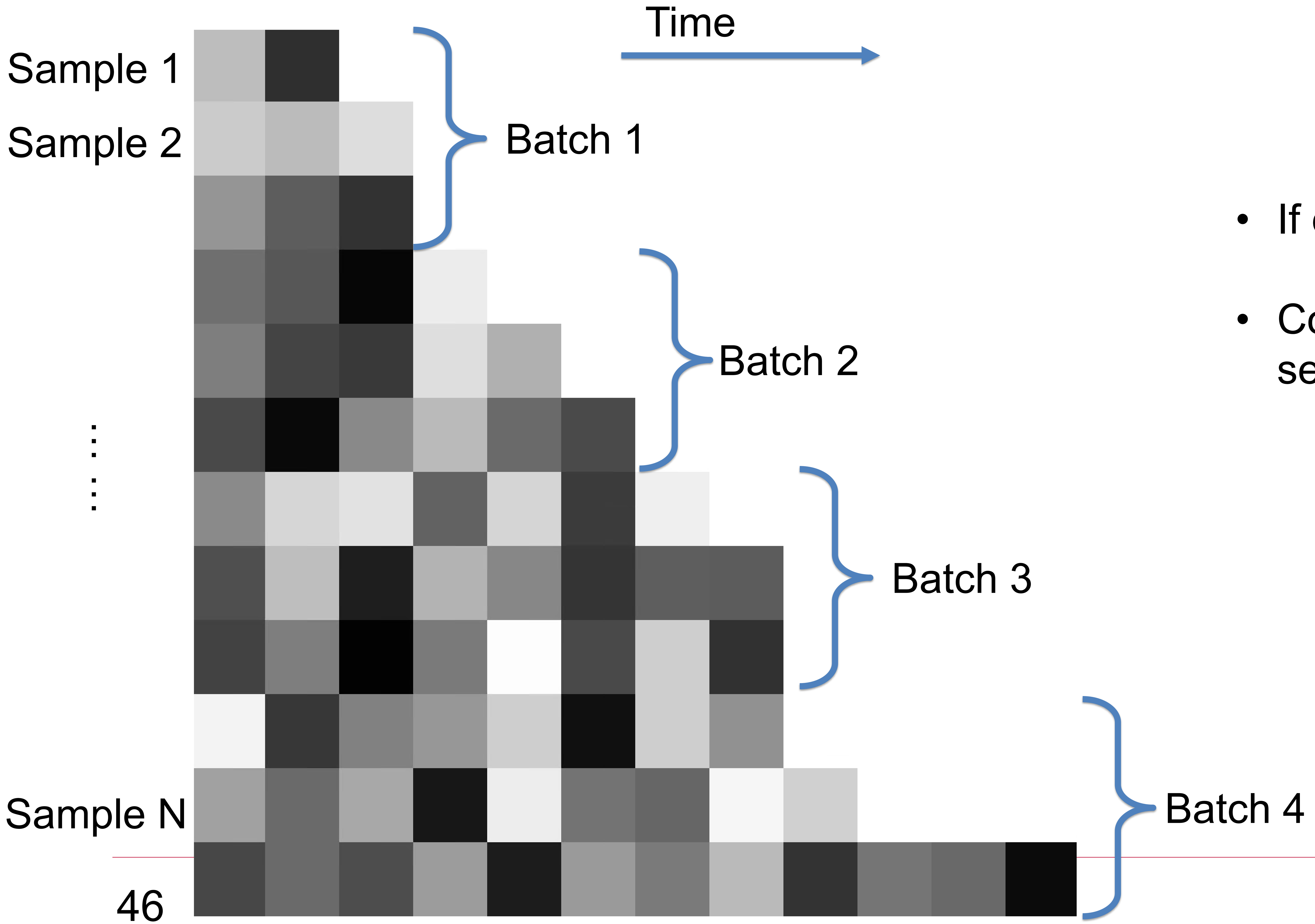
Split to equal length chunk

Sequence pre-processing

Multiple instances of sequences : padding to equal length

Sequence pre-processing

Multiple instances of sequences : padding to equal length, with sorting

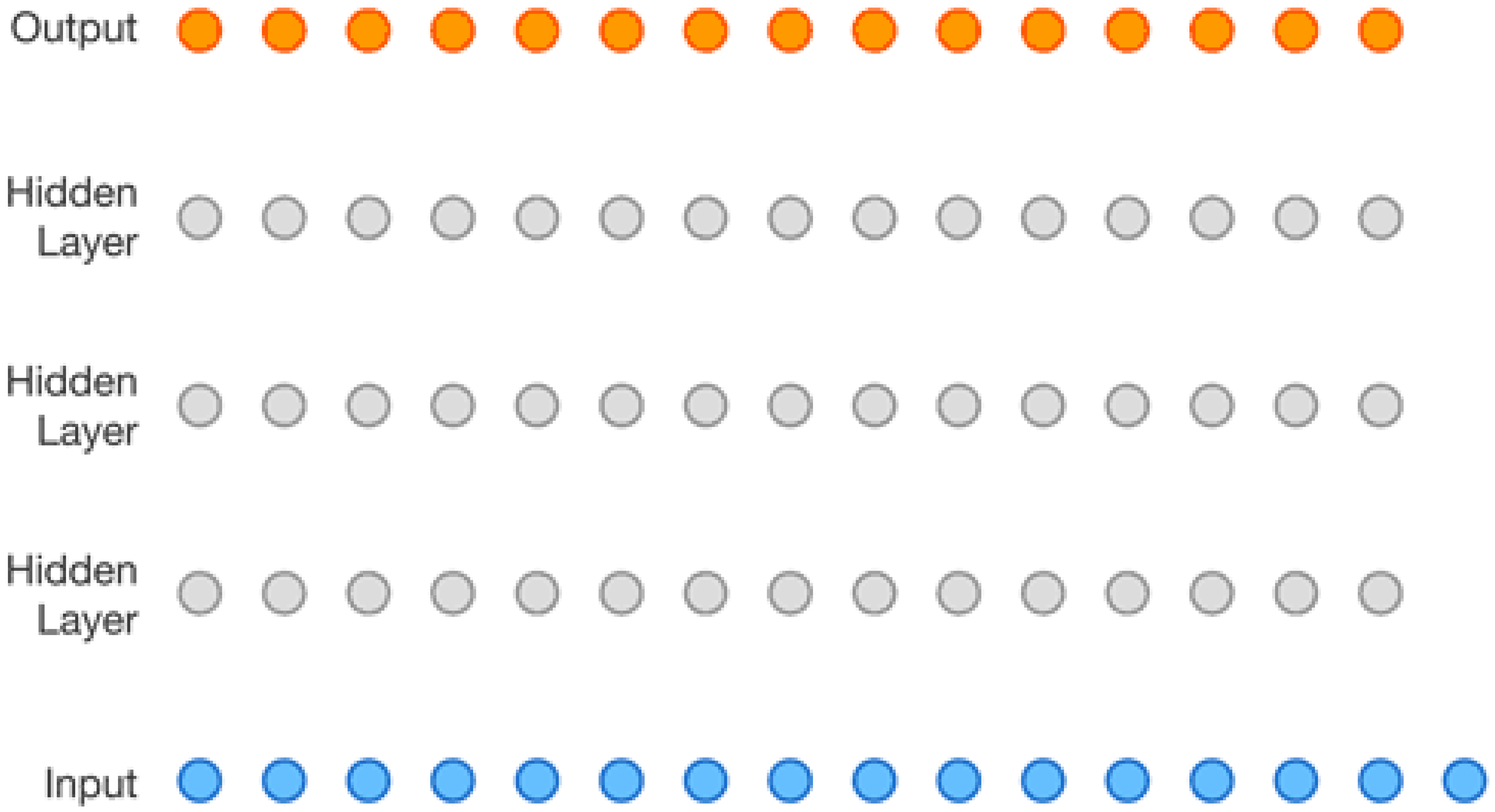


- If data distribution is not biased by its length
- Consider to reduce batch size for longer sequences

Temporal Convolution Network

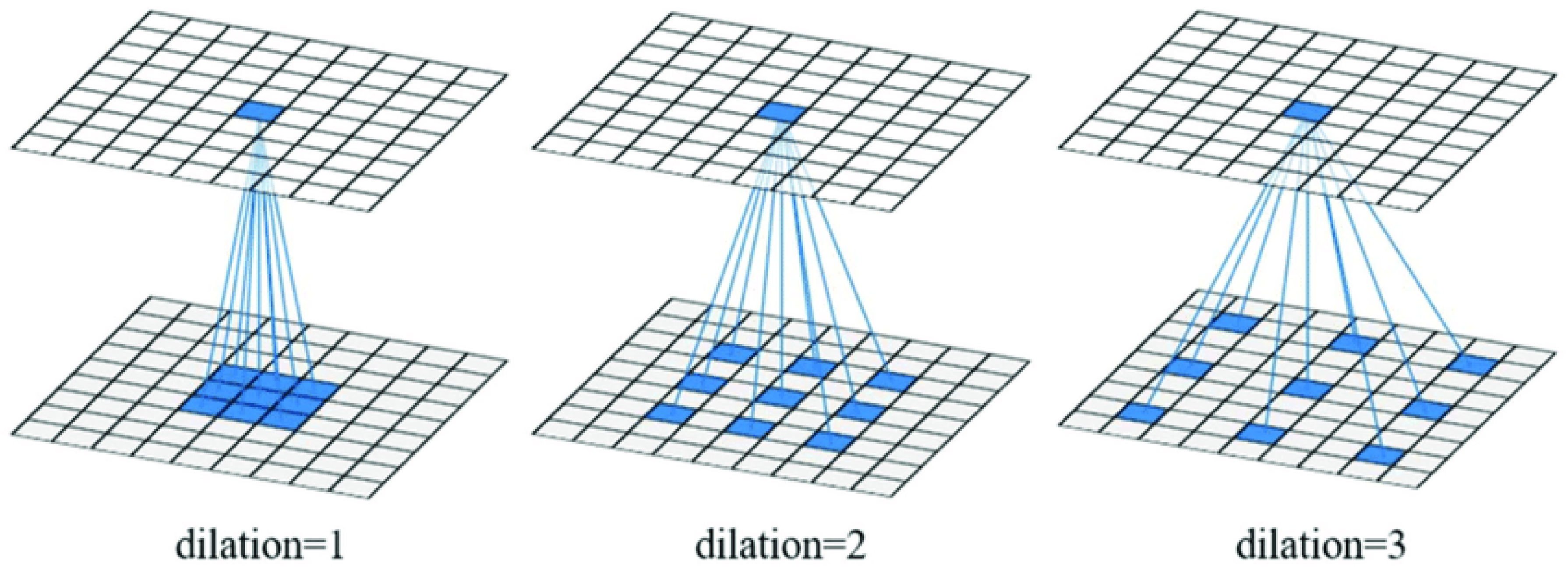
- Use convolution to process sequence data
- Dilated + Casual convolution

Receptive_length_in_time $= 2^L(kernel_{size} - 1)$

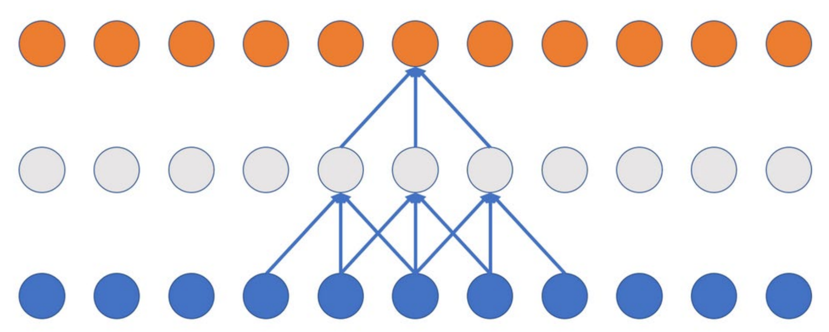


https://deepmind.com/blog/article/wavenet-generative-model-raw-audio

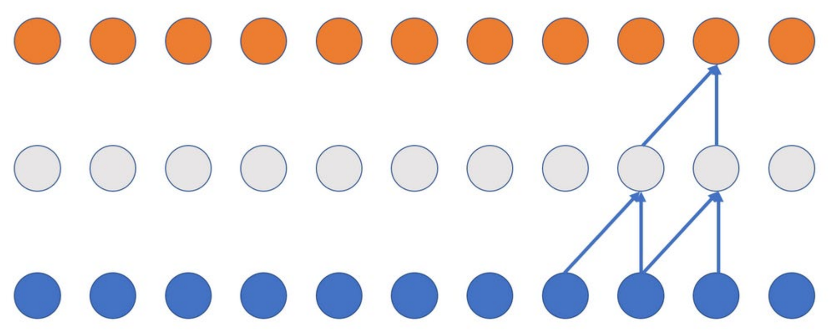
Dilated convolution



Normal convolution



Casual convolution



- Fast training, fast inference
- Google Wavenet – text to audio generation, used in e.g. Amazon Alexa