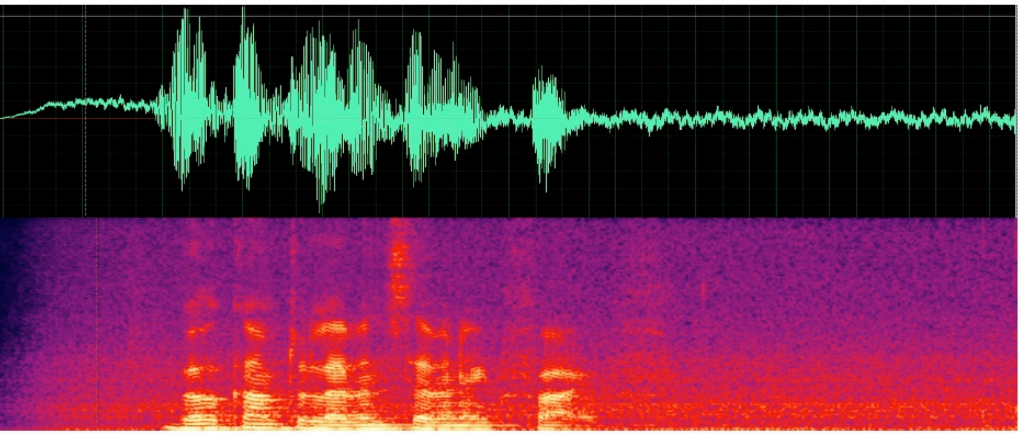
Recurrent Neural Networks

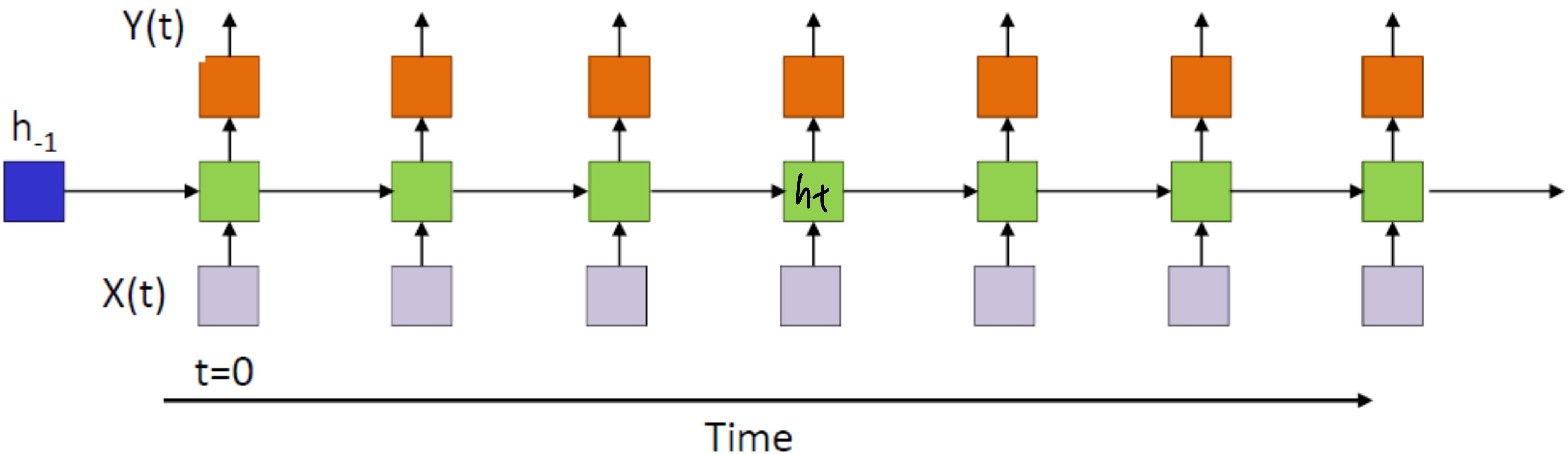
Sequence Data



State-Space Model

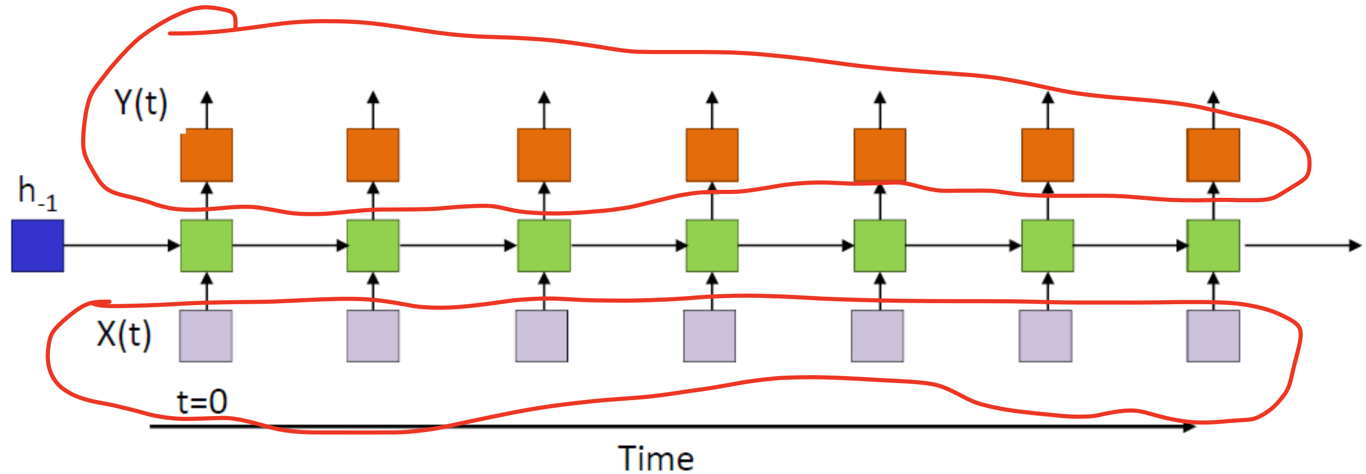
compress information before time t

- h_t: hidden state
- X_t: input
- Y_t: output
- $Y_t, h_t = f(h_{t-1}, X_t; \theta)$
- h_{-1}: initial state



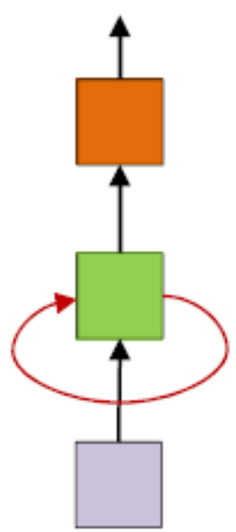
Recurrent Neural Network

- h_t: hidden state
- X_t: input
- Y_t: output
- $Y_t, h_t = f(h_{t-1}, X_t; \theta)$
- h_{-1}: initial state



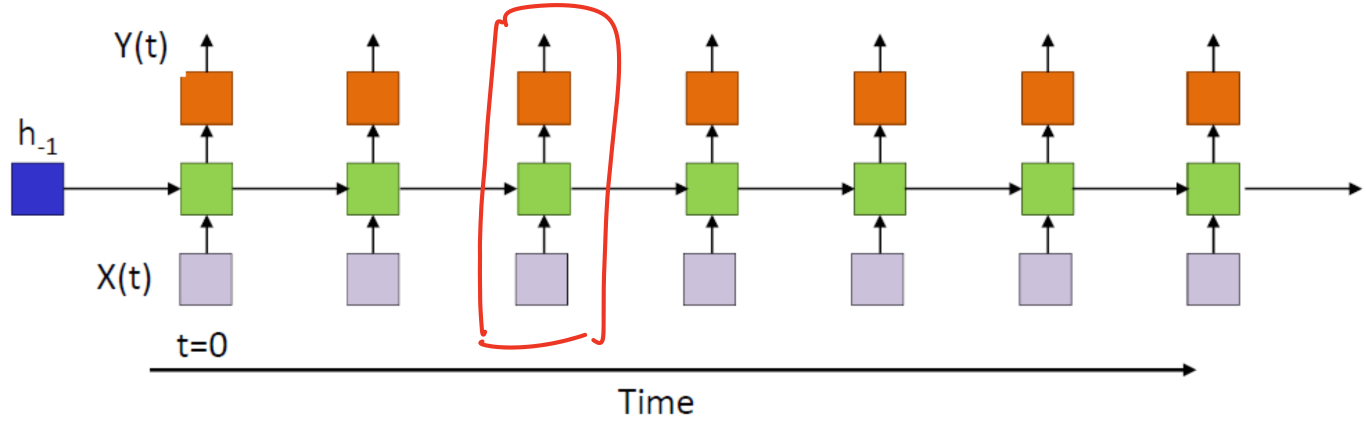
Fully-connect NN vs. RNN

- h_t: a vector summarizes all past inputs (a.k.a. "memory")
- h_{-1} affects the entire dynamics (typically set to zero)
- X_t affects all the outputs and states after t
- Y_t depends on $X_0, \ldots, X_t$



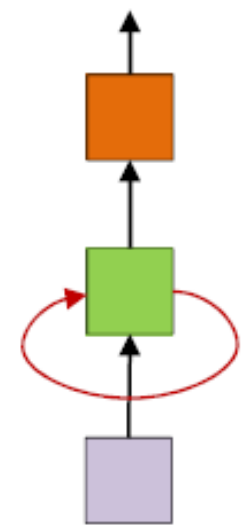
Recurrent Neural Network

- h_t: hidden state
- X_t: input
- Y_t: output
- $Y_t, h_t = f(h_{t-1}, X_t; \theta)$
- h_{-1}: initial state

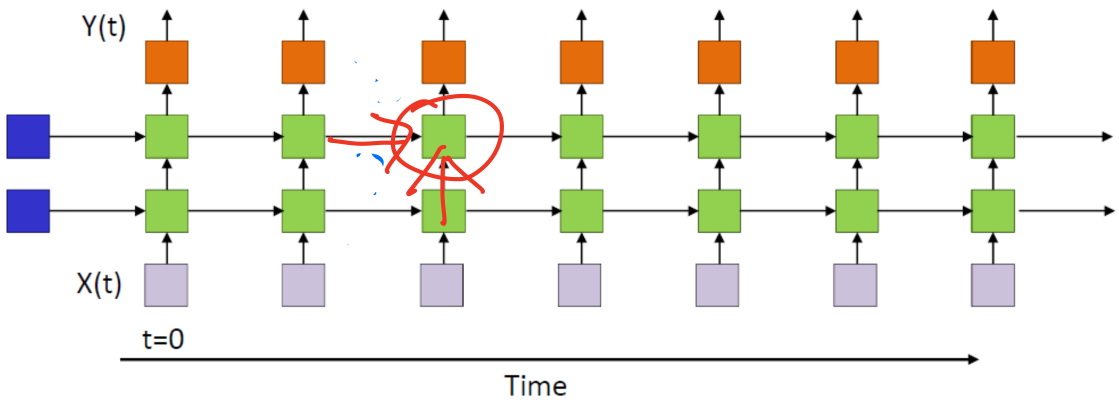


Fully-connect NN vs. RNN

- RNN can be viewed as repeated applying fully-connected NNs
- $h_t = \sigma_1(W^{(1)}X_t + W^{(11)}h_{t-1} + b^{(1)})$
- $Y_t = \sigma_2(W^{(2)}h_t + b^{(2)})$
- σ_1, σ_2 are activation functions (sigmoid, ReLU, tanh, etc)



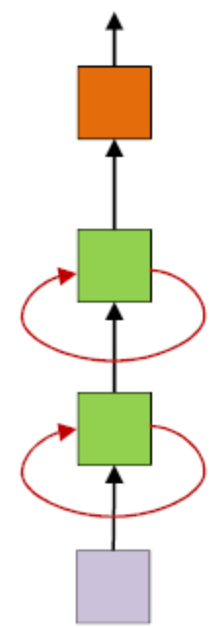
Recurrent Neural Network



Stack K layers of fully-connected NN

- $h_t^{(k)}$: hidden state
- X_t: input
- Y_t: output
- $h_t^{(1)} = f_1^{(1)}(h_{t-1}^{(1)}, X_t; \theta)$
- $h_t^{(k)} = f_1^{(k)}(h_{t-1}^{(k)}, h_t^{(k-1)}; \theta)$
- $Y_t = f_2(h_t^{(K)}; \theta)$
- $h_{-1}^{(k)}$: initial states

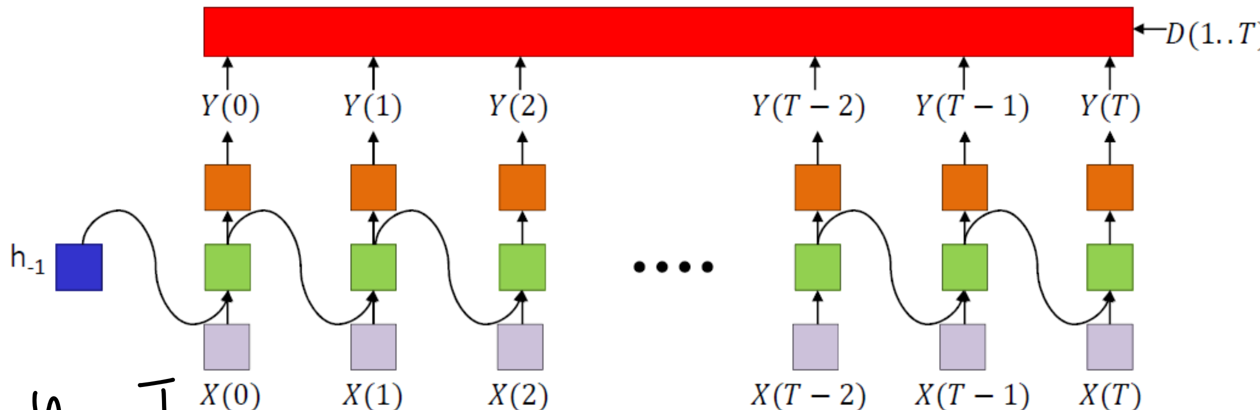
one layer FC NN



Training Recurrent Neural Network

$y(t)$: true label

- h_t: hidden state
- X_t: input
- Y_t: output
- $Y_t, h_t = f(h_{t-1}, X_t; \theta)$
- h_{-1}: initial state

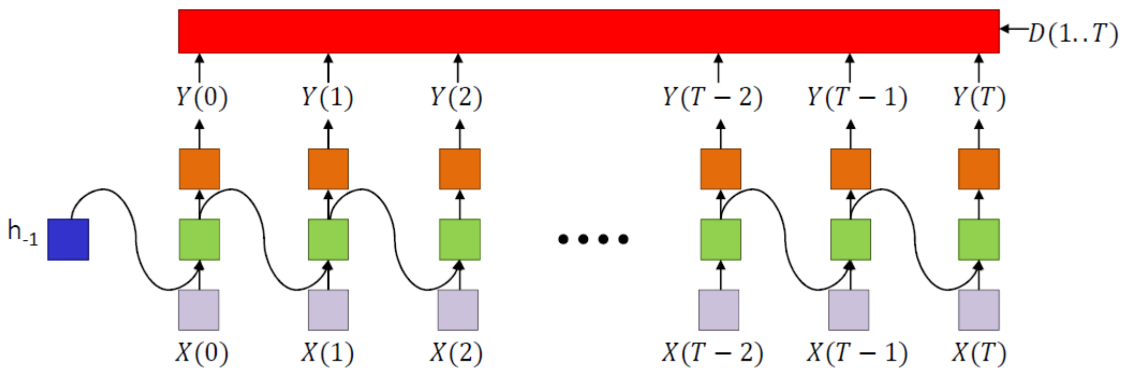


$$\sum_{i=1}^{n} \sum_{t=1}^{T} (Y(t) - y(t))^2$$

- Data: $\{(X_t, D_t)\}_{t=1}^{T}$ (RNN can handle more general data format)
- Loss $L(\theta) = \sum_{t=1}^{T} \ell(Y_t, D_t)$
- Goal: learn θ by gradient-based method
  - Back propagation

Back Propagation Through Time

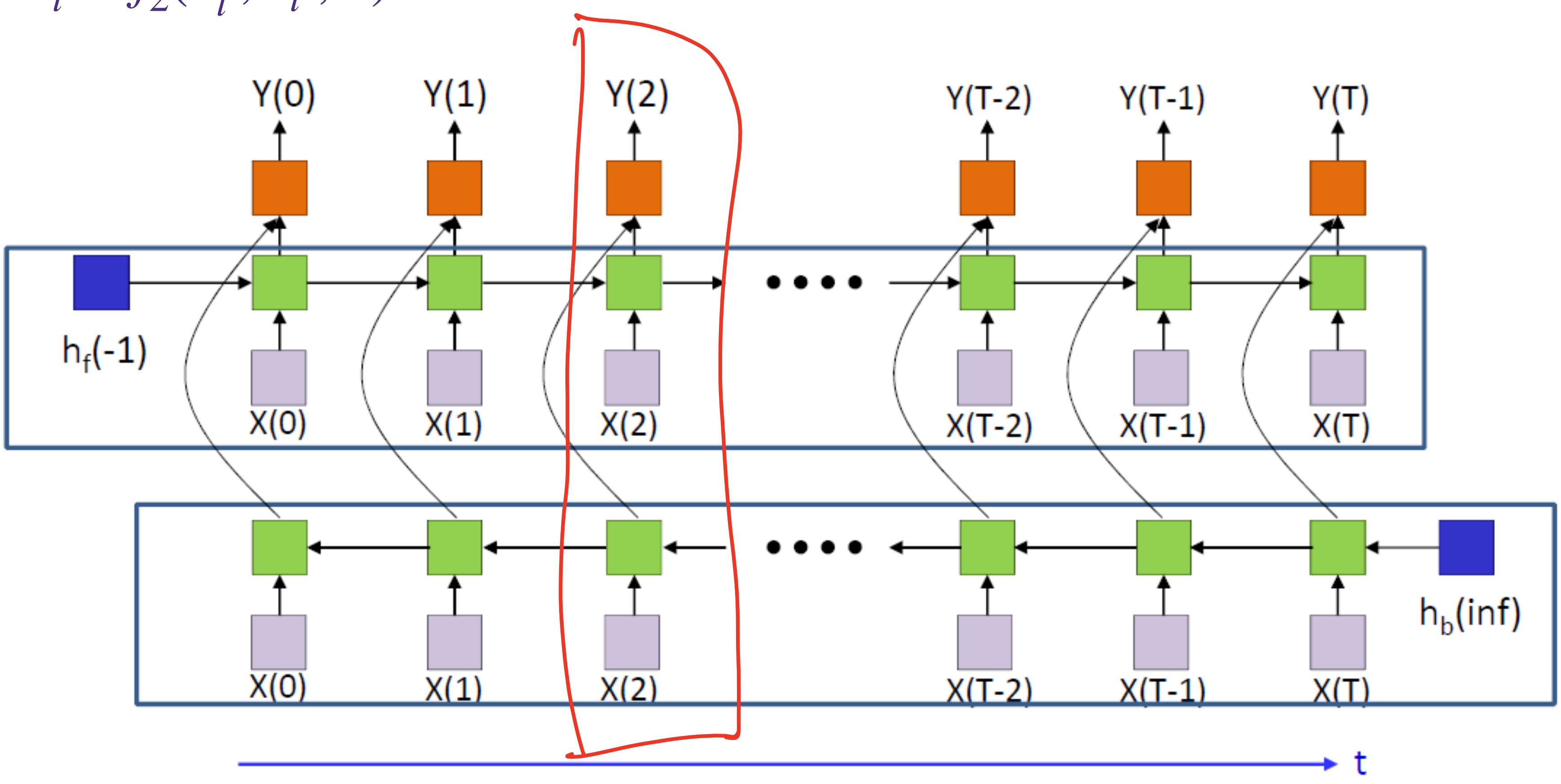
- $h_t = \sigma_1(W^{(1)}X_t + W^{(11)}h_{t-1} + b^{(1)})$
- $Y_t = \sigma_2(W^{(2)}h_t + b^{(2)})$
- $Z_t^{(1)}$: pre-activation of hidden state
  $(h_t = \sigma_1(Z_t^{(1)}))$
- $Z_t^{(2)}$: pre-activation of output
  $(Y_t = \sigma_2(Z_t^{(2)}))$



Extensions

What if Y_t depends on the entire inputs?

- Biredictional RNN:
  - AN RNN for forward dependencies: t= 0,…,T
  - An RNN for backward dependencies: t= T,…0
  - $Y_t = f_2(h_t^f, h_t^b; \theta)$



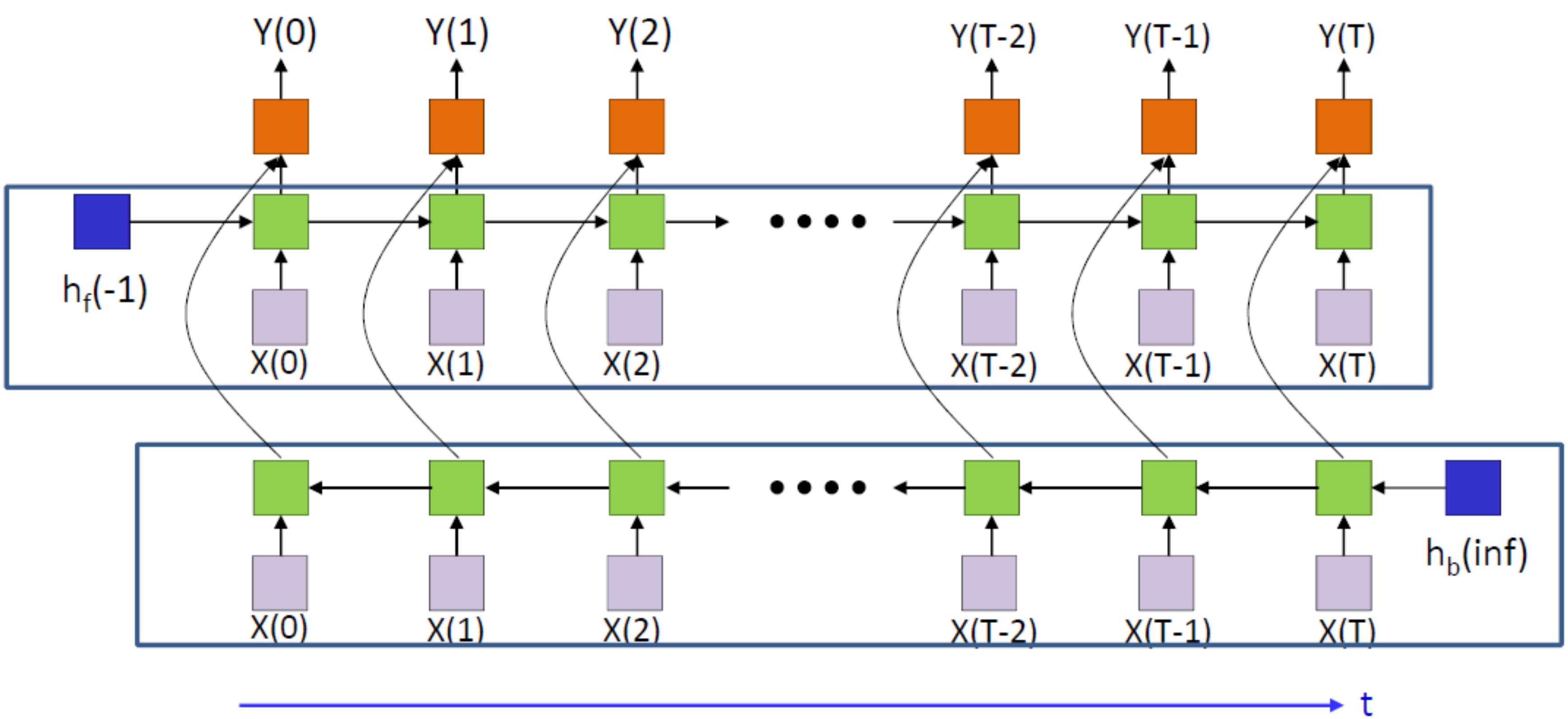
Extensions

RNN for sequence classification (sentiment analysis)

- $\boxed{Y = \max_t Y_t}$
- Cross-entropy loss

$\{ \{X_t\}_{t=1}^{T}, y \}$

$\ell(Y, y)$



Practical issues of RNN

$$\boxed{\sigma(z) = z}$$

Linear RNN derivation

- $h_t = W^{(11)} h_{t-1} + W^{(1)} x_t$
- $h_k = W^{(1)} x_k + W^{(11)} h_{k-1}$

$$= W^{(1)} x_k + W^{(11)} \left(W^{(1)} x_{k-1} + W^{(11)} h_{k-2} \right)$$

$$\cdots$$

$$= \boxed{\left(W^{(11)}\right)^{k+1} h_{-1} + \sum_{i=0}^{k} \left(W^{(11)}\right)^{k-i} W^{(1)} x_i}$$

$$\exp()$$

If $\lambda_{max}(W^{(11)}) > 1 \rightarrow$ exp large

$< 1 \rightarrow$ exp small

Practical issues of RNN: training

Gradient explosion and gradient vanishing

gradient $\propto \left(W^{(11)}\right)^k$ xxx

Techniques for avoiding gradient explosion

- Gradient clipping

threshold if $\|g\| > \text{thres}$

$$g \leftarrow \frac{\text{thres}}{\|g\|} \cdot g$$

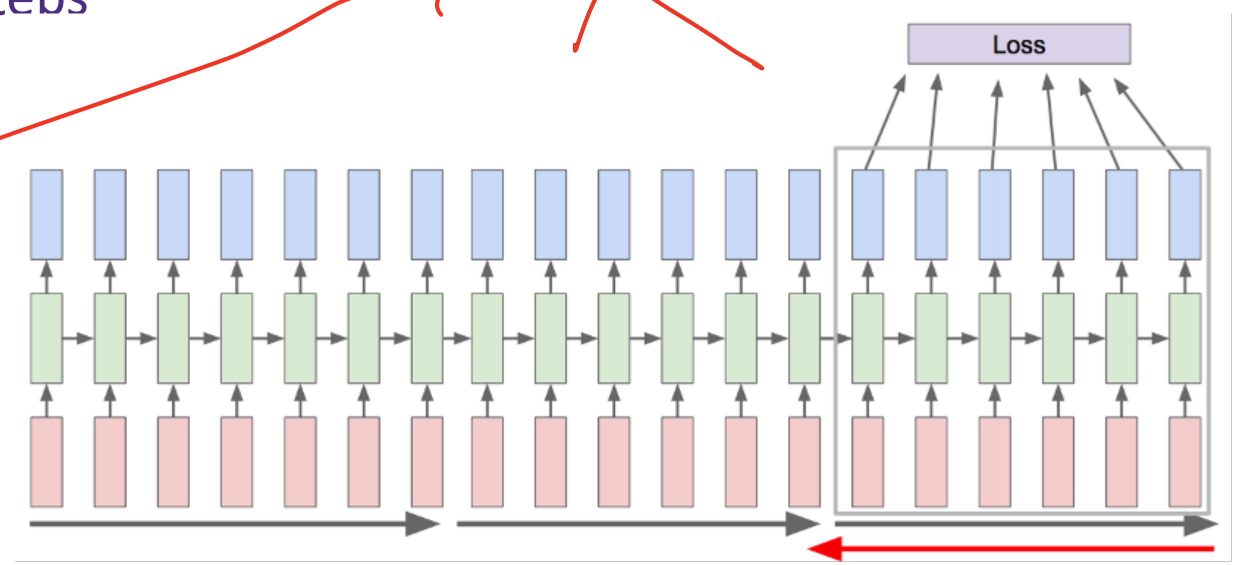
- Identity initialization

$$W^{(hh)} = I$$

- Truncated backprop through time
  - Only backprop for a few steps

y

CNN

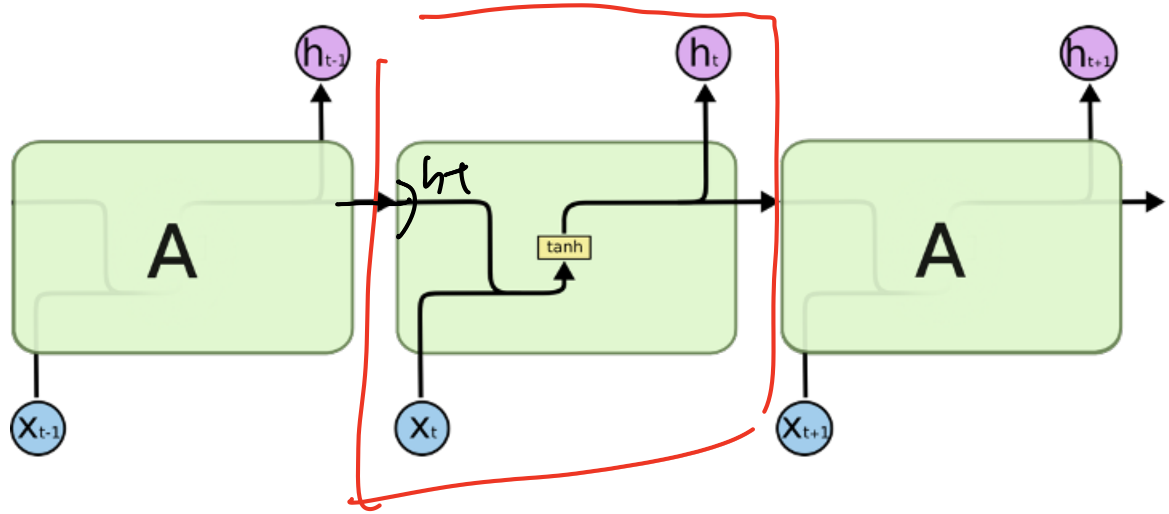


(image, sentence) $\rightarrow$ y

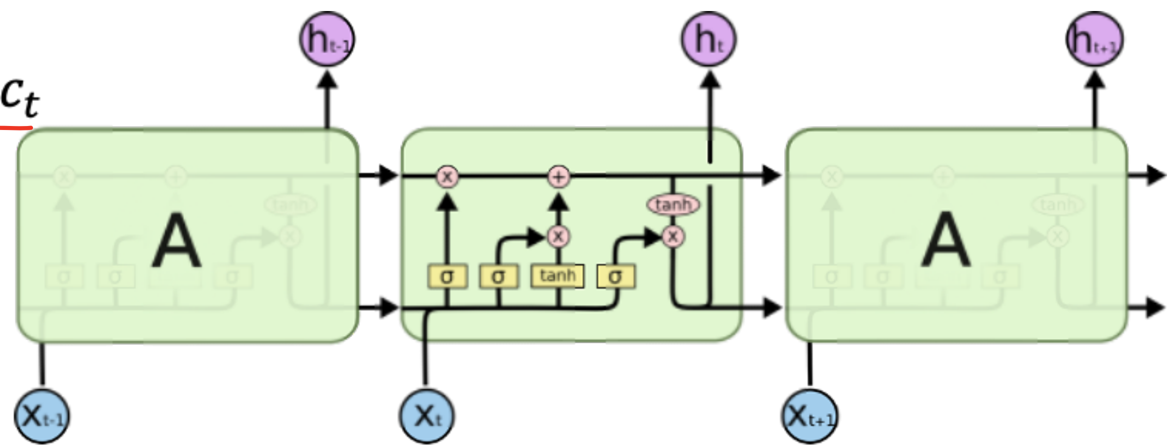
Preserve Long-Term Memory

Y_t depends on $x_0, \dots, x_t$

- Difficult for RNN to preserve long-term memory
  - The hidden state h_t is constantly being written (short-term memory)
  - Use a separate cell to maintain long-term memory



Memory c_t

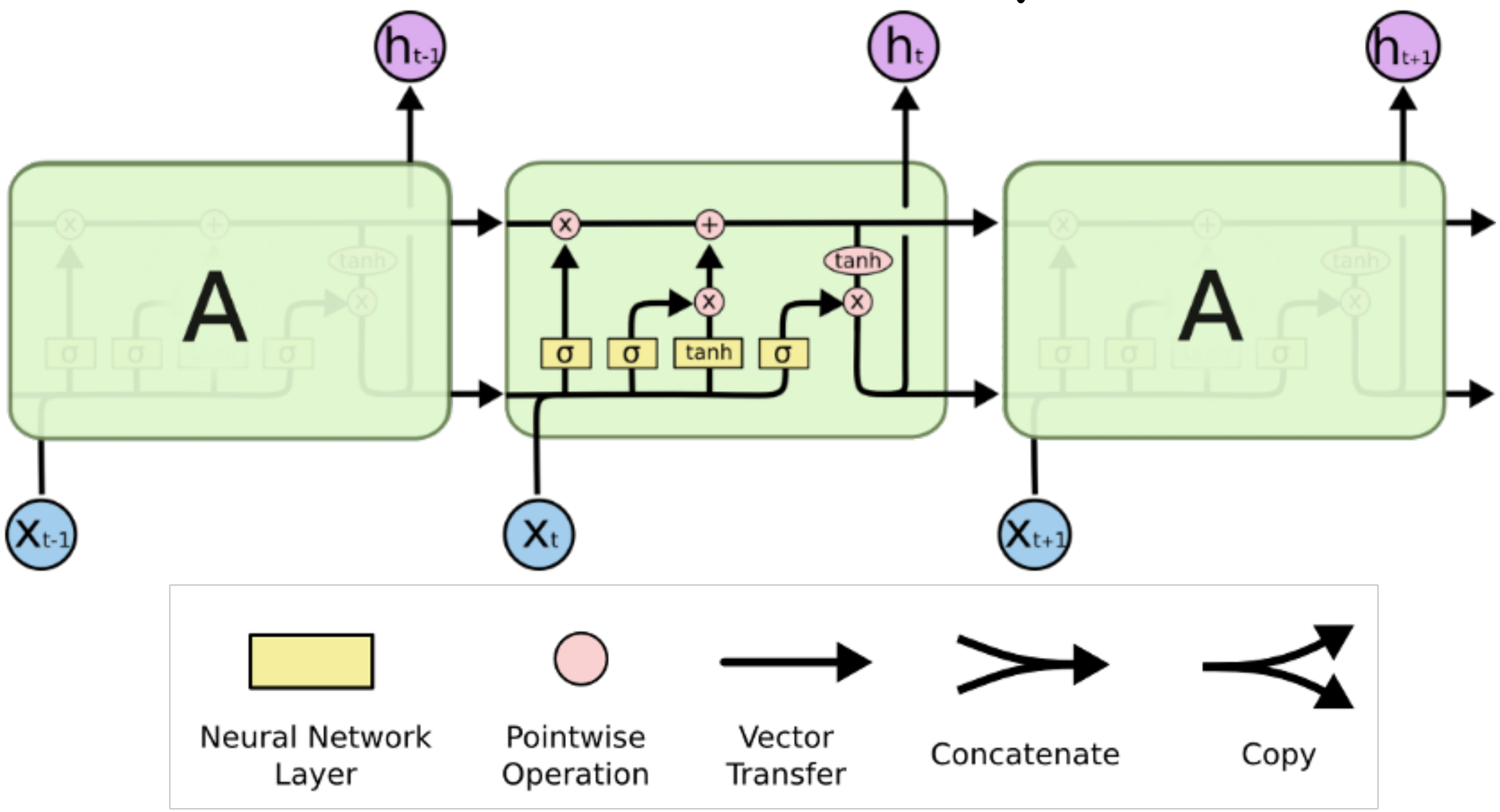


Long Short-Term Memory Network

LSTM (Hochreitcher & Schmidhuber, '97)

- RNN architecture for learning long-term dependencies
- σ: layer with sigmoid activation

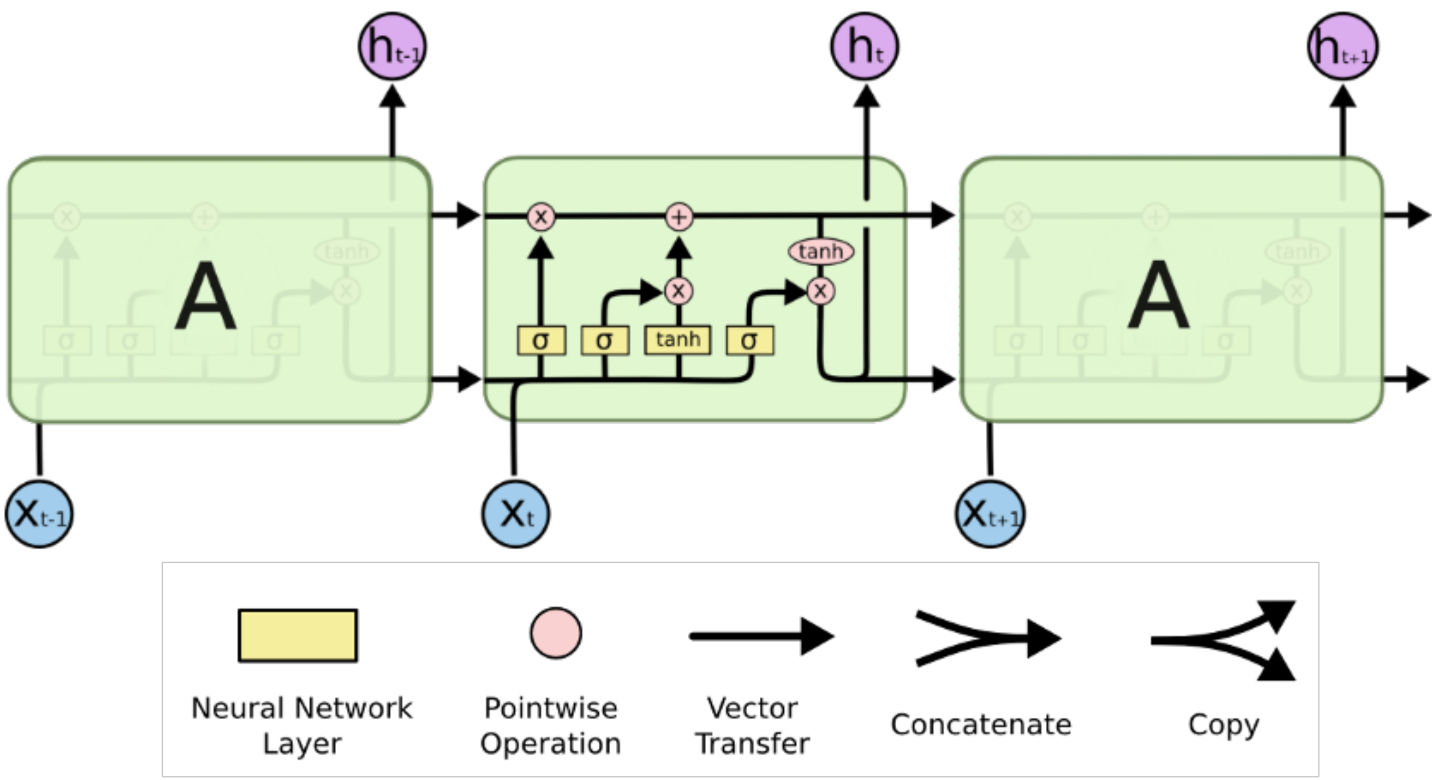
$$a \in \mathbb{R}^d, b \in \mathbb{R}^d \quad a \otimes b = \begin{pmatrix} a_1 b_1 \\ \vdots \\ a_d b_d \end{pmatrix}$$



Long Short-Term Memory Network

LSTM (Hochreitcher & Schmidhuber, '97)

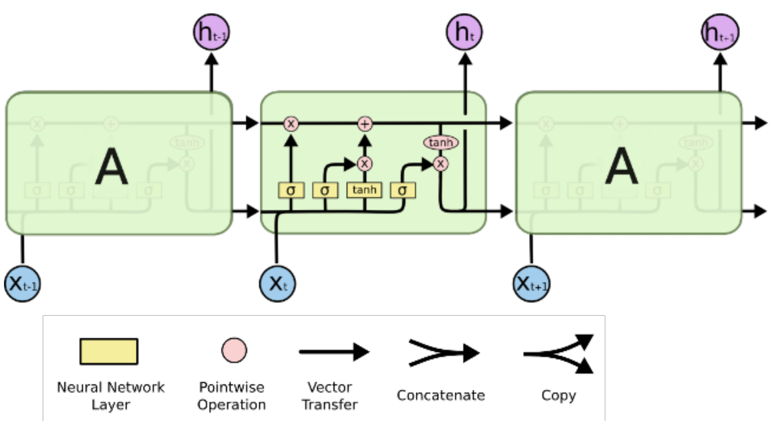
- Core idea: maintain separate state h_t and cell c_t (memory)
- h_t: full update every step
- c_t: only *partially* update through gates
  - σ layer outputs importance ($[0,1]$) for each entry and only modify those entries of c_t

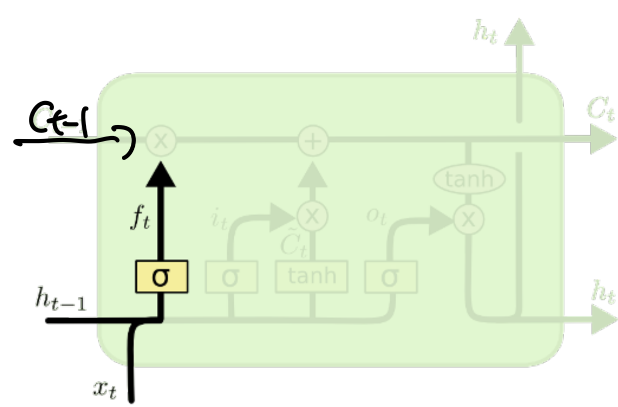


Long Short-Term Memory Network

Forget gate f_t

- f_t outputs whether we want to "forget" things in c_t
  - Compute $c_{t-1} \otimes f_t$ (element-wise)
  - $f_t(i) \rightarrow 0$: want to forget $c_t(i)$
  - $f_t(i) \rightarrow 1$: we want to keep the information in $c_t(i)$





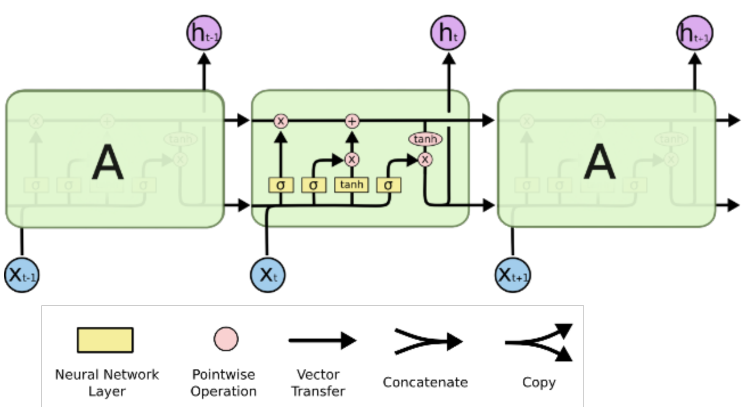
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

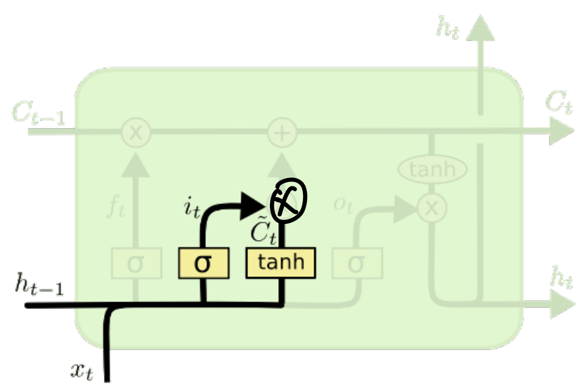
Sigmoid

Long Short-Term Memory Network

Input gate i_t

- i_t extracts useful information from X_t to update memory
  - $\tilde{c}_t$: information from X_t h_{t-1} to update memory
  - i_t: which dimension in the memory should be updated by X_t
    - $i_t(j) \rightarrow 1$: we want to use the information in $\tilde{c}_t(j)$ to update memory
    - $i_t(t) \rightarrow 0$: $\tilde{c}_t(j)$ should not contribute to memory





Sigmoid

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

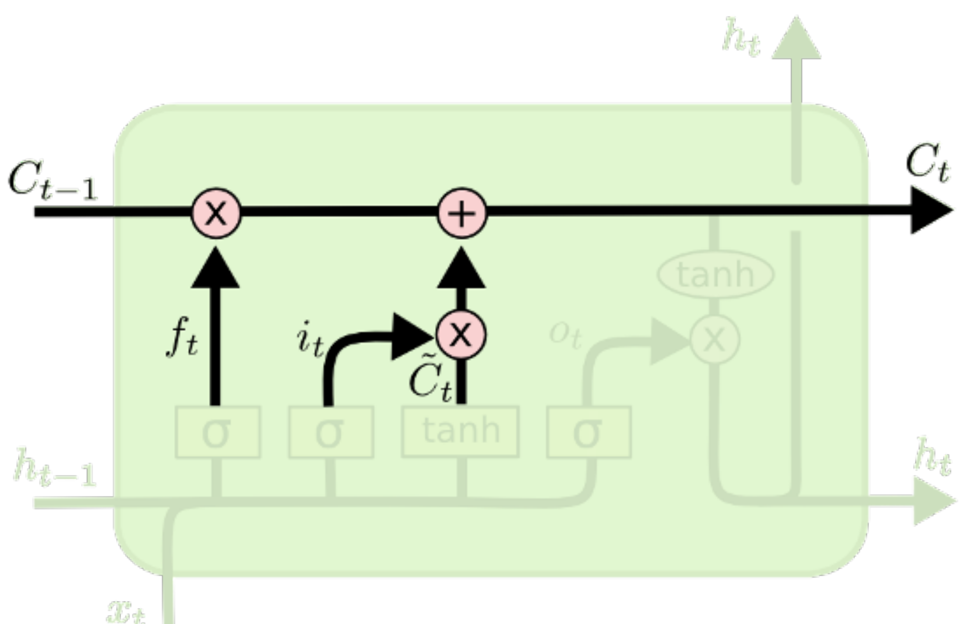
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

$i_t \otimes \tilde{C}_t$

Long Short-Term Memory Network

Memory update

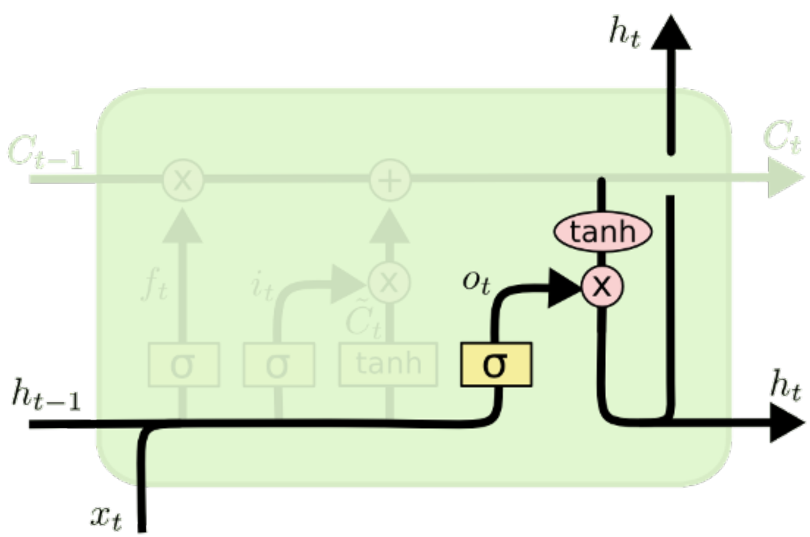
- $c_t = f_t \otimes c_{t-1} + i_t \otimes \tilde{c}_t$
- f_t forget gate; i_t input date
- $f_t \otimes c_{t-1}$: drop useless information in old memory
- $i_t \otimes \tilde{c}_t$: add selected new information from current input



Long Short-Term Memory Network

Output gate o_t

- Next hidden state $h_t = o_t \odot \tanh(c_t)$
  - $\tanh(c_t)$: non-linear transformation over all past information
  - o_t: choose important dimensions for the next state
    - $o_t(j) \to 1$: $\tanh(c_t(j))$ is important for the next state
    - $o_t(j) \to 0$: $\tanh(c_t(j))$ is not important

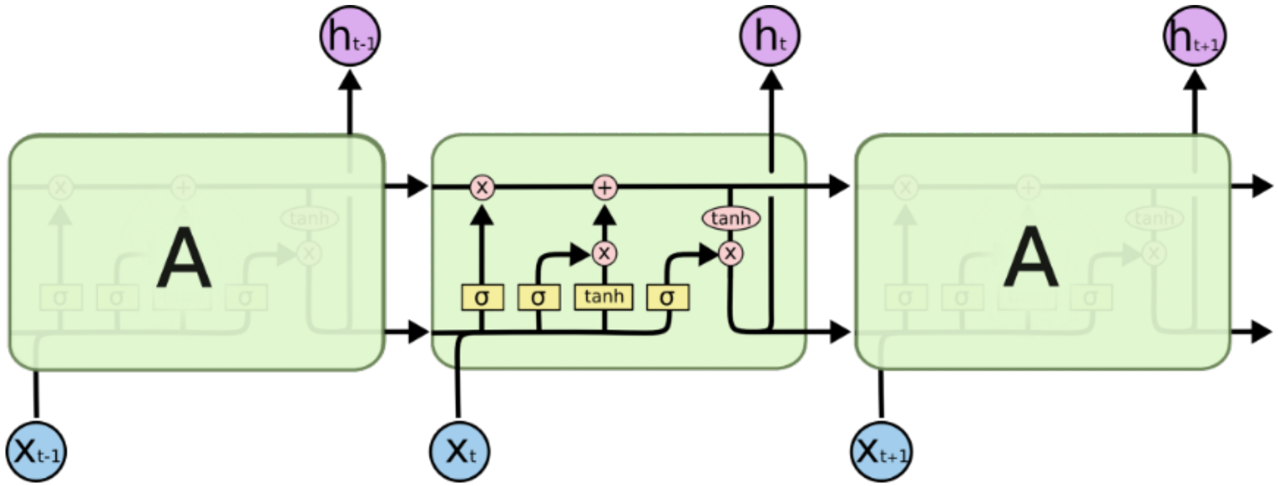


$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

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

Long Short-Term Memory Network

- $h_t = o_t \odot \tanh(c_t)$
- $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$
- $Y_t = g(h_t)$



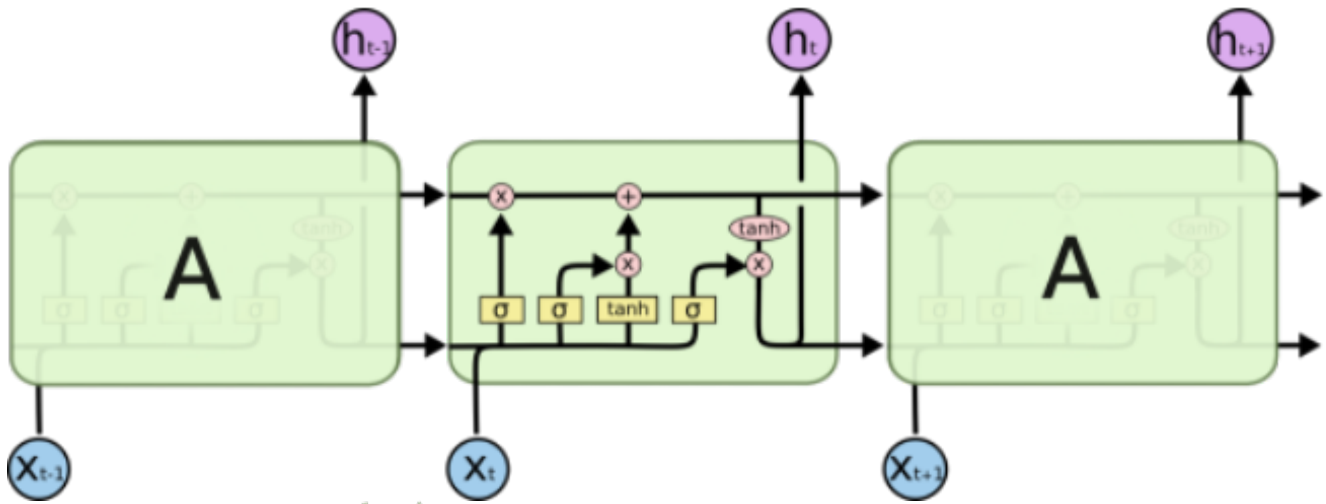
Remarks:

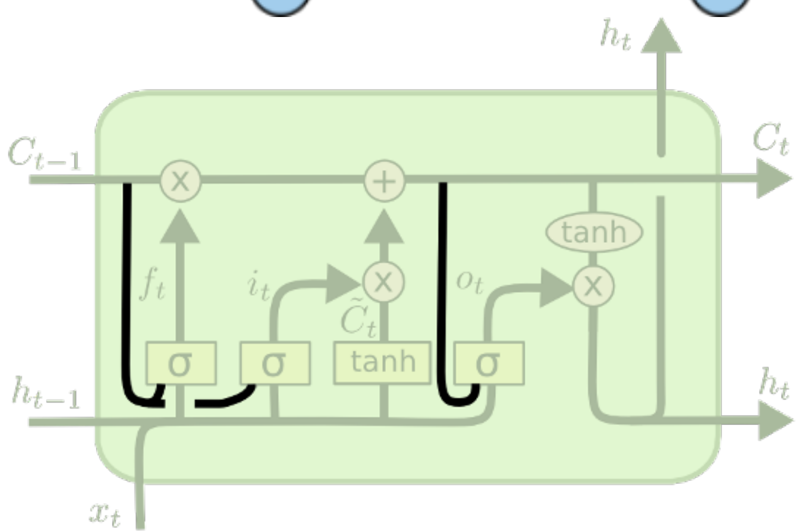
1. No more matrix multiplications for c_t
2. LSTM does not have guarantees for gradient explosion/vanishing
3. LSTM is the dominant architecture for sequence modeling from ’13 - ’16.
4. Why tanh

LSTM Variant

Peephold Connections (Gers & Schmidhuber '00)

- Allow gates to take in c_t information



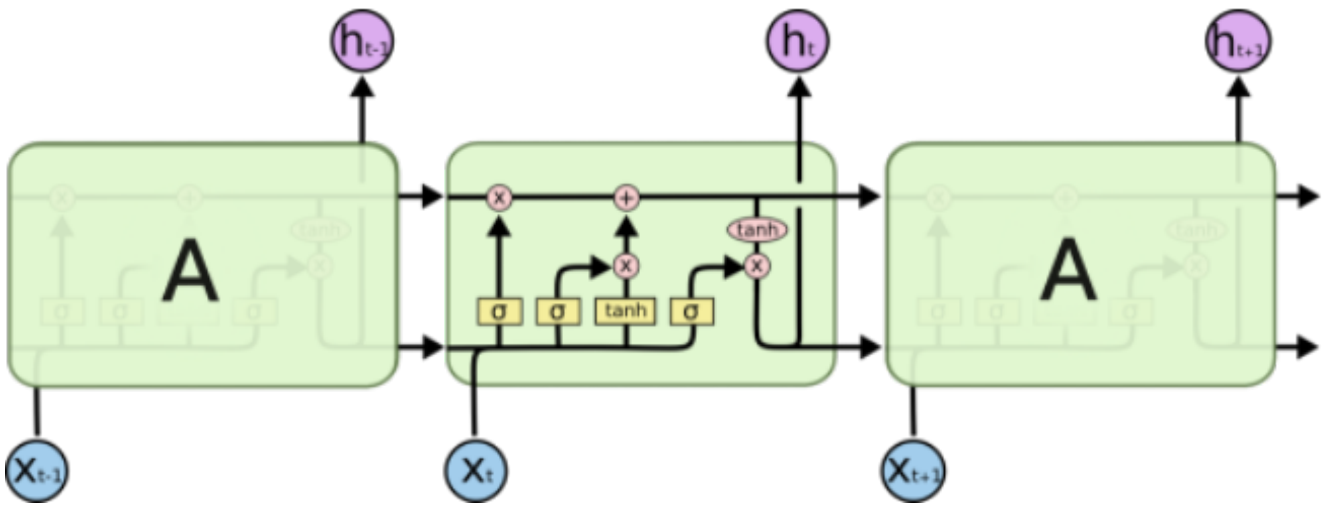


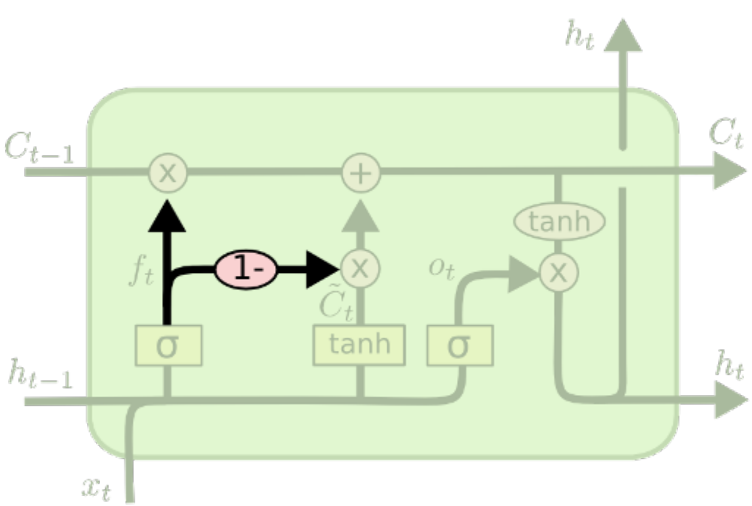
$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \ + \ b_o\right)$$

LSTM Variant

Simplified LSTM

- Assume $i_t = 1 - f_t$
- Only two gates are needed: fewer parameters



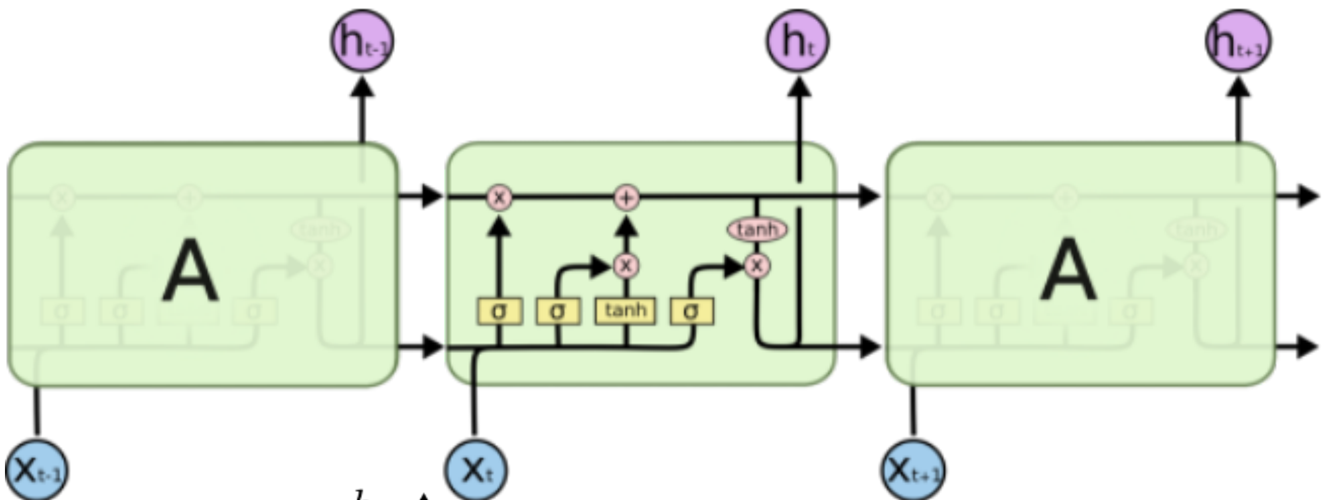


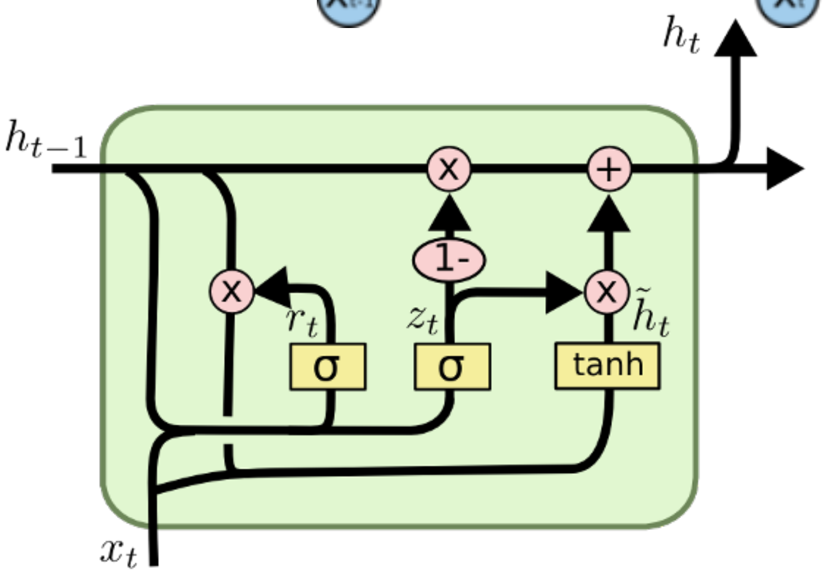
$$C_t = f_t * C_{t-1} + (\mathbf{1} - \boldsymbol{f_t}) * \tilde{C}_t$$

LSTM Variant

Gated Recurrent Unit (GRU, Cho et al. '14)

- Merge h_t and c_t: much fewer parameters





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

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

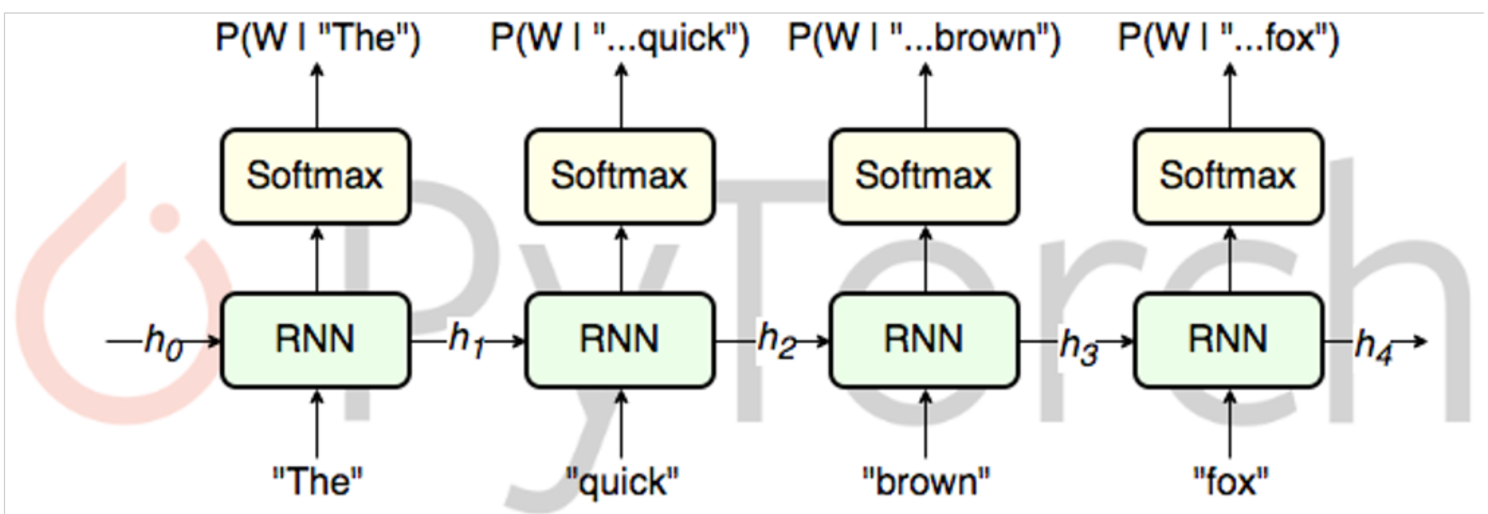
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

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

LSTM application: language model

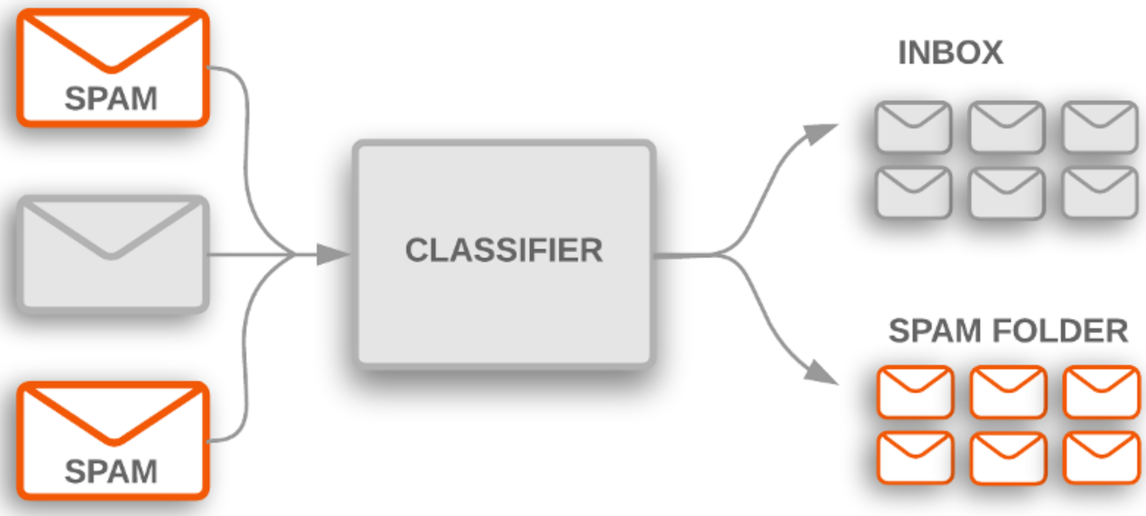
- Autoregressive language model: $\boxed{P(X;\theta)} = \boxed{\Pi_{t=1}^{L} P(X_t \mid X_{i<t};\theta)}$
  - X: a sentence
  - Sequential generation
- LSTM language model
  - X_t: word at position t.
  - Y_t: softmax over all words
- Data: a collection of texts:
  - Wiki

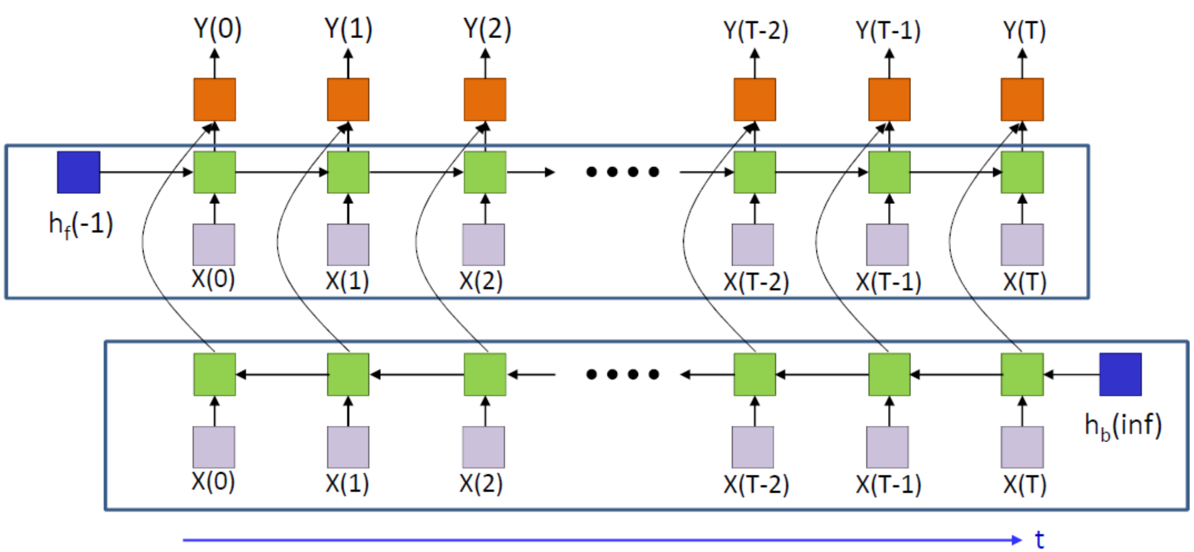
LSTM



LSTM application: text classification

Bi-dreictional LSTM and them run softmax on the final hidden state.





Attention Mechanism

Machine Translation

- Before 2014: Statistical Machine Translation (SMT)
  - Extremely complex systems that require massive human efforts
  - Separately designed components
  - A lot of feature engineering
  - Lots of linguistic domain knowledge and expertise

- Before 2016:
  - Google Translate is based on statistical machine learning

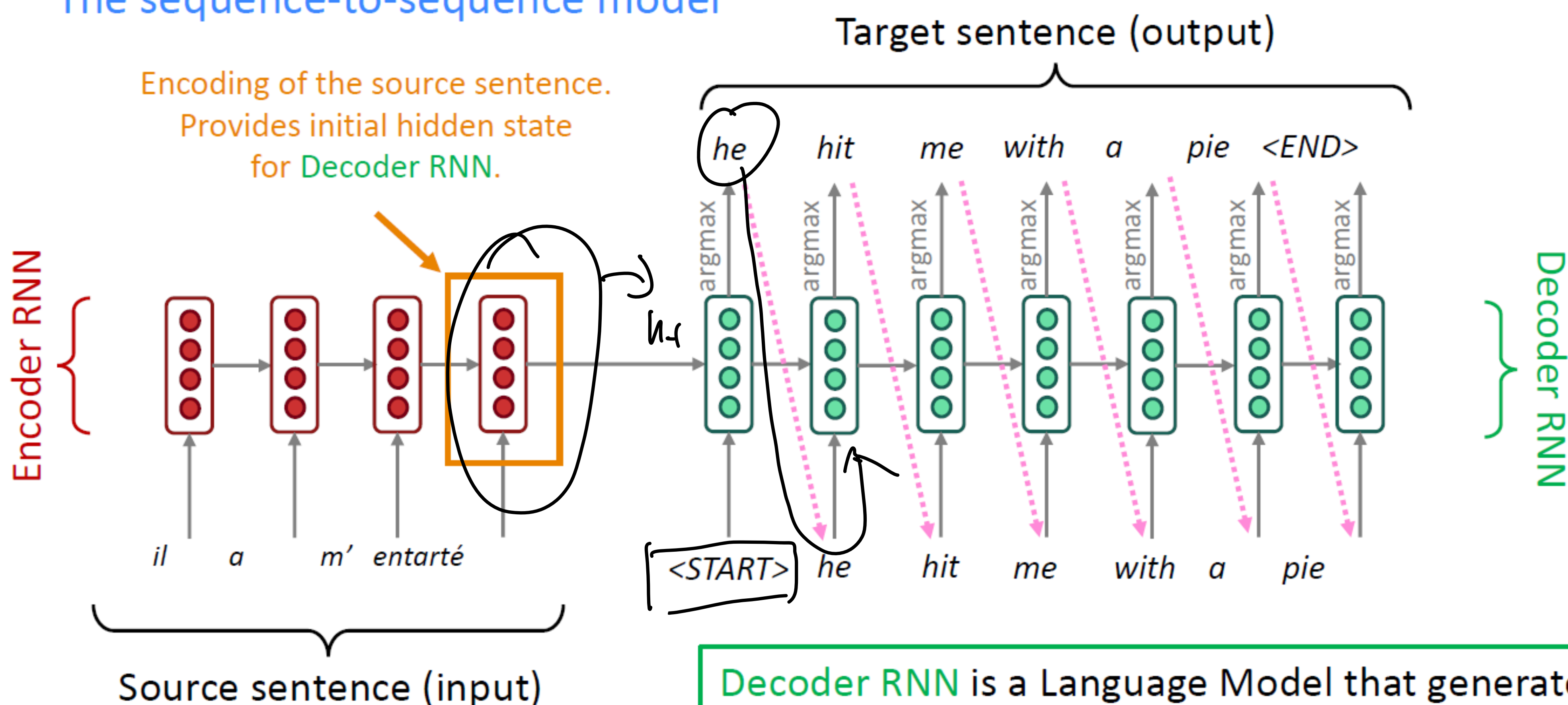
- What happened in 2014?
  - Neural machine translation (NMT)

Sequence to Sequence Model

- Neural Machine Translation (NMT)
  - Learning to translate via a **single end-to-end** neural network.
  - Source language sentence X, target language sentence $Y = f(X; \theta)$
- Sequence to Sequence Model (Seq2Seq, Sutskever et al. , '14)
  - Two RNNs: f_{enc} and f_{dec}
  - Encoder f_{enc}:
    - Takes X as input, and output the initial hidden state for decoder
    - Can use bidirectional RNN
  - Decoder f_{dec}:
    - It takes in the hidden state from f_{enc} to generate Y
    - Can use autoregressive language model

Sequence to Sequence Model

The sequence-to-sequence model



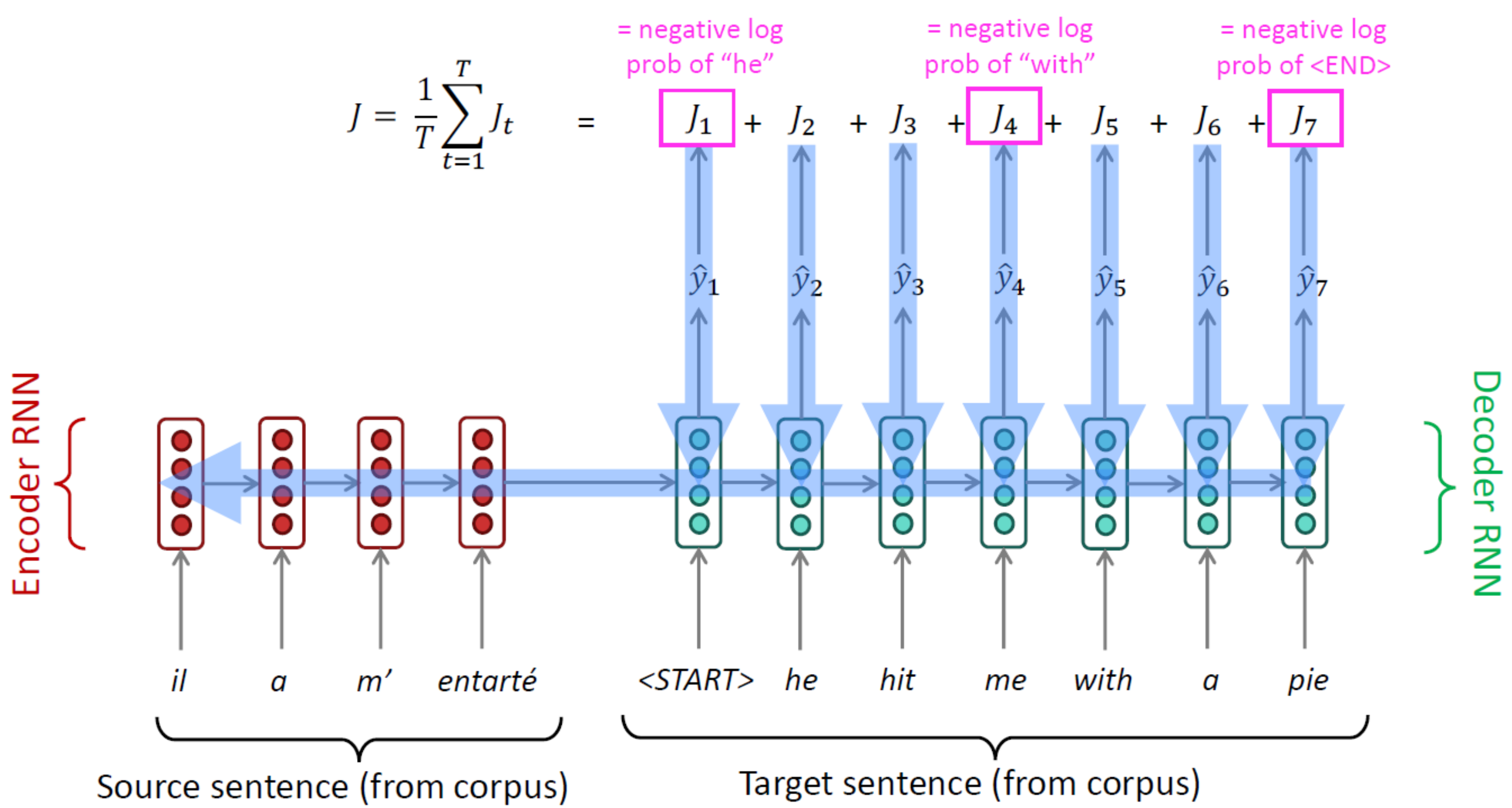
Decoder RNN is a Language Model that generates target sentence, *conditioned on encoding*.

Encoder RNN produces an encoding of the source sentence.

Note: This diagram shows **test time** behavior: decoder output is fed in as next step's input

Training Sequence to Sequence Model

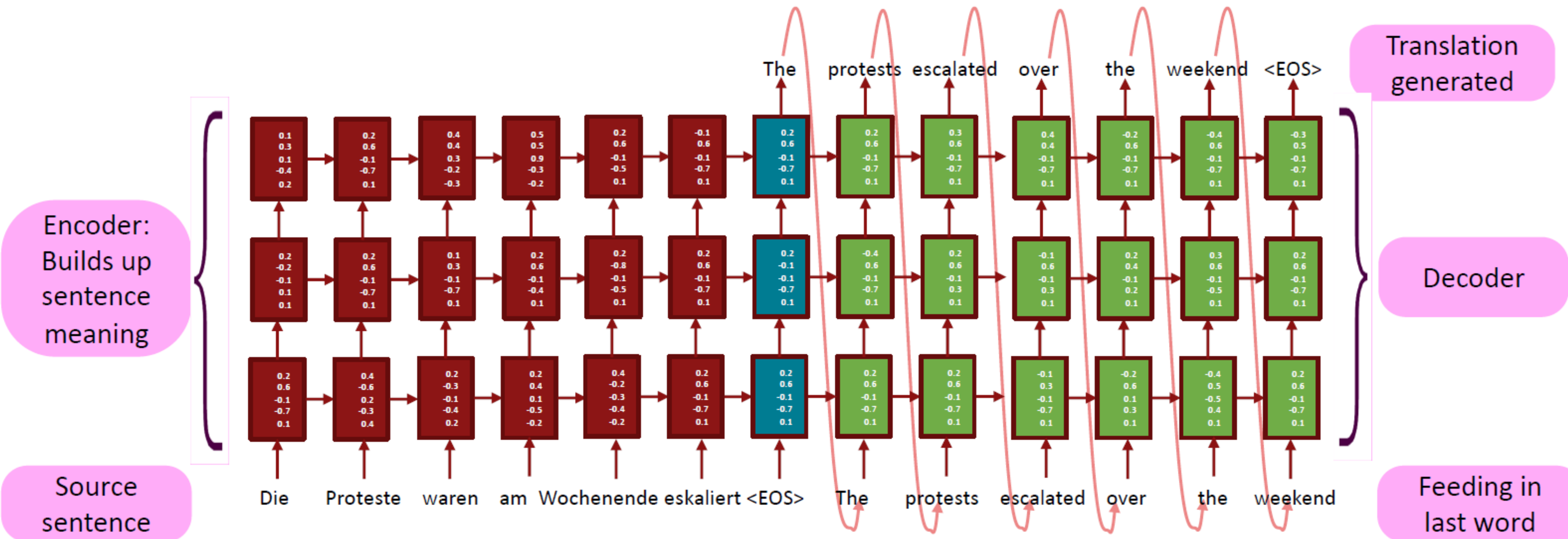
- Collect a huge paired dataset and train it end-to-end via BPTT
- Loss induced by MLE $P(Y|X) = P(Y|f_{enc}(X))$



Seq2seq is optimized as a **single system.** Backpropagation operates *"end-to-end"*.

Deep Sequence to Sequence Model

- Stacked seq2seq model



Machine Translation

- 2016: Google switched Google Translate from SMT to NMT