Advanced NN Architectures

Recurrent Neural Networks

Learning goals

- Understand recurent structure of RNNs
- Learn the different types of RNNs
- Understand applicability of RNNs

WHAT ARE RNNS?

- Assumption: Text is written sequentially, so our model should read it sequentially
- "RNN": class of Neural Network architectures that process text sequentially (left-to-right or right-to-left)
- Generally speaking:
  - Internal "state" $\vec{h}$
  - RNN consumes one input $\vec{x}^{(j)}$ per time step j
  - Update function: $\vec{h}^{(j)} = f(\vec{x}^{(j)}, \vec{h}^{(j-1)}; \theta)$
    - where parameters θ are shared across all time steps

VANILLA RNN (1)

- Let $\vec{x}^{(1)} \ldots \vec{x}^{(J)}$, with $\mathbf{x}^{(j)} \in \mathbb{R}^{d'}$ be our input (e.g., a sequence of d'-dimensional word embeddings)
- Let $\mathbf{h}^{(0)} = \{0\}^d$ be our initial state
- Let $\theta = \{\vec{W} \in \mathbb{R}^{d \times d'}, \vec{V} \in \mathbb{R}^{d \times d}, \vec{b} \in \mathbb{R}^d\}$ be our parameters
- Then the j'th update is defined as:

$$\vec{h}^{(j)} = \tanh(\vec{W}\vec{x}^{(j)} + \vec{V}\vec{h}^{(j-1)} + \vec{b})$$

VANILLA RNN (2)

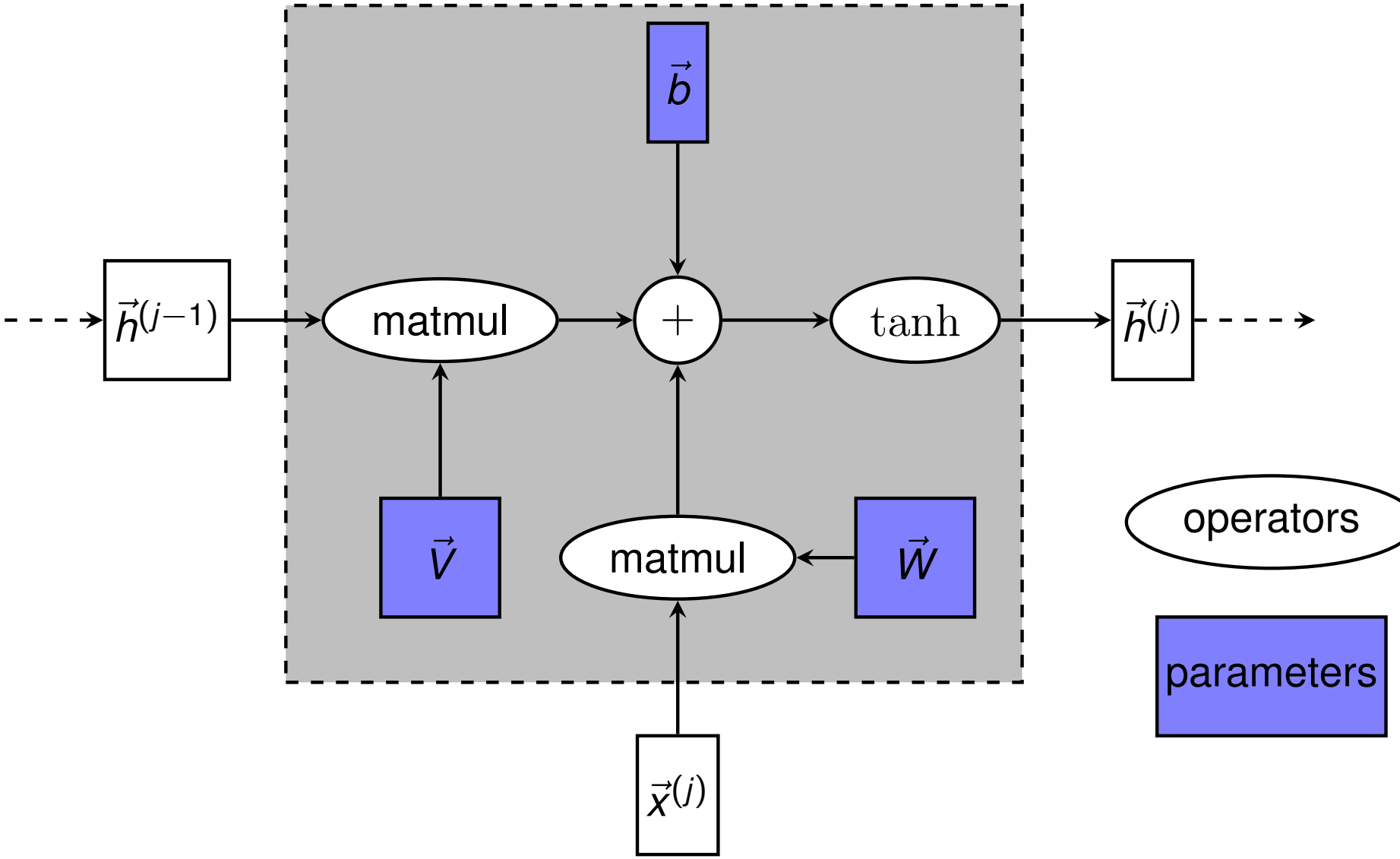
- Sentence: “the cat sat”
- $(\vec{x}^{(1)}, \vec{x}^{(2)}, \vec{x}^{(3)}) = (\vec{x}^{(\text{the})}, \vec{x}^{(\text{cat})}, \vec{x}^{(\text{sat})})$

$$\vec{h}^{(1)} = \tanh(\vec{W}\vec{x}^{(\text{the})} + \vec{V}\vec{h}^{(0)} + \vec{b}) = \tanh(\vec{W}\vec{x}^{(\text{the})} + \vec{b})$$
$$\vec{h}^{(2)} = \tanh(\vec{W}\vec{x}^{(\text{cat})} + \vec{V}\vec{h}^{(1)} + \vec{b}) = \tanh(\vec{W}\vec{x}^{(\text{cat})} + \vec{V}\tanh(\vec{W}\vec{x}^{(\text{the})} + \vec{b}) + \vec{b})$$
$$\vec{h}^{(3)} = \tanh(\vec{W}\vec{x}^{(\text{sat})} + \vec{V}\vec{h}^{(2)} + \vec{b}) = \ldots$$

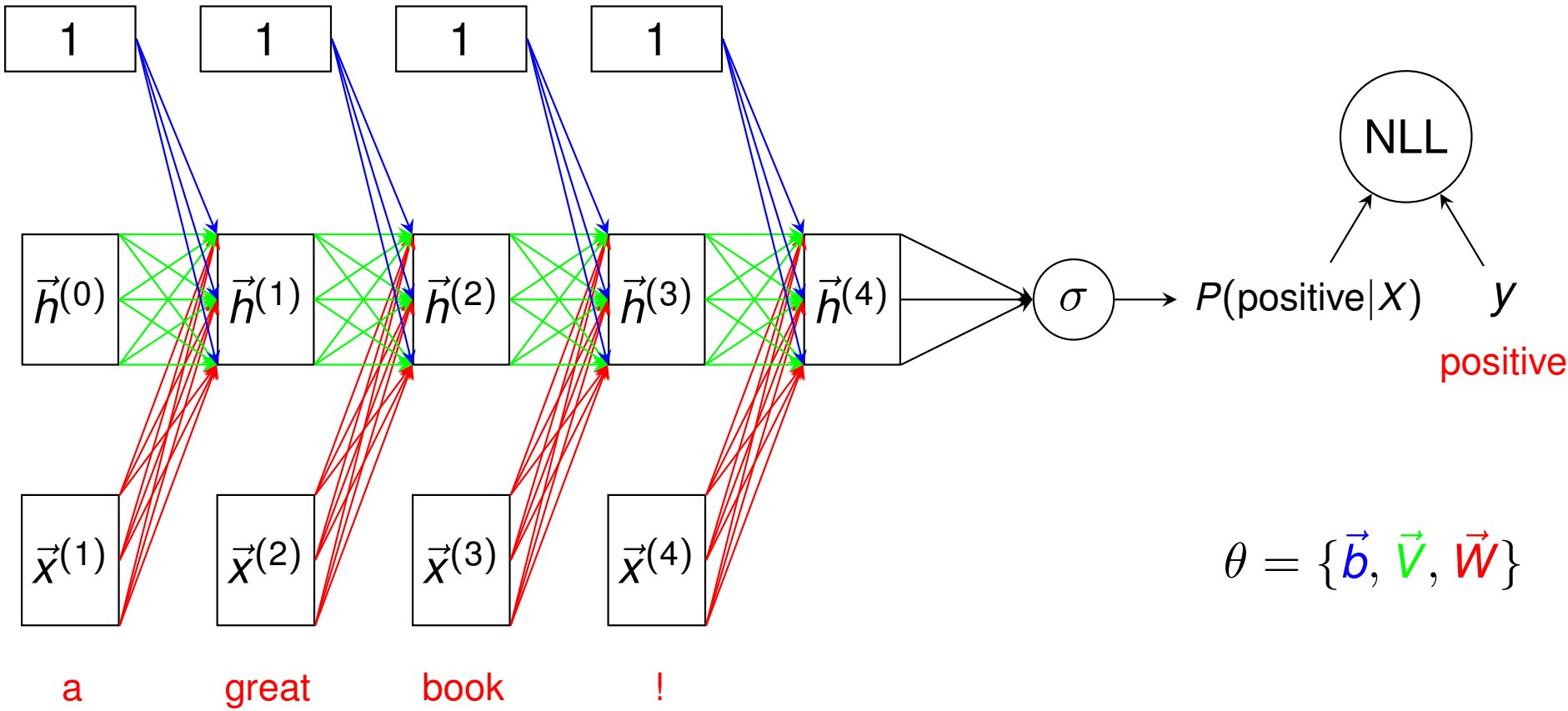
- Question: Which word(s) does $\vec{h}^{(1)}$ depend on?
  - “the”
- Question: Which word(s) does $\vec{h}^{(2)}$ depend on?
  - “the cat”
- Question: Which word(s) does $\vec{h}^{(3)}$ depend on?
  - “the cat sat”

VANILLA RNN CELL

VANILLA RNN CELL



EXAMPLE: BINARY SENTIMENT CLASSIFIER



$$\theta = \{\vec{b}, \vec{V}, \vec{W}\}$$

(Non-linearity omitted for readability.)

BACKPROPAGATION THROUGH TIME

- RNNs are trained via “backpropagation through time”
- To understand how this works, imagine the RNN as a Feed-Forward Net (FFN), whose depth is equal to the sentence length
- For now, let’s pretend that every time step (every layer) has its own dummy parameters $\theta^{(j)}$, which are identical copies of θ

VANISHING GRADIENTS (1)

- On long inputs, Vanilla RNNs suffer from “vanishing” gradients
- Vanishing gradients mean that the impact that an input has on the gradient of the loss becomes smaller when it is further away from the loss in the computation graph.

VANISHING GRADIENTS (2)

- We assume that we have backpropagated the partial derivatives of the loss to $\vec{h}^{(J)}$:

$$\frac{\partial L}{\partial \vec{h}^{(J)}}$$

- Backpropagate to $\vec{h}^{(j)}$ via chain rule:

$$\frac{\partial L}{\partial \vec{h}^{(j)}} = \Big[\prod_{j'=j+1}^{J} \Big(\frac{\partial \vec{h}^{(j')}}{\partial \vec{h}^{(j'-1)}} \Big)^T \Big] \frac{\partial L}{\partial \vec{h}^{(J)}}$$

$$= \Big[\prod_{j'=j+1}^{J} \Big(\frac{\partial \vec{V} \vec{h}^{(j'-1)}}{\partial \vec{h}^{(j'-1)}} \Big)^T \frac{\partial \tanh\big(\vec{V} \vec{h}^{(j'-1)} + \ldots \big)}{\partial \vec{V} \vec{h}^{(j'-1)}} \Big] \frac{\partial L}{\partial \vec{h}^{(J)}}$$

VANISHING GRADIENTS (3)

$$\frac{\partial L}{\partial \vec{h}^{(j)}} = \Big[\prod_{j'=j+1}^{J} \Big(\frac{\partial \vec{V} \vec{h}^{(j'-1)}}{\partial \vec{h}^{(j'-1)}} \Big)^T \frac{\partial \tanh(\vec{V} \vec{h}^{(j'-1)} + \ldots)}{\partial \vec{V} \vec{h}^{(j'-1)}} \Big] \frac{\partial L}{\partial \vec{h}^{(J)}}$$

- What happens to $\frac{\partial L}{\partial \vec{h}^{(j)}}$ when the distance $J - j$ grows?
  - Remember that $\tanh$ is applied elementwise, and that the derivative of $\tanh$ is between 0 an 1. So the red Jacobian matrix is a diagonal matrix with entries between 0 and 1. The product of many such matrices approaches zero.
  - Furthermore, the blue Jacobian matrix is just $\vec{V}$. When initialized with small enough values, $\prod \vec{V}$ will approach zero as well.
  - As a result, $\frac{\partial L}{\partial \vec{h}^{(j)}}$ approaches zero (“vanishes”)

VANISHING GRADIENTS (4)

- What does this mean?
  - Since the "dummy parameter gradients" of step j, $\nabla_{\theta^{(j)}} L$ are upstream from $\frac{\partial L}{\partial \vec{h}^{(j)}}$, they approach zero too, i.e., their effect on the "dummy gradient sum" is negligible.
  - This means that if the words that your RNN should be paying attention to are far from the loss, the network will not (or slowly) adjust its weights to those words

EXPLODING GRADIENTS

- So why don't we just use a nonlinearity with a derivative larger than 1, or initialize $\vec{V}$ differently?
  - $||\frac{\partial L}{\partial \vec{h}^{(j)}}||$ would become very large ("explode"). This is even worse than vanishing gradients, because it leads to non-convergence of gradient descent.
  - So vanishing gradients is the lesser of two evils.

LONG-SHORT TERM MEMORY NETWORK (1)

- Proposed in Hochreiter and Schmidhuber, 1997
- Became popular around 2010 for handwriting recognition, speech recognition, and many NLP problems
- Addresses vanishing gradients by changing the architecture of the RNN cell

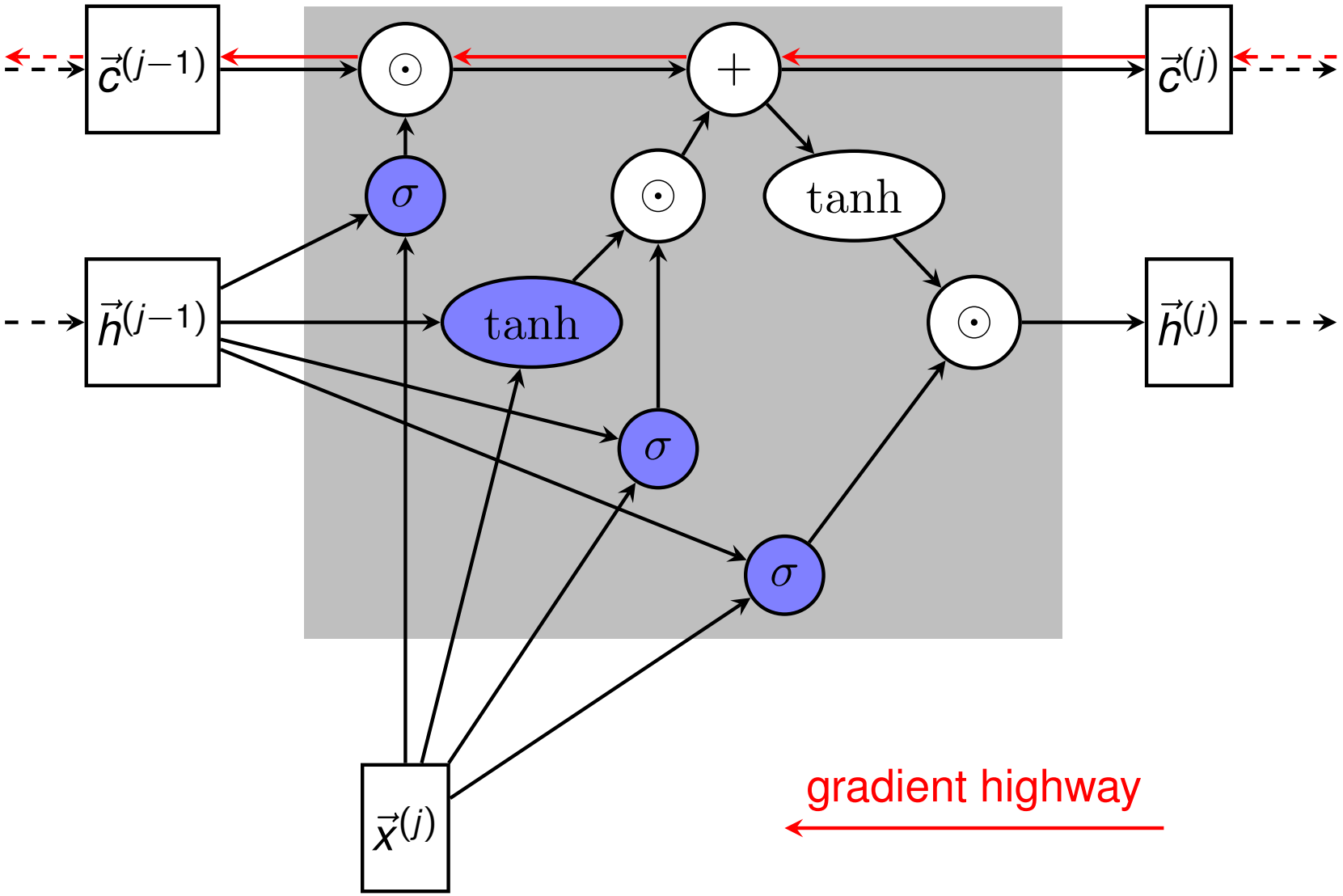
LONG-SHORT TERM MEMORY NETWORK (2)

- Two states: $\vec{h}$ ("short-term memory") and $\vec{c}$ ("long-term memory")
- Candidate state $\vec{\bar{h}} \in \mathbb{R}^d$ corresponds to $\vec{h}$ in the Vanilla RNN
- Interactions are mediated by "gates" $\in (0,1)^d$, which apply elementwise:
  - Forget gate $\vec{f}$ decides what information from $\vec{c}$ should be forgotten
  - Input gate $\vec{i}$ decides what information from $\vec{\bar{h}}$ should be added to $\vec{c}$
  - Output gate $\vec{o}$ decides what information from $\vec{c}$ should be exposed to $\vec{h}$
- Each gate and the candidate state have their own parameters $\theta^{(i)}, \theta^{(f)}, \theta^{(o)}, \theta^{(\bar{h})}$
- "Gradient highway" from $\vec{c}^{(j)}$ to $\vec{c}^{(j-1)}$, with no non-linearities or matrix multiplications

LSTM DEFINITION

$$
\begin{aligned}
\vec{h}^{(0)} &= \vec{c}^{(0)} = \{0\}^d \\
\vec{f}^{(j)} &= \sigma(\vec{W}^{(f)}\vec{x}^{(j)} + \vec{V}^{(f)}\vec{h}^{(j-1)} + \vec{b}^{(f)}) \\
\vec{i}^{(j)} &= \sigma(\vec{W}^{(i)}\vec{x}^{(j)} + \vec{V}^{(i)}\vec{h}^{(j-1)} + \vec{b}^{(i)}) \\
\vec{o}^{(j)} &= \sigma(\vec{W}^{(o)}\vec{x}^{(j)} + \vec{V}^{(o)}\vec{h}^{(j-1)} + \vec{b}^{(o)}) \\
\vec{\bar{h}}^{(j)} &= \tanh(\vec{W}^{(\bar{h})}\vec{x}^{(j)} + \vec{V}^{(\bar{h})}\vec{h}^{(j-1)} + \vec{b}^{(\bar{h})}) \\
\vec{c}^{(j)} &= \vec{f}^{(j)} \odot \vec{c}^{(j-1)} + \vec{i}^{(j)} \odot \vec{\bar{h}}^{(j)} \\
\vec{h}^{(j)} &= \vec{o}^{(j)} \odot \tanh(\vec{c}^{(j)})
\end{aligned}
$$

LSTM CELL


$\vec{c}(j-1)$
$\vec{c}(j)$
σ
tanh
$\vec{h}(j-1)$
tanh
$\vec{h}(j)$
σ
σ
$\vec{x}(j)$
gradient highway

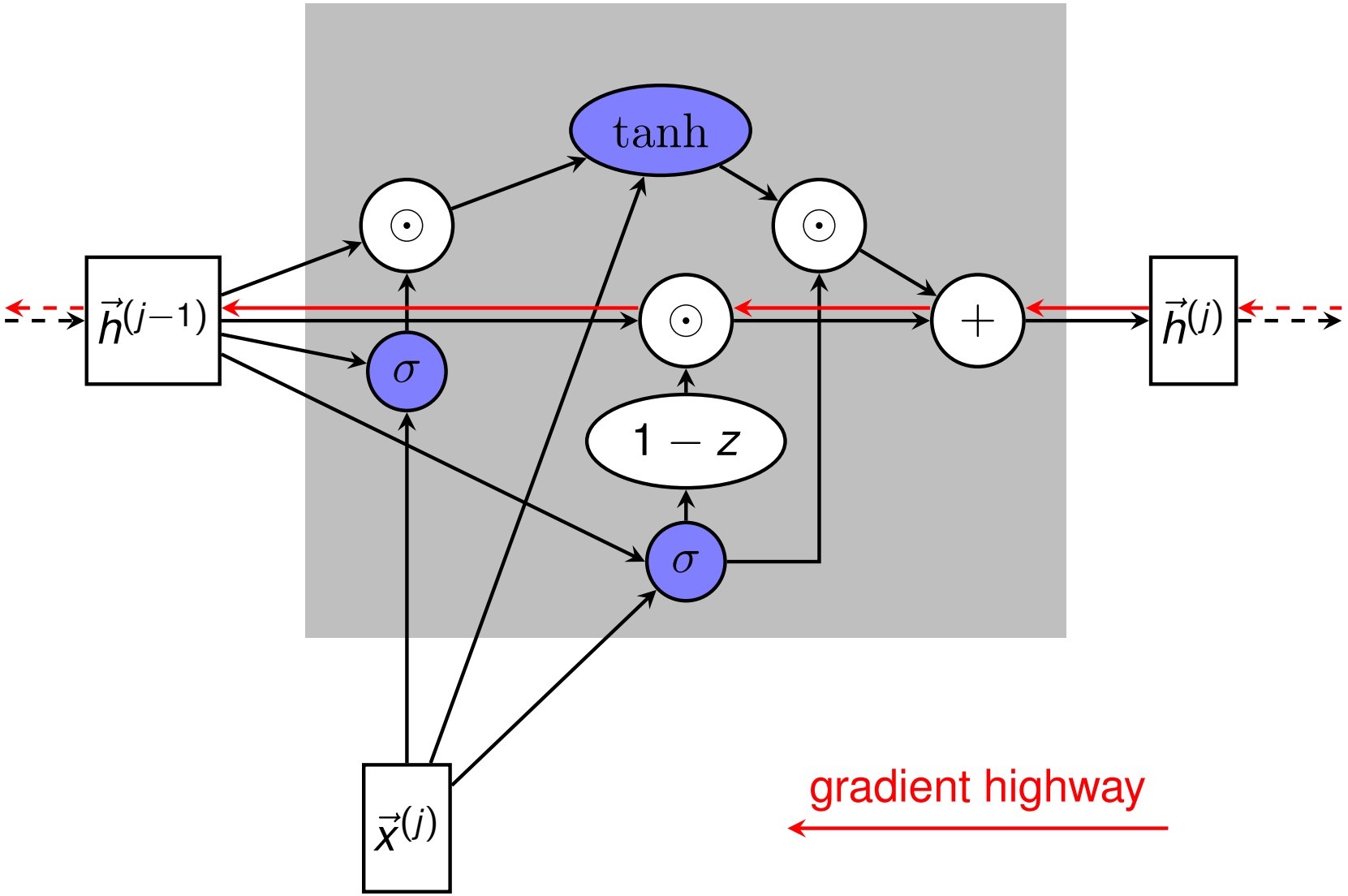
GATED RECURRENT UNIT (GRU)

- Proposed by Cho et al. (2014)
- Lightweight alternative to LSTM, with only one state and three sets of parameters
- State $\vec{h}$ is a dynamic "interpolation" of long and short term memory
- Reset gate $\vec{r} \in (0,1)^d$ controls what information passes from $\vec{h}$ to candidate state $\vec{\bar{h}}$
- Update gate $\vec{z} \in (0,1)^d$ interpolates between $\vec{h}$ and $\vec{\bar{h}}$
- Separate set of parameters $\theta^{(r)}, \theta^{(z)}, \theta^{(\bar{h})}$ for each gate and the candidate state.

GRU DEFINITION

$$
\begin{aligned}
\vec{h}^{(0)} &= \{0\}^d \\
\vec{r}^{(j)} &= \sigma(\vec{W}^{(r)}\vec{x}^{(j)} + \vec{V}^{(r)}\vec{h}^{(j-1)} + \vec{b}^{(r)}) \\
\vec{z}^{(j)} &= \sigma(\vec{W}^{(z)}\vec{x}^{(j)} + \vec{V}^{(z)}\vec{h}^{(j-1)} + \vec{b}^{(z)}) \\
\vec{\bar{h}}^{(j)} &= \tanh(\vec{W}^{(\bar{h})}\vec{x}^{(j)} + \vec{V}^{(\bar{h})}(\vec{r}^{(j)} \odot \vec{h}^{(j-1)}) + \vec{b}^{(\bar{h})}) \\
\vec{h}^{(j)} &= (1 - \vec{z}^{(j)}) \odot \vec{h}^{(j-1)} + \vec{z}^{(j)} \odot \vec{\bar{h}}^{(j)}
\end{aligned}
$$

GRU CELL



VANILLA VS. GRU VS. LSTM

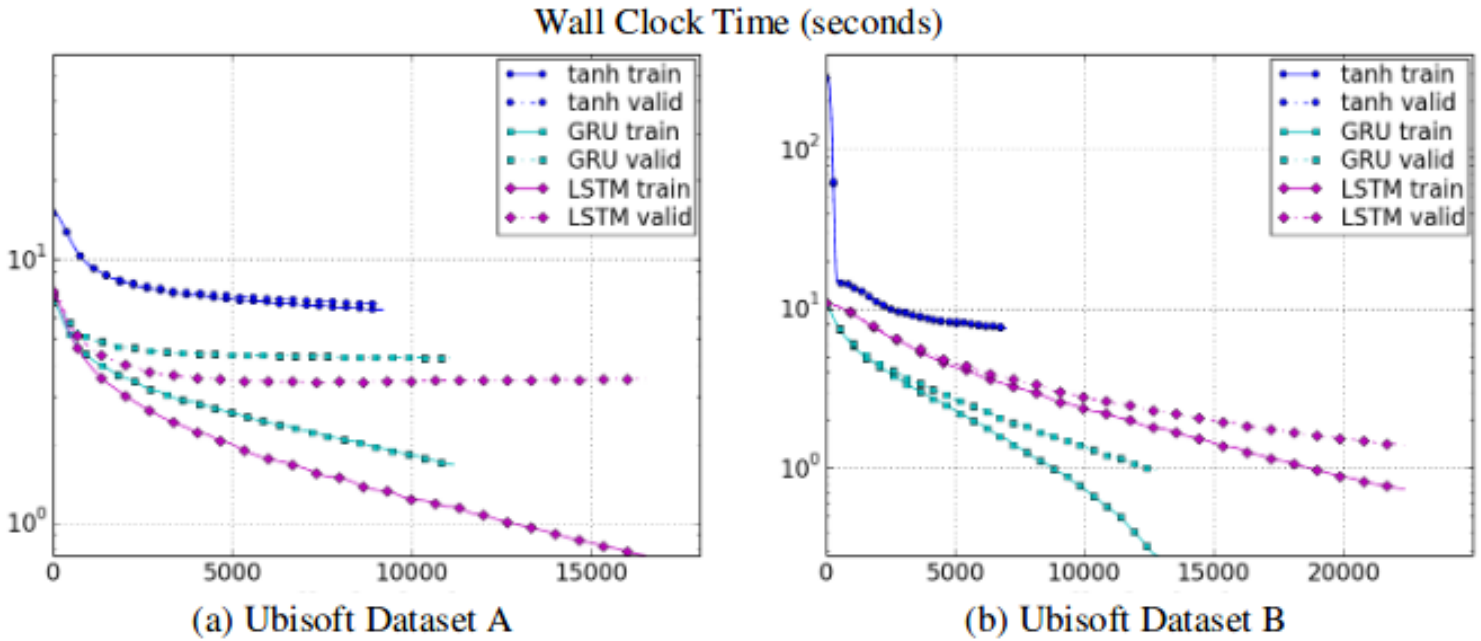
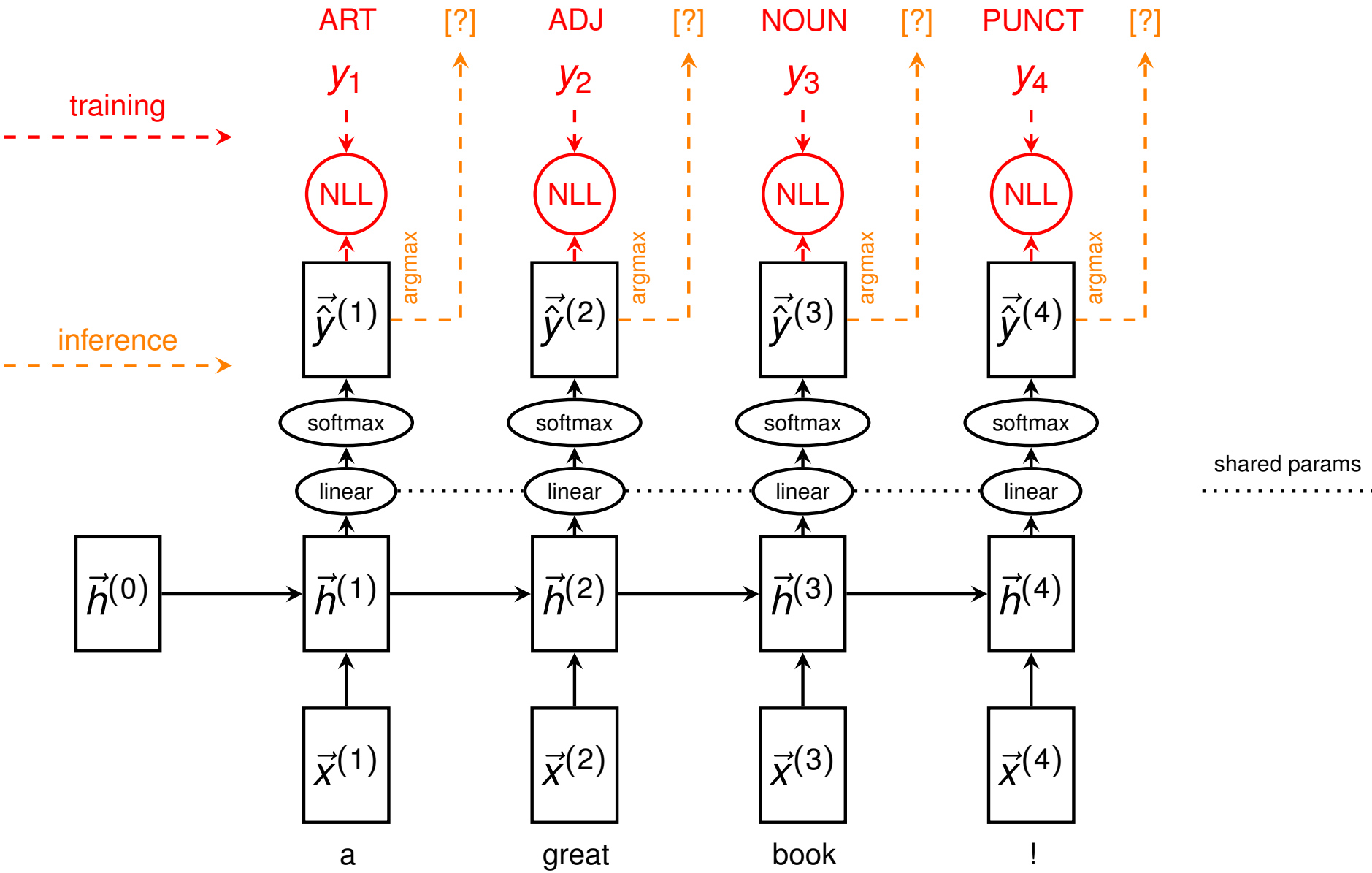
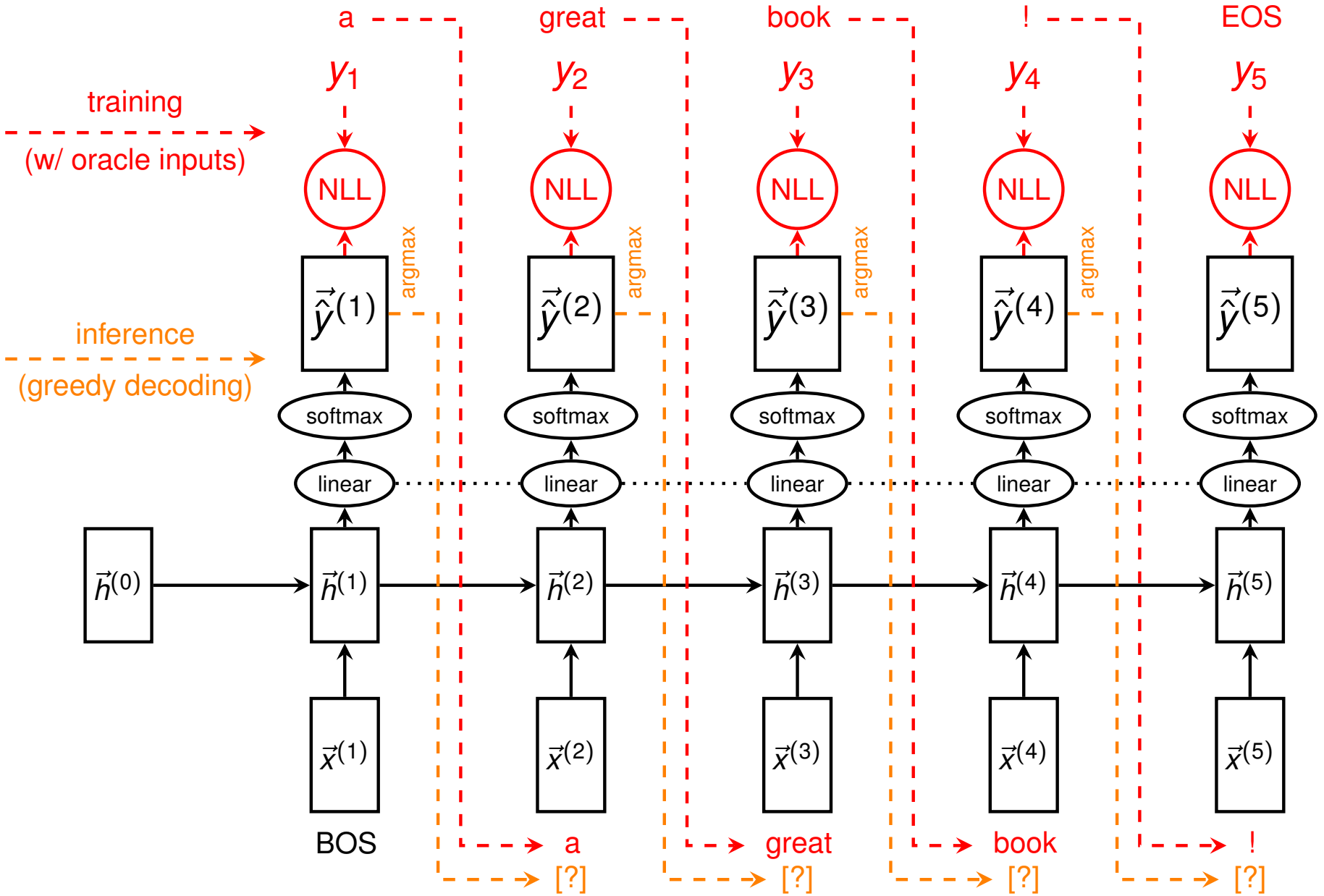


Figure from Chung et al. 2014: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling

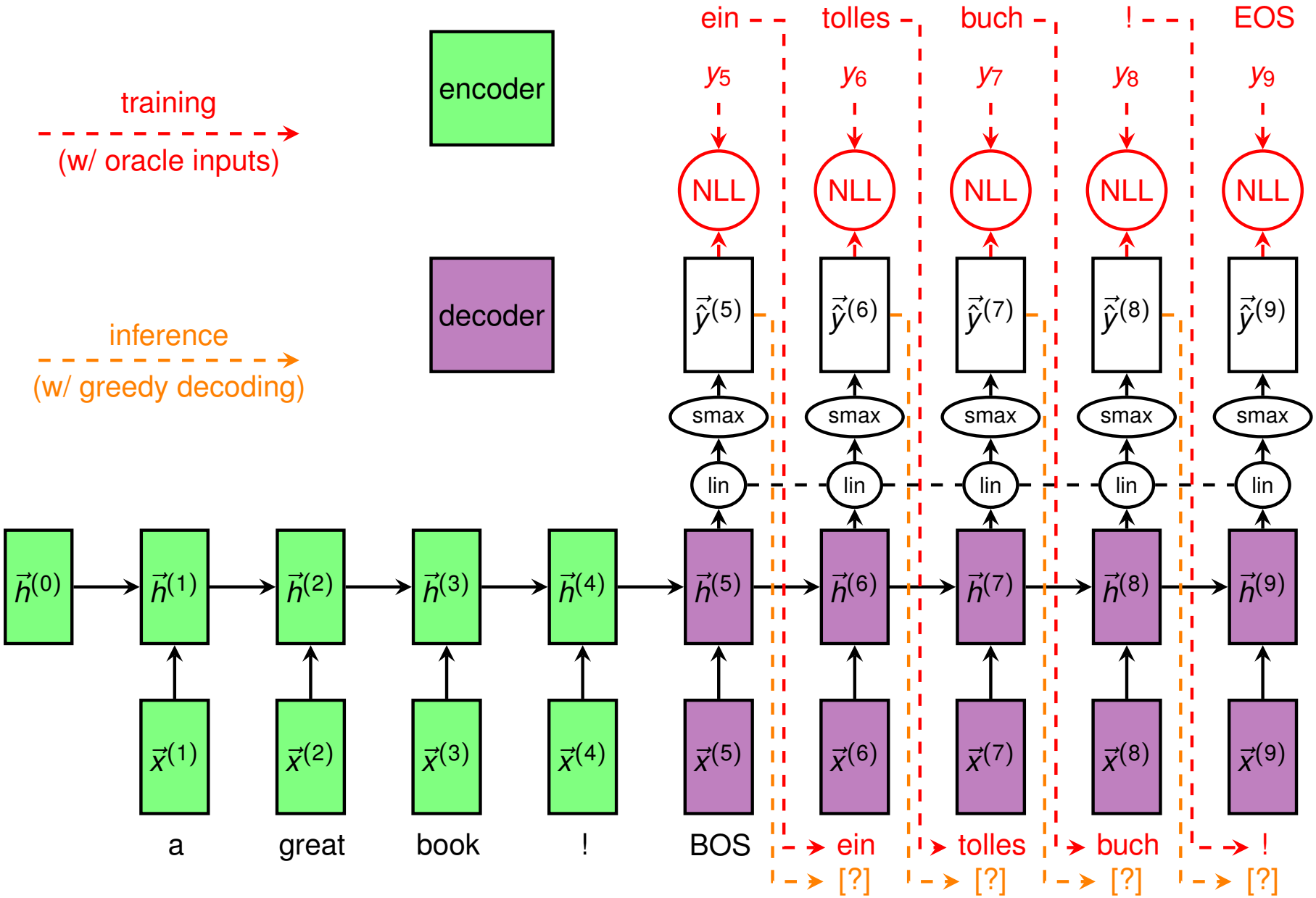
TAGGING (EXAMPLE: PART-OF-SPEECH)



AUTOREGRESSIVE LANGUAGE MODELING



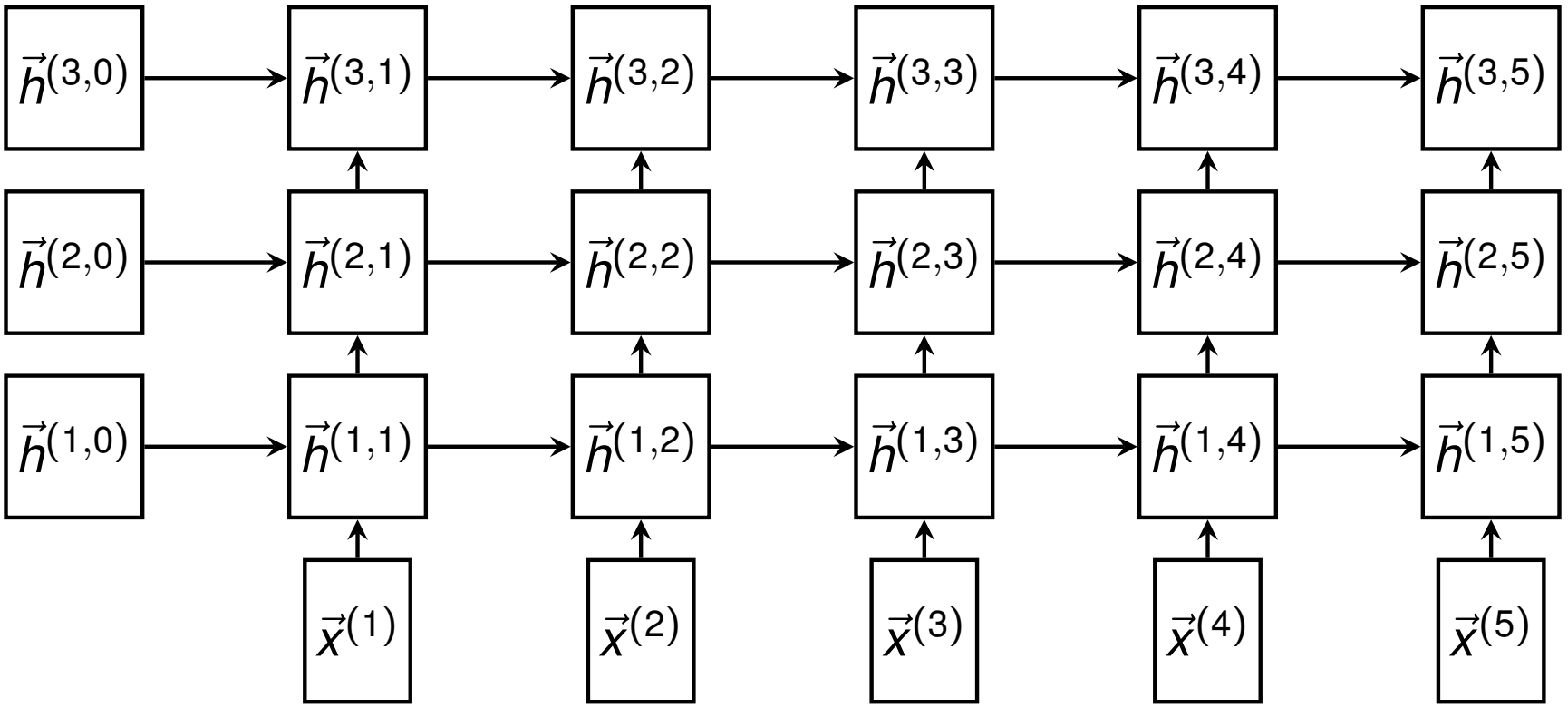
SEQ-TO-SEQ (MACHINE TRANSLATION)



MULTI-LAYER RNNS (1)

- Stack of several RNNs (Vanilla RNNs, LSTMs, GRUs, etc.)
- Each RNN in the stack has its own parameters
- The input vectors of the l'th RNN are the hidden states of the $l-1$'th RNN
- The input vectors to the first RNN are the word embeddings, as usual
- We can output the hidden states of the last RNN, or a combination (concatenation, average, etc.) of the states of all RNNs.

MULTI-LAYER RNNS (2)



BIDIRECTIONAL RNNS (1)

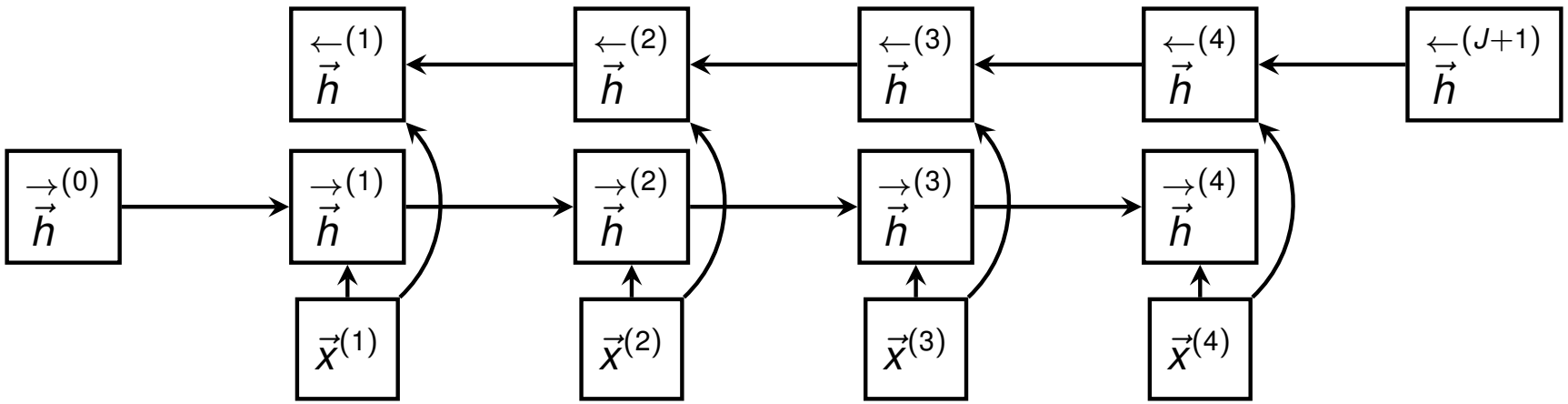
- Two RNNs with separate parameters $\overrightarrow{\theta}, \overleftarrow{\theta}$
- Let $\vec{x}^{(1)} \ldots \vec{x}^{(J)}$ be our input
- Let $\overrightarrow{\vec{h}}^{(0)} = \overleftarrow{\vec{h}}^{(J+1)} = \{0\}^d$ be our initial states.
- The forward RNN runs left-to-right over the input:

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

- The backward RNN runs right-to-left over the input:

$$\overleftarrow{\vec{h}}^{(j)} = f(\vec{x}^{(j)}, \overleftarrow{\vec{h}}^{(j+1)}; \overleftarrow{\theta})$$

BIDIRECTIONAL RNNS (2)



BIDIRECTIONAL RNNS (3)

- The bidirectional RNN yields two sequences of hidden states:

$$(\vec{h}^{\rightarrow(1)} \ldots \vec{h}^{\rightarrow(J)}), (\vec{h}^{\leftarrow(1)} \ldots \vec{h}^{\leftarrow(J)})$$

- Question: If we are dealing with a sentence classification task, which states should we use to represent the sentence?
  - Concatenate $[\vec{h}^{\rightarrow(J)}; \vec{h}^{\leftarrow(1)}]$, because they have "seen" the entire sentence
- For tagging task, represent the j'th word as $[\vec{h}^{\rightarrow(j)}; \vec{h}^{\leftarrow(j)}]$

BIDIRECTIONAL RNNS (4)

- Question: Can we use a bidirectional RNN for autoregressive language modeling?
  - No. In autoregressive language modeling, future inputs must be unknown to the model (since we want to learn to predict them).
  - We could train two separate autoregressive RNNs (one per direction), but we cannot combine their hidden states before making a prediction
- In sequence-to-sequence (e.g., Machine Translation), the encoder can be bidirectional, but the decoder cannot (same reason)