CS5740: Natural Language Processing

# Recurrent Neural Networks

Instructor: Yoav Artzi

Adapted from Yoav Goldberg's Book and slides by Sasha Rush

# Overview

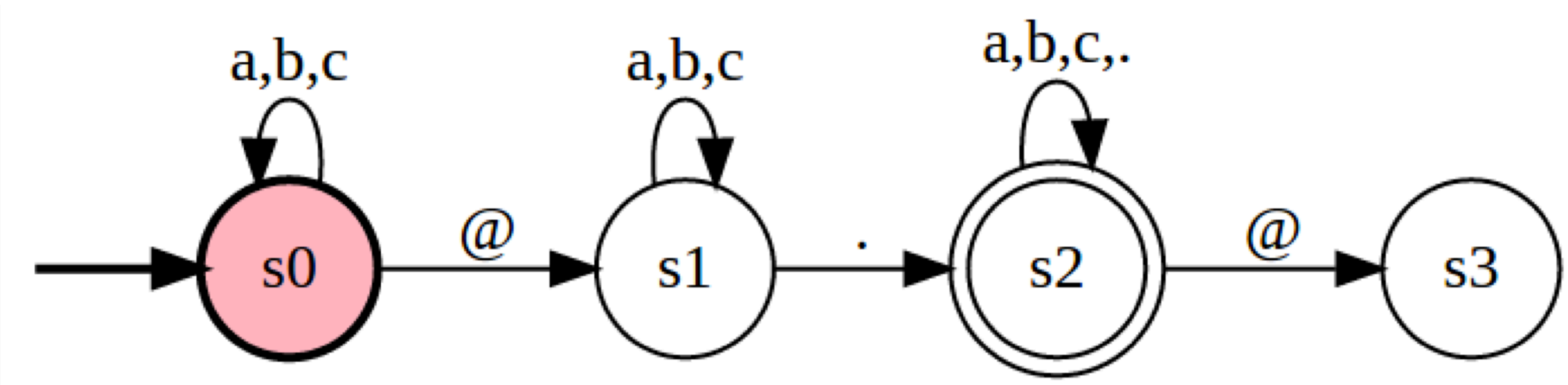
- Finite state models
- Recurrent neural networks (RNNs)
- Training RNNs
- RNN Models
- Long short-term memory (LSTM)
- Attention

# Text Classification

- Consider the example:
  - Goal: classify sentiment
    *How can you not see this movie?*
    *You should not see this movie.*
- Model: bag of words
- How well will the classifier work?
  - Similar unigrams and bigrams
- Generally: need to maintain a **state** to capture distant influences

# Finite State Machines

- Simple, classical way of representing state
- Current state: saves necessary past information
- Example: email address parsing



# Deterministic Finite State Machines

- $S$ – states
- $\Sigma$ – vocabulary
- $s_0 \in S$ – start state
- $R: S \times \Sigma \rightarrow S$ – transition function

- What does it do?
  - Maps input $w_1, \ldots, w_n$ to states $s_1, \ldots, s_n$
  - For all $i \in \{1, \ldots, n\}$
$$s_i = R(s_{i-1}, w_i)$$
- Can we use it for POS tagging? Language modeling?

# Types of State Machines

- Acceptor
  - Compute final state $s_n$ and make a decision based on it: $y = O(s_n)$
- Transducers
  - Apply function $y_i = O(s_i)$ to produce output for each intermediate state
- Encoders
  - Compute final state $s_n$, and use it in another model

# Recurrent Neural Networks

- Motivation:
  - Neural network model, but with state
  - How can we borrow ideas from FSMs?
- RNNs are FSMs …
  - … with a twist
  - No longer finite in the same sense

# RNN

- $S = \mathbb{R}^{d_{hid}}$ - hidden state space
- $\Sigma = \mathbb{R}^{d_{in}}$ - input state space
- $\boldsymbol{s}_0 \in S$ - initial state vector
- $R : \mathbb{R}^{d_{in}} \times \mathbb{R}^{d_{hid}} \rightarrow \mathbb{R}^{d_{hid}}$ - transition function
- Simple definition of $R$:

$$R_{Elman}(\boldsymbol{s}, \boldsymbol{x}) = \tanh([\boldsymbol{x}, \boldsymbol{s}]\boldsymbol{W} + \boldsymbol{b})$$

* Notation: vectors and matrices are bold

Elman (1990)

# RNN

- Map from dense sequence to dense representation
  - $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n \rightarrow \boldsymbol{s}_1, \dots, \boldsymbol{s}_n$
  - For all $i \in \{1, \dots, n\}$

$$\boldsymbol{s}_i = R(\boldsymbol{s}_{i-1}, \boldsymbol{x}_i)$$

  - $R$ is parameterized, and parameters are shared between all steps
  - Example:

$$\boldsymbol{s}_4 = R(\boldsymbol{s}_3, \boldsymbol{x}_4) = \cdots = R(R(R(R(\boldsymbol{s}_0, \boldsymbol{x}_1), \boldsymbol{x}_2), \boldsymbol{x}_3), \boldsymbol{x}_4)$$

# RNNs

- Hidden states $\boldsymbol{s}_i$ can be used in different ways
- Similar to finite state machines
  - Acceptor
  - Transducer
  - Encoder
- Output function maps vectors to symbols:

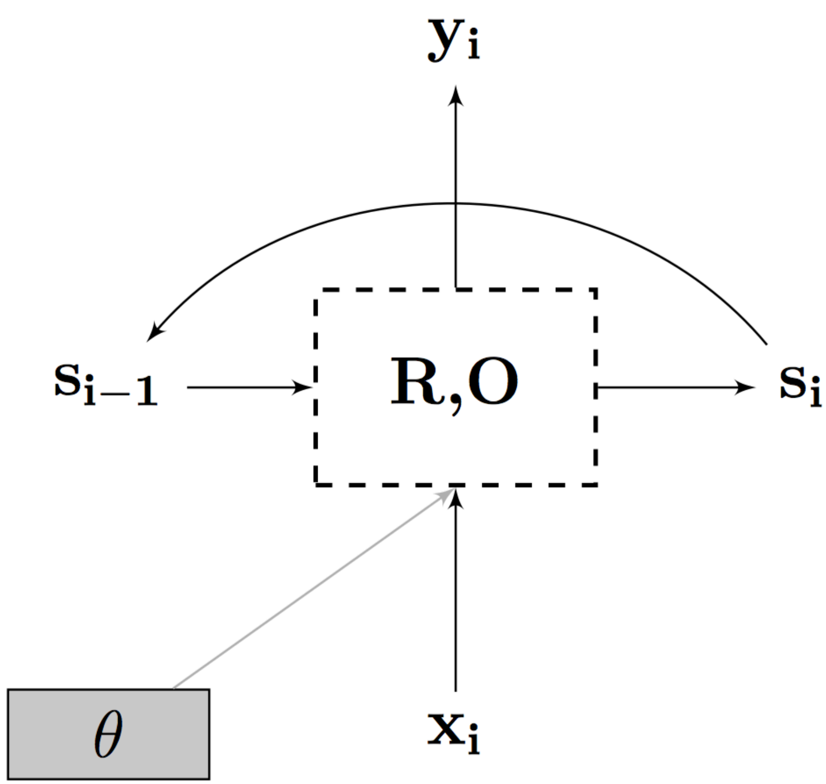
$$\boldsymbol{O}: \mathbb{R}^{d_{hid}} \rightarrow \mathbb{R}^{d_{out}}$$

- For example: single layer + softmax

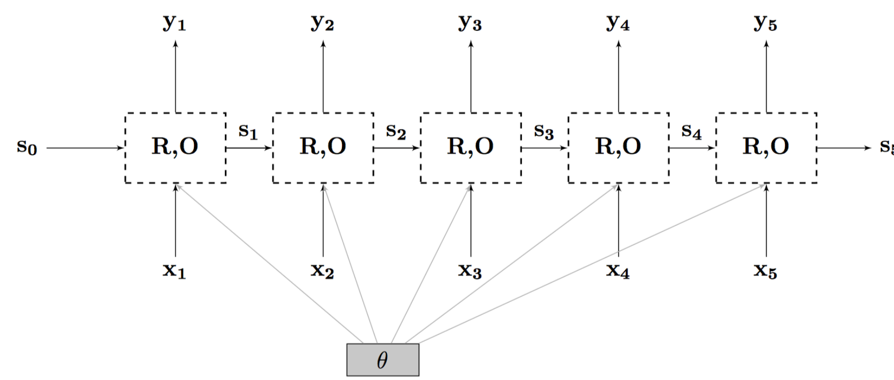
$$O(\boldsymbol{s}_i) = \text{softmax}(\boldsymbol{s}_i \boldsymbol{W} + \boldsymbol{b})$$

# Graphical Representation

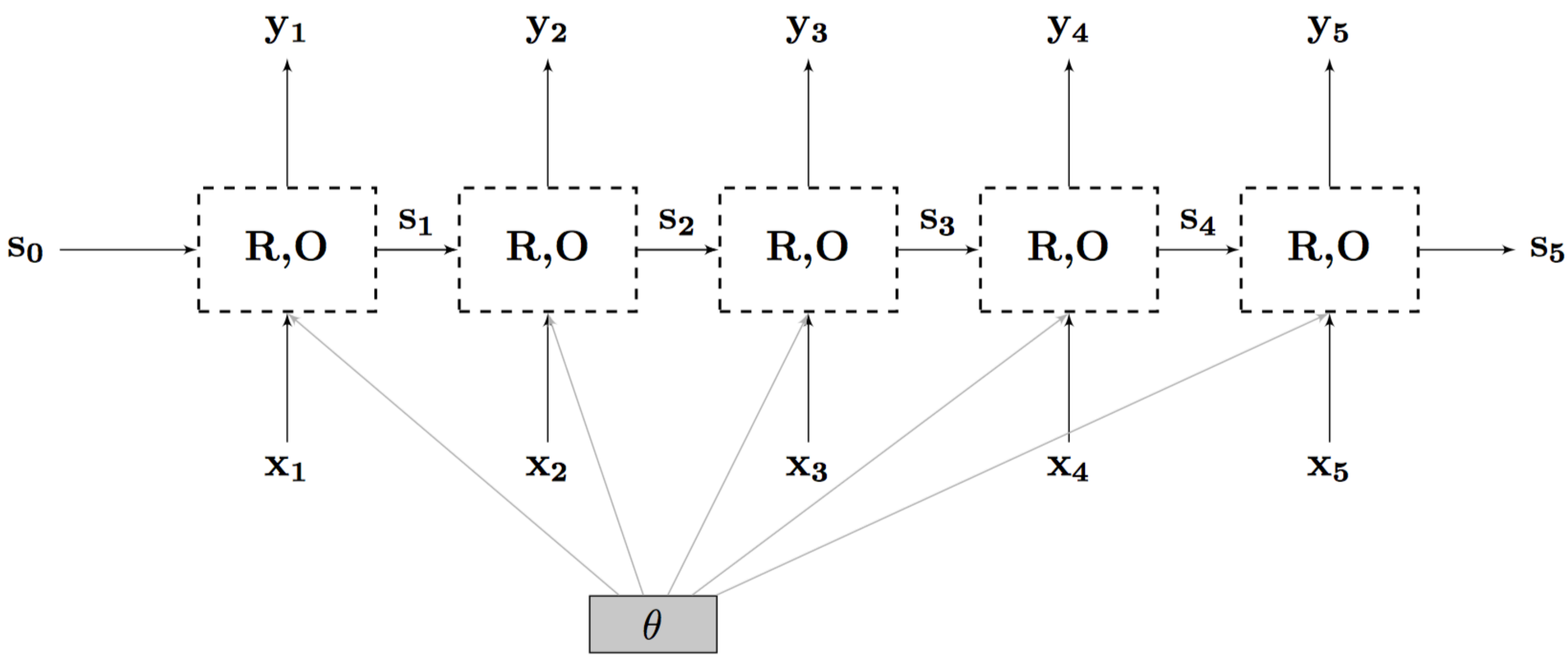
## Recursive Representation




## Unrolled Representation



# Graphical Representation

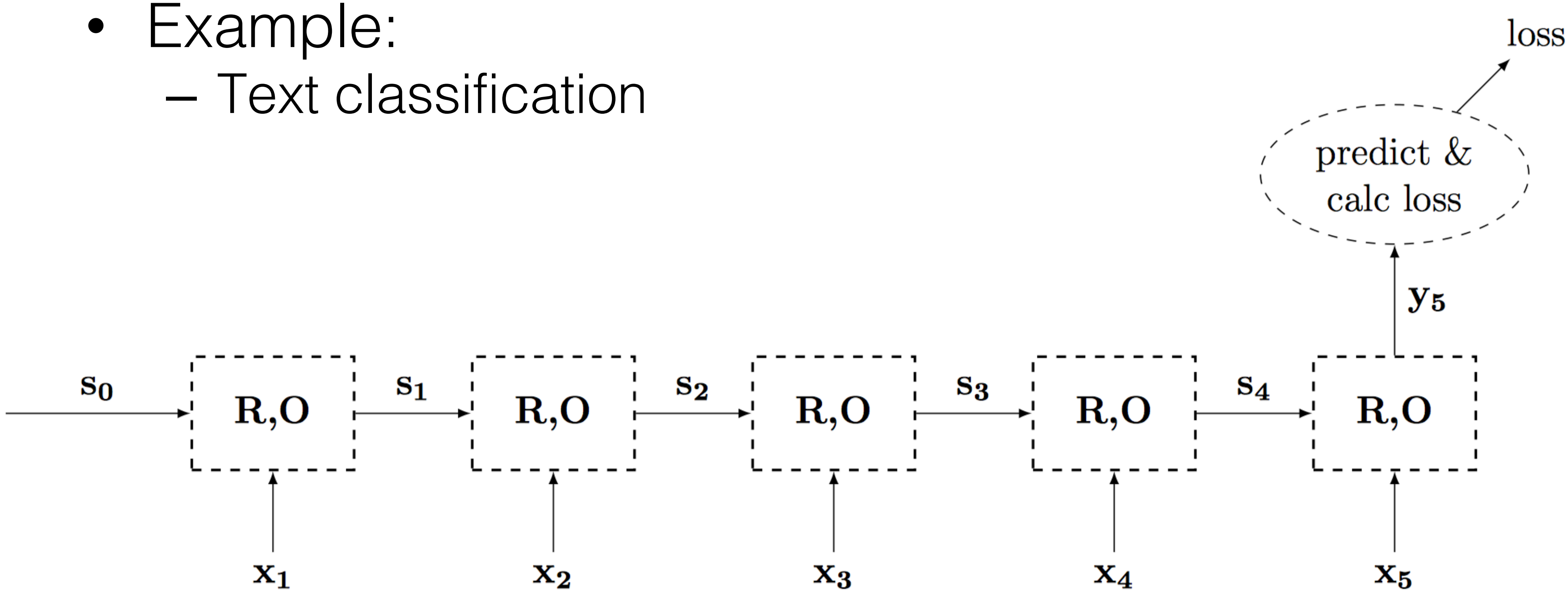


# Training

- RNNs are trained with SGD and Backprop
- Define loss over outputs
  - Depends on supervision and task
- Backpropagation through time (BPTT)
  - Use unrolled representation
  - Run forward propagation
  - Run backward propagation
  - Update all weights
- Weights are shared between time steps
  - Sum the contributions of each time step to the gradient
- Inefficient
  - Batch helps, common but tricky to implement with variable-size models (good helper methods in PyTorch, non-issue with auto batching in DyNet)

# RNN: Acceptor Architecture

- Only care about the output from the last hidden state
- Train: supervised, loss on prediction
- Example:
  - Text classification



# Language Modeling

- Input: $X = x_1, \ldots, x_n$
- Goal: compute $p(X)$
- Bi-gram decomposition:

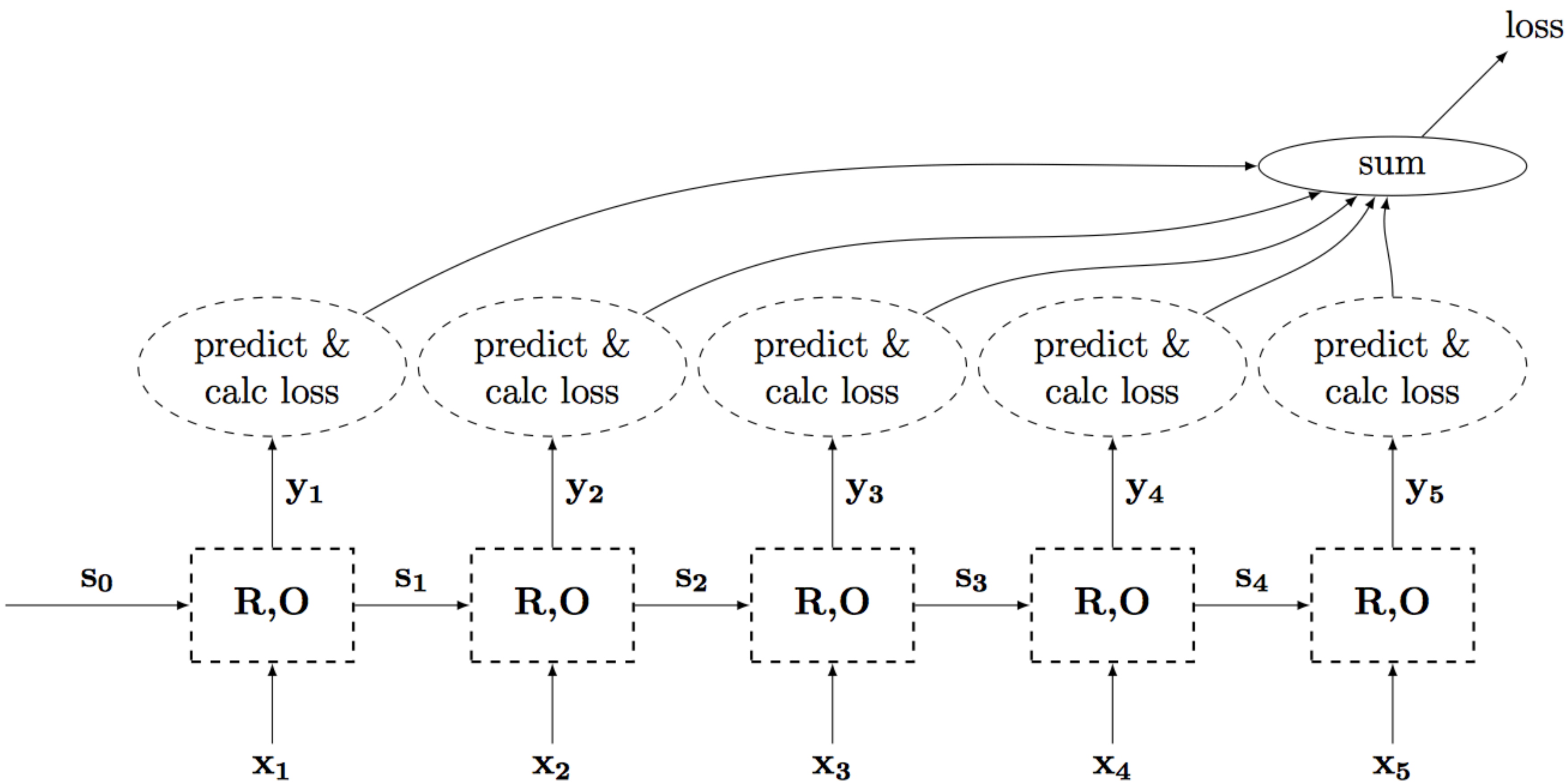
$$p(X) = \prod_{i=1}^{n} p(x_i \mid x_{i-1})$$

- With RNNs, can do non-Markovian models:

$$p(X) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1})$$

# RNN: Transducer Architecture

- Predict output for every time step



# Language Modeling

- Input: $X = x_1, \ldots, x_n$
- Goal: compute $p(X)$
- Model:

$$p(X) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1})$$

$$p(\, x_i \mid x_1, \ldots, x_{i-1}\,) = O(\boldsymbol{s}_i) = O(R(\boldsymbol{s}_i, \boldsymbol{x}_{i-1}))$$

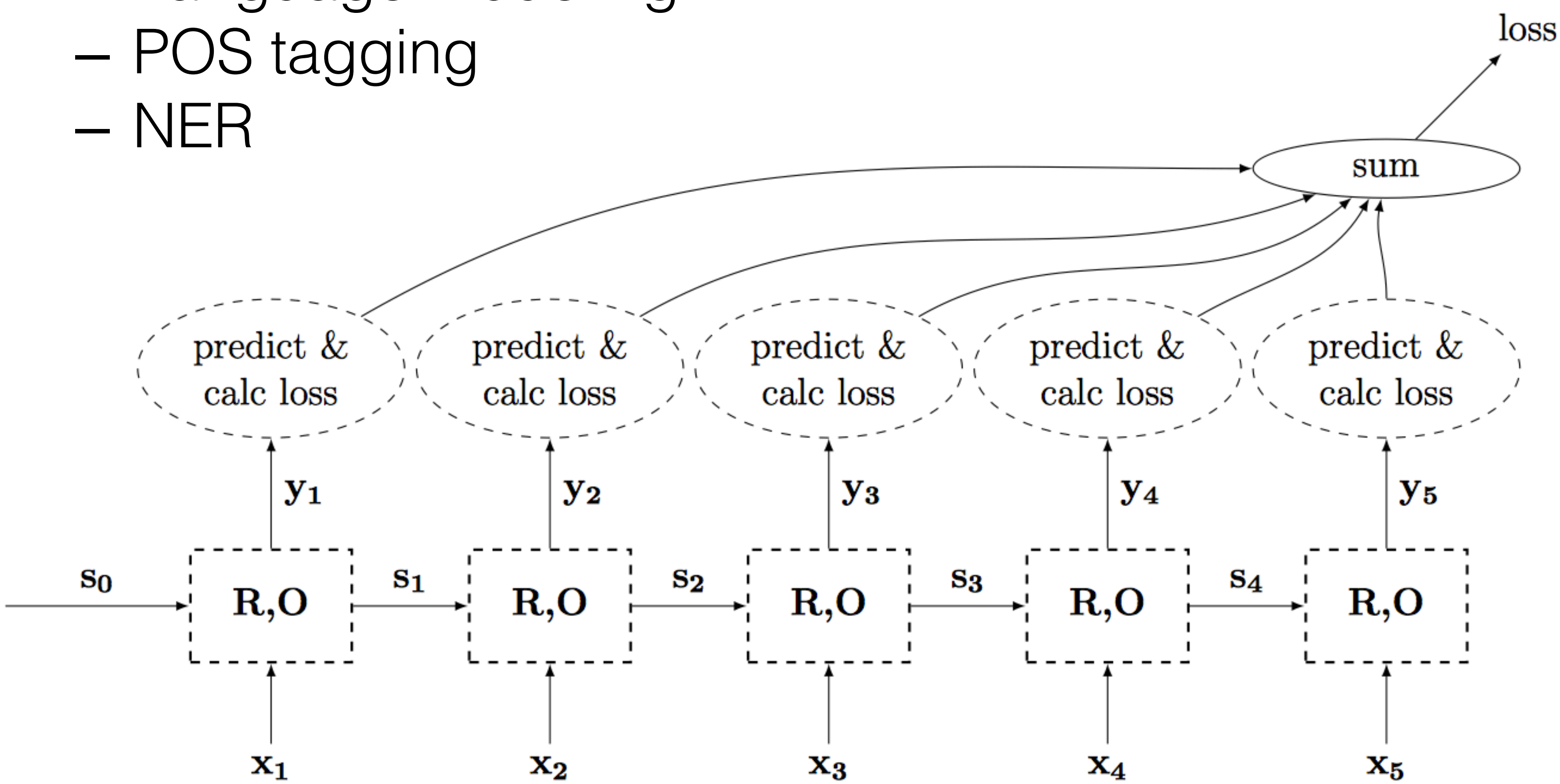
$$O(\boldsymbol{s}_i) = \text{softmax}(s_i \boldsymbol{W} + \boldsymbol{b})$$

- Predict next token $\hat{y}_i$ as we go:

$$\hat{y}_i = \text{argmax} O(\boldsymbol{s}_i)$$

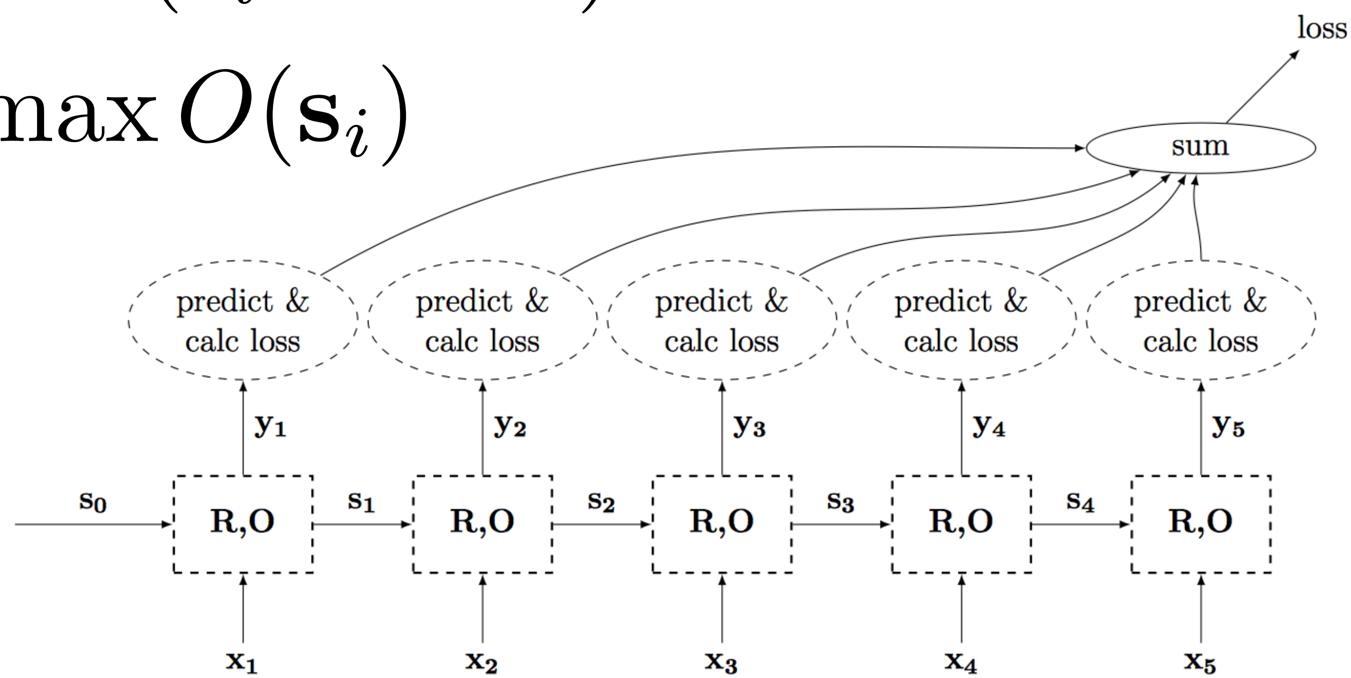
# RNN: Transducer Architecture

- Predict output for every time step
- Examples:
  - Language modeling
  - POS tagging
  - NER



# RNN: Transducer Architecture

$$
\begin{aligned}
X =& \mathbf{x}_1, \ldots, \mathbf{x}_n \\
\mathbf{s}_i =& R(\mathbf{s}_{i-1}, \mathbf{x}_i), i = 1, \ldots, n \\
O(\mathbf{s}_i) =& \text{softmax}(\mathbf{s}_i \mathbf{W} + \mathbf{b}) \\
\hat{y}_i =& \arg\max O(\mathbf{s}_i)
\end{aligned}
$$

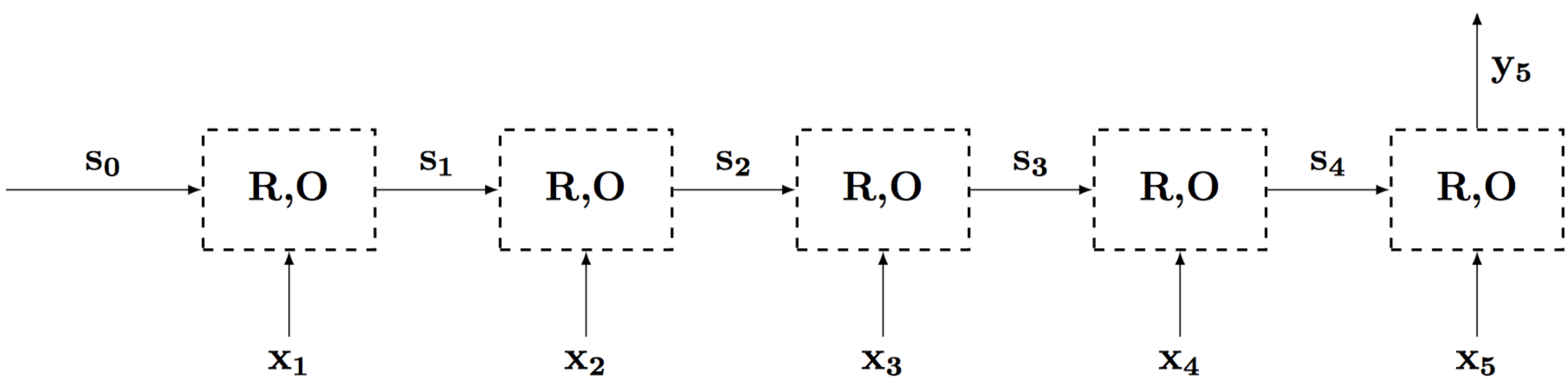


# RNN: Encoder Architecture

- Similar to acceptor
- Difference: last state is used as input to another model and not for prediction

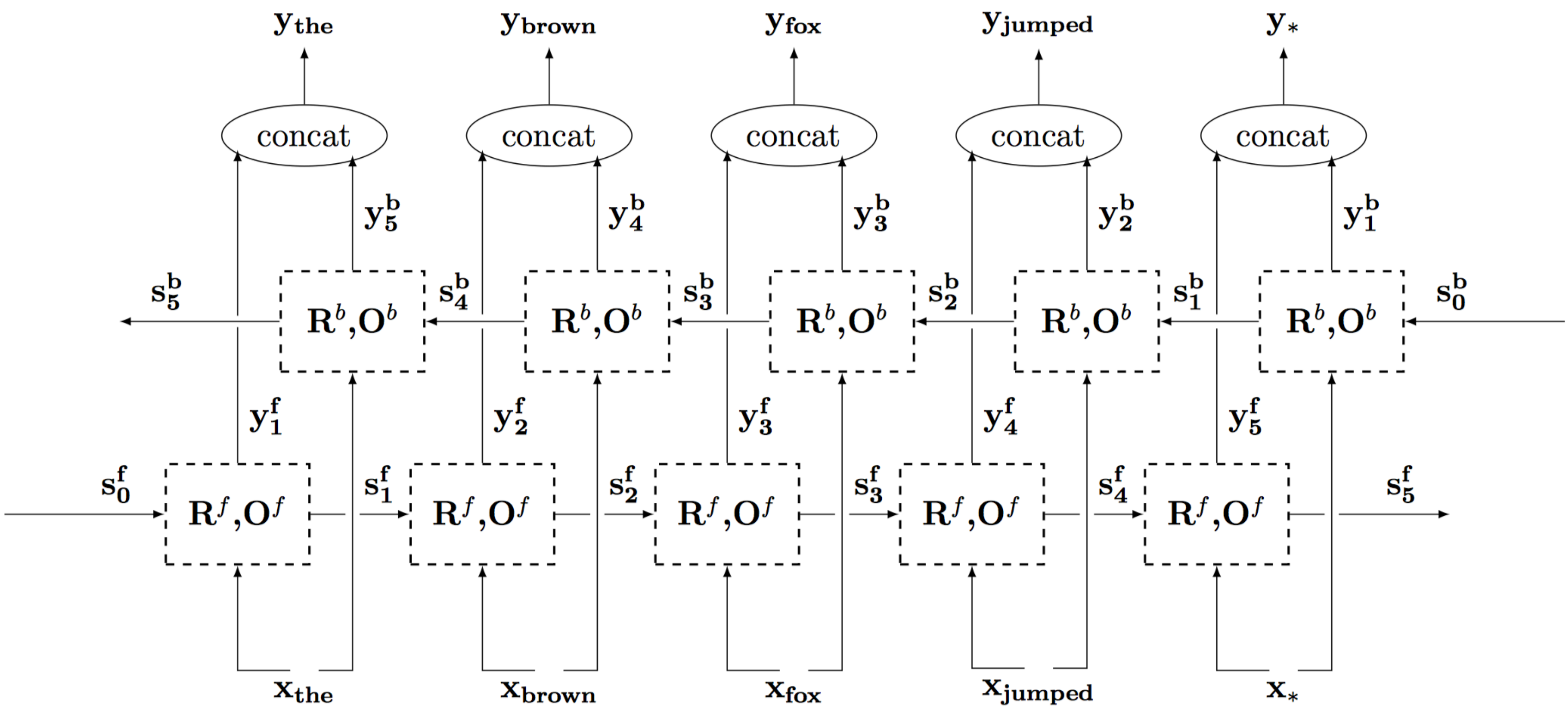
$$O(s_i) = s_i \rightarrow y_n = s_n$$

- Example:
  - Sentence embedding



# Bidirectional RNNs

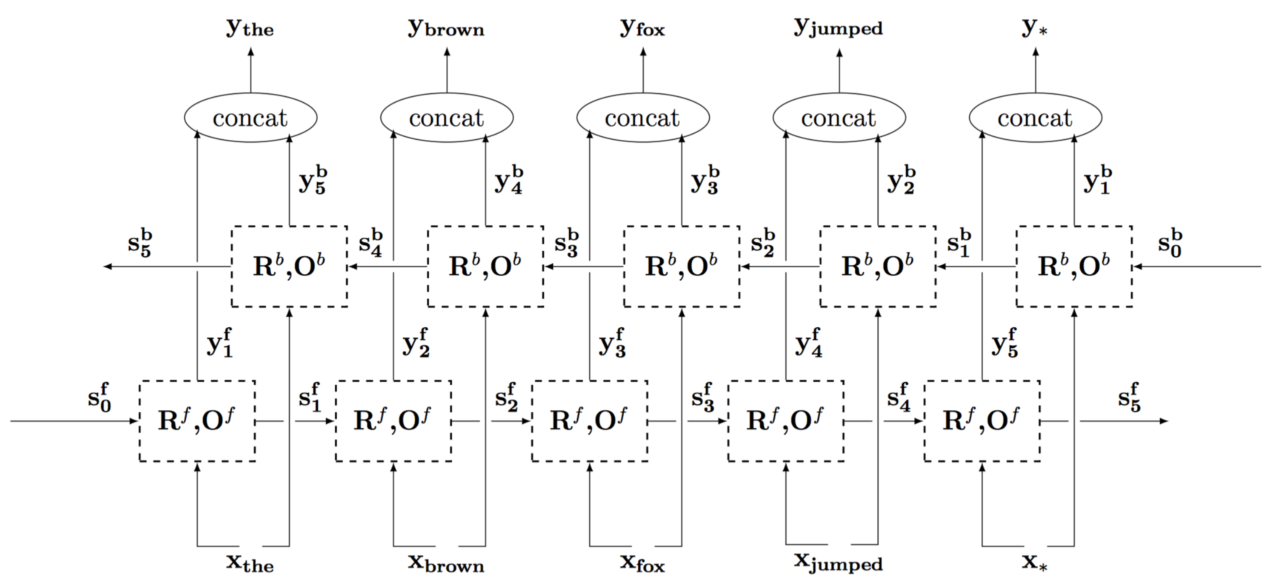
- RNN decisions are based on historical data only
  - How can we account for future input?
- When is it relevant? Feasible?



# Bidirectional RNNs

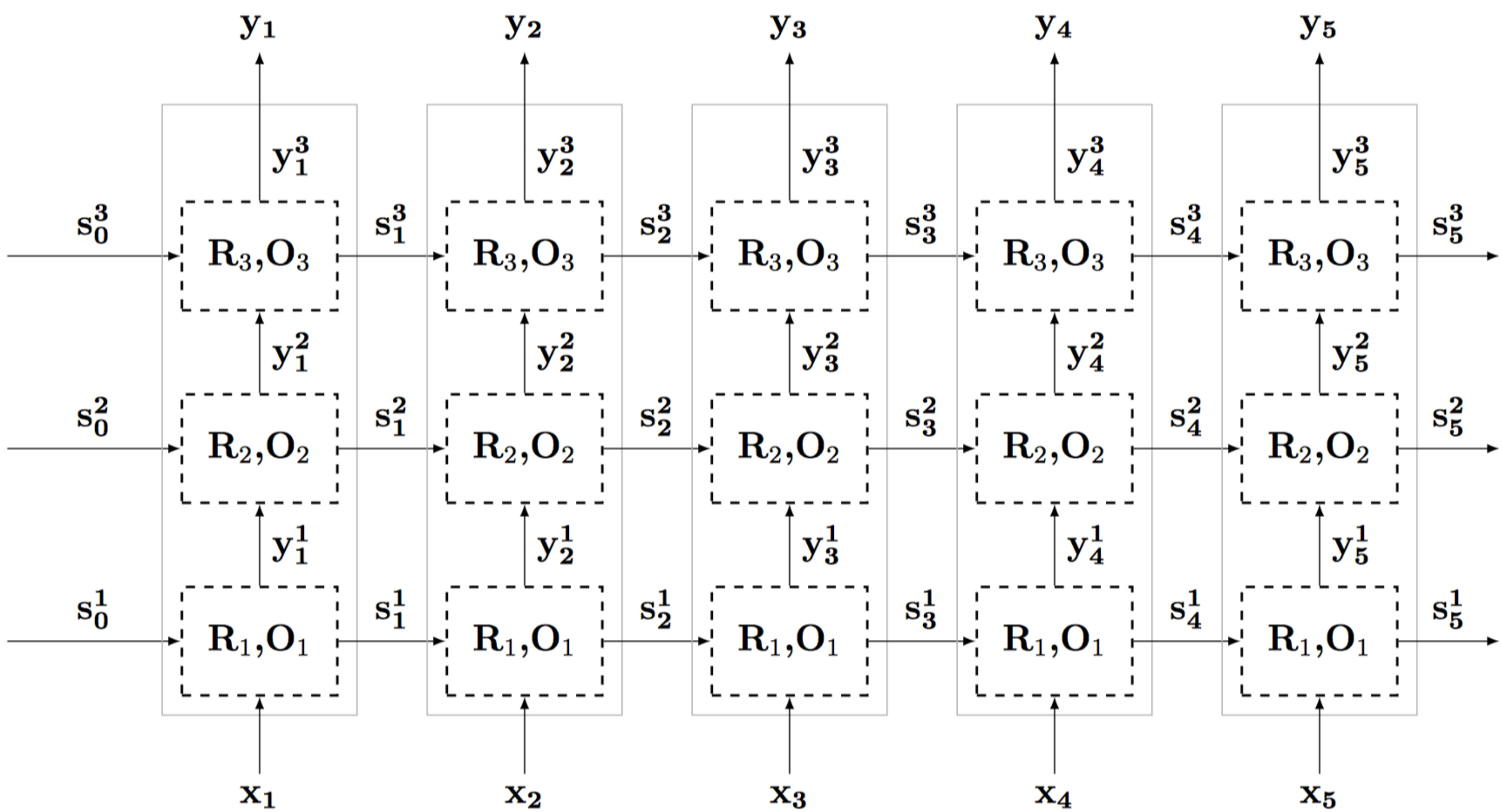
- RNN decisions are based on historical data only
  - How can we account for future input?
- When is it relevant? Feasible?
  - When all the input is available. Not for real-time input.
- Probabilistic model, for example for language modeling:

$$p(X) = \prod_{i=1}^{n} p(x_i \mid x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$$



# Deep RNNs

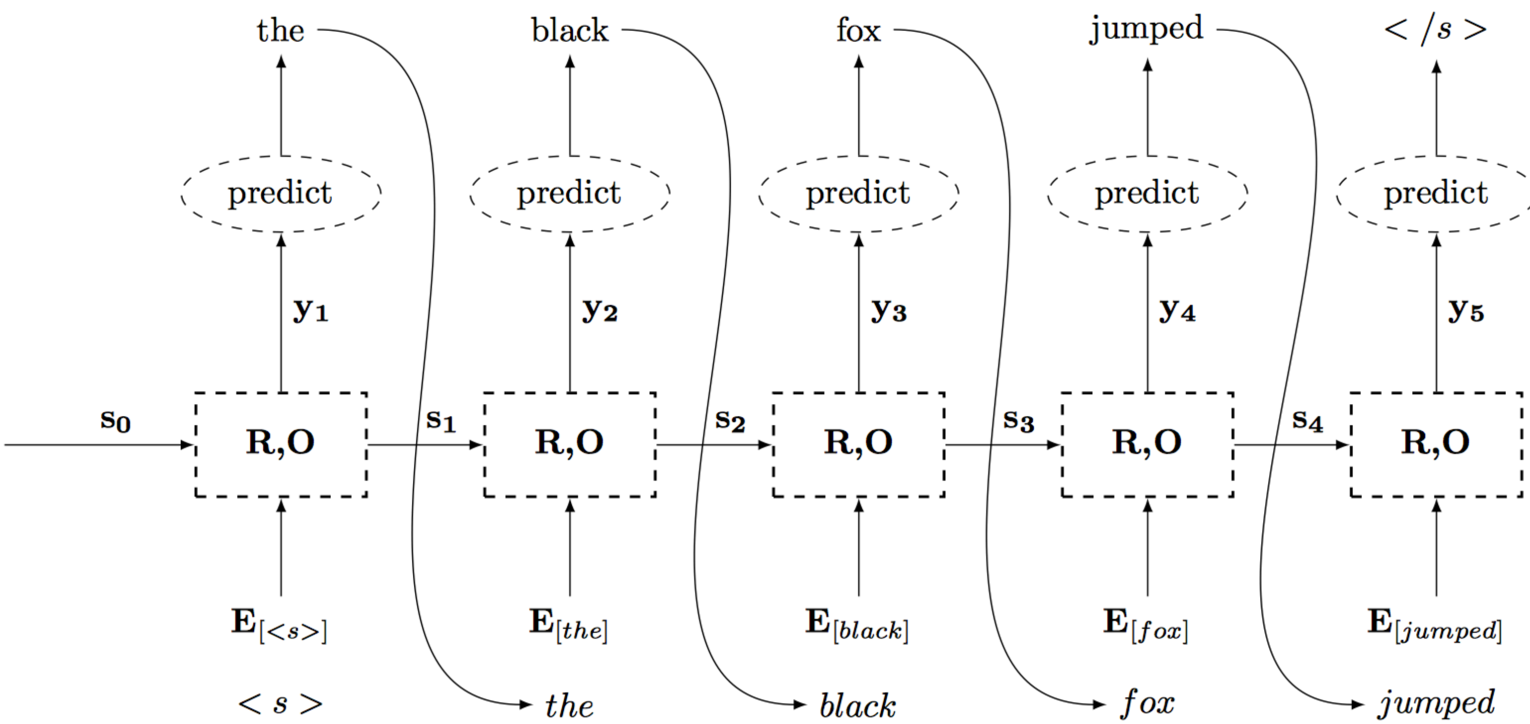
- Can also make RNNs deeper (vertically) to increase model capacity



# RNN: Generator

- Special case of the transducer architecture
- Generation conditioned on $\boldsymbol{s}_0$
- Probabilistic model:

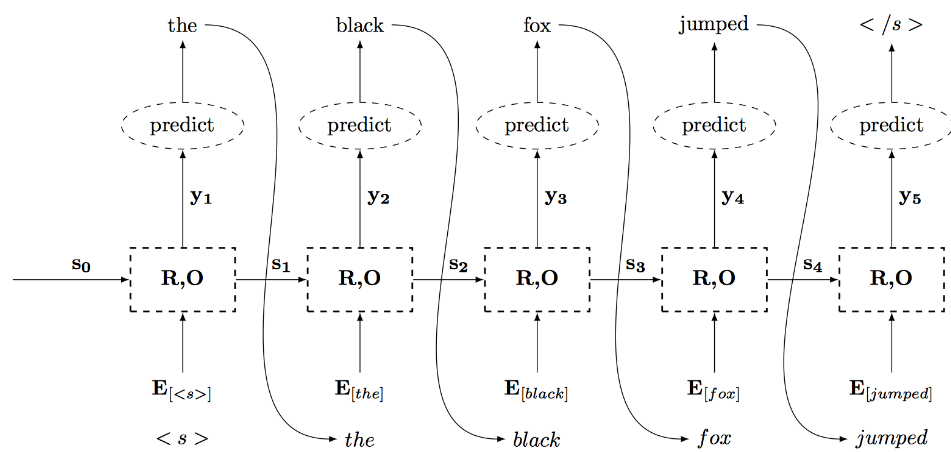
$$p(X \mid s_0) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1}, s_0)$$



# RNN: Generator

- Stop when generating the STOP token
- During learning (usually): force predicting the annotated token and compute loss

$$\mathbf{s}_j = R(\mathbf{s}_{j-1}, E(\hat{\mathbf{t}}_{j-1}))$$
$$O(\mathbf{s}_j) = \text{softmax}(\mathbf{s}_j \mathbf{W} + \mathbf{b})$$
$$\hat{\mathbf{t}}_j = \arg\max O(\mathbf{s}_j)$$



# Example: Caption Generation

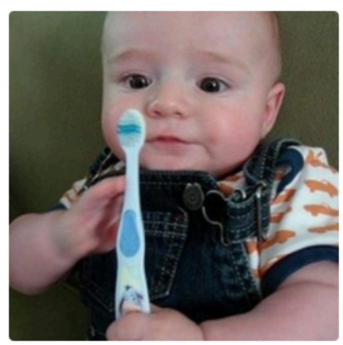
- Given: image $I$
- Goal: generate caption
- Set $\boldsymbol{s}_0 = \mathrm{CNN}(I)$
- Model:

$$p(X \mid I) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1}, I)$$

"little girl is eating piece of cake."

"baseball player is throwing ball in game."

"woman is holding bunch of bananas."



"a young boy is holding a baseball bat."

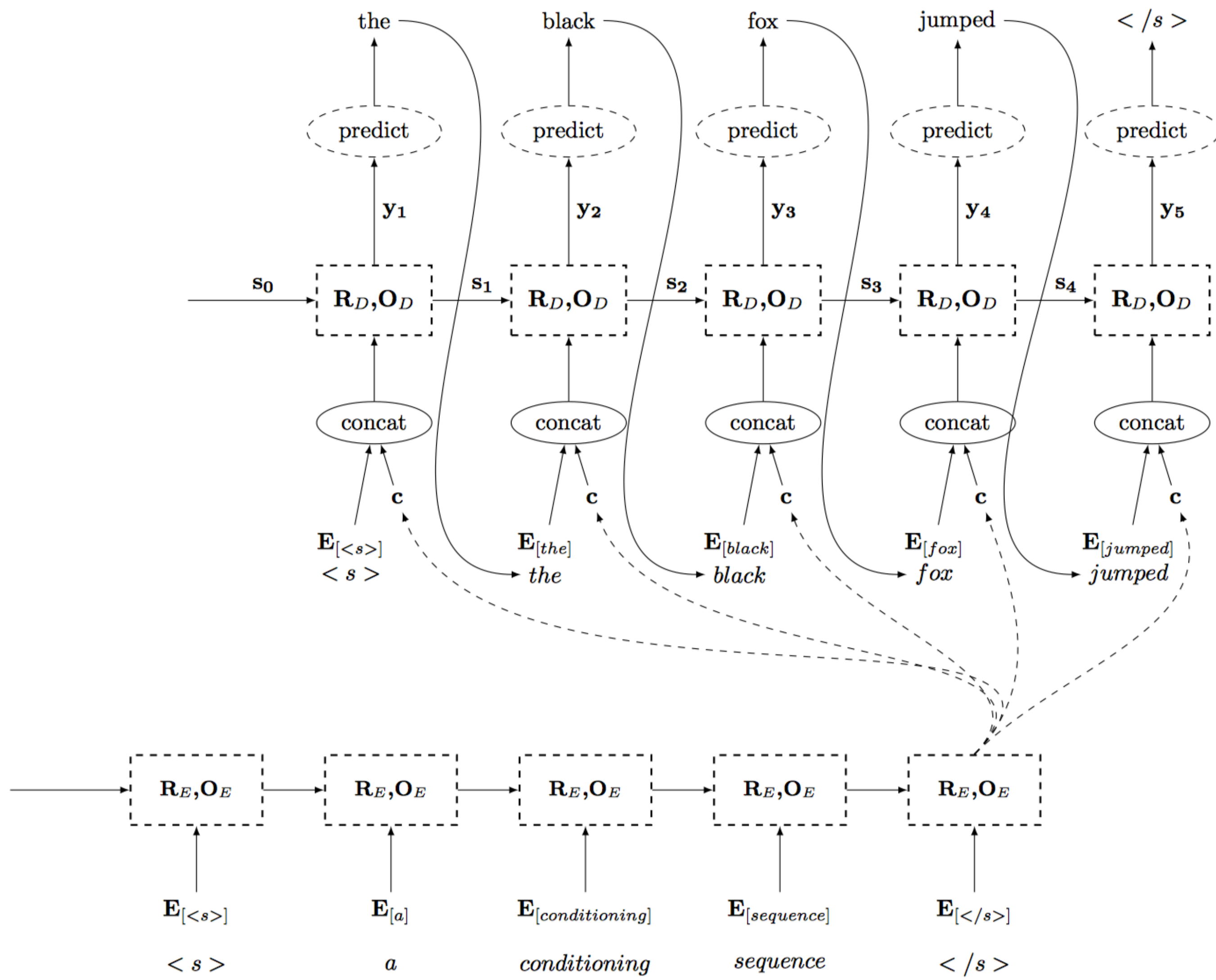
"a cat is sitting on a couch with a remote control."

"a woman holding a teddy bear in front of a mirror."

Examples from Karpathy and Fei-Fei 2015

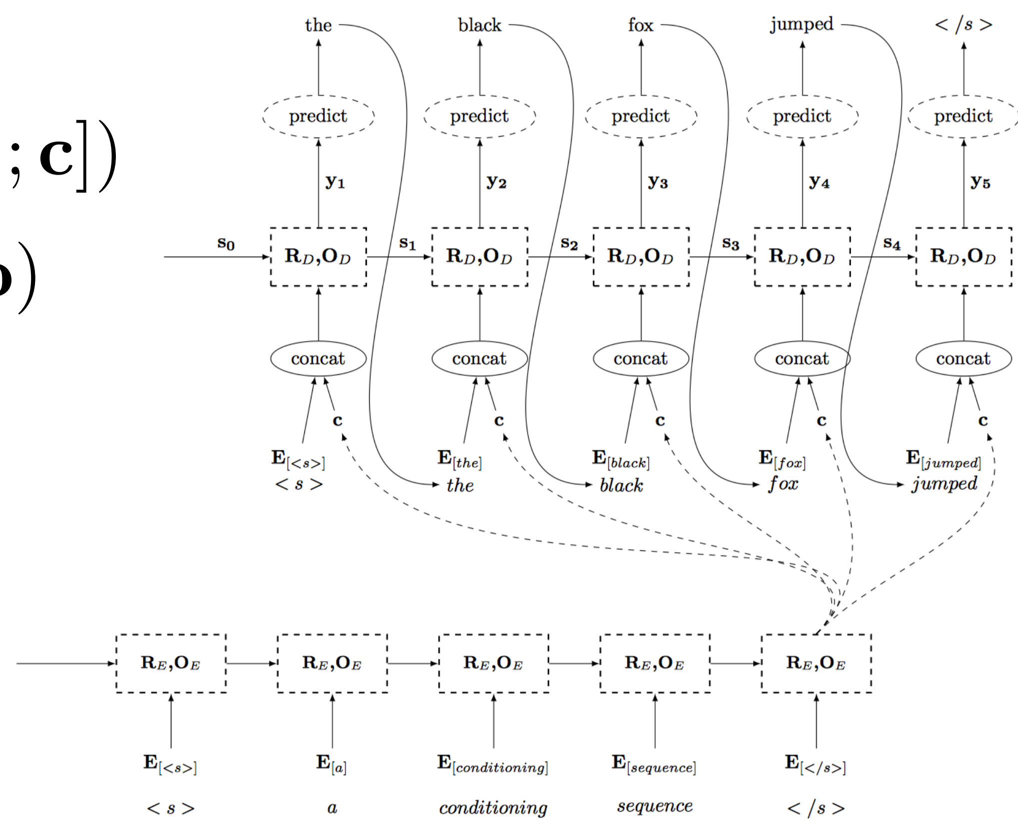
# Sequence-to-Sequence

- Connect encoder and generator
- Many alternatives:
  - Set generator $\boldsymbol{s}_0^d$ to encoder output $\boldsymbol{s}_n^e$
  - Concatenate $\boldsymbol{s}_n^e$ with each step input during generation
- Examples:
  - Machine translation
  - Chatbots
  - Dialog systems
- Can also generate other sequences – not only natural language!

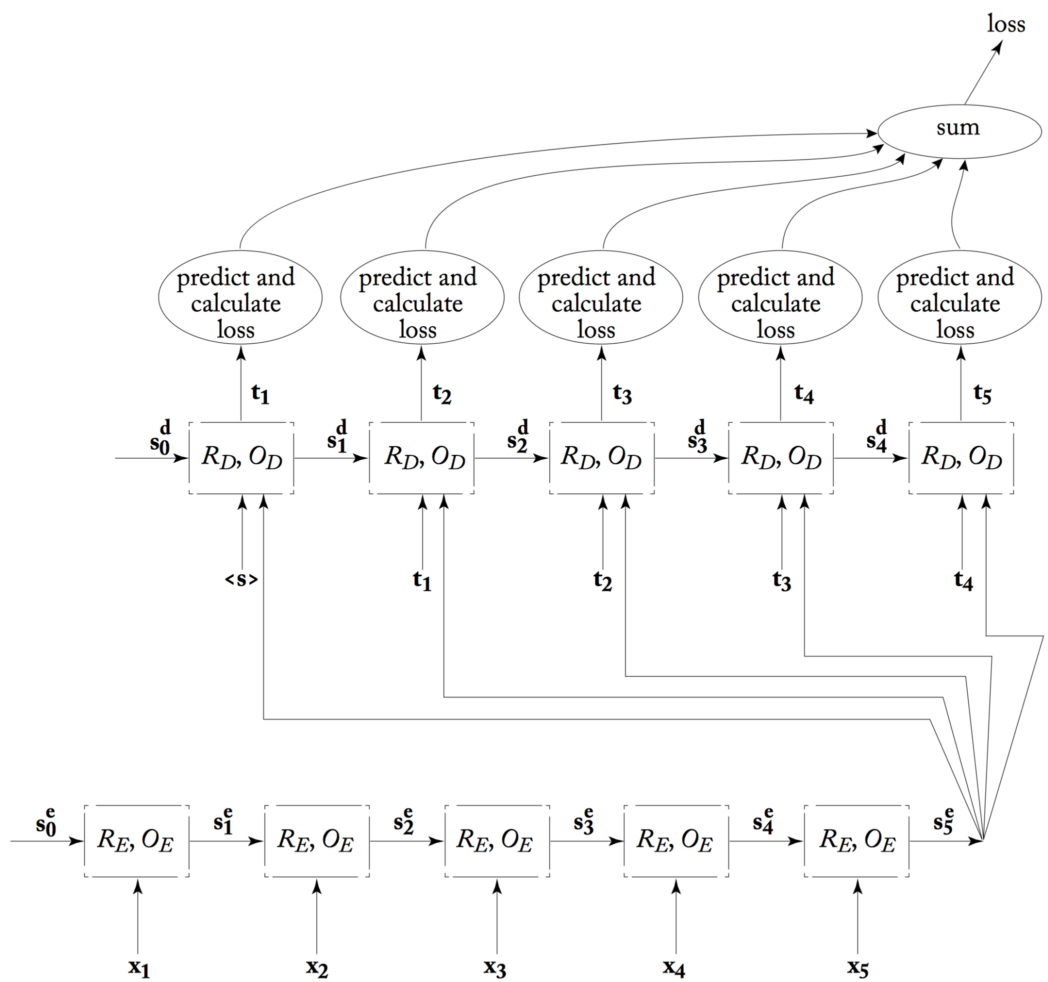


# Sequence-to-Sequence

$$
\begin{aligned}
X &= \mathbf{x}_1, \ldots, \mathbf{x}_n \\
\mathbf{s}_i^E &= R_E(\mathbf{s}_{i-1}^E, \mathbf{x}_i), i = 1, \ldots, n \\
\mathbf{c} &= O_E(\mathbf{s}_n^E) \\
\mathbf{s}_j^D &= R_D(\mathbf{s}_{j-1}^D, [E(\hat{\mathbf{t}}_{j-1}); \mathbf{c}]) \\
O_D(\mathbf{s}_j^D) &= \text{softmax}(\mathbf{s}_j^D \mathbf{W} + \mathbf{b}) \\
\hat{\mathbf{t}}_j &= \arg\max O_D(\mathbf{s}_j^D)
\end{aligned}
$$



# Sequence-to-Sequence Training Graph



# Long-range Interactions

- Promise: Learn long-range interactions of language from data
- Example:

  How can you not see this movie?

  You should not see this movie.
- Sometimes: requires ”remembering” early state
  - Key signal here is at $s_1$, but gradient is at $s_n$

# Long-term Gradients

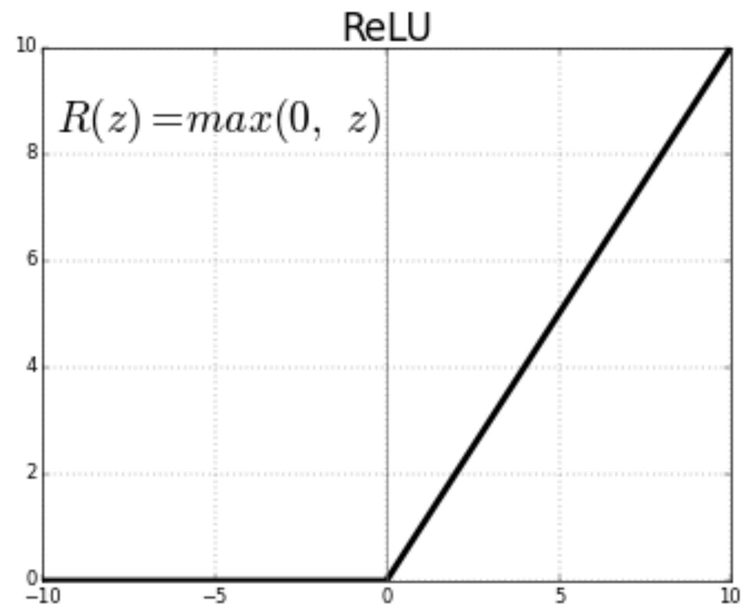
- Gradient go through (many) multiplications
- OK at end layers → close to the loss
- But: issue with early layers
- For example, derivative of $\mathbf{tanh}$

$$\frac{d}{dx}\mathbf{tanh}\, x = 1 - \mathbf{tanh}^2 x$$

  – Large activation → gradient disappears (vanish)
- In other activation functions, values can become larger and larger (explode)

# Exploding Gradients

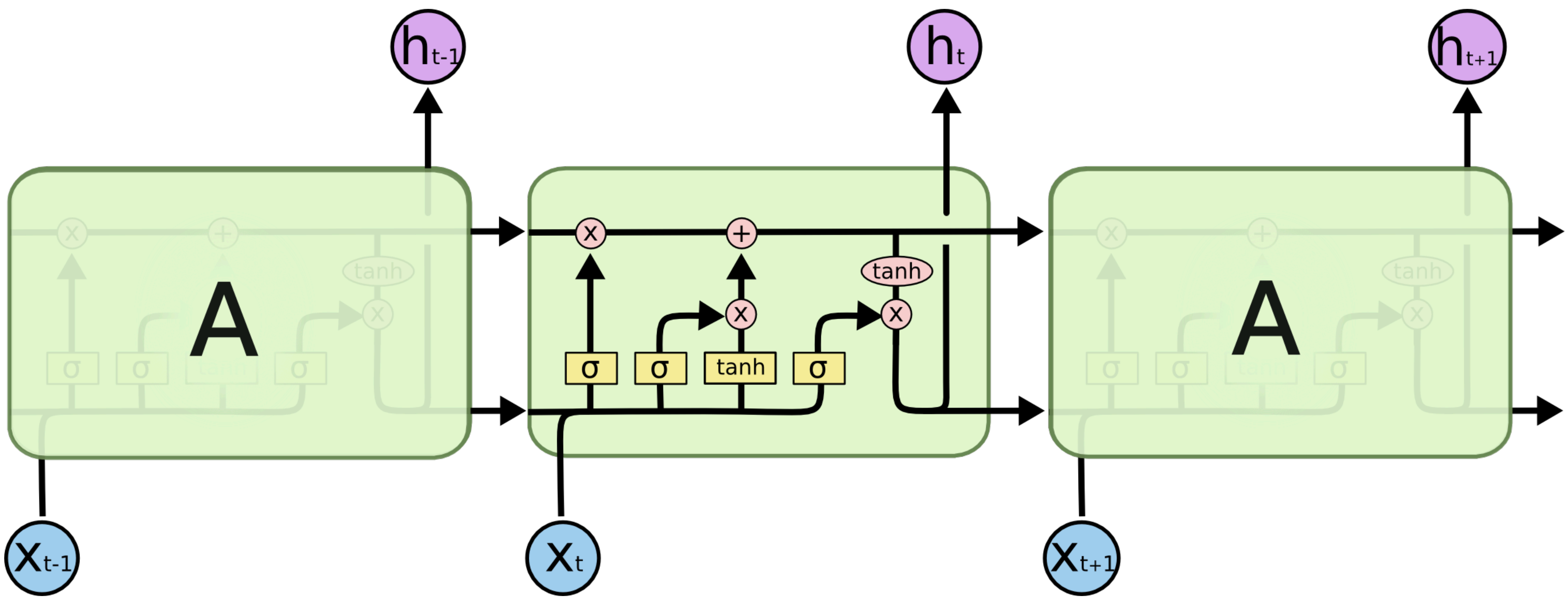
- Common when there is no saturation in activation (e.g., ReLu) and we get exponential blowup
- Result: reasonable short-term gradient, but bad long-term ones
- Common heuristic:
  - Gradient clipping: bounding all gradients by maximum value



# Vanishing Gradients

- Occurs when multiplying small values
  - For example: when $\tanh$ saturates
- Mainly affects long-term gradients
- Solving this is more complex

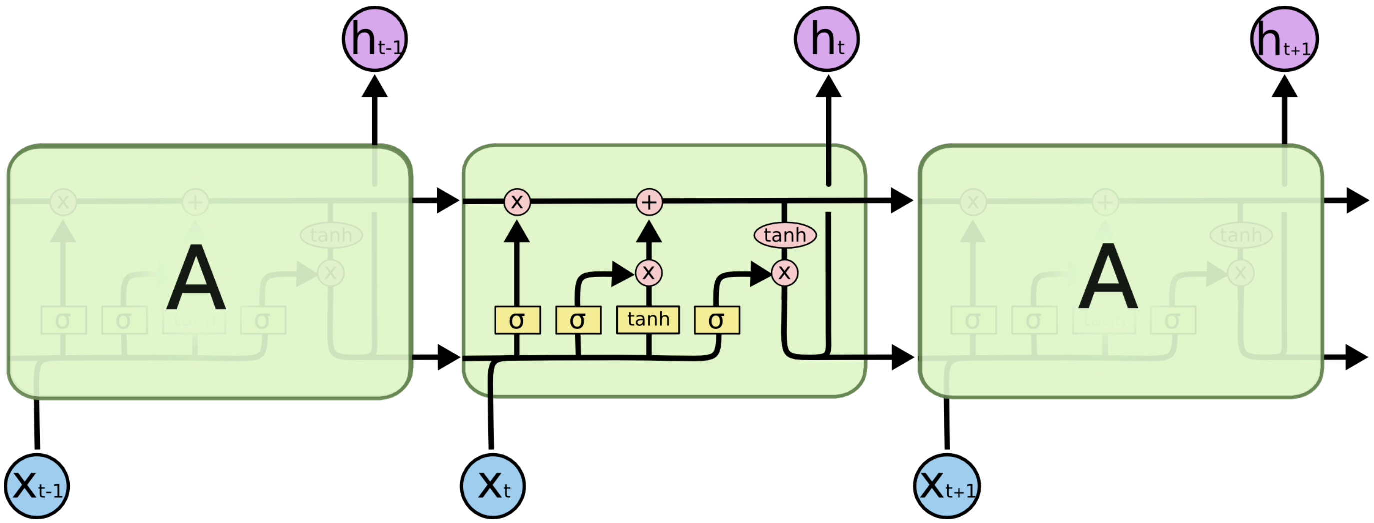
# Long Short-term Memory (LSTM)

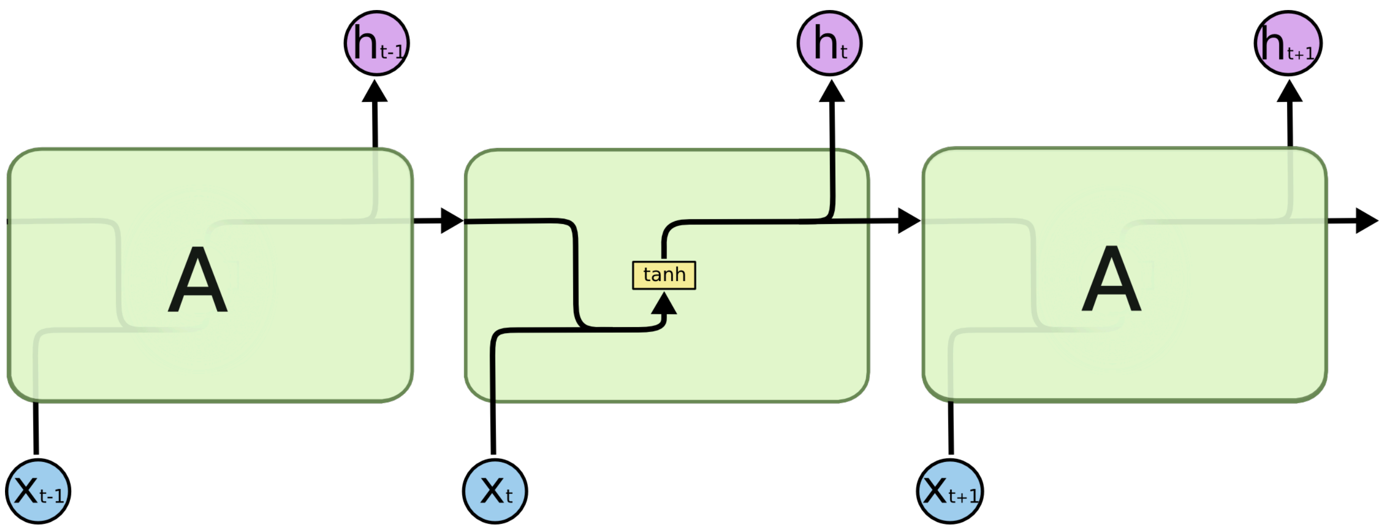



https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Hochreiter and Schmidhuber (1997)

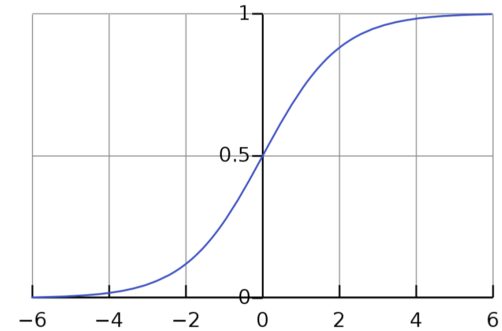
# LSTM vs. Elman RNN



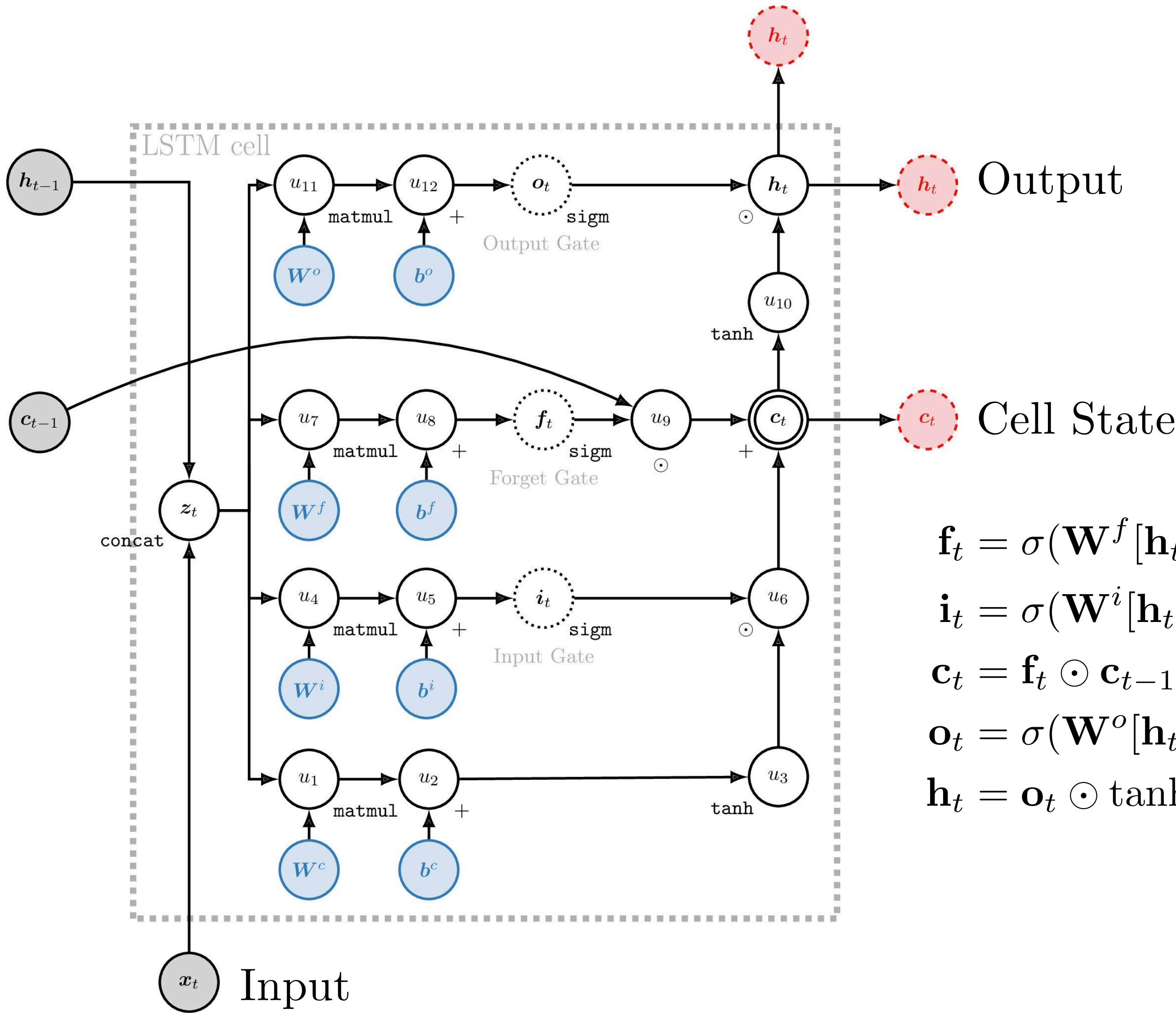




https://colah.github.io/posts/2015-08-Understanding-LSTMs/

# LSTM




$\sigma(\cdot)$




$$\mathbf{f}_t = \sigma(\mathbf{W}^f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}^f)$$

$$\mathbf{i}_t = \sigma(\mathbf{W}^i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}^f)$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(\mathbf{W}^c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}^c)$$

$$\mathbf{o}_t = \sigma(\mathbf{W}^o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}^o)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Image by Tim Rocktäschel

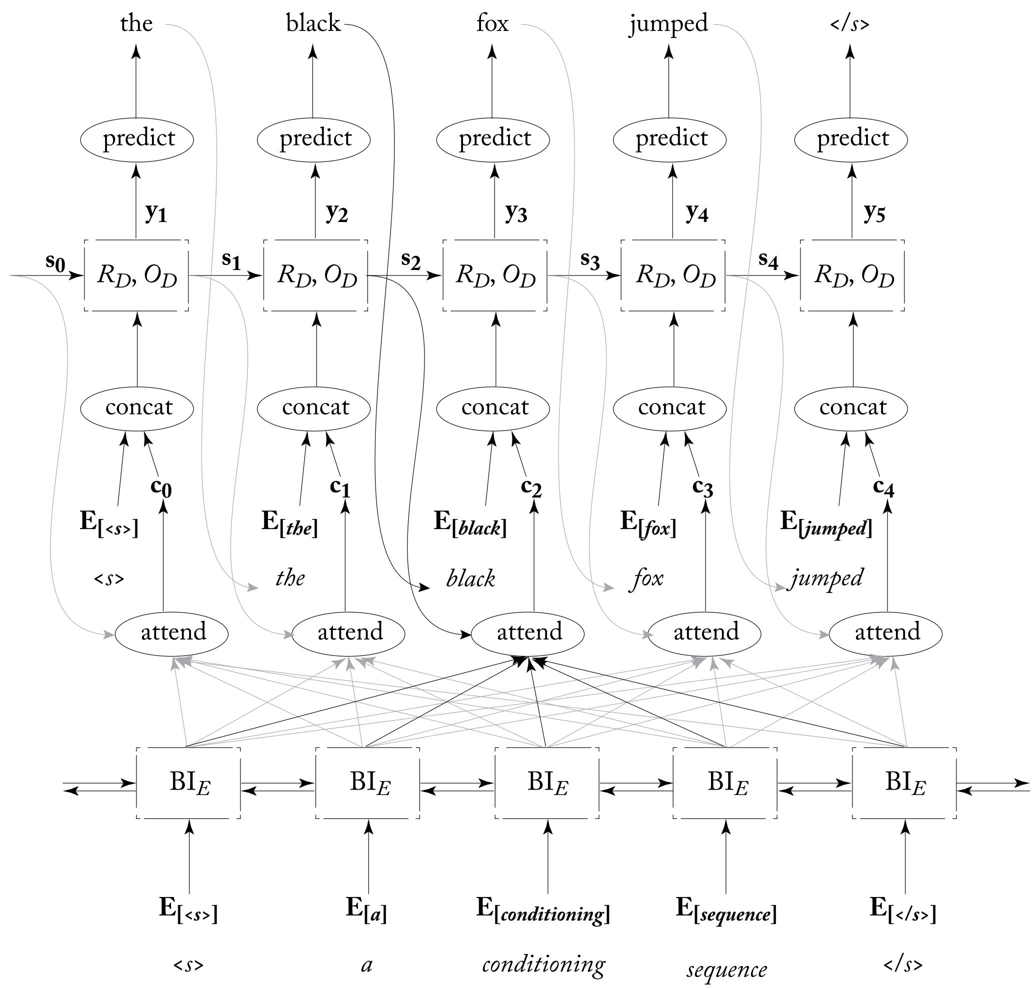
# Attention

- In seq-to-seq models, a single vector connects encoding and decoding
  - Any concern?
  - All the input string information must encoded into a fixed-length vector
  - The decoder must recover all this information from a fixed-length vector
- Attention relaxes the assumption that a single vector must be used to encode the input sentence regardless of length

# Attention

- Encode input sentence as a sequence of vectors
- At each step: pick what vector to use
- But: discrete choice is not differentiable
  - Make the choice soft

# Attention

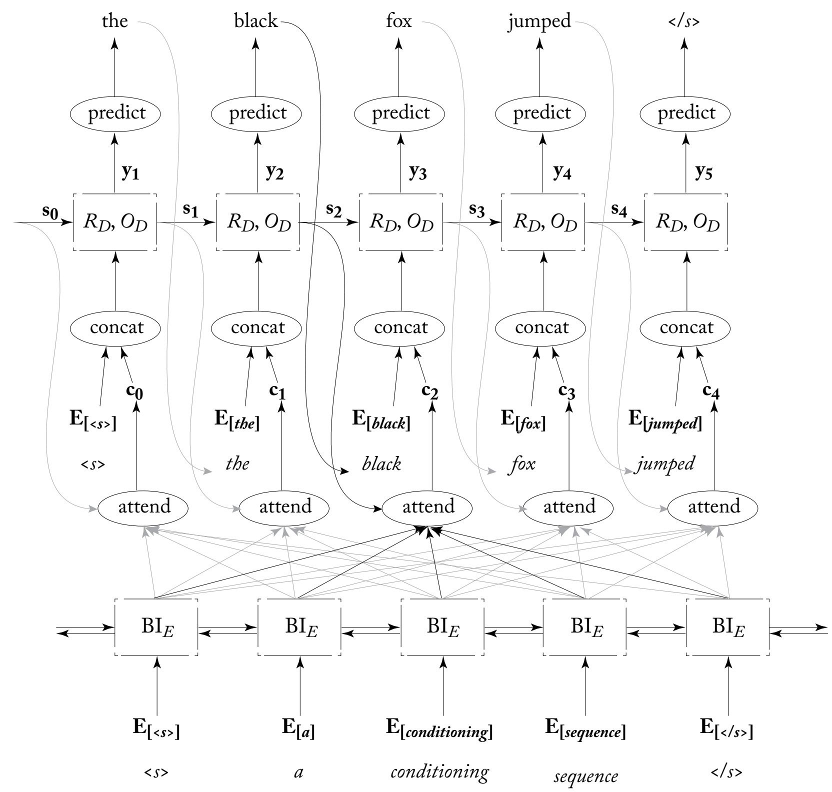

the
black
fox
jumped
</s>
predict
$\mathbf{y_1}$ $\mathbf{y_2}$ $\mathbf{y_3}$ $\mathbf{y_4}$ $\mathbf{y_5}$
$\mathbf{s_0}$ $\mathbf{s_1}$ $\mathbf{s_2}$ $\mathbf{s_3}$ $\mathbf{s_4}$
$R_D, O_D$
concat
$\mathbf{c_0}$ $\mathbf{c_1}$ $\mathbf{c_2}$ $\mathbf{c_3}$ $\mathbf{c_4}$
$\mathbf{E}_{[<s>]}$ $\mathbf{E}_{[the]}$ $\mathbf{E}_{[black]}$ $\mathbf{E}_{[fox]}$ $\mathbf{E}_{[jumped]}$
<s> the black fox jumped
attend
$\mathrm{BI}_E$
$\mathbf{E}_{[<s>]}$ $\mathbf{E}_{[a]}$ $\mathbf{E}_{[conditioning]}$ $\mathbf{E}_{[sequence]}$ $\mathbf{E}_{[</s>]}$
<s> a conditioning sequence </s>

# Attention

$$
\begin{aligned}
X &= \mathbf{x}_1, \ldots, \mathbf{x}_n \\
\mathbf{s}_i^E &= R_E(\mathbf{s}_{i-1}^E, \mathbf{x}_i), i = 1, \ldots, n \\
\bar{\mathbf{c}}_i &= O_E(\mathbf{s}_i^E)
\end{aligned}
$$

$$
\text{Attend}\left\{
\begin{aligned}
\bar{\alpha}_i^j &= \mathbf{s}_{j-1}^D \cdot \bar{\mathbf{c}}_i \\
\alpha^j &= \mathrm{softmax}(\bar{\alpha}_1^j, \ldots, \bar{\alpha}_n^j) \\
\mathbf{c}_j &= \sum_{i=1}^{n} \alpha_i^j \bar{\mathbf{c}}_i
\end{aligned}
\right.
$$

$$
\begin{aligned}
\mathbf{s}_j^D &= R(\mathbf{s}_{j-1}^D, [\mathbf{E}(\hat{\mathbf{t}}_{j-1}); \mathbf{c}_j]) \\
O_D(\mathbf{s}_j^D) &= \mathrm{softmax}(\mathbf{s}_j^D \mathbf{W} + \mathbf{b}) \\
\hat{\mathbf{t}}_j &= \arg\max O_D(\mathbf{s}_j^D)
\end{aligned}
$$



# Attention

- Many variants of attention function
  - Dot product (previous slide)
  - MLP
  - Bi-linear transformation
- Various ways to combine context vector into decoder computation
- See Luong et al. 2015