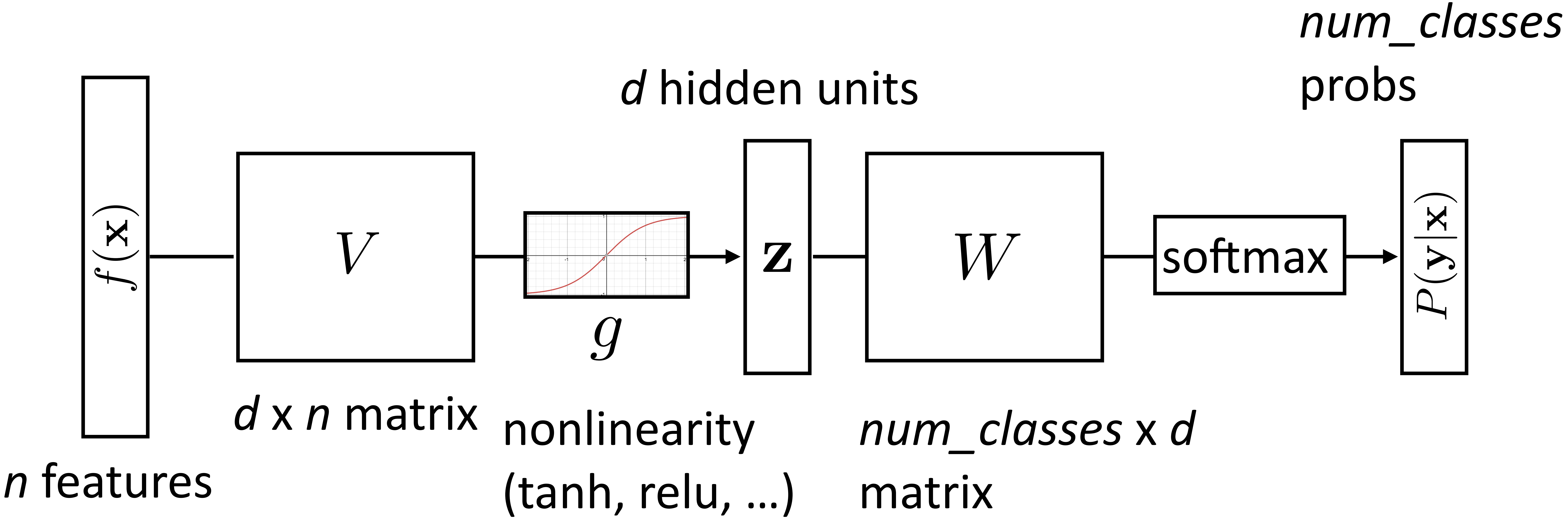
ANLP

12 - Recurent NNs (NNs, part II)

David Schlangen
University of Potsdam, MSc Cognitive Systems
Winter 2019 / 2020

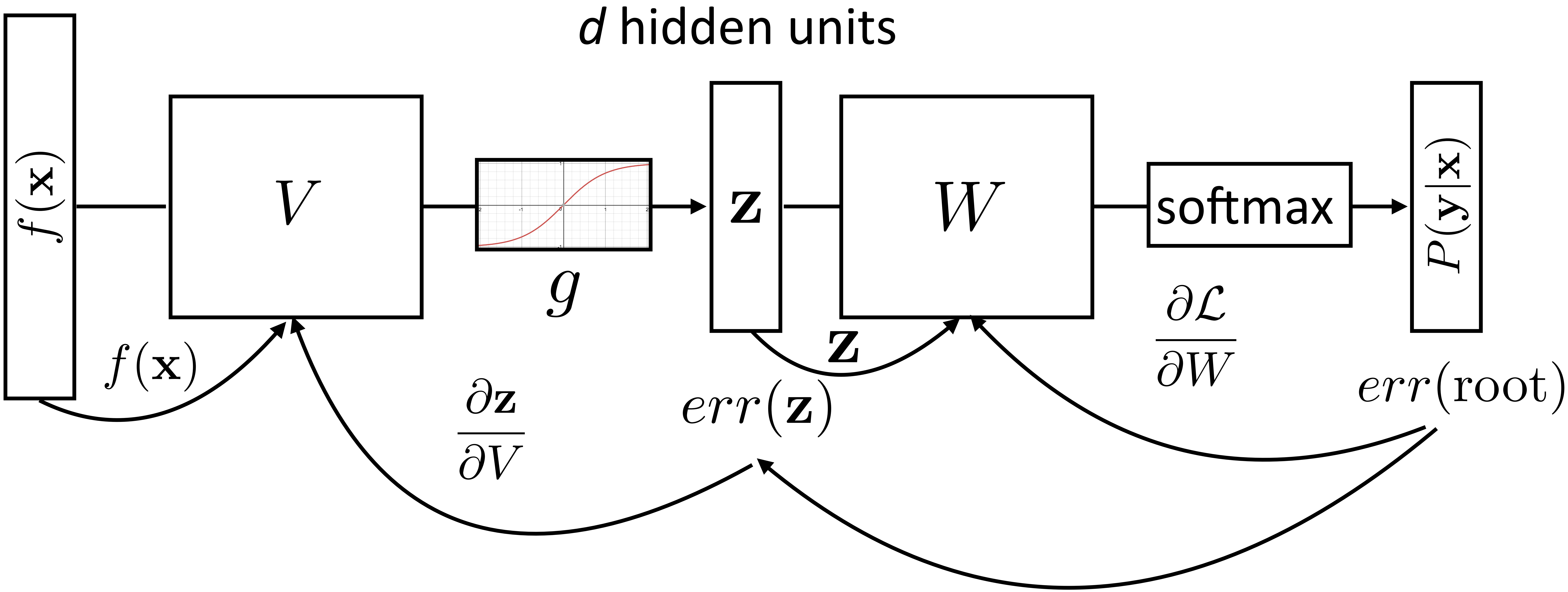
Recall: Feedforward NNs

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



Recall: Backpropagation

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



Applications

NLP with Feedforward Networks

- Part-of-speech tagging with FFNNs

??

Fed raises ***interest*** *rates in order to …*

- Word embeddings for each word form input
- ~1000 features here — smaller feature vector than in sparse models, but every feature fires on every example
- Weight matrix learns position-dependent processing of the words

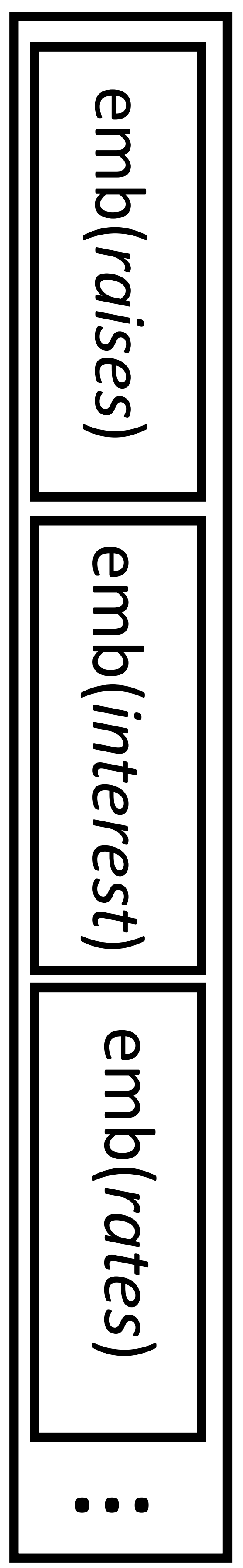
$f(x)$

previous word

curr word

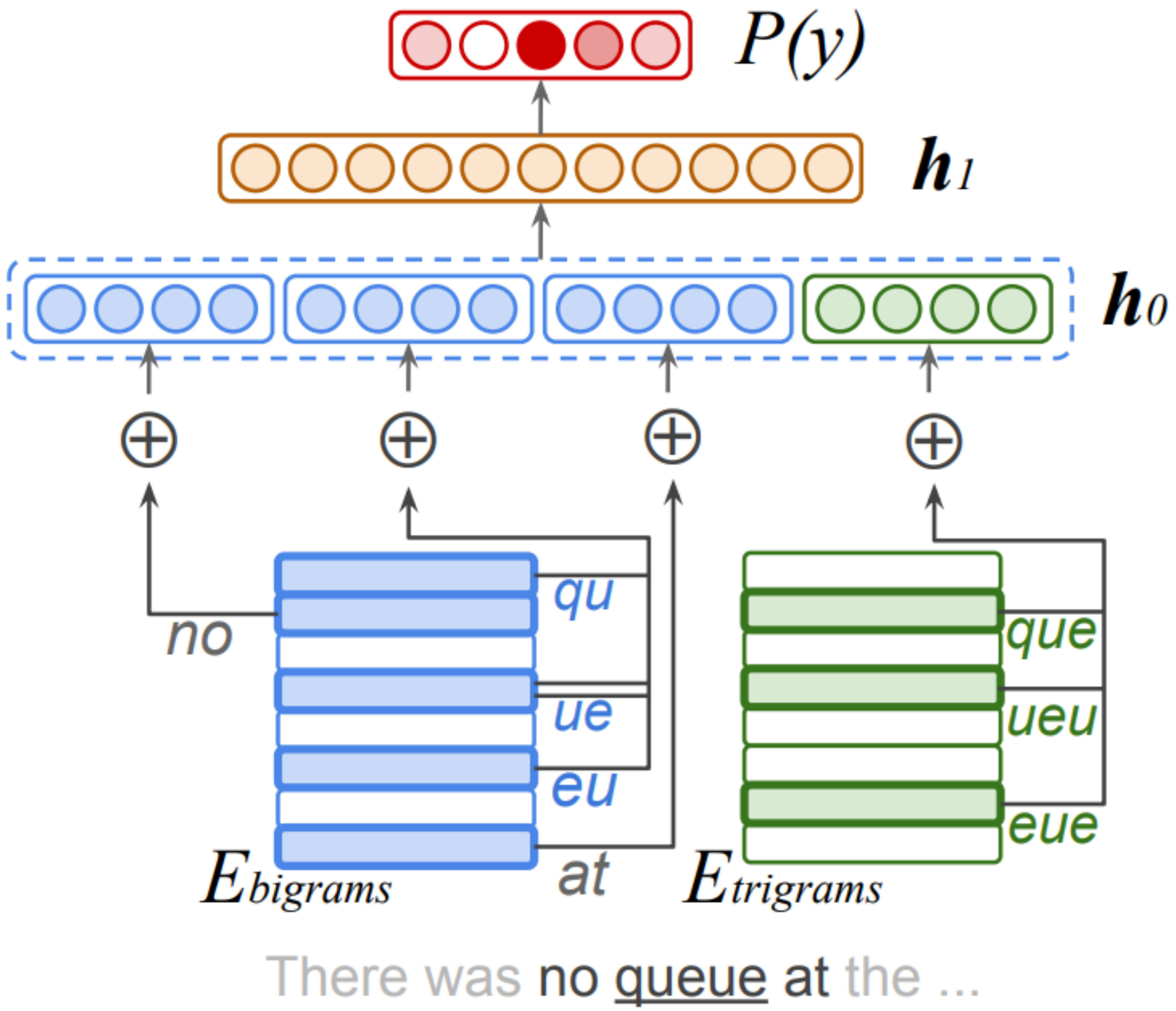
next word

other words, feats, etc.



Botha et al. (2017)

NLP with Feedforward Networks



- Hidden layer mixes these different signals and learns feature conjunctions

Botha et al. (2017)

NLP with Feedforward Networks

- Multilingual tagging results:

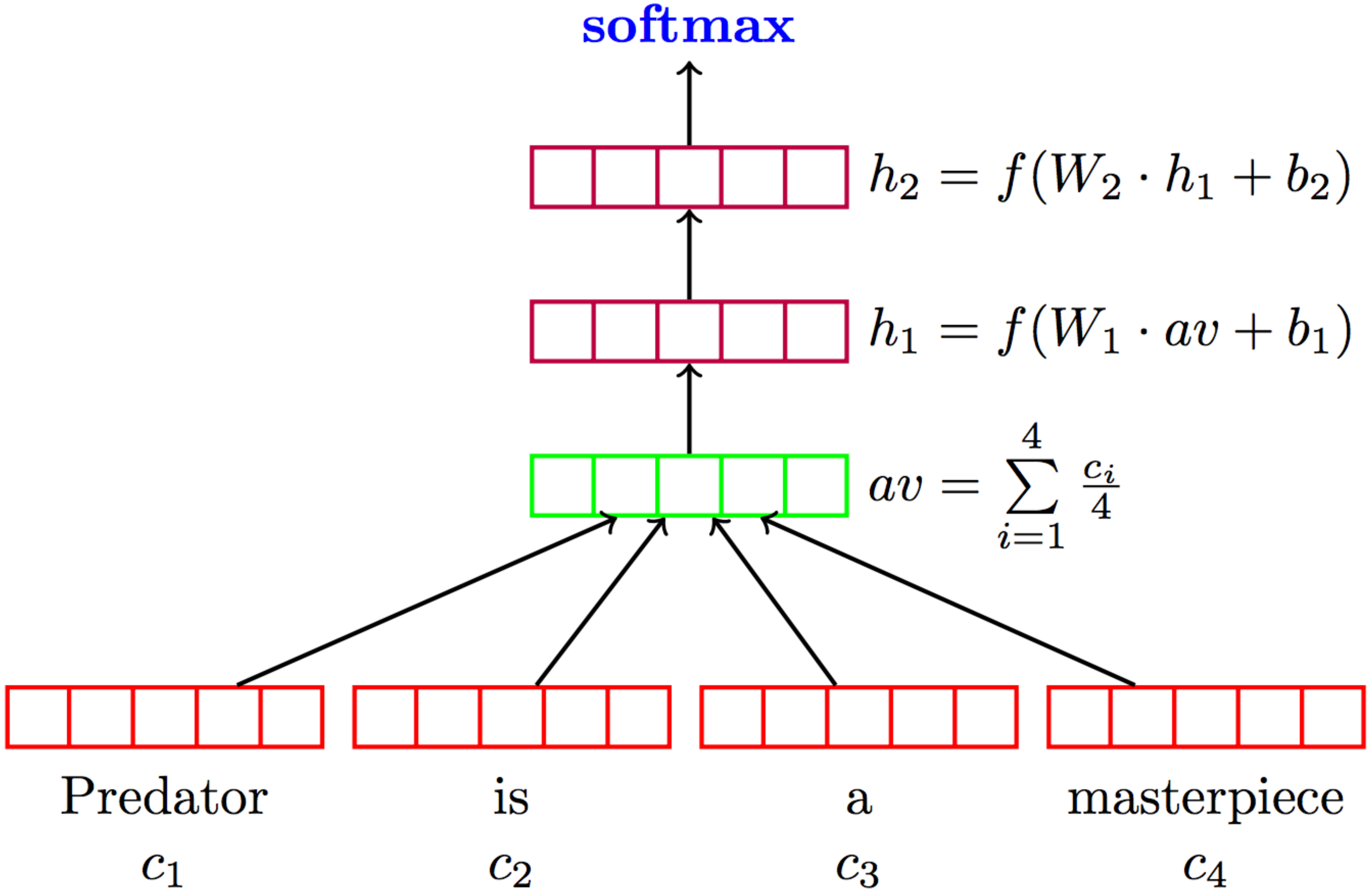
Model	Acc.	Wts.	MB	Ops.
Gillick et al. (2016)	95.06	900k	-	6.63m
Small FF	94.76	241k	0.6	0.27m
+Clusters	95.56	261k	1.0	0.31m
$\frac{1}{2}$ Dim.	95.39	143k	0.7	0.18m

- Gillick used LSTMs; this is smaller, faster, and better

Botha et al. (2017)

Sentiment Analysis

- Deep Averaging Networks: feedforward neural network on average of word embeddings from input



Iyyer et al. (2015)

Sentiment Analysis

Model	RT	SST fine	SST bin	IMDB	Time (s)
DAN-ROOT	—	46.9	85.7	—	**31**
DAN-RAND	77.3	45.4	83.2	88.8	136
DAN	80.3	47.7	86.3	89.4	136
NBOW-RAND	76.2	42.3	81.4	88.9	91
NBOW	79.0	43.6	83.6	89.0	91
BiNB	—	41.9	83.1	—	—
NBSVM-bi	79.4	—	—	91.2	—
RecNN*	77.7	43.2	82.4	—	—
RecNTN*	—	45.7	85.4	—	—
DRecNN	—	49.8	86.6	—	431
TreeLSTM	—	**50.6**	86.9	—	—
DCNN*	—	48.5	86.9	89.4	—
PVEC*	—	48.7	87.8	**92.6**	—
CNN-MC	**81.1**	47.4	**88.1**	—	2,452
WRRBM*	—	—	—	89.2	—

Iyyer et al. (2015)

Bag-of-words

Wang and Manning (2012)

Tree RNNs / CNNS / LSTMS

Kim (2014)

Implementation Details

Computation Graphs

- Computing gradients is hard! Computation graph abstraction allows us to define a computation symbolically and will do this for us

- Automatic differentiation: keep track of derivatives / be able to backpropagate through each function:

$$\texttt{y = x * x} \xrightarrow[\text{codegen}]{} \texttt{(y,dy) = (x * x, 2 * x * dx)}$$

- Use a library like Pytorch or Tensorflow. This class: Pytorch

Computation Graphs in Pytorch

- Define forward pass for $P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$

```
class FFNN(nn.Module):
    def __init__(self, inp, hid, out):
        super(FFNN, self).__init__()
        self.V = nn.Linear(inp, hid)
        self.g = nn.Tanh()
        self.W = nn.Linear(hid, out)
        self.softmax = nn.Softmax(dim=0)

    def forward(self, x):
        return self.softmax(self.W(self.g(self.V(x))))
```

Computation Graphs in Pytorch

$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$$

ei*: one-hot vector of the label (e.g., [0, 1, 0])

```
ffnn = FFNN()
def make_update(input, gold_label):
    ffnn.zero_grad() # clear gradient variables
    probs = ffnn.forward(input)
    loss = torch.neg(torch.log(probs)).dot(gold_label)
    loss.backward()
    optimizer.step()
```

Training a Model

Define a computation graph

For each epoch:

For each batch of data:

Compute loss on batch

Autograd to compute gradients

Take step with optimizer

Decode test set

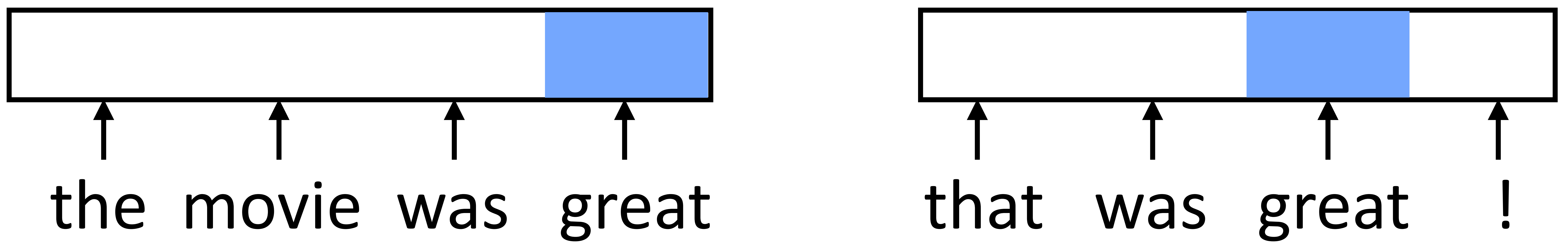
Today

- Recurrent neural networks
- Vanishing gradient problem
- LSTMs / GRUs
- Applications / visualizations

RNN Basics

RNN Motivation

- Feedforward NNs can't handle variable length input: each position in the feature vector has fixed semantics



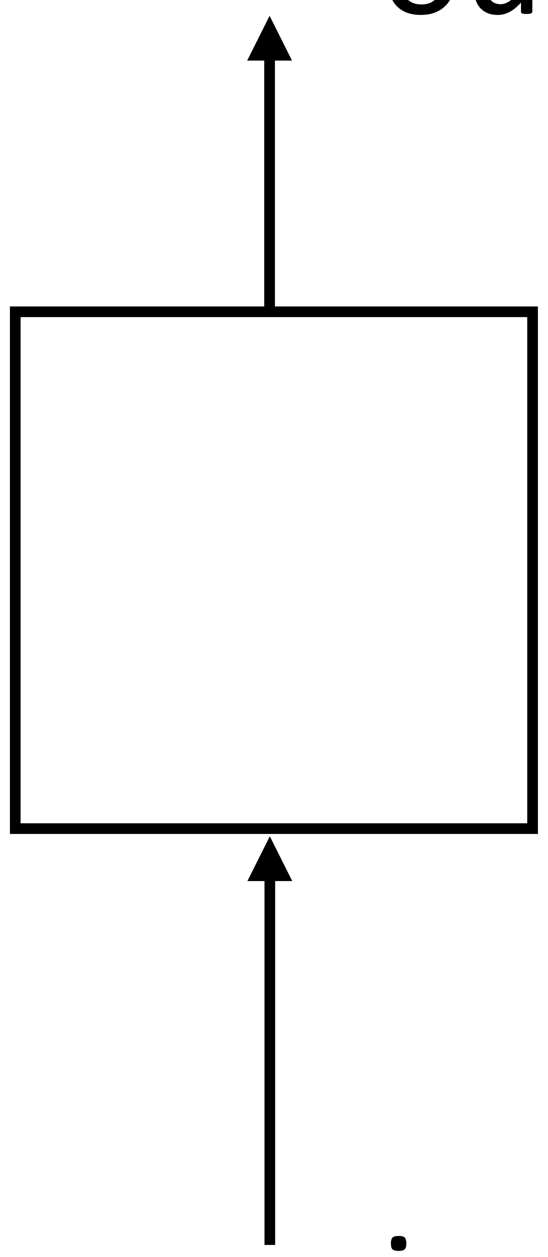
- These don't look related (*great* is in two different orthogonal subspaces)

- Instead, we'd like to:

1) Process each word in a uniform way

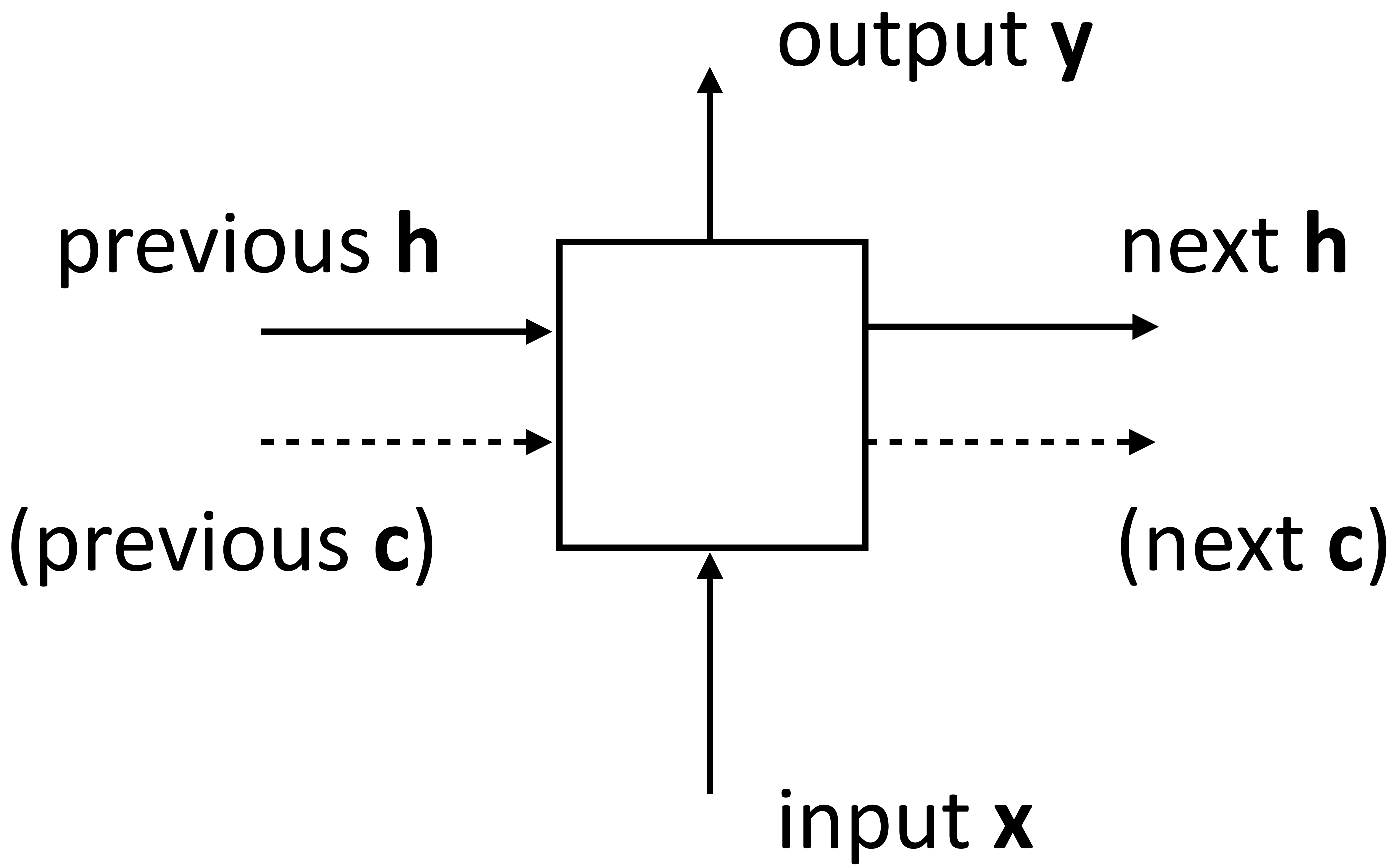
2) …while still exploiting the context that that token occurs in

Feed Forward



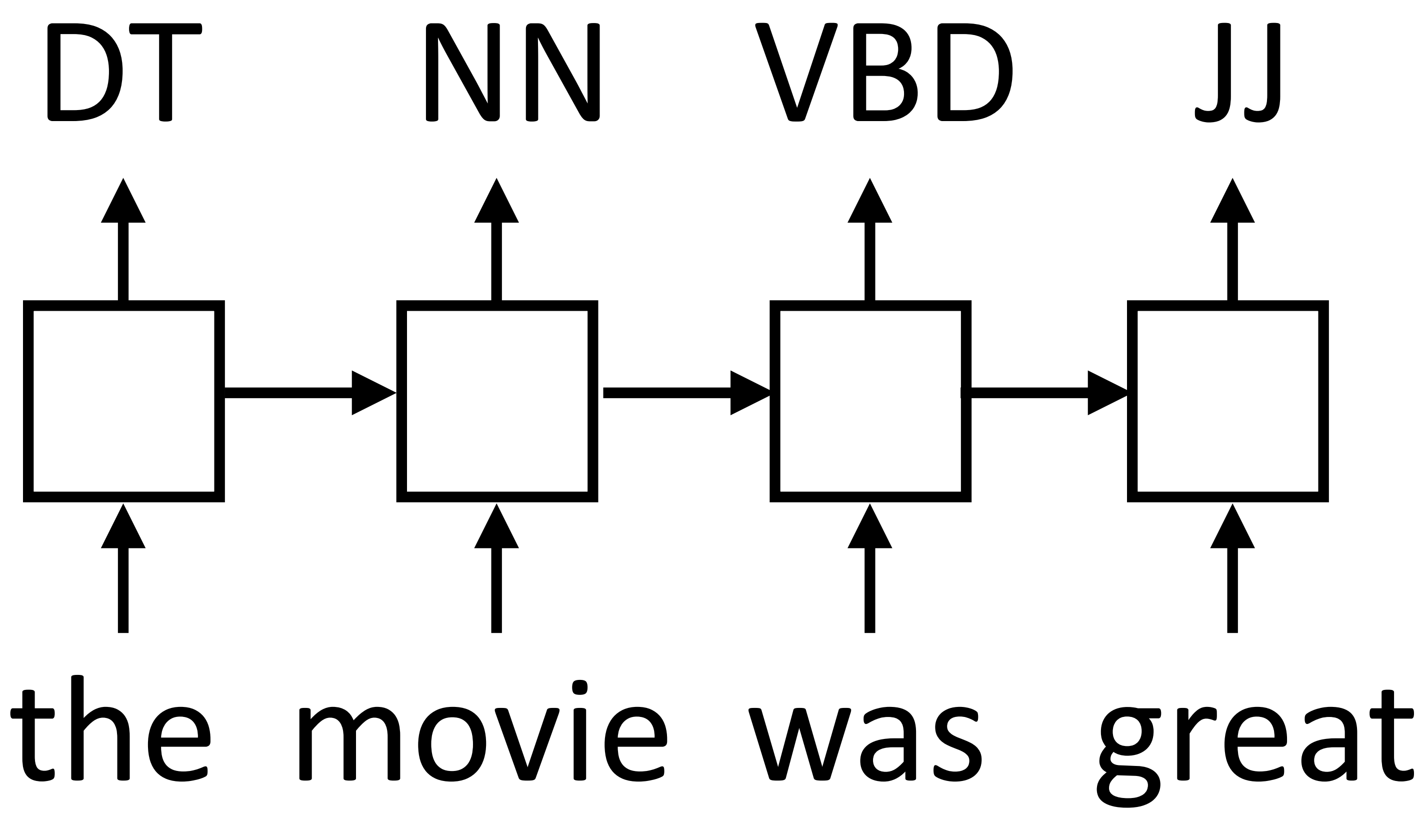
RNN Abstraction

- Cell that takes some input **x**, has some hidden state **h**, and updates that hidden state and produces output **y** (all vector-valued)



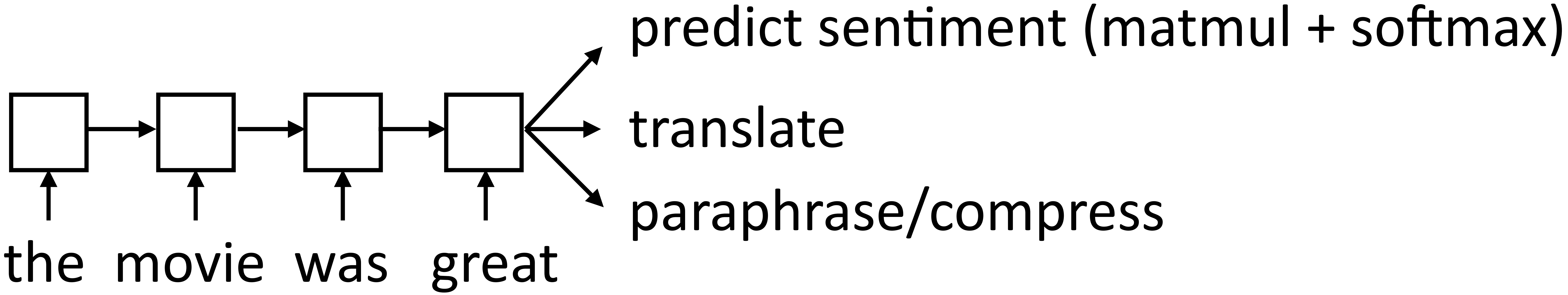
RNN Uses

- Transducer: make some prediction for each element in a sequence

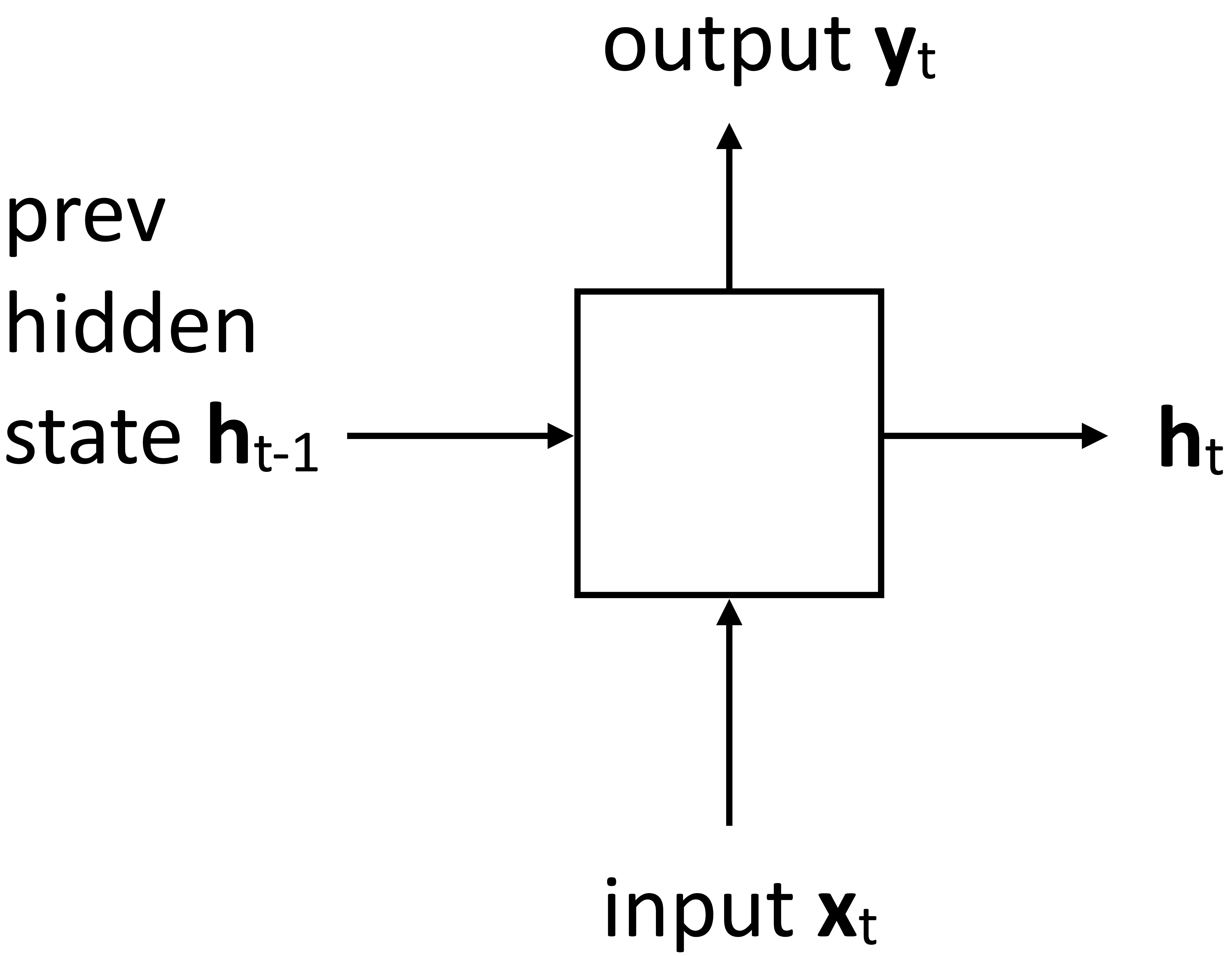


output **y** = score for each tag, then softmax

- Acceptor/encoder: encode a sequence into a fixed-sized vector and use that for some purpose



Elman Networks



$$\mathbf{h}_t = \tanh(W\mathbf{x}_t + V\mathbf{h}_{t-1} + \mathbf{b}_h)$$

- Updates hidden state based on input and current hidden state

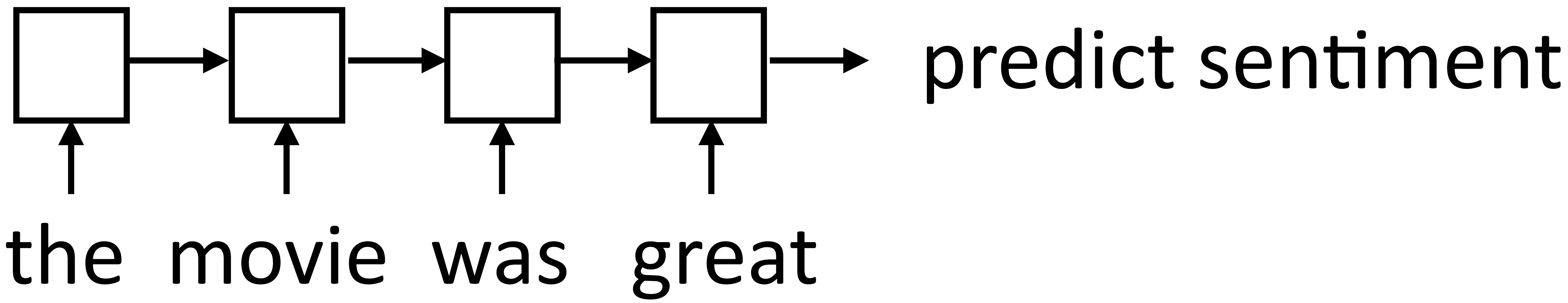
$$\mathbf{y}_t = \tanh(U\mathbf{h_t} + \mathbf{b}_y)$$

- Computes output from hidden state

- Long history! (invented in the late 1980s)

Elman (1990)

Training Elman Networks



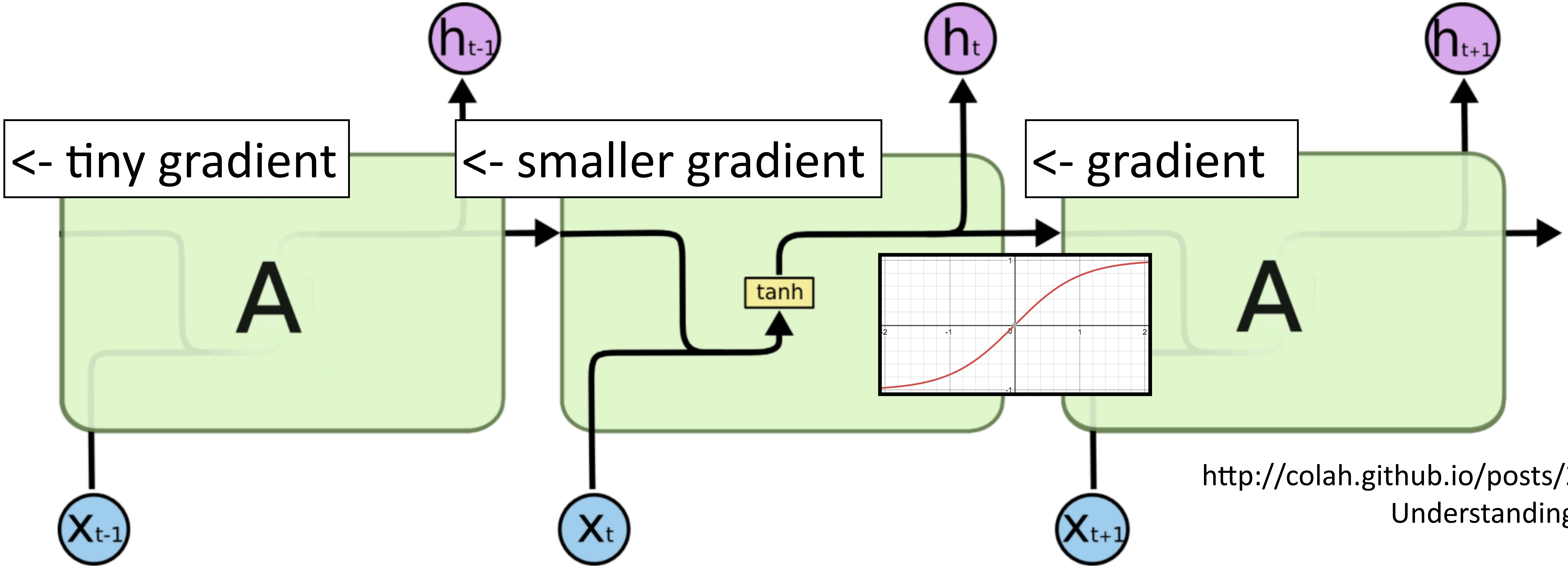
- "Backpropagation through time": build the network as one big computation graph, some parameters are shared

- RNN potentially needs to learn how to "remember" information for a long time!

it was my favorite movie of 2016, though it wasn't without problems -> +

- "Correct" parameter update is to do a better job of remembering the sentiment of *favorite*

Vanishing Gradient



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

- Gradient diminishes going through tanh; if not in [-2, 2], gradient is almost 0
- Repeated multiplication by V causes problems $\mathbf{h}_t = \tanh(W\mathbf{x}_t + V\mathbf{h}_{t-1} + \mathbf{b}_h)$

LSTMs/GRUs

Gated Connections

- Designed to fix "vanishing gradient" problem using *gates*

$$\mathbf{h}_t = \mathbf{h}_{t-1} \odot \mathbf{f} + \mathrm{func}(\mathbf{x}_t)$$

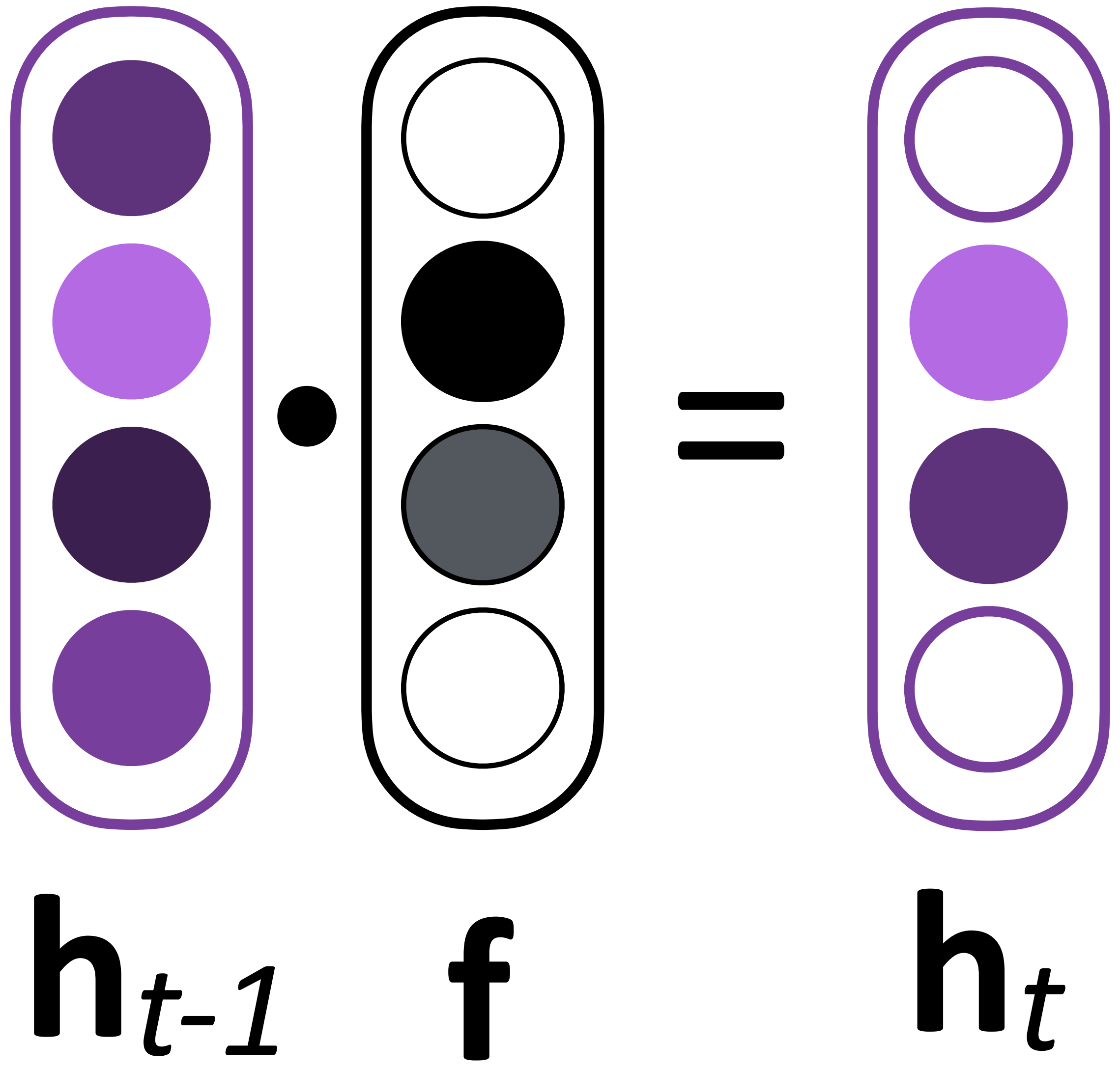
gated

$$\mathbf{h}_t = \tanh(W\mathbf{x}_t + V\mathbf{h}_{t-1} + \mathbf{b}_h)$$

Elman

- Vector-valued "forget gate" **f** computed based on input and previous hidden state

$$\mathbf{f} = \sigma(W^{xf}\mathbf{x}_t + W^{hf}\mathbf{h}_{t-1})$$



- Sigmoid: elements of **f** are in (0, 1)

- If **f** ≈ **1**, we simply sum up a function of all inputs — gradient doesn't vanish! More stable without matrix multiply (*V*) as well

LSTMs

- "Long short-term memory" network: hidden state is a "short-term" memory
- (Hochreiter & Schmidhuber 1997)
- "Cell" **c** in addition to hidden state **h**

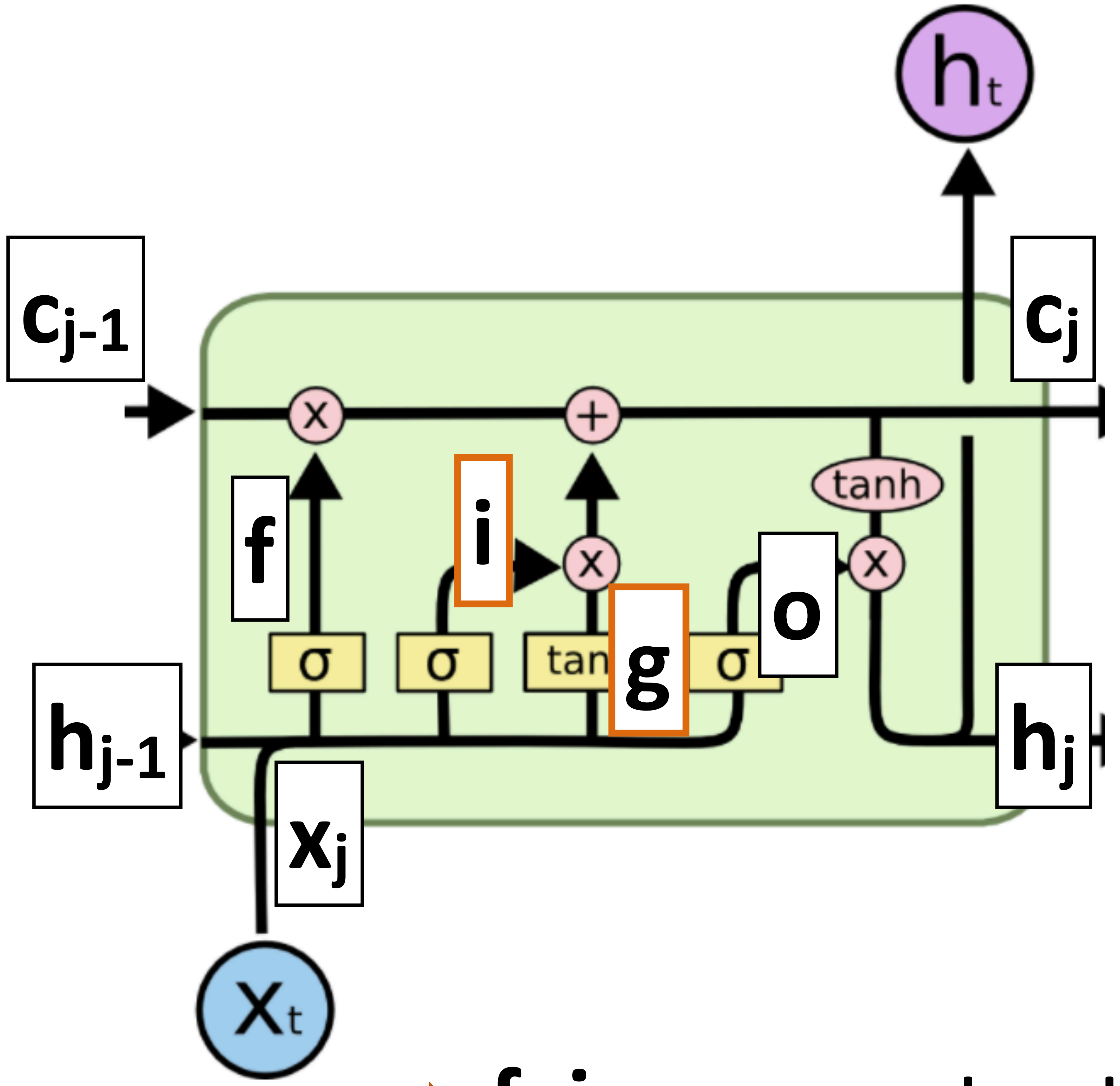
$$\mathbf{c}_t = \mathbf{c}_{t-1} \odot \mathbf{f} + \boxed{\mathrm{func}(\mathbf{x}_t, \mathbf{h}_{t-1})}$$

- Vector-valued forget gate **f** depends on the **h** hidden state

$$\mathbf{f} = \sigma(W^{xf}\mathbf{x}_t + W^{hf}\mathbf{h}_{t-1})$$

- Basic communication flow: **x** -> **c** -> **h ->** output**,** each step of this process is gated in addition to gates from previous timesteps

LSTMs



$$\mathbf{c_j} = \mathbf{c_{j-1}} \odot \mathbf{f} + \boxed{\mathbf{g} \odot \mathbf{i}}$$

$$\mathbf{f} = \sigma(\mathbf{x_j W^{xf}} + \mathbf{h_{j-1} W^{hf}})$$

$$\boxed{\begin{aligned} \mathbf{g} &= \tanh(\mathbf{x_j W^{xg}} + \mathbf{h_{j-1} W^{hg}}) \\ \mathbf{i} &= \sigma(\mathbf{x_j W^{xi}} + \mathbf{h_{j-1} W^{hi}}) \end{aligned}}$$

$$\mathbf{h_j} = \tanh(\mathbf{c_j}) \odot \mathbf{o}$$

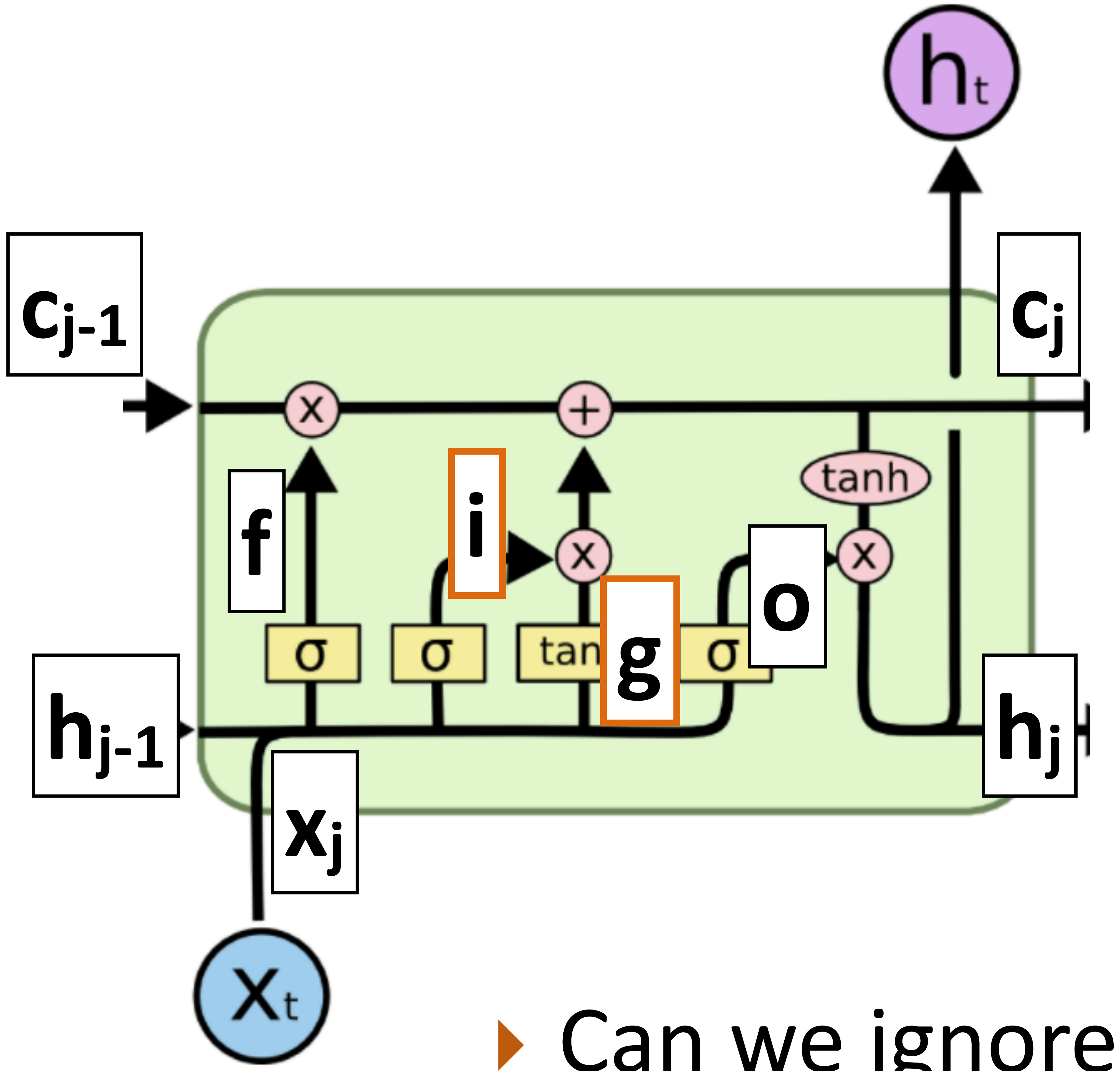
$$\mathbf{o} = \sigma(\mathbf{x_j W^{xo}} + \mathbf{h_{j-1} W^{ho}})$$

- **f**, **i**, **o** are gates that control information flow
- **g** reflects the main computation of the cell

Goldberg lecture notes

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

LSTMs



$$\mathbf{c_j} = \mathbf{c_{j-1}} \odot \mathbf{f} + \boxed{\mathbf{g} \odot \mathbf{i}}$$

$$\mathbf{f} = \sigma(\mathbf{x_j}\mathbf{W^{xf}} + \mathbf{h_{j-1}}\mathbf{W^{hf}})$$

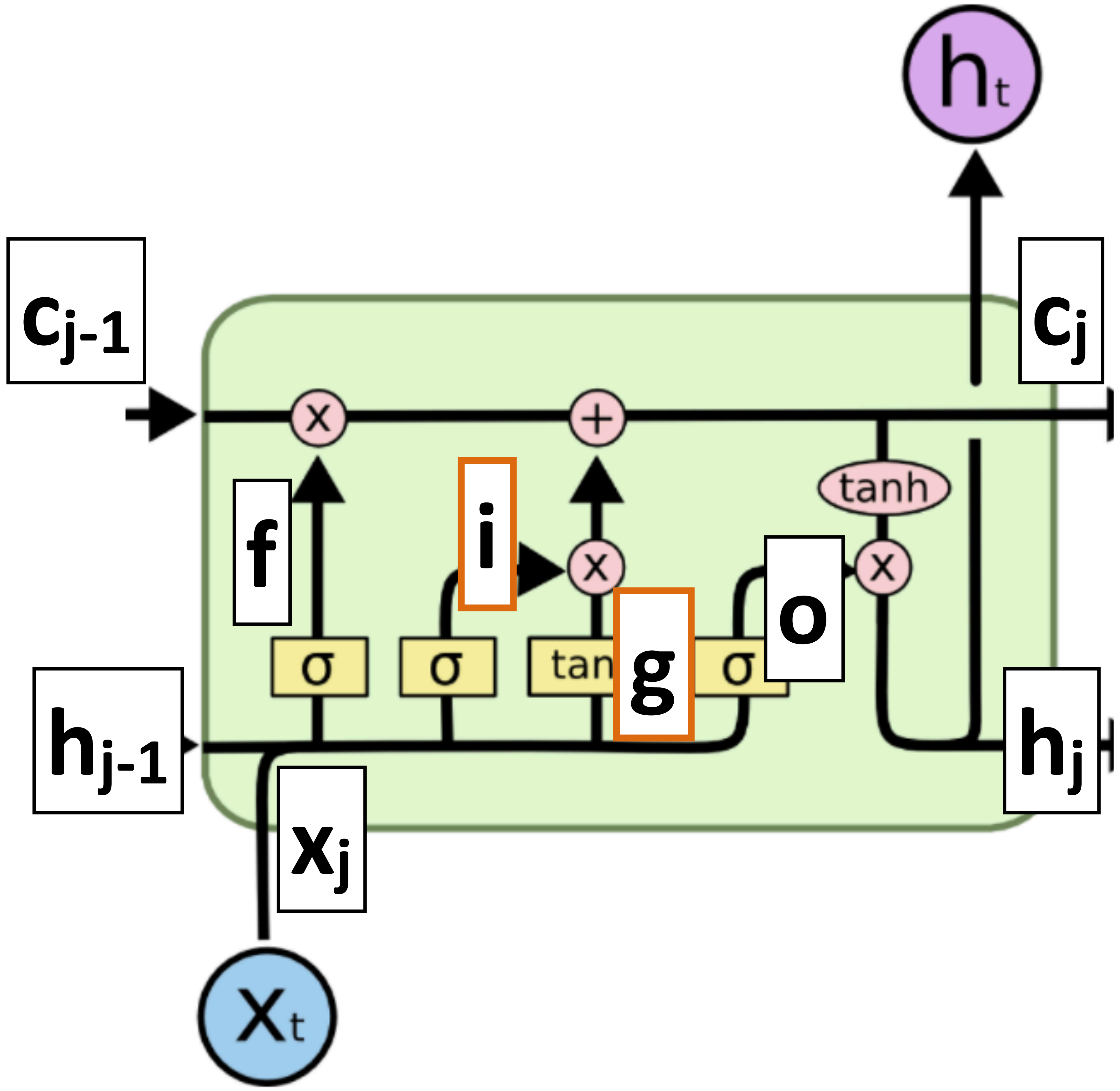
$$\boxed{\begin{aligned}\mathbf{g} &= \tanh(\mathbf{x_j}\mathbf{W^{xg}} + \mathbf{h_{j-1}}\mathbf{W^{hg}}) \\ \mathbf{i} &= \sigma(\mathbf{x_j}\mathbf{W^{xi}} + \mathbf{h_{j-1}}\mathbf{W^{hi}})\end{aligned}}$$

$$\mathbf{h_j} = \tanh(\mathbf{c_j}) \odot \mathbf{o}$$

$$\mathbf{o} = \sigma(\mathbf{x_j}\mathbf{W^{xo}} + \mathbf{h_{j-1}}\mathbf{W^{ho}})$$

- Can we ignore the old value of **c** for this timestep?
- Can an LSTM sum up its inputs **x**?
- Can we ignore a particular input **x**?
- Can we output something without changing **c**?

LSTMs

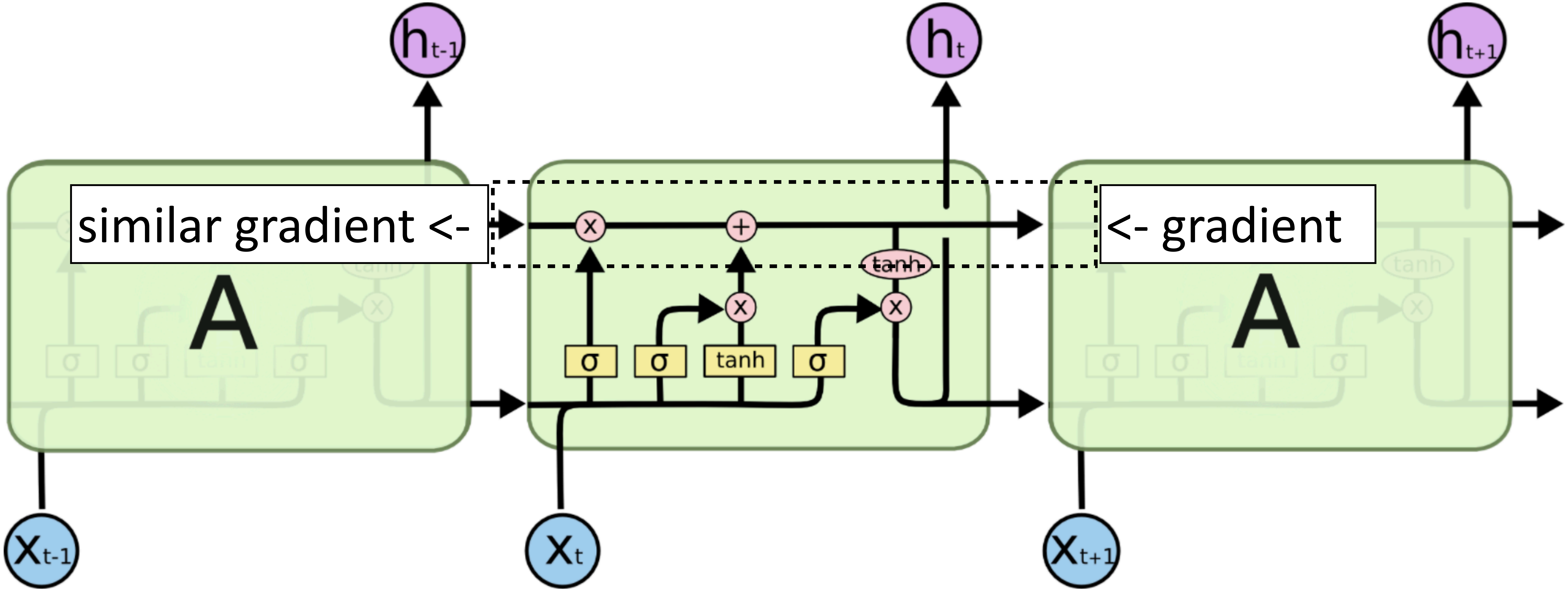


- Ignoring recurrent state entirely:
  - Lets us get feedforward layer over token
- Ignoring input:
  - Lets us discard stopwords
- Summing inputs:
  - Lets us compute a bag-of-words representation

Goldberg lecture notes

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

LSTMs

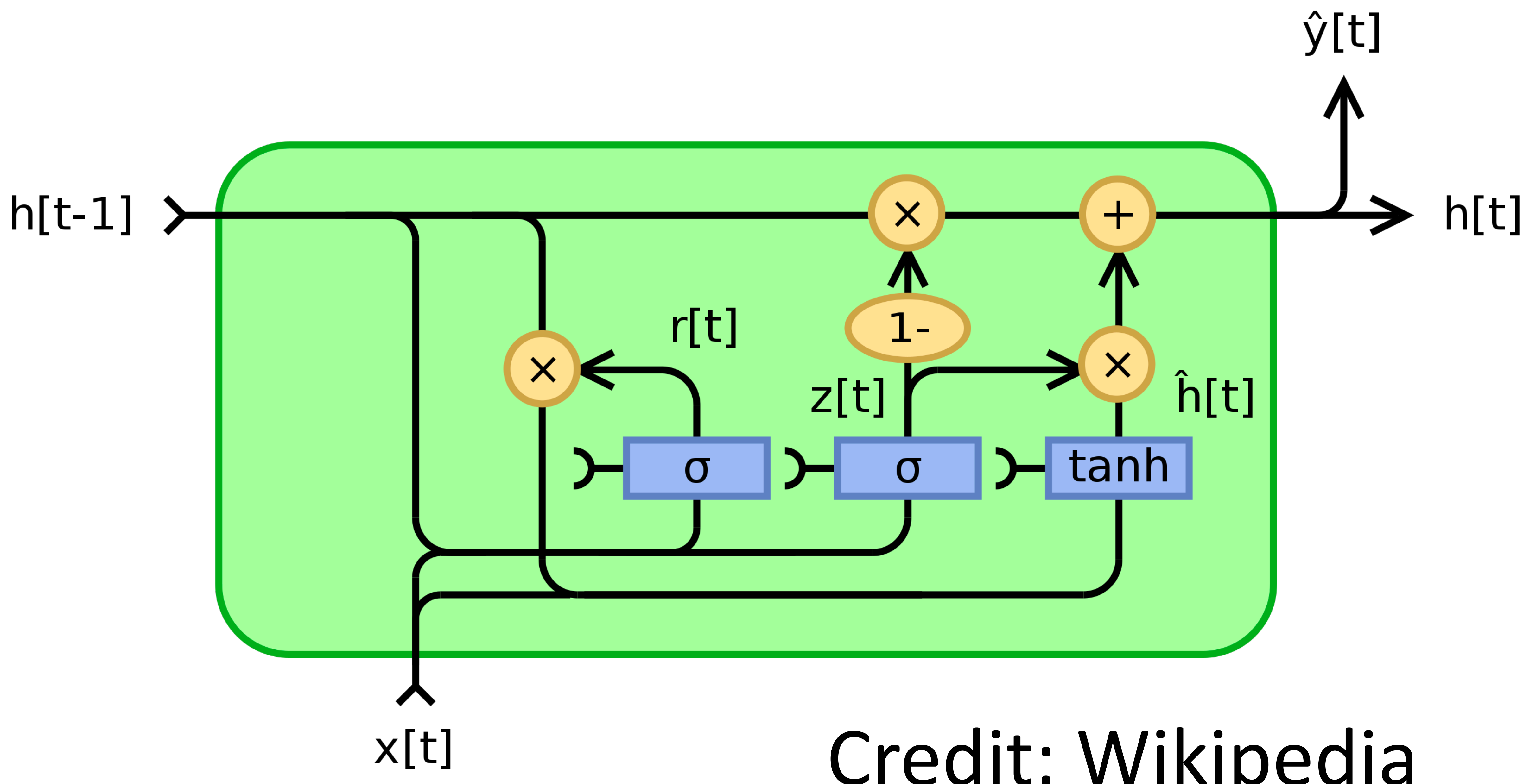


- Gradient still diminishes, but in a controlled way and generally by less — usually initialize forget gate = 1 to remember everything to start

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

Gated Recurrent Unit (GRU)

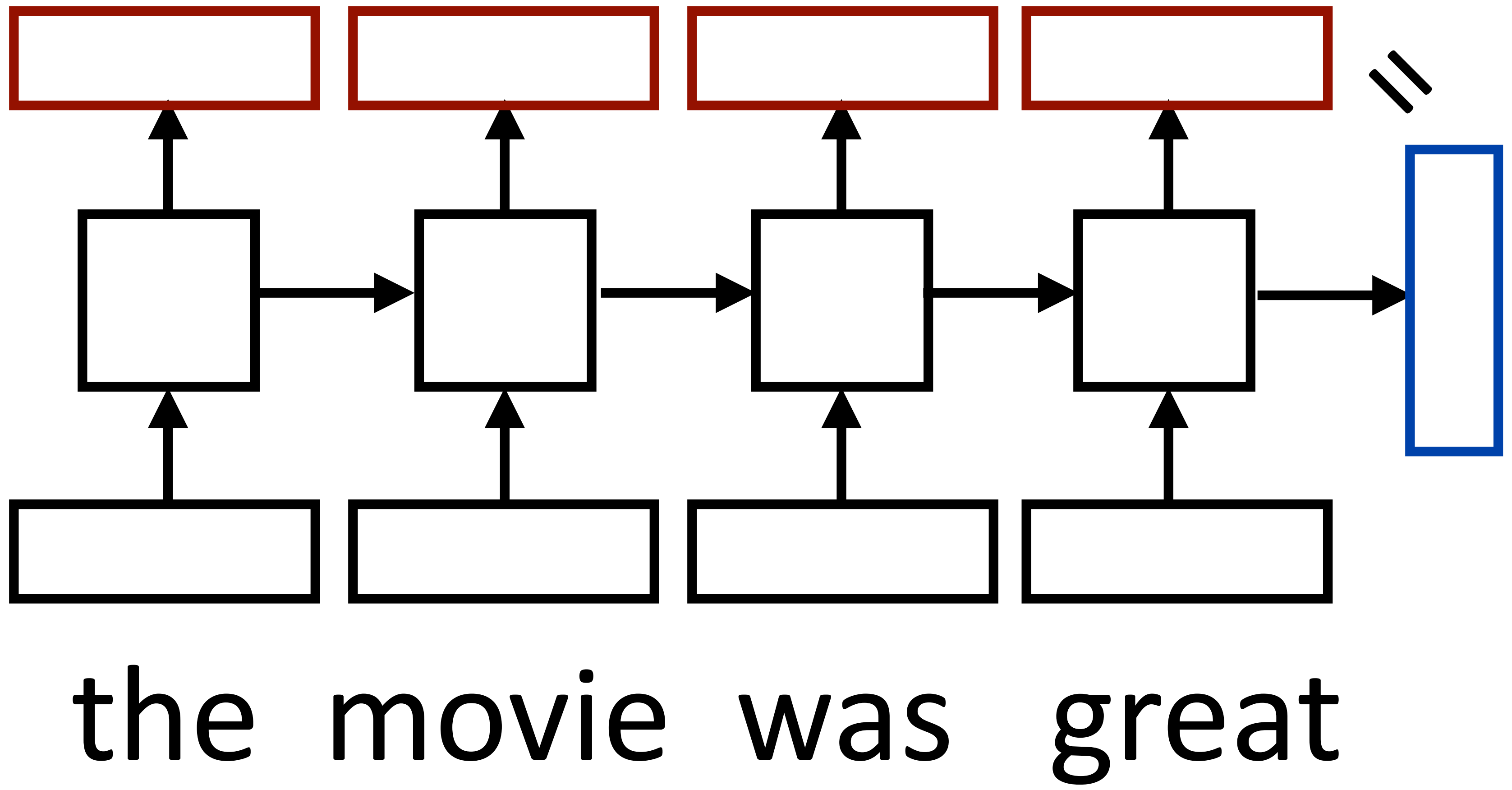
- **z** is update, **r** is reset
- The single hidden state and simpler update gate gives simpler mixing semantics than in LSTMs
- Faster to train and sometimes works better than LSTMs, often a tossup



Credit: Wikipedia

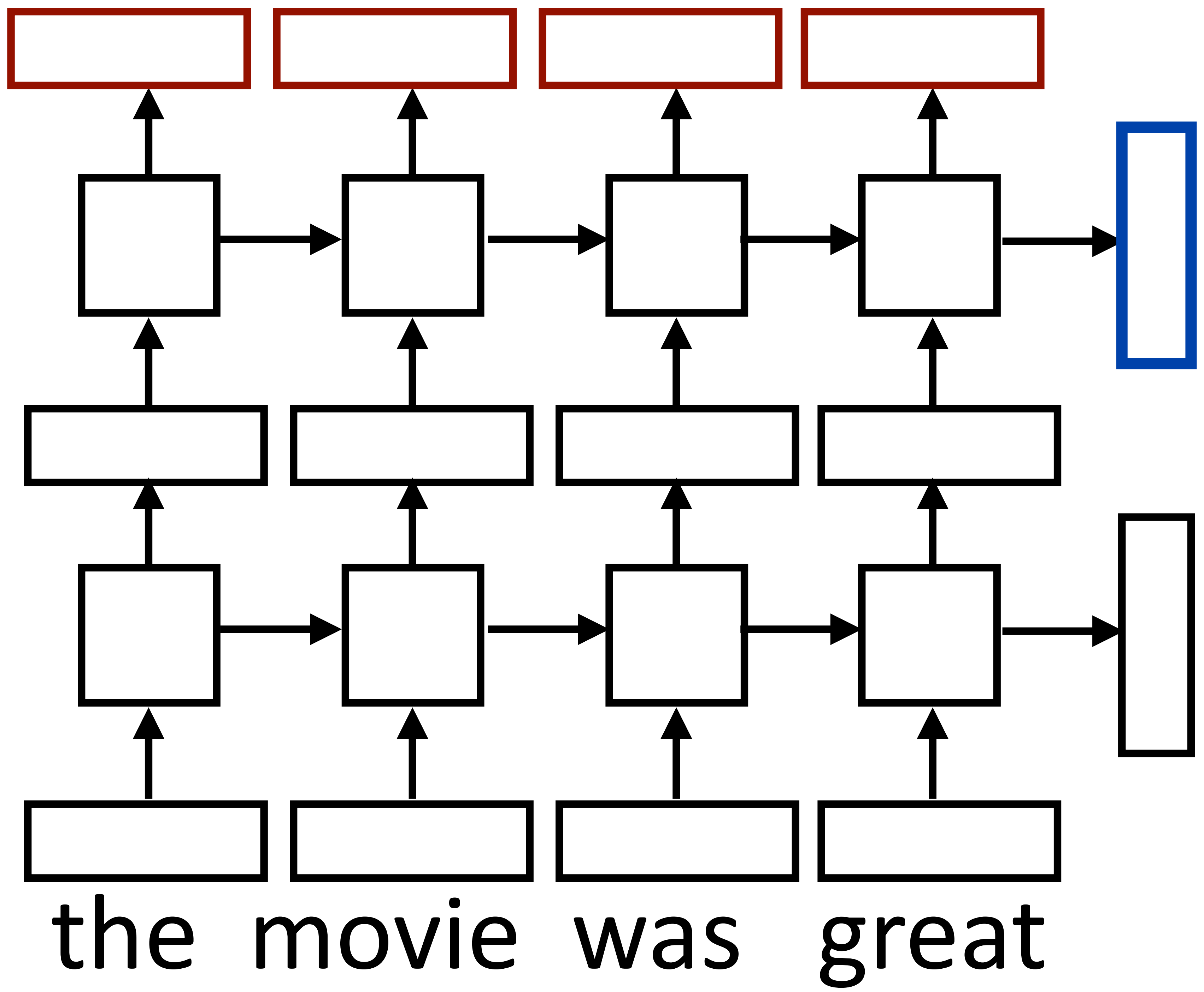
$$
\begin{aligned}
z_t &= \sigma_g(W_z x_t + U_z h_{t-1} + b_z) \\
r_t &= \sigma_g(W_r x_t + U_r h_{t-1} + b_r) \\
h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \sigma_h(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h)
\end{aligned}
$$

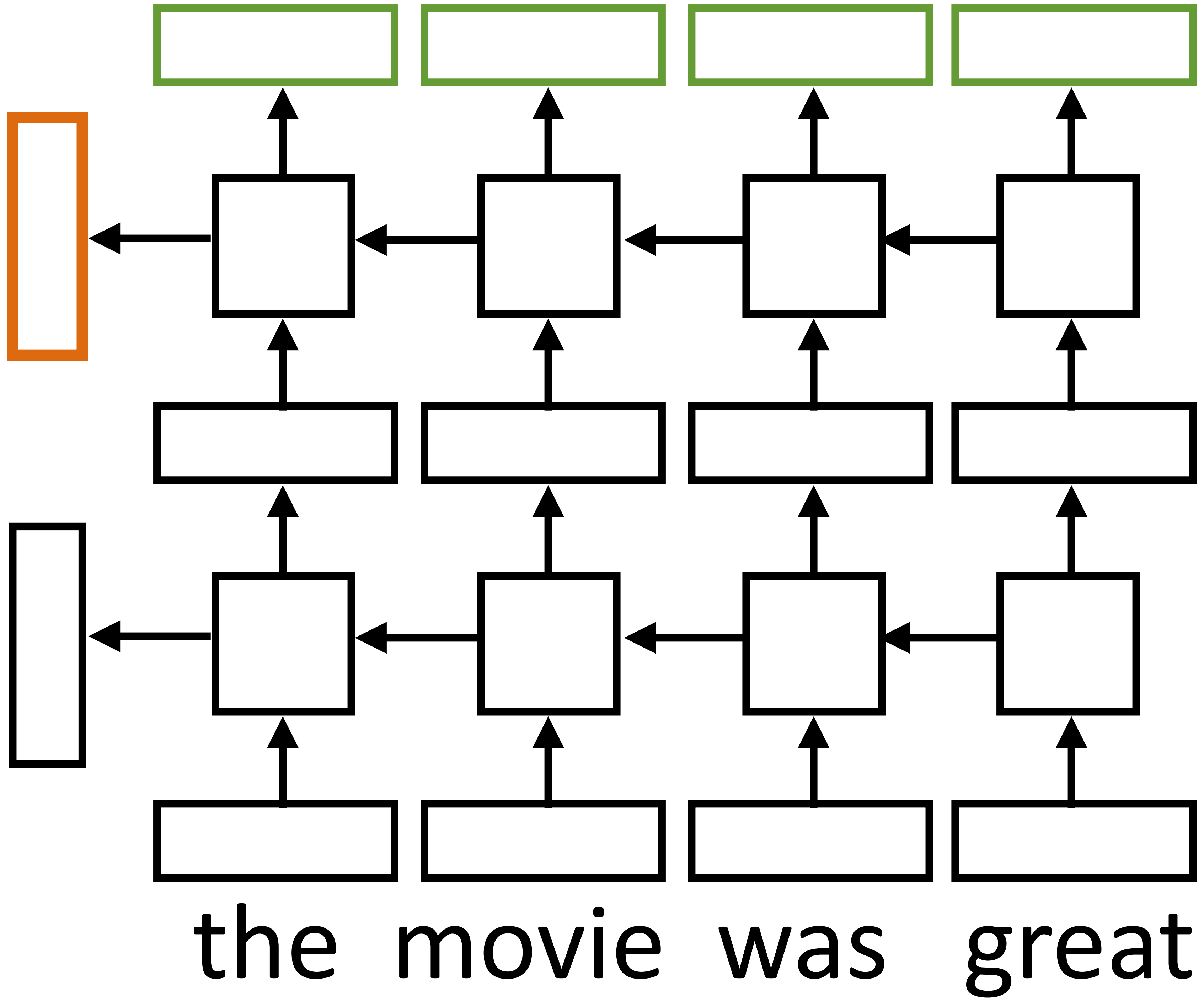
What do RNNs produce?



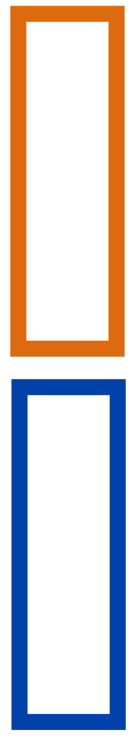
- Encoding of the sentence — can pass this a decoder or make a classification decision about the sentence
- Encoding of each word — can pass this to another layer to make a prediction (can also pool these to get a different sentence encoding)
- RNN can be viewed as a transformation of a sequence of vectors into a sequence of context-dependent vectors

Multilayer Bidirectional RNN

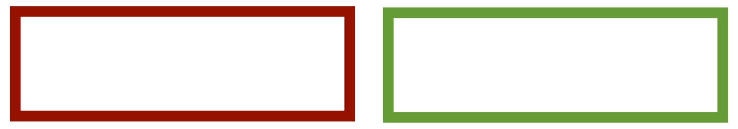




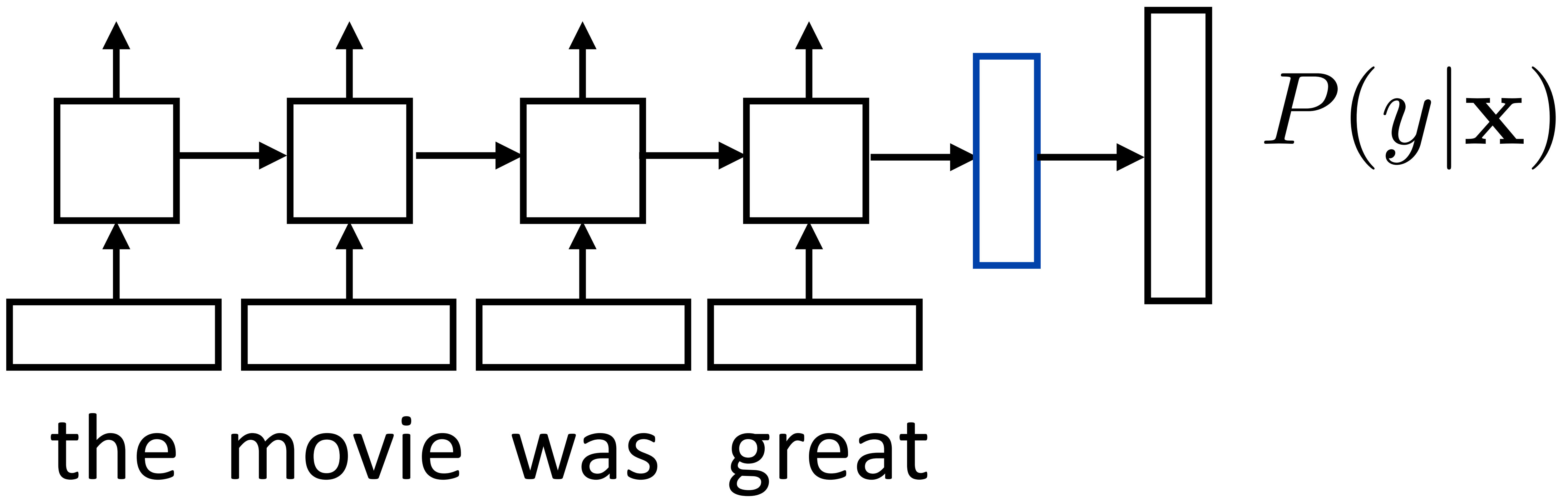
- Sentence classification based on concatenation of both final outputs



- Token classification based on concatenation of both directions' token representations

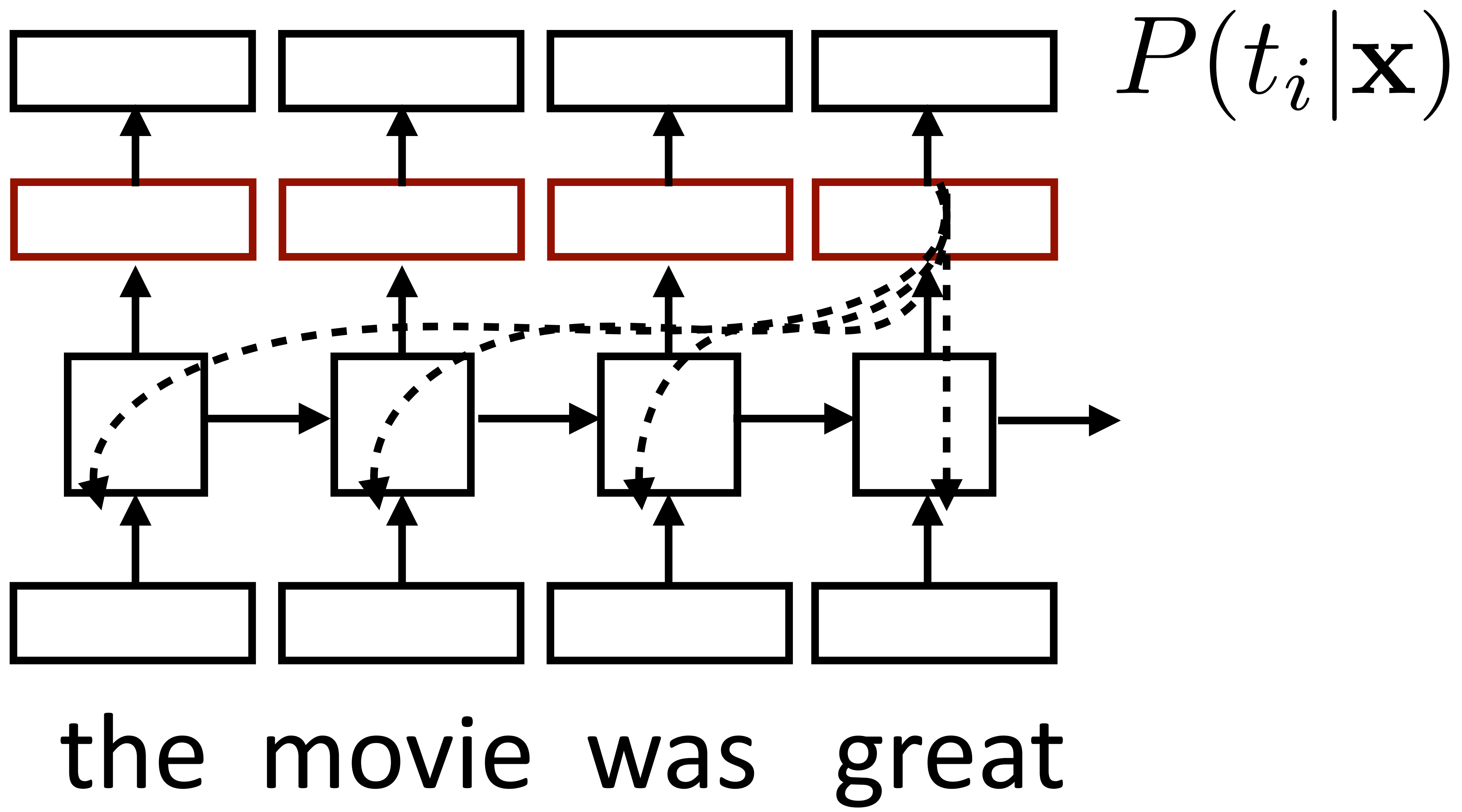


Training RNNs



- Loss = negative log likelihood of probability of gold label (or use SVM or other loss)
- Backpropagate through entire network
- Example: sentiment analysis

Training RNNs



- Loss = negative log likelihood of probability of gold predictions, summed over the tags
- Loss terms filter back through network
- Example: language modeling (predict next word given context) or POS tagging

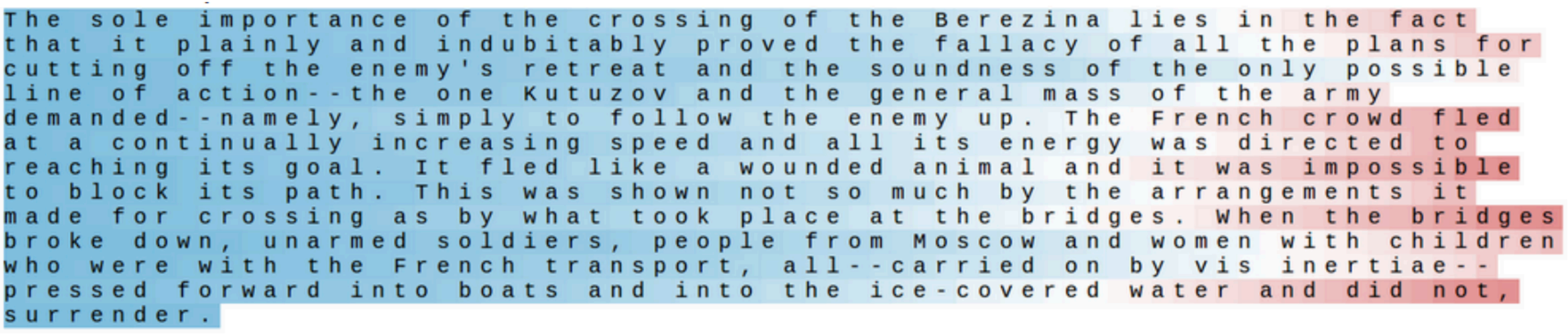
Applications

What can LSTMs model?

- Sentiment
  - Encode one sentence, predict
- Language models
  - Move left-to-right, per-token prediction
- Translation
  - Encode sentence + then decode, use token predictions for attention weights (later in the course)

Visualizing LSTMs

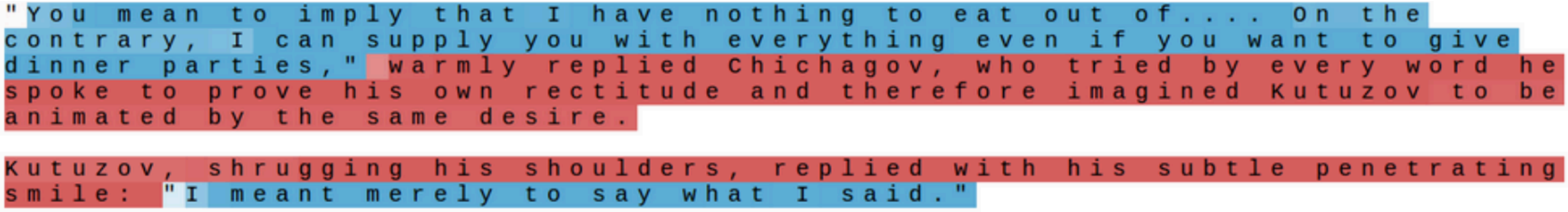
- Train *character* LSTM language model (predict next character based on history) over two datasets: War and Peace and Linux kernel source code
- Visualize activations of specific cells (components of **c**) to understand them
- Counter: know when to generate \n



Karpathy et al. (2015)

Visualizing LSTMs

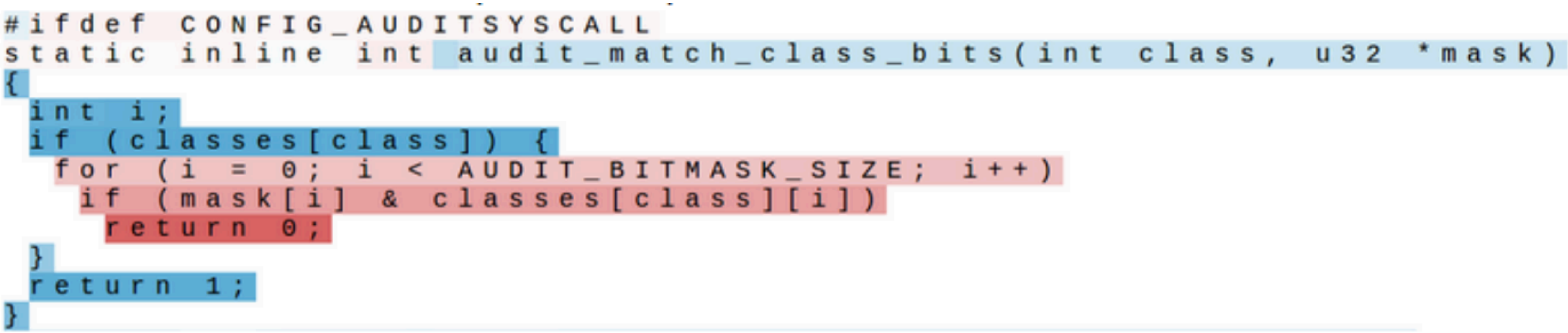
- Train *character* LSTM language model (predict next character based on history) over two datasets: War and Peace and Linux kernel source code
- Visualize activations of specific cells to see what they track
- Binary switch: tells us if we're in a quote or not



Karpathy et al. (2015)

Visualizing LSTMs

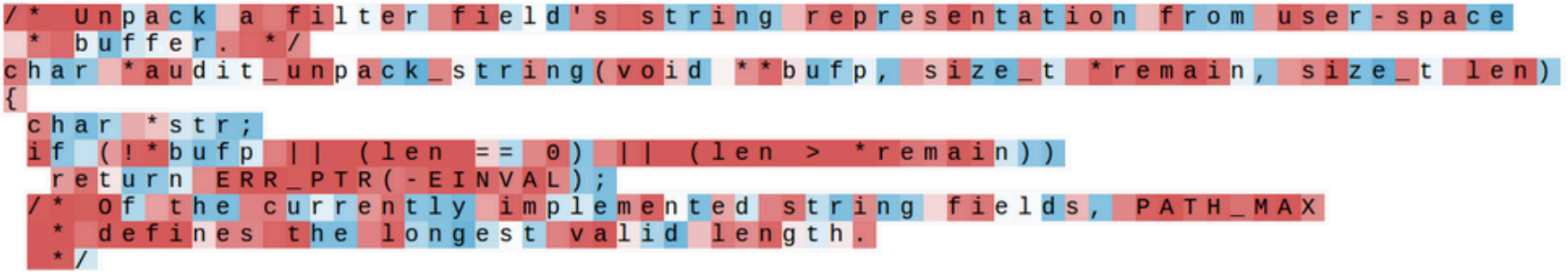
- Train *character* LSTM language model (predict next character based on history) over two datasets: War and Peace and Linux kernel source code
- Visualize activations of specific cells to see what they track
- Stack: activation based on indentation



Karpathy et al. (2015)

Visualizing LSTMs

- Train *character* LSTM language model (predict next character based on history) over two datasets: War and Peace and Linux kernel source code
- Visualize activations of specific cells to see what they track
- Uninterpretable: probably doing double-duty, or only makes sense in the context of another activation



Karpathy et al. (2015)

What can LSTMs model?

- Sentiment
  - Encode one sentence, predict
- Language models
  - Move left-to-right, per-token prediction
- Translation
  - Encode sentence + then decode, use token predictions for attention weights (later in the course)
- Textual entailment
  - Encode two sentences, predict

Sentiment Analysis

- Semi-supervised method: initialize the language model by training to reproduce the document in a seq2seq fashion (discussed in a few lectures), called a sequential autoencoder

Model	Test error rate
LSTM with tuning and dropout	13.50%
LSTM initialized with word2vec embeddings	10.00%
LM-LSTM (see Section 2)	7.64%
SA-LSTM (see Figure 1)	7.24%
Full+Unlabeled+BoW [21]	11.11%
WRRBM + BoW (bnc) [21]	10.77%
NBSVM-bi (Naïve Bayes SVM with bigrams) [35]	8.78%
seq2-bow*n*-CNN (ConvNet with dynamic pooling) [11]	7.67%
Paragraph Vectors [18]	7.42%

better than tuned Naive Bayes when using the SA trick

Dai and Le (2015)

Natural Language Inference

Premise		Hypothesis
A boy plays in the snow	*entails*	A boy is outside
A man inspects the uniform of a figure	*contradicts*	The man is sleeping
An older and younger man smiling	*neutral*	Two men are smiling and laughing at cats playing

- Long history of this task: "Recognizing Textual Entailment" challenge in 2006 (Dagan, Glickman, Magnini)
- Early datasets: small (hundreds of pairs), very ambitious (lots of world knowledge, temporal reasoning, etc.)

SNLI Dataset

- Show people captions for (unseen) images and solicit entailed / neural / contradictory statements
- >500,000 sentence pairs
- Encode each sentence and process

100D LSTM: 78% accuracy

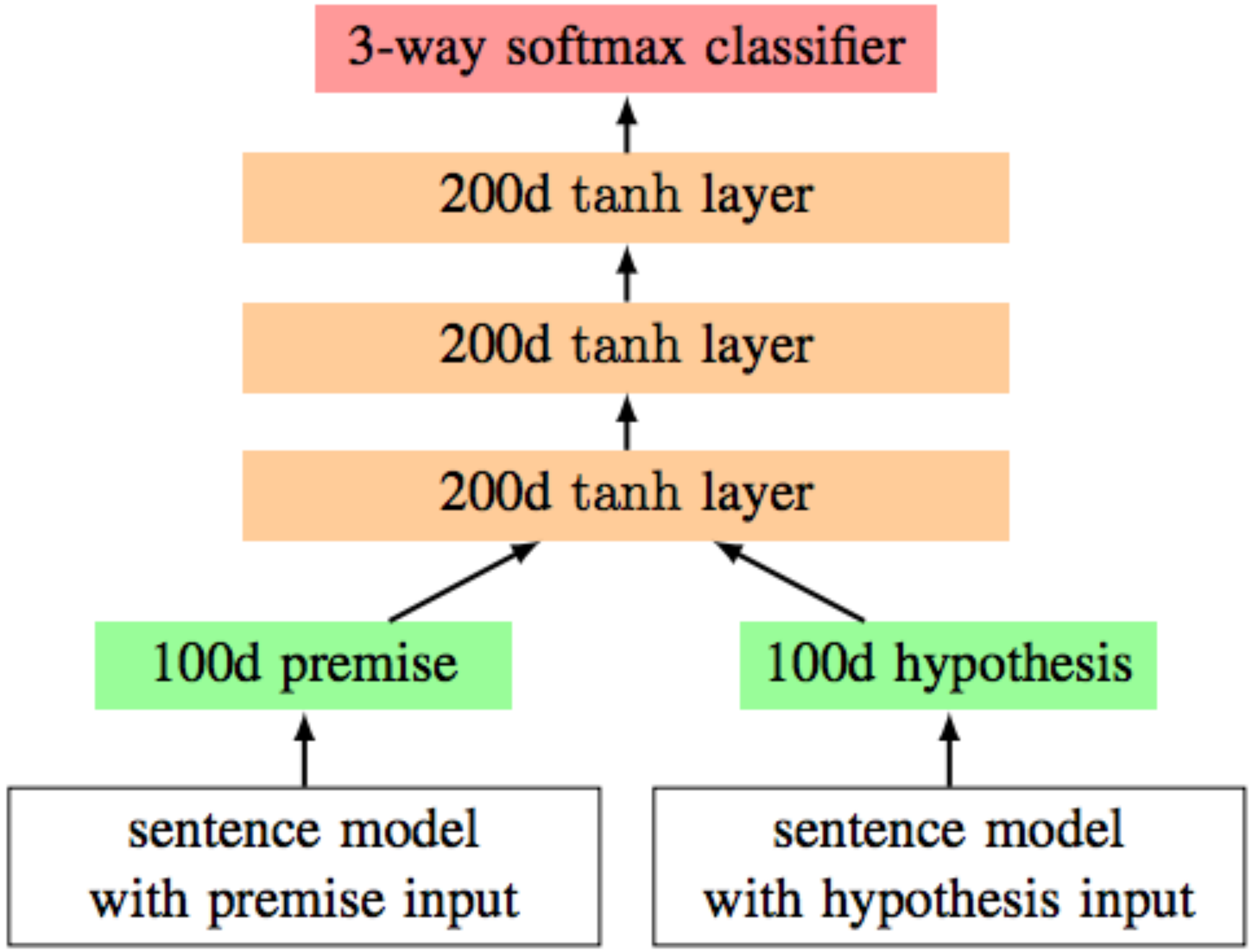
300D LSTM: 80% accuracy

(Bowman et al., 2016)

300D BiLSTM: 83% accuracy

(Liu et al., 2016)

- Later: better models for this



Bowman et al. (2015)

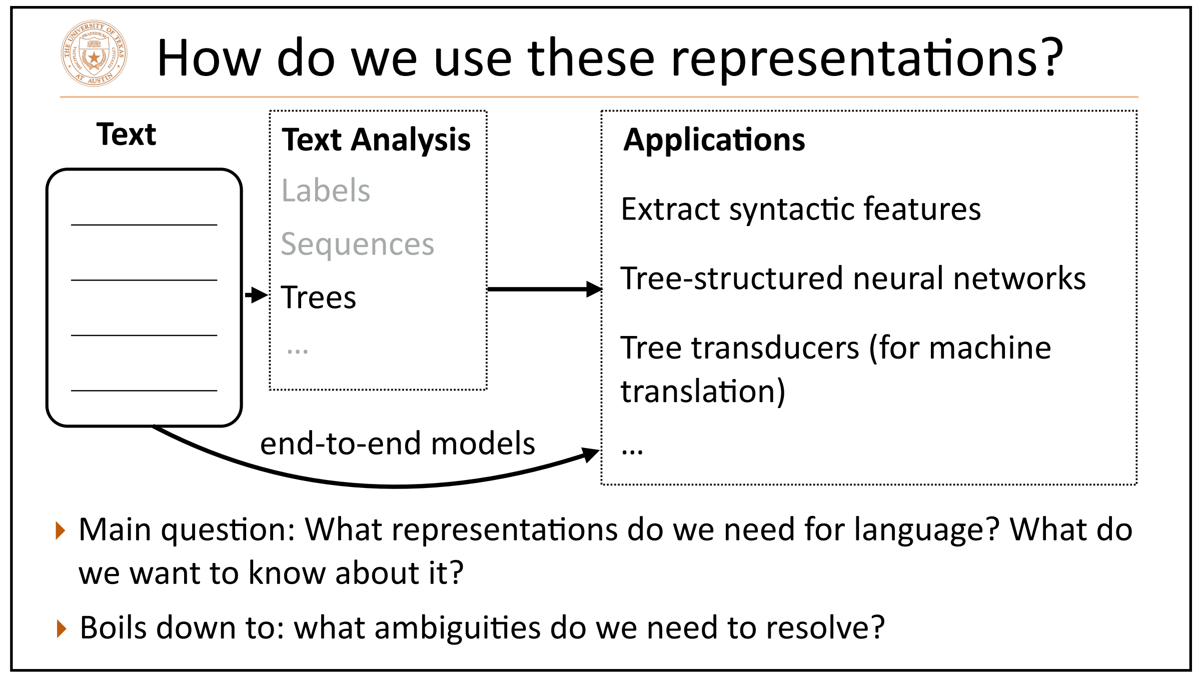
Takeaways

- RNNs can transduce inputs (produce one output for each input) or compress the whole input into a vector

- Useful for a range of tasks with sequential input: sentiment analysis, language modeling, natural language inference, machine translation

slide credits

slides that look like this



come from

CS388 given by Greg Durrett at U Texas, Austin

and their use is gratefully acknowledged. I try to make any modifications obvious, but if there are errors on a slide, assume that I added them.