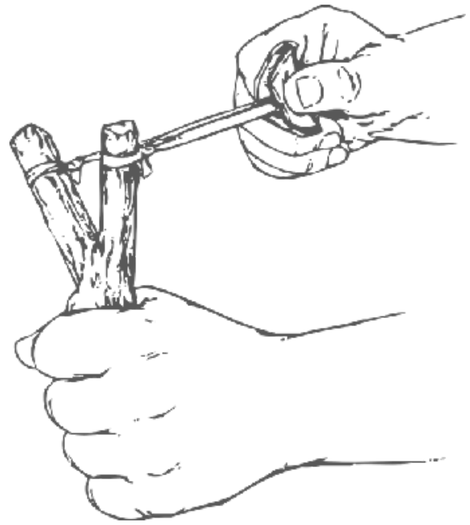
Understanding Large Language Models

Carsten Eickhoff, Michael Franke and Polina Tsvilodub

Session 02: PyTorch, Optimization, ANNs, LMs, RNNs, LSTMs

Main learning goals



1. PyTorch

- simple optmization problems with stochastic gradient descent (SGD)

2. Optimization (via Backpropagation)

- basic concepts: loss function, gradients, brackpropagation, SGD
- anatomy of update step & training loop (in PyTorch)

3. Artificial Neural Networks (specifically: Feed-Forward Networks)

- definitions of neurons & FFNs, mathematical notation in matrix-vector form
- concepts: weights & biases, activation function, hidden layers, score, prediction

4. Language Models

- definition of (autoregressive) language models, loss functions, decoding

5. Recurrent Neural Networks & LSTMs

- definitions & simple example (character-level RNN for surname generation)
- training, variations & applications

PyTorch

Key features

PyTorch

- high-level framework for ML
  - especially for artificial neural networks
- efficient tensor algebra
  - ability to run on GPUs
- pre-defined building blocks for ANNs
  - standard layers, data handling etc.
- automatic differentiation
  - enables efficient optimization

```
for i in range(n_training_steps):
    prediction = torch.distributions.Normal(loc=location, scale=1.0)
    loss = -torch.sum(prediction.log_prob(train_data))
    loss.backward()
    opt.step()
    opt.zero_grad()
```


Patzke et al. (2019) “PyTorch: An Imperative Style, High-Performance Deep Learning Library”

demo
PyTorch

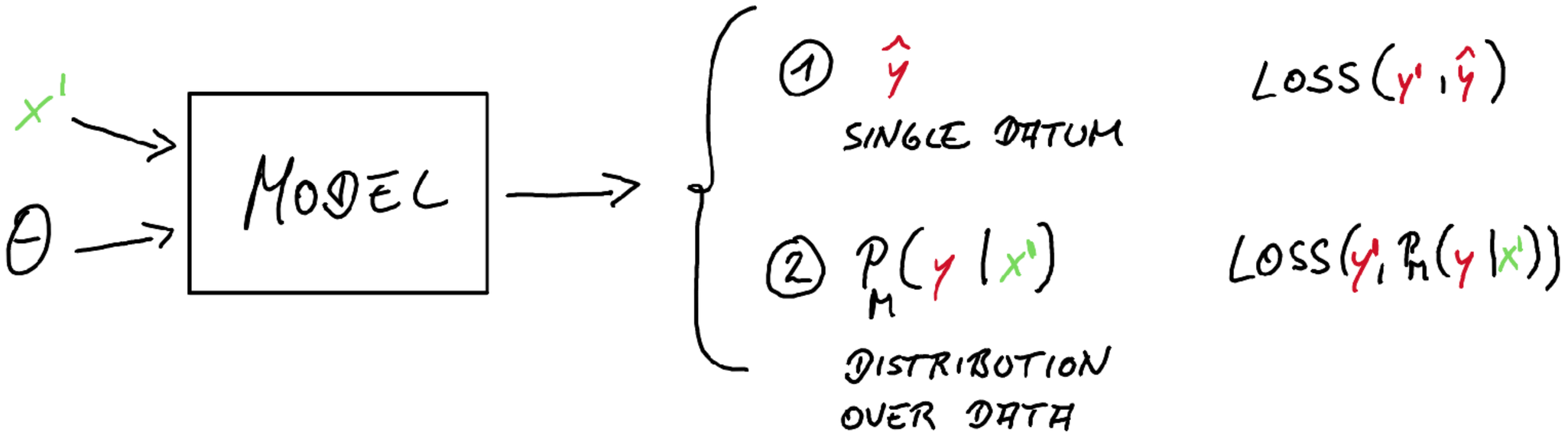
demo 01: PyTorch essentials

Optimization

Models, parameters, predictions & loss

DATA : $\langle X, Y \rangle$

MODEL : $\Theta, X \to \Delta(Y)$



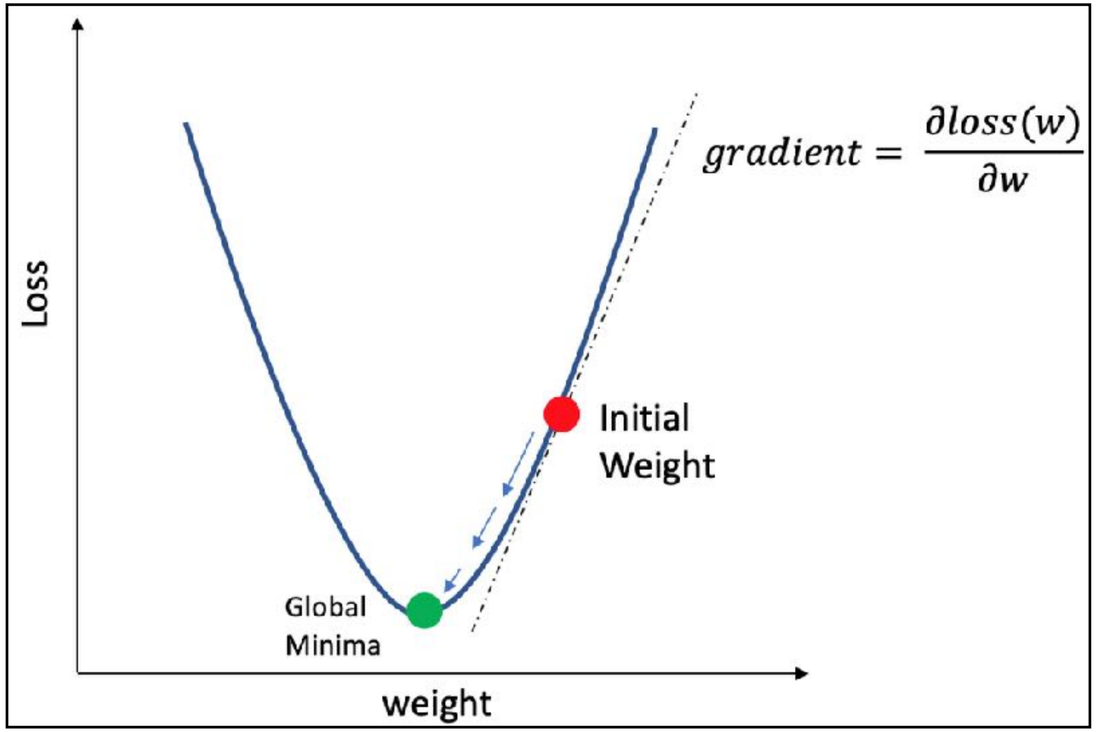
Optimization
for probabilistic models

- given:
  - data $D = \langle X, Y \rangle$
  - probabilistic model $M\colon \Theta, X \rightarrow \Delta(Y)$
  - loss function $L\colon \Theta, X, Y \rightarrow \mathbb{R}$
    - most commonly used is negative log likelihood:

$$L\left(\theta, x, y\right) = -\log P_{M(\theta)}\left(y \mid x\right)$$

- find parameters that minimize loss for data:

$$\hat{\theta} = \arg\min_{\theta \in \Theta} = \sum_{x \in X, y \in Y} L\left(\theta, x, y\right)$$



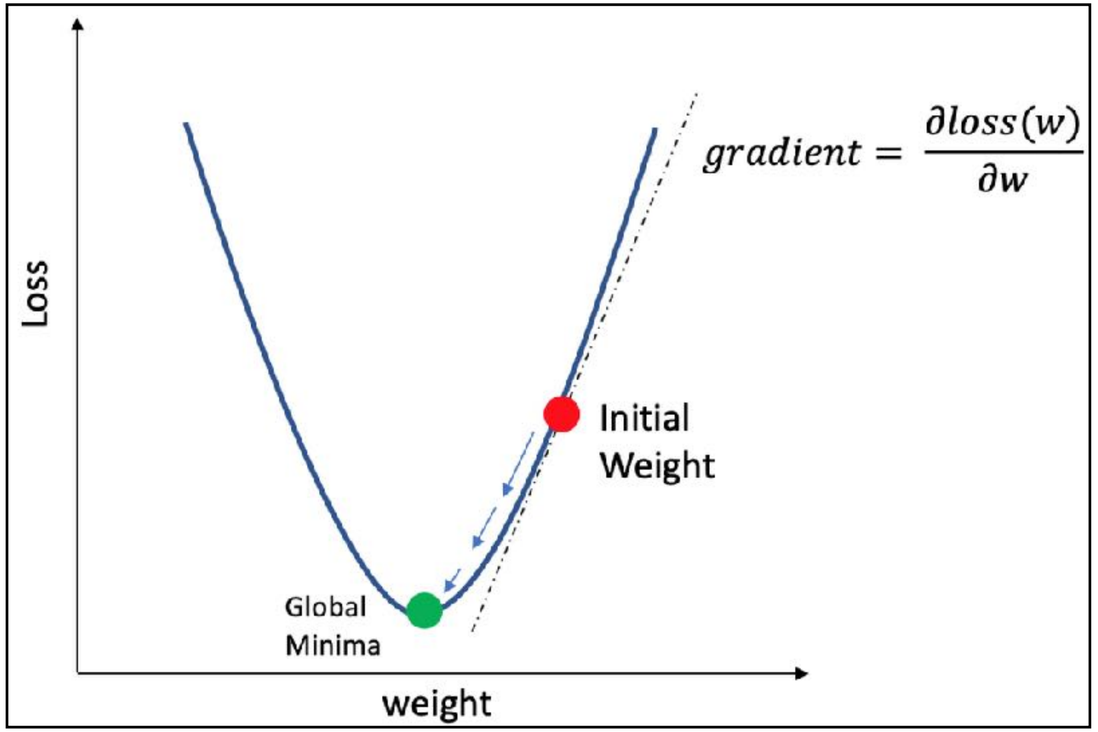
Stochastic gradient descent

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
μ (momentum), τ (dampening), *nesterov*, *maximize*

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  if $\mu \neq 0$
    if $t > 1$
      $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1-\tau)g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    if *nesterov*
      $g_t \leftarrow g_t + \mu\mathbf{b}_t$
    else
      $g_t \leftarrow \mathbf{b}_t$
  if *maximize*
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

return θ_t

from PyTorch Documentation of SGD



from this paper

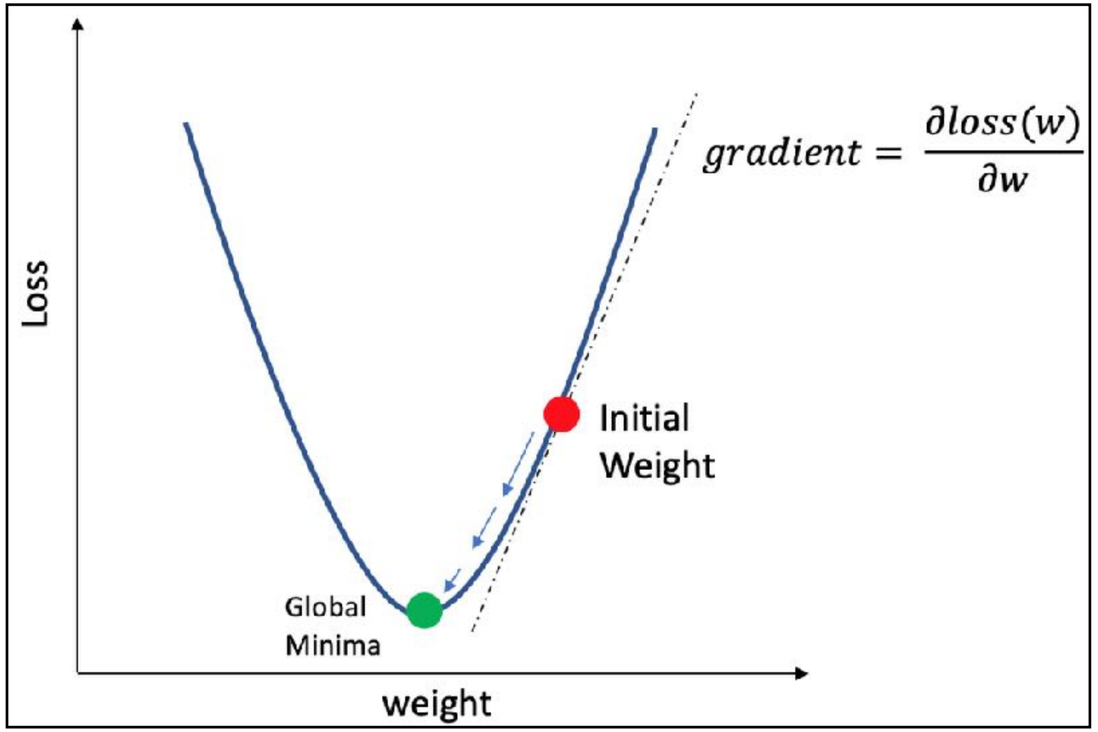
Stochastic gradient descent

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
μ (momentum), τ (dampening), *nesterov*, *maximize*

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  if $\mu \neq 0$
    if $t > 1$
      $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1-\tau)g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    if *nesterov*
      $g_t \leftarrow g_t + \mu\mathbf{b}_t$
    else
      $g_t \leftarrow \mathbf{b}_t$
  if *maximize*
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

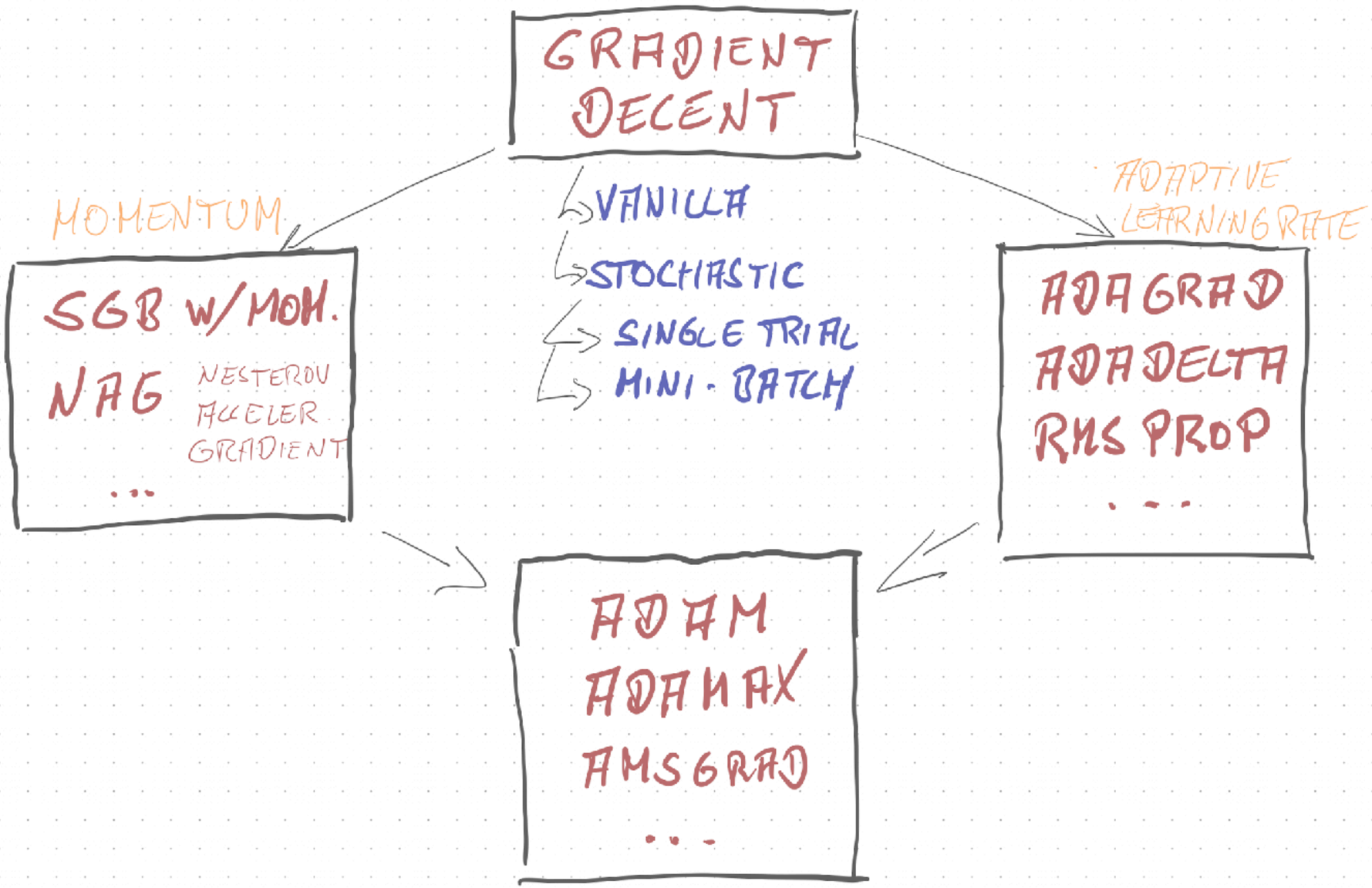
return θ_t

from PyTorch Documentation of SGD



from this paper

Common optimization algorithms



Anatomy of a training step

```
for i in range(n_training_steps):
    prediction = torch.distributions.Normal(loc=location, scale=1.0)
    loss = -torch.sum(prediction.log_prob(train_data))
    loss.backward()
    opt.step()
    opt.zero_grad()
```

1. compute predictions
   - what do we predict in the current state?
2. compute the loss
   - how good is this prediction (for the training data)?
3. backpropagate the error
   - in which direction would we need to change the relevant parameters to make the prediction better?
4. update the parameters
   - change the parameters (to a certain degree, the so-called learning rate) in the direction that should make them better
5. zero the gradients
   - reset the information about “which direction to tune” for the next training step

demo
optimization

demo 02: Maximum Likelihood Estimation (in PyTorch)

Artificial Neural Networks

Units neurons

- input vector:

  $\mathbf{x} = [x_1, \ldots, x_n]^\intercal$

- weight vector:

  $\mathbf{w} = [w_1, \ldots, w_n]^\intercal$

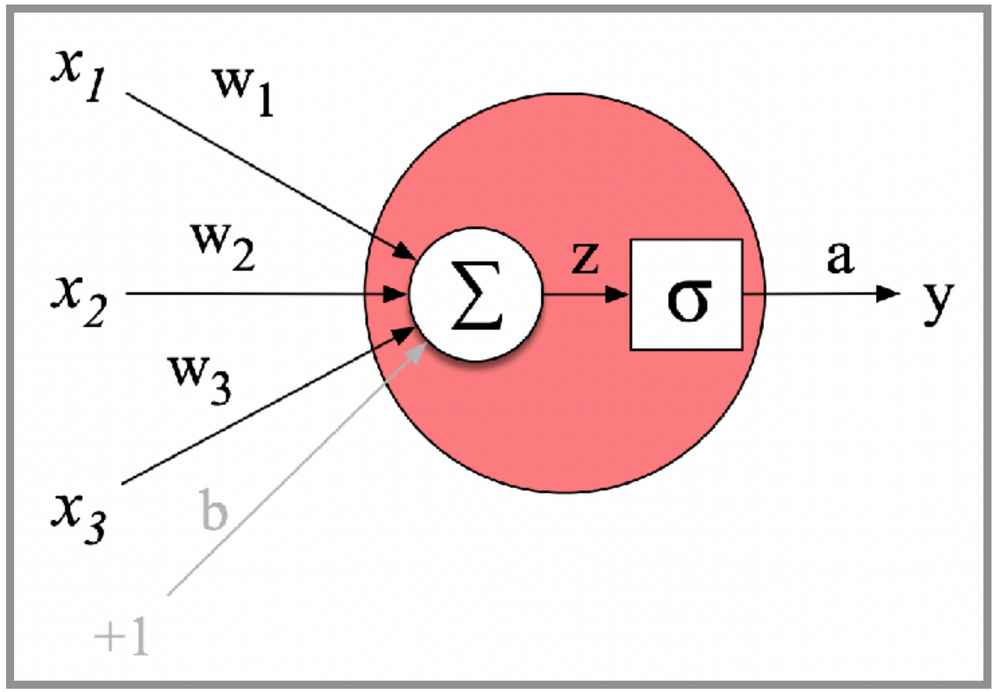
- bias:

  b

- score:

  $$z = b + \sum_{j=1}^{n} w_j x_j = b + \mathbf{w} \cdot \mathbf{x}$$

- activation level:
  - $a = f(z)$, where f is the **activation function**



Common activation functions

- perceptron:

$$f(z) = \delta_{z>0}$$

- sigmoid:

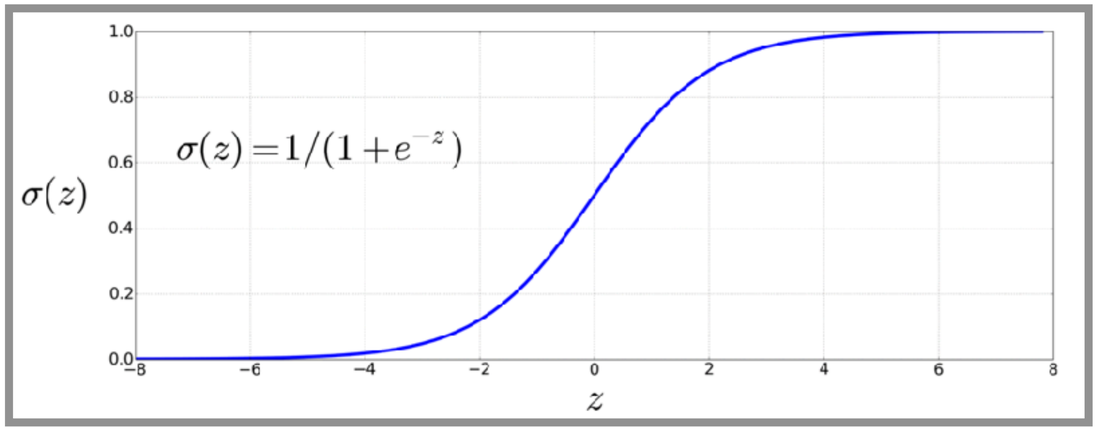
$$f(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

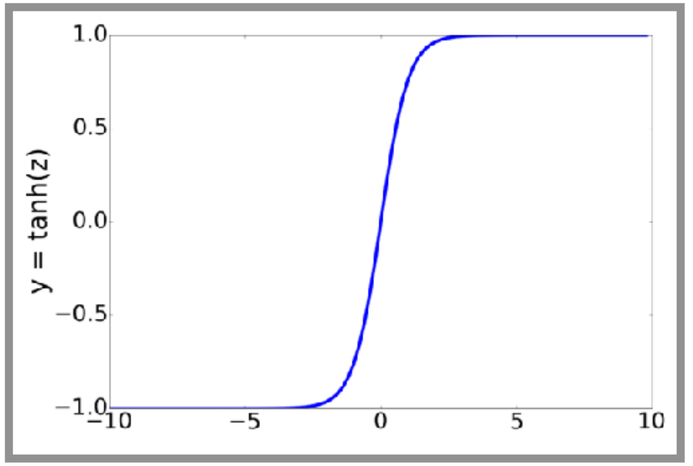
- hyperbolic tangent:

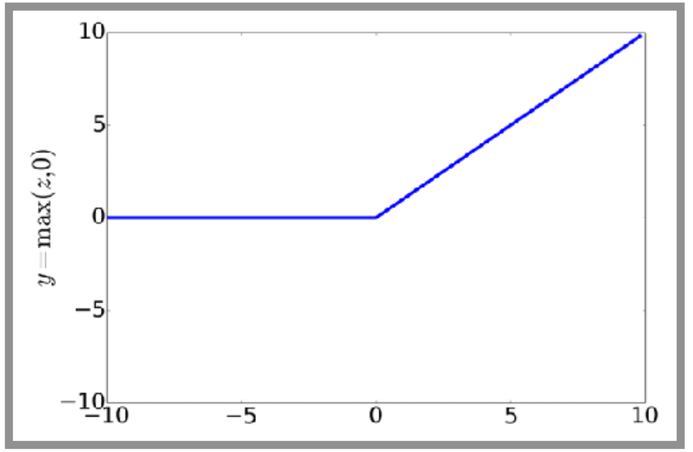
$$f(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

- rectified linear unit:

$$f(z) = \mathrm{ReLU}(z) = \max(z,0)$$







Matrix multiplication

recap

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

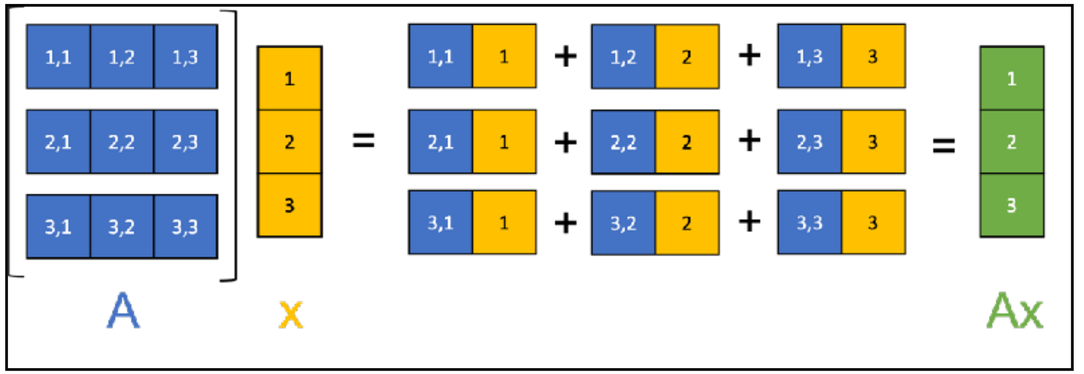
2 x 4 4 x 3 2 x 3

$$c_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

Matrix-vector multiplication

recap



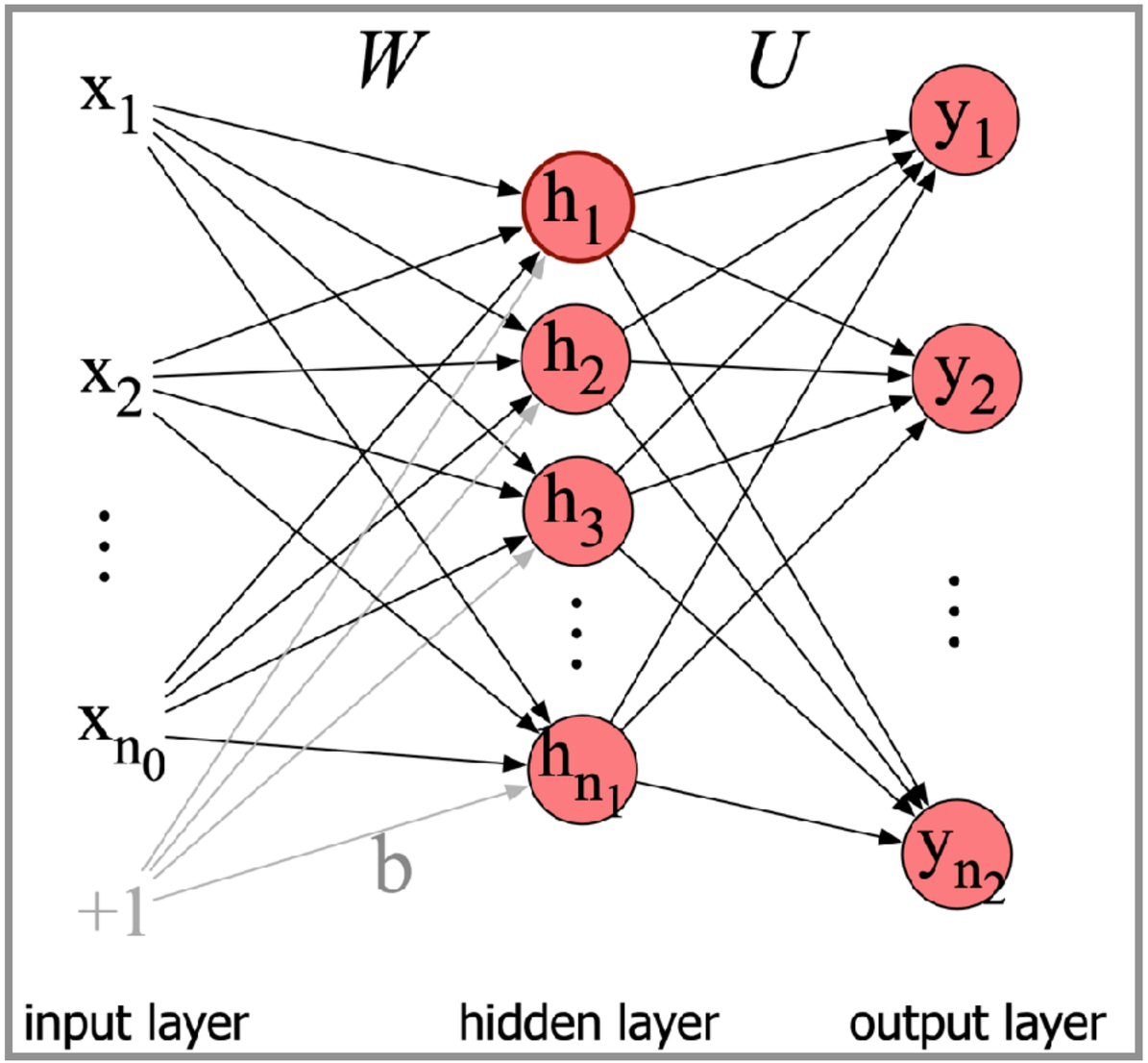
- think of matrix **A** with dimensions (n, m) as a **linear mapping** $f_{\mathbf{A}} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ from vectors of length m to vectors for length n, so that with $\mathbf{x} = [x_1, \ldots, x_m]$:

$$f_{\mathbf{A}}(\mathbf{x}) = \mathbf{Ax}$$

Feed-forward neural network

one hidden layer

- input:
  $\mathbf{x} = [x_1, \ldots, x_{n_x}]^T$
- weight matrix:
  $\mathbf{W} \in \mathbb{R}^{n_k \times n_x}$
- bias vector:
  $\mathbf{b} = [b_1, \ldots, b_{n_k}]^T$
- activation vector hidden layer:
  $\mathbf{h} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$, with $f \in \{\sigma, \tanh, \ldots\}$
- weight matrix:
  $\mathbf{U} \in \mathbb{R}^{n_y \times n_k}$
- prediction vector:
  $\mathbf{y} = g(\mathbf{U}\mathbf{h})$, with $g \in \{\sigma, \text{soft-max}, \ldots\}$



Feed-forward neural network

multi-layer perceptron

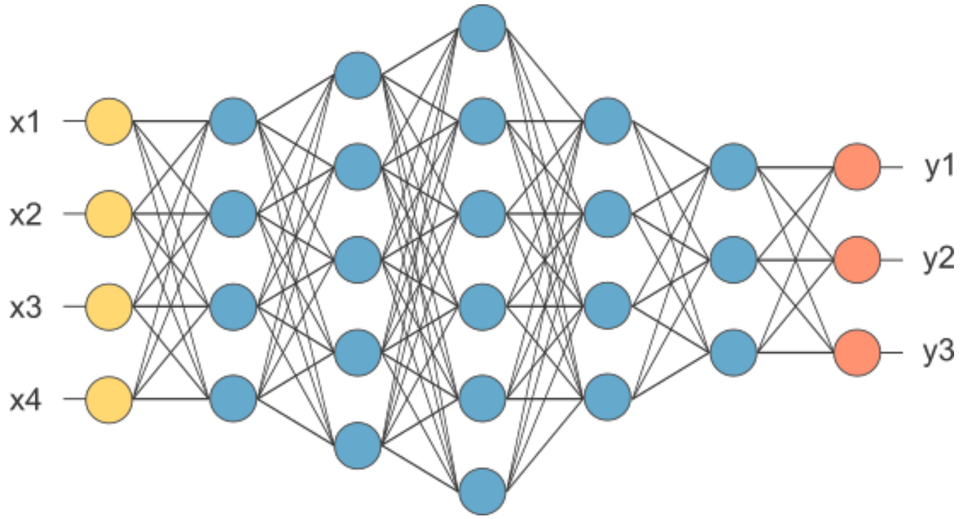
- anchoring in input:

$$\mathbf{a}^{[0]} = \mathbf{x} = [x_1, \ldots, x_{n_x}]^T$$

- activation at layer n:

$$\mathbf{a}^{[n]} = f^{[n]}\left(\mathbf{W}^{[n]}\mathbf{a}^{[n-1]} + \mathbf{b}^{[n]}\right)$$

  - with $f^{[n]} \in \{\sigma, \text{tanh}, \ldots\}$ if n is a hidden layer, or
  - with $f^{[n]} \in \{\sigma, \text{soft-max}, \ldots\}$ if n is the output layer



demo
MLP

demo 03: Multi-Layer Perceptron for Non-Linear Regression

Language Models

Language model

$$LM \in \Delta(S)$$

where S is a set of finite sequences of discrete units

arguably much better terminology: (finite, discrete) sequence model

Language model
high-level definition

- let $\mathscr{V}$ be a (finite) **vocabulary**, a set of tokens
  - tokens can be characters, sub-words, phrases, …
- let $w_{1:n} = \langle w_1, \ldots, w_n \rangle$ be a finite sequence of tokens
  - it is still common to use w (reminiscent of "words") for tokens
- let S be a the set of all (finite) sequences of tokens
- let X be a set of input conditions
  - e.g., images, text in a different language …
- a **language model** LM is function that assigns to each input X a probability distribution over S, given parameters $\theta \in \Theta$:

$$LM_\theta \; : \; X \mapsto \Delta(S)$$

  - if there is only one input in set X, the LM is just a probability distribution over all sequences of words
  - LMs originally meant to capture the true occurrence frequency
  - a **neural language model** is an LM realized as a neural network
  - in the following we skip the dependence on X

Language model

left-to-right / autoregressive / causal model

- an **autoregressive language model** is defined as a **function** that maps an initial token sequence to a **next-token distribution**:

  $$LM : w_{1:n} \mapsto \Delta(\mathcal{V})$$

  - we write $P_{LM}(w_{n+1} \mid w_{1:n})$ for the **next-token probability**
  - the **surprisal** of w_{n+1} after sequence $w_{1:n}$ is $-\log\left(P_{LM}(w_{n+1} \mid w_{1:n})\right)$
- the **sequence probability** follows from the chain rule:

  $$P_{LM}(w_{1:n}) = \prod_{i=1}^{n} P_{LM}(w_i \mid w_{1:i-1})$$
- measures of **goodness of fit** for observed sequence $w_{1:n}$:
  - **perplexity**:

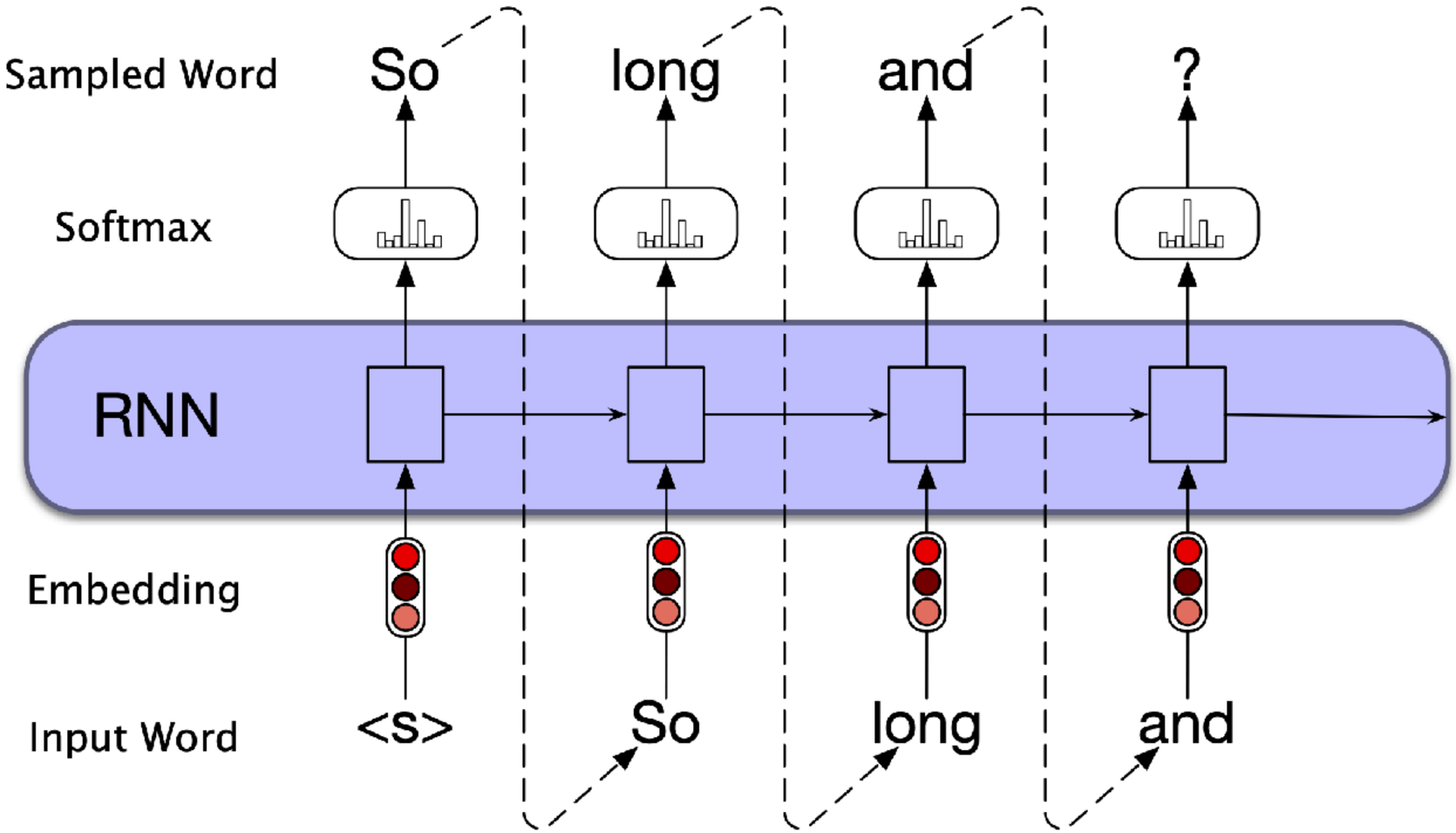
    $$\mathrm{PP}_{LM}(w_{1:n}) = P_{LM}(w_{1:n})^{-\frac{1}{n}}$$
  - **average surprisal**:

    $$\text{Avg-Surprisal}_{LM}(w_{1:n}) = -\frac{1}{n}\log P_{LM}(w_{1:n})$$

$$\log \mathrm{PP}_{M}(w_{1:n}) = \text{Avg-Surprisal}_{M}(w_{1:n})$$

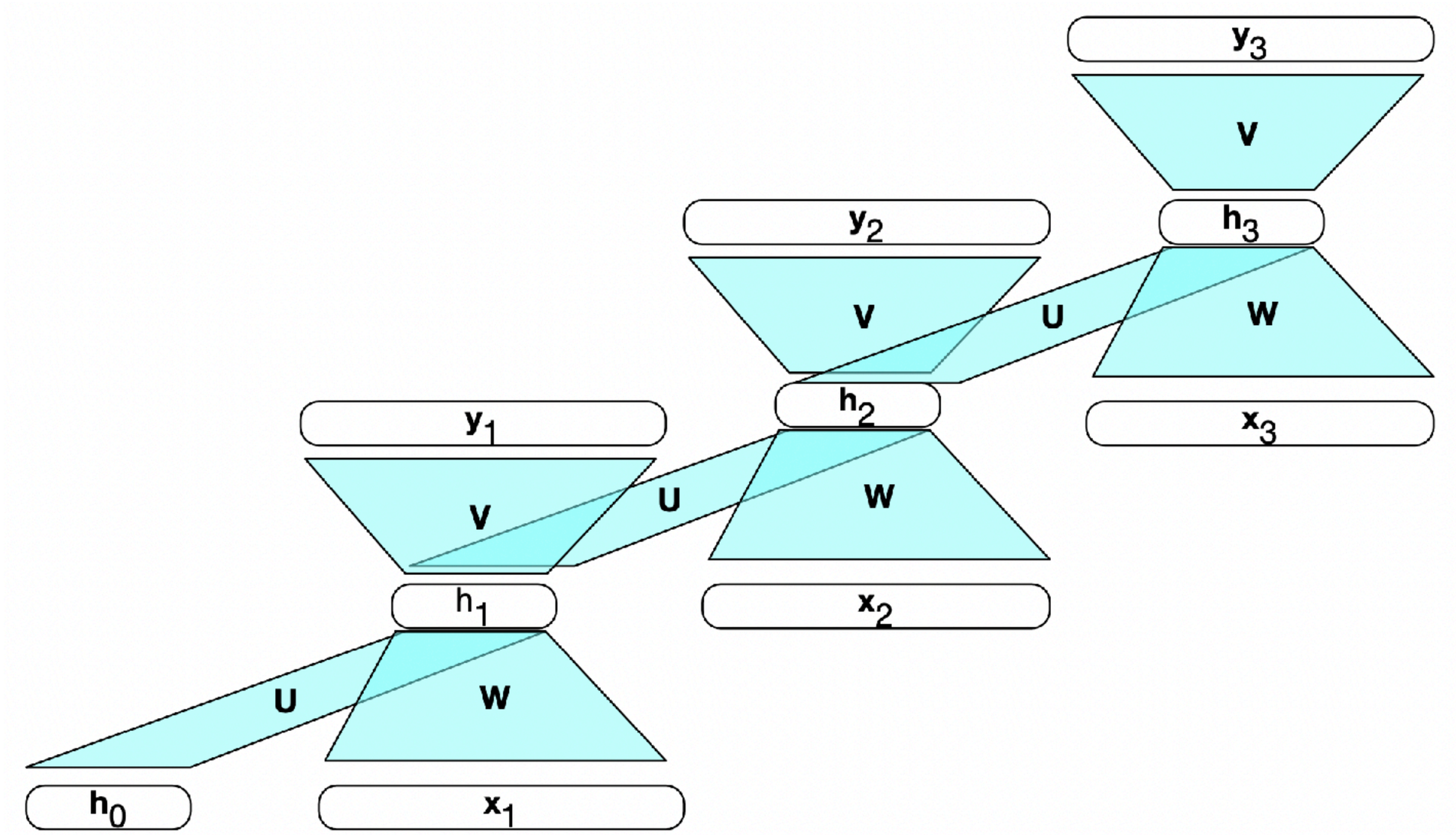
Autoregressive generation

left-to-right / causal model



Recurrent Neural Networks

Recurrent neural networks

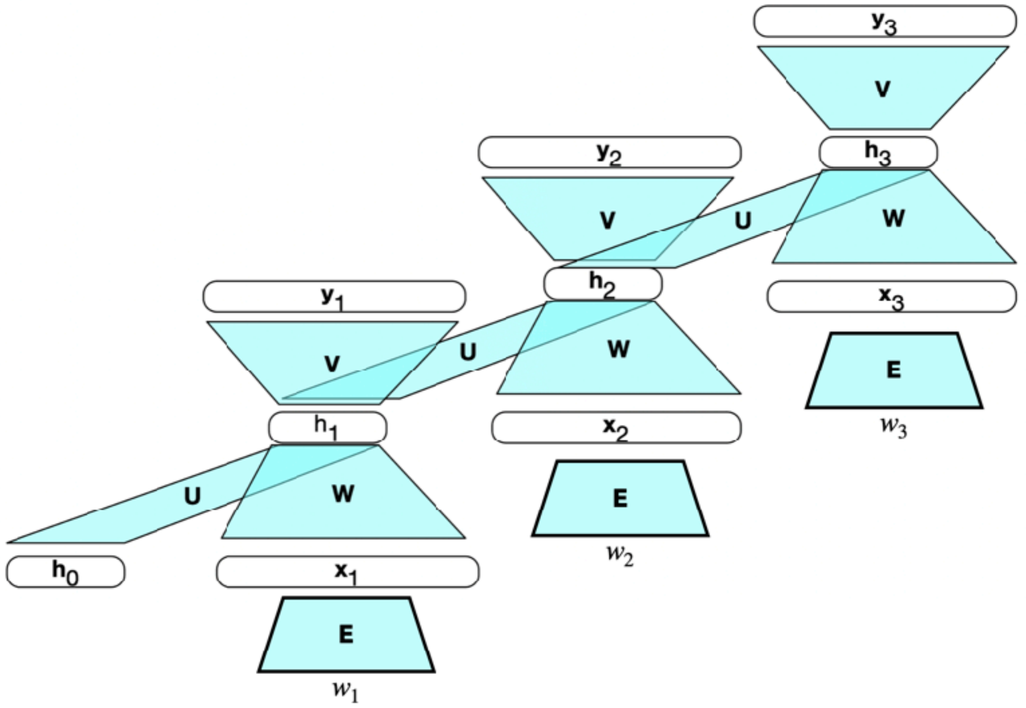


RNN-based language model

one of many similar architectures

- dimensions:
  - n_V : # of tokens in vocabulary
  - n_h : # units in hidden layer
  - n_x : length of input $\mathbf{x}$ (token embedding)
- what is what?
  - $\mathbf{w}_t \in \mathbb{R}^{n_V}$: one-hot vector representing token $\mathbf{w}_t$
  - $\mathbf{x}_t \in \mathbb{R}^{n_x}$: word embedding of token $\mathbf{w}_t$
  - $\mathbf{h}_t \in \mathbb{R}^{n_h}$: hidden layer activation at time t (with $\mathbf{h}_0 = 0$)
  - $\mathbf{y}_t \in \Delta(\mathscr{V})$: probability distribution over tokens
  - $f \in \{\sigma, \texttt{tanh}, \dots\}$: activation function (as usual)
  - $\mathbf{U} \in \mathbb{R}^{n_h \times n_h}$: mapping hidden-to-hidden
  - $\mathbf{V} \in \mathbb{R}^{n_V \times n_h}$: mapping hidden-to-word
  - $\mathbf{E} \in \mathbb{R}^{n_x \times n_V}$: mapping word-to-embedding
  - $\mathbf{W} \in \mathbb{R}^{n_h \times n_x}$: mapping embedding-to-hidden

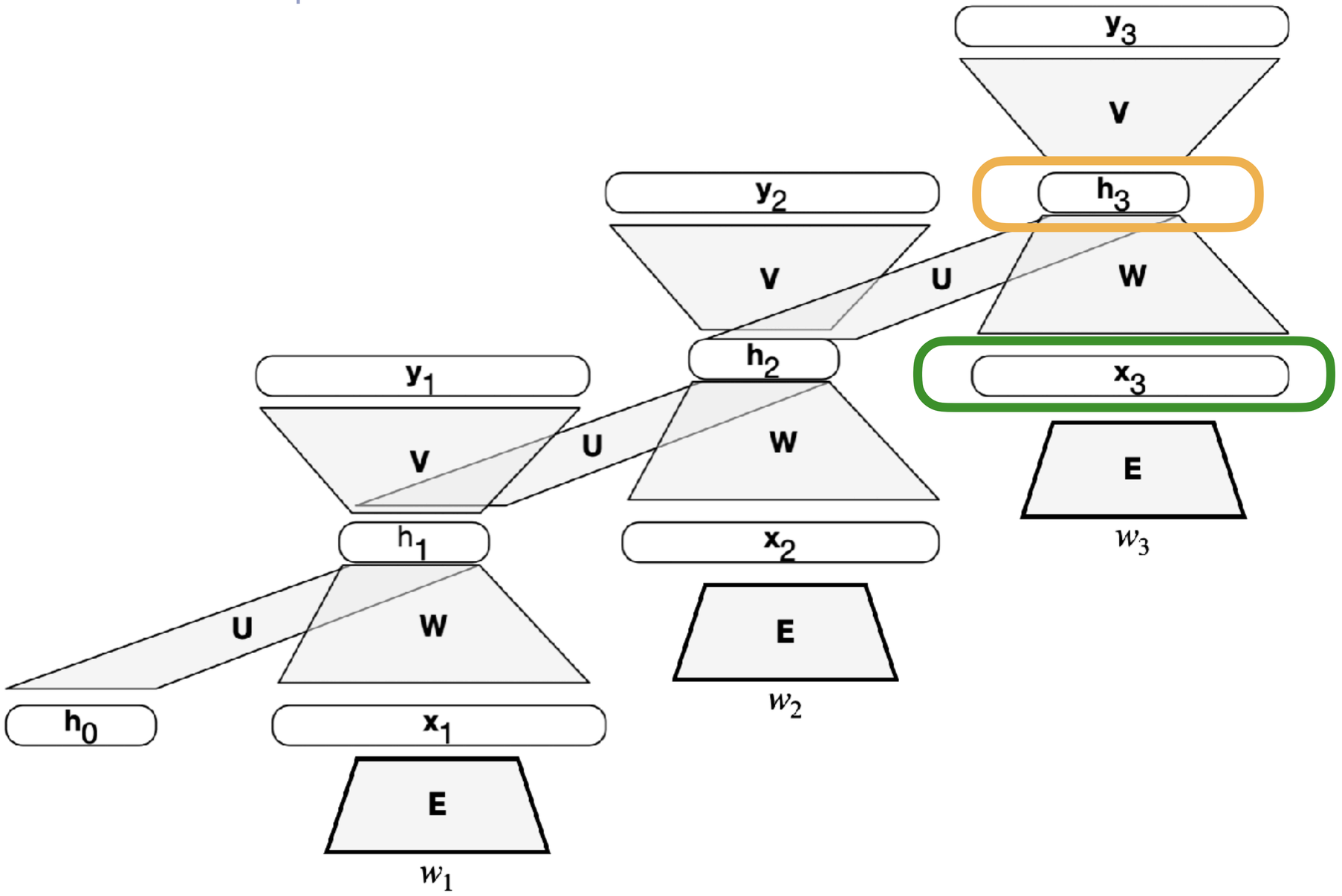
- definition (forward pass):
  - $\mathbf{x}_t = \mathbf{E}\mathbf{w}_t$
  - $\mathbf{h}_t = f\left[\mathbf{U}\mathbf{h}_t + \mathbf{W}\mathbf{x}_t\right]$
  - $\mathbf{y}_t = \texttt{softmax}(\mathbf{V}\mathbf{h}_t)$



based on Jurafsky & Martin “NLP”

Embeddings

for words and sequences



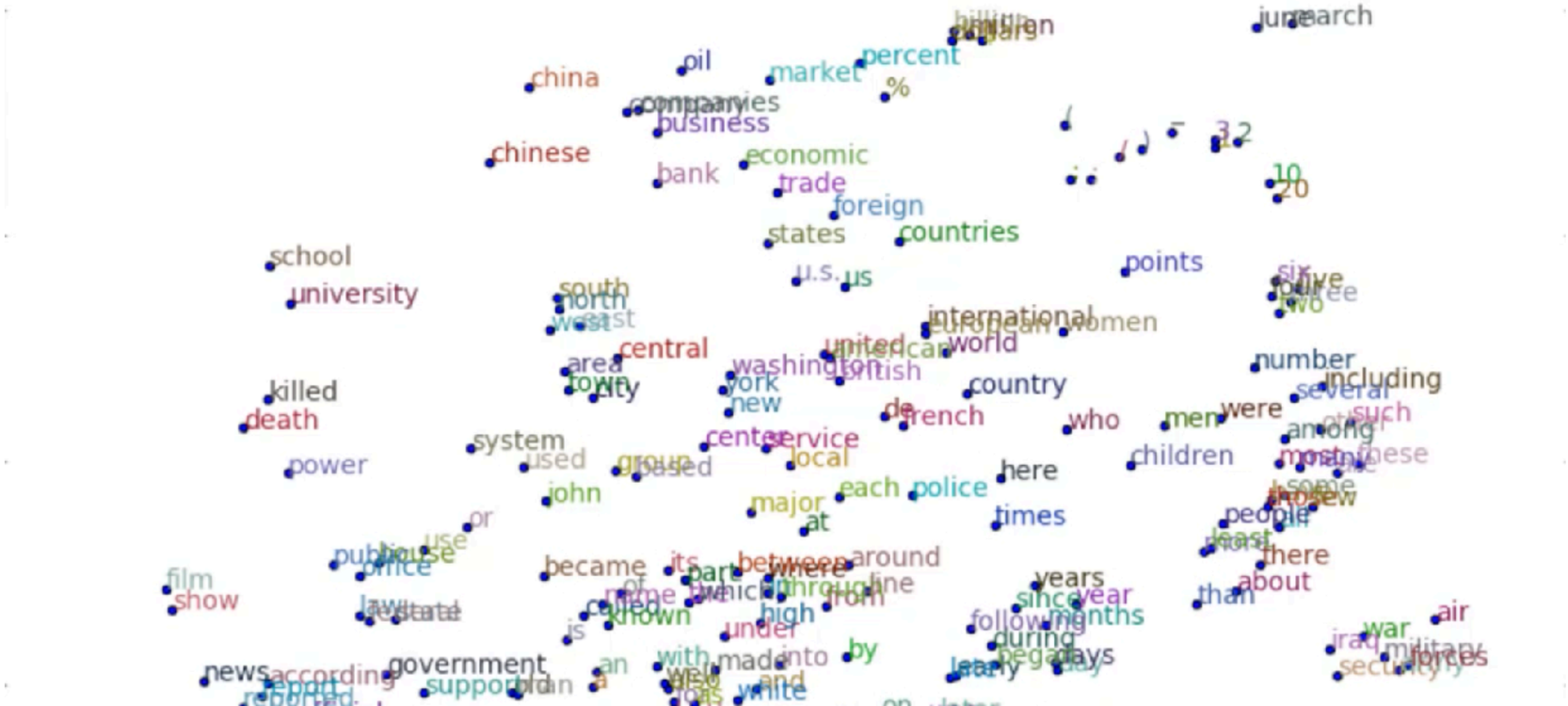
sequence embedding for w_1, w_2, w_3

word embedding for w_3

Aside: Embeddings

Distributed Representations: dense vectors (aka embeddings) that aim to convey information about tokens (e.g., aspects of the meanings of words):

- "word embeddings" refer to when you have type-based representations
- "contextualized embeddings" refer to when you have token-based representations

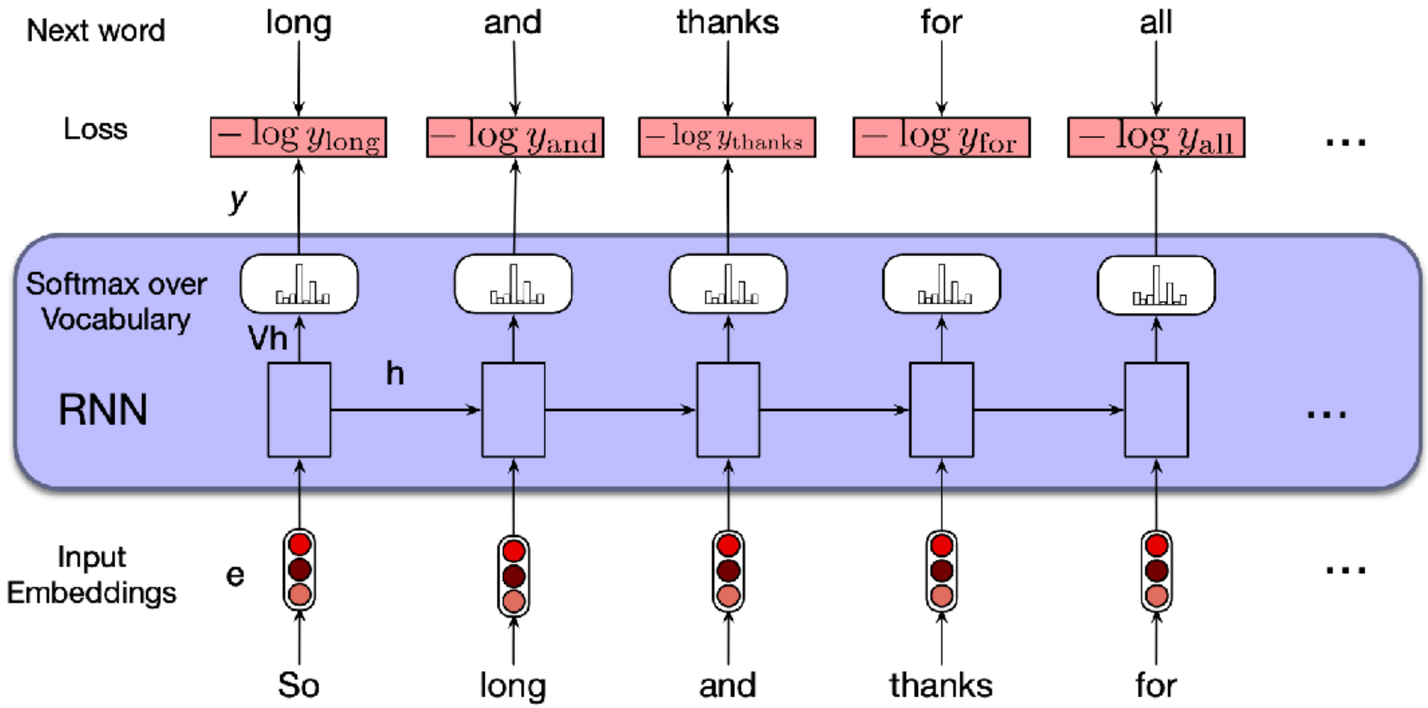


Training RNNs

Training RNNs

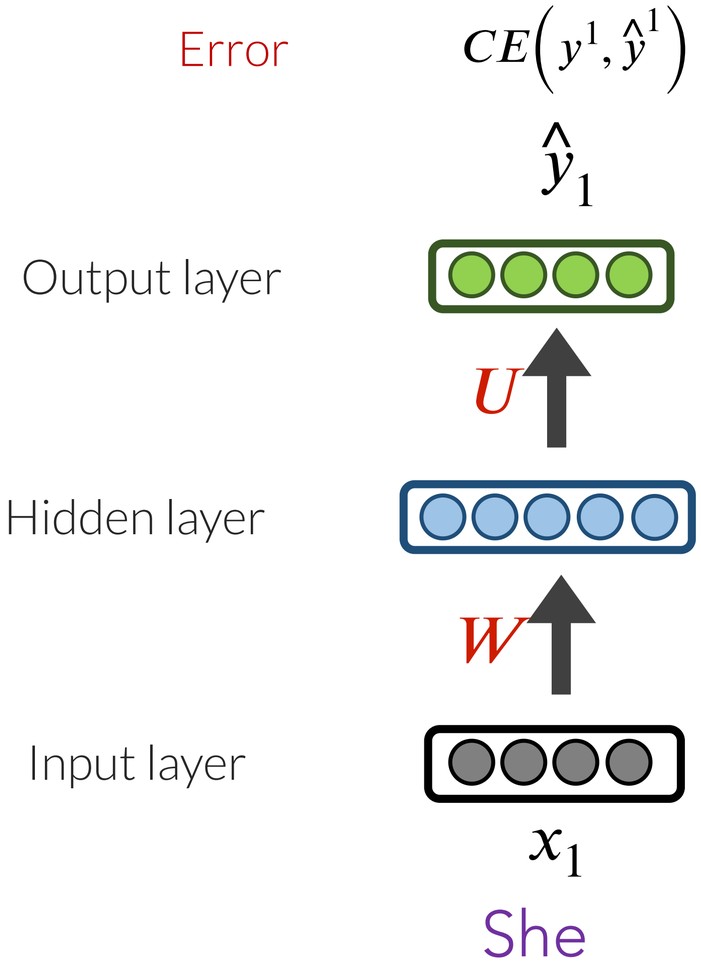
using teacher forcing & next-word surprisal (CE loss)

- teacher forcing
  - predict each next token given the preceding input (not the model-generated sequence)
- next-word surprisal (CE loss)
  - loss function is (average) next-token surprisal



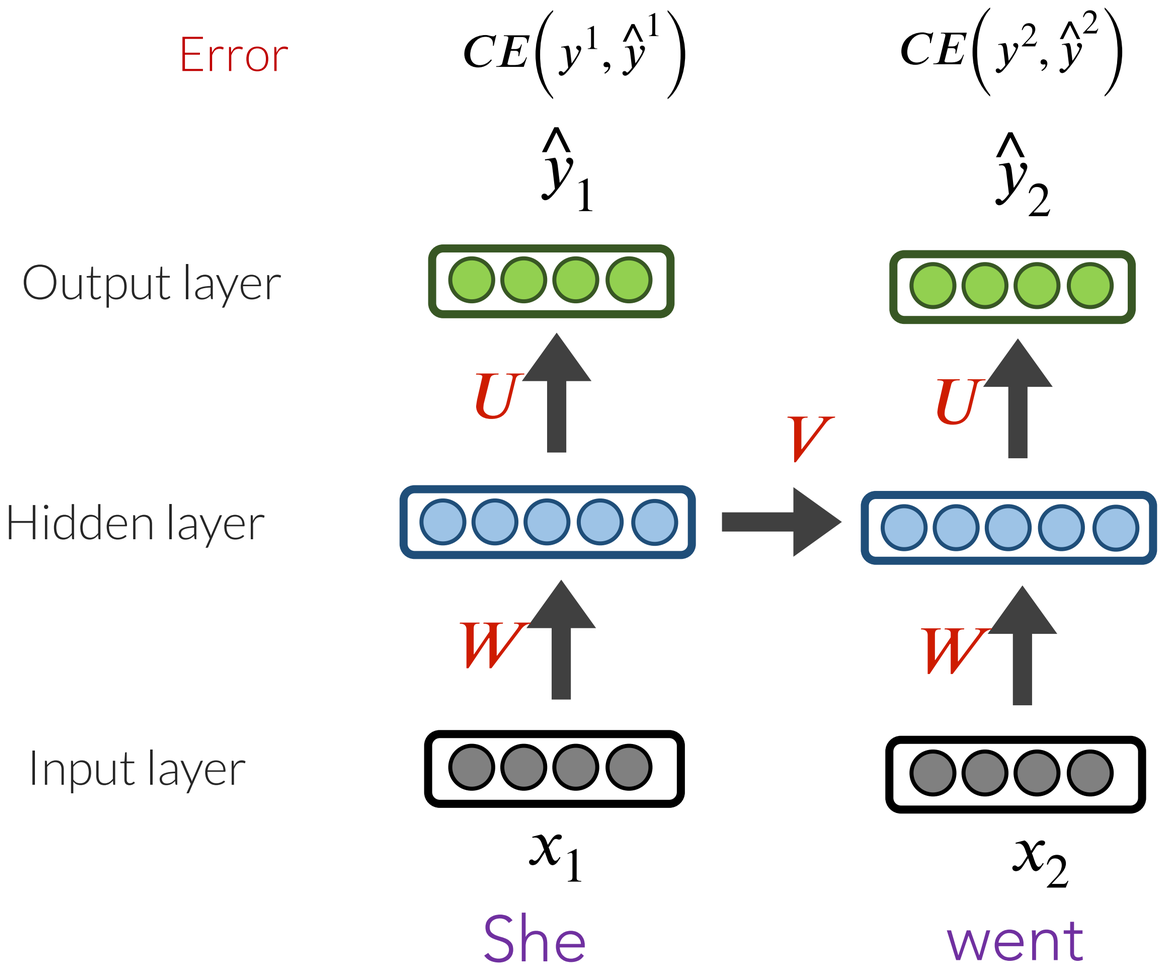
Training Process

$$CE\left(y^i, \hat{y}^i\right) = -\sum_{w \in V} y_w^i \log(\hat{y}_w^i)$$



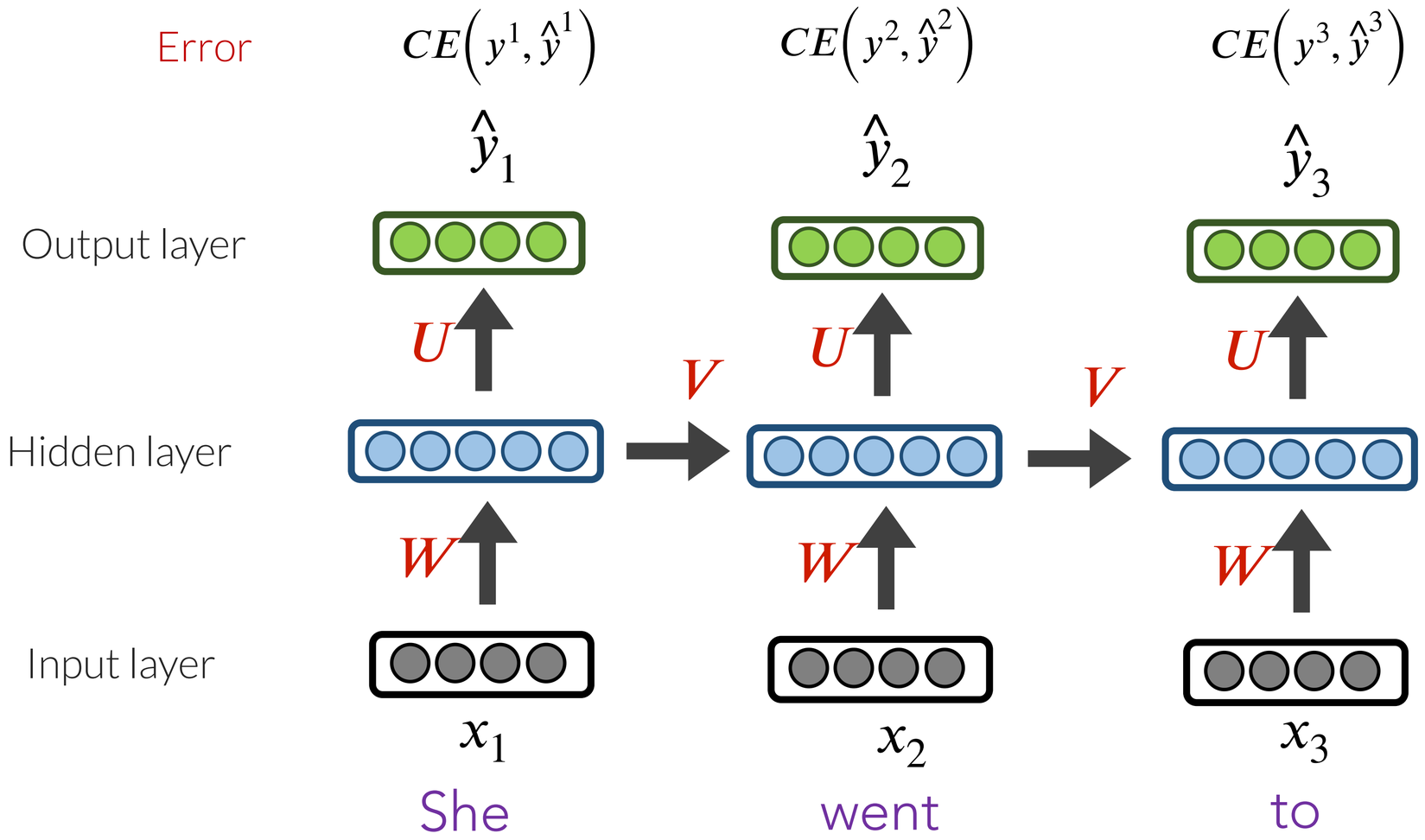
Training Process

$$CE\left(y^i, \hat{y}^i\right) = -\sum_{w \in V} y_w^i \log(\hat{y}_w^i)$$



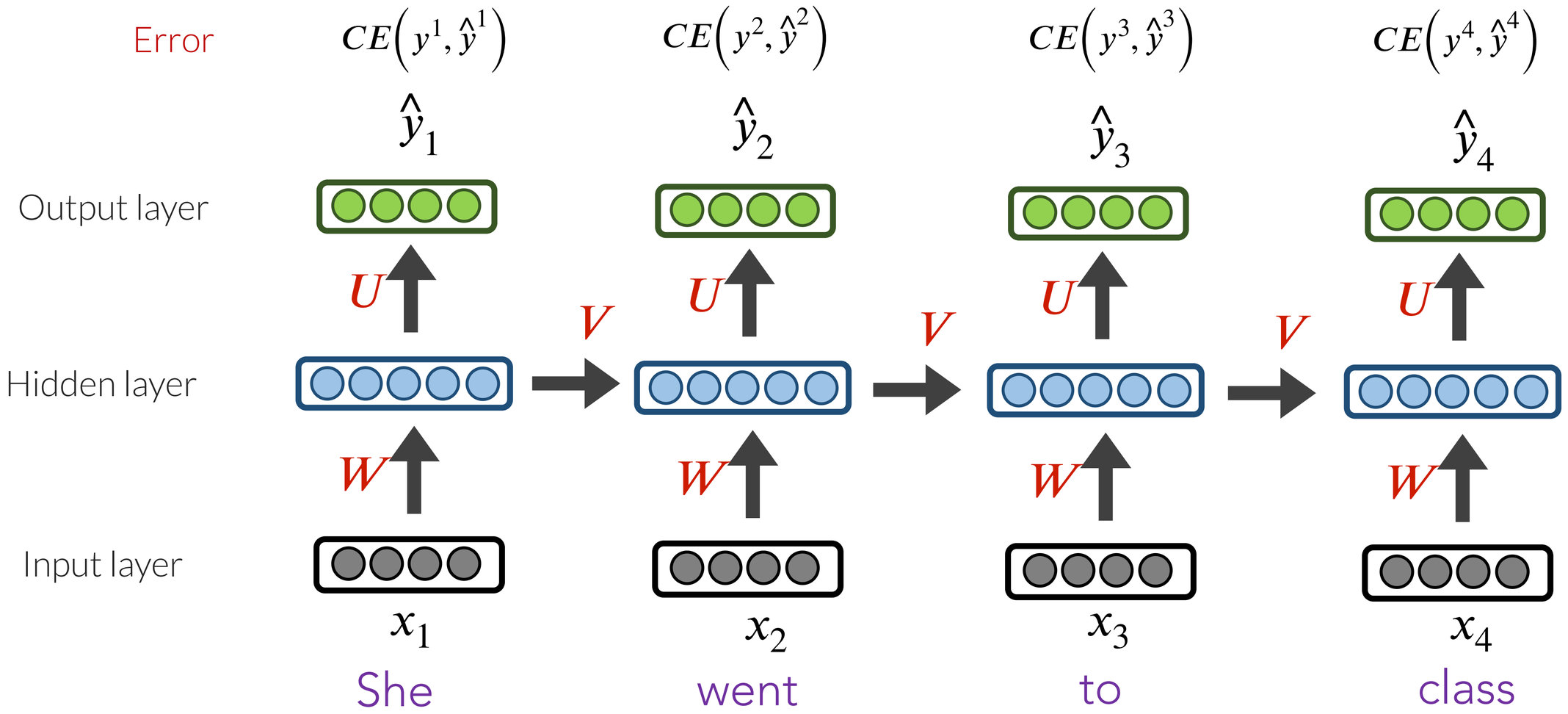
Training Process

$$CE\left(y^i, \hat{y}^i\right) = -\sum_{w \in V} y_w^i \log(\hat{y}_w^i)$$



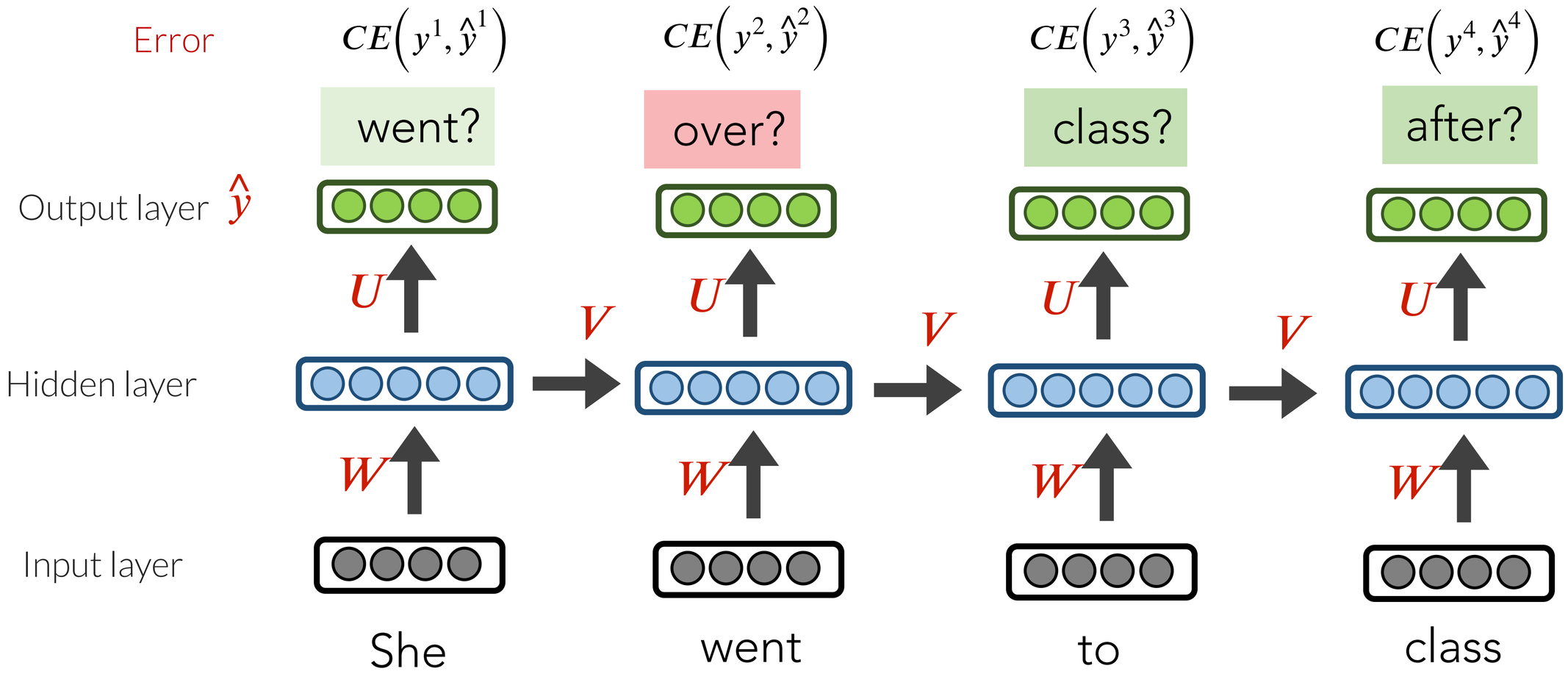
Training Process

$$CE\left(y^i, \hat{y}^i\right) = -\sum_{w \in V} y_w^i \log(\hat{y}_w^i)$$

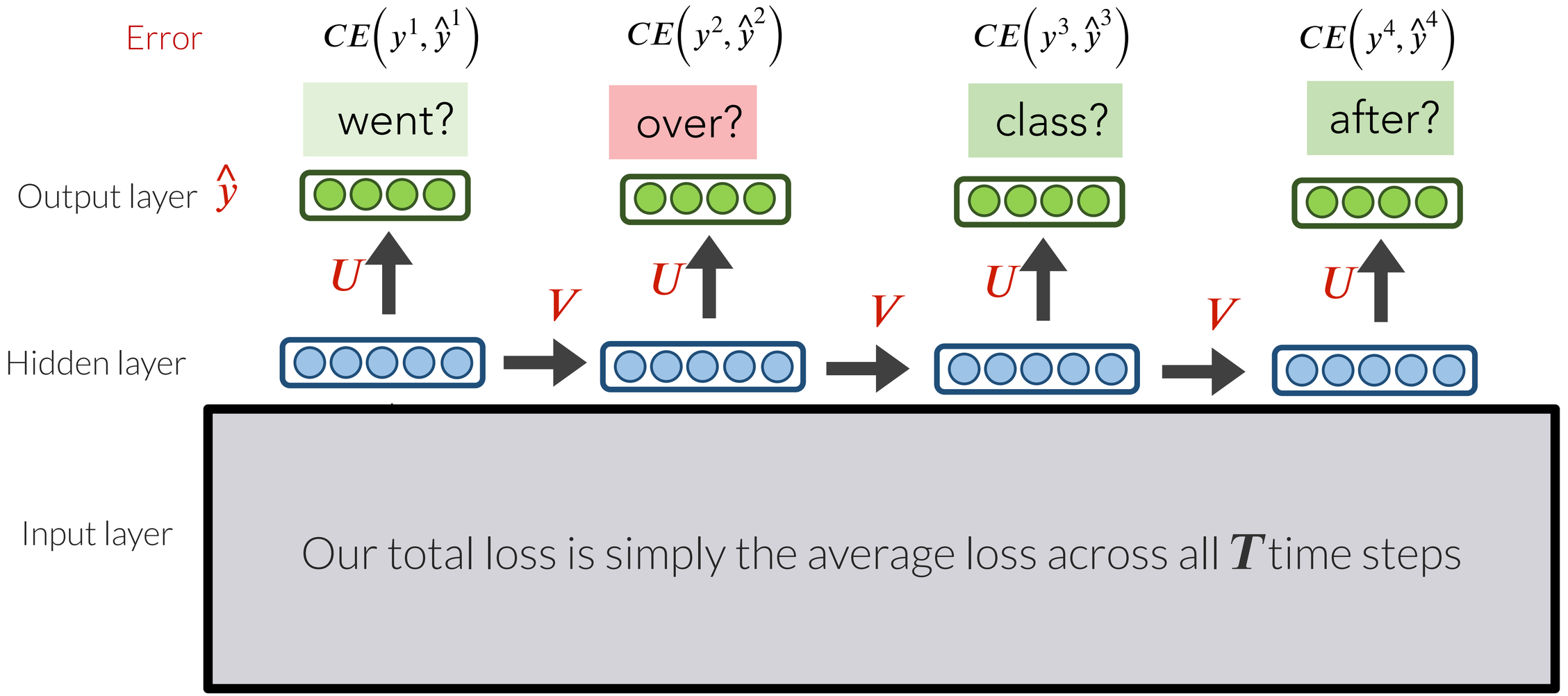


Training Process

$$CE\left(y^i, \hat{y}^i\right) = -\sum_{w \in V} y_w^i \log(\hat{y}_w^i)$$



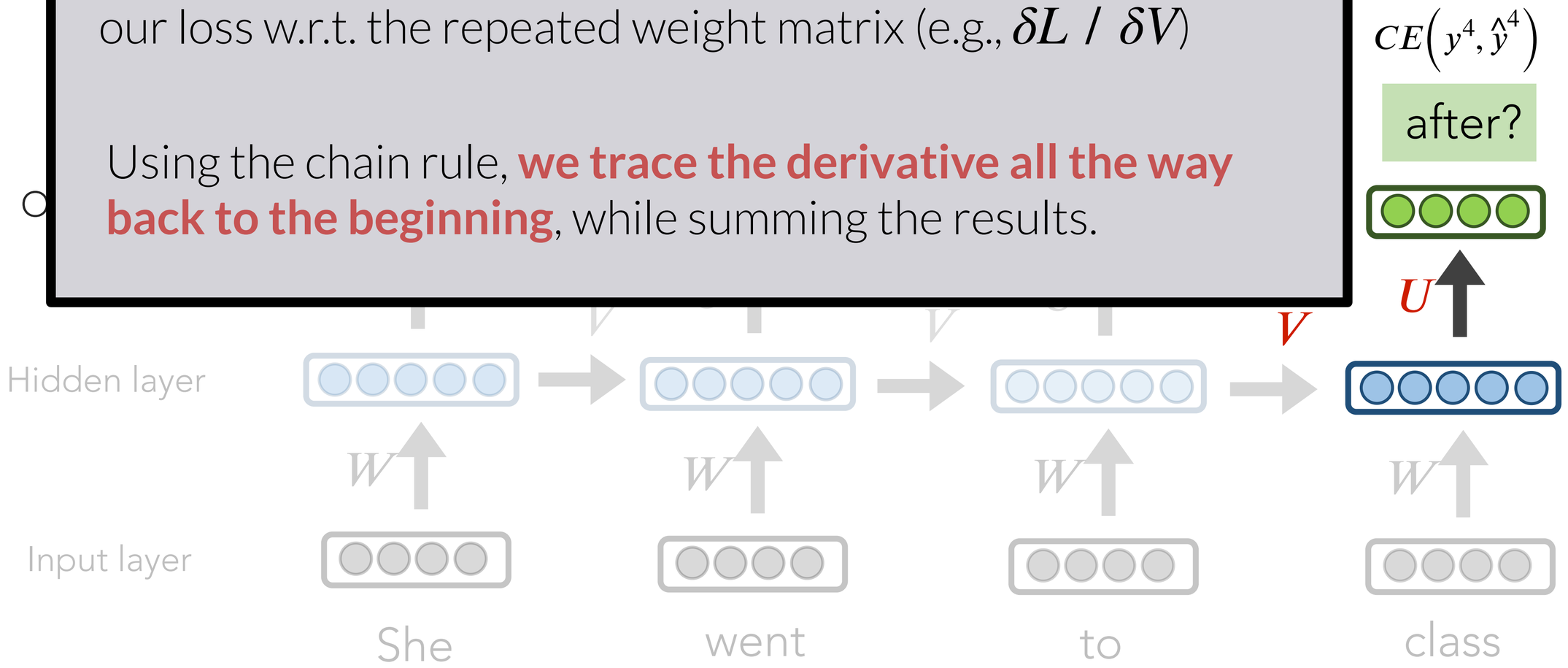
Training Process



Backpropagation through time

To update our weights (e.g. V), we calculate the gradient of our loss w.r.t. the repeated weight matrix (e.g., $\delta L \ / \ \delta V$)

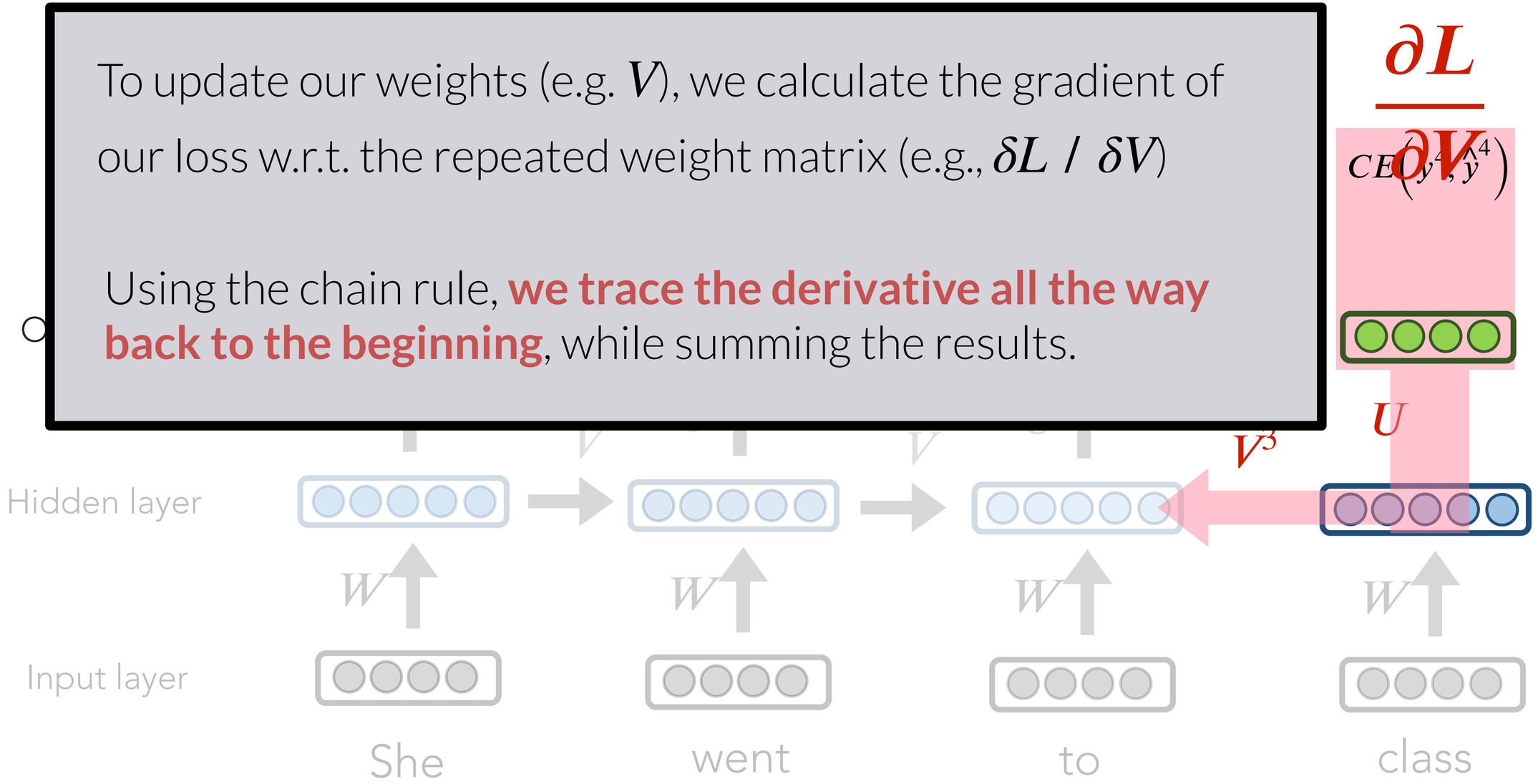
Using the chain rule, **we trace the derivative all the way back to the beginning**, while summing the results.



Backpropagation through time

To update our weights (e.g. V), we calculate the gradient of our loss w.r.t. the repeated weight matrix (e.g., $\delta L \ / \ \delta V$)

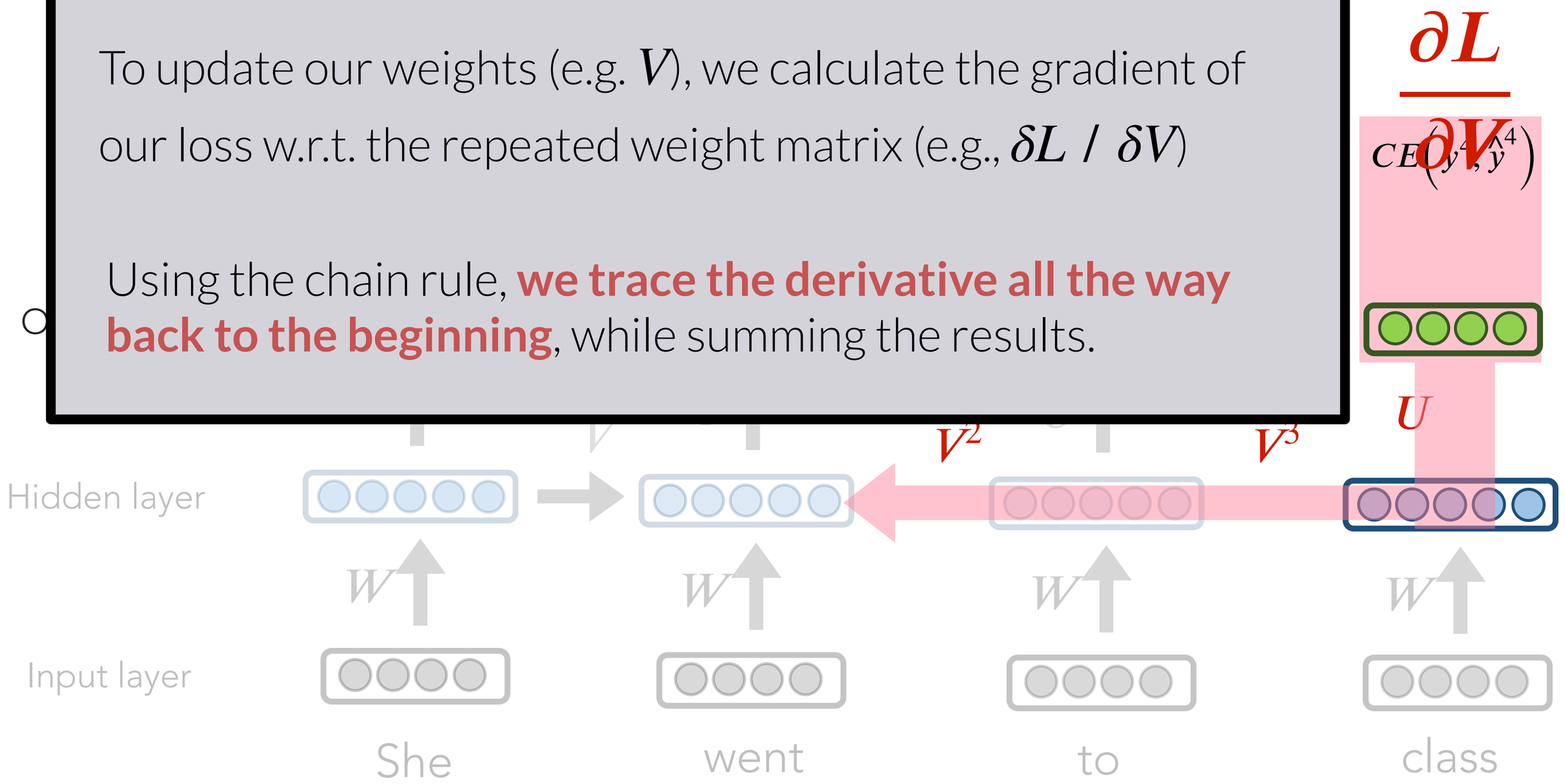
Using the chain rule, **we trace the derivative all the way back to the beginning**, while summing the results.



Backpropagation through time

To update our weights (e.g. V), we calculate the gradient of our loss w.r.t. the repeated weight matrix (e.g., $\delta L \ / \ \delta V$)

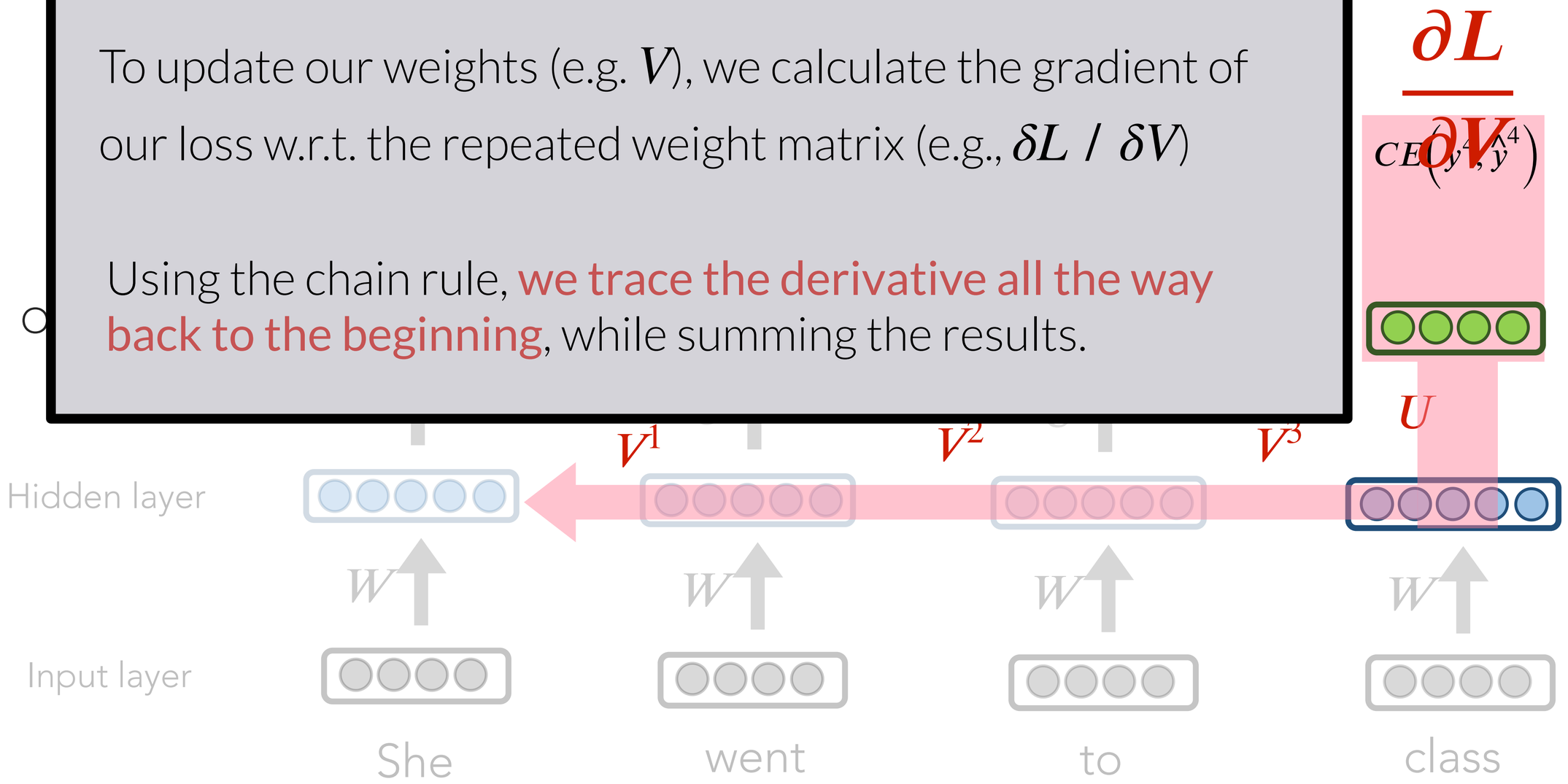
Using the chain rule, **we trace the derivative all the way back to the beginning**, while summing the results.



Backpropagation through time

To update our weights (e.g. V), we calculate the gradient of our loss w.r.t. the repeated weight matrix (e.g., $\delta L \ / \ \delta V$)

Using the chain rule, we trace the derivative all the way back to the beginning, while summing the results.

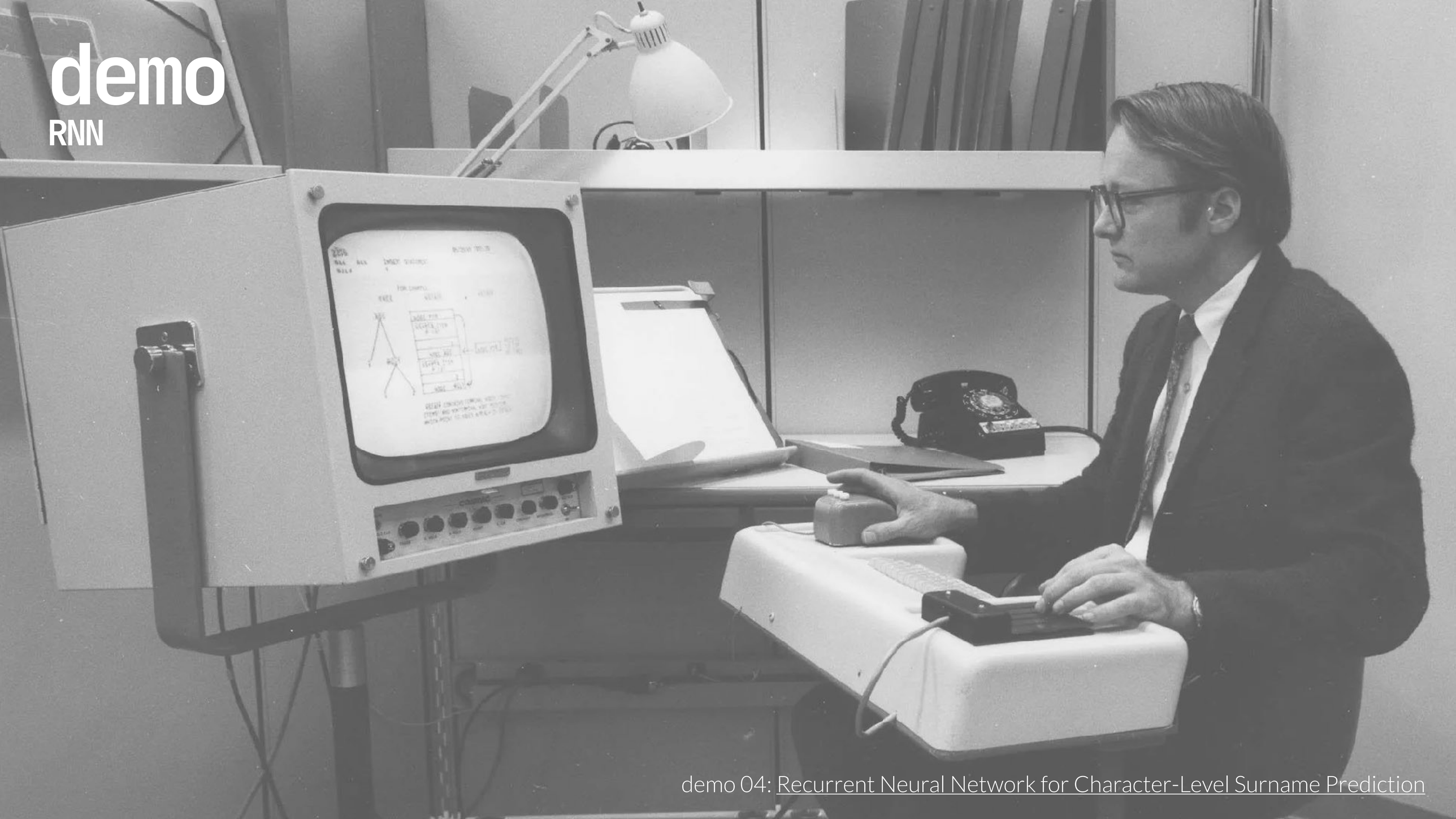


Training Process

- This backpropagation through time (BPTT) process is expensive
- Instead of continual updating after every time step, we tend to do update only every T steps (e.g., every sentence or paragraph)
  - not equivalent to using only a window size T (a la n-grams) because we still have “infinite memory”

demo

RNN

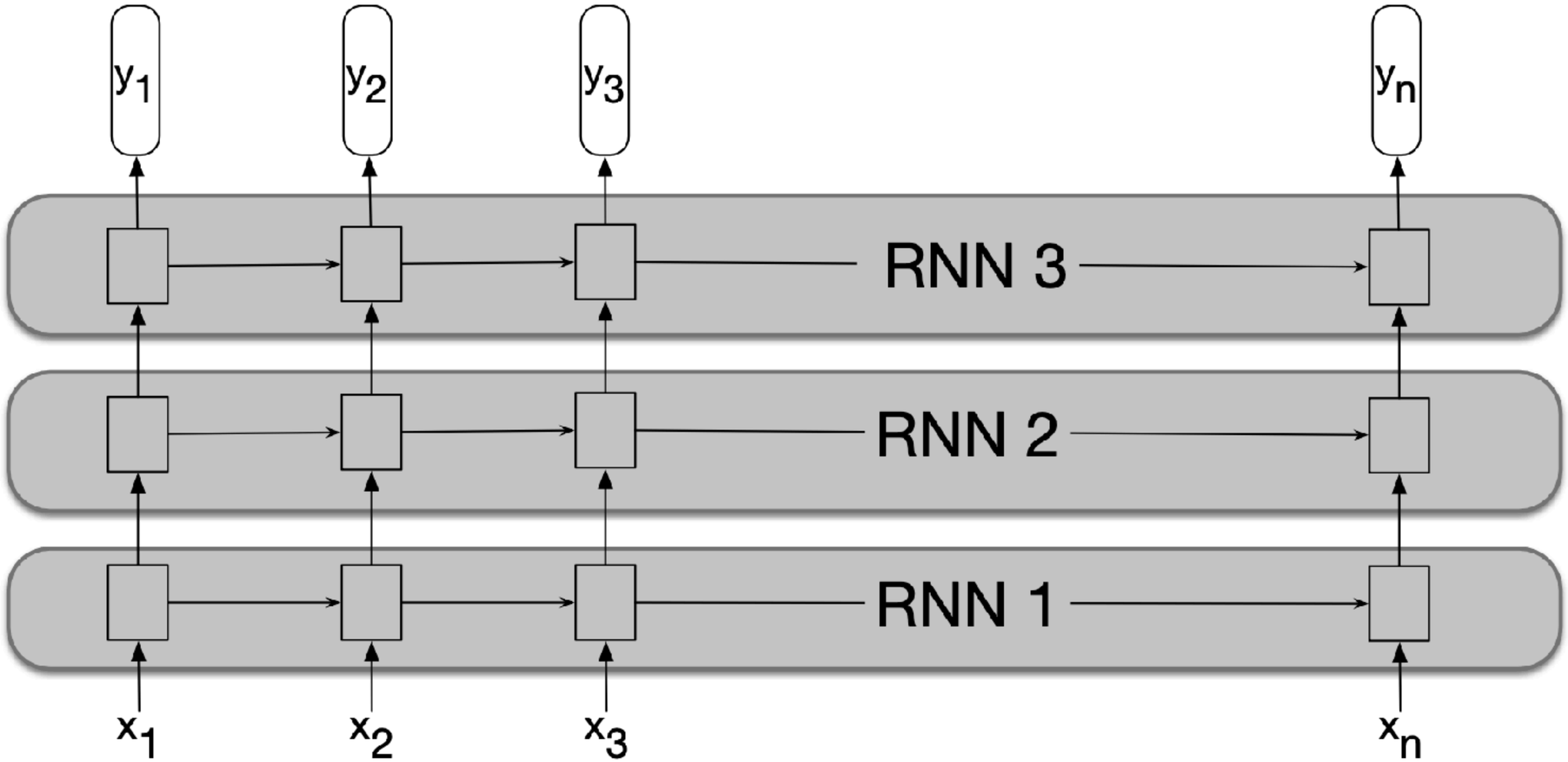


demo 04: Recurrent Neural Network for Character-Level Surname Prediction

Stacked LMs

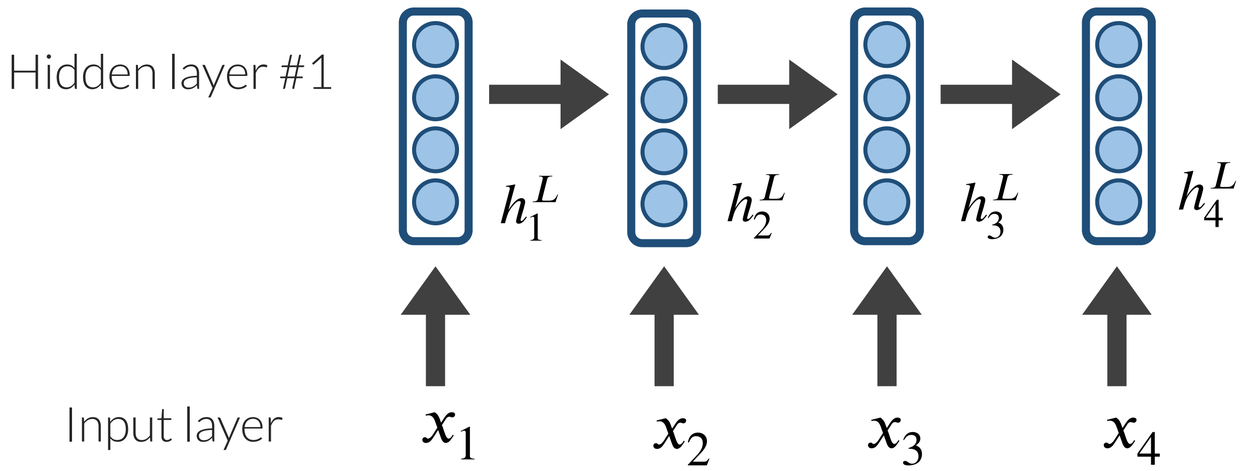
Stacked RNNs

multiple layers



Stacked RNNs

multiple layers

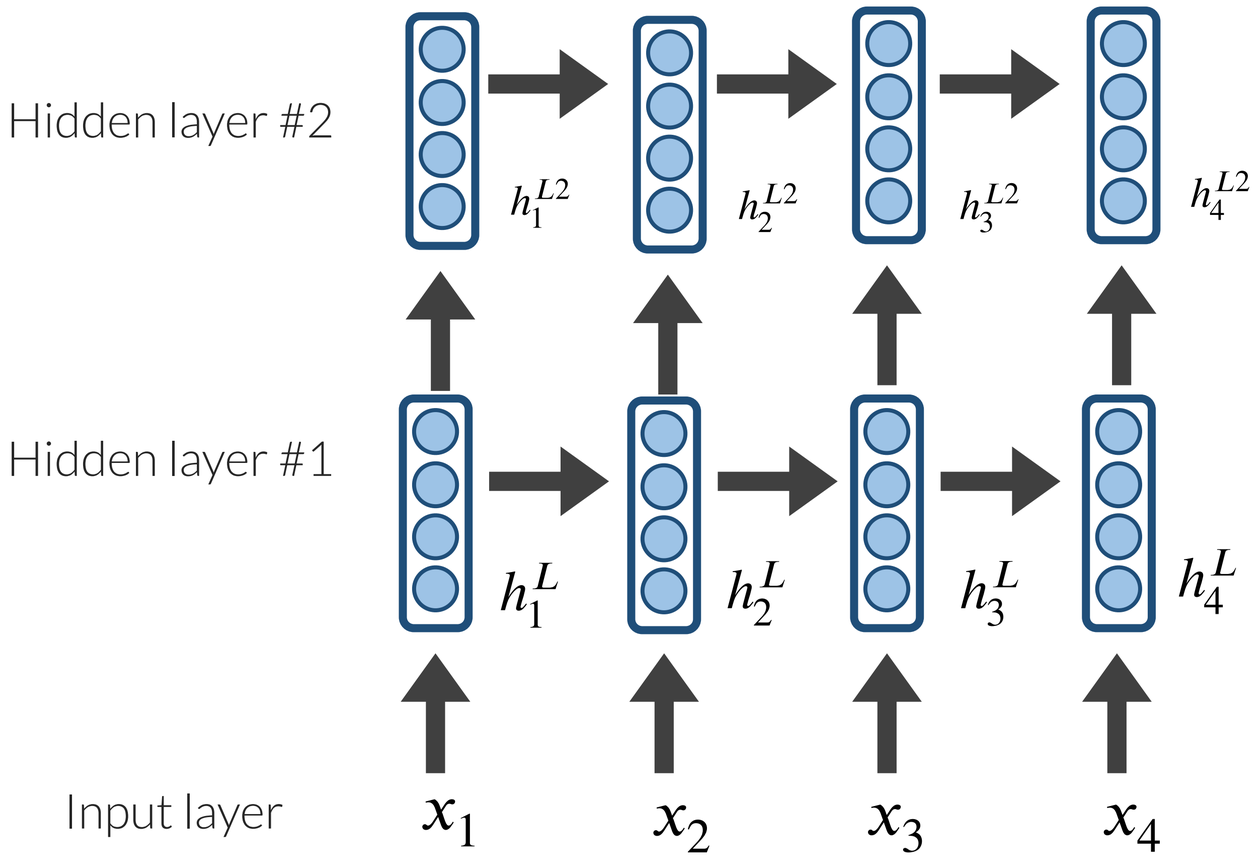


Hidden layers provide an abstraction (holds “meaning”).

Stacking hidden layers provides increased abstractions.

Stacked RNNs

multiple layers

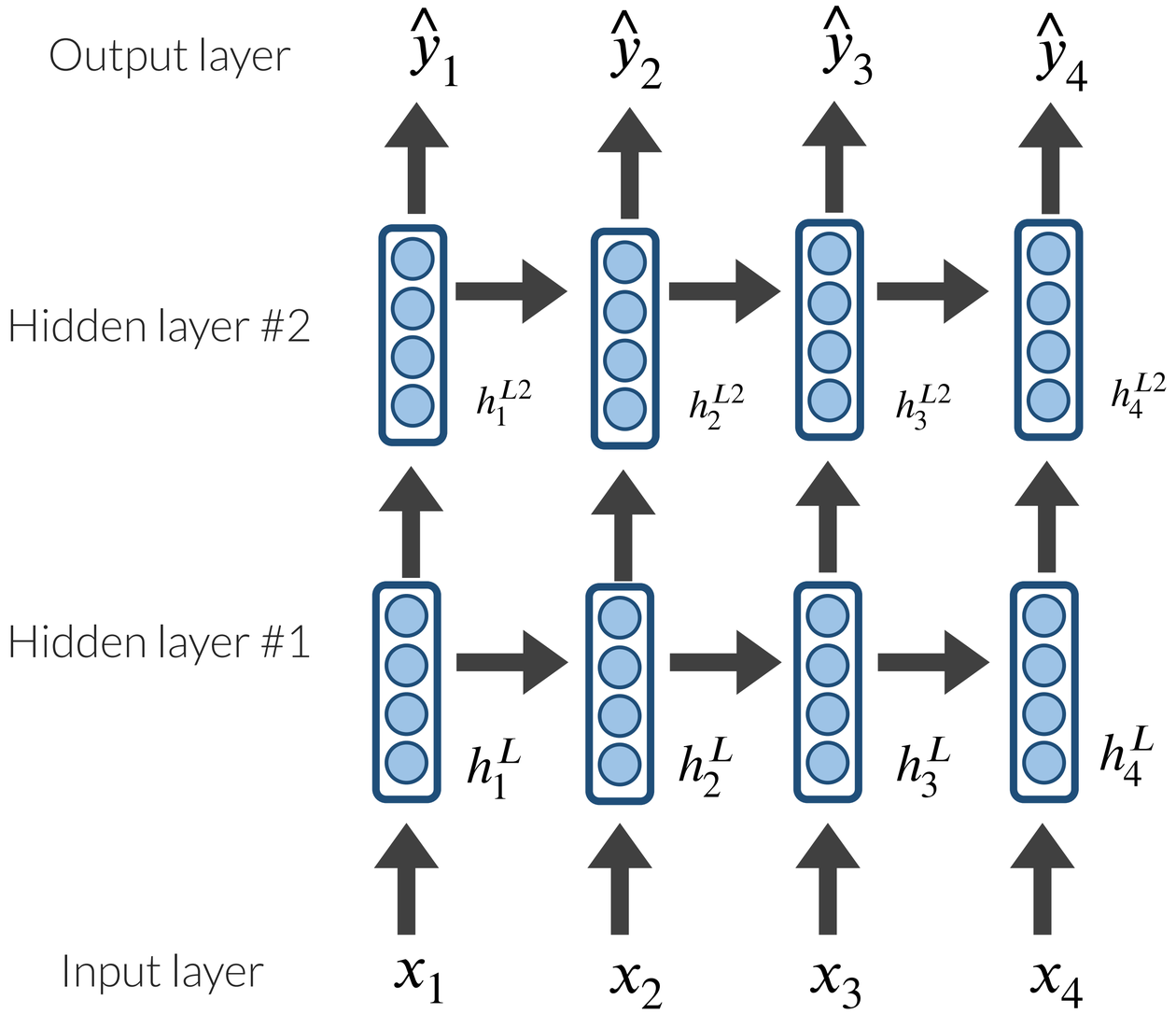


Hidden layers provide an abstraction (holds “meaning”).

Stacking hidden layers provides increased abstractions.

Stacked RNNs

multiple layers

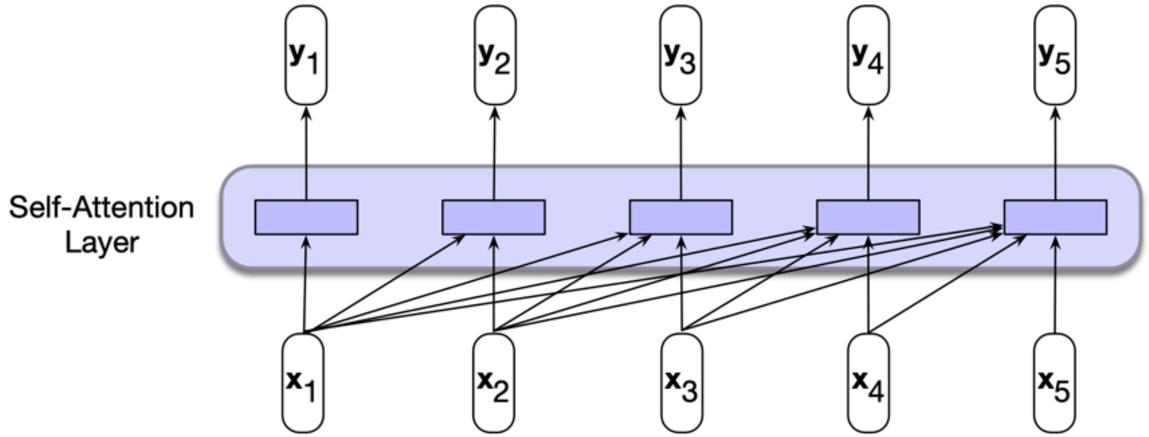


Hidden layers provide an abstraction (holds “meaning”).

Stacking hidden layers provides increased abstractions.

Bi-Directional LMs

Autoregressive / left-to-right

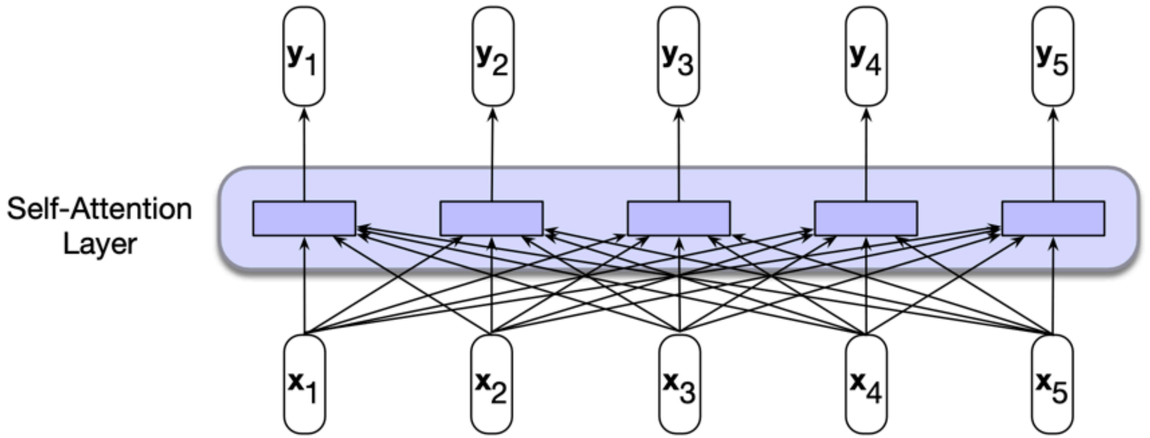


computation for input $\mathbf{x}_1, \ldots, \mathbf{x}_3$ blind to $\mathbf{x}_4$ and $\mathbf{x}_5$

$\mathbf{y}_5$ is embedding for input $\mathbf{x}_1, \ldots, \mathbf{x}_5$

$\mathbf{y}_5$ is a "left-contextual embedding"

Bidirectional

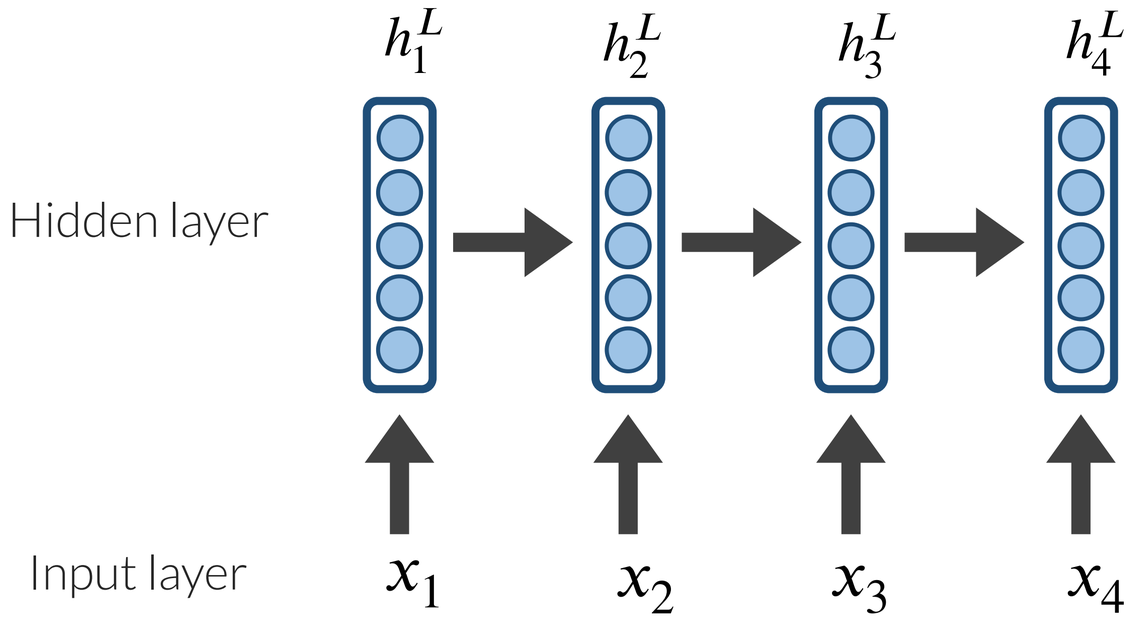


computation for input $\mathbf{x}_1, \ldots, \mathbf{x}_3$ sees $\mathbf{x}_4$ and $\mathbf{x}_5$

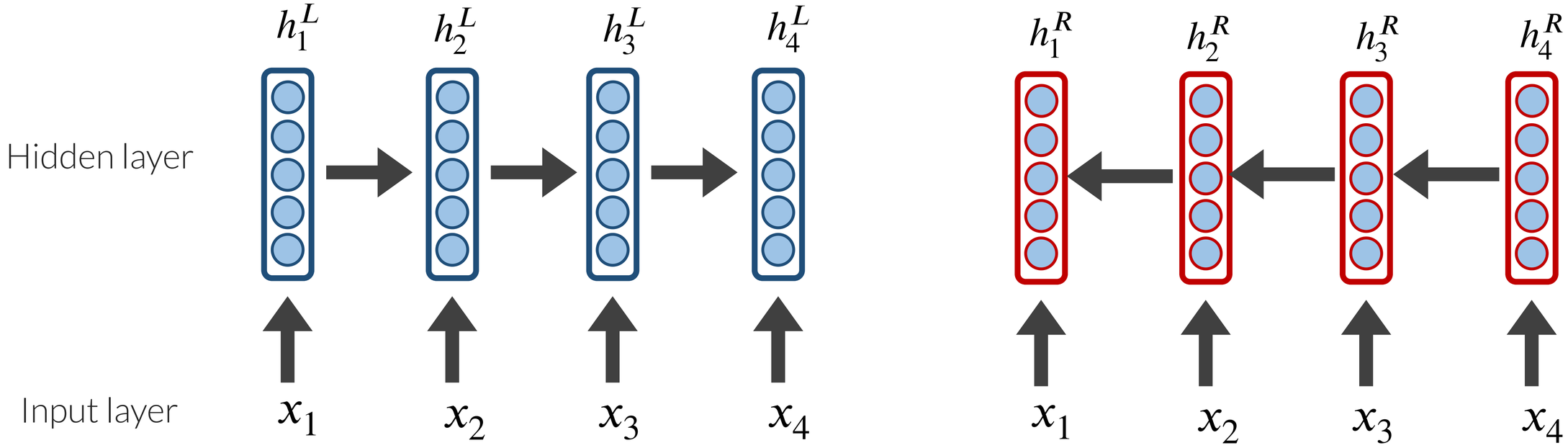
$\mathbf{y}_1, \ldots, \mathbf{y}_5$ is embedding for input $\mathbf{x}_1, \ldots, \mathbf{x}_5$

$\mathbf{y}_i$ are bidirectional "contextual embeddings"

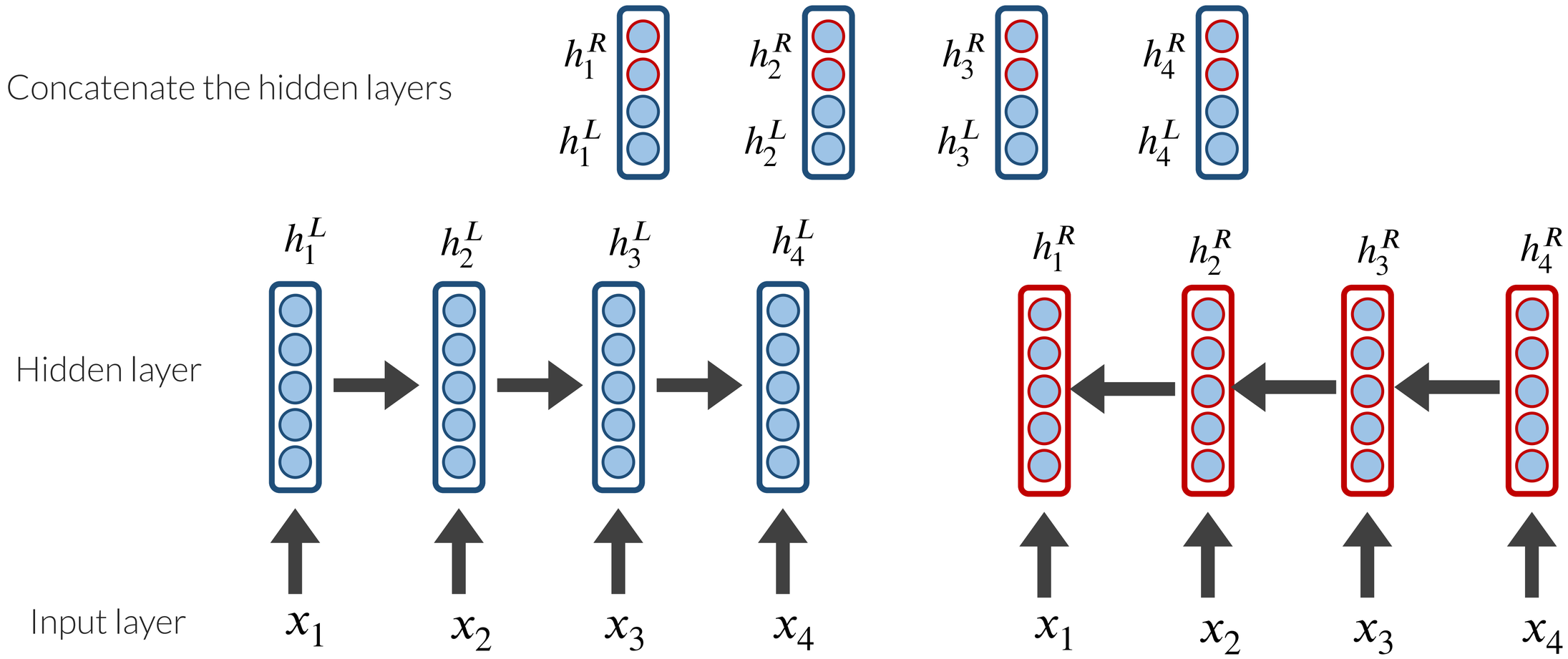
Reading Bidirectionally



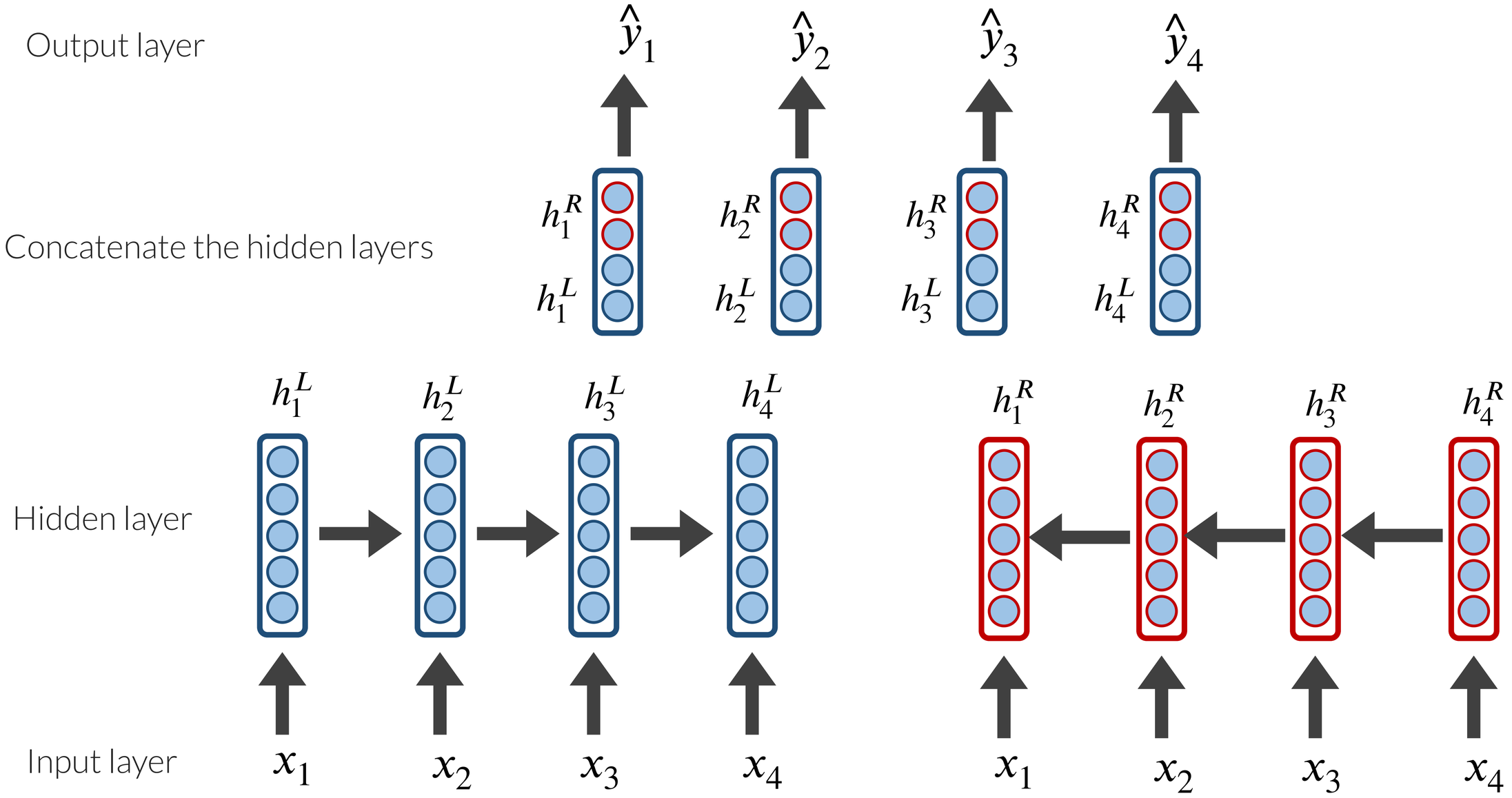
Reading Bidirectionally



Reading Bidirectionally



Reading Bidirectionally



Bi-directional LMs

Strengths

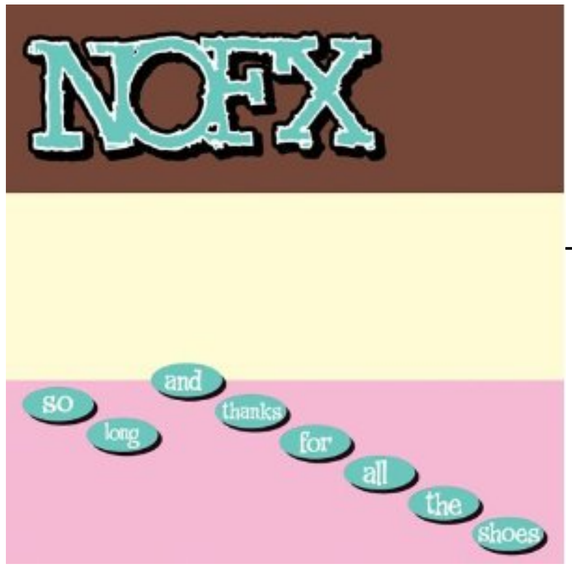
- Usually performs at least as well as uni-directional RNNs/LSTMs
- More encompassing (left & right) contextualized embeddings

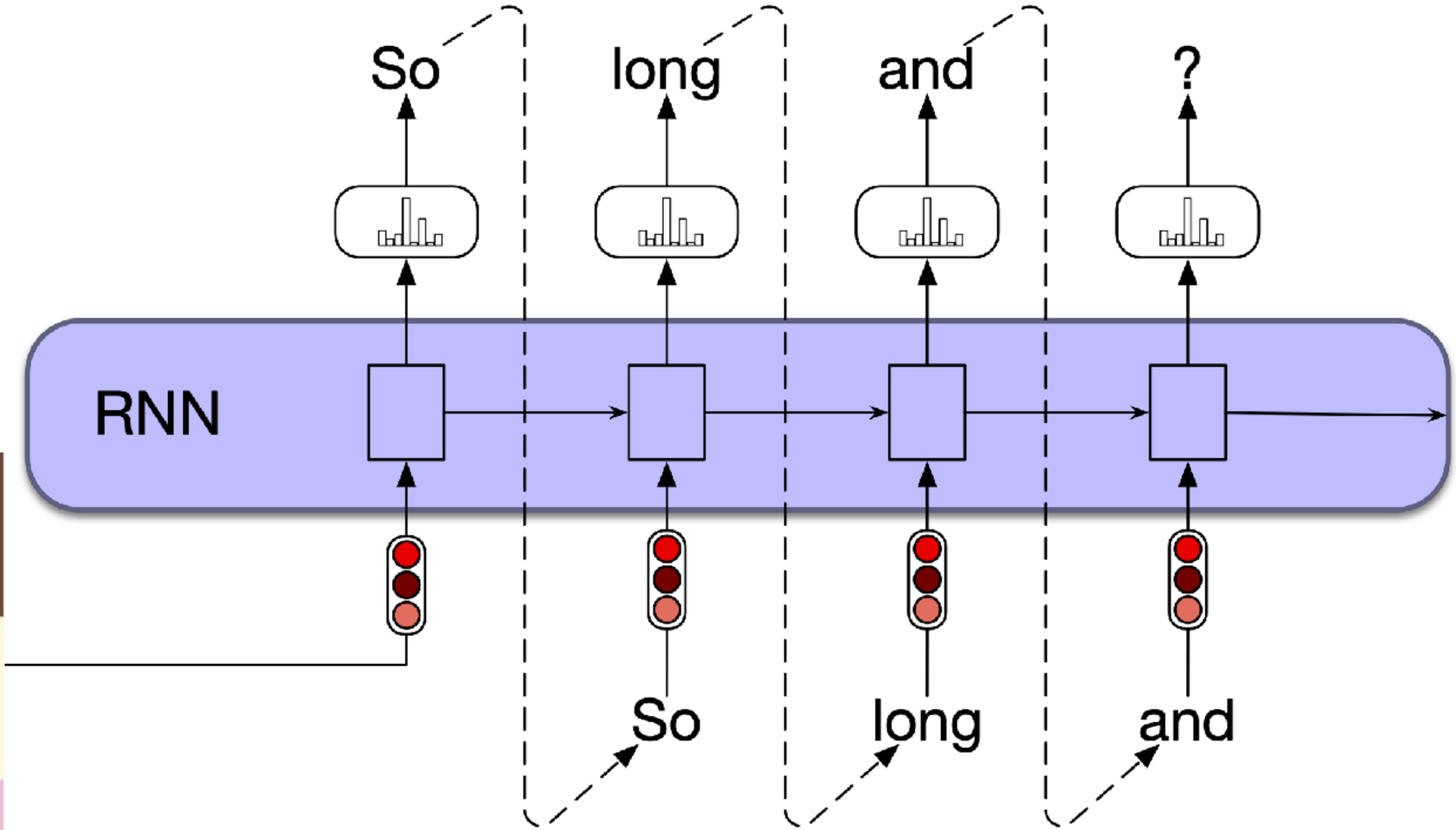
Weaknesses

- Slower to train
- Only possible if access to full data is allowed
- Less suitable for (autoregressive) language generation

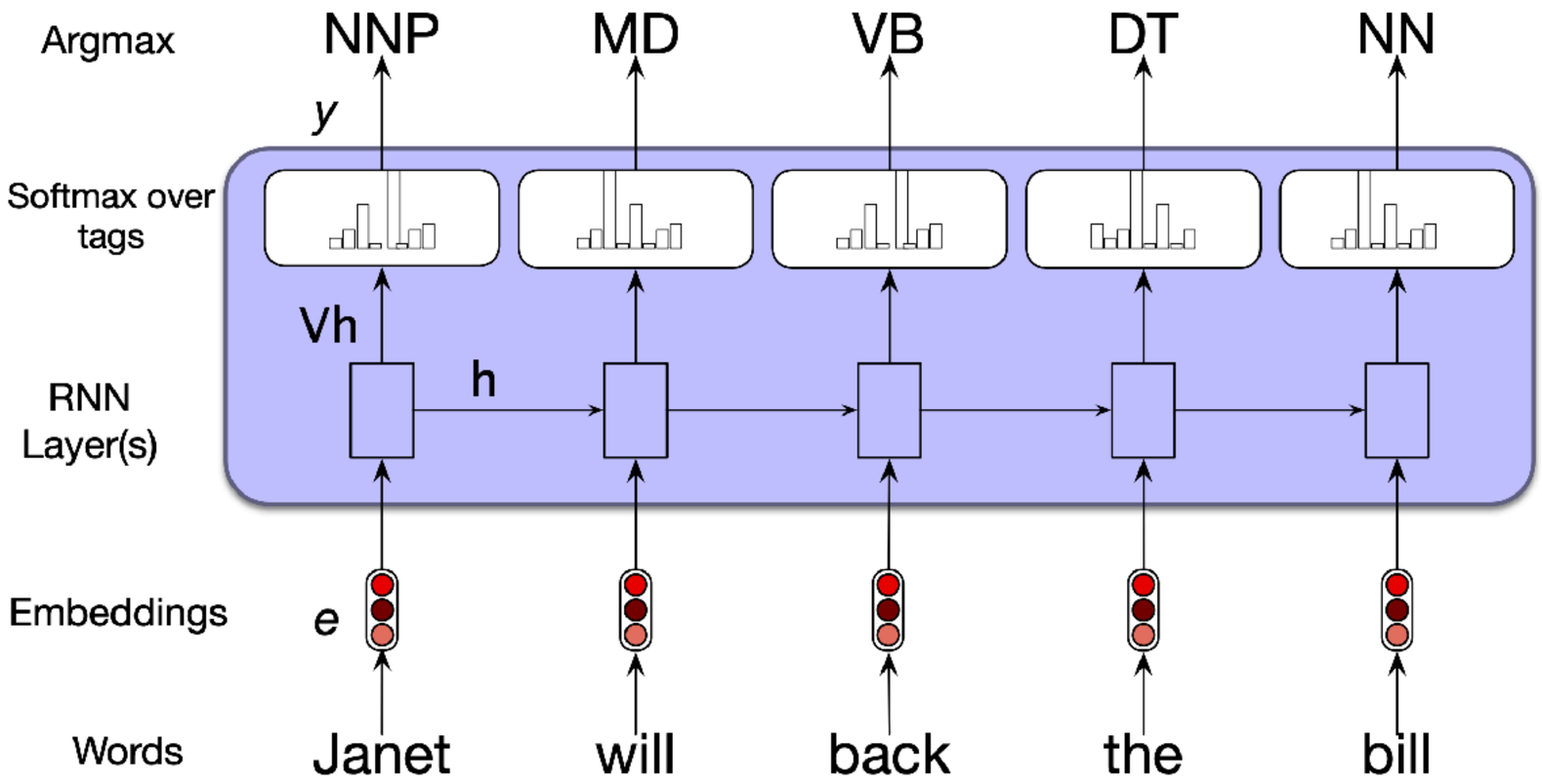
Beyond Sequence Completion

Image-to-text

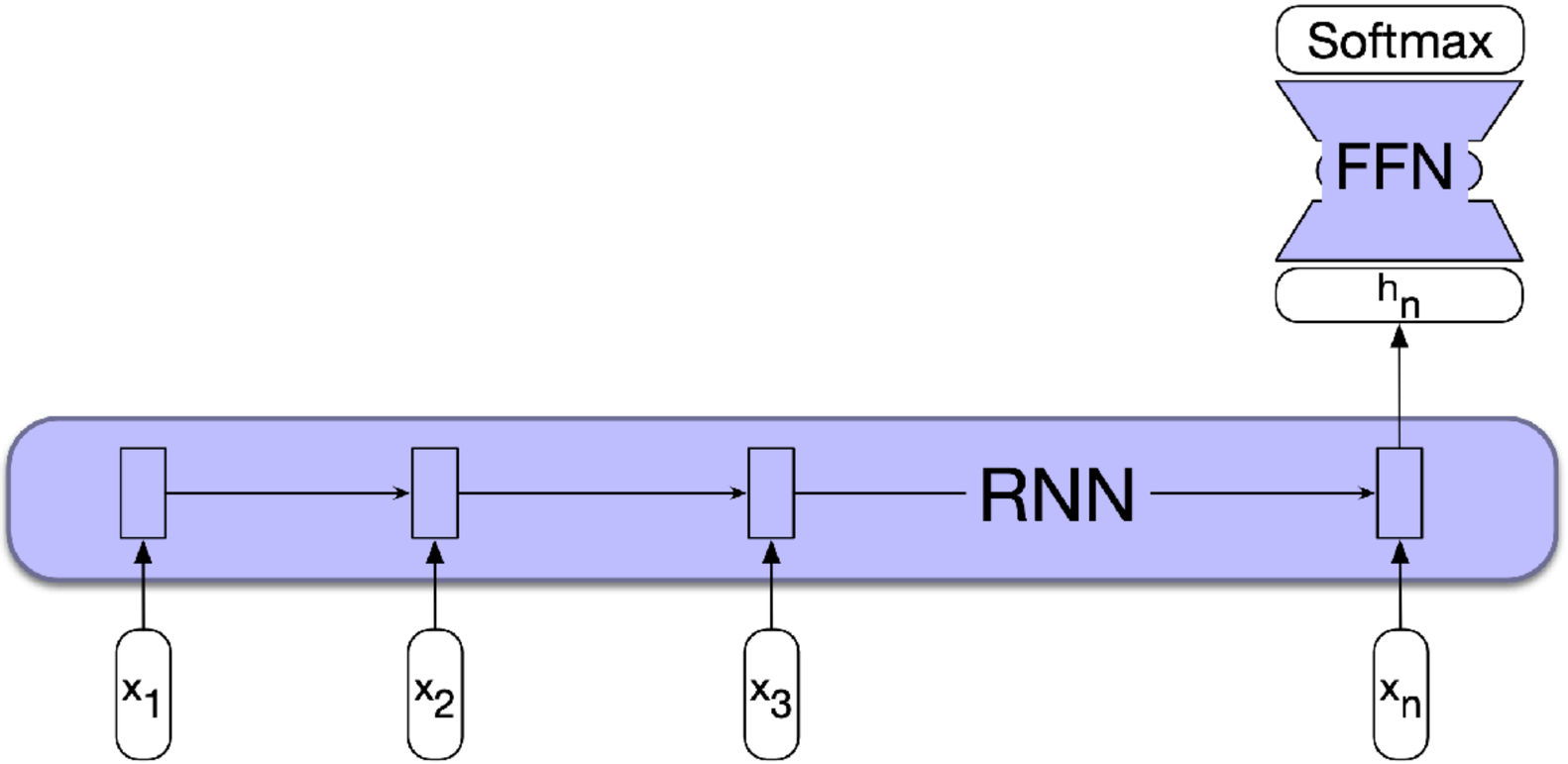




Sequence-to-sequence

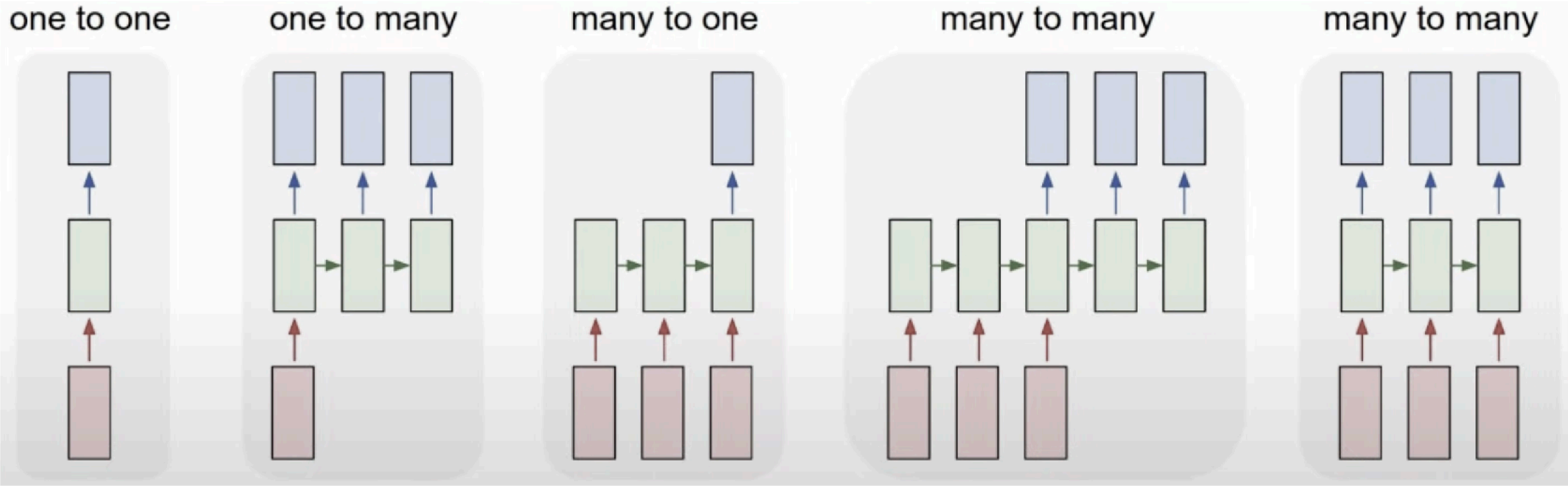


Text classification



Different kinds of sequence processing models

sequence as input and/or (simultaneous) output



Think break:

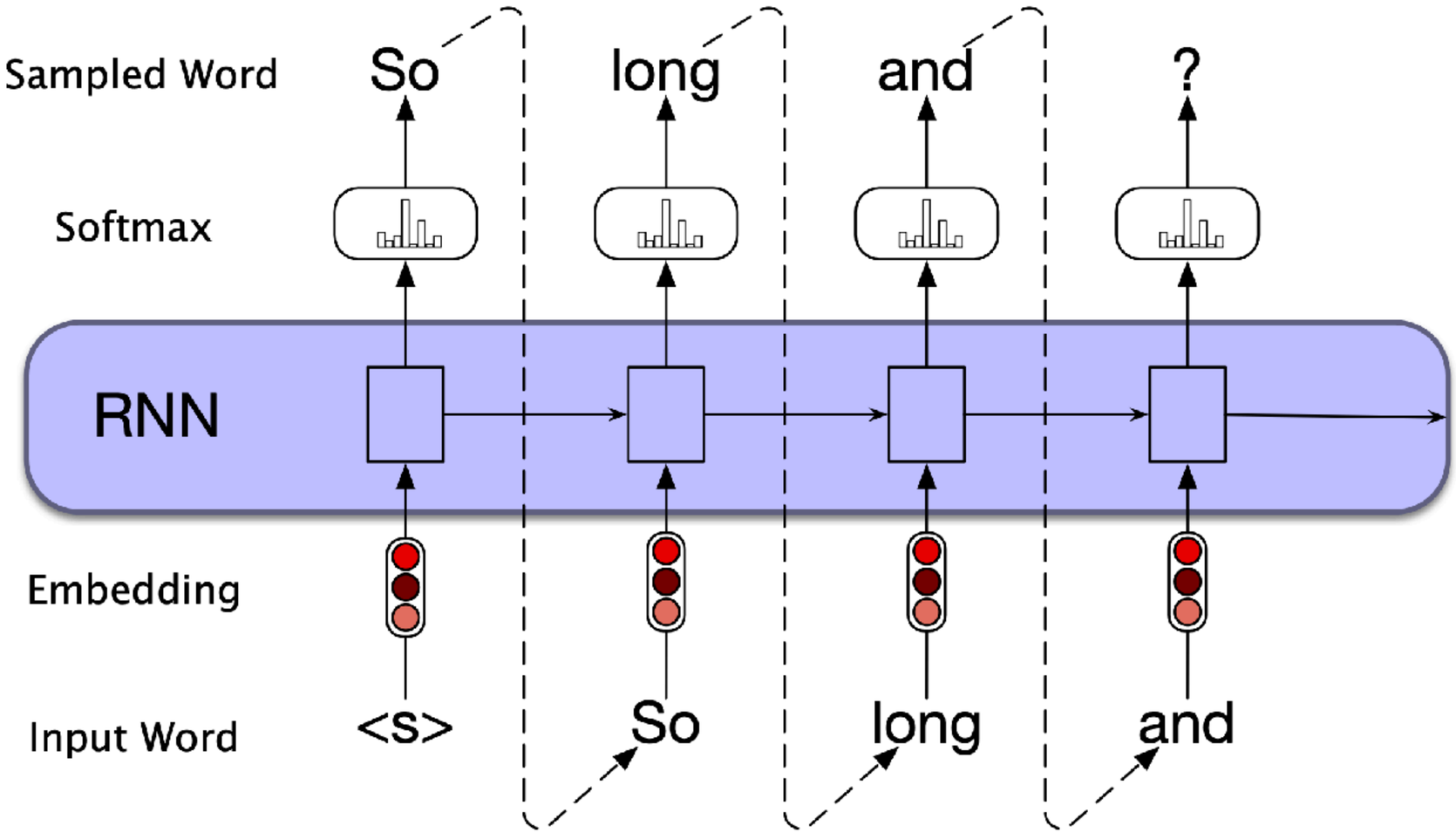
Which of these architectures go with which kinds of applications?

Where, if at all, would you locate autoregressive / masked language modeling?

excursion on Decoding & Training

Autoregressive generation

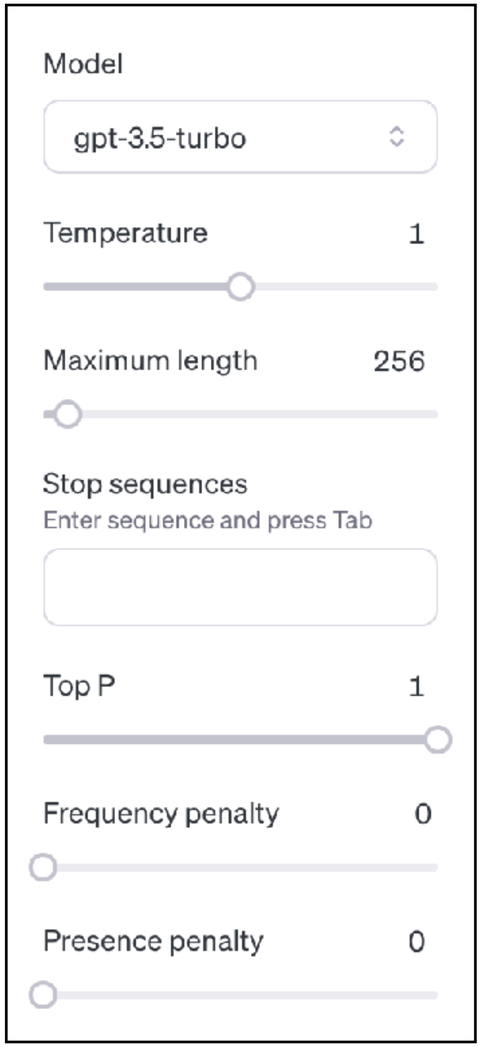
left-to-right / causal model



Predictions from different decoding schemes

based on next-token probability $P(w_{i+1} \mid w_{1:i})$

- **pure sampling**
  - next token is sampled from NTP distribution: $w_{i+1} \sim P(\cdot \mid w_{1:i})$
- **greedy decoding**
  - next token is the one with the highest NTP: $w_{i+1} = \arg\max_{w'} P(w' \mid w_{1:i})$
- **softmax sampling**
  - next token is sampled from softmax of NTP distribution: $w_{i+1} \sim SM_{\alpha}\left(P(\cdot \mid w_{1:i})\right)$
- **top-k sampling**
  - next token is sampled from NTP distribution after restricting to the **k** most likely words
- **top-p sampling**
  - next token is sampled from NTP distribution after restricting to the smallest set of the most likely tokens which together comprise at least NTP **p**
- **beam search**
  - frequently use, if relevant we will cover it later

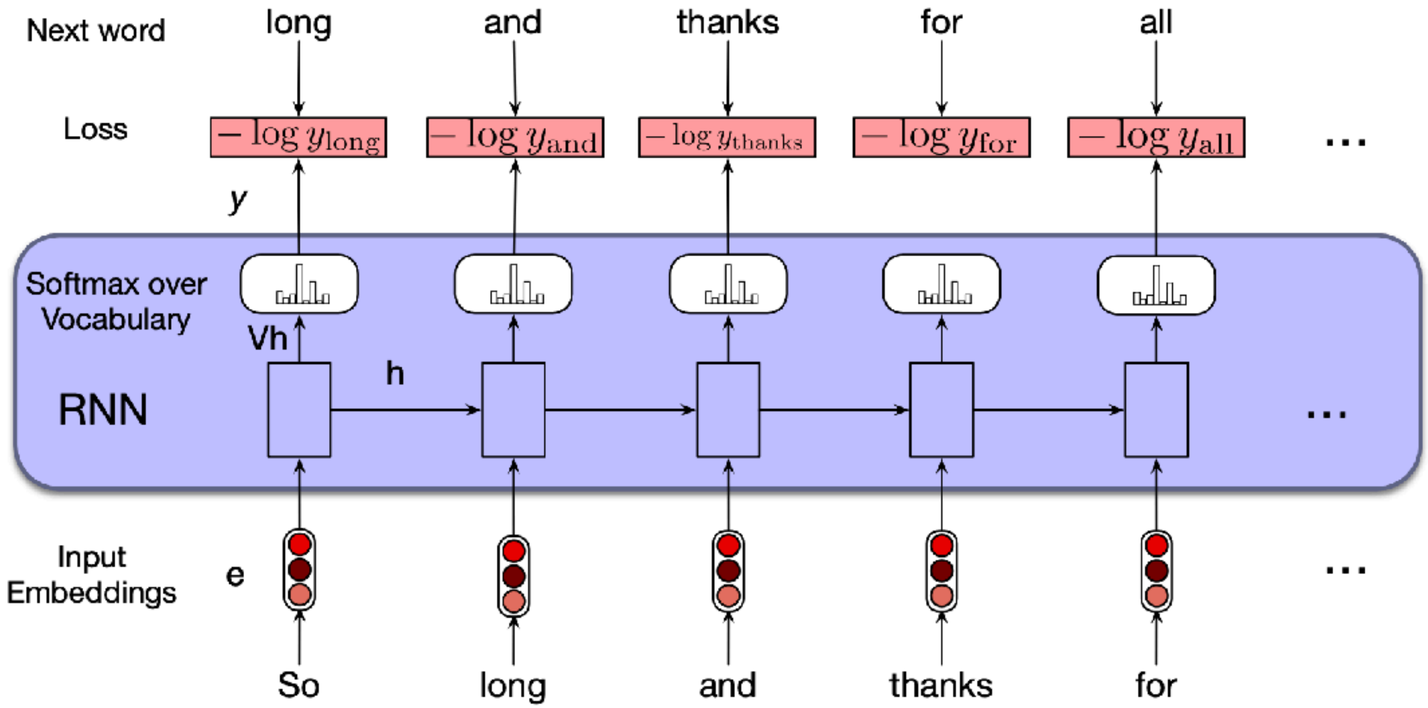


from OpenAI's Playground

Training RNNs

using teacher forcing & next-word surprisal

- teacher forcing
  - predict each next token given the preceding input (not the model-generated sequence)
- next-word surprisal (CE loss)
  - loss function is (average) next-token surprisal



Excursion: Different training regimes

- **teacher forcing**
  - LM is fed true word sequence
  - training signal is next-word assigned to true word
- **autoregressive training** (aka free-running mode)
  - LM autoregressively generates a sequence
  - training signal is next-word probability assigned to true word
- **curriculum learning** (aka scheduled sampling)
  - combine teacher-forced and autoregressive training
  - start with mostly teacher forcing, then increase amount of autoregressive training
- **professor forcing**
  - combines teacher forcing with adversarial training
  - generative adversarial network GAN is trained to discriminate (autoregressive) predictions from actual data
  - LM is trained to minimized this discriminability
- **decoding-based**
  - use prediction function (decoding scheme) to optimize based on *actual* output

Plain causal LMs in a nutshel

- **definition**
  - sequence probabilities given by product of next-word probabilities
- **training**
  - minimize next-word surprisal
- **prediction**
  - sample auto regressively, using next-word probabilities
- **evaluation**
  - perplexity or average surprisal
- **consistent def-train-pred-eval scheme**

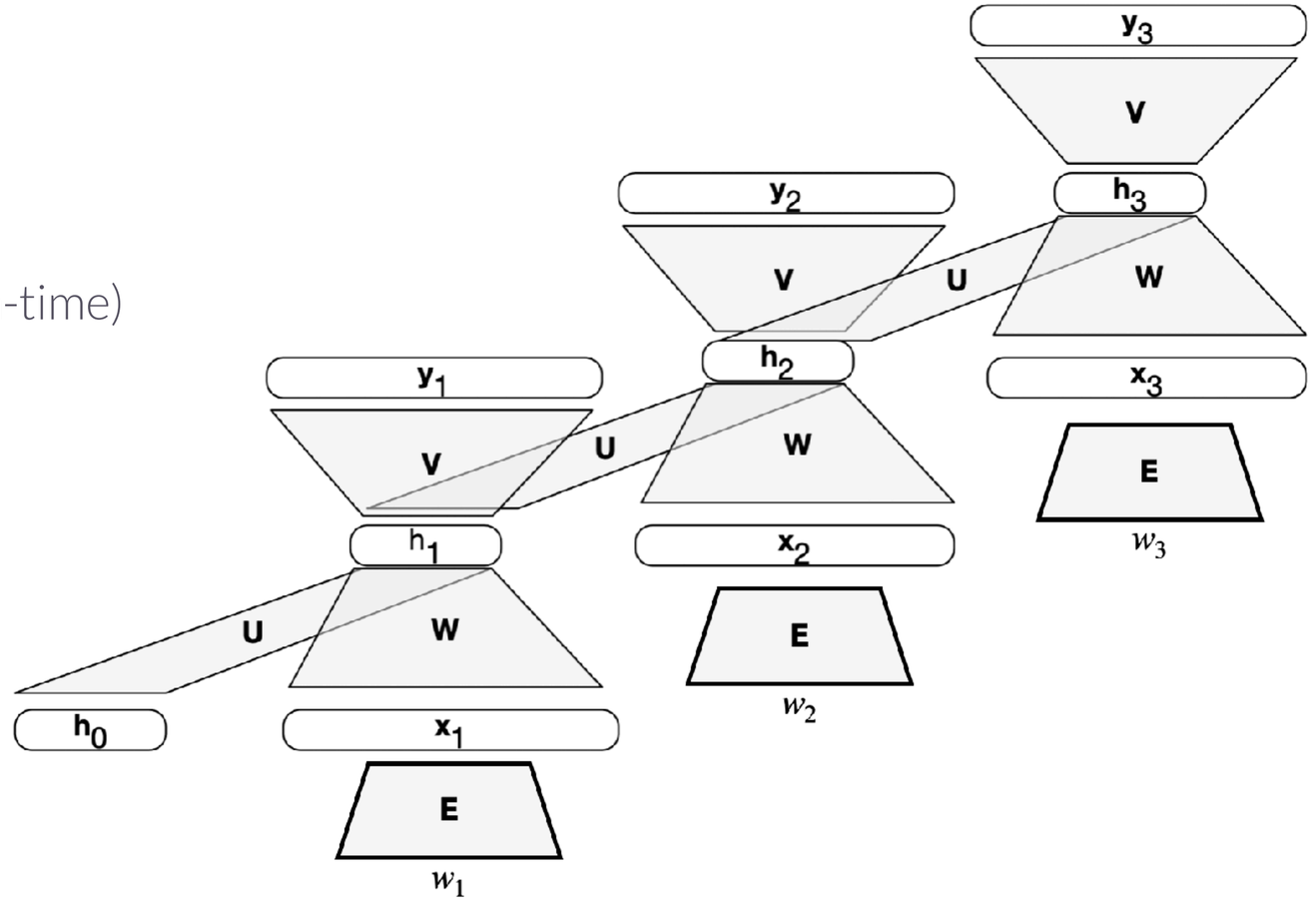
Dirty reality

- **definition**
  - usually only implicit, often unclear
  - task-dependent
- **training**
  - usually based on next-word surprisal
  - other (mixed) **training regimes** exist
- **prediction**
  - whole battery of **decoding strategies**
- **evaluation**
  - baseline: perplexity or average surprisal
  - additional measure of text quality
- **possibly inconsistent**

Problems with RNNS

Problems with RNNs

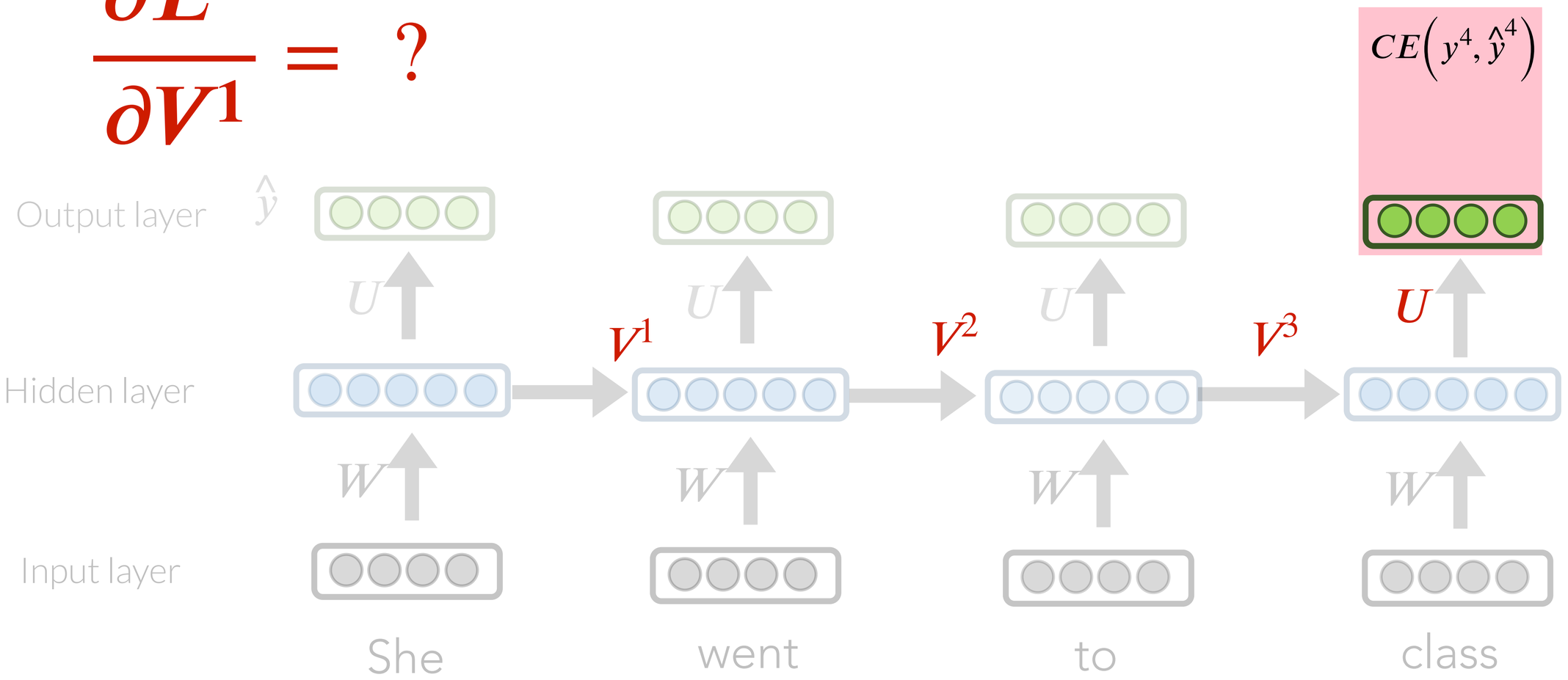
- **conceptual**
  - two-fold role of hidden state:
    - memory for past sequence
    - recommend what to do now
- **technical**
  - difficult to train (backpropagation-through-time)
  - vanishing & exploding gradients



On Vanishing & Exploding Gradients

$$\frac{\partial L^4}{\partial V^1} = \; ?$$

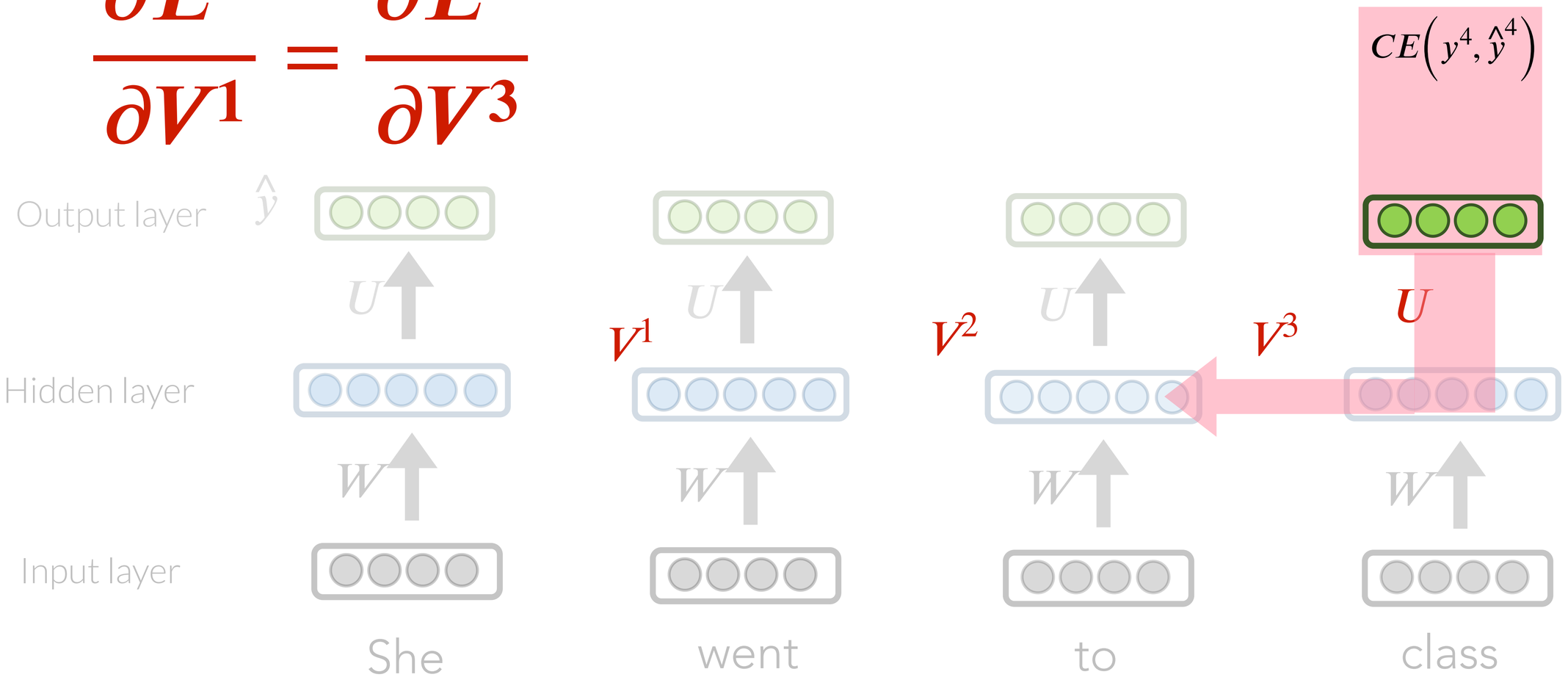
$$\frac{\delta L^4}{\delta V^1}$$



On Vanishing & Exploding Gradients

$$\frac{\delta L^4}{\delta V^1}$$

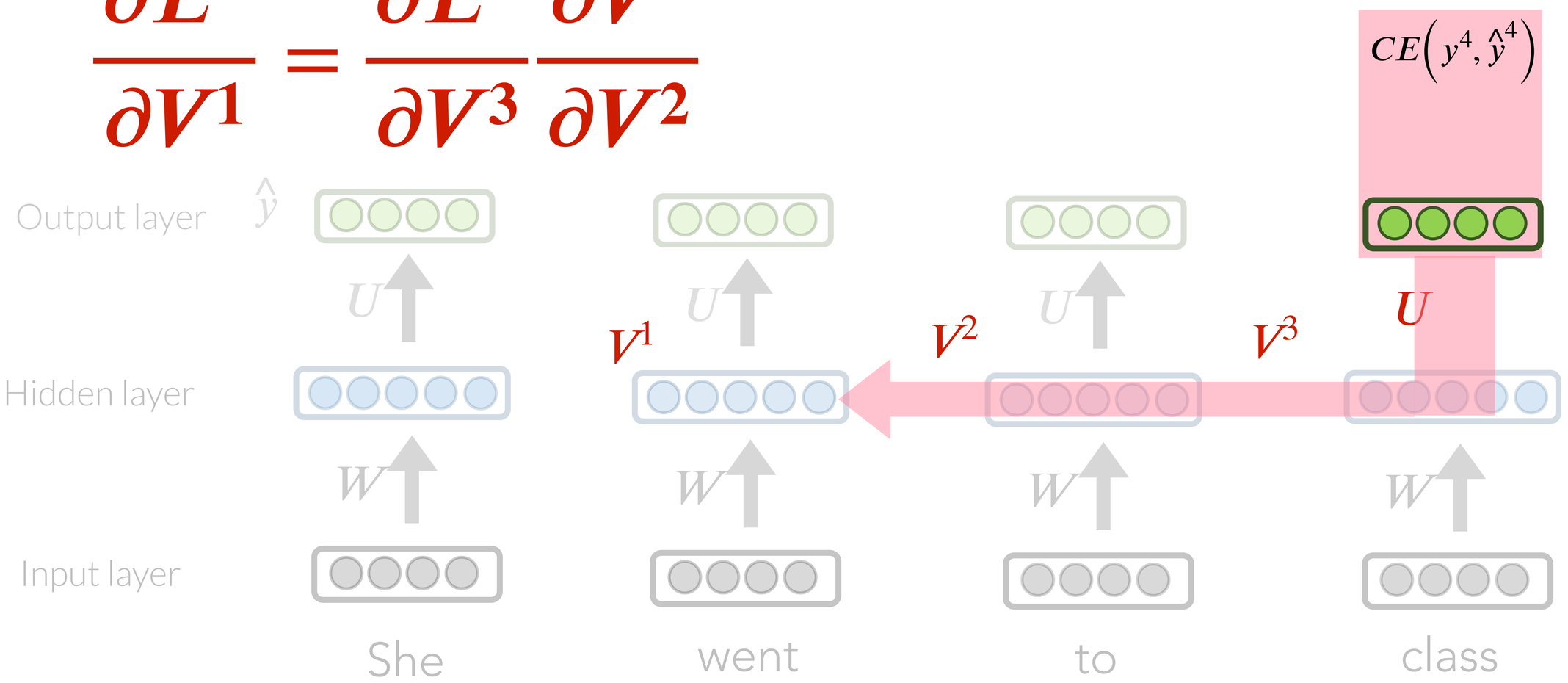
$$\frac{\partial L^4}{\partial V^1} = \frac{\partial L^4}{\partial V^3}$$



On Vanishing & Exploding Gradients

$$\frac{\partial L^4}{\partial V^1} = \frac{\partial L^4}{\partial V^3}\frac{\partial V^3}{\partial V^2}$$

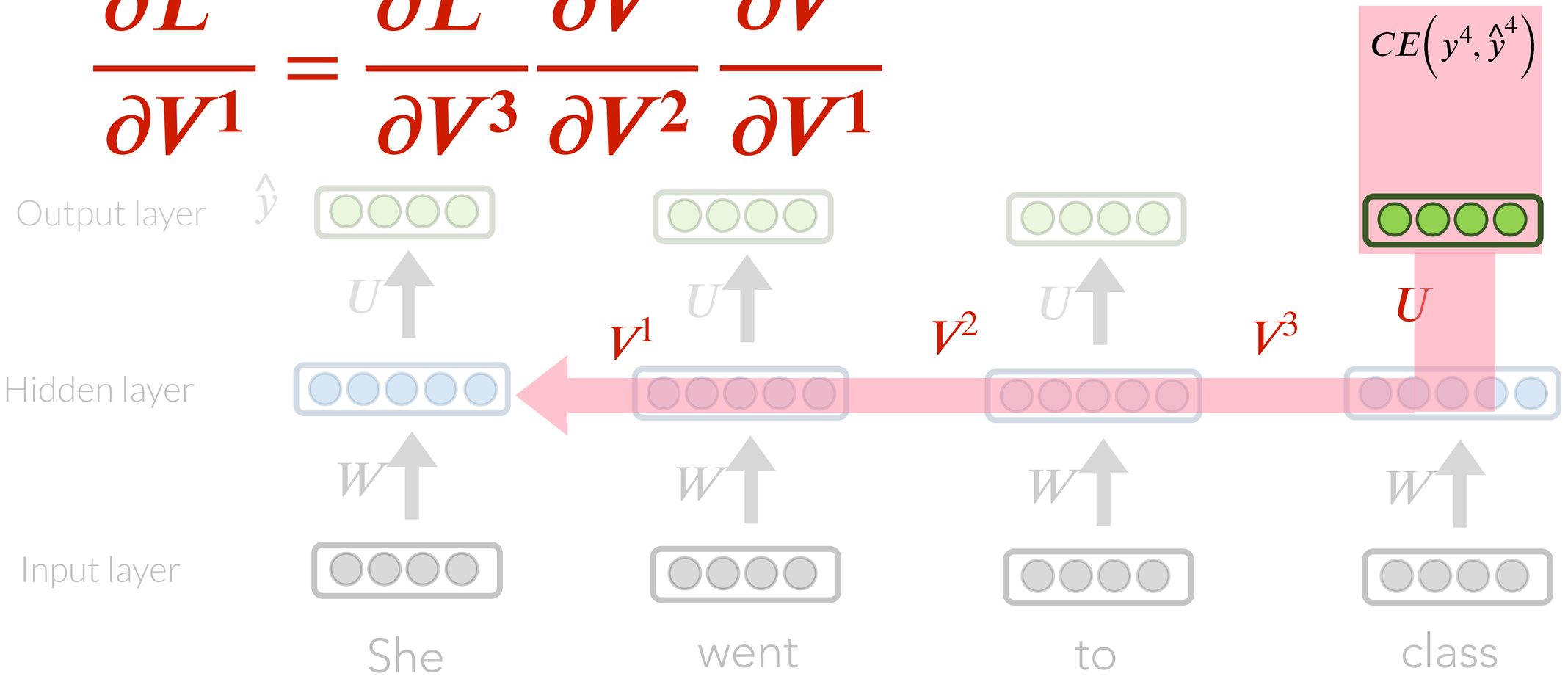
$$\frac{\delta L^4}{\delta V^1}$$



On Vanishing & Exploding Gradients

$$\frac{\delta L^4}{\delta V^1}$$

$$\frac{\partial L^4}{\partial V^1} = \frac{\partial L^4}{\partial V^3}\frac{\partial V^3}{\partial V^2}\frac{\partial V^2}{\partial V^1}$$



On Vanishing & Exploding Gradients

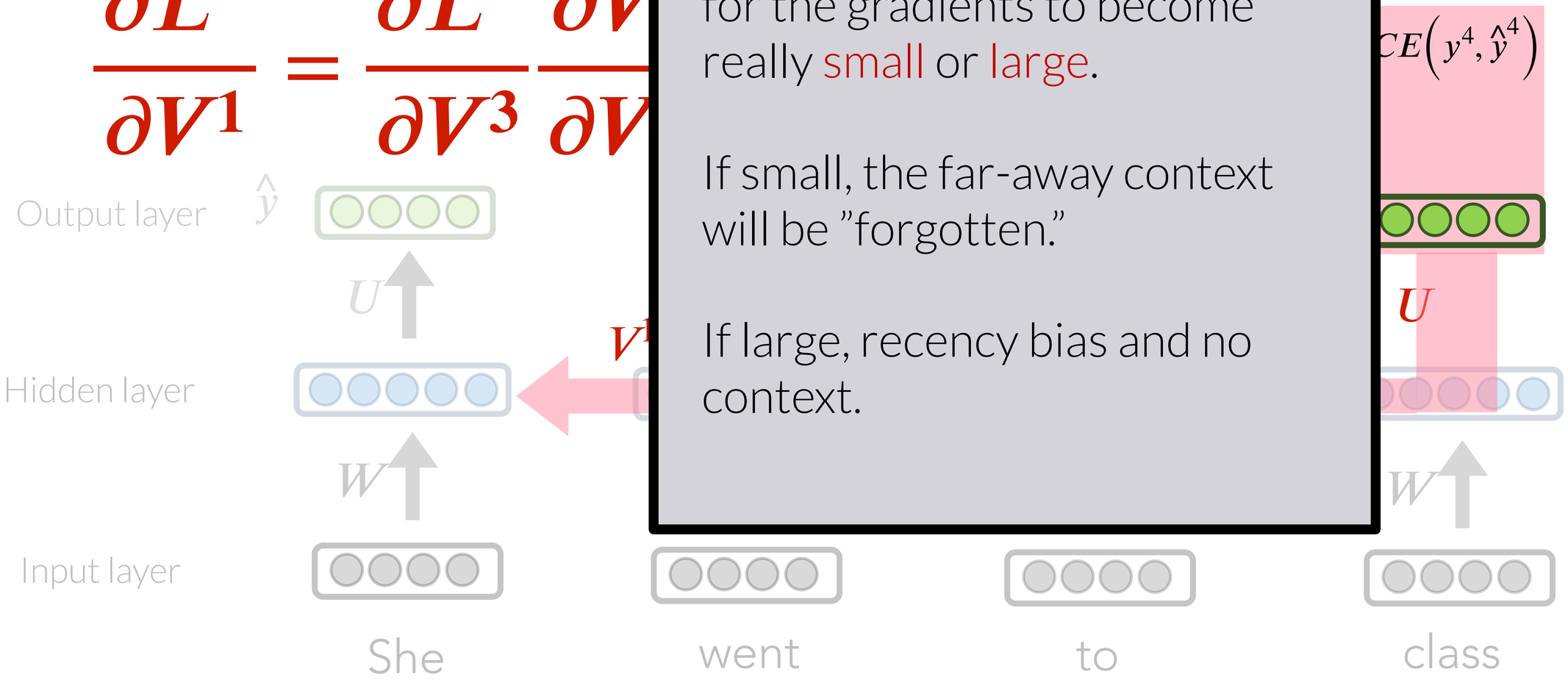
$$\frac{\delta L^4}{\delta V^1}$$

$$\frac{\partial L^4}{\partial V^1} = \frac{\partial L^4}{\partial V^3} \frac{\partial V}{\partial V}$$

This long path makes it easy for the gradients to become really small or large.

If small, the far-away context will be ”forgotten.”

If large, recency bias and no context.



Long/Short-Term Memory Models

LSTM Motivation

- A type of RNN that is designed to better handle long-range dependencies

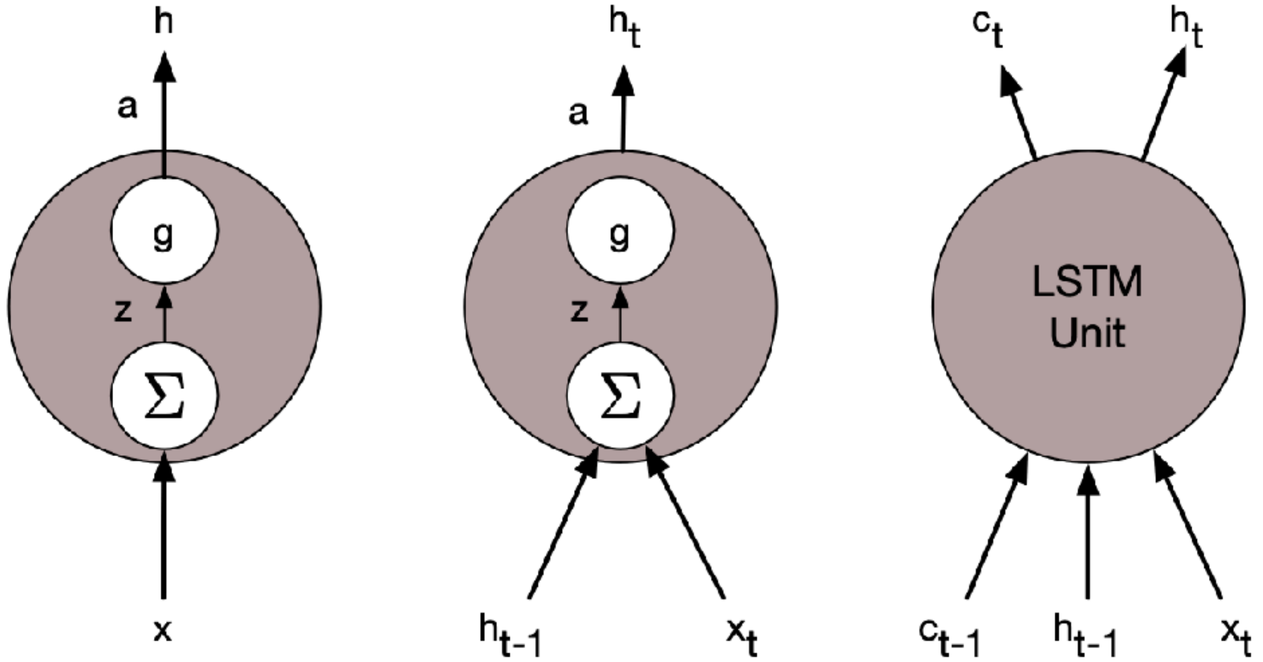
- In ”vanilla” RNNs, the hidden state is perpetually being rewritten

- In addition to a traditional hidden state h, let’s have a dedicated memory cell c for long-term events. More power to relay sequence information.

Modular architectures

cells / units

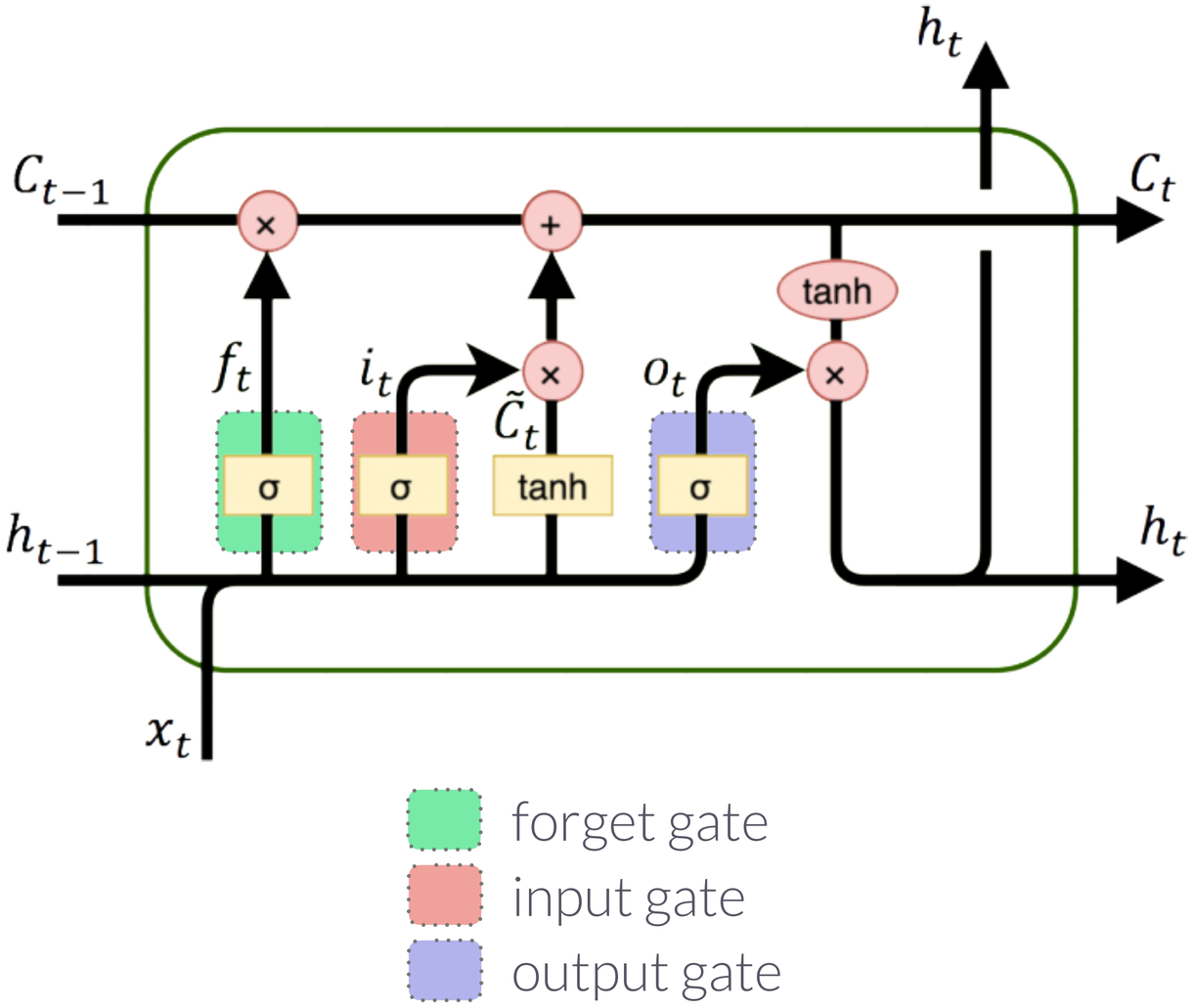
common mapping: input to hidden state: x ↦ h



MLP RNN LSTM

LSTM cell

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$



LSTM Architecture

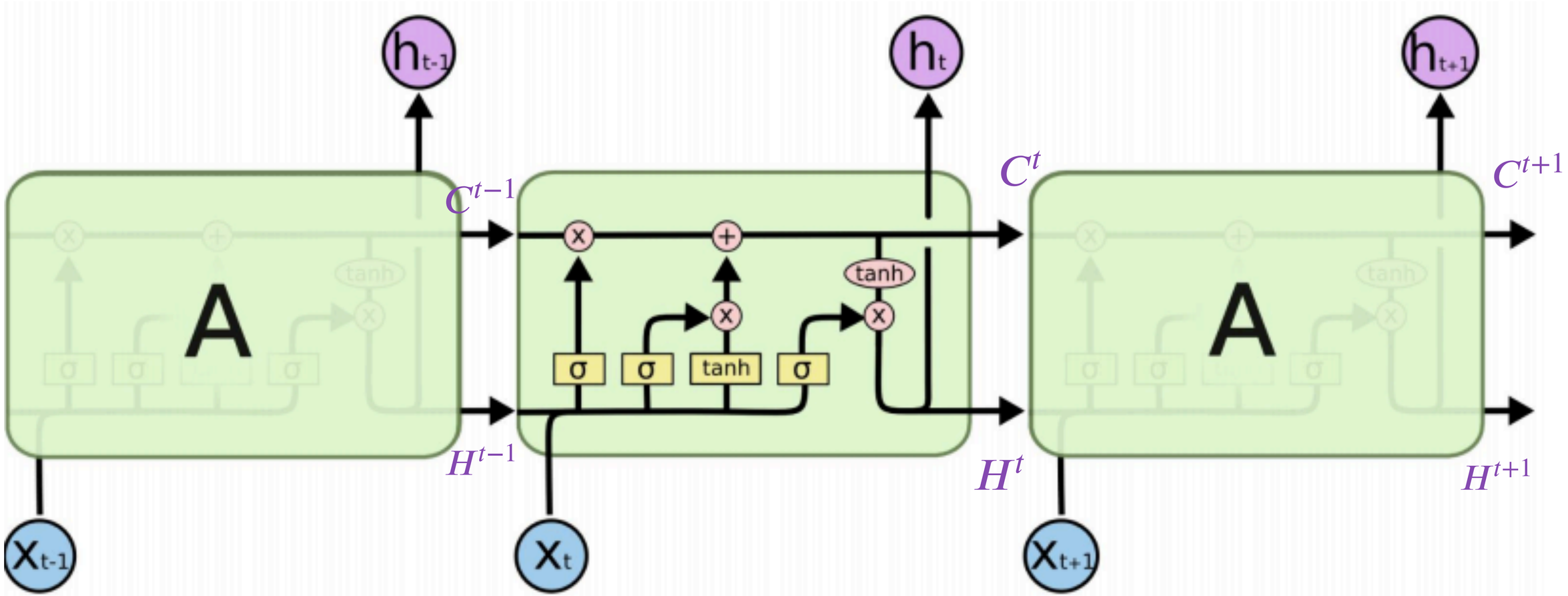


Diagram: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM Architecture

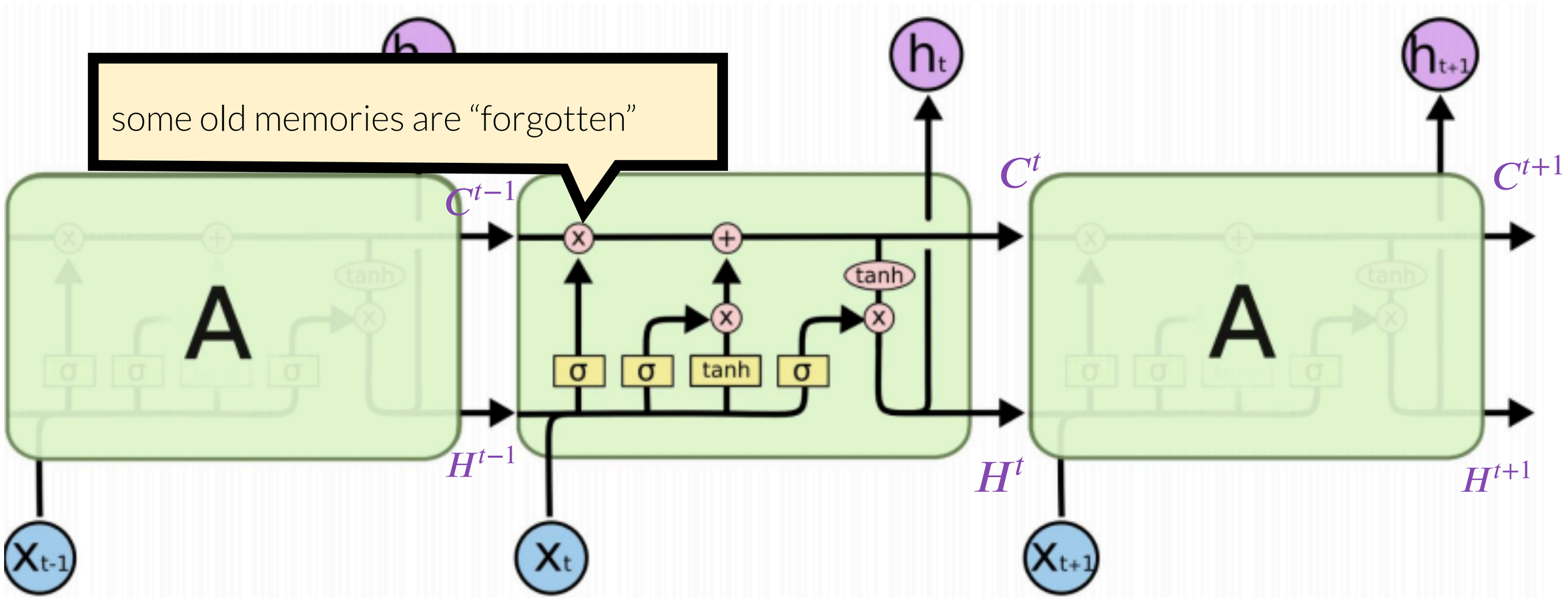


Diagram: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM Architecture

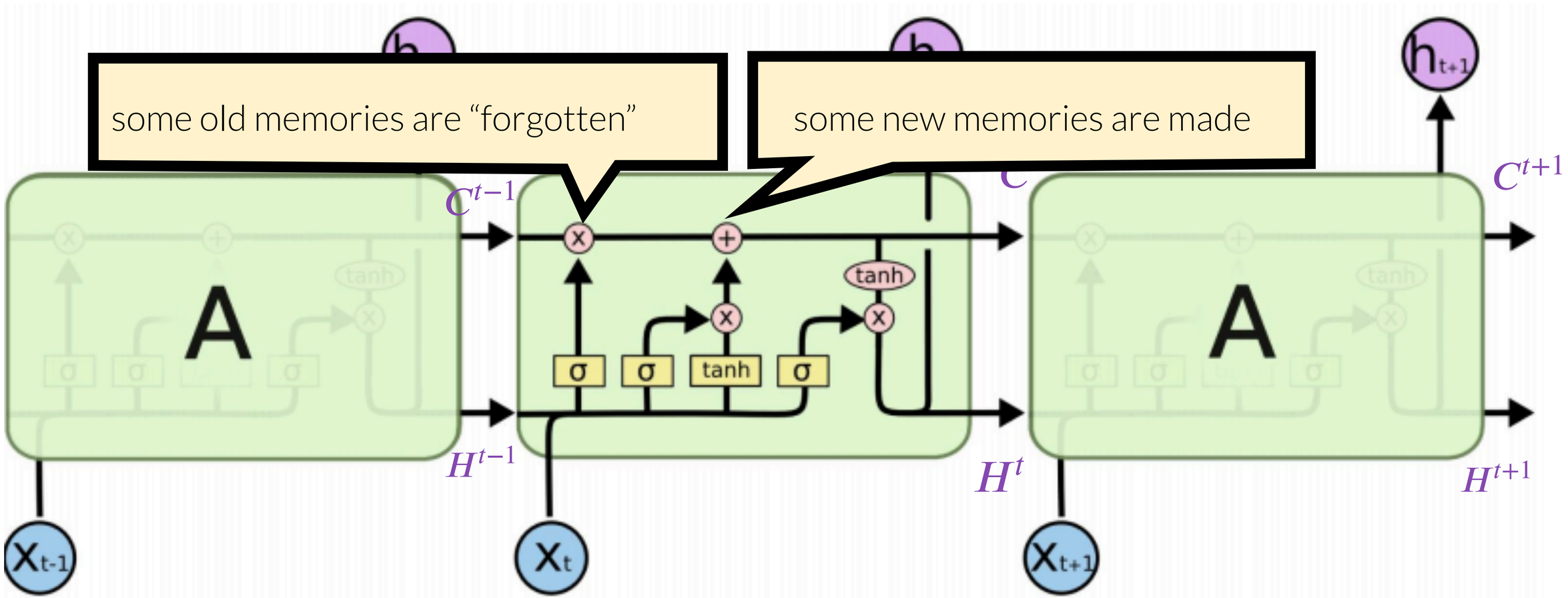


Diagram: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM Architecture

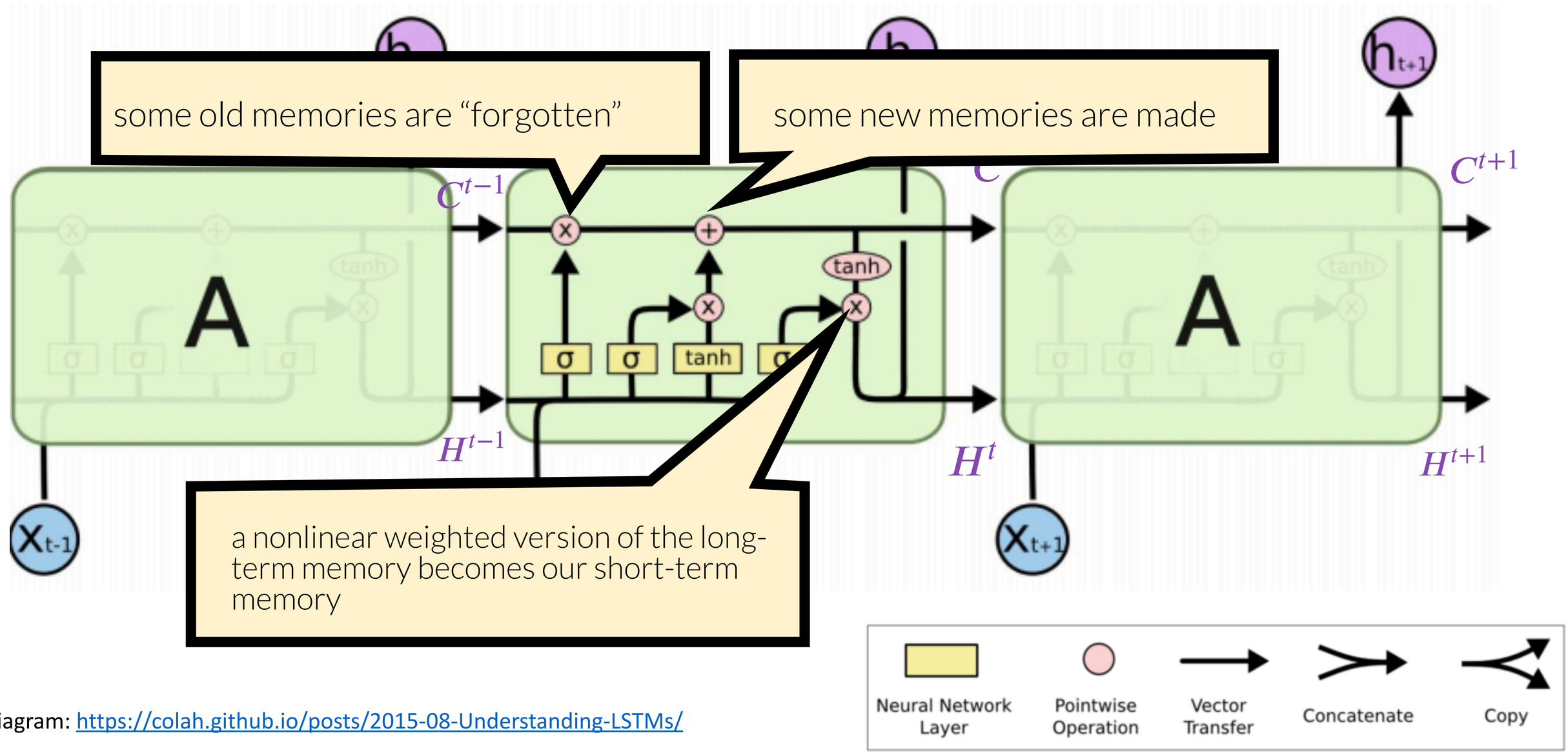


Diagram: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM Architecture

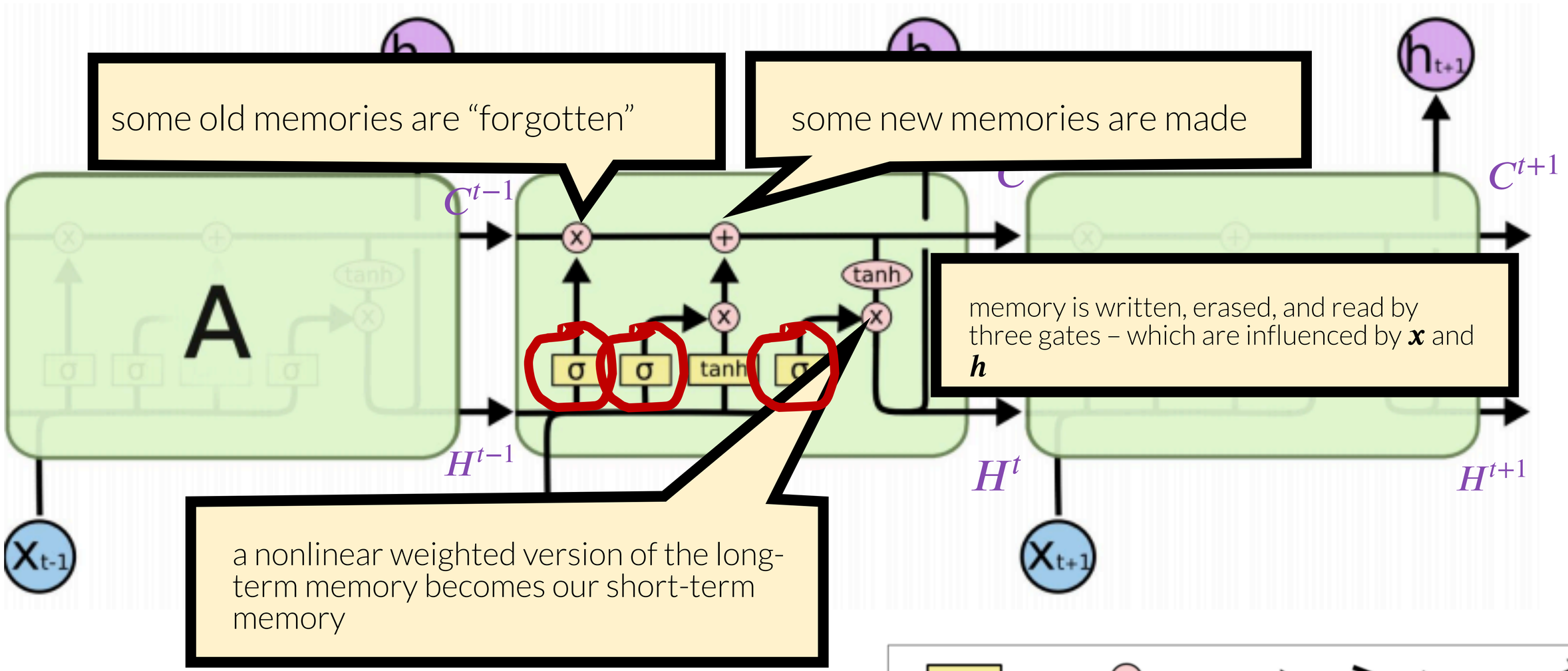


Diagram: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Total Recall?

It is still possible for LSTMs to suffer from vanishing/exploding gradients, but it's way less likely than with vanilla RNNs:

- If an RNN wishes to preserve info over long contexts, it must delicately find a recurrent weight matrix W_h that isn't too large or small
- LSTMs have 3 separate mechanism that adjust the flow of information (e.g., forget gate, if turned off, will preserve all info)

LSTM Summary

Strengths

- Almost always outperforms RNNs
- Captures long-range dependencies very well

Issues?

- More weights to learn, higher demand on training data
- Can still suffer from vanishing/exploding gradients

LSTM Trained on Harry Potter

"Sorry," Harry shouted, panicking—"I'll leave those brooms in London, are they?"

"No idea," said Nearly Headless Nick, casting low close by Cedric, carrying the last bit of treacle Charms, from Harry's shoulder, and to answer him the common room perched upon it, four arms held a shining knob from when the spider hadn't felt it seemed. He reached the teams too.

https://medium.com/deep-writing/harry-potter-written-by-artificial-intelligence-8a9431803da6

Bi-LSTM: ELMO

ELMo

General Idea:

- Goal is to obtain highly rich embeddings for each word (unique type)
- Use both directions of context (bi-directional), with increasing abstractions (stacked)
- Linearly combine all abstract representations (hidden layers) and optimize w.r.t. a particular task (e.g., sentiment classification)

ELMo Slides: https://www.slideshare.net/shuntaroy/a-review-of-deep-contextualized-word-representations-peters-2018

ELMo

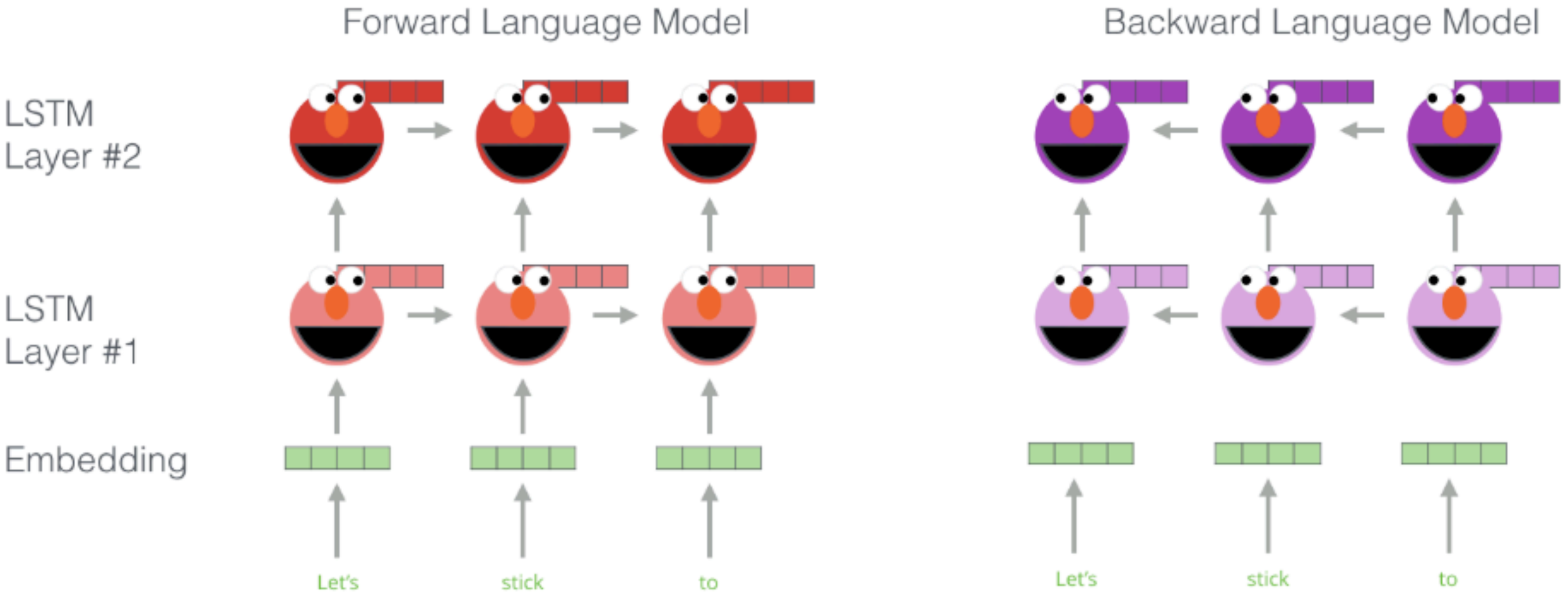


Illustration: http://jalammar.github.io/illustrated-bert/

Embedding of “stick” in “Let’s stick to” - Step #2

1- Concatenate hidden layers

2- Multiply each vector by a weight based on the task

x s_2

x s_1

x s_0

3- Sum the (now weighted) vectors

ELMo embedding of “stick” for this task in this context

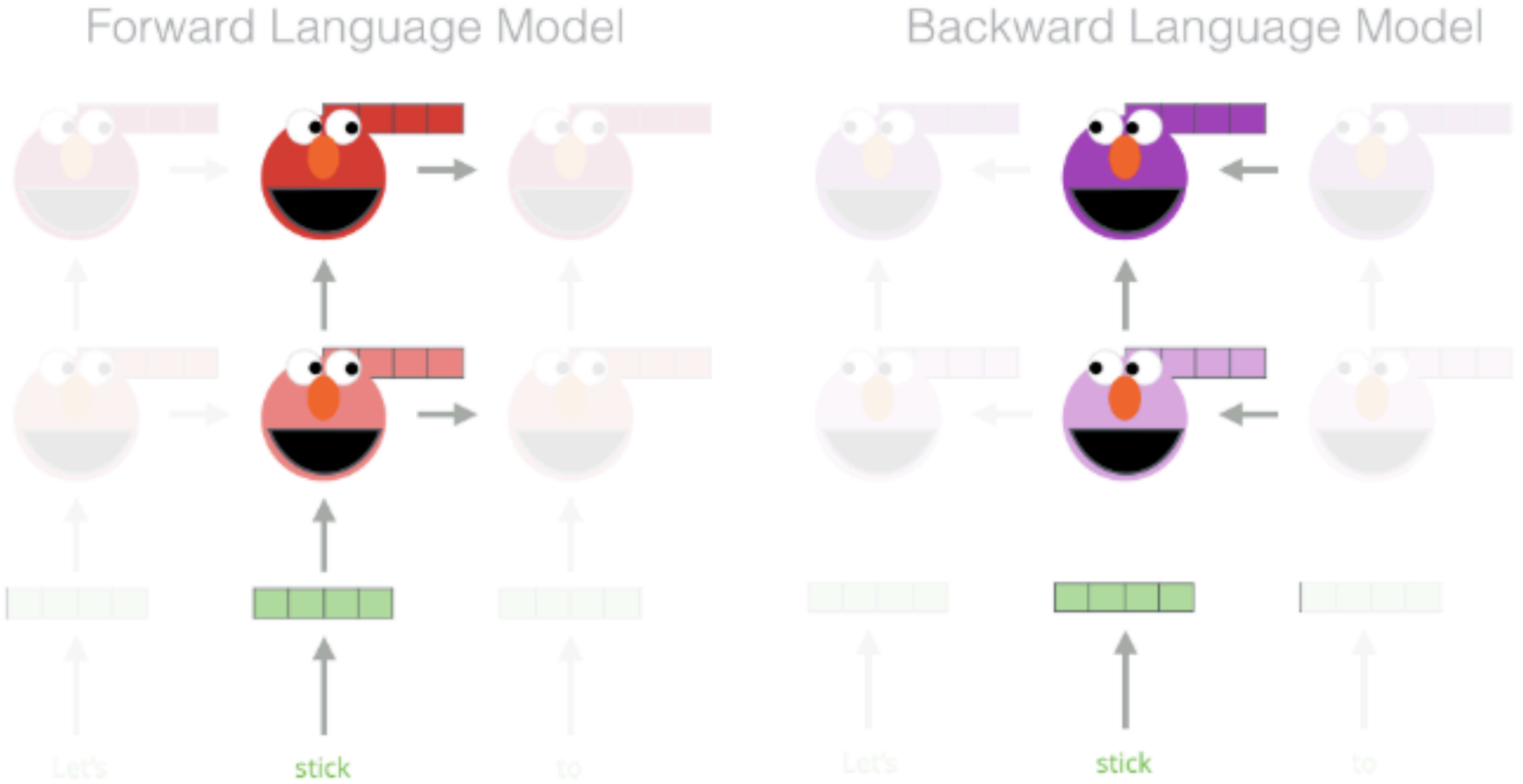


Illustration: http://jalammar.github.io/illustrated-bert/

ELMo

Task	Previous SOTA		Our baseline	ELMo + baseline	Increase (absolute/ relative)
SQuAD	Liu et al. (2017)	84.4	81.1	85.8	4.7 / 24.9%
SNLI	Chen et al. (2017)	88.6	88.0	88.7 ± 0.17	0.7 / 5.8%
SRL	He et al. (2017)	81.7	81.4	84.6	3.2 / 17.2%
Coref	Lee et al. (2017)	67.2	67.2	70.4	3.2 / 9.8%
NER	Peters et al. (2017)	91.93 ± 0.19	90.15	92.22 ± 0.10	2.06 / 21%
SST-5	McCann et al. (2017)	53.7	51.4	54.7 ± 0.5	3.3 / 6.8%

Deep contextualized word representations. Peters et al. NAACL 2018.

ELMo

- ELMo yielded very good contextualized embeddings, which re-defined SOTA when applied to many NLP tasks.
- Main ELMo takeaway: given enough training data, having tons of explicit connections between your vectors is useful (the system can determine how to best use context)

ELMo Slides: https://www.slideshare.net/shuntaroy/a-review-of-deep-contextualized-word-representations-peters-2018

Summary

- Distributed Representations can be:
  - Type-based (“word embeddings”)
  - Token-based (“contextualized representations/embeddings”)
- Type-based models include Bengio’s 2003 and word2vec 2013
- Token-based models include RNNs/LSTMs, which:
  - demonstrated profound results from 2015 onward.
  - can be used for essentially any NLP task.

Fini

Homework for next week

- solve exercises in worksheets from section 2
  - just for yourself; no submission, no grading
- ask questions
  - moodle, tutorial, class