# Natural Language Processing

AI51701/CSE71001

Lecture 8

09/26/2023

Instructor: Taehwan Kim

# Announcements

- Due to attending a conference, we will have **no class on Oct. 5** and instead **a substitute class on Oct. 17 at 4:00pm**

# Announcements

- Final project proposal
  - Presentation sign-up sheet (by 10/8 at 11:59pm, otherwise you will get 0 points):

    https://docs.google.com/spreadsheets/d/1BVijSZ1RUGmt53r32Pkvdk9waz6dejjdgGX3Jj5p_R0/edit?usp=sharing
  - 3–4 pages (in latex, *EMNLP 2023* format), due Oct. 11, 11:59pm
  - Submit **both written proposal and presentation slides** via BlackBoard (Assignments>Final project proposal)

# Project Proposal

- ❑ Find a relevant (key) research paper for your topic
- ❑ Write a summary of that research paper and what you took away from it as key ideas that you hope to use
- ❑ Write what you plan to work on and how you can innovate in your final project work
- ❑ Required components:
  - A project plan, relevant existing literature, the kind(s) of models you will use/explore; the **data** you will use (and how it is obtained), and how you will **evaluate** success
  - Title, team members, and your planned contributions
- ❑ Grading based on both your written proposal and presentation

# Deep Learning Classification Task: Named Entity Recognition (NER)

- The task: **find** and **classify** names in text, for example:

  Last night , Paris Hilton wowed in a sequin gown .

  PER PER

  Samuel Quinn was arrested in the Hilton Hotel in Paris in April 1989 .

  PER PER LOC LOC LOC DATE DATE

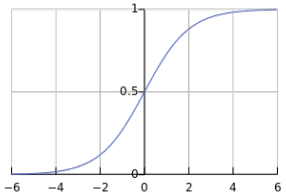
- Possible uses:
  - Tracking mentions of particular entities in documents
  - For question answering, answers are usually named entities
  - Relating sentiment analysis to the entity under discussion
- Often followed by Entity Linking/Canonicalization into a Knowledge Base such as Wikidata

# NER: Binary classification for center word being location

- We do supervised training and want high score if it's a location

$$J_t(\theta) = \sigma(s) = \frac{1}{1 + e^{-s}}$$

predicted model probability of class




$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{x} \quad \text{(input)}$$




*f* = Some element-wise non-linear function, e.g., logistic, tanh, ReLU

From Stanford CS224n course

# Stochastic Gradient Descent

- Update equation: $\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$

$\alpha$ = *step size* or *learning rate*

  - I.e., for each parameter: $\theta_j^{new} = \theta_j^{old} - \alpha \frac{\partial J(\theta)}{\partial \theta_j^{old}}$
  - In deep learning, $\theta$ includes the data representation (e.g., word vectors) too!
- How can we compute $\nabla_\theta J(\theta)$?
  - By hand
  - Algorithmically: the backpropagation algorithm

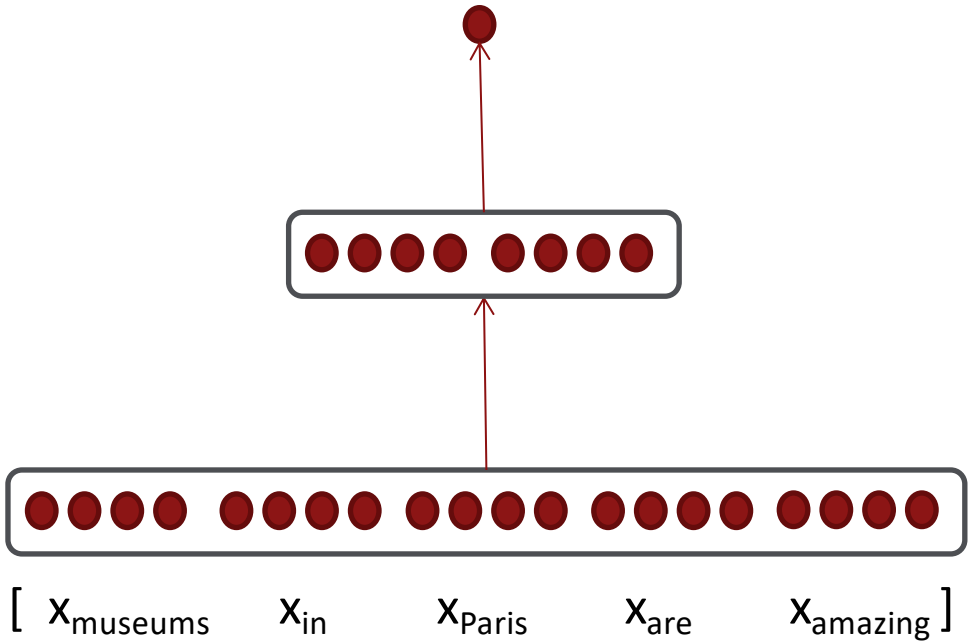
# Back to our Neural Net!

- Let's find $\frac{\partial s}{\partial \boldsymbol{b}}$
- Really, we care about the gradient of the loss $J_t$ but we will compute the gradient of the score for simplicity

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{x} \quad \text{(input)}$$



# 1. Break up equations into simple pieces

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{x} \quad \text{(input)}$$



$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad \text{(input)}$$

- Carefully define your variables and keep track of their dimensionality!

# 2. Apply the chain rule

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad (\text{input})$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

# 3. Write out the Jacobians

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad (\text{input})$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

$$= \boldsymbol{u}^T \operatorname{diag}(f'(\boldsymbol{z}))\boldsymbol{I}$$

$$= \boldsymbol{u}^T \odot f'(\boldsymbol{z})$$

Useful Jacobians from previous slide

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T \boldsymbol{h}) = \boldsymbol{h}^T$$

$$\frac{\partial}{\partial \boldsymbol{z}}(f(\boldsymbol{z})) = \operatorname{diag}(f'(\boldsymbol{z}))$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{I}$$

$\odot$ = Hadamard product = element-wise multiplication of 2 vectors to give vector

# Backpropagation

- We've almost shown you backpropagation
  - It's taking derivatives and using the (generalized, multivariate, or matrix) chain rule
- Other trick:
  - We **re-use** derivatives computed for higher layers in computing derivatives for lower layers to minimize computation

# Computation Graphs and Backpropagation

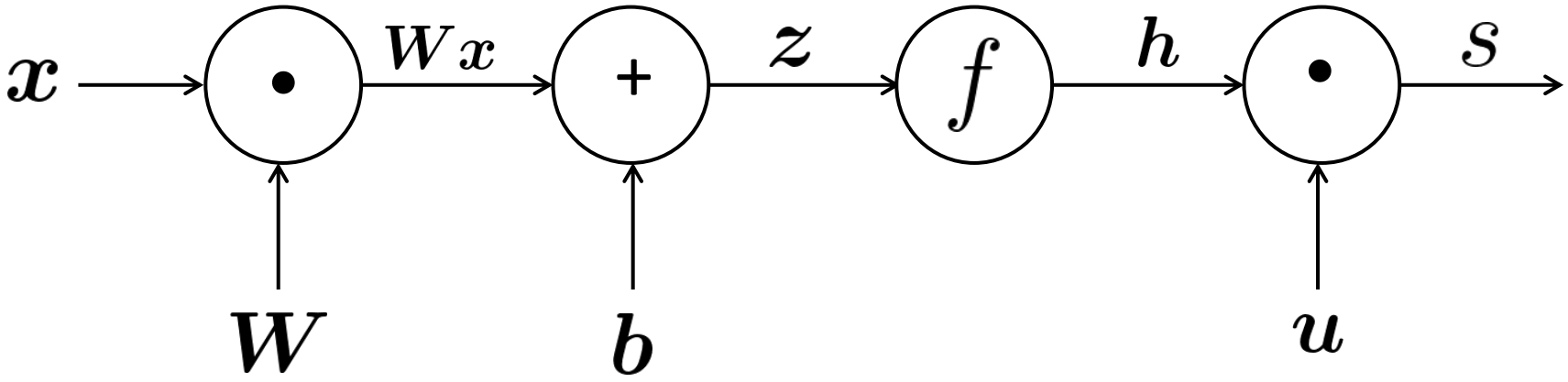
- Software represents our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations
  - Edges pass along result of the operation

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad (\text{input})$$

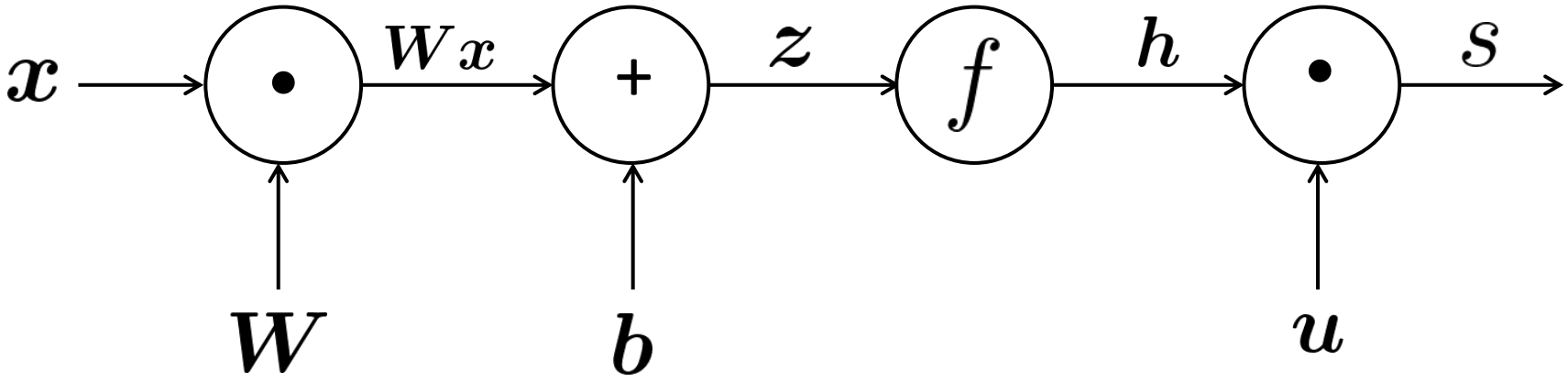


# Computation Graphs and Backpropagation

- Software represents our neural net equations as a graph
  - Source nodes: inputs
  - Interior r
  - Edges pa

"Forward Propagation"

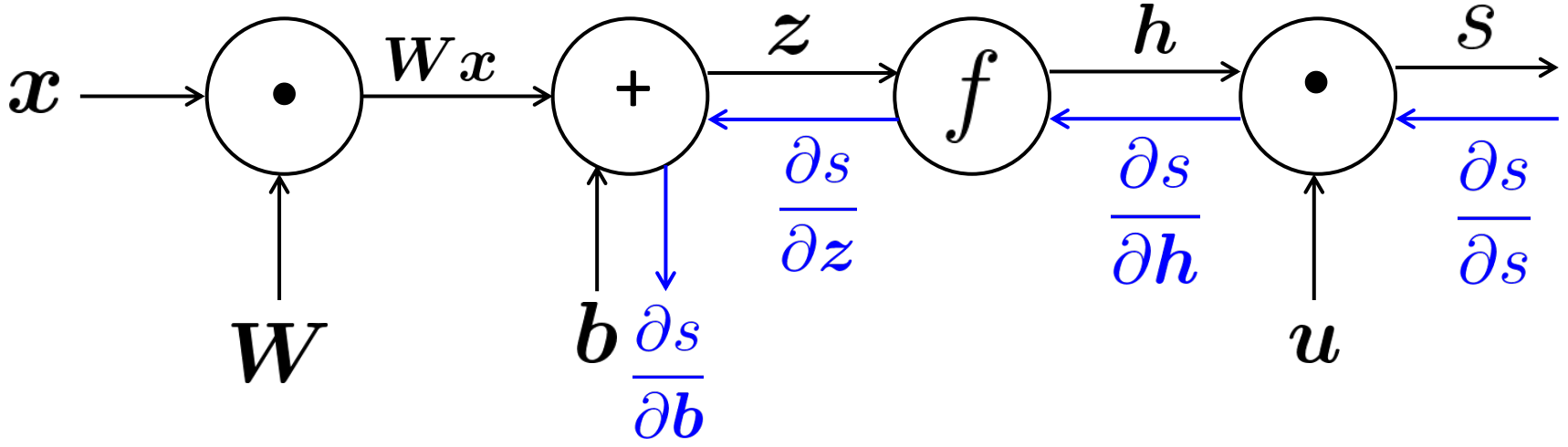
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



# Computation Graphs and Backpropagation

- Then go backwards along edges
  - Pass along **gradients**

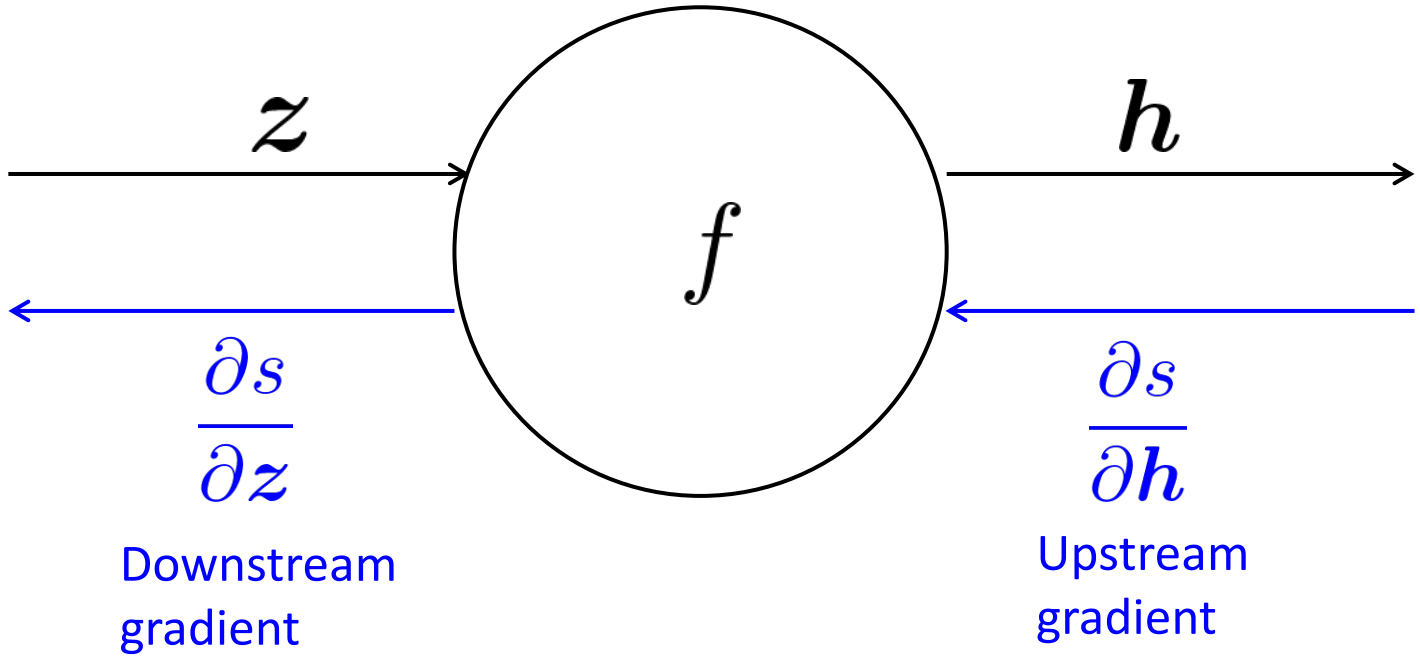
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



# Backpropagation: Single Node

- Node receives an "upstream gradient"
- Goal is to pass on the correct "downstream gradient"

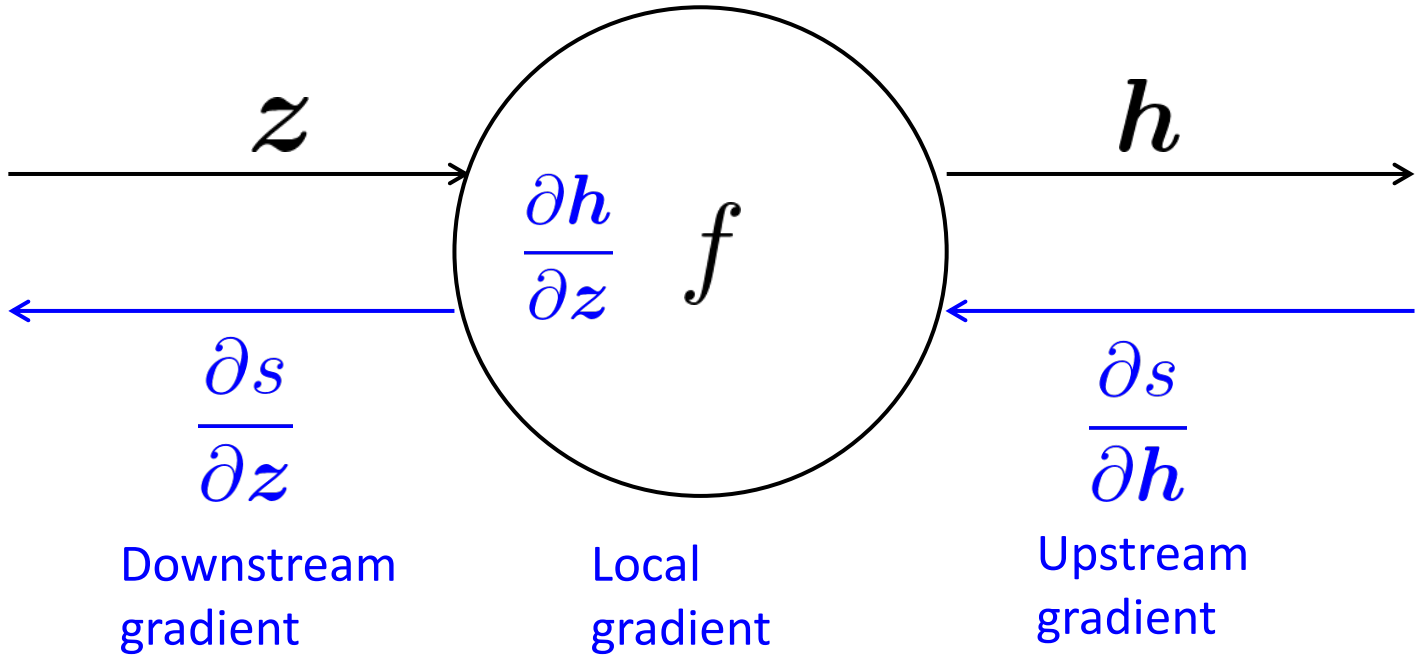
$$\boldsymbol{h} = f(\boldsymbol{z})$$



# Backpropagation: Single Node

- Each node has a local gradient
  - The gradient of its output with respect to its input

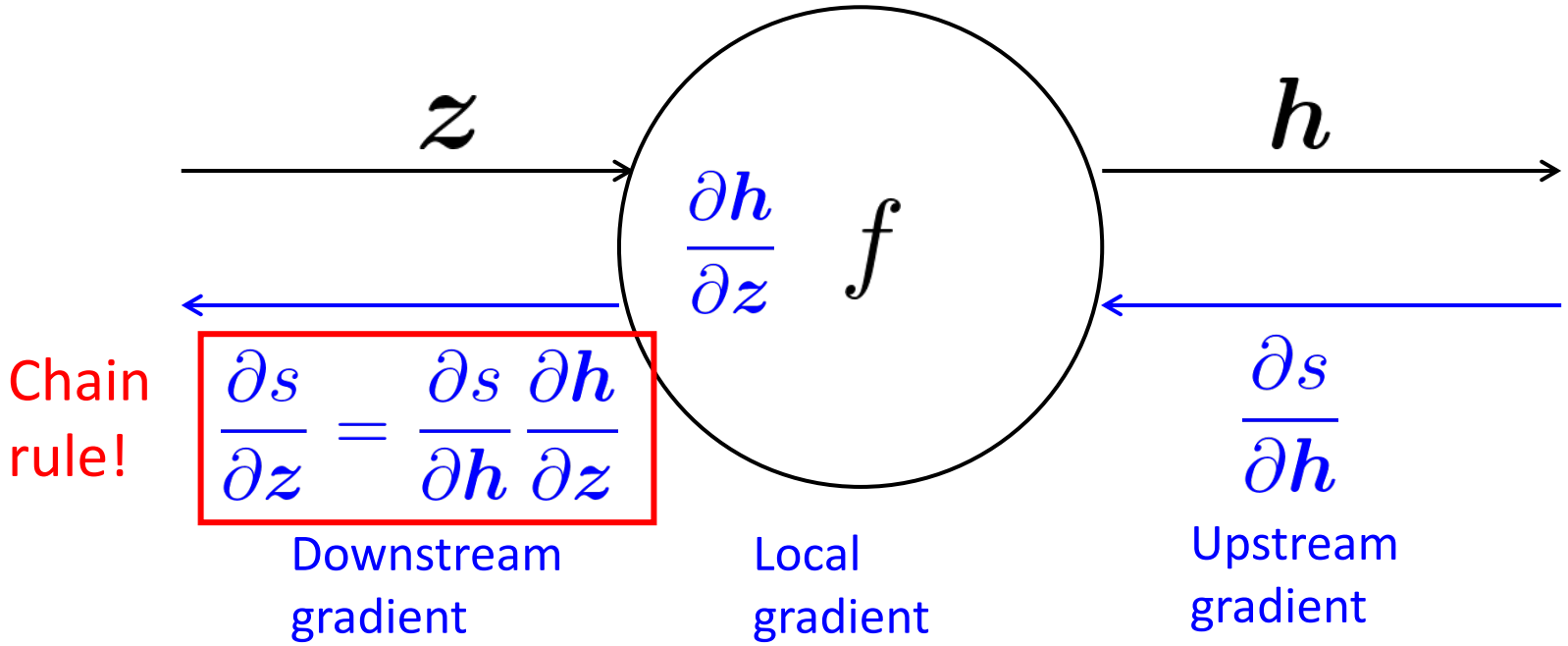
$$\boldsymbol{h} = f(\boldsymbol{z})$$



# Backpropagation: Single Node

- Each node has a local gradient
  - The gradient of its output with respect to its input

$$\boldsymbol{h} = f(\boldsymbol{z})$$

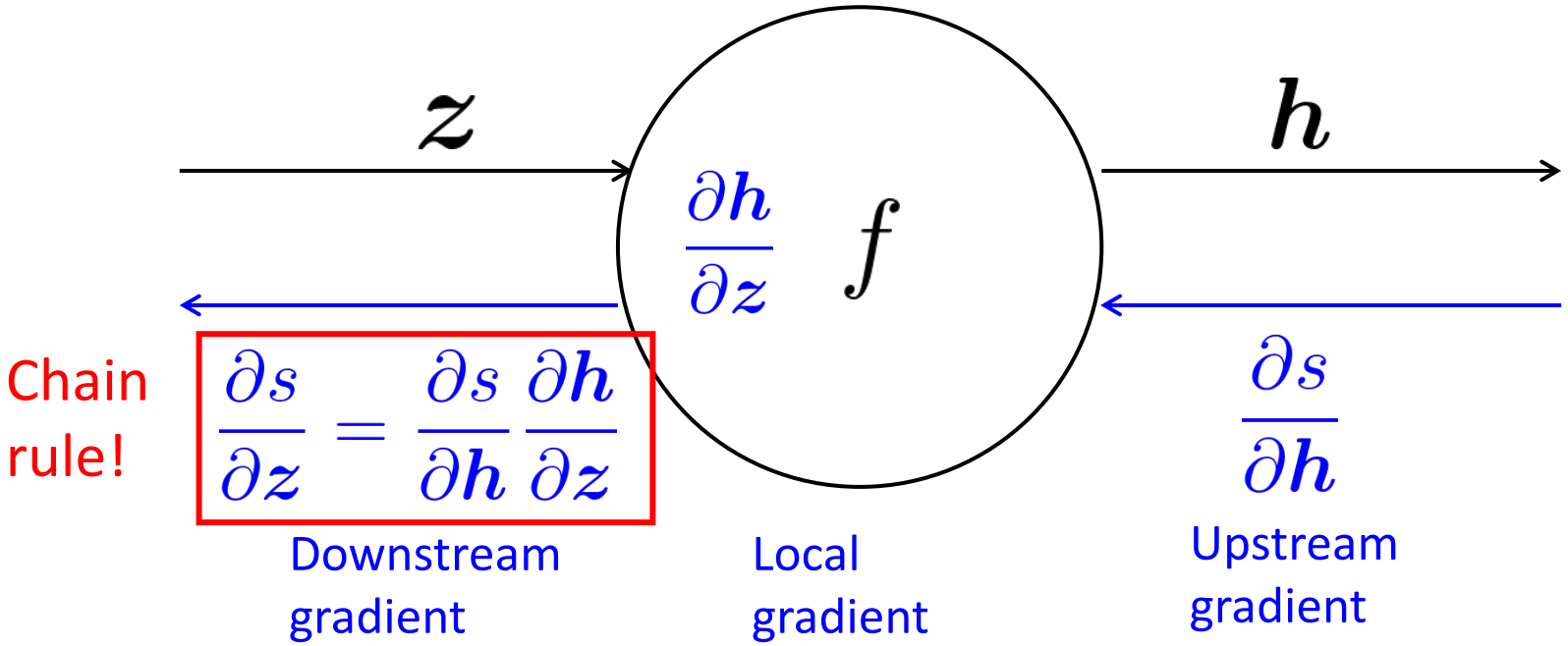


# Backpropagation: Single Node

- Each node has a local gradient
  - The gradient of its output with respect to its input

$$\boldsymbol{h} = f(\boldsymbol{z})$$

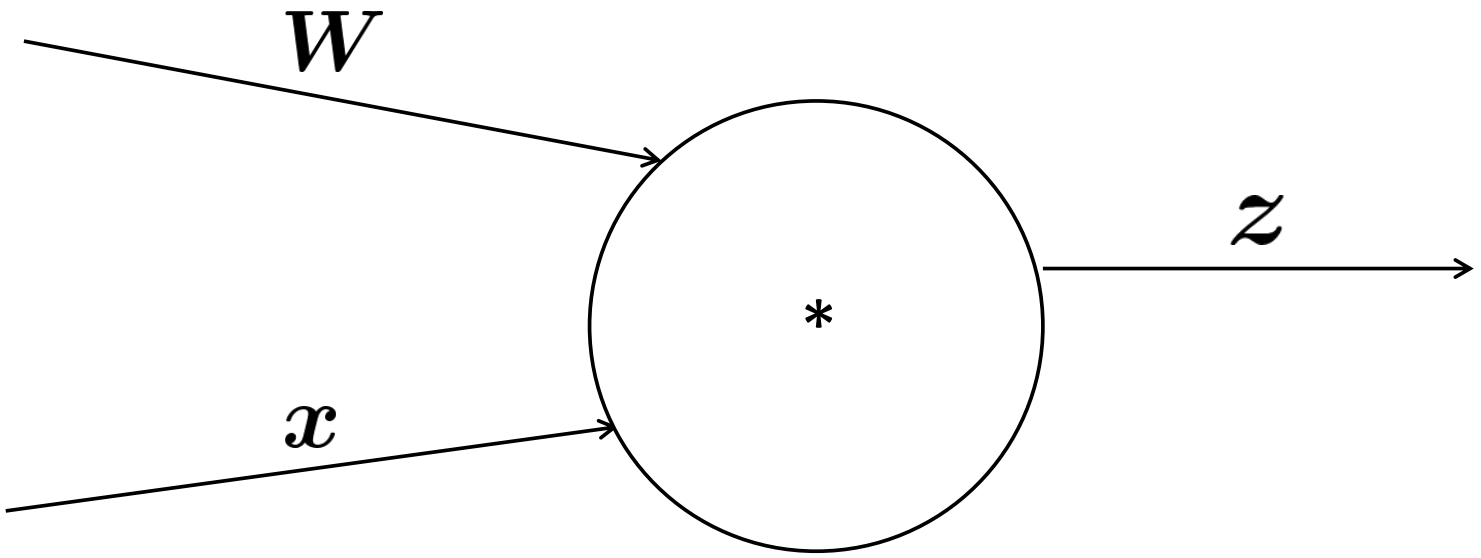
- [downstream gradient] = [upstream gradient] x [local gradient]



# Backpropagation: Single Node

- What about nodes with multiple inputs?

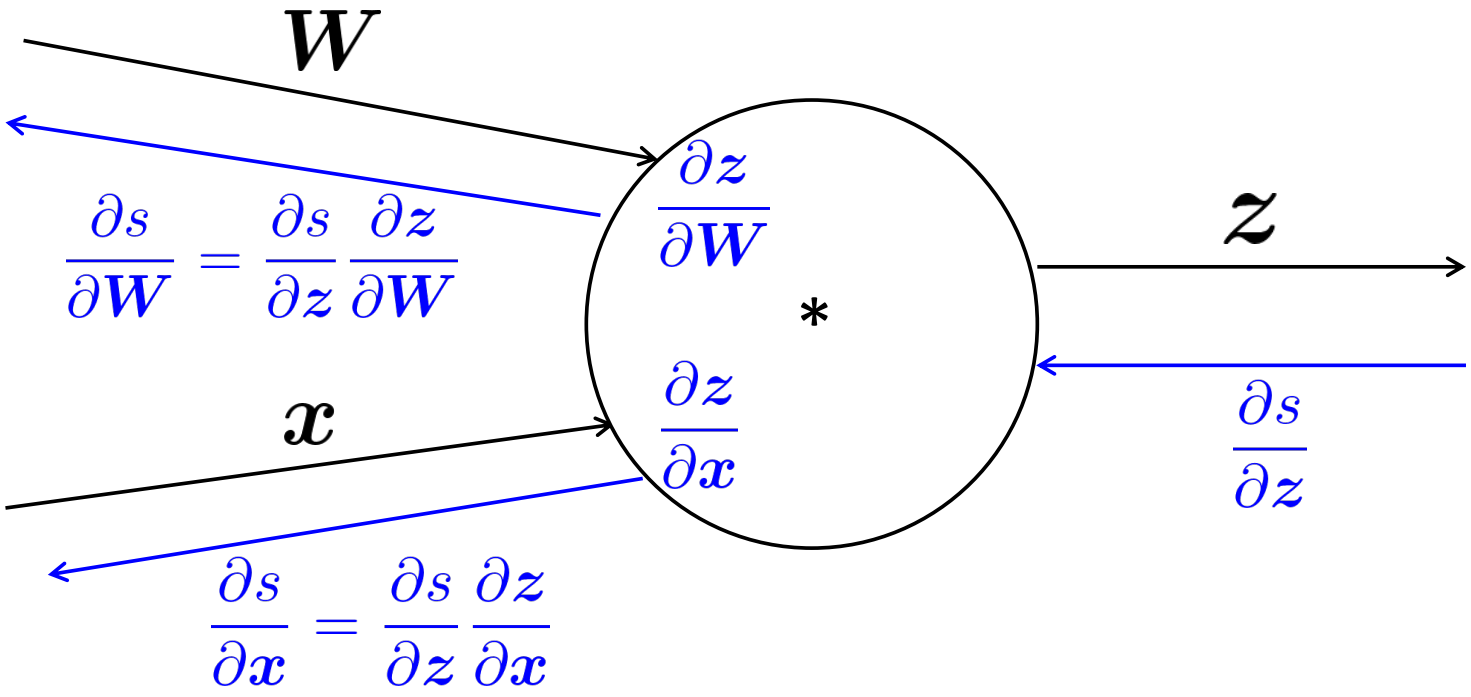
$$z = Wx$$



# Backpropagation: Single Node

- Multiple inputs → multiple local gradients

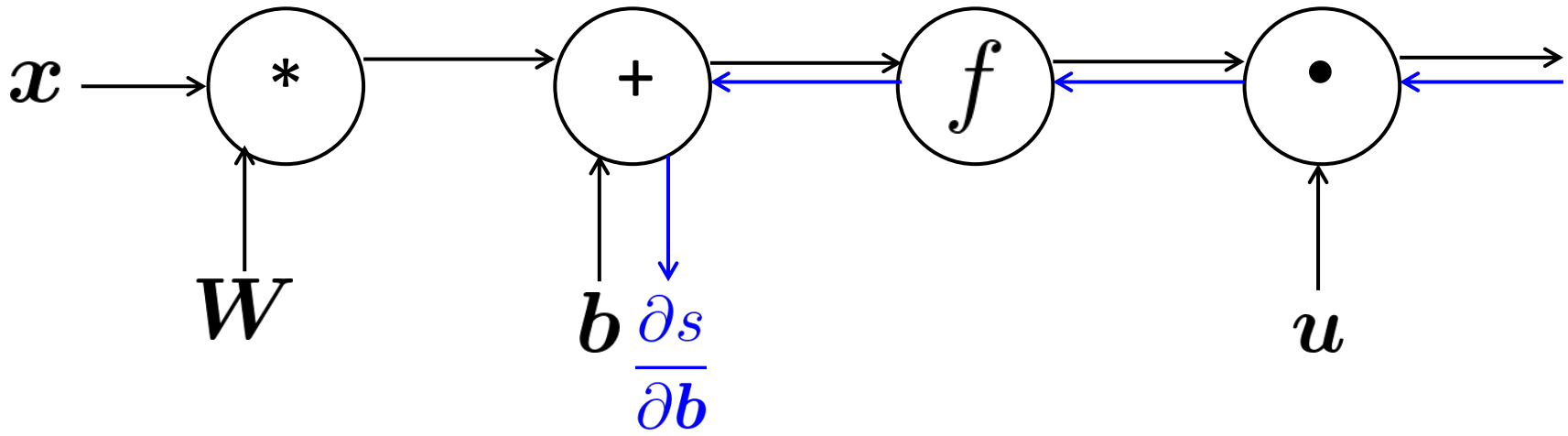
$$z = Wx$$



# Efficiency: compute all gradients at once

- Incorrect way of doing backprop:
  - First compute $\frac{\partial s}{\partial \boldsymbol{b}}$

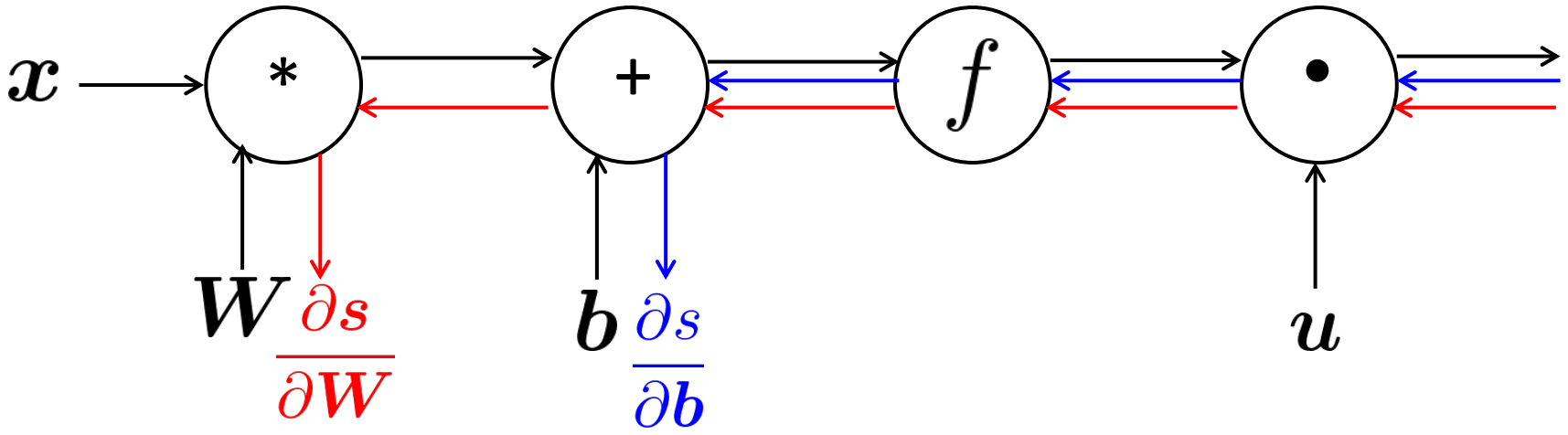
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



# Efficiency: compute all gradients at once

- Incorrect way of doing backprop:
  - First compute $\frac{\partial s}{\partial \boldsymbol{b}}$
  - Then independently compute $\frac{\partial \boldsymbol{s}}{\partial \boldsymbol{W}}$
  - Duplicated computation!

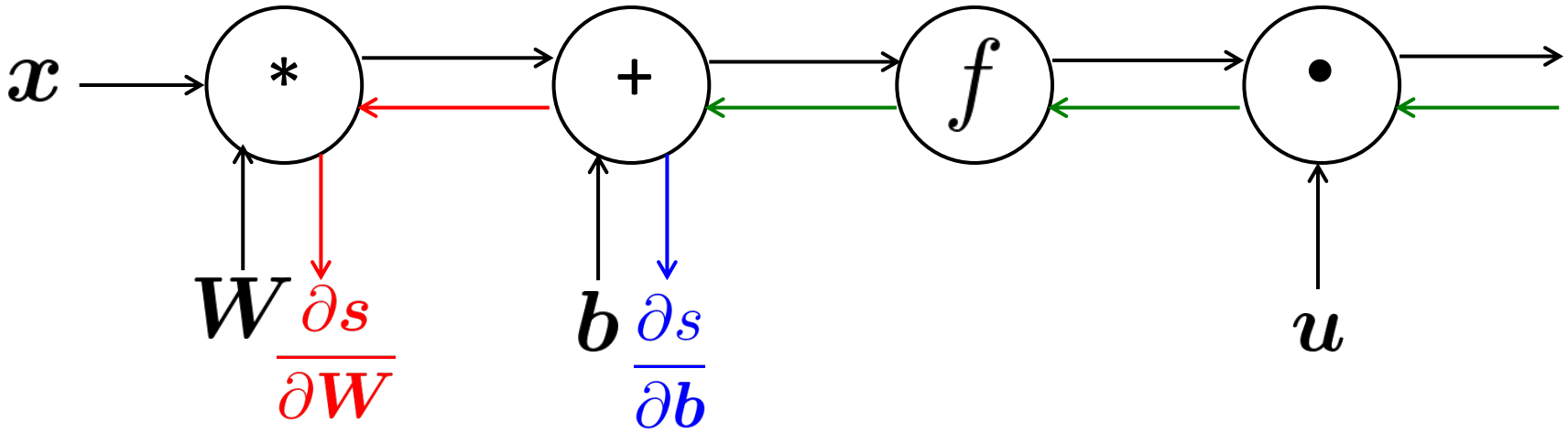
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



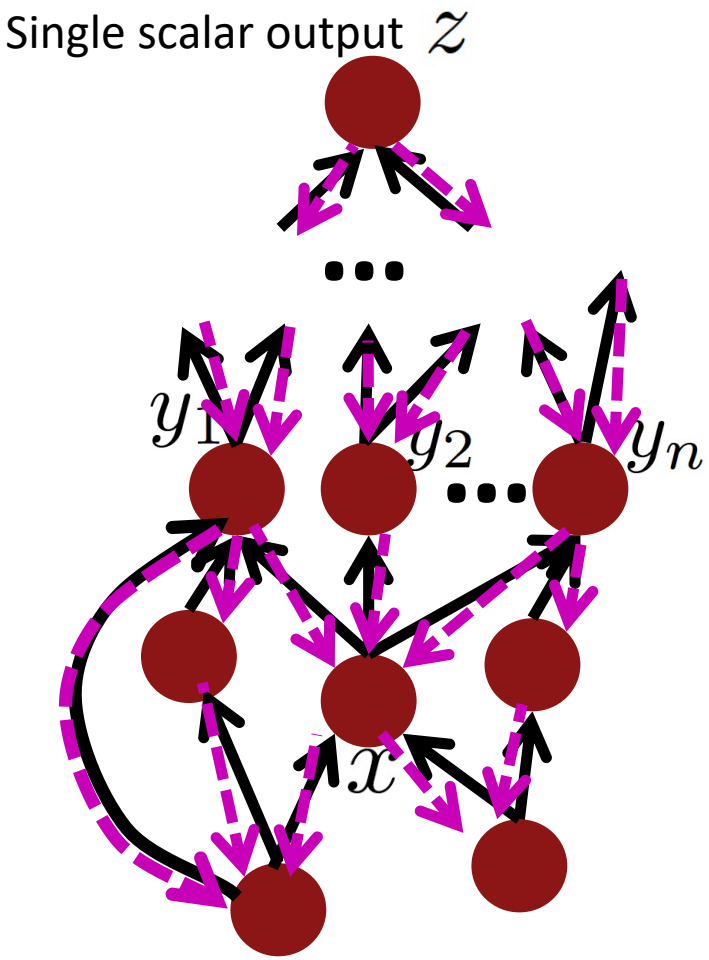
# Efficiency: compute all gradients at once

- Correct way:
  - Compute all the gradients at once
  - Analogous to using $\boldsymbol{\delta}$ when we computed gradients by hand

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



# Back-Prop in General Computation Graph




1. Fprop: visit nodes in topological sort order
   - Compute value of node given predecessors
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
   Compute gradient wrt each node using gradient wrt successors

$\{y_1, y_2, \ldots y_n\}$ = successors of $x$

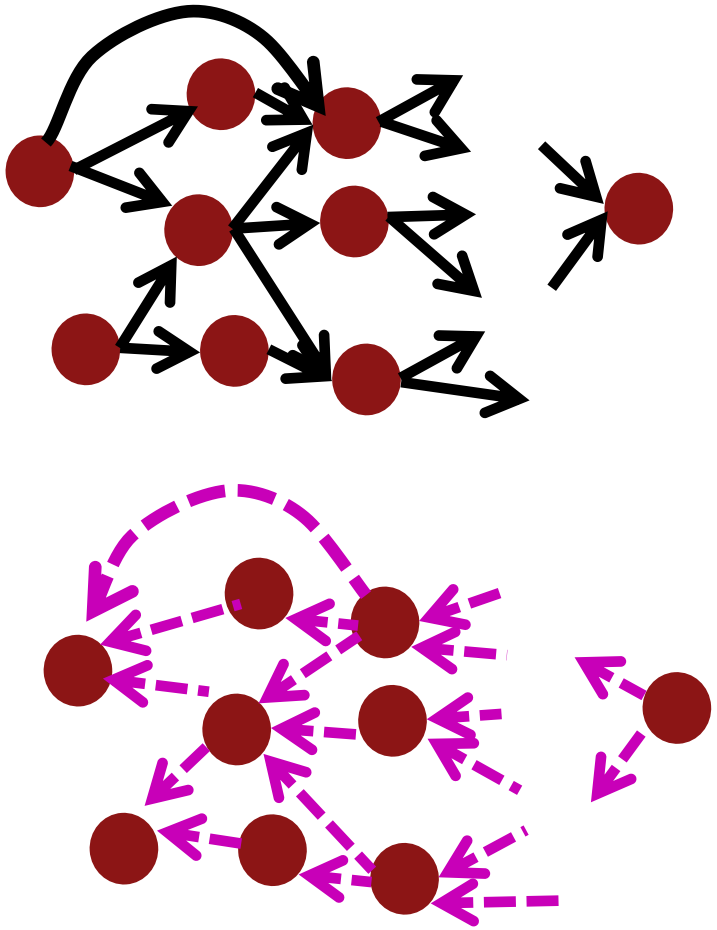
$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Done correctly, big O() complexity of fprop and bprop is **the same**

In general, our nets have regular layer-structure and so we can use matrices and Jacobians…

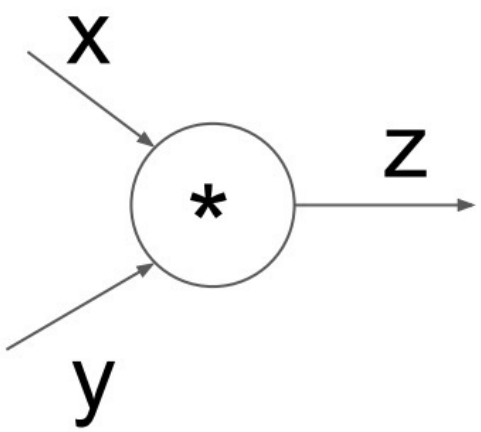
From Stanford CS224n course

# Automatic Differentiation



- The gradient computation can be automatically inferred from the symbolic expression of the fprop
- Each node type needs to know how to compute its output and how to compute the gradient w.r.t. its inputs given the gradient w.r.t. its output
- Modern DL frameworks (PyTorch, Tensorflow, etc.) do backpropagation for you but mainly leave layer/node writer to hand-calculate the local derivative

# Implementation: forward/backward API



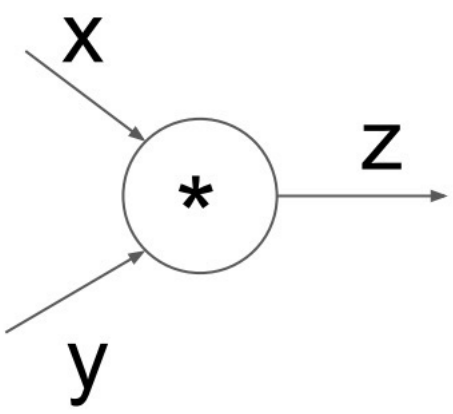

(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

$\frac{\partial L}{\partial z}$

$\frac{\partial L}{\partial x}$

# Implementation: forward/backward API




(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

# Manual Gradient checking: Numeric Gradient

- For small h (≈ 1e-4),

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

- Easy to implement correctly
- But approximate and **very** slow:
  - You have to recompute *f* for **every parameter** of our model
- Useful for checking your implementation
  - In the old days, we hand-wrote everything, doing this everywhere was the key test
  - Now much less needed; you can use it to check layers are correctly implemented

# Summary

- Backpropagation: recursively (and hence efficiently) apply the chain rule along computation graph
  - [downstream gradient] = [upstream gradient] x [local gradient]

- Forward pass: compute results of operations and save intermediate values
- Backward pass: apply chain rule to compute gradients

# Why learn all these details about gradients?

- Modern deep learning frameworks compute gradients for you!
  - There will a lecture for PyTorch soon

- But why take a class on compilers or systems when they are implemented for you?
  - Understanding what is going on under the hood is useful!

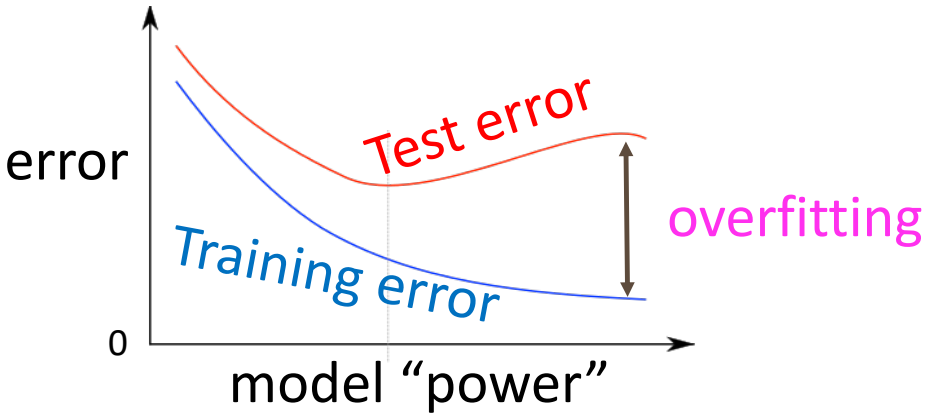
- Backpropagation doesn't always work perfectly out of the box
  - Understanding why is crucial for debugging and improving models
  - See Karpathy article: https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b
  - Example in future lecture: exploding and vanishing gradients

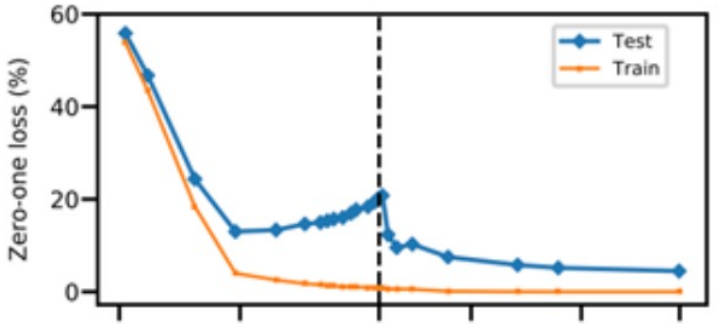
# We have models with many parameters! Regularization!

- A full loss function includes **regularization** over all parameters $\theta$, e.g., L2 regularization:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right) + \lambda \sum_k \theta_k^2$$

- Classic view: Regularization works to prevent **overfitting** when we have a lot of features (or later a very powerful/deep model, etc.)
- Now: Regularization **produces models that generalize well** when we have a "big" model
  - We do not care that our models overfit on the training data, even though they are **hugely** overfit!



# Dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov 2012/JMLR 2014)

- Preventing Feature Co-adaptation = Good Regularization Method! Use it widely!
  - Training time: at each instance of evaluation (in online SGD-training), randomly set 50% of the inputs to each neuron to 0
  - Test time: halve the model weights (now twice as many)

    (Except usually only drop first layer inputs a little (~15%) or not at all)
  - This prevents feature co-adaptation: A feature cannot only be useful in the presence of particular other features
  - In a single layer: A kind of middle-ground between Naïve Bayes (where all feature weights are set independently) and logistic regression models (where weights are set in the context of all others)
  - Can be thought of as a form of model bagging (i.e., like an ensemble model)
  - Nowadays usually thought of as strong, feature-dependent regularizer [Wager, Wang, & Liang 2013]

# “Vectorization”

- E.g., looping over word vectors versus concatenating them all into one large matrix and then multiplying the softmax weights with that matrix:

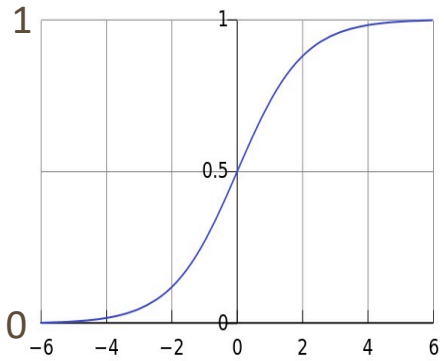
```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

- 1000 loops, best of 3: 639 µs per loop
  10000 loops, best of 3: 53.8 µs per loop <- Now using a single a C x N matrix
- Matrices are awesome!!! Always try to use vectors and matrices rather than for loops!
- The speed gain goes from 1 to 2 orders of magnitude with GPUs!

# Non-linearities, old and new

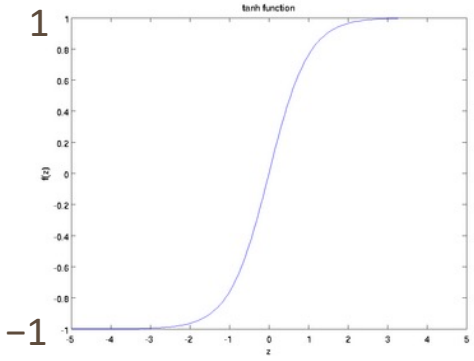
logistic ("sigmoid")

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



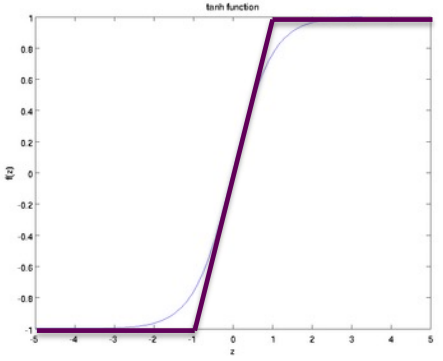

tanh

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$



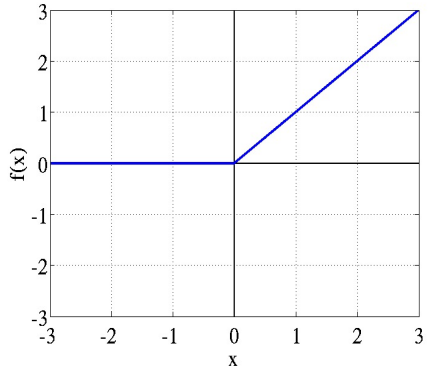

hard tanh

$$\text{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$



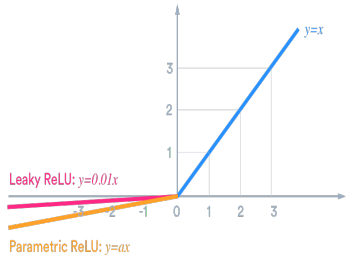

ReLU (Rectified Linear Unit)

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

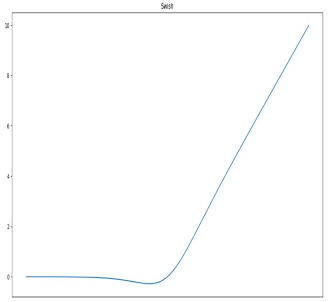



- Both logistic and tanh are still used in various places (e.g., to get a probability), but are no longer the defaults for making deep networks
- For building a deep network, the first thing you should try is ReLU — it trains quickly and performs well due to good gradient backflow

Leaky ReLU / Parametric ReLU




Swish [Ramachandran, Zoph & Le 2017]



# Parameter Initialization

- You normally must initialize weights to small random values (i.e., not zero matrices!)
  - To avoid symmetries that prevent learning/specialization
- Initialize hidden layer biases to 0 and output (or reconstruction) biases to optimal value if weights were 0 (e.g., mean target or inverse sigmoid of mean target)
- Initialize **all other weights** ~ Uniform(*–r, r*), with r chosen so numbers get neither too big or too small (later the need for this is removed with use of layer normalization)
- Xavier initialization has variance inversely proportional to fan-in nin (previous layer size) and fan-out $n_{out}$ (next layer size):

$$\mathrm{Var}(W_i) = \frac{2}{n_{\mathrm{in}} + n_{\mathrm{out}}}$$

# Optimizers

- Usually, plain SGD will work just fine!
  - However, getting good results will often require hand-tuning the learning rate
    - E.g., start it higher and halve it every *k* epochs (passes through full data, **shuffled** or sampled)
- For more complex nets and situations, or just to avoid worry, you often do better with one of a family of more sophisticated "adaptive" optimizers that scale the adjustment to individual parameters by an accumulated gradient.
  - These models give differential per-parameter learning rates
    - Adagrad
    - RMSprop
    - Adam <- A fairly good, safe place to begin in many cases
    - AdamW
    - SparseAdam
    - ...
  - Can just start them with an initial learning rate, around 0.001

# Language Modeling + RNNs

# Language Modeling

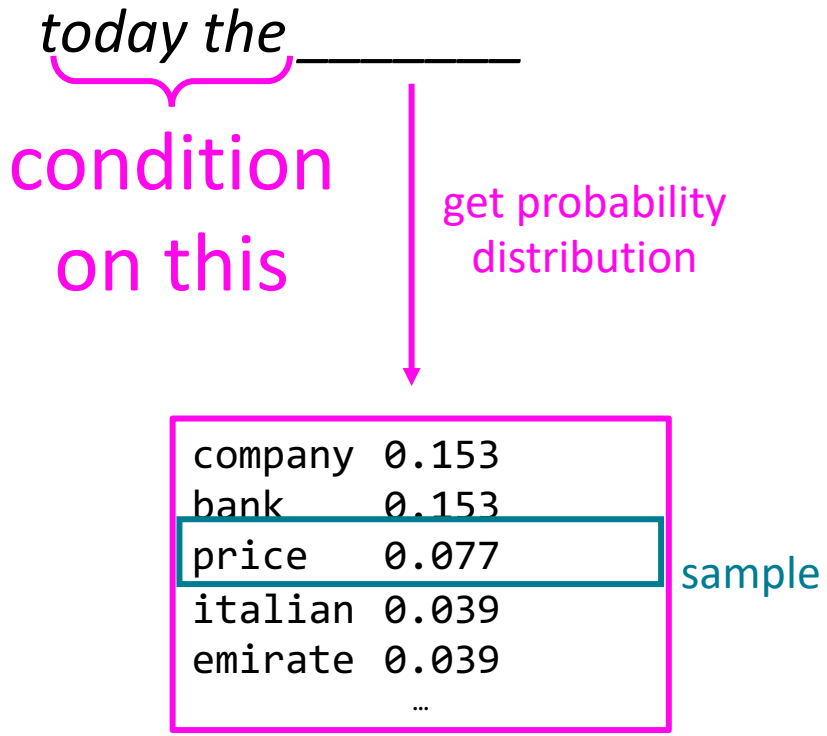
- **Language Modeling** is the task of predicting what word comes next

$$P(w_k \mid w_1, w_2, \ldots, w_{k-1})$$

- A system that compute the probability distribution of the next word is called a **Language Model**

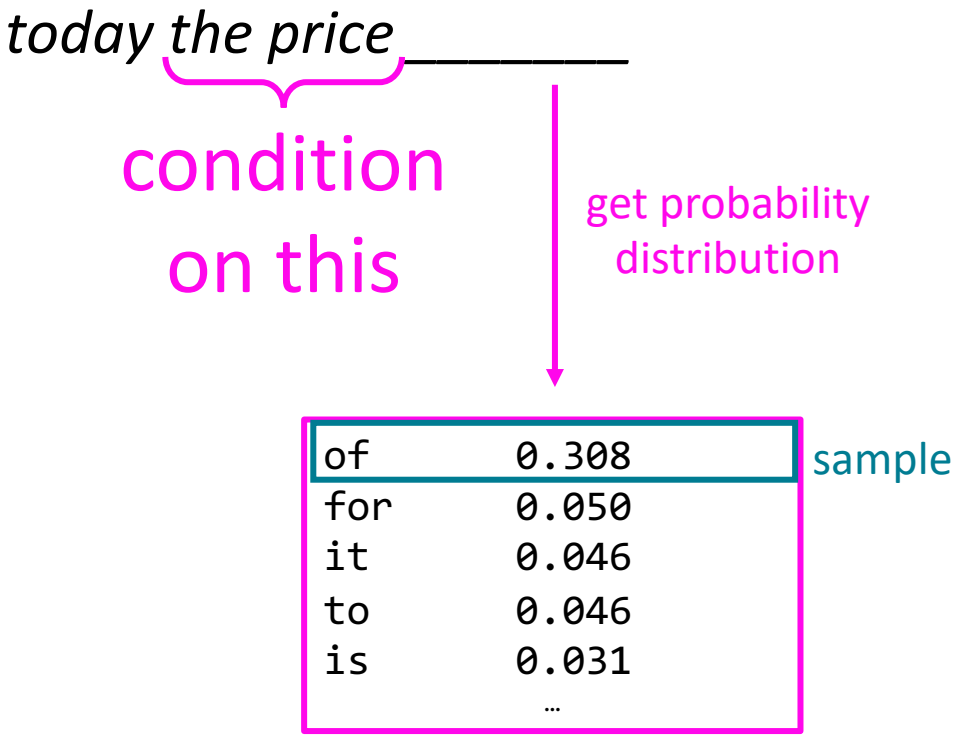
# Generating text with a n-gram Language Model

- You can use a Language Model to generate text



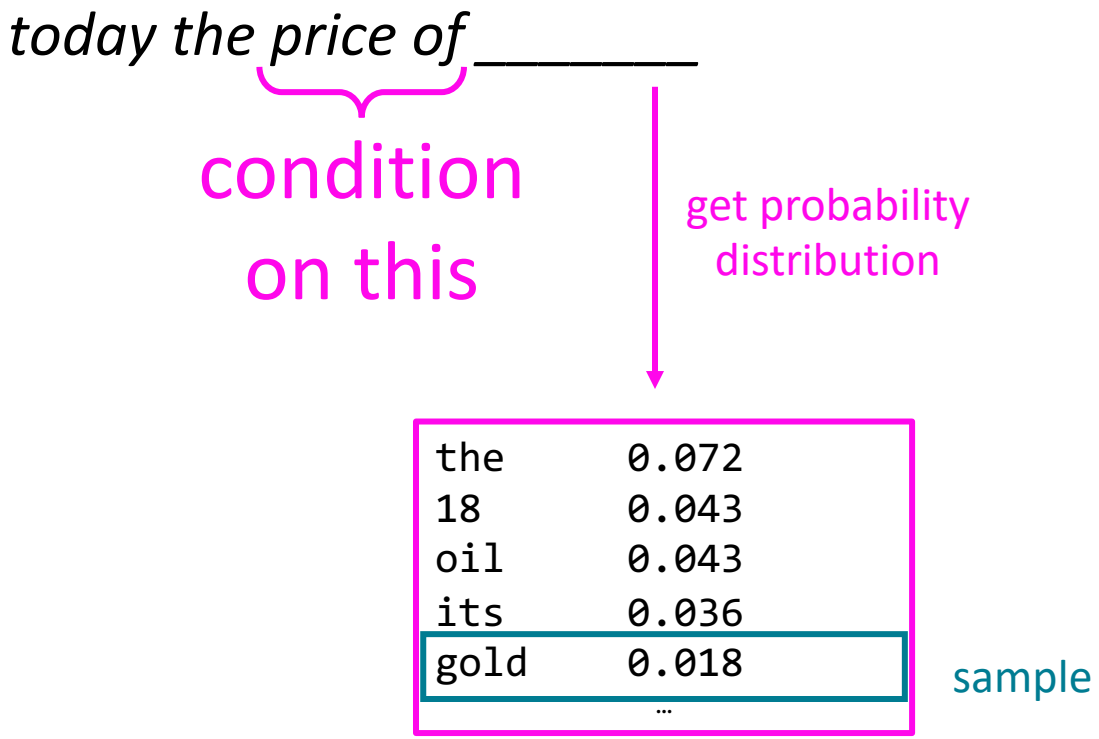
# Generating text with a n-gram Language Model

- You can use a Language Model to generate text



# Generating text with a n-gram Language Model

- You can use a Language Model to generate text



# A fixed-window neural Language Model

~~as the proctor started the clock~~ the students opened their _____

discard

fixed window

# A fixed-window neural Language Model

output distribution

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$

hidden layer

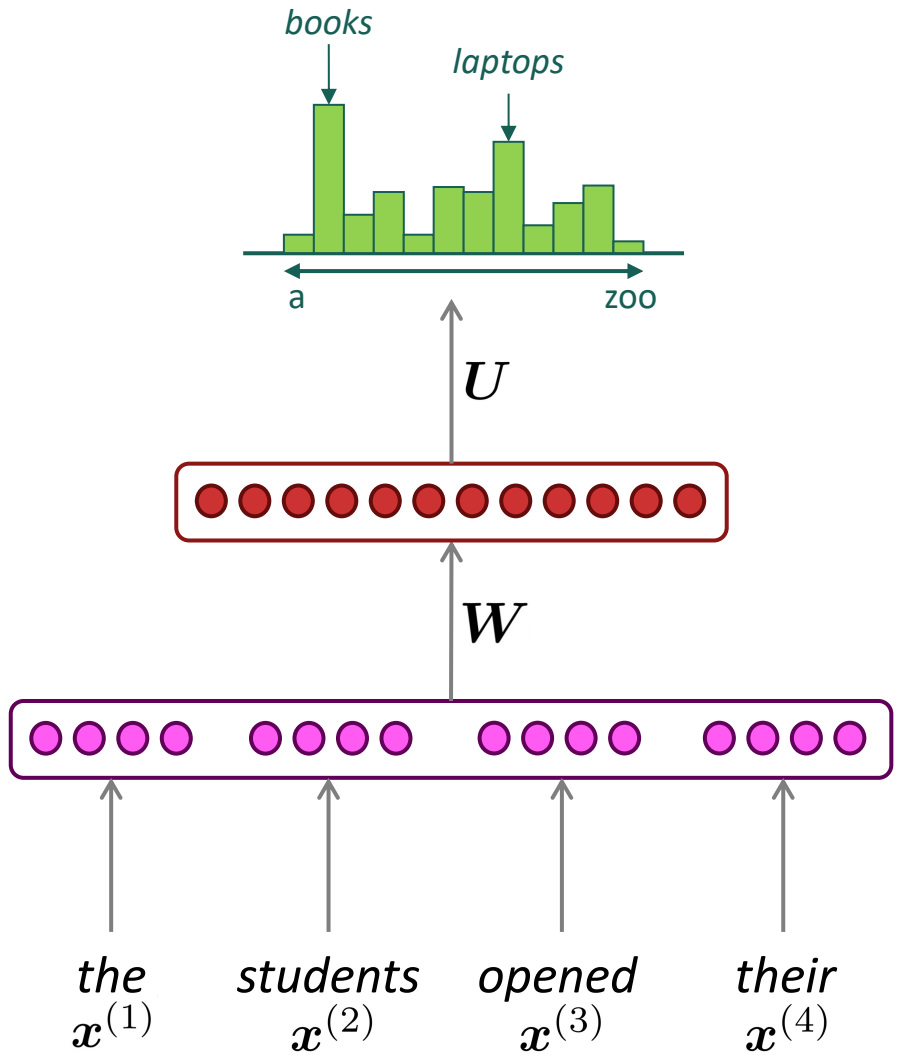
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

concatenated word embeddings

$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

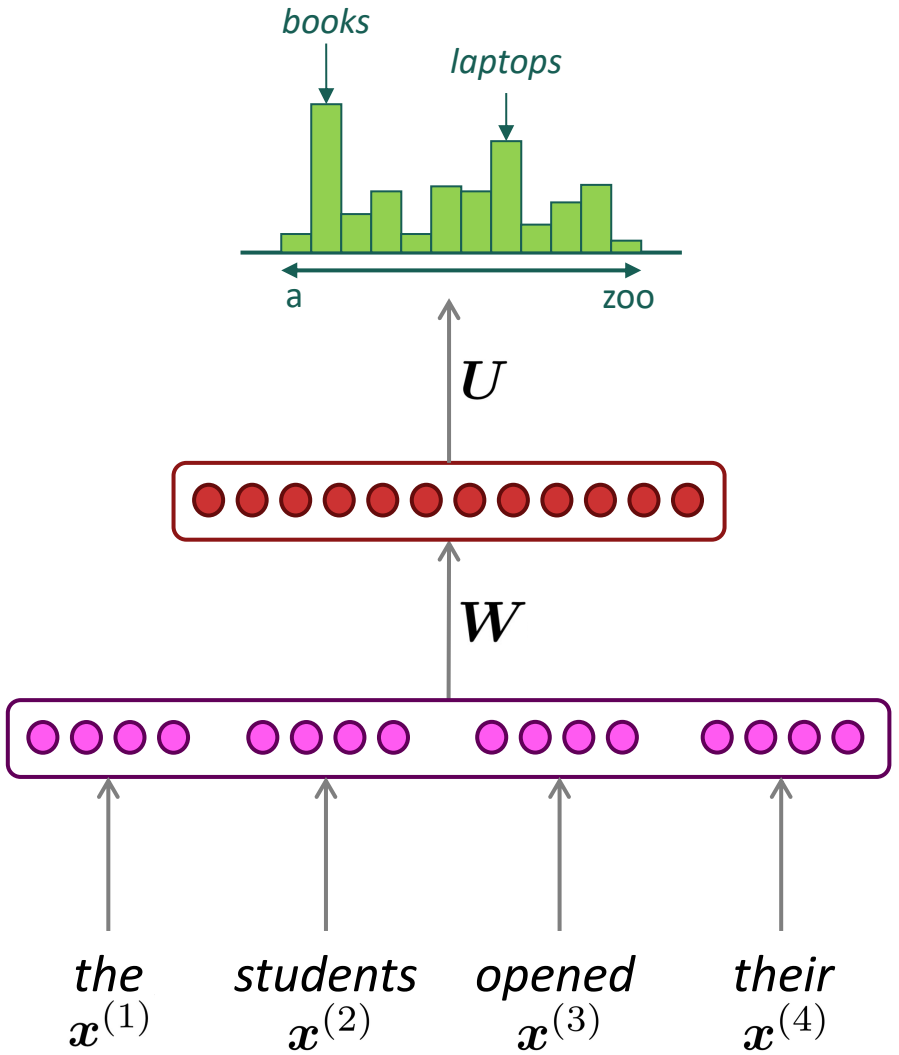
words / one-hot vectors

$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \boldsymbol{x}^{(4)}$$



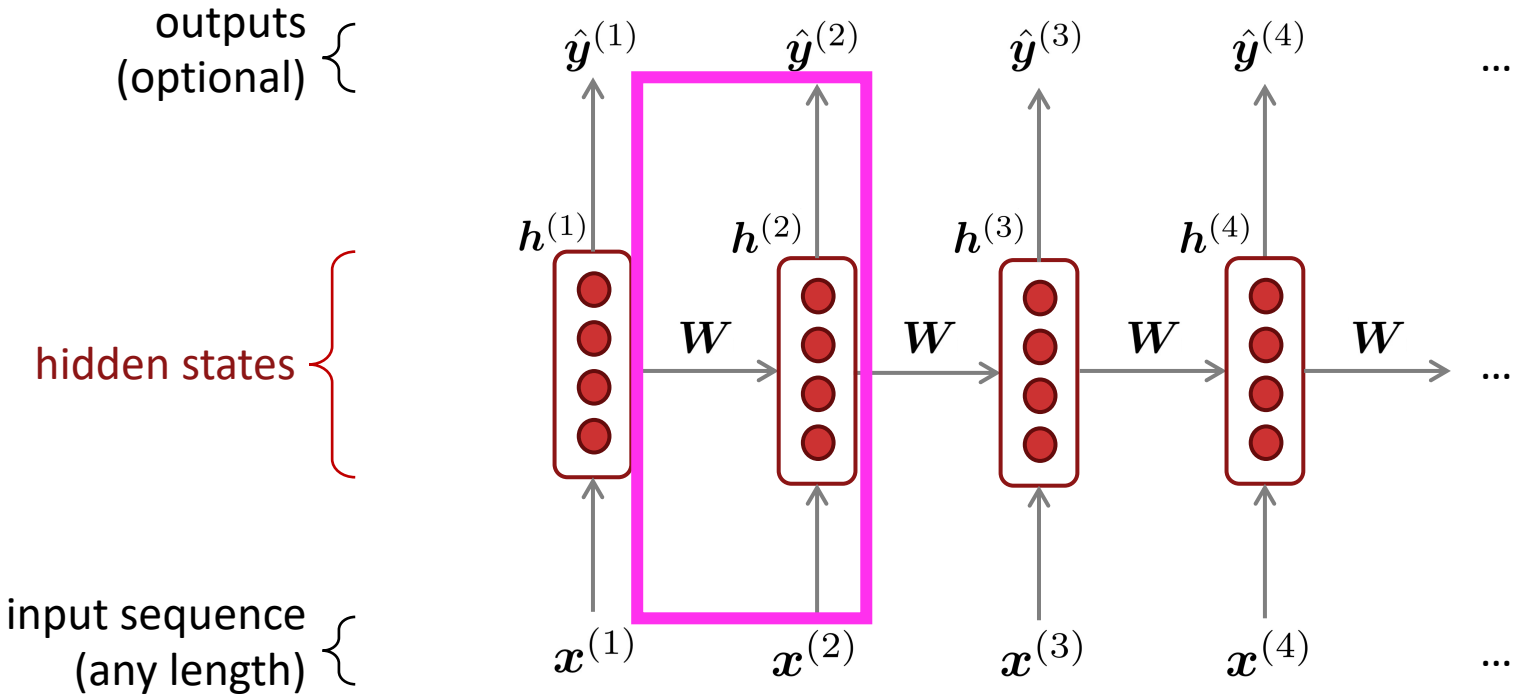
# A fixed-window neural Language Model

- Improvements over n-gram LM:
  - No sparsity problem
  - Don't need to store all observed n-grams
- Remaining problems:
  - Fixed window is too small
  - Enlarging window enlarges $W$
  - Window can never be large enough!
  - $x(0)$ and $x(1)$ are multiplied by completely different weights in $W$. No symmetry in how the inputs are processed.
  - We need a neural architecture that can process **any length input**!



# Recurrent Neural Networks (RNN)

- A family of neural architectures
- Core idea: Apply the same weights $W$ repeatedly



# A Simple RNN Language Model

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

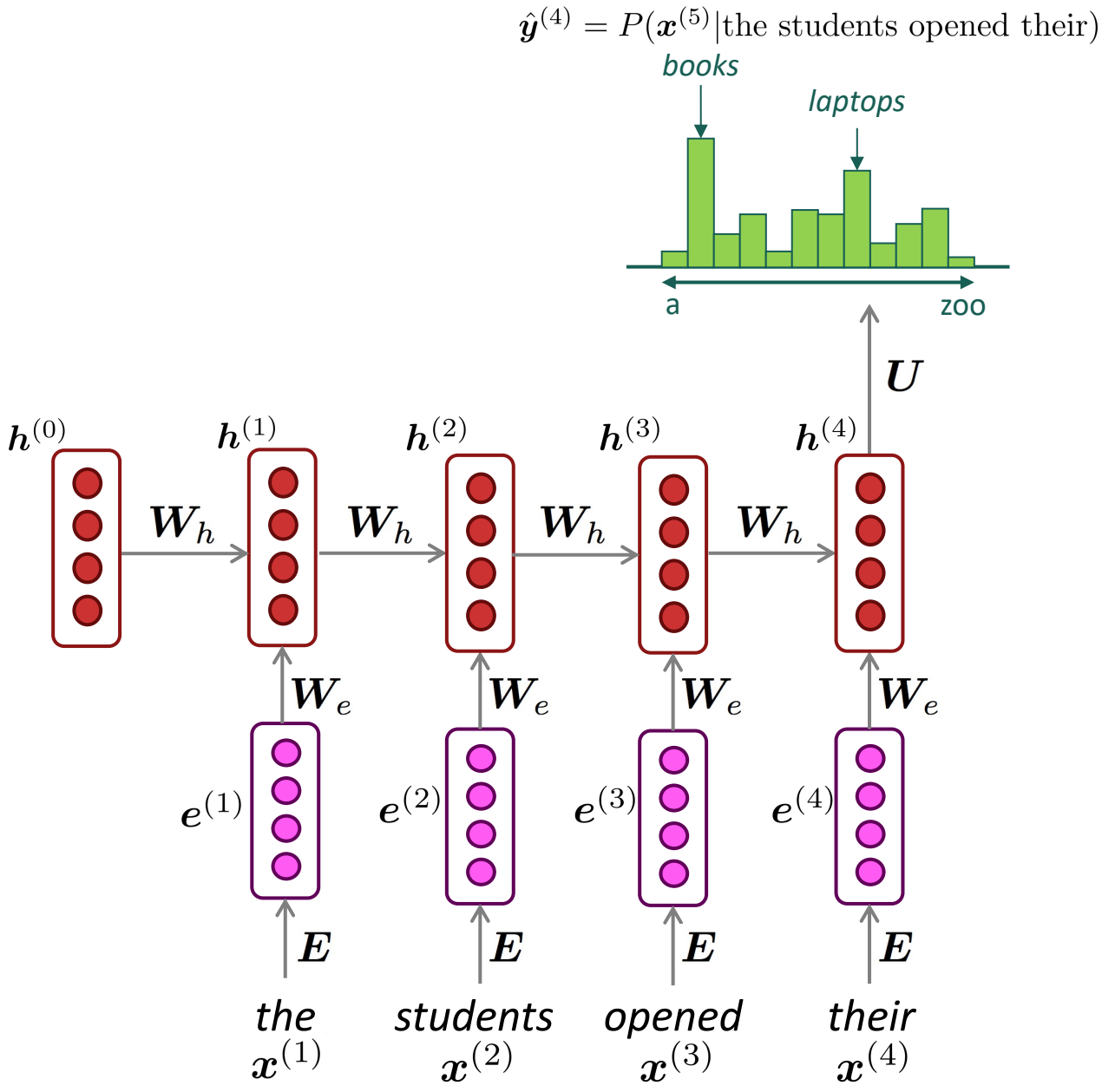
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

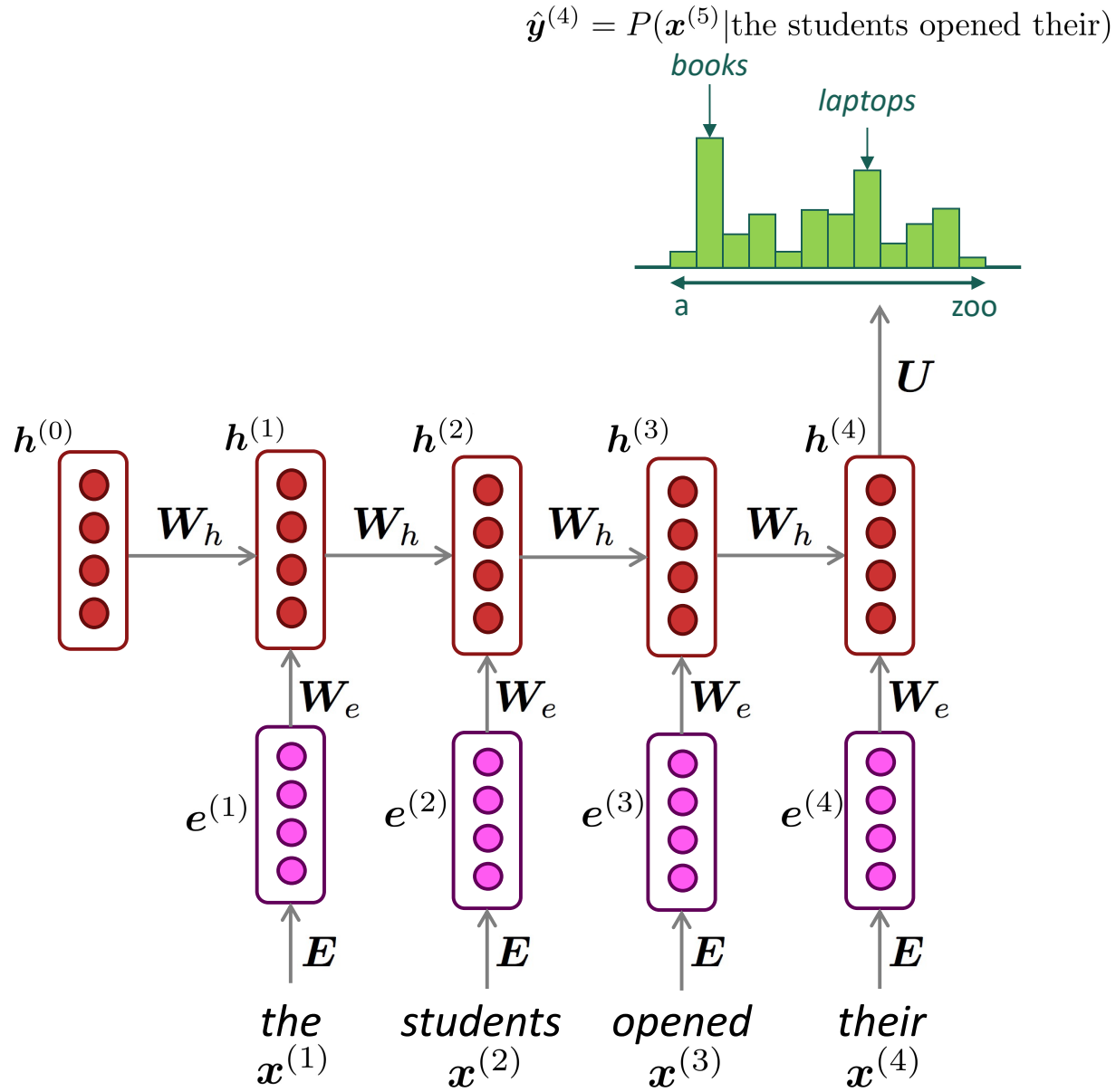
words / one-hot vectors

$$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$$



# RNN Language Models

- RNN **Advantages**:
  - Can process any length input
  - Computation for step t can (in theory) use information from many steps back
  - Model size doesn't increase for longer input context
  - Same weights applied on every timestep, so there is symmetry in how inputs are processed.
- RNN **Disadvantages**:
  - Recurrent computation is slow
  - In practice, difficult to access information from many steps back



# Training an RNN Language Model

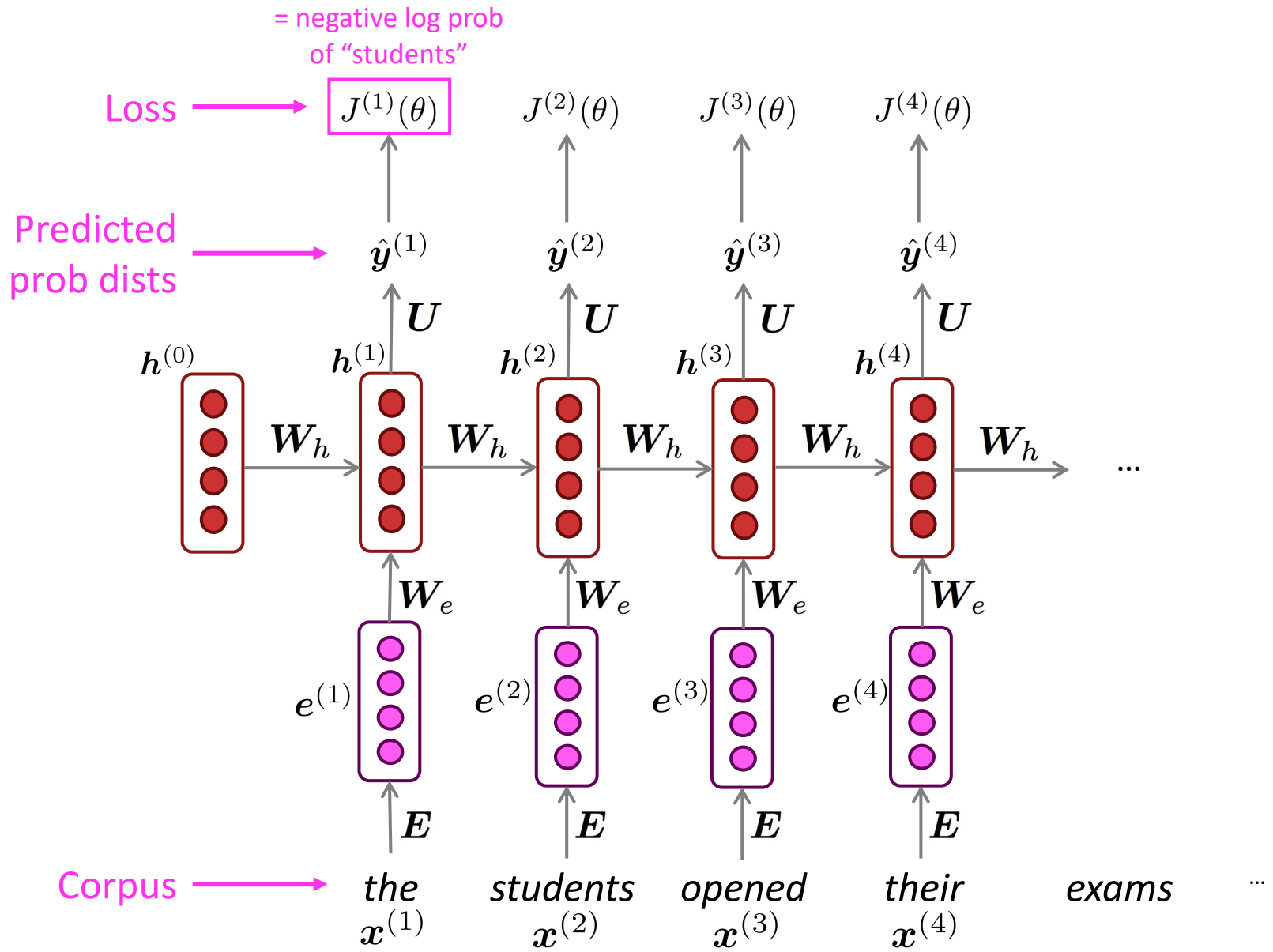
- Get a **big corpus of text** which is a sequence of words $x^{(1)}, \ldots, x^{(T)}$
- Feed into RNN-LM; compute output distribution $\hat{y}^{(t)}$ *for every step t*
  - i.e. predict probability distribution of *every word*, given words so far
- **Loss function** on step t is **cross-entropy** between predicted probability distribution $\hat{y}^{(t)}$, and the true next word $y^{(t)}$ (one-hot for $x^{(t+1)}$):

$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{w \in V} \boldsymbol{y}_w^{(t)} \log \hat{\boldsymbol{y}}_w^{(t)} = -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

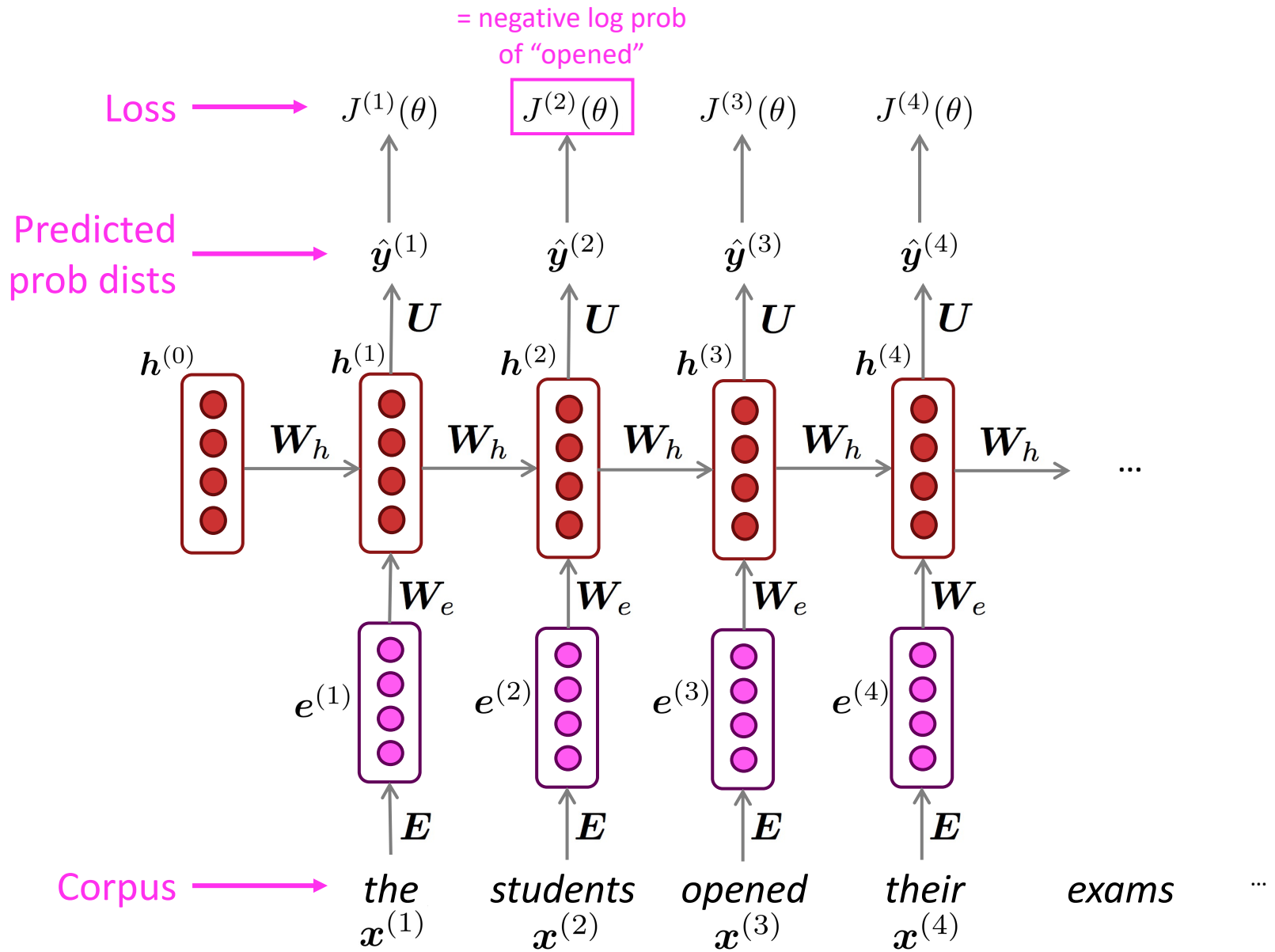
- Average this to get **overall loss** for entire training set:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

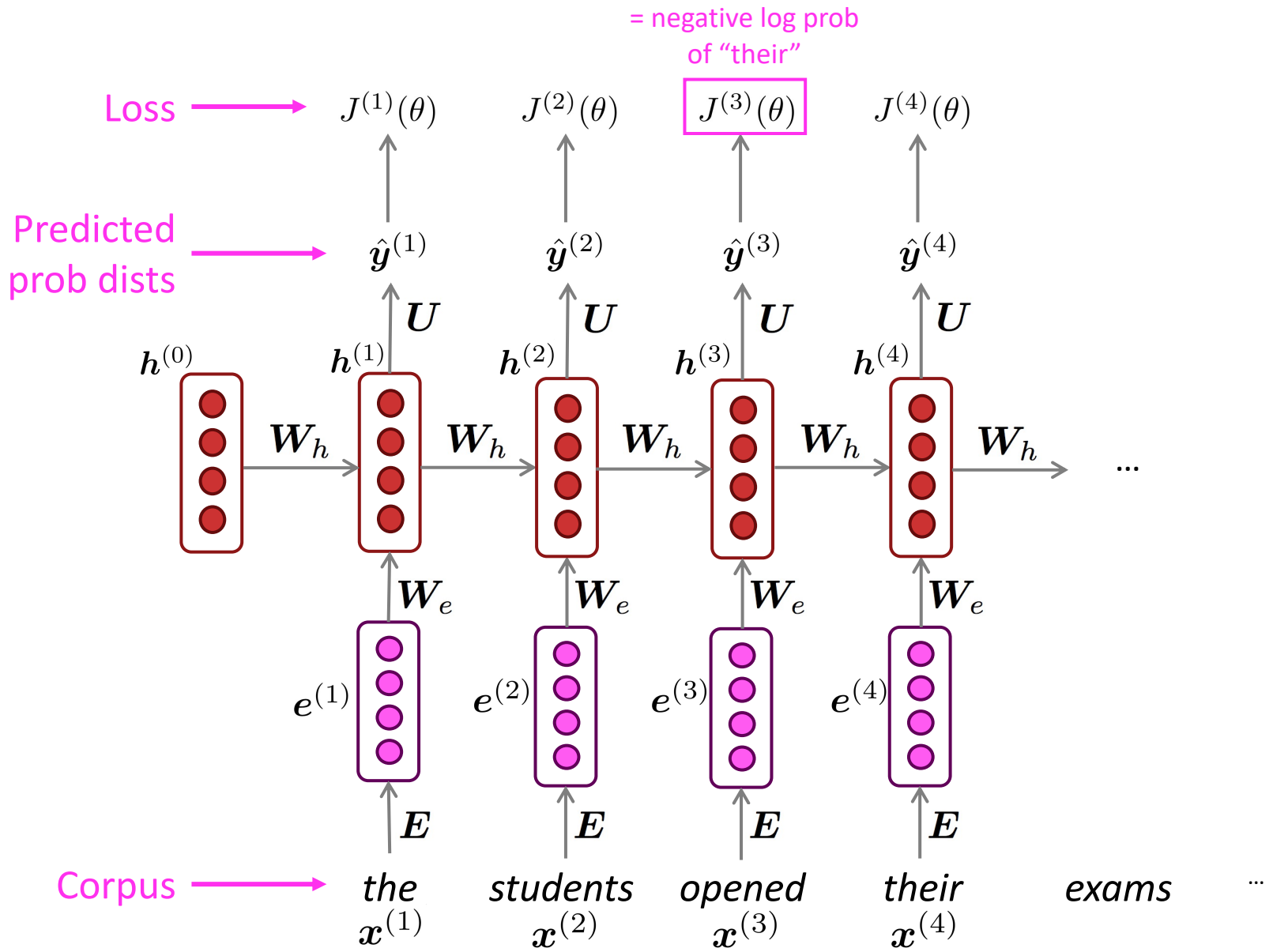
# Training an RNN Language Model



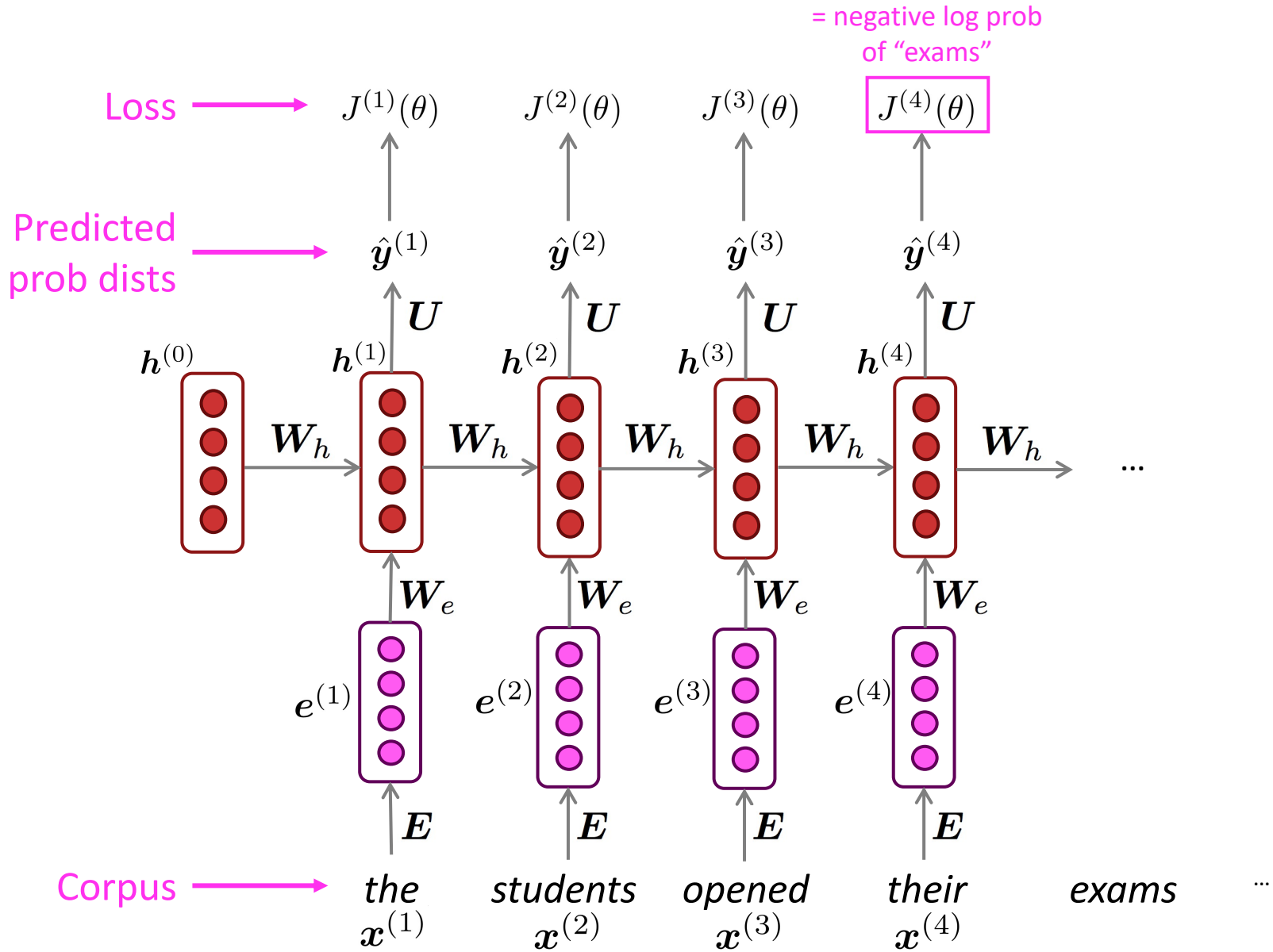
# Training an RNN Language Model



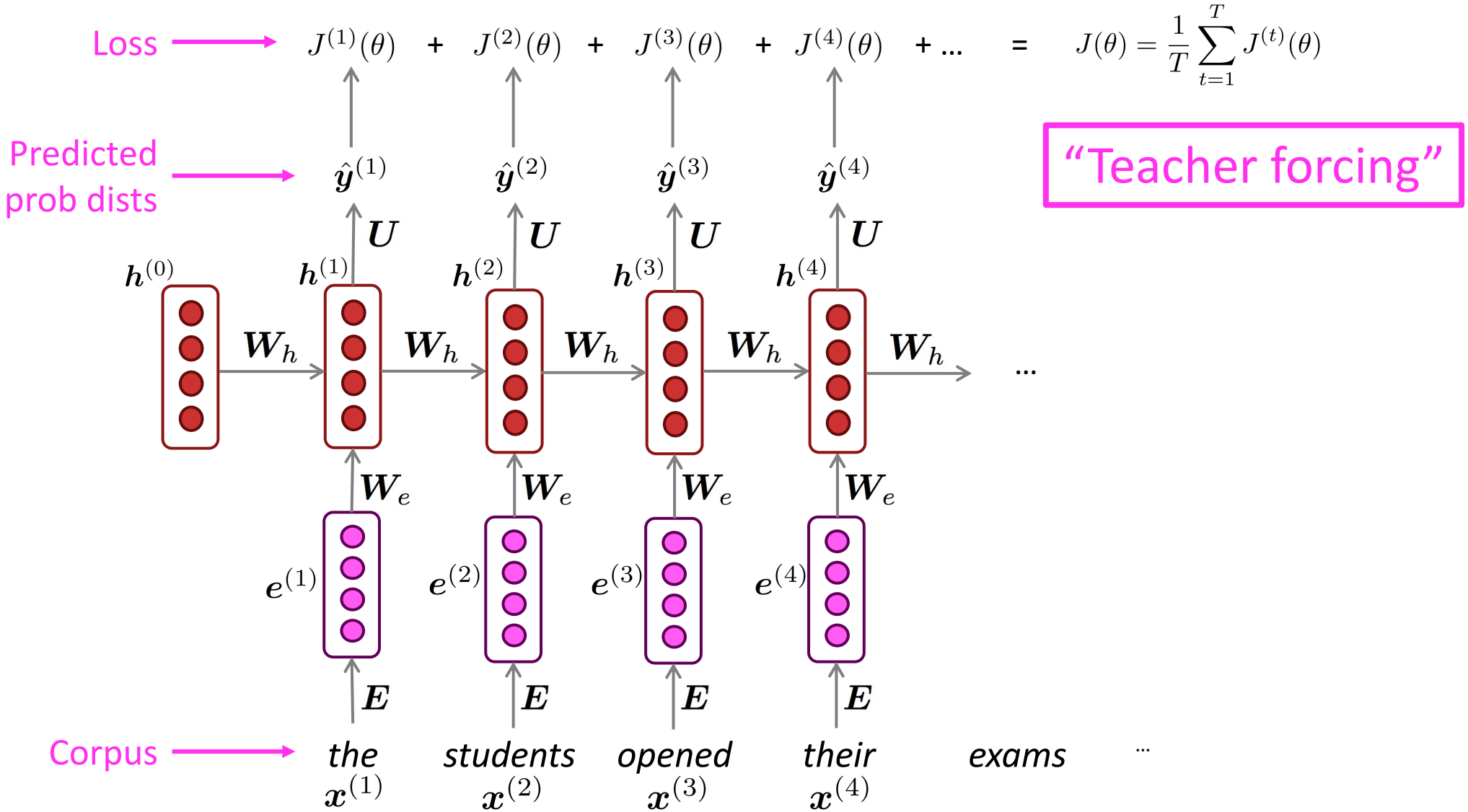
# Training an RNN Language Model



# Training an RNN Language Model



# Training an RNN Language Model



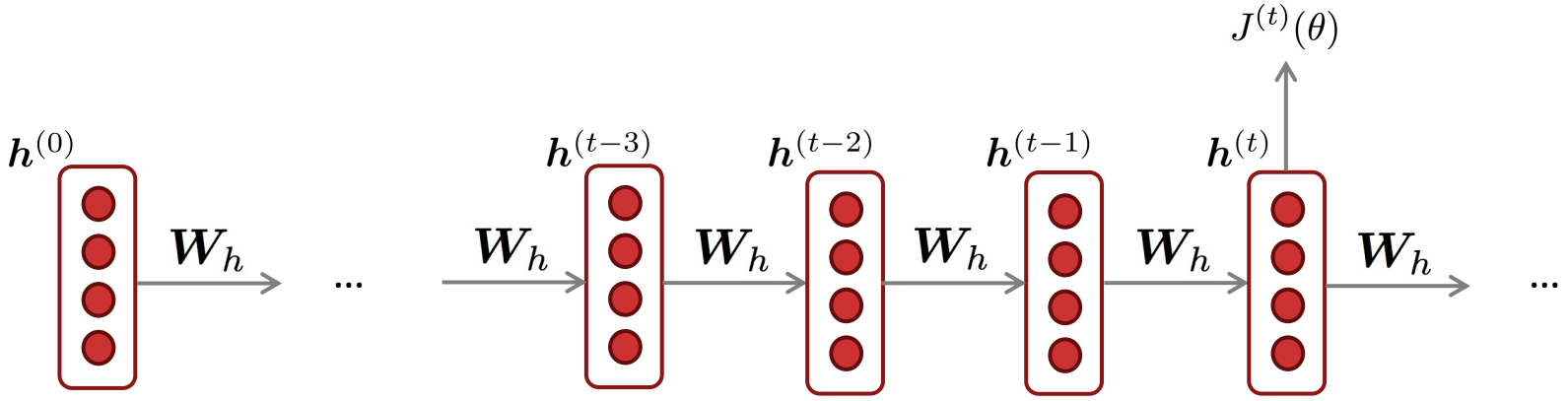
# Training an RNN Language Model

- However: Computing loss and gradients across **entire corpus** $x^{(1)}, \ldots, x^{(T)}$ is too expensive!

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta)$$

- In practice, consider $x^{(1)}, \ldots, x^{(T)}$ as a **sentence** (or a **document**)
- Recall: **Stochastic Gradient Descent** allows us to compute loss and gradients for small
- Compute loss $J(\theta)$ for a sentence (actually, a batch of sentences), compute gradients and update weights. Repeat.

# Backpropagation for RNNs




- **Question**: What's the derivative of $J^{(t)}(\theta)$ w.r.t. the repeated weight matrix $\boldsymbol{W_h}$?

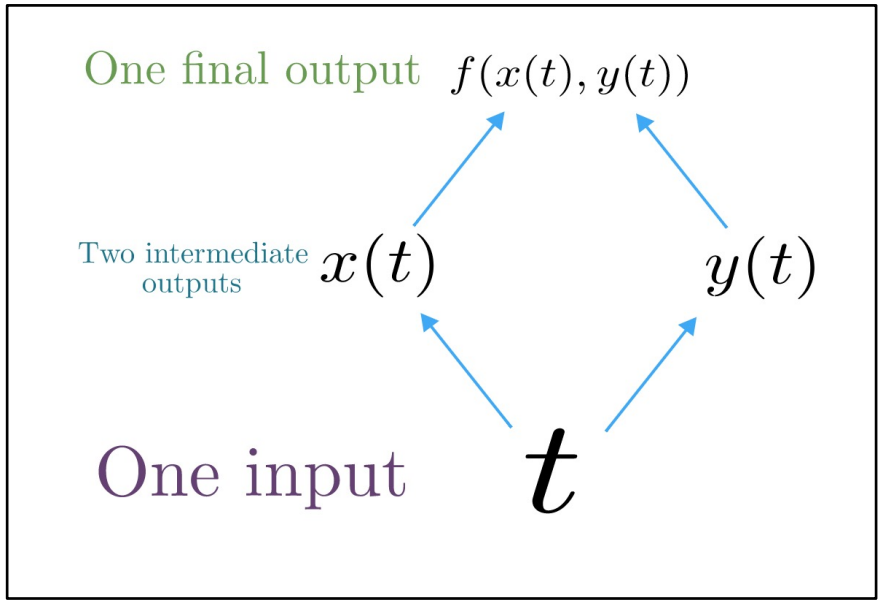
- **Answer:** $$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}}\bigg|_{(i)}$$

- "The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears"

# Multivariable Chain Rule

- Given a multivariable function $f(x,y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$



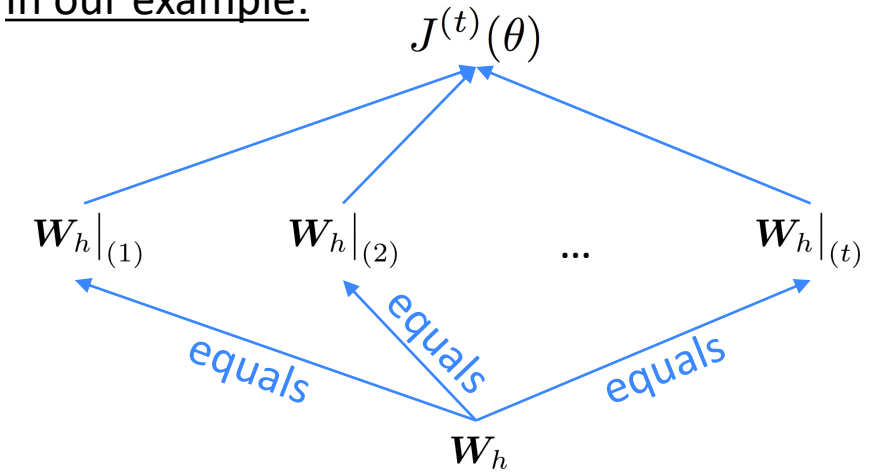

**Source**: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version

# Backpropagation for RNNs: Proof sketch

- Given a multivariable function $f(x,y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

In our example:




Apply the multivariable chain rule:

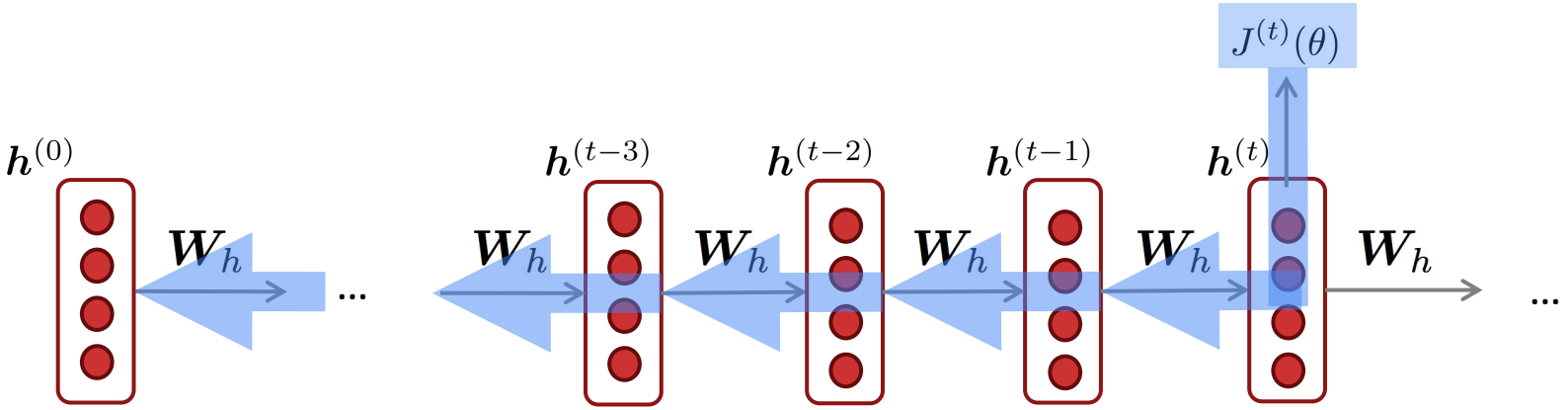
= 1

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\bigg|_{(i)} \frac{\partial \boldsymbol{W}_h|_{(i)}}{\partial \boldsymbol{W}_h}$$

$$= \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\bigg|_{(i)}$$

**Source**: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version

# Backpropagation for RNNs




$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} \right|_{(i)}$$

**Question:** How do we calculate this?

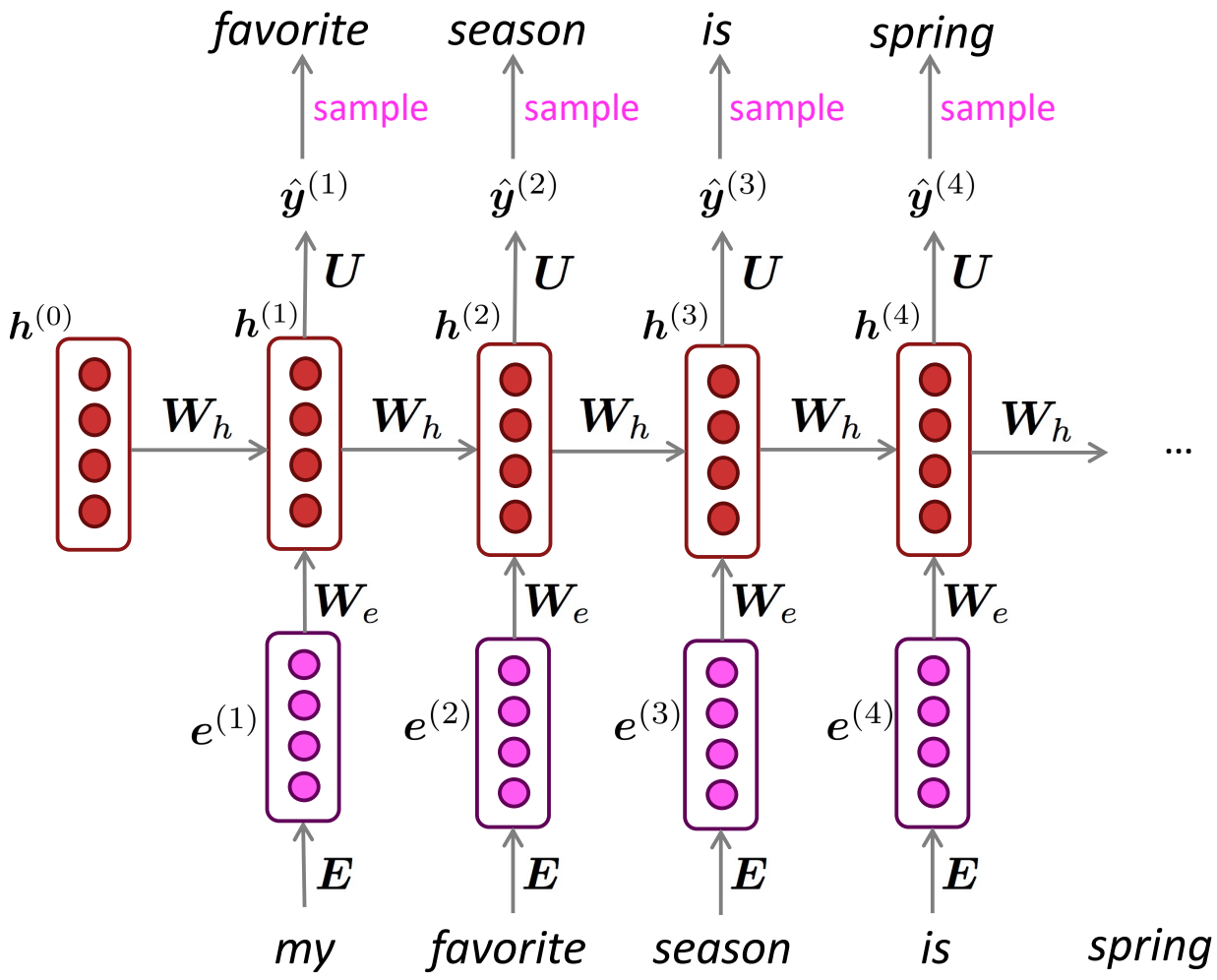
**Answer:** Backpropagate over timesteps *i=t,…,0*, summing gradients as you go. This algorithm is called **"backpropagation through time"** [Werbos, P.G., 1988, *Neural Networks* **1**, and others]

In practice, often "truncated" after ~20 timesteps for training efficiency reasons

From Stanford CS224n course

# Generating text with a RNN Language Model

- Just like a n-gram Language Model, you can use a RNN Language Model to **generate text** by **repeated sampling**. Sampled output becomes next step's input.



# Generating text with a RNN Language Model

- Let's have some fun!
  - You can train an RNN-LM on any kind of text, then generate text in that style.
  - RNN-LM trained on *Obama speeches*:

*The United States will step up to the cost of a new challenges of the American people that will share the fact that we created the problem. They were attacked and so that they have to say that all the task of the final days of war that I will not be able to get this done.*

**Source:** https://medium.com/@samim/obama-rnn-machine-generated-political-speeches-c8abd18a2ea0

# Generating text with a RNN Language Model

- Let's have some fun!
  - You can train an RNN-LM on any kind of text, then generate text in that style.
  - RNN-LM 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.

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

# Generating text with a RNN Language Model

- Let's have some fun!
  - You can train an RNN-LM on any kind of text, then generate text in that style.
  - RNN-LM trained on *recipes*:

```
      Title: CHOCOLATE RANCH BARBECUE
Categories: Game, Casseroles, Cookies, Cookies
      Yield: 6 Servings

      2 tb Parmesan cheese -- chopped
      1 c  Coconut milk
      3    Eggs, beaten

  Place each pasta over layers of lumps. Shape mixture into the moderate oven and simmer
  until firm. Serve hot in bodied fresh, mustard, orange and cheese.

  Combine the cheese and salt together the dough in a large skillet; add the ingredients
  and stir in the chocolate and pepper.
```

**Source:** https://gist.github.com/nylki/1efbaa36635956d35bcc

# Evaluating Language Models

- The standard **evaluation metric** for Language Models is **perplexity**.

$$\text{perplexity} = \prod_{t=1}^{T} \left( \frac{1}{P_{\text{LM}}(\boldsymbol{x}^{(t+1)} | \ \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})} \right)^{1/T}$$

Normalized by number of words

Inverse probability of corpus, according to Language Model

- This is equal to the exponential of the cross-entropy loss $J(\theta)$:

$$= \prod_{t=1}^{T} \left( \frac{1}{\hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}}} \right)^{1/T} = \exp\left( \frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}} \right) = \exp(J(\theta))$$

**Lower** perplexity is better!

# RNNs have greatly improved perplexity

*n*-gram model →

Increasingly complex RNNs ↓

| Model | Perplexity |
|---|---|
| Interpolated Kneser-Ney 5-gram (Chelba et al., 2013) | 67.6 |
| RNN-1024 + MaxEnt 9-gram (Chelba et al., 2013) | 51.3 |
| RNN-2048 + BlackOut sampling (Ji et al., 2015) | 68.3 |
| Sparse Non-negative Matrix factorization (Shazeer et al., 2015) | 52.9 |
| LSTM-2048 (Jozefowicz et al., 2016) | 43.7 |
| 2-layer LSTM-8192 (Jozefowicz et al., 2016) | 30 |
| **Ours small** (LSTM-2048) | 43.9 |
| **Ours large** (2-layer LSTM-2048) | 39.8 |

↓ Perplexity improves (lower is better)

**Source:** https://research.fb.com/building-an-efficient-neural-language-model-over-a-billion-words/

# Why should we care about Language Modeling?

- Language Modeling is a **benchmark task** that helps us **measure our progress** on understanding language
- Language Modeling is a **subcomponent** of many NLP tasks, especially those involving **generating text** or **estimating the probability of text**:
  - Predictive typing
  - Speech recognition
  - Handwriting recognition
  - Spelling/grammar correction
  - Authorship identification
  - Machine translation
  - Summarization
  - Dialogue
  - etc.
- Language Modeling has been extended to cover everything else in NLP: **GPT-3 is an LM!**

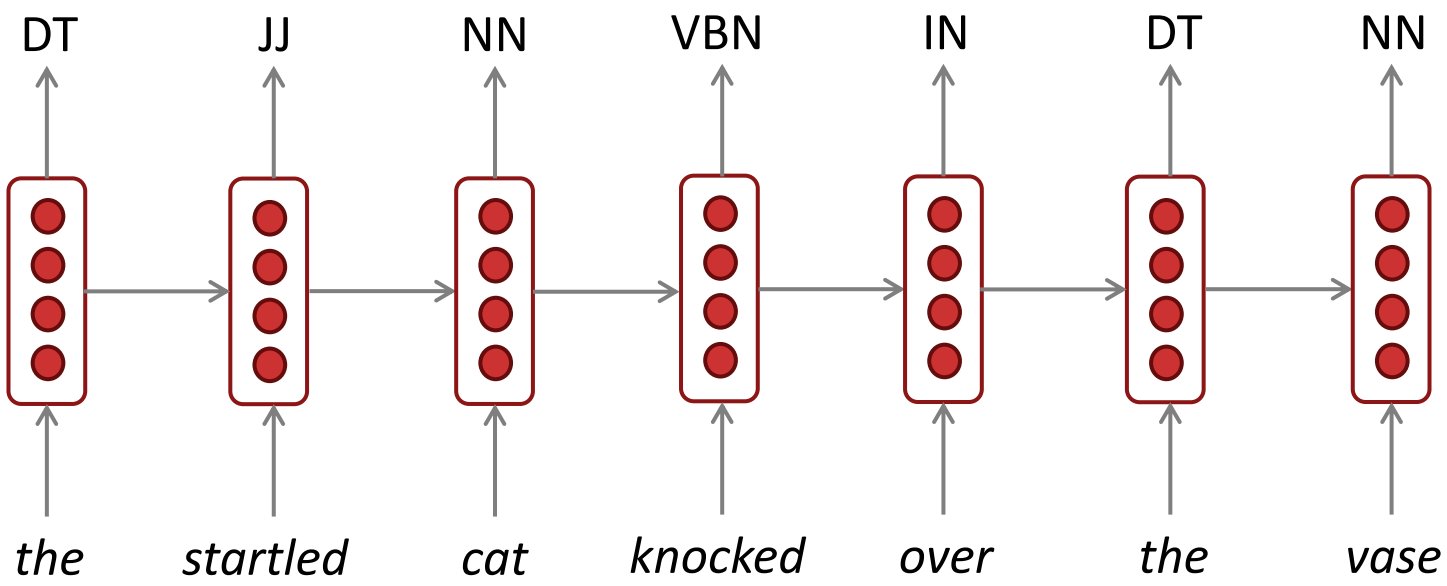
# Summary

- Language Model: A system that predicts the next word
- **Recurrent Neural Network**: A family of neural networks that:
  - Take sequential input of any length
  - Apply the same weights on each step
  - Can optionally produce output on each step

- Recurrent Neural Network ≠ Language Model
- We've shown that RNNs are a great way to build a LM.
- But RNNs are useful for much more!

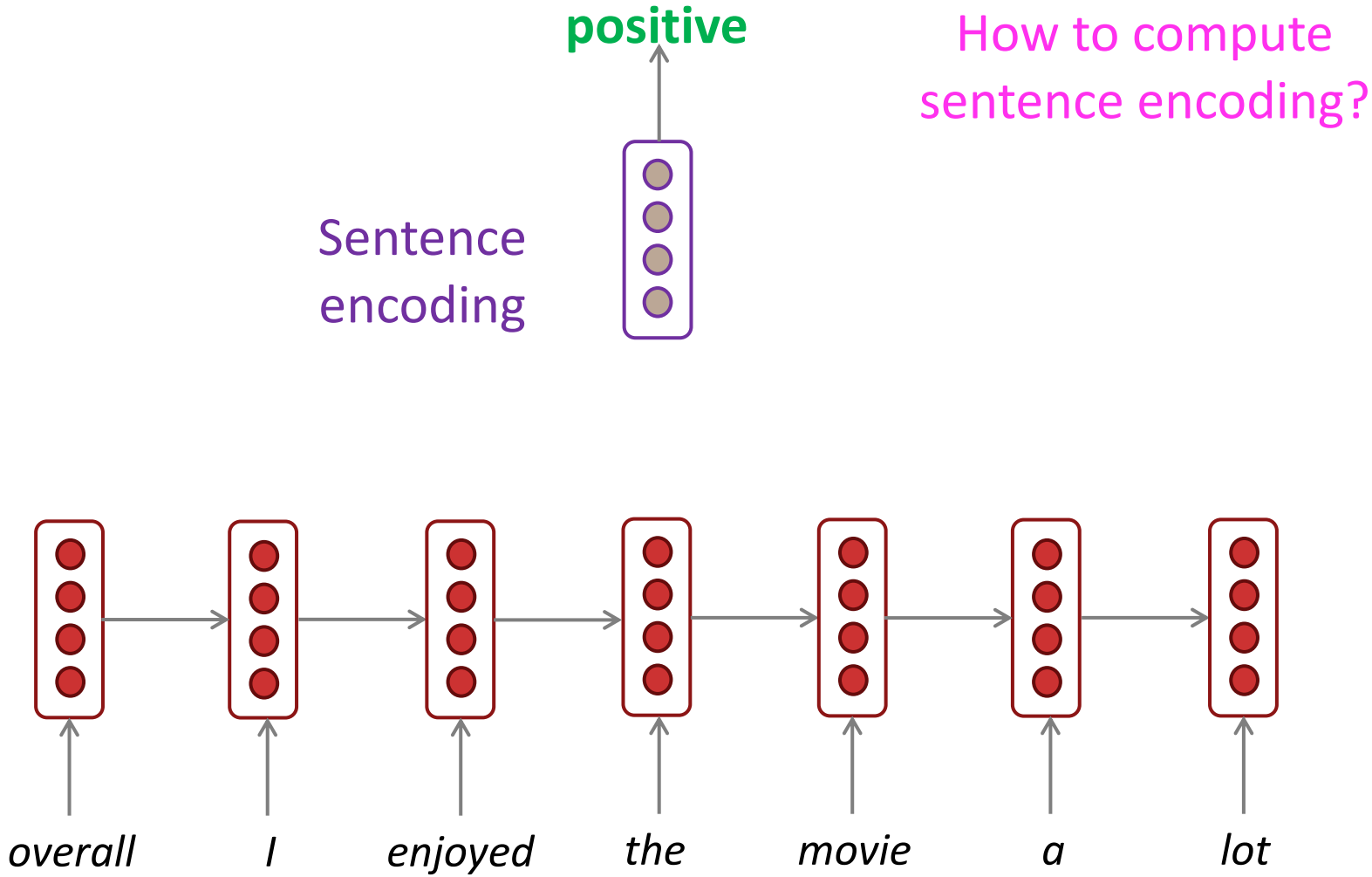
# RNNs can be used for tagging

- e.g., **part-of-speech tagging**, named entity recognition



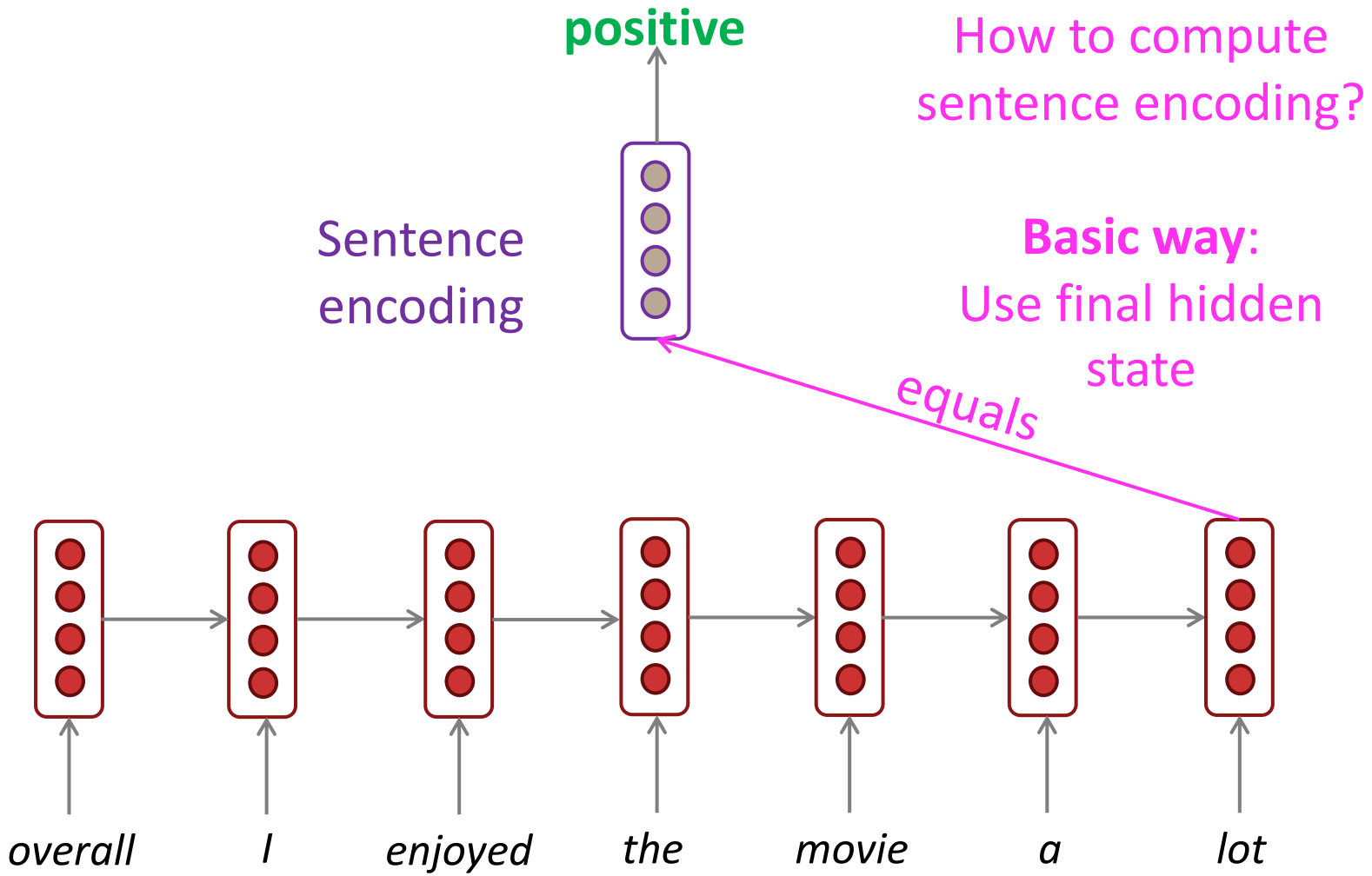
# RNNs can be used for sentence classification

- e.g., sentiment classification



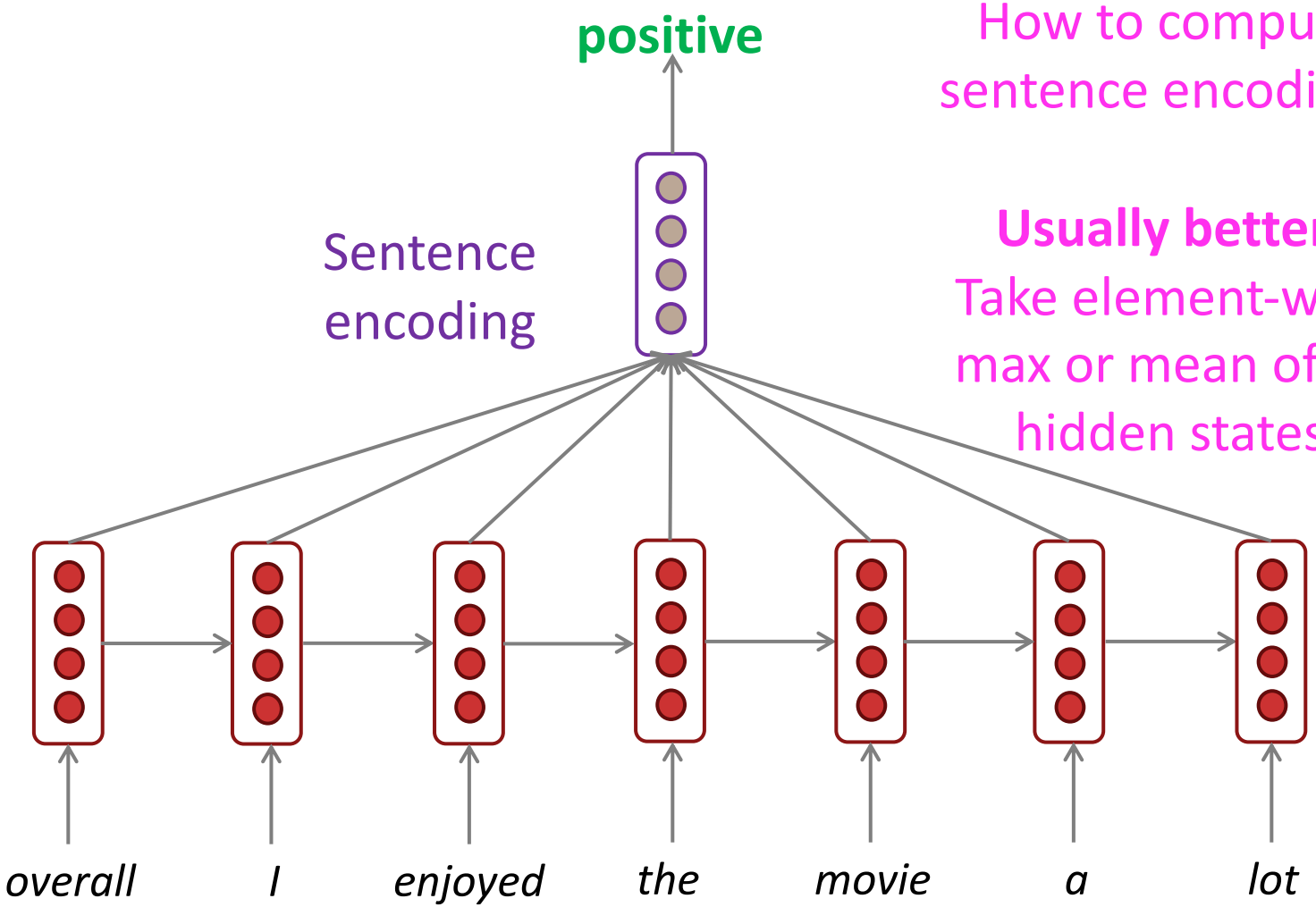
# RNNs can be used for sentence classification

- e.g., sentiment classification



# RNNs can be used for sentence classification

- e.g., sentiment classification



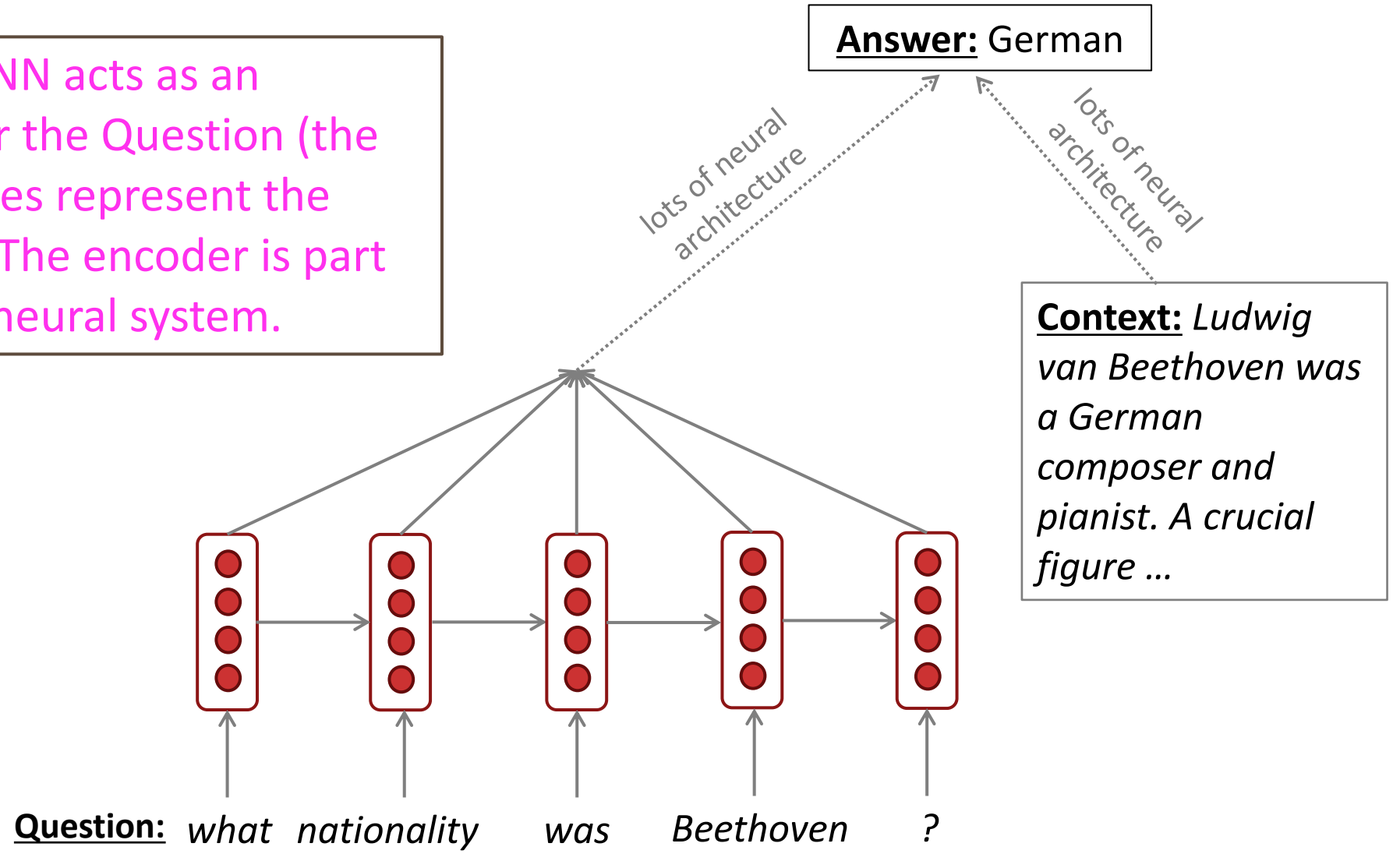

How to compute sentence encoding?

**Usually better**: Take element-wise max or mean of all hidden states

# RNNs can be used as an encoder module

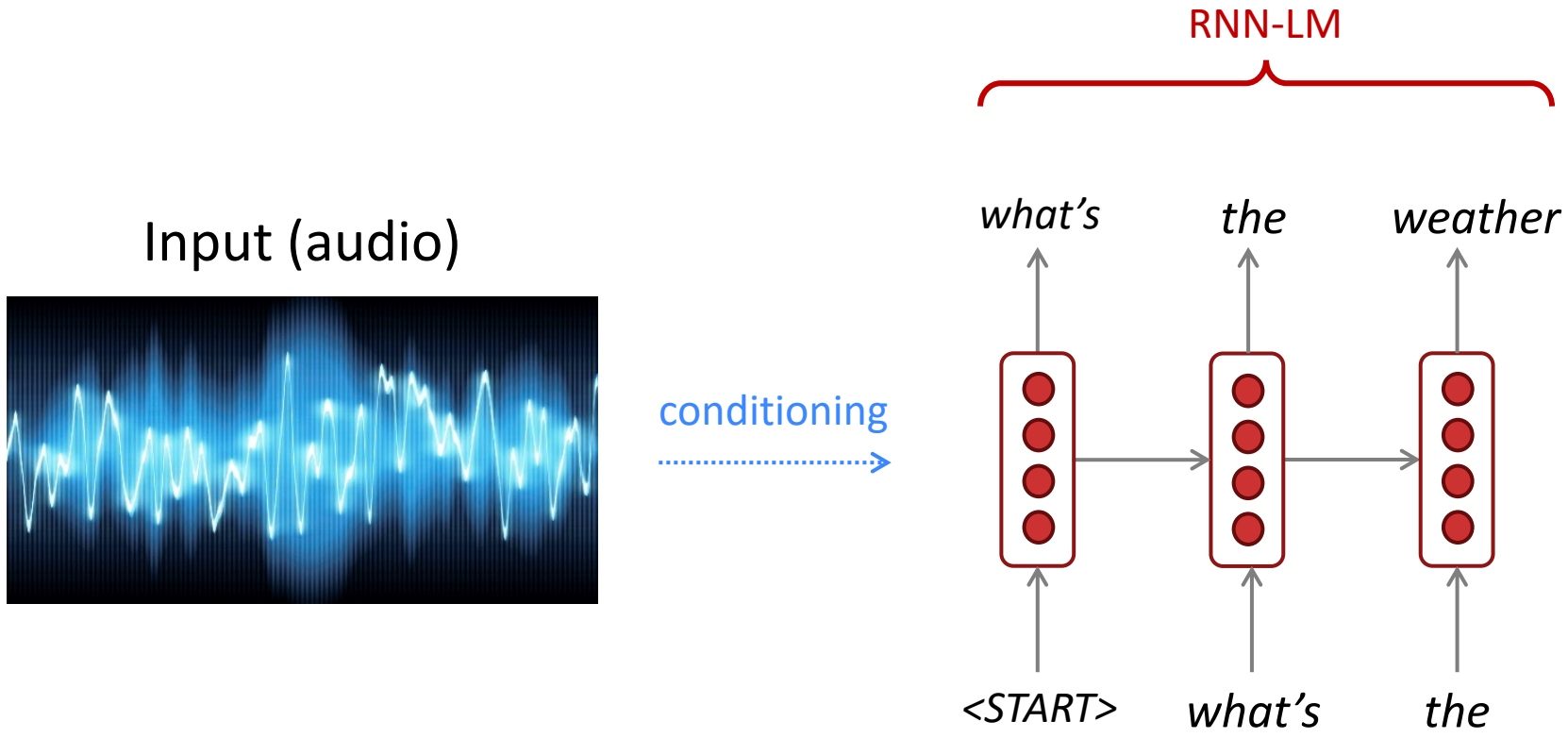
- e.g., **question answering**, machine translation, many other tasks!

Here the RNN acts as an **encoder** for the Question (the hidden states represent the Question). The encoder is part of a larger neural system.



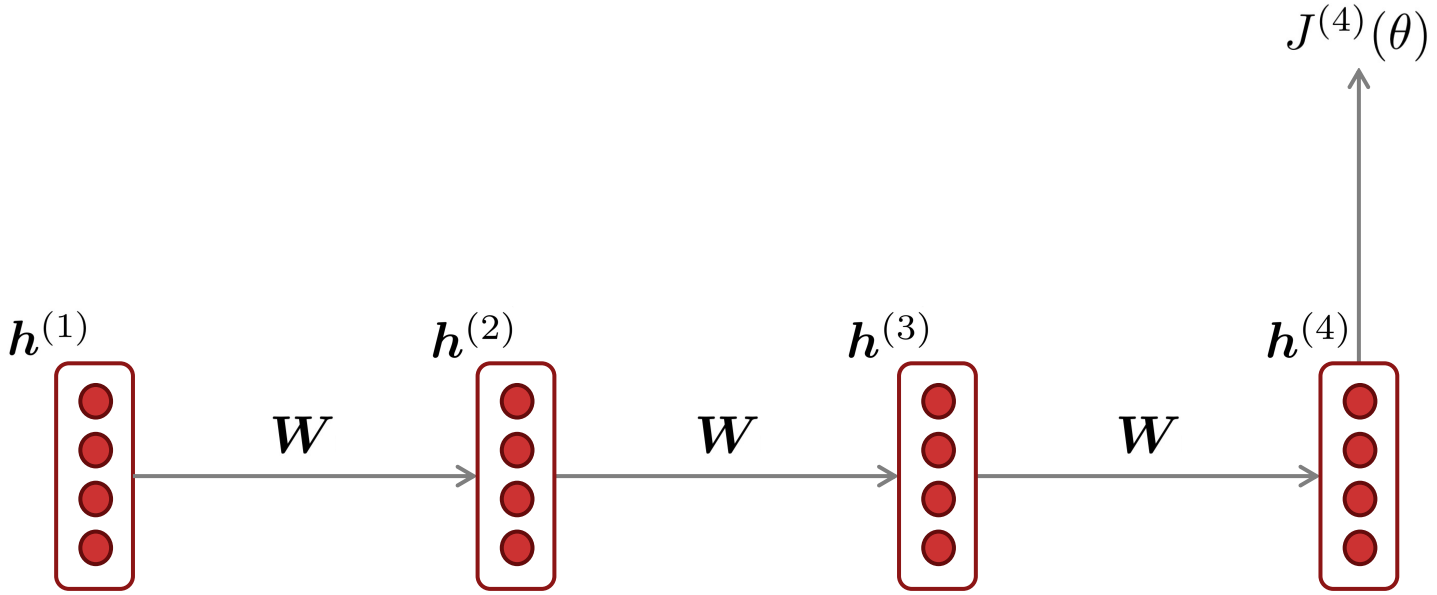
# RNN-LMs can be used to generate text

- e.g., speech recognition, machine translation, summarization

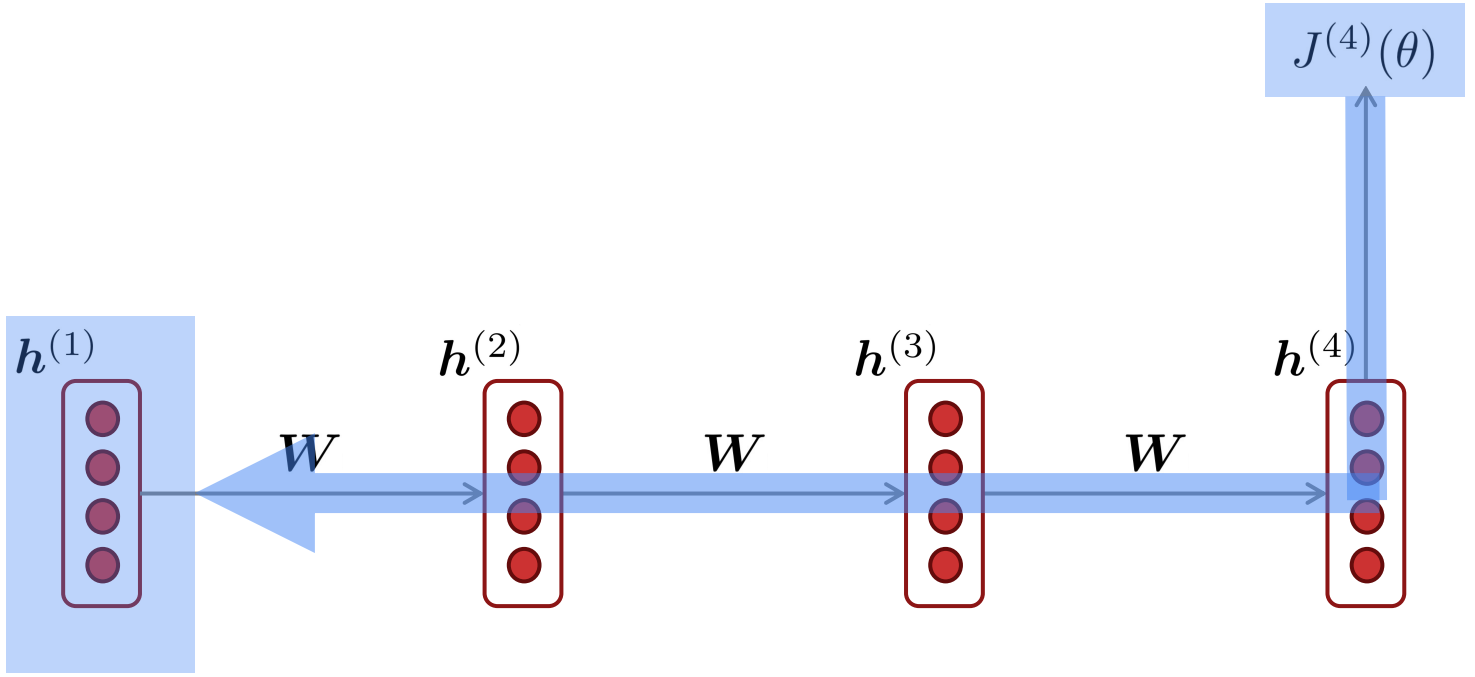



This is an example of a *conditional language model*.
We'll see Machine Translation in much more detail later.

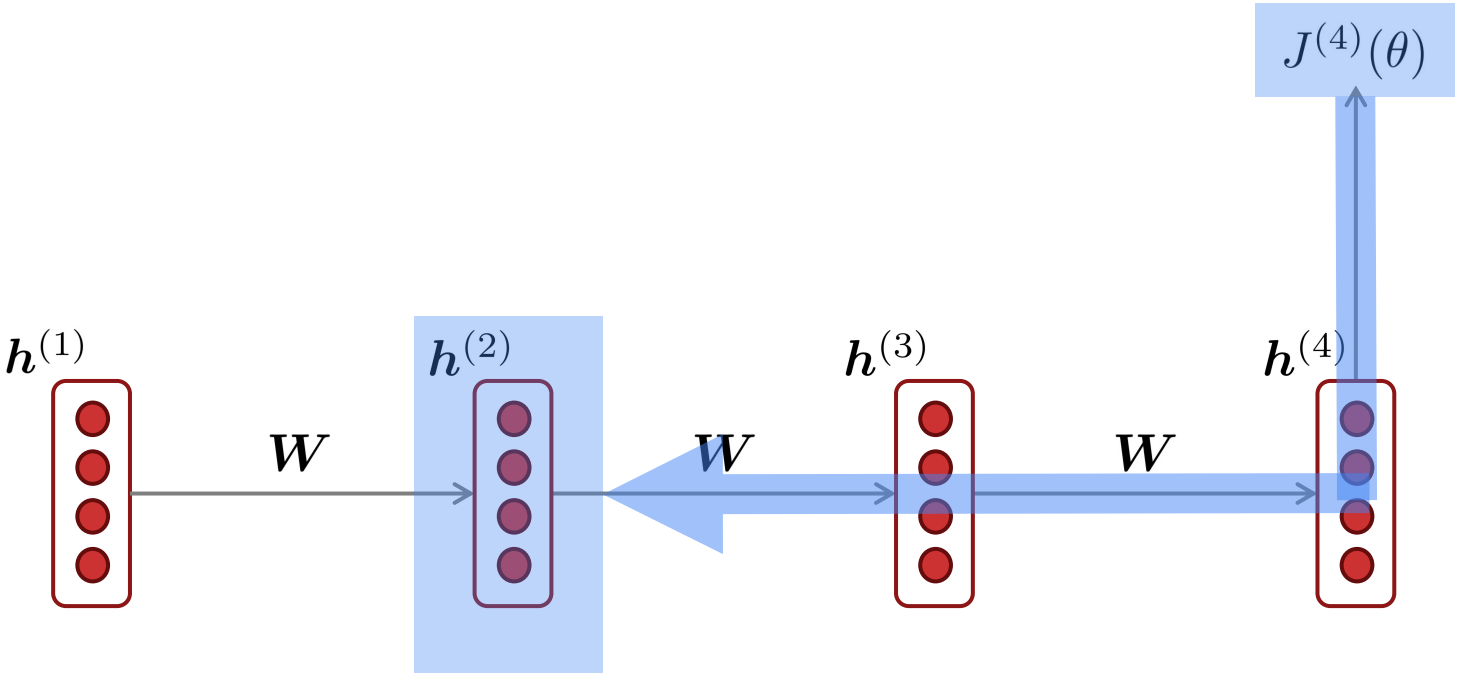
# Problems with RNNs: Vanishing and Exploding Gradients



# Vanishing gradient intuition



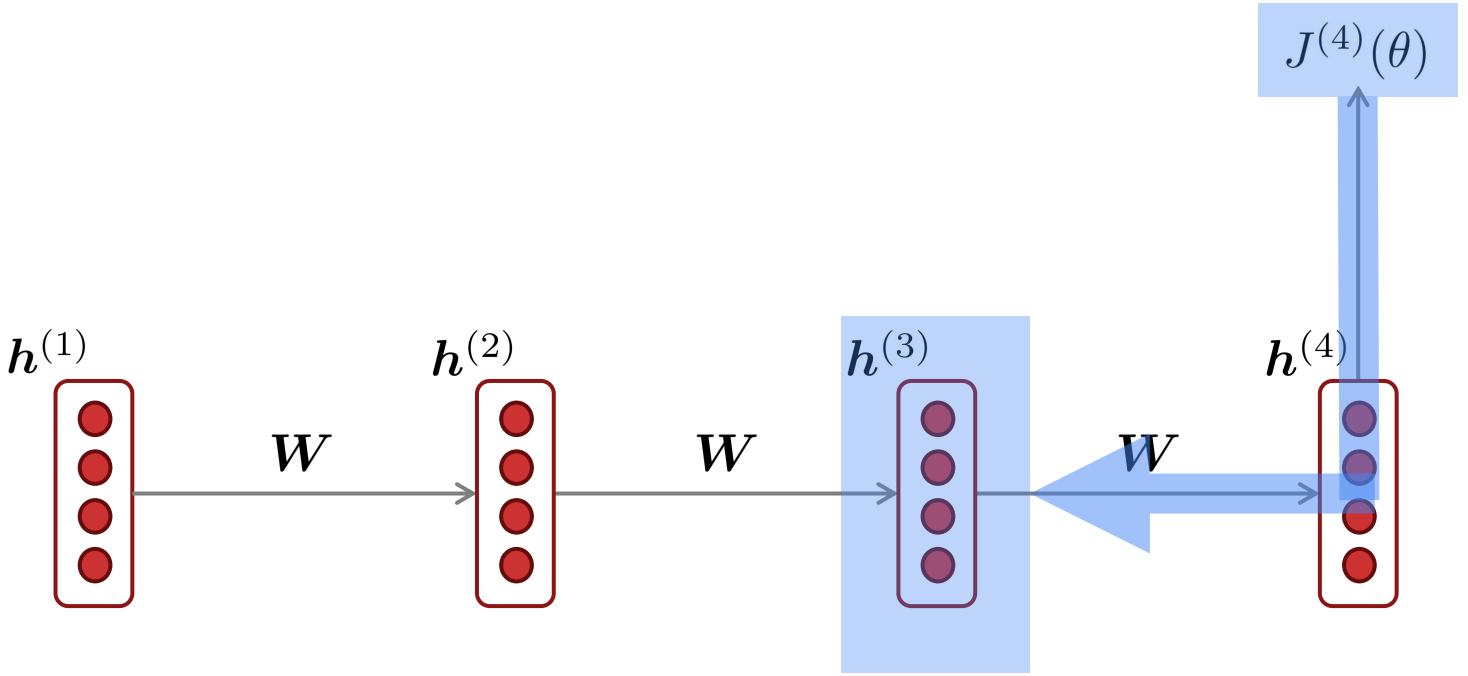
# Vanishing gradient intuition




$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial J^{(4)}}{\partial h^{(2)}}$$

chain rule!

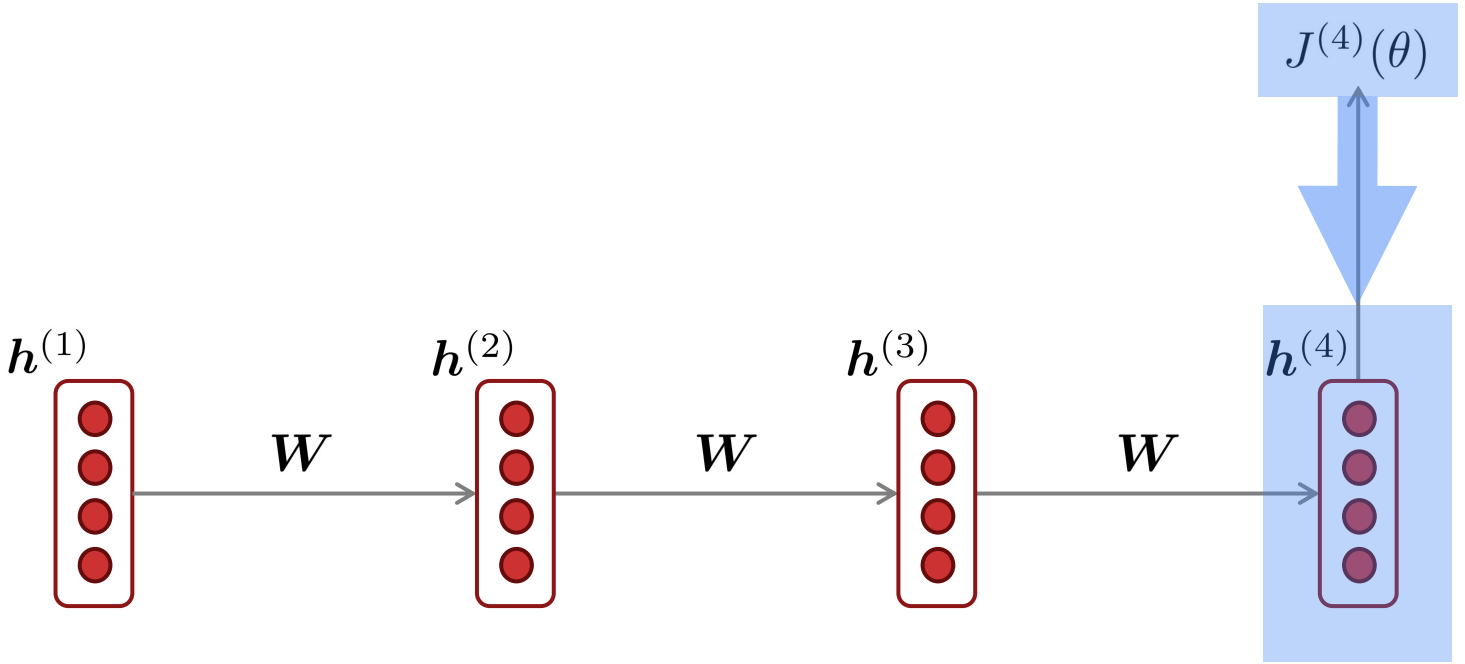
# Vanishing gradient intuition




$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial J^{(4)}}{\partial h^{(3)}}$$

chain rule!

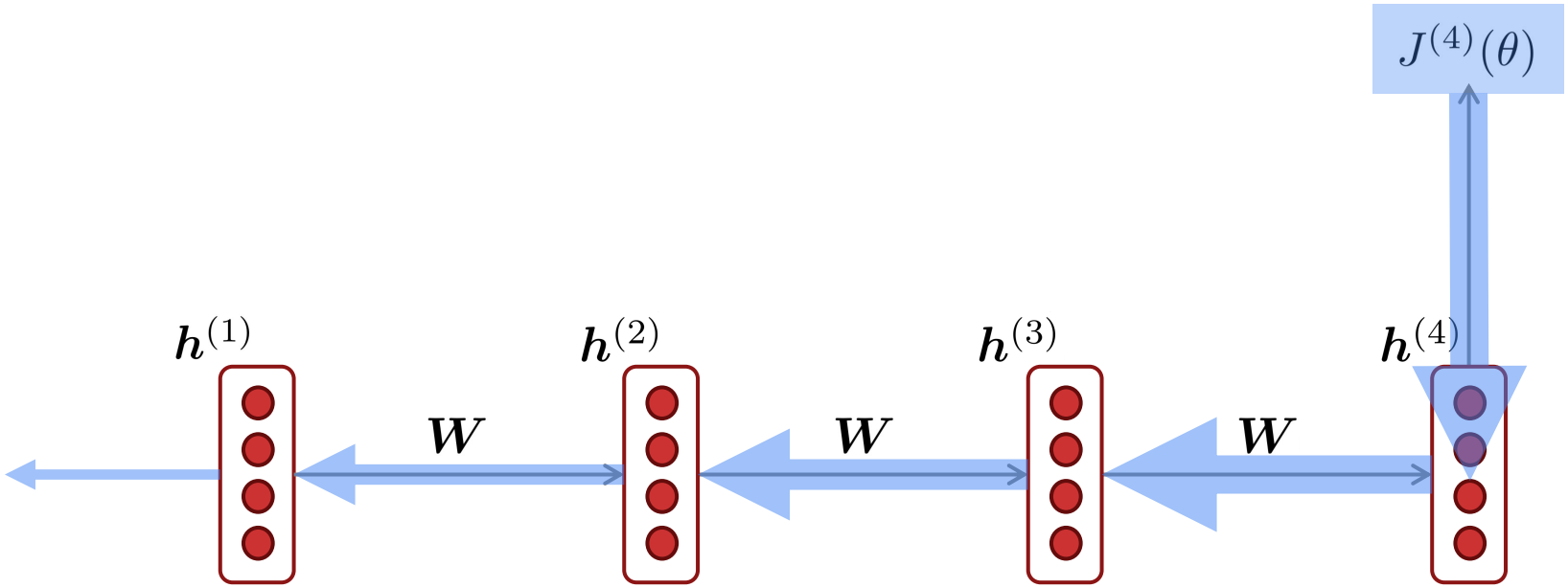
# Vanishing gradient intuition




$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \times \frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}} \times \frac{\partial \boldsymbol{h}^{(4)}}{\partial \boldsymbol{h}^{(3)}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(4)}}$$

chain rule!

# Vanishing gradient intuition




$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

What happens if these are small?

**Vanishing gradient problem:**
When these are small, the gradient signal gets smaller and smaller as it backpropagates further

# Vanishing gradient proof sketch (linear case)

- Recall: $$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)$$
- What if $\sigma$ were the identity function, $\sigma(x) = x$

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} = \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right) \boldsymbol{W}_h \qquad \text{(chain rule)}$$

$$= \boldsymbol{I}\, \boldsymbol{W}_h = \boldsymbol{W}_h$$

- Consider the gradient of the loss $J^{(i)}(\theta)$ on step $i$, with respect to the hidden state $\boldsymbol{h}^{(j)}$ on some previous step $j$. Let $\ell = i - j$

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j < t \leq i} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \qquad \text{(chain rule)}$$

$$= \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j < t \leq i} \boldsymbol{W}_h = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boxed{\boldsymbol{W}_h^{\ell}} \qquad \text{(value of } \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}\text{)}$$

If $W_h$ is "small", then this term gets exponentially problematic as $\ell$ becomes large

**Source**: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013

# Vanishing gradient proof sketch (linear case)

- What's wrong with $\boldsymbol{W}_h^\ell$ ?
- Consider if the eigenvalues of $\boldsymbol{W}_h$ are all less than 1

$$\lambda_1, \lambda_2, \ldots, \lambda_n < 1$$
$$\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n \text{ (eigenvectors)}$$

- We can write $\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^\ell$ using the eigenvectors of $\boldsymbol{W}_h$ as a basis:

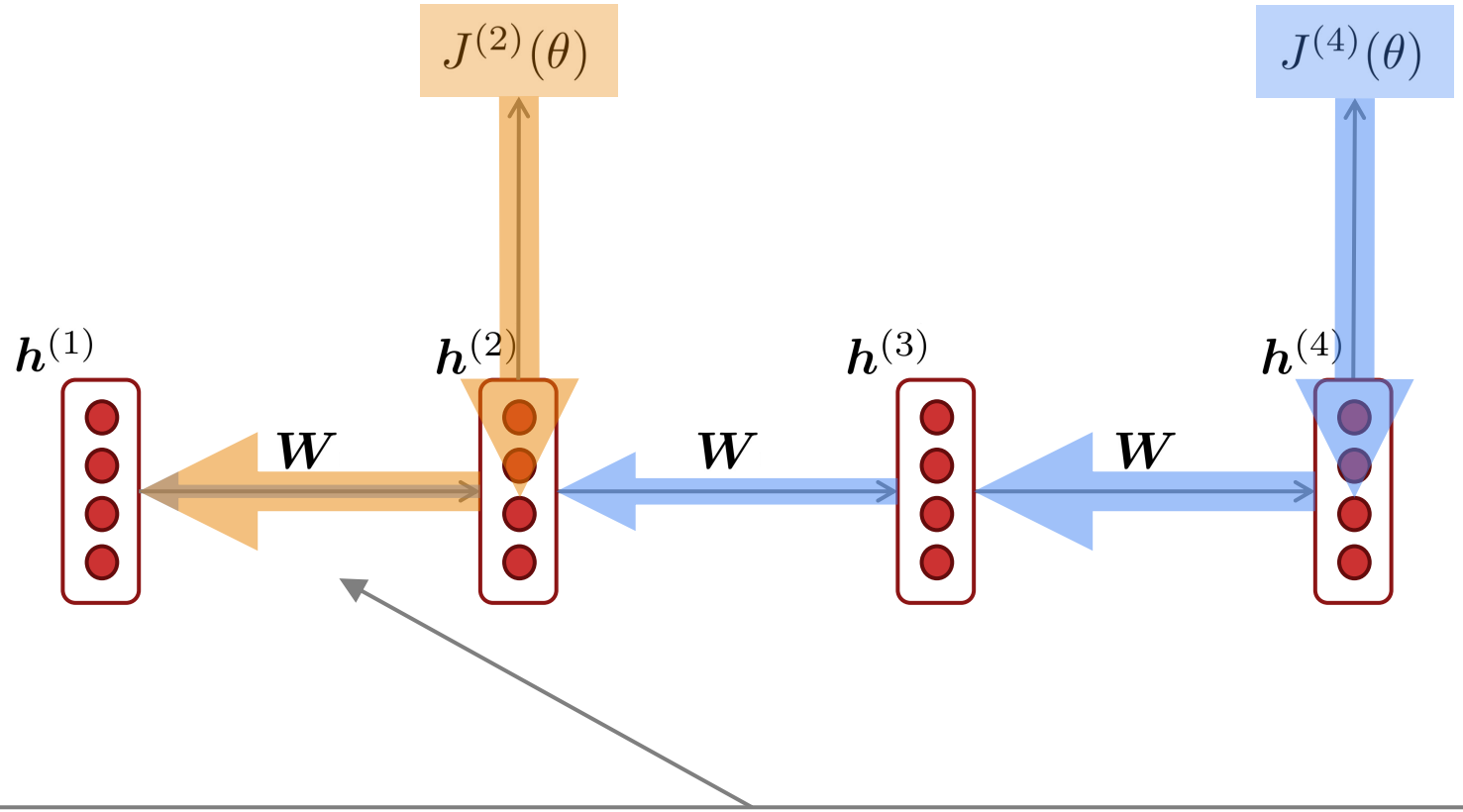
$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^\ell = \sum_{i=1}^{n} c_i \lambda_i^\ell \boldsymbol{q}_i \approx \mathbf{0} \text{ (for large } \ell)$$

Approaches 0 as $\ell$ grows, so gradient vanishes

- What about nonlinear activations $\sigma$ (i.e., what we use?)
  - Pretty much the same thing, except the proof requires $\lambda_i < \gamma$ for some $\gamma$ dependent on dimensionality and $\sigma$

**Source**: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013

# Why is vanishing gradient a problem?




Gradient signal from far away is lost because it's much smaller than gradient signal from close-by.

So, model weights are updated only with respect to near effects, not long-term effects.

# Why is exploding gradient a problem?

- If the gradient becomes too big, then the SGD update step becomes too big:

$$\theta^{new} = \theta^{old} - \overbrace{\alpha}^{\text{learning rate}} \underbrace{\nabla_\theta J(\theta)}_{\text{gradient}}$$

- This can cause **bad updates**: we take too large a step and reach a weird and bad parameter configuration (with large loss)
- In the worst case, this will result in **Inf** or **NaN** in your network (then you have to restart training from an earlier checkpoint)

# Gradient clipping: solution for exploding gradient

- **Gradient clipping**: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

**Algorithm 1** Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
**if** $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
**end if**

- **Intuition**: take a step in the same direction, but a smaller step
- In practice, remembering to clip gradients is important, but exploding gradients are an easy problem to solve

**Source**: "On the difficulty of training recurrent neural networks", Pascanu et al, 2013

# How to fix the vanishing gradient problem?

- The main problem is that *it's too difficult for the RNN to learn to preserve information over many timesteps*.
- In a vanilla RNN, the hidden state is constantly being **rewritten**

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}\right)$$

- How about an RNN with separate **memory** which is added to?

# Long Short-Term Memory RNNs (LSTMs)

- A type of RNN proposed by Hochreiter and Schmidhuber in 1997 as a solution to the vanishing gradients problem.
  - Everyone cites that paper but really a crucial part of the modern LSTM is from Gers et al. (2000)

- On step t, there is a **hidden state** $h^{(t)}$ *and* a **cell state** $c^{(t)}$
  - Both are vectors length n
  - The cell stores **long-term information**
  - The LSTM can **read**, **erase**, and **write** information from the cell
    - The cell becomes conceptually rather like RAM in a computer

"Long short-term memory", Hochreiter and Schmidhuber, 1997. https://www.bioinf.jku.at/publications/older/2604.pdf
"Learning to Forget: Continual Prediction with LSTM", Gers, Schmidhuber, and Cummins, 2000. https://dl.acm.org/doi/10.1162/089976600300015015

# Long Short-Term Memory RNNs (LSTMs)

- On step t, there is a **hidden state** $h^{(t)}$ *and* a **cell state** $c^{(t)}$
  - Both are vectors length n
  - The cell stores long-term information
  - The LSTM can read, erase, and write information from the cell
    - The cell becomes conceptually rather like RAM in a computer

- The selection of which information is erased/written/read is controlled by three corresponding gates
  - The gates are also vectors of length n
  - On each timestep, each element of the gates can be **open** (1), **closed** (0), or somewhere in-between
  - The gates are **dynamic**: their value is computed based on the current context

"Long short-term memory", Hochreiter and Schmidhuber, 1997. https://www.bioinf.jku.at/publications/older/2604.pdf
"Learning to Forget: Continual Prediction with LSTM", Gers, Schmidhuber, and Cummins, 2000. https://dl.acm.org/doi/10.1162/089976600300015015

# Long Short-Term Memory RNNs (LSTMs)

- We have a sequence of inputs $x^{(t)}$, and we will compute a sequence of hidden states $h^{(t)}$ and cell states $c^{(t)}$. On timestep $t$: