BBM 495
COMPUTATION GRAPHS AND DYNET

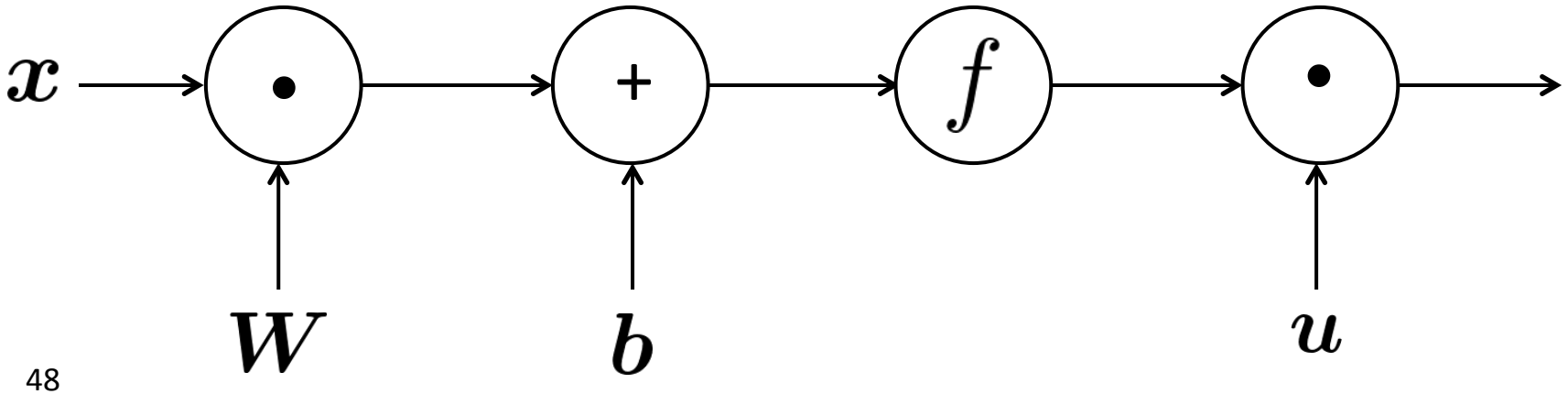
LECTURER: BURCU CAN

2019-2020 SPRING

Computational Graphs

- Representing our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations

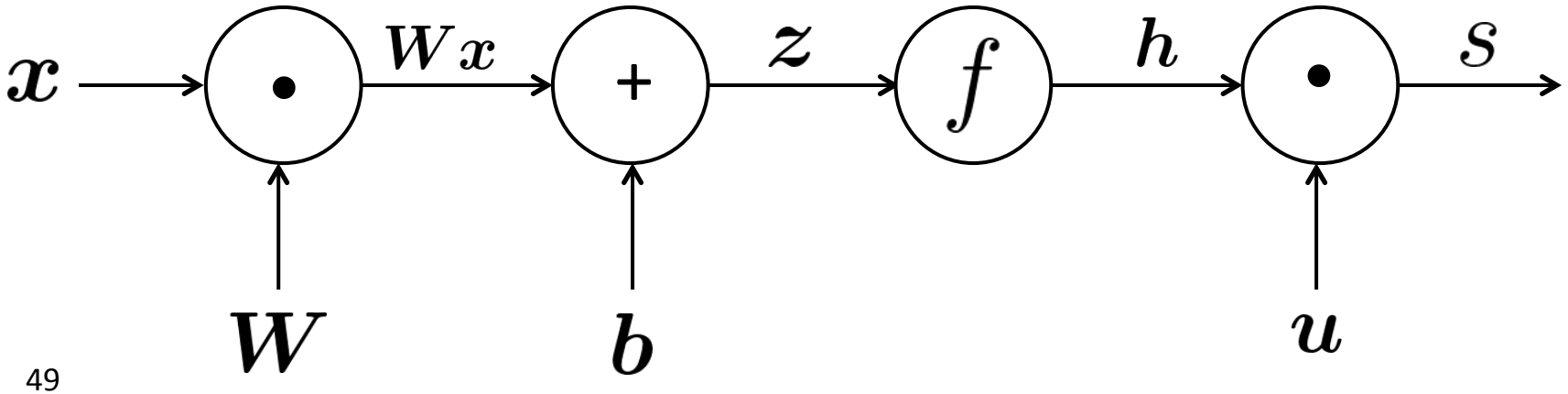
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad (\text{input})$$



Computational Graphs

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

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



Computational Graphs

- Representing our neural net equations as a graph

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

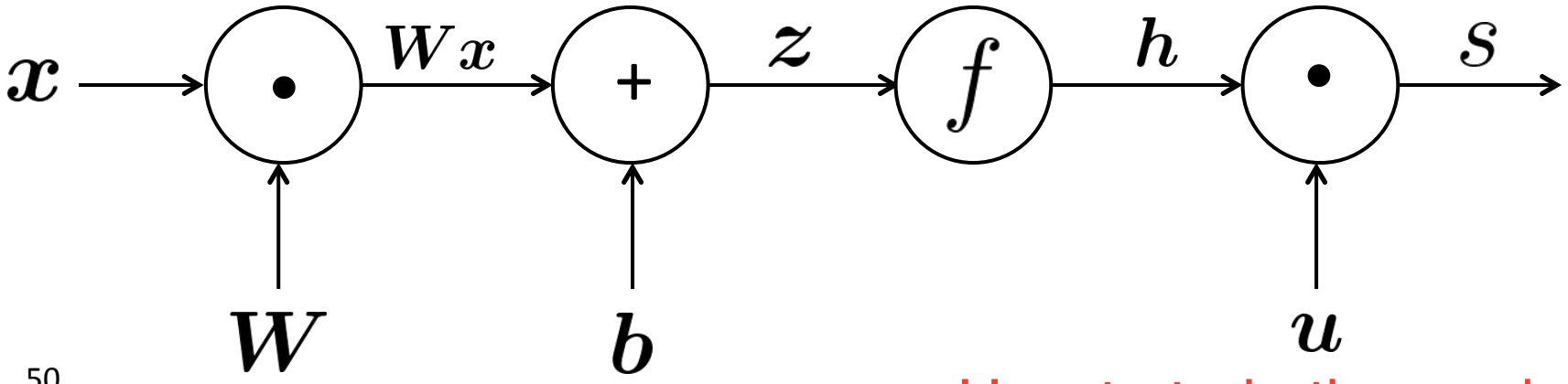
$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{c} + \boldsymbol{b}$$

ut)

"Forward Propagation"

operation

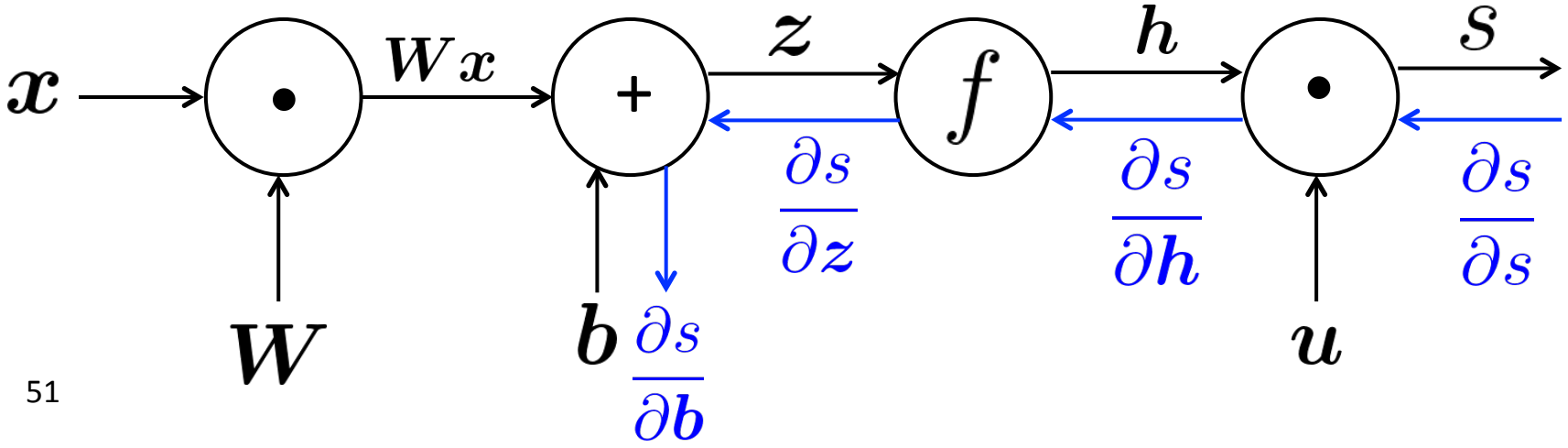


How to train the model?

Backpropagation

- Go backwards along edges
  - Pass along **gradients**

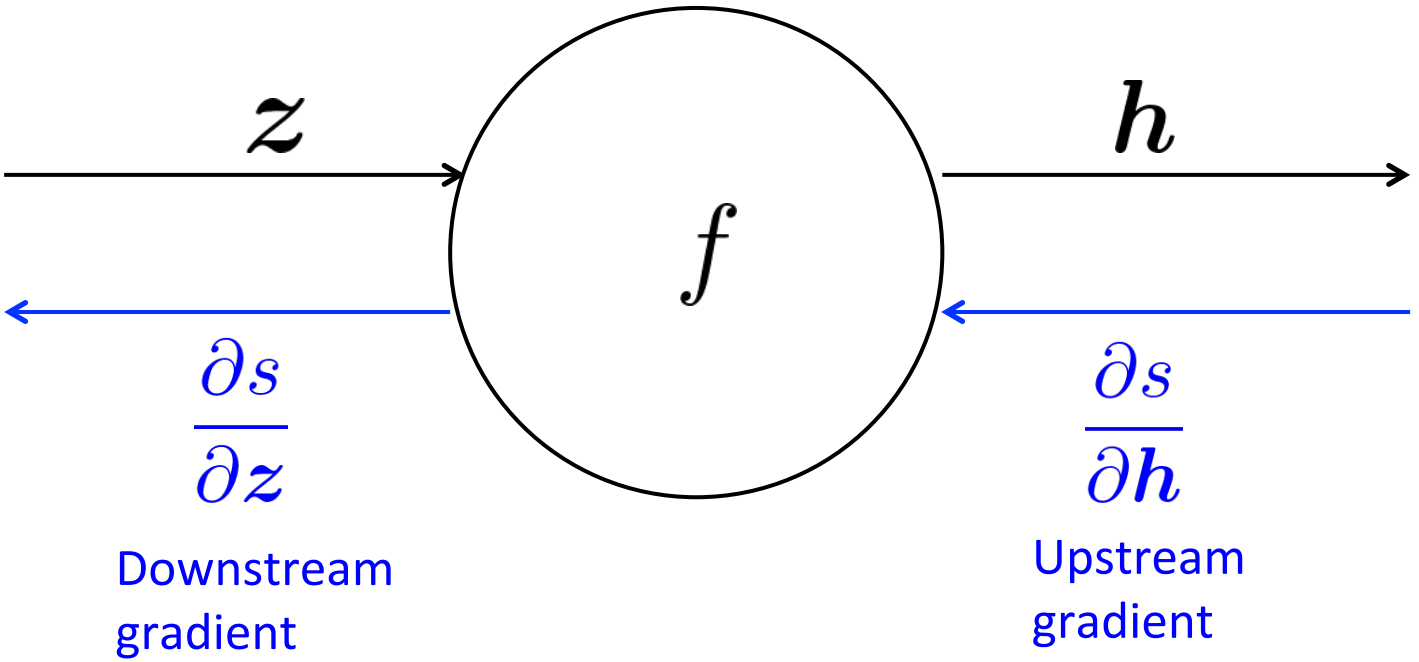
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



Backpropagation: Single Node

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

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

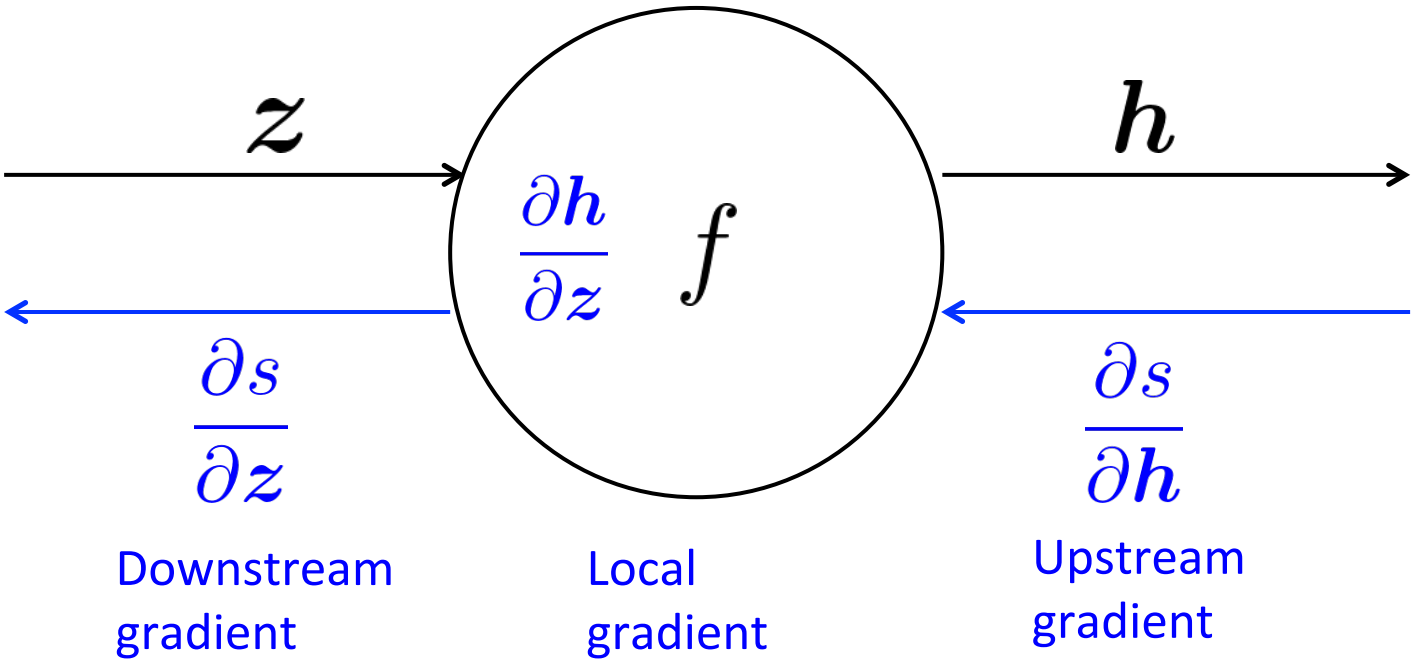


Backpropagation: Single Node

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

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

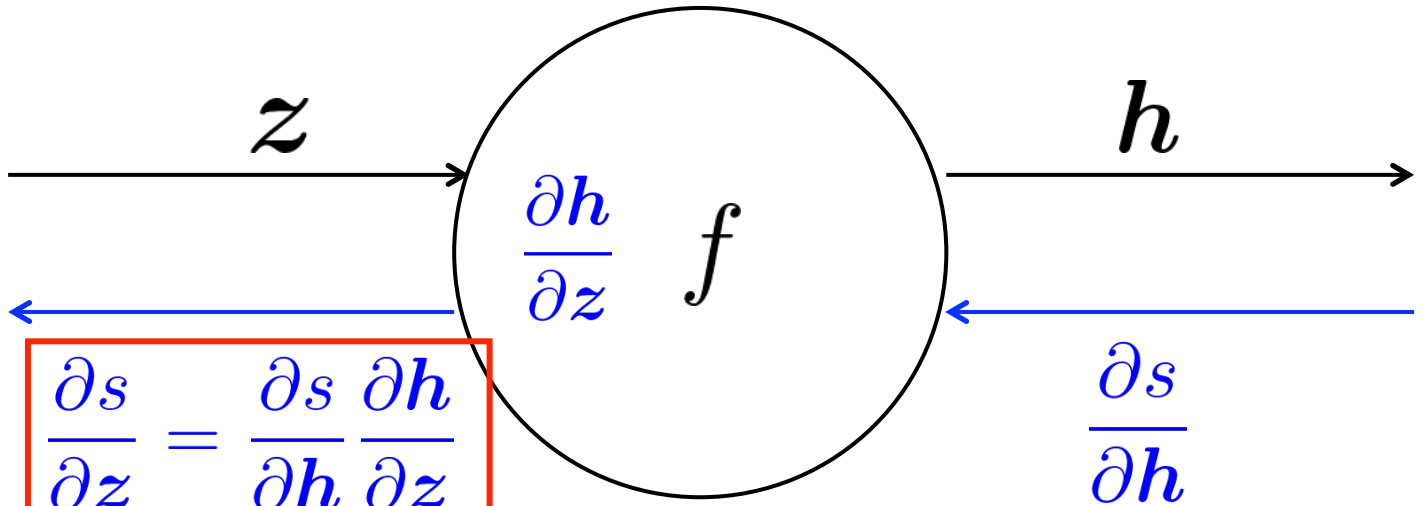
Can you see the chain rule here?



Backpropagation: Single Node

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

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



Summary

- Backpropagation: recursively apply the chain rule along computational graph
  - [downstream gradient] = [upstream gradient] x [local gradient]
- Forward pass: compute results of operation and save intermediate values
- Backward: apply chain rule to compute gradient

Neural Network Frameworks

Static Frameworks

theano

Caffe

mxnet

TensorFlow

+Gluon

+Fold

Dynamic Frameworks (Recommended!)

∂y/net

Chainer

PYTORCH

Basic Process in Dynamic Neural Network Frameworks

- Create a model
- For each example
  - **create a graph** that represents the computation you want
  - **calculate the result** of that computation
  - if training, perform **back propagation and update**

DyNet

- Examples in this class will be in DyNet:
  - **intuitive**, program like you think (c.f. TensorFlow, Theano)
  - **fast for complicated networks** on CPU (c.f. autodiff libraries, Chainer, PyTorch)
  - has **nice features to make efficient implementation easier** (automatic batching)

Computation Graph and Expressions

```
import dynet as dy

dy.renew_cg() # create a new computation graph

v1 = dy.inputVector([1,2,3,4])
v2 = dy.inputVector([5,6,7,8])
# v1 and v2 are expressions

v3 = v1 + v2
v4 = v3 * 2
v5 = v1 + 1

v6 = dy.concatenate([v1,v2,v3,v5])

print v6
print v6.npvalue()
```

Computation Graph and Expressions

```
import dynet as dy

dy.renew_cg() # create a new computation graph

v1 = dy.inputVector([1,2,3,4])
v2 = dy.inputVector([5,6,7,8])
# v1 and v2 are expressions

v3 = v1 + v2
v4 = v3 * 2
v5 = v1 + 1

v6 = dy.concatenate([v1,v2,v3,v5])

print v6  expression 5/1
print v6.npvalue()
```

Computation Graph and Expressions

```
import dynet as dy

dy.renew_cg() # create a new computation graph

v1 = dy.inputVector([1,2,3,4])
v2 = dy.inputVector([5,6,7,8])
# v1 and v2 are expressions

v3 = v1 + v2
v4 = v3 * 2
v5 = v1 + 1

v6 = dy.concatenate([v1,v2,v3,v5])

print v6
print v6.npvalue()
```

```
array([  1.,   2.,   3.,   4.,   2.,   4.,   6.,   8.,   4.,   8.,  12.,  16.])
```

Computation Graph and Expressions

- Create basic expressions.

- Combine them using *operations*.

- Expressions represent *symbolic computations*.

- Use:
  `.value()`
  `.npvalue()`
  `.scalar_value()`
  `.vec_value()`
  `.forward()`
  to perform actual computation.

Model and Parameters

- **Parameters** are the things that we optimize over (vectors, matrices).
- **Model** is a collection of parameters.
- Parameters **out-live** the computation graph.

Model and Parameters

```
model = dy.Model()

pW = model.add_parameters((20,4))
pb = model.add_parameters(20)
```

```
dy.renew_cg()
x = dy.inputVector([1,2,3,4])
W = dy.parameter(pW) # convert params to expression
b = dy.parameter(pb) # and add to the graph

y = W * x + b
```

Trainers and Backdrop

- Initialize a **Trainer** with a given model.
- Compute gradients by calling `expr.backward()` from a scalar node.
- Call `trainer.update()` to update the model parameters using the gradients.

Trainers and Backdrop

```
model = dy.Model()

trainer = dy.SimpleSGDTrainer(model)

p_v = model.add_parameters(10)

for i in xrange(10):
    dy.renew_cg()

    v = dy.parameter(p_v)
    v2 = dy.dot_product(v,v)
    v2.forward()

    v2.backward()  # compute gradients

    trainer.update()
```

Trainers and Backdrop

```
model = dy.Model()

trainer =

p_v = mode

for i in x
    dy.rer

    v = dy
    v2 = d
    v2.for

    v2.backward()  # compute gradients

    trainer.update()
```

```
dy.SimpleSGDTrainer(model,...)

dy.MomentumSGDTrainer(model,...)

dy.AdagradTrainer(model,...)

dy.AdadeltaTrainer(model,...)

dy.AdamTrainer(model,...)
```

Training with DyNet

- Create model, add parameters, create trainer.
- For each training example:
  - create computation graph for the loss
  - run forward (compute the loss)
  - run backward (compute the gradients)
  - update parameters

Example Implementation (in DyNet)

Predicting the next word in a sentence…

Are These Sentences OK?

- Jane went to the store.
- store to Jane went the.
- Jane went store.
- Jane goed to the store.
- The store went to Jane.
- The food truck went to Jane.

How to check whether these sentences are ok or not?

Calculating the Probability of a Sentence

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

Next Word

Context

The big problem: How do we predict

$$P(x_i \mid x_1, \ldots, x_{i-1})$$

?!?!

Review: Count-based Language Models

Count-based Language Models

- Count up the frequency and divide:

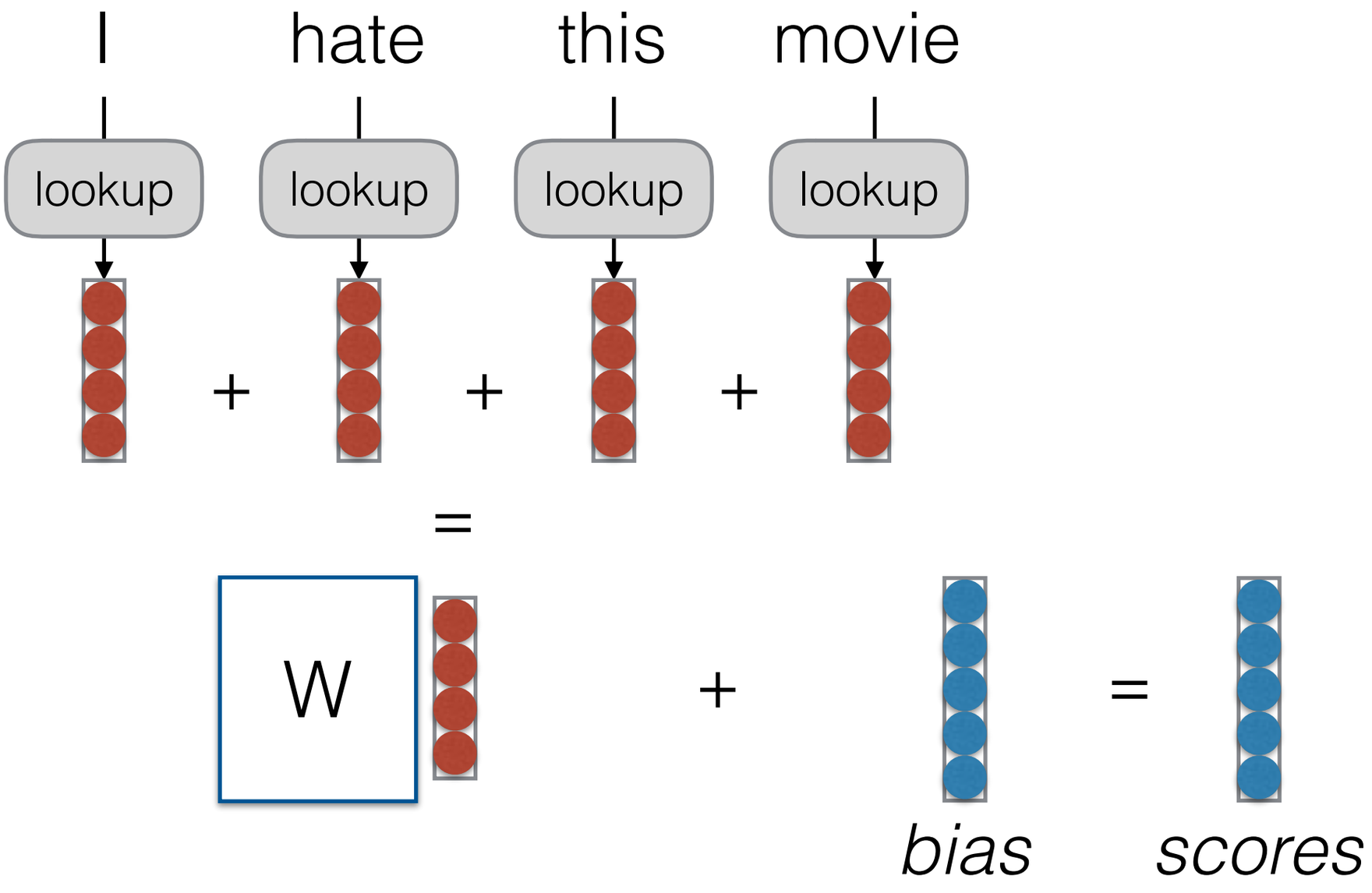
$$P_{ML}(x_i \mid x_{i-n+1}, \ldots, x_{i-1}) := \frac{c(x_{i-n+1}, \ldots, x_i)}{c(x_{i-n+1}, \ldots, x_{i-1})}$$

  - Add smoothing, to deal with zero counts:

$$\begin{aligned} P(x_i \mid x_{i-n+1}, \ldots, x_{i-1}) =& \lambda P_{ML}(x_i \mid x_{i-n+1}, \ldots, x_{i-1}) \\ &+ (1-\lambda) P(x_i \mid x_{1-n+2}, \ldots, x_{i-1}) \end{aligned}$$

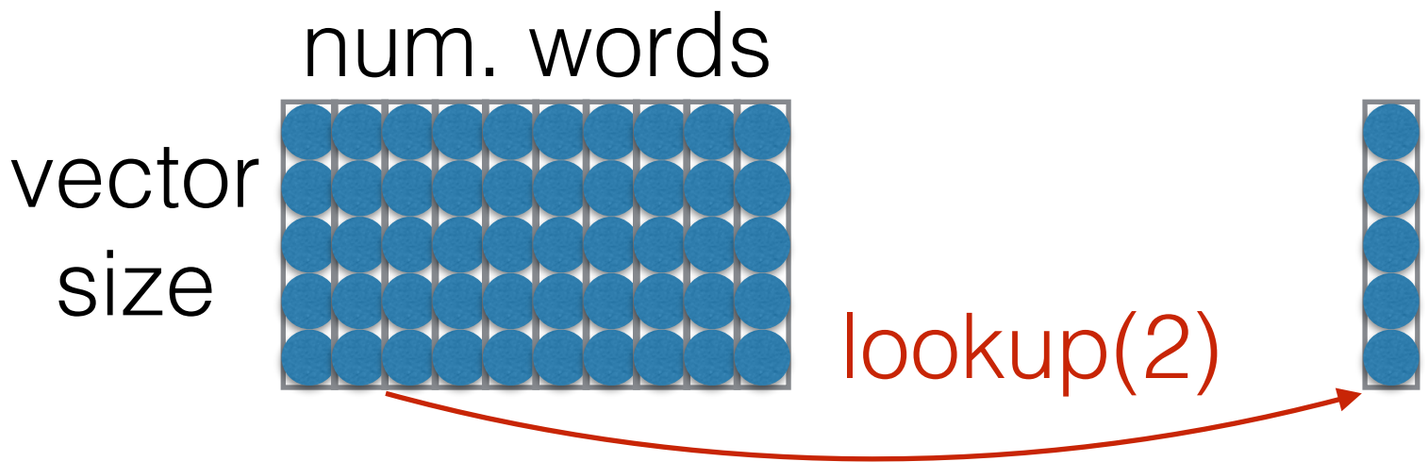
  - Modified Kneser-Ney smoothing

Continuous Bag of Words (CBOW)

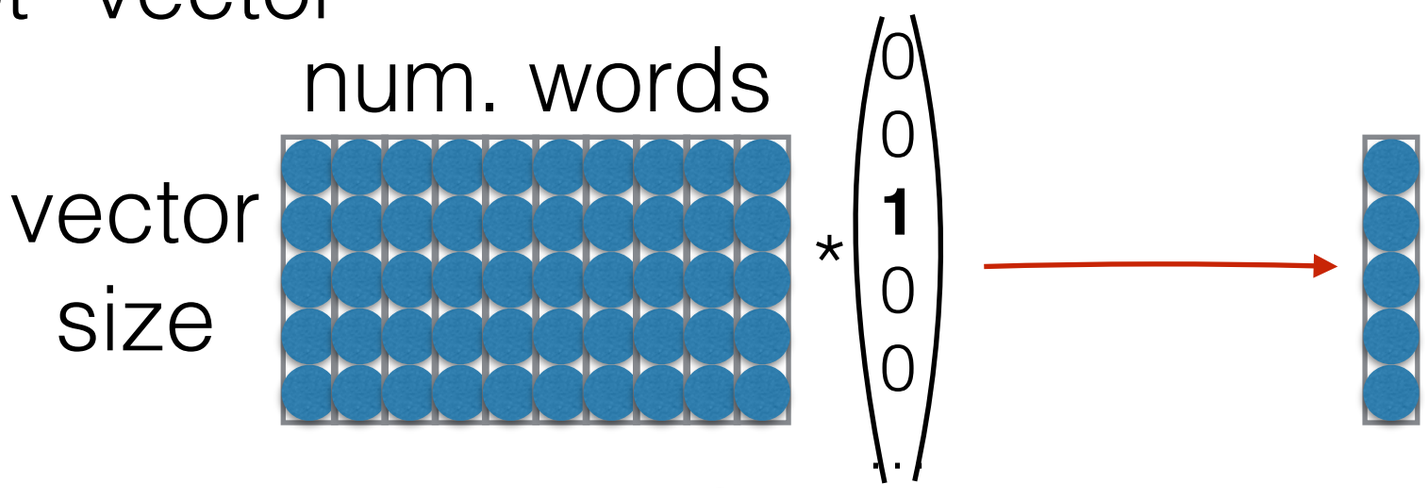


A Note: “Lookup”

- Lookup can be viewed as “grabbing” a single vector from a big matrix of word embeddings



- Similarly, can be viewed as multiplying by a “one-hot” vector



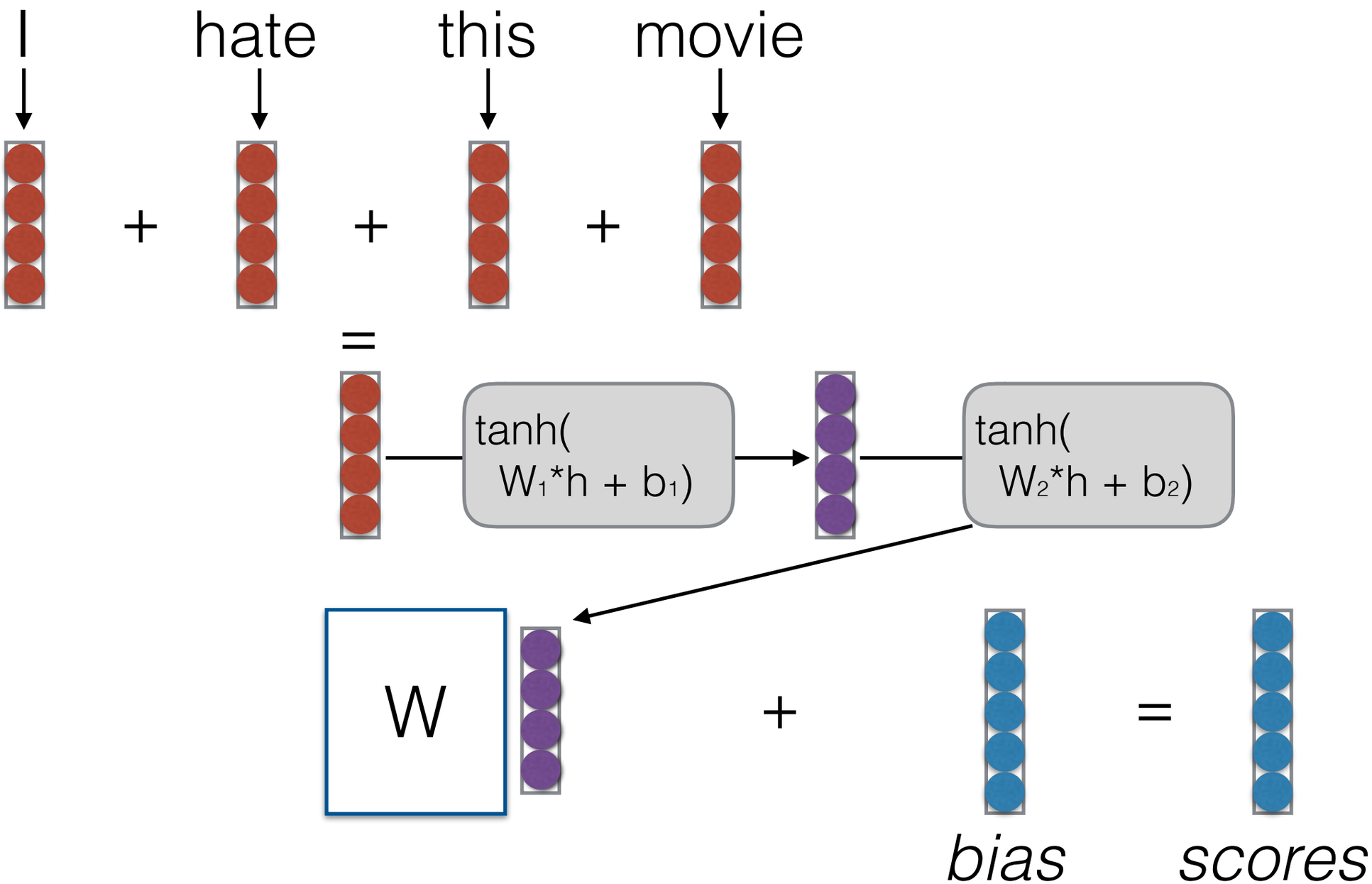
- Former tends to be faster

WORD EMBEDDINGS AND LOOKUP PARAMETERS

- In DyNet, embeddings are implemented using LookupParameters.

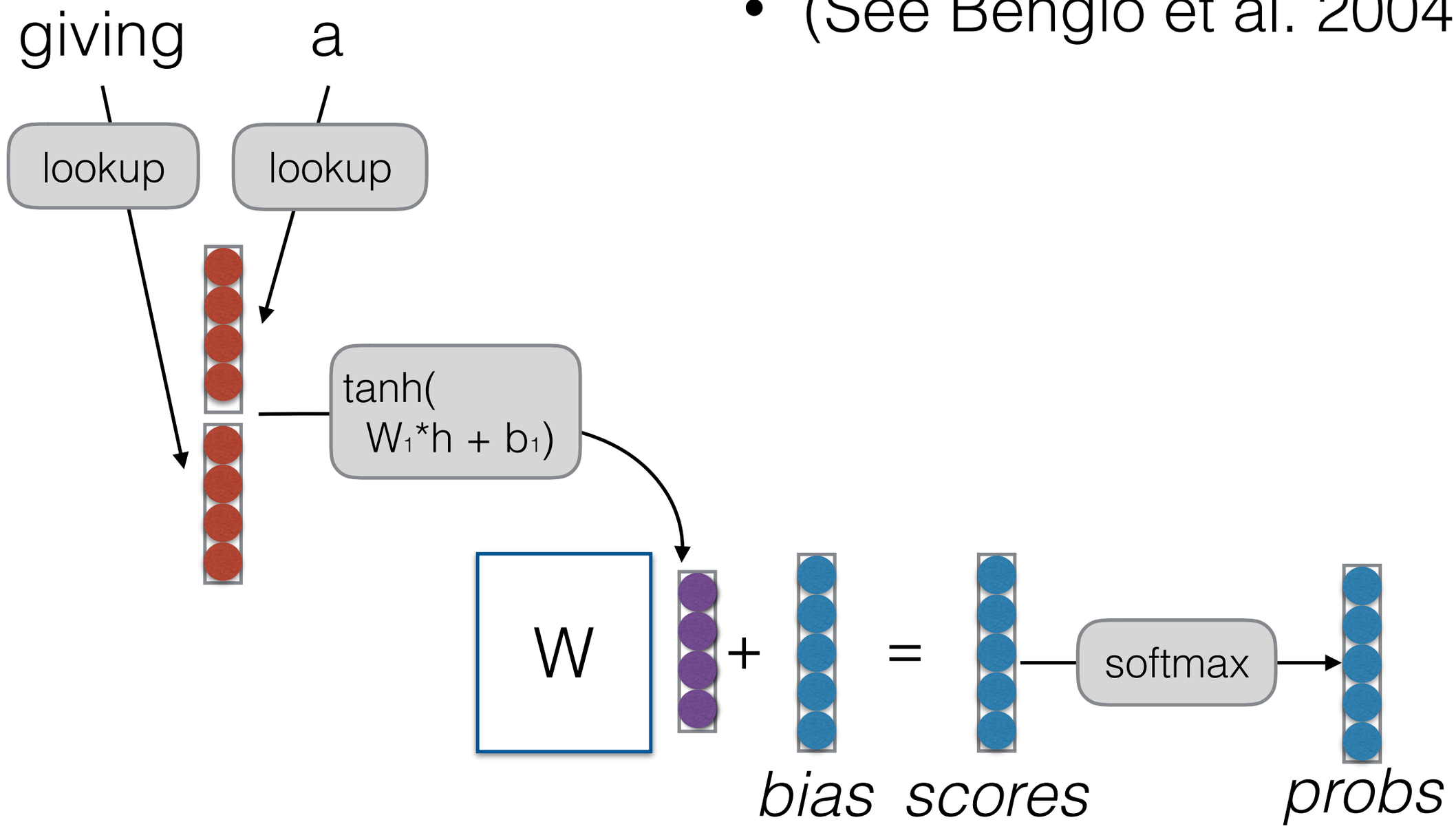
```
vocab_size = 10000
emb_dim = 200


E = model.add_lookup_parameters((vocab_size, emb_dim))


dy.renew_cg()
x=dy.lookup(E,5)
# or
x = E[5]  #x is an expression
```

Deep CBOW

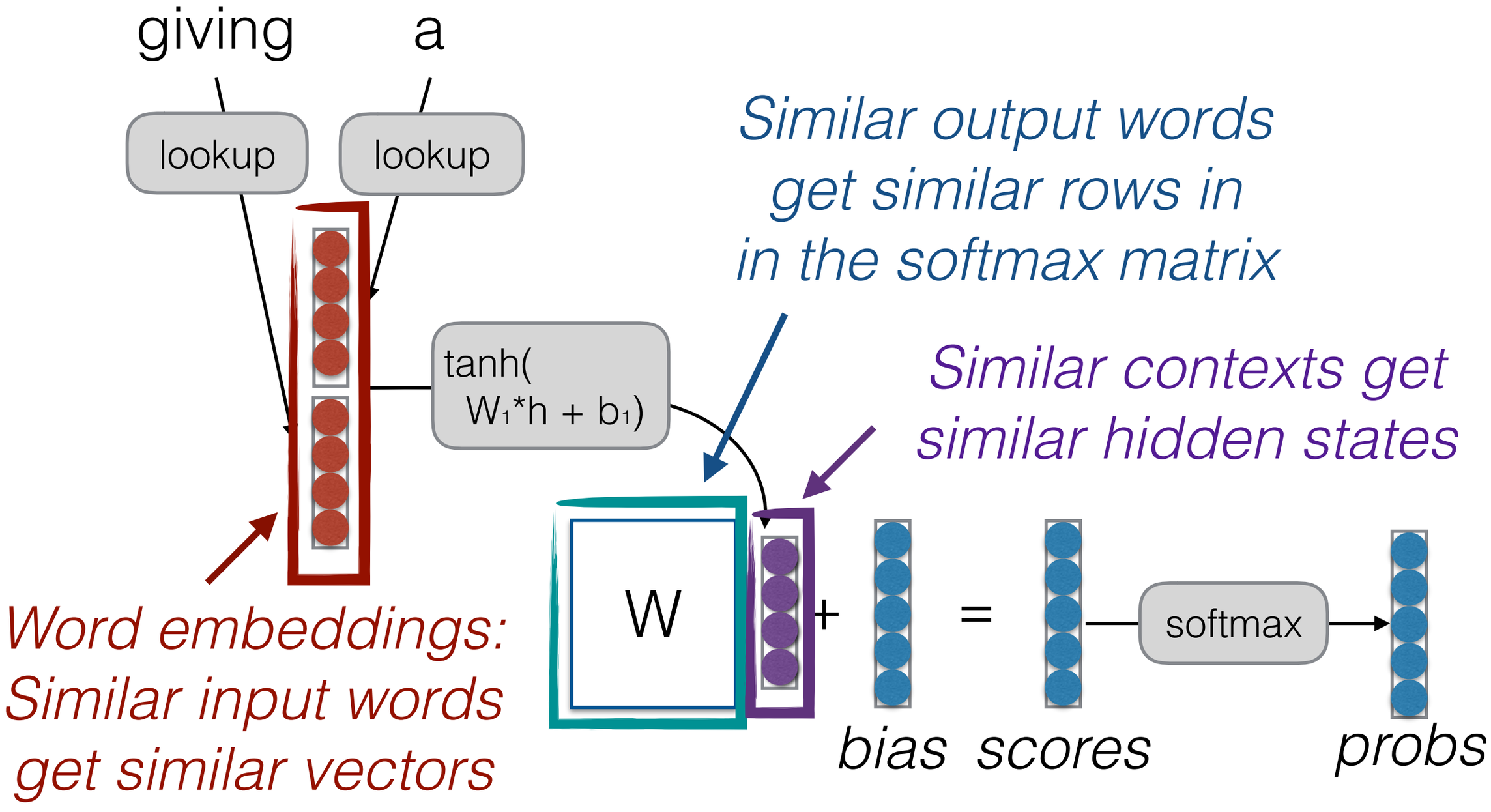


Neural Language Models

- (See Bengio et al. 2004)



Where is Strength Shared?



Similar output words get similar rows in in the softmax matrix

Similar contexts get similar hidden states

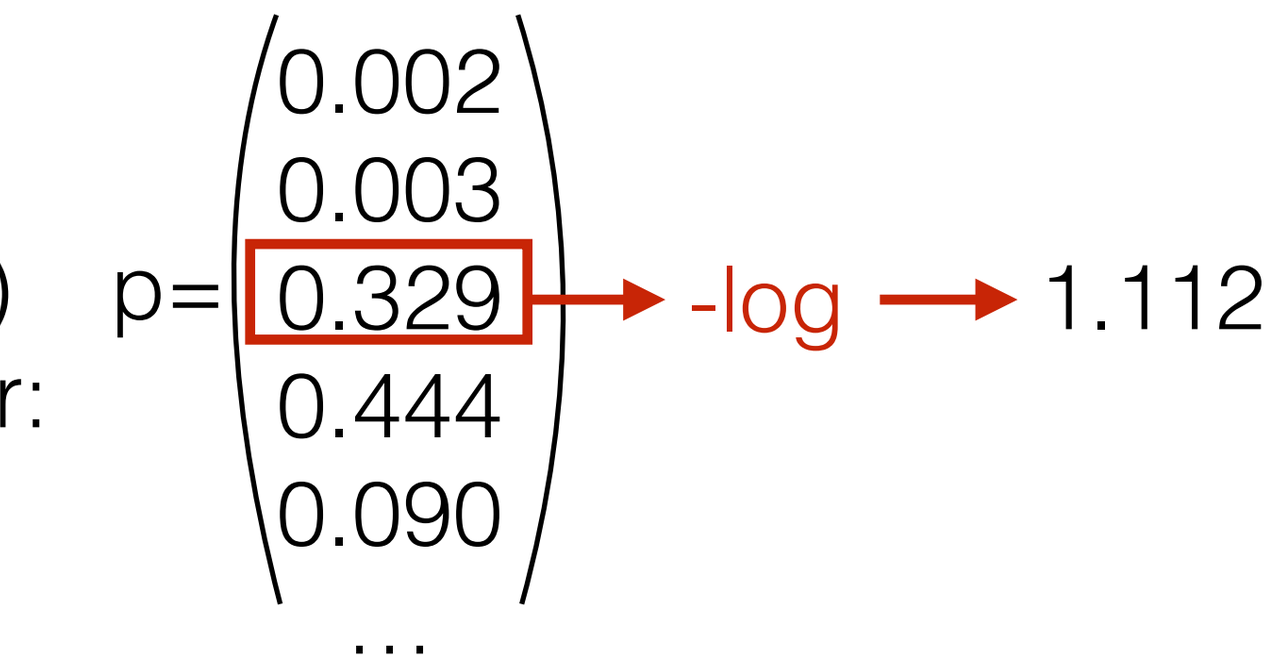
Word embeddings: Similar input words get similar vectors

Training a Model

- **Reminder:** to train, we calculate a "loss function" (a measure of how bad our predictions are), and move the parameters to reduce the loss

- The most common loss function for probabilistic models is "negative log likelihood"

If element 3 (or zero-indexed, 2) is the correct answer:



Parameter Update

- Back propagation allows us to calculate the derivative of the loss with respect to the parameters

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}}$$

- Simple stochastic gradient descent optimizes parameters according to the following rule

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\partial \ell}{\partial \boldsymbol{\theta}}$$

Choosing a Vocabulary

Unknown Words

- Necessity for UNK words
  - We won't have all the words in the world in training data
  - Larger vocabularies require more memory and computation time
- Common ways:
  - Frequency threshold (usually UNK <= 1)
  - Rank threshold

REFERENCES

- Introduction to Neural Networks, Philipp Koehn, 3 October 2017
- Artificial Neural Networks and Deep Learning, Christian Borgelt, University of Konstanz, Germany
- Natural Language Processing with Deep Learning, Richard Socher, Kevin Clark, Stanford University, 2017
- Richard Socher, Neural Networks, 2018