

CS11-711 Advanced NNLP

# Word Representation and Text Classifiers

Graham Neubig



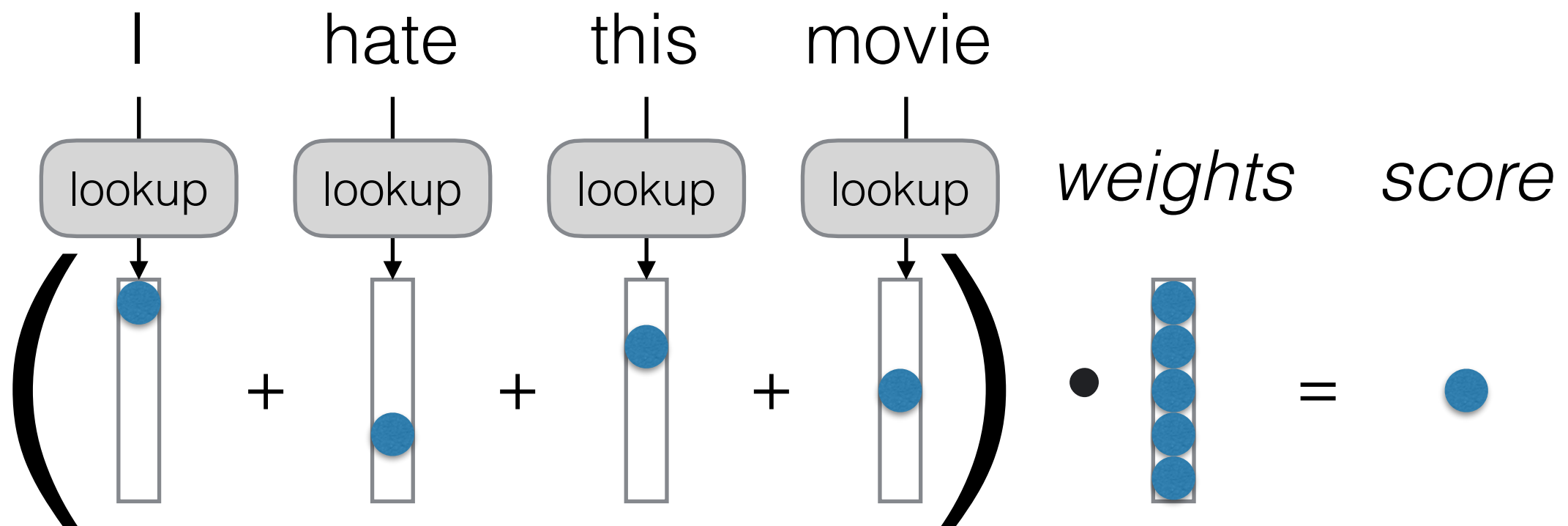
**Carnegie Mellon University**

Language Technologies Institute

<https://phontron.com/class/anlp-fall2024/>

# Reminder:

## Bag of Words (BOW)



Features  $f$  are based on word identity, weights  $w$  learned

Which problems mentioned before would this solve?

# What's Missing in BOW?

- Handling of *conjugated or compound words*
  - I **love** this move -> I **loved** this movie

Subword  
Models

- Handling of *word similarity*
  - I **love** this move -> I **adore** this movie

Word  
Embeddings

- Handling of *combination features*
  - I **love** this movie -> I **don't love** this movie
  - I **hate** this movie -> I **don't hate** this movie

Neural  
Networks

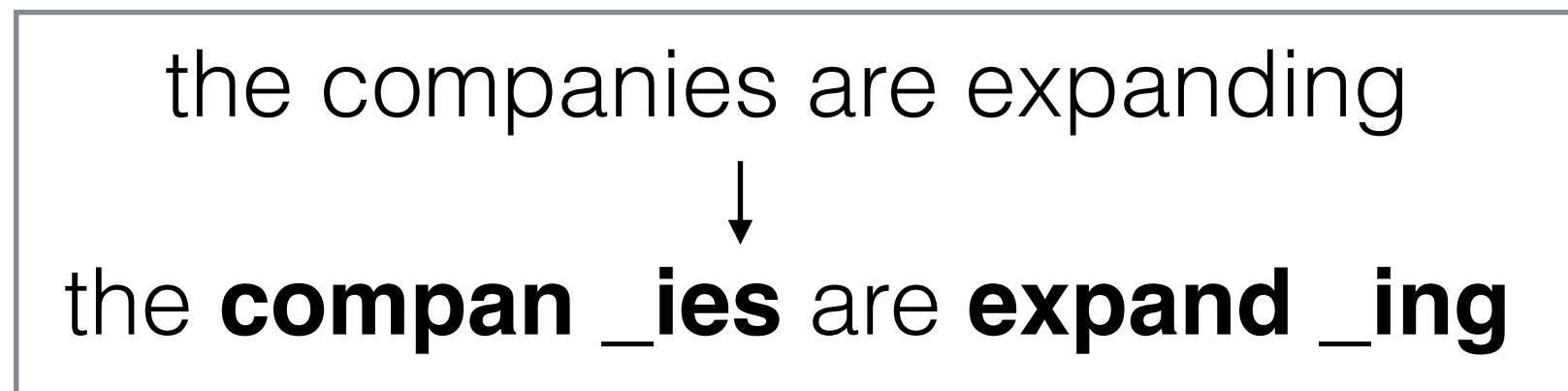
- Handling of *sentence structure*
  - It has an interesting story, **but** is boring overall

Sequence  
Models

# Subword Models

# Basic Idea

- Split less common words into multiple **subword tokens**



- Benefits:
  - **Share parameters** between word variants, compound words
  - Reduce parameter size, **save compute+memory**

# Byte Pair Encoding

(Sennrich+ 2015)

- Incrementally combine together the most frequent token pairs

```
{'l o w </w>': 5, 'l o w e r </w>': 2, 'n e w e s t </w>': 6, 'w i d e s t </w>': 3}
```

```
pairs = get_stats(vocab)
```

```
[(('e', 's'), 9), (('s', 't'), 9), (('t', '</w>'), 9), (('w', 'e'), 8), (('l', 'o'), 7), ...]
```

```
vocab = merge_vocab(pairs[0], vocab)
```

```
{'l o w </w>': 5, 'l o w e r </w>': 2, 'n e w e s t </w>': 6, 'w i d e s t </w>': 3}
```

```
pairs = get_stats(vocab)
```

```
[(('es', 't'), 9), (('t', '</w>'), 9), (('l', 'o'), 7), (('o', 'w'), 7), (('n', 'e'), 6)]
```

```
vocab = merge_vocab(pairs[0], vocab)
```

```
{'l o w </w>': 5, 'l o w e r </w>': 2, 'n e w e s t </w>': 6, 'w i d e s t </w>': 3}
```

Example code:

<https://github.com/neubig/anlp-code/tree/main/02-subwords>

# Unigram Models (Kudo 2018)

- Use a *unigram* LM that generates all words in the sequence independently (more next lecture)
- **Pick a vocabulary** that maximizes the log likelihood of the corpus given a fixed vocabulary size
  - Optimization performed using the EM algorithm (details not important for most people)
- **Find the segmentation** of the input that maximizes unigram probability

# SentencePiece

- A highly optimized library that makes it possible to train and use BPE and Unigram models

```
% spm_train --input=<input> \  
            --model_prefix=<model_name>  
            --vocab_size=8000 --character_coverage=1.0  
            --model_type=<type>  
  
% spm_encode --model=<model_file>  
            -output_format=piece < input > output
```

- Python bindings also available

<https://github.com/google/sentencepiece>



# Subword Considerations

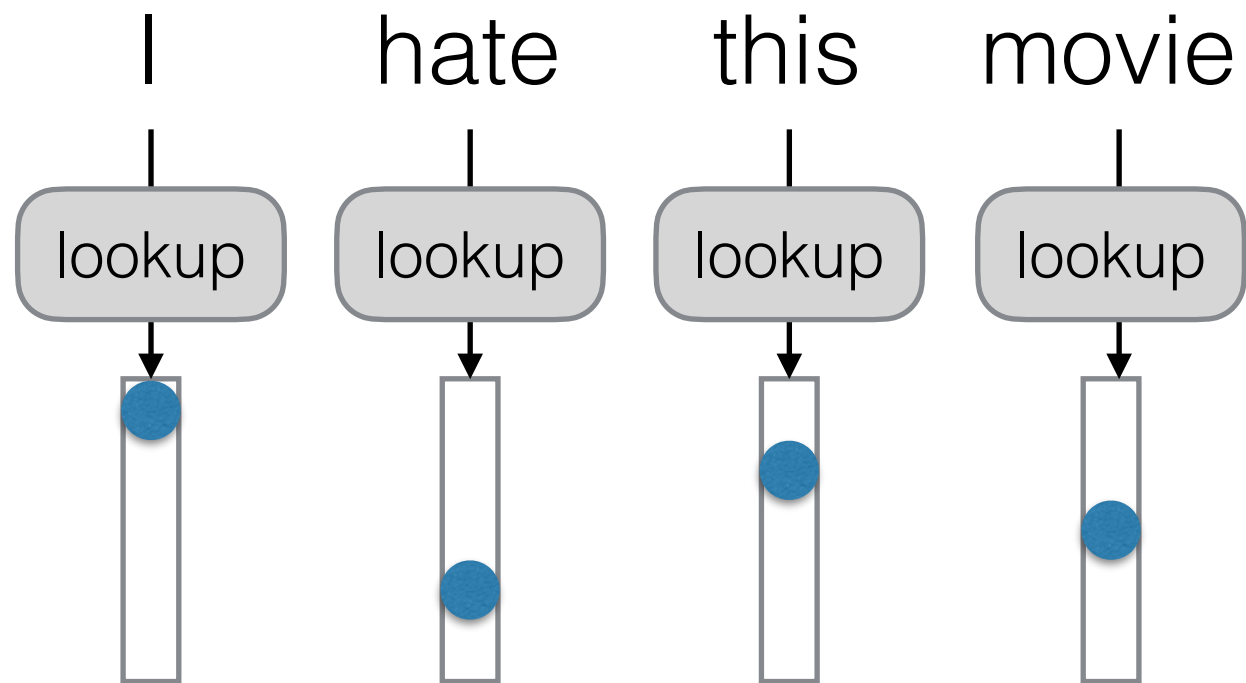
- **Multilinguality:** Subword models are hard to use multilingually because they will over-segment less common languages naively (Ács 2019)
  - *Work-around:* Upsample less represented languages
- **Arbitrariness:** Do we do “es t” or “e st”?
  - *Work-around:* “Subword regularization” samples different segmentations at training time to make models robust (Kudo 2018)

# Continuous Word Embeddings

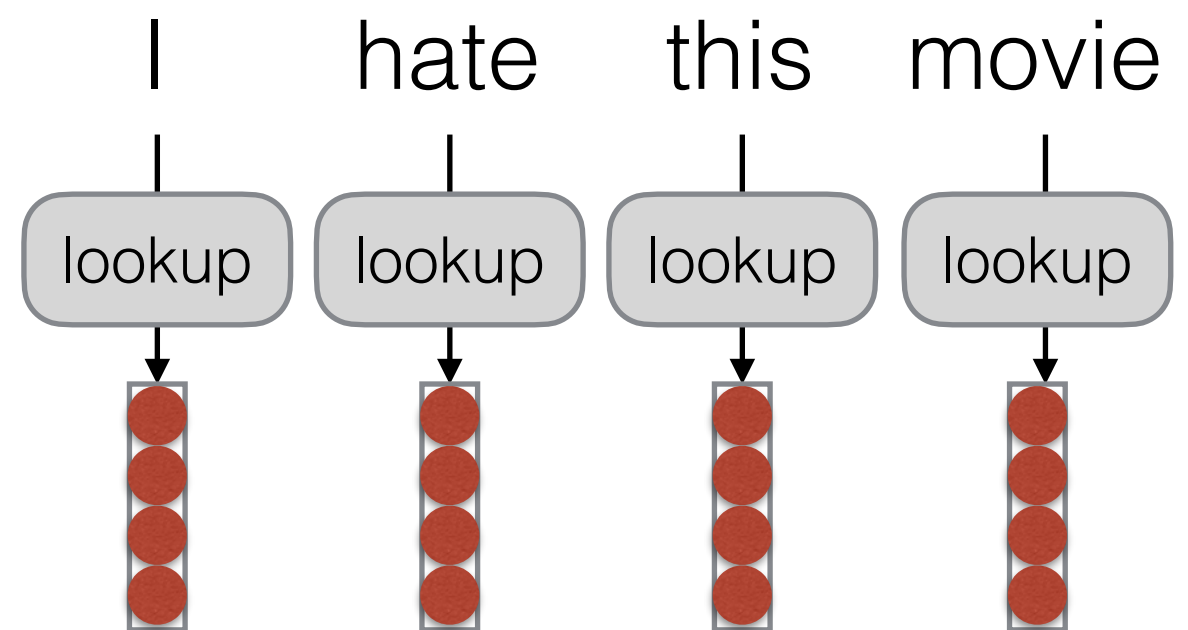
# Basic Idea

- Previously we represented words with a *sparse* vector with a single “1” — a **one-hot** vector
- Continuous word embeddings look up a *dense* vector

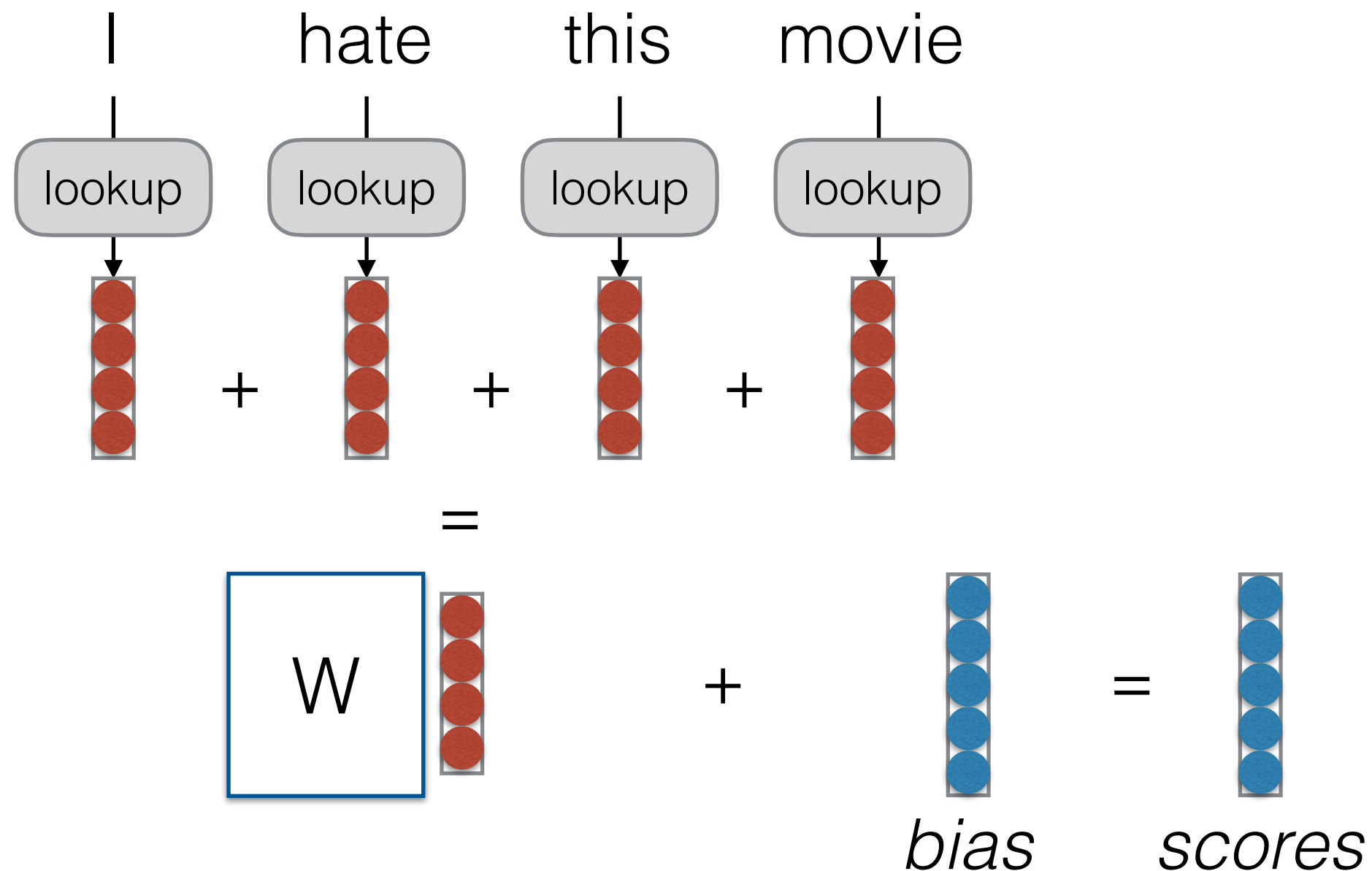
## One-hot Representations



## Dense Representations

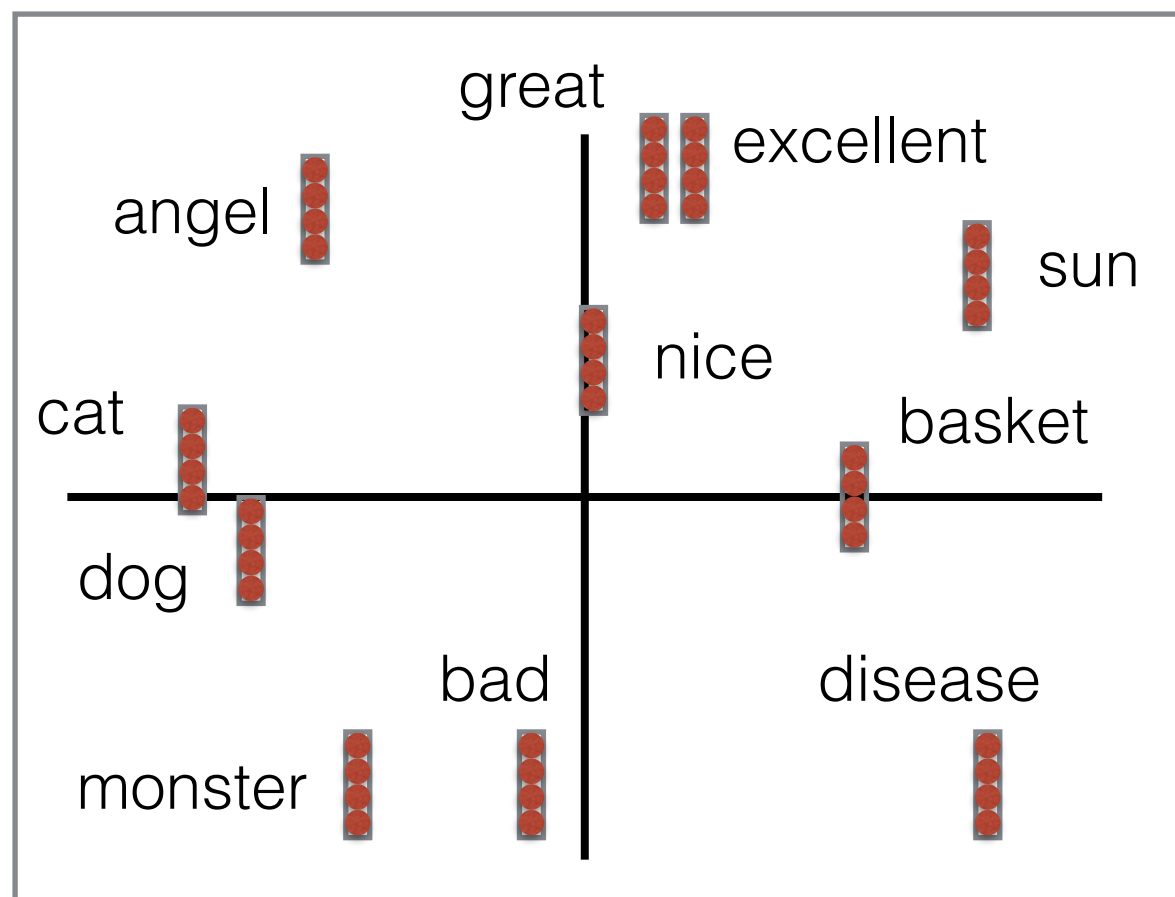


# Continuous Bag of Words (CBOW)



# What do Our Vectors Represent?

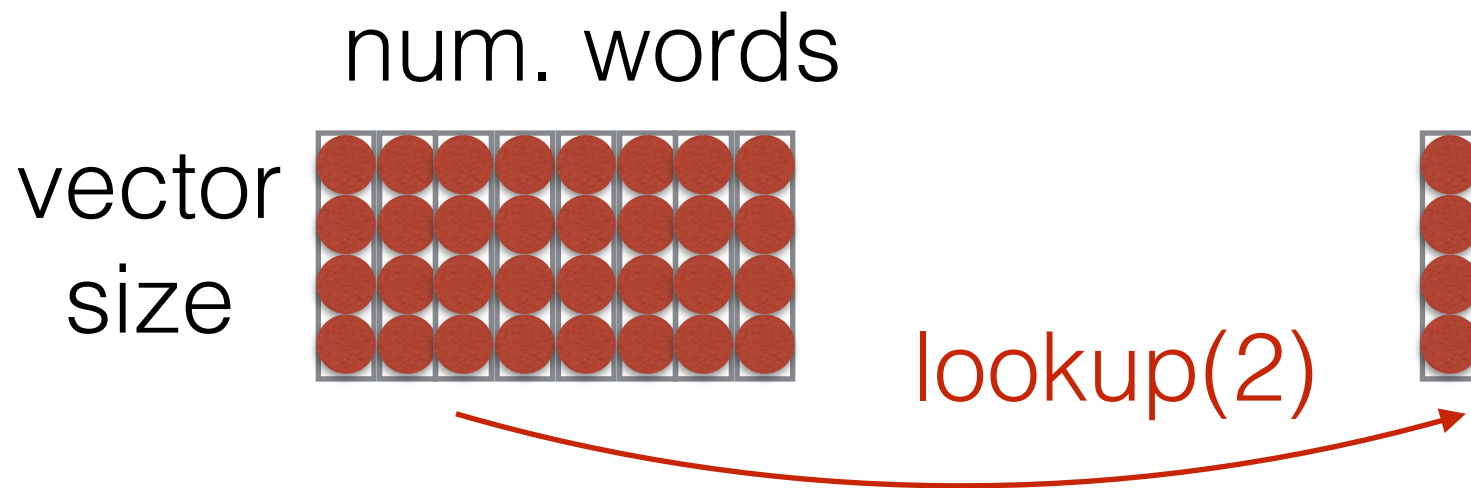
- No guarantees, but we hope that:
  - Words that are **similar** (syntactically, semantically, same language, etc.) are **close** in vector space
  - Each vector element is a **features** (e.g. is this an animate object? is this a positive word, etc.)



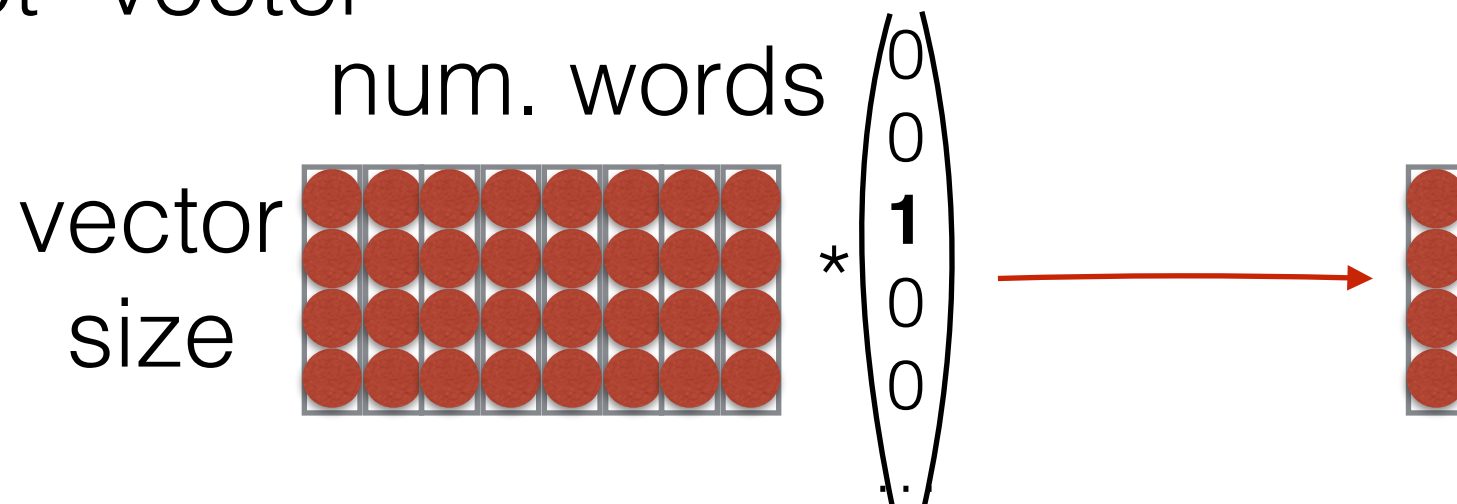
Shown in 2D, but  
in reality we use  
512, 1024, etc.

# 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

# Training a More Complex Model

# Reminder: Simple Training of BOW Models

- Use an algorithm called “structured perceptron”

```
feature_weights = {}  
for x, y in data:  
    # Make a prediction  
    features = extract_features(x)  
    predicted_y = run_classifier(features)  
    # Update the weights if the prediction is wrong  
    if predicted_y != y:  
        for feature in features:  
            feature_weights[feature] = (  
                feature_weights.get(feature, 0) +  
                y * features[feature]  
            )
```

Full Example:

<https://github.com/neubig/anlp-code/tree/main/01-simpleclassifier>



# How do we Train More Complex Models?

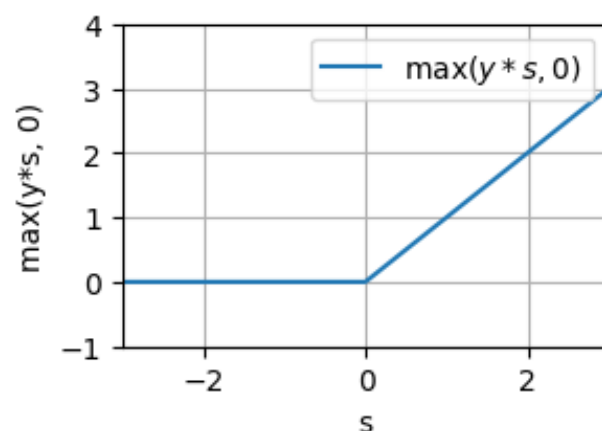
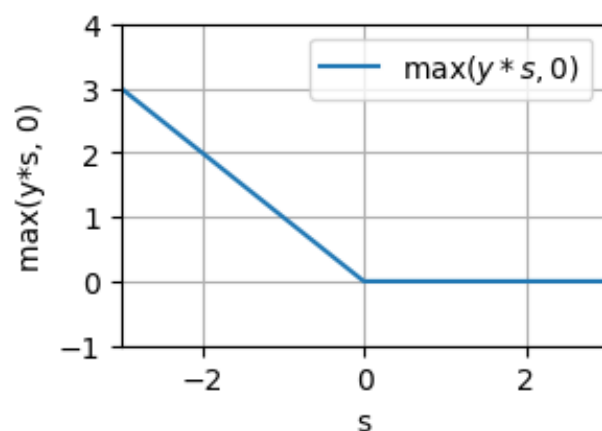
- We use *gradient descent*
  - Write down a *loss function*
  - *Calculate derivatives* of the loss function wrt the parameters
  - Move in the parameters in the direction that *reduces the loss function*

# Loss Function

- A value that gets lower as the model gets better
- Examples from binary classification using score  $s(x)$

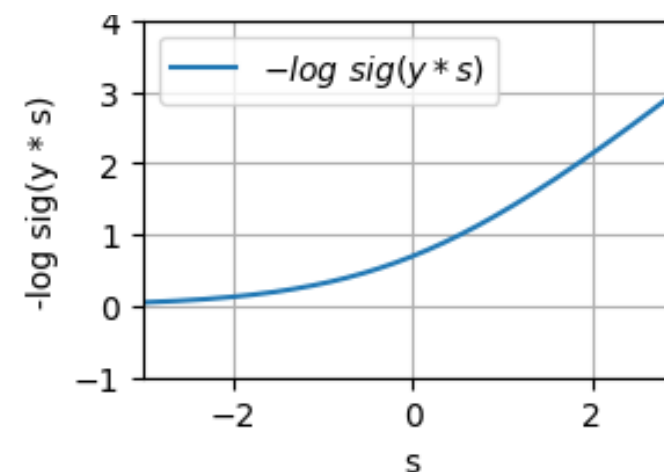
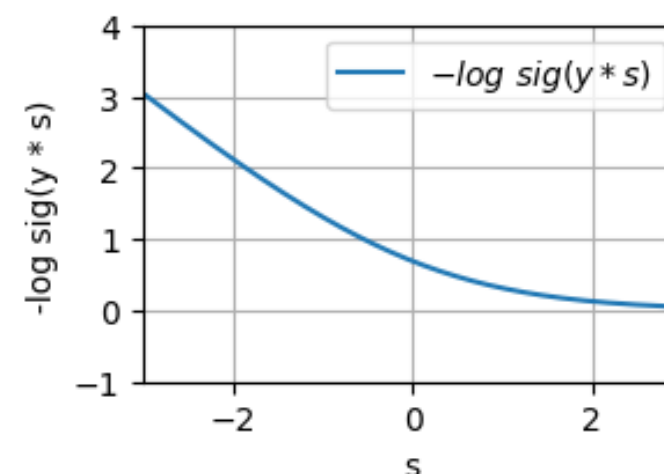
## Hinge Loss

$$\ell = \max(-y * s)$$



## Sigmoid + Negative Log Likelihood

$$\sigma(y * s) = \frac{1}{1 + e^{-(y*s)}} \quad \ell = -\log \sigma(y * s)$$



more closely linked to acc

probabilistic interpretation, gradients everywhere

# Calculating Derivatives

- Calculate the derivative of the parameter given the loss function
- Example from BOW model + hinge loss

$$\frac{\partial \max(0, -y * \sum_i^{|\mathcal{V}|} w_i \text{freq}(v_i, x))}{\partial w_i} =$$

$$\begin{cases} -y \cdot \text{freq}(v_i, x) & \text{if } -y \cdot \sum_i^{|\mathcal{V}|} w_i \text{freq}(v_i, x) > 0 \\ 0 & \text{otherwise} \end{cases}$$

# Optimizing Parameters

- Standard stochastic gradient descent does

$$g_t = \underbrace{\nabla_{\theta_{t-1}} \ell(\theta_{t-1})}_{\text{Gradient of Loss}}$$

$$\theta_t = \theta_{t-1} - \underbrace{\eta}_{\text{Learning Rate}} g_t$$

- There are many other optimization options! (see Ruder 2016 in references)

# What is this Algorithm?

```
feature_weights = {}  
for x, y in data:  
    # Make a prediction  
    features = extract_features(x)  
    predicted_y = run_classifier(features)  
    # Update the weights if the prediction is wrong  
    if predicted_y != y:  
        for feature in features:  
            feature_weights[feature] = (  
                feature_weights.get(feature, 0) +  
                y * features[feature]  
            )
```

- **Loss function:** Hinge Loss
- **Optimizer:** SGD w/ learning rate 1

# Combination Features

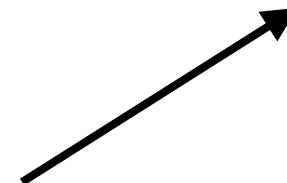
# Combination Features

I don't love this movie



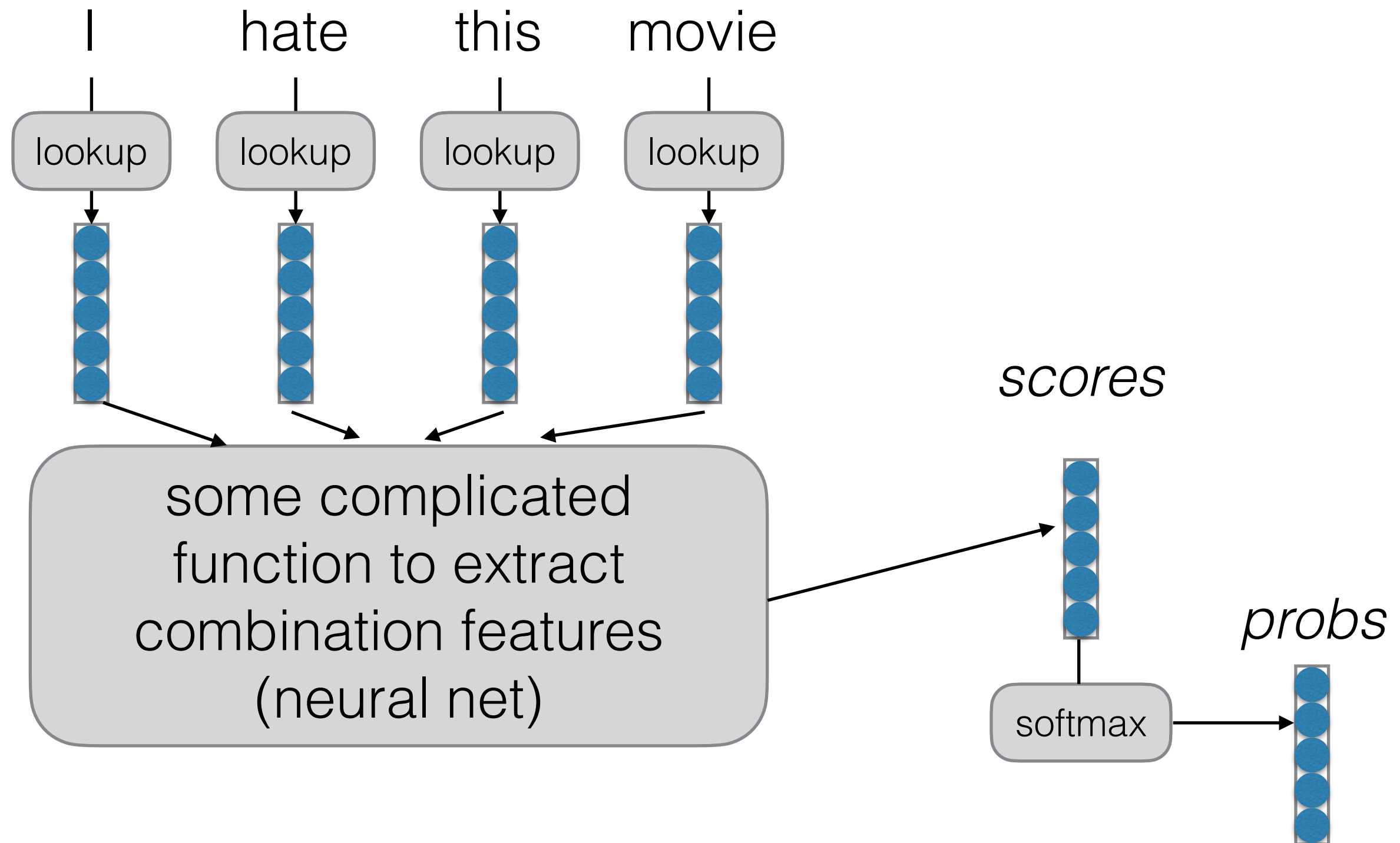
very good  
good  
neutral  
bad  
very bad

There's nothing I don't  
love about this movie



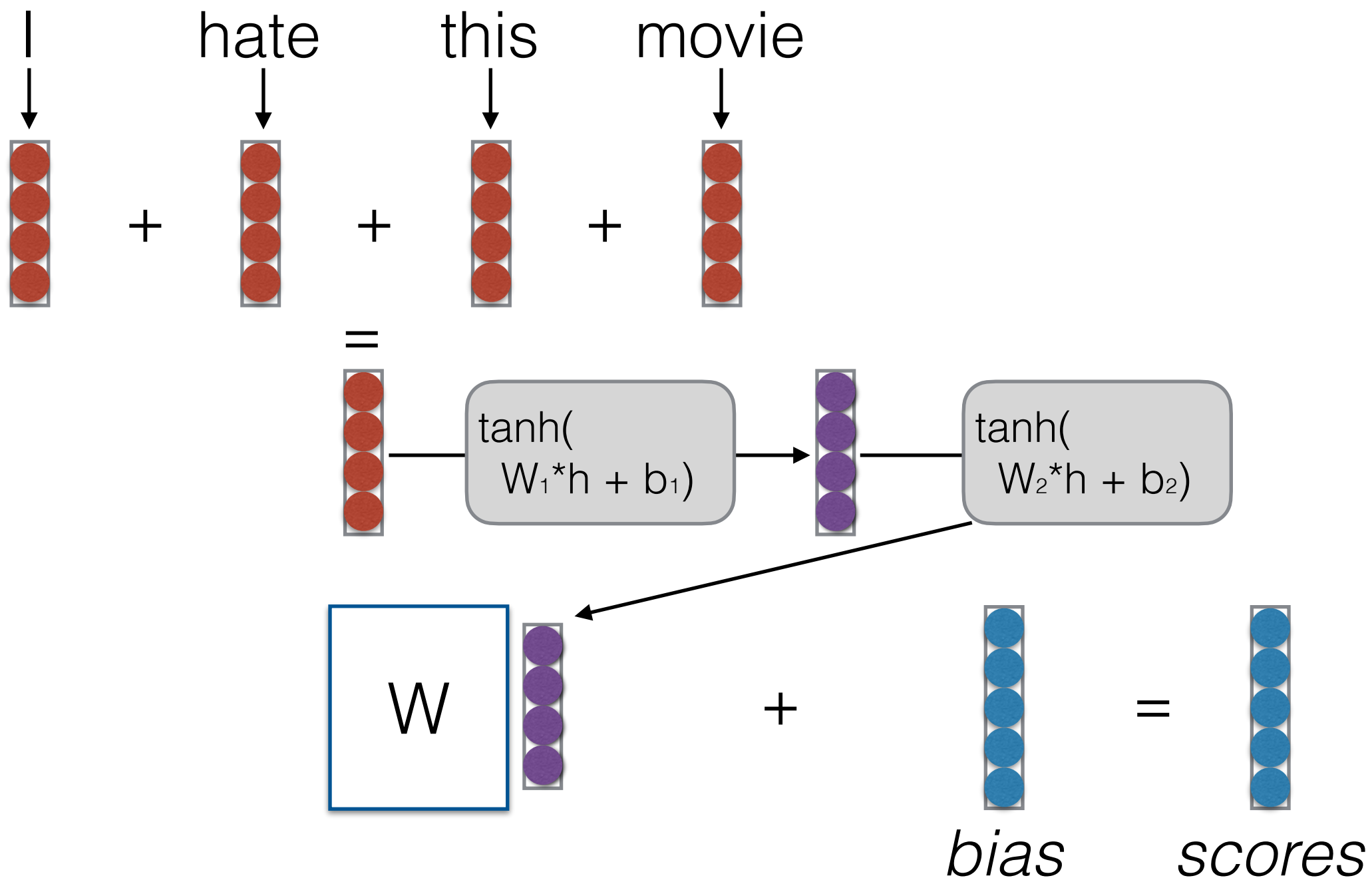
very good  
good  
neutral  
bad  
very bad

# Basic Idea of Neural Networks (for NLP Prediction Tasks)





# Deep CBOW



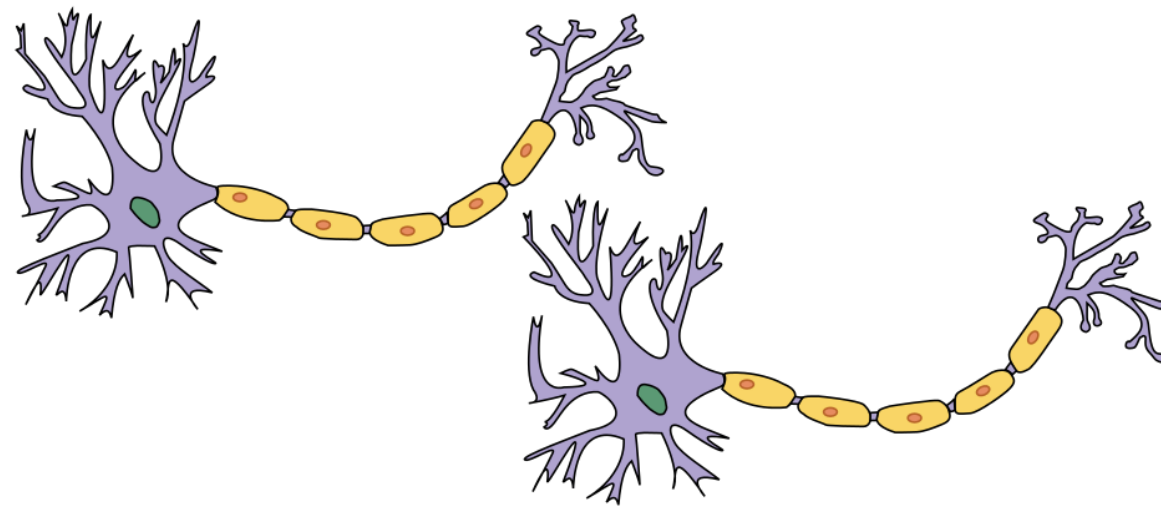
# What do Our Vectors Represent?

- Now things are more interesting!
- We can learn feature combinations (a node in the second layer might be “feature 1 AND feature 5 are active”)
- e.g. capture things such as “not” AND “hate”

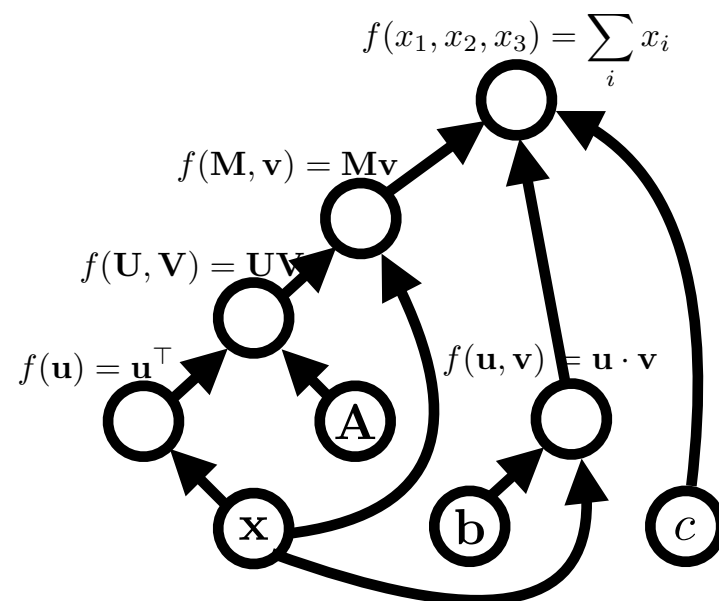
# What is a Neural Net?: Computation Graphs

# “Neural” Nets

Original Motivation: Neurons in the Brain



Current Conception: Computation Graphs



expression:

**x**

graph:

A **node** is a {tensor, matrix, vector, scalar} value

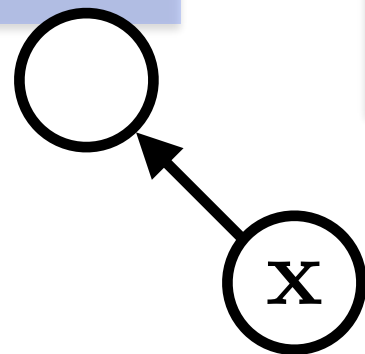
**x**

An **edge** represents a function argument (and also an data dependency). They are just pointers to nodes.

A **node** with an incoming **edge** is a **function** of that edge's tail node.

A **node** knows how to compute its value and the *value of its derivative w.r.t each argument (edge) times a derivative of an arbitrary input*  $\frac{\partial \mathcal{F}}{\partial f(\mathbf{u})}$ .

$$f(\mathbf{u}) = \mathbf{u}^\top$$



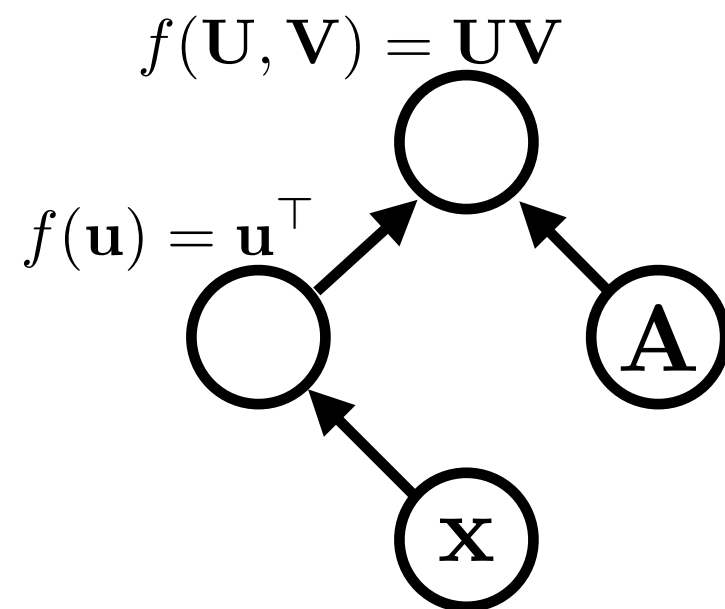
$$\frac{\partial f(\mathbf{u})}{\partial \mathbf{u}} \frac{\partial \mathcal{F}}{\partial f(\mathbf{u})} = \left( \frac{\partial \mathcal{F}}{\partial f(\mathbf{u})} \right)^\top$$

expression:

$$\mathbf{x}^\top \mathbf{A}$$

graph:

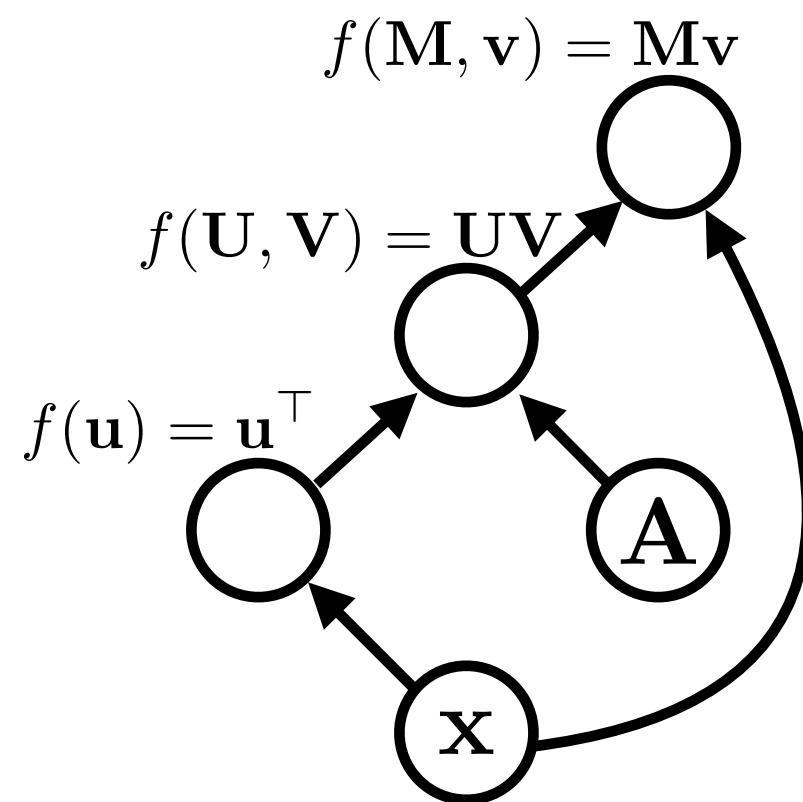
Functions can be nullary, unary,  
binary, ...  $n$ -ary. Often they are unary or binary.



expression:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x}$$

graph:



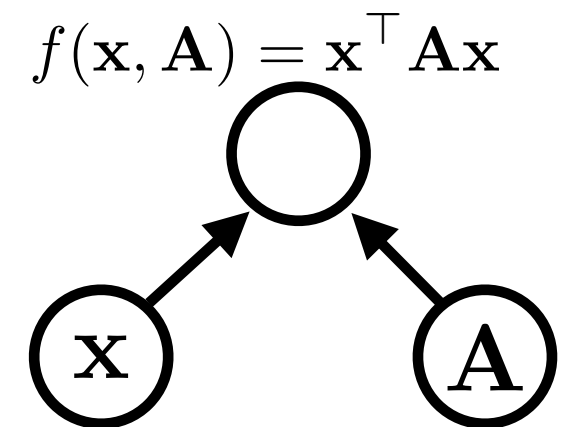
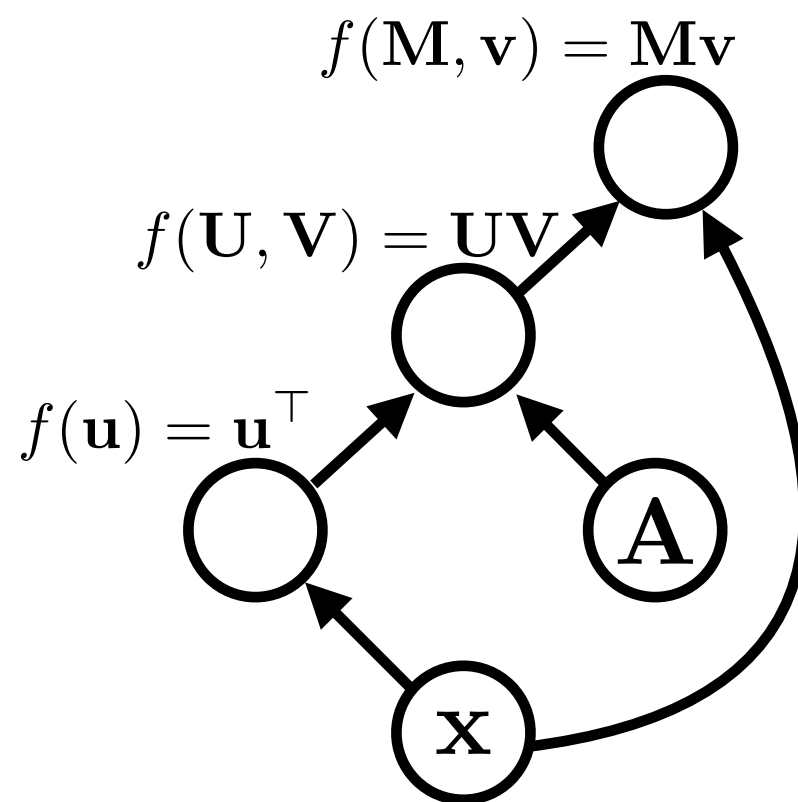
Computation graphs are directed and acyclic (in DyNet)



expression:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x}$$

graph:



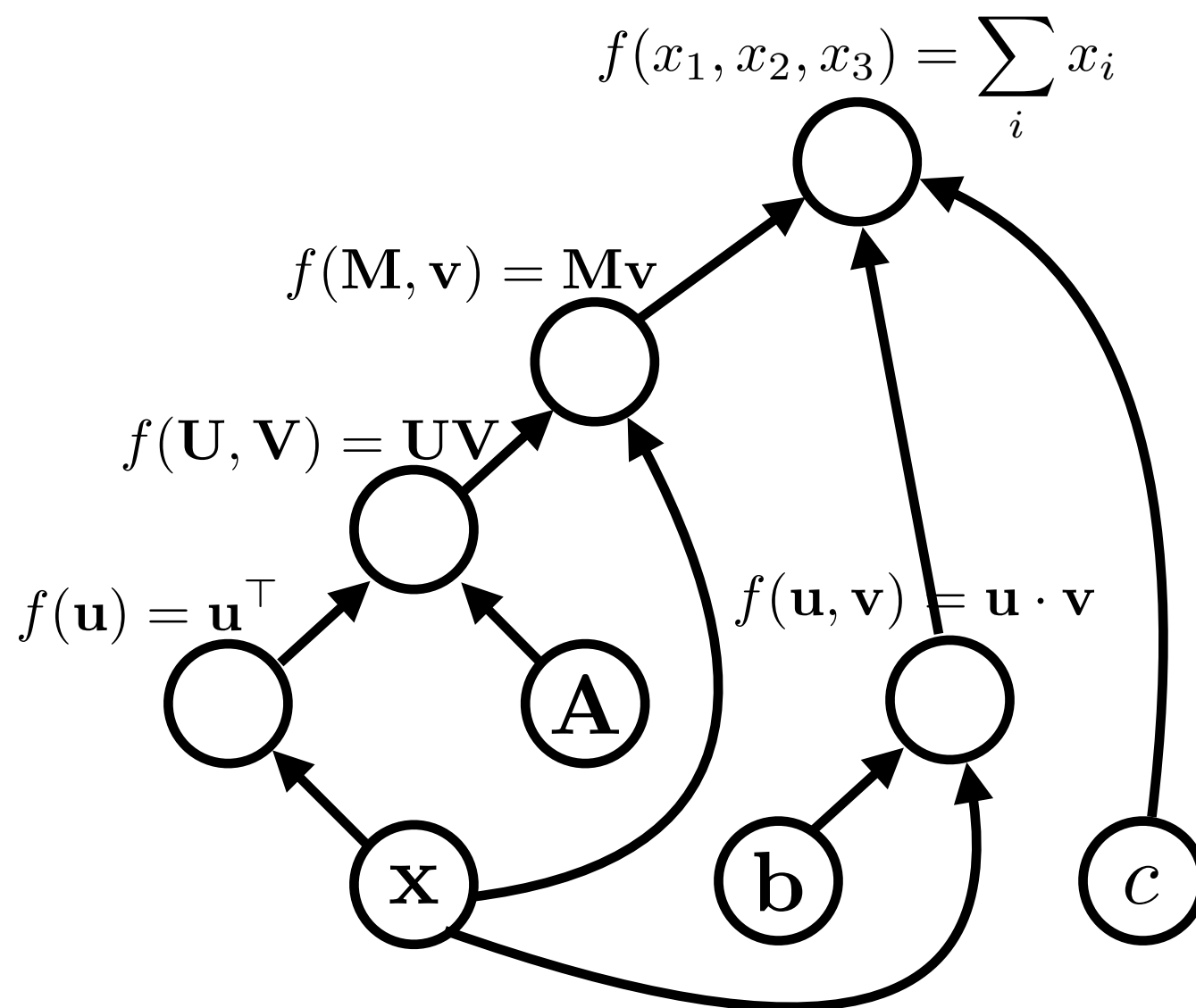
$$\frac{\partial f(\mathbf{x}, \mathbf{A})}{\partial \mathbf{x}} = (\mathbf{A}^\top + \mathbf{A})\mathbf{x}$$

$$\frac{\partial f(\mathbf{x}, \mathbf{A})}{\partial \mathbf{A}} = \mathbf{x}\mathbf{x}^\top$$

expression:

$$\mathbf{x}^\top \mathbf{A} \mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

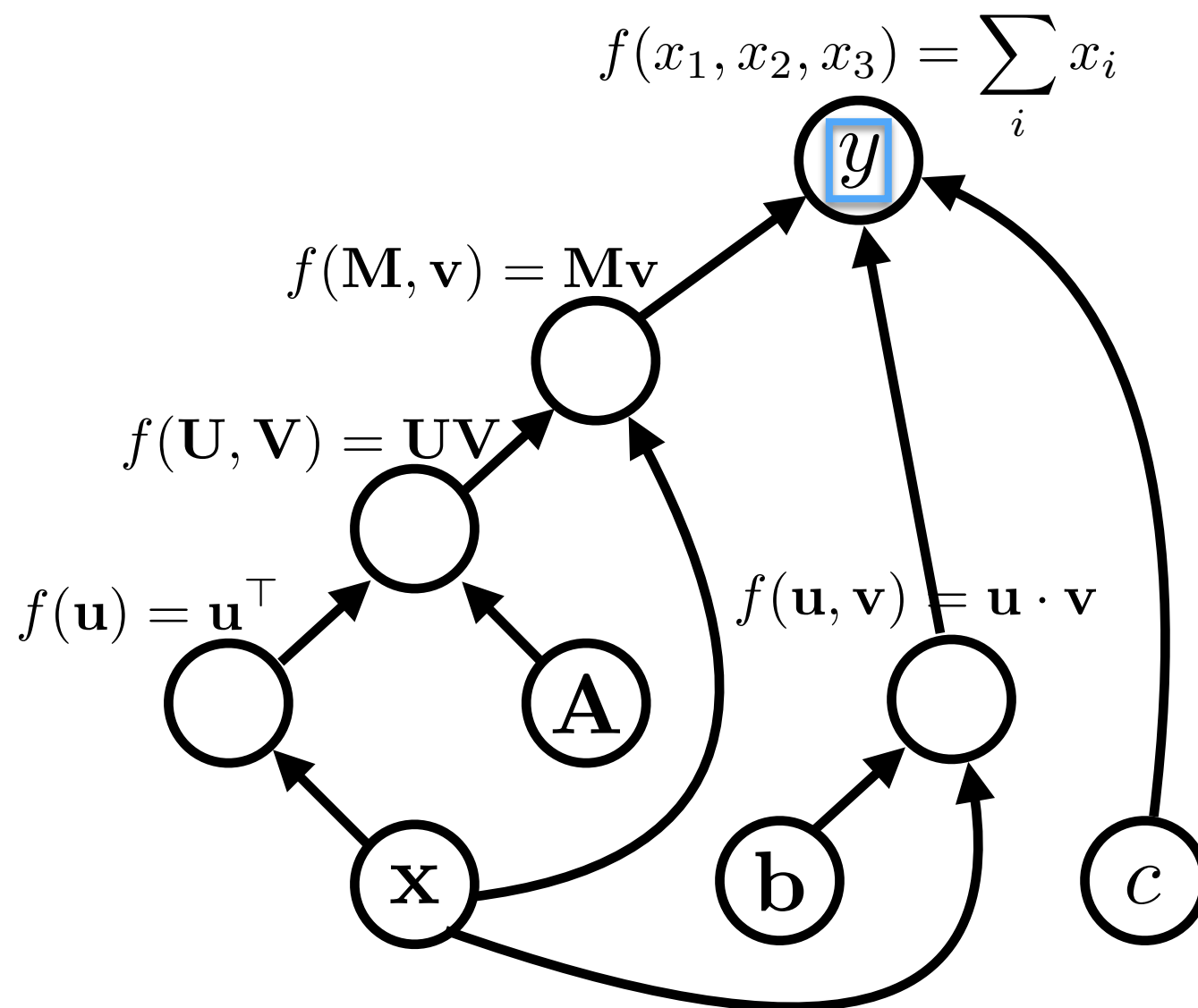
graph:



expression:

$$y = \mathbf{x}^\top \mathbf{A} \mathbf{x} + \mathbf{b} \cdot \mathbf{x} + c$$

graph:



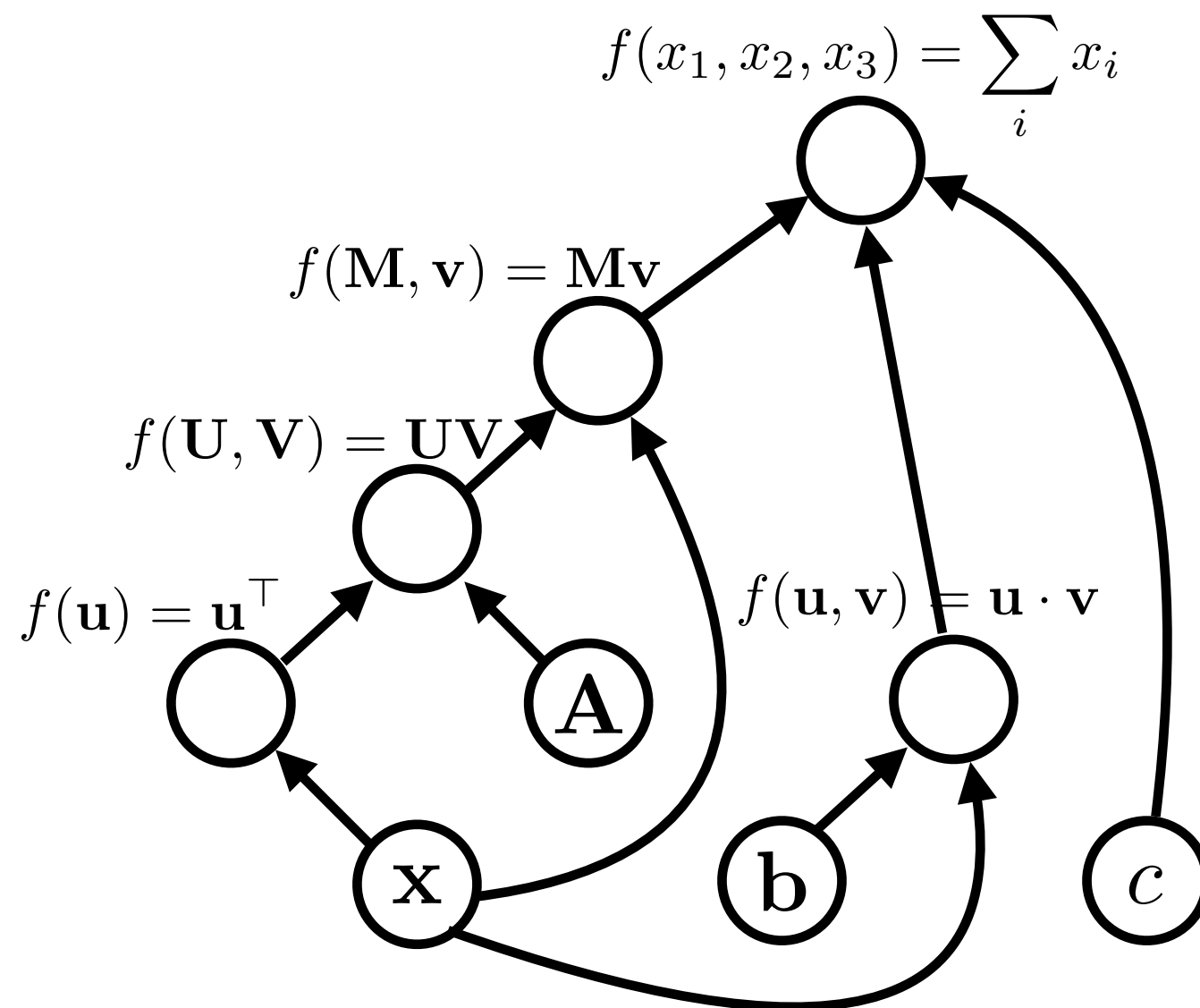
variable names are just labelings of nodes.

# Algorithms (1)

- **Graph construction**
- **Forward propagation**
  - In topological order, compute the **value** of the node given its inputs

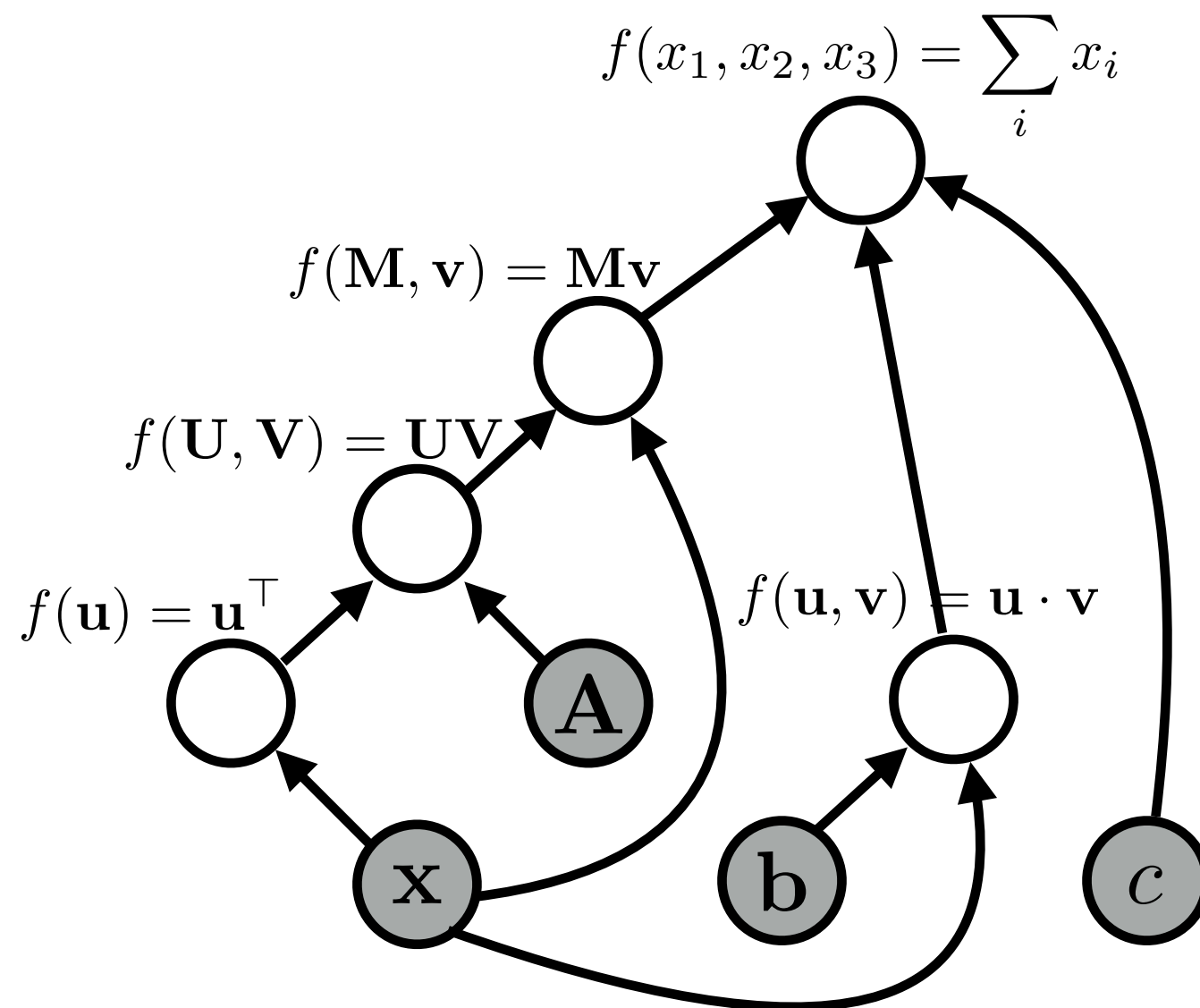
# Forward Propagation

graph:



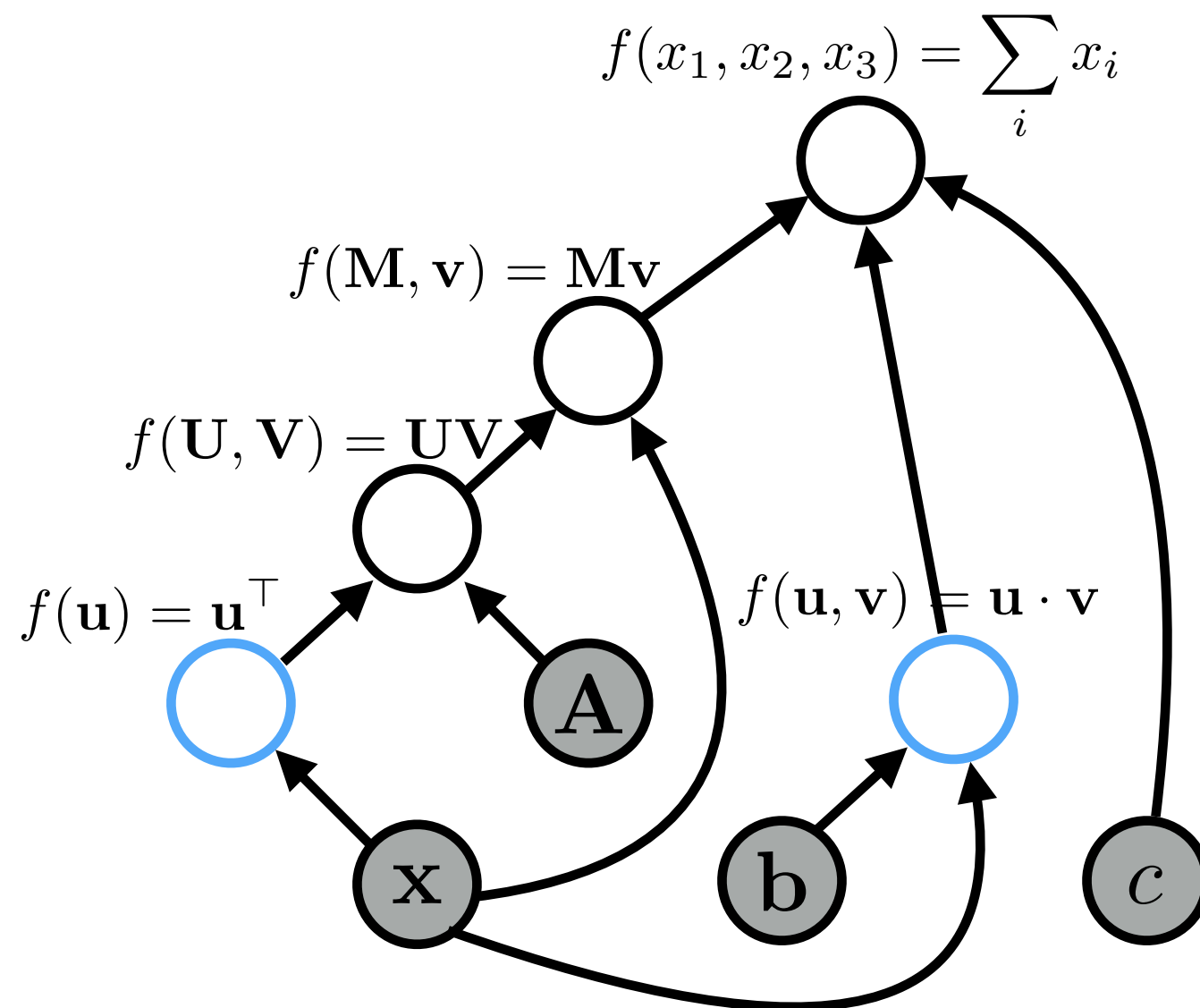
# Forward Propagation

graph:



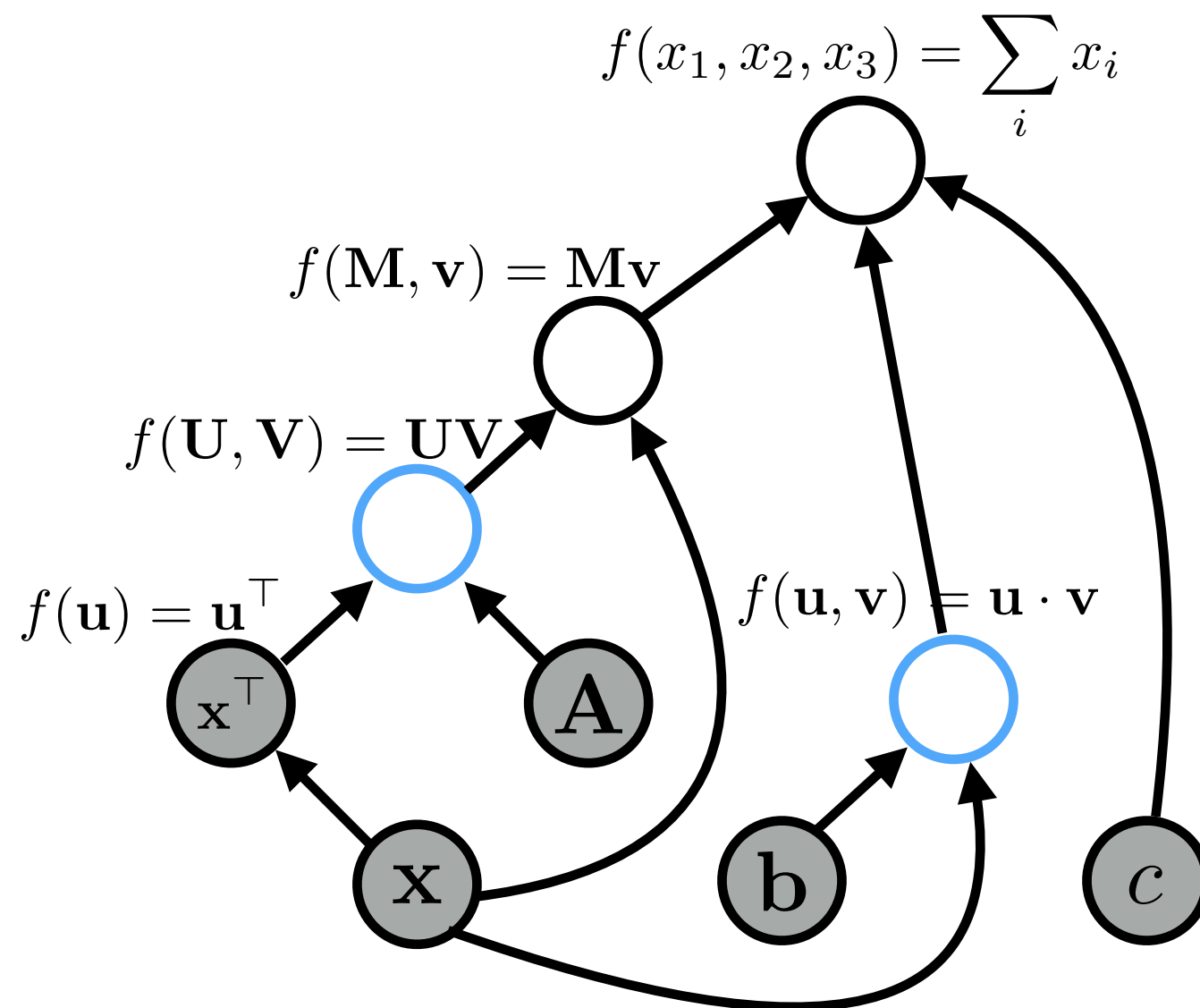
# Forward Propagation

graph:



# Forward Propagation

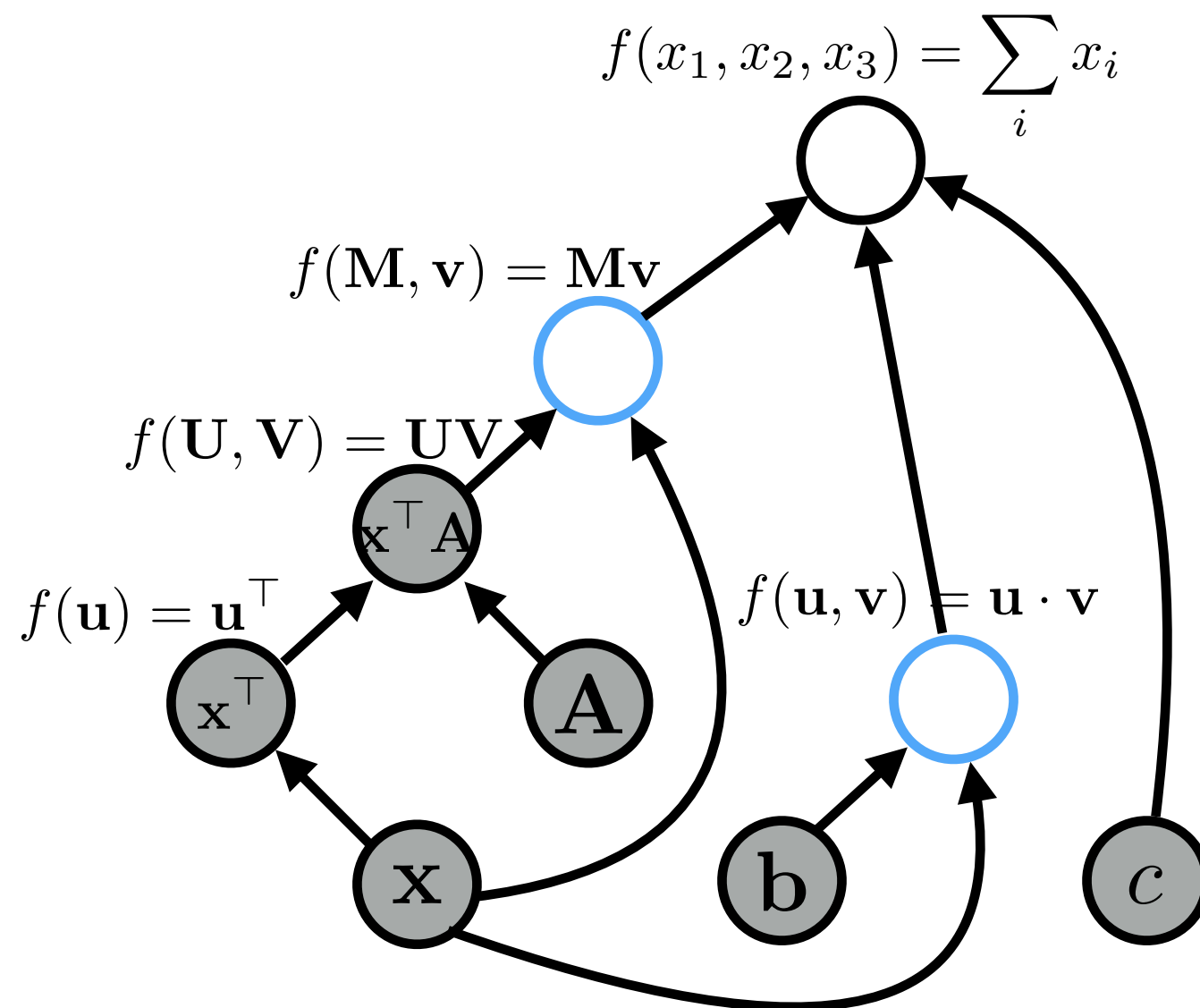
graph:





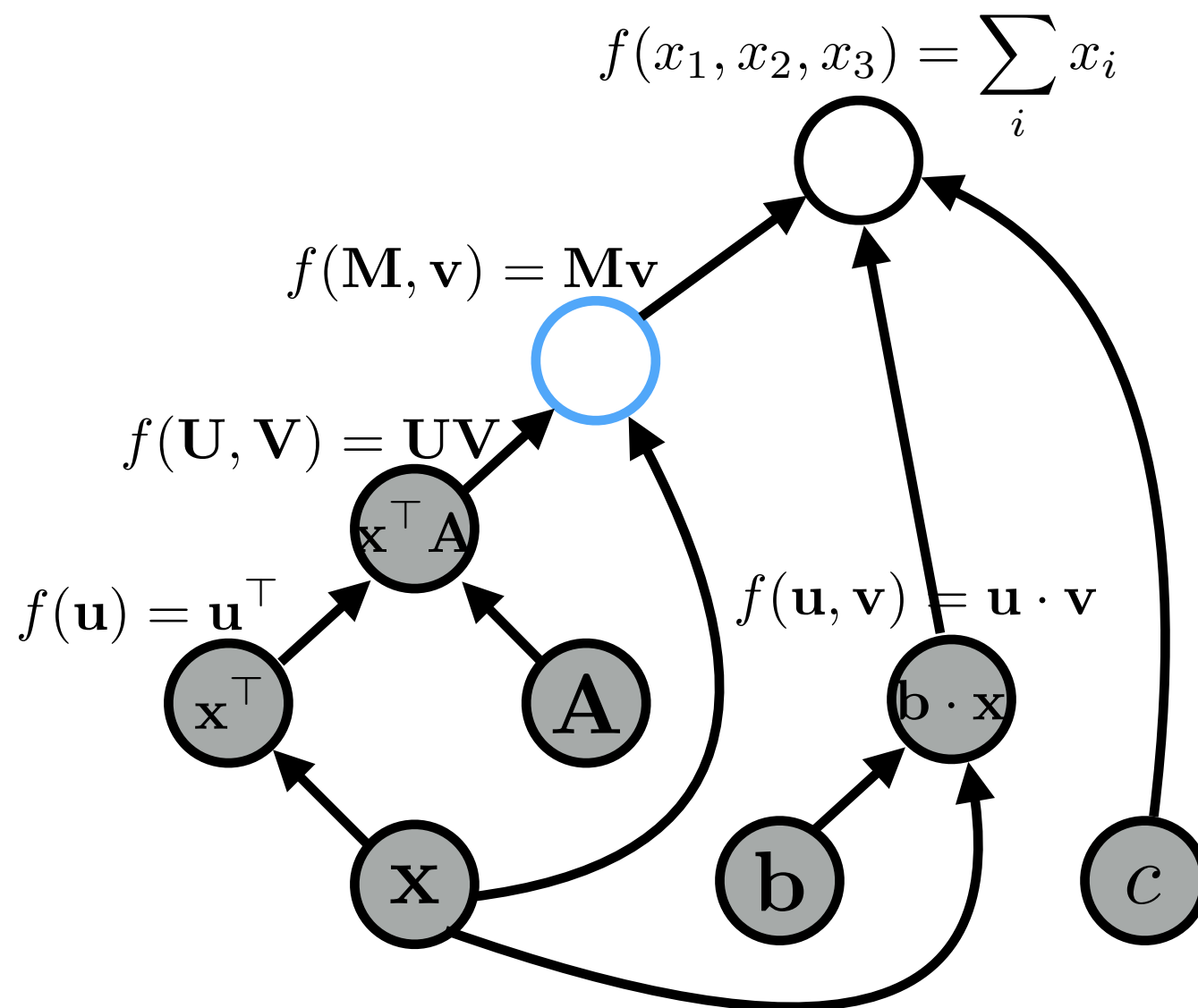
# Forward Propagation

graph:



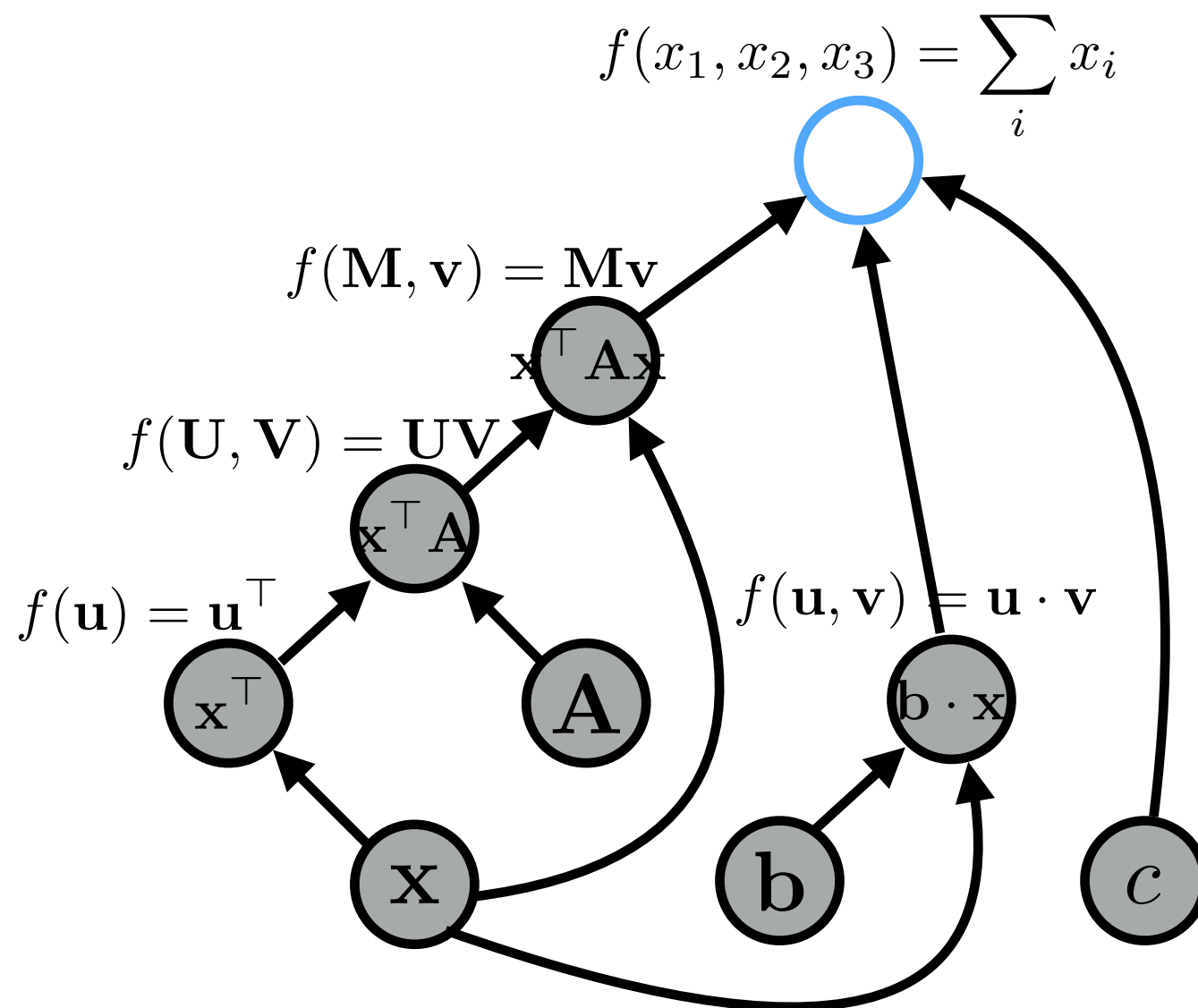
# Forward Propagation

graph:



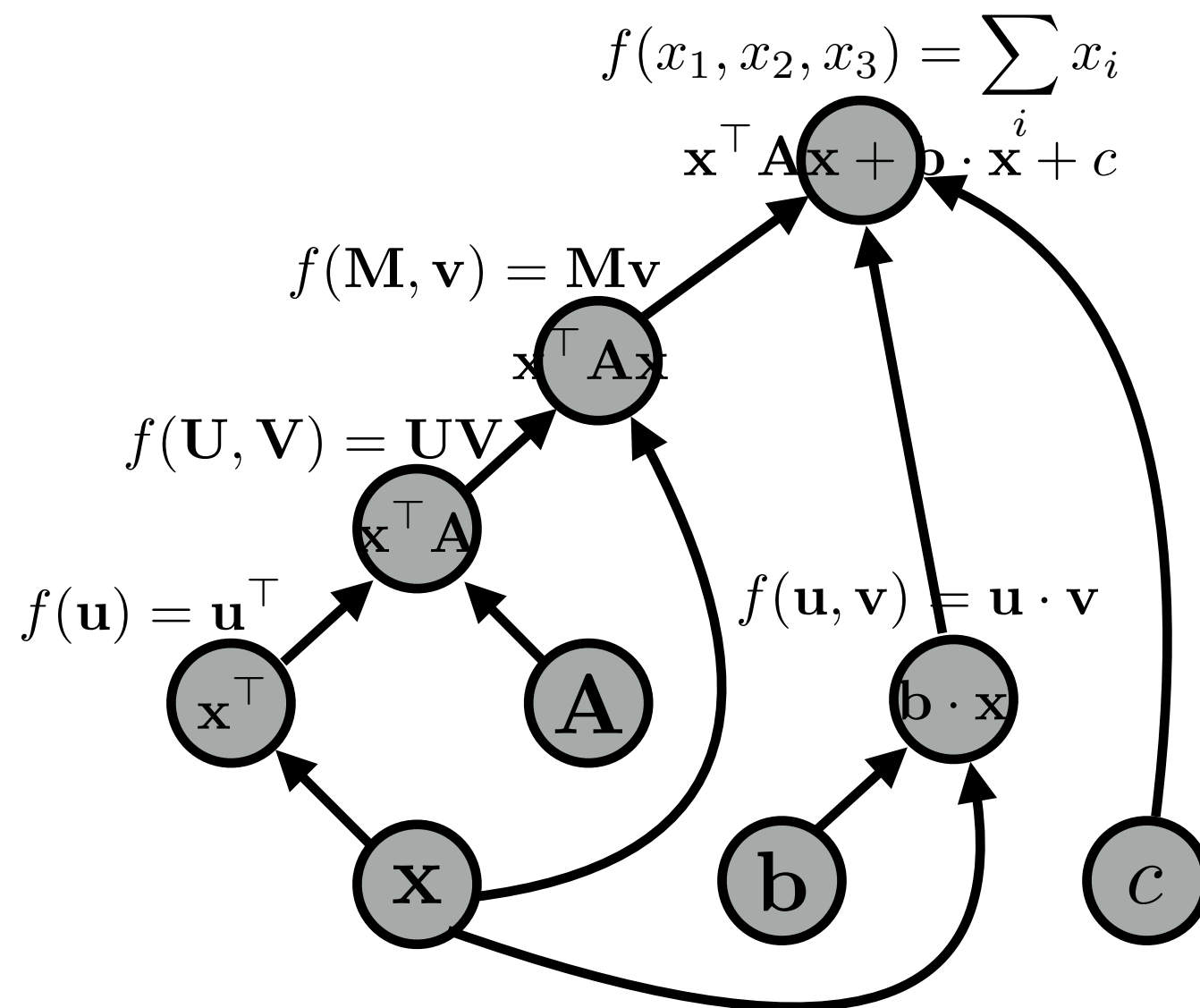
# Forward Propagation

graph:



# Forward Propagation

graph:

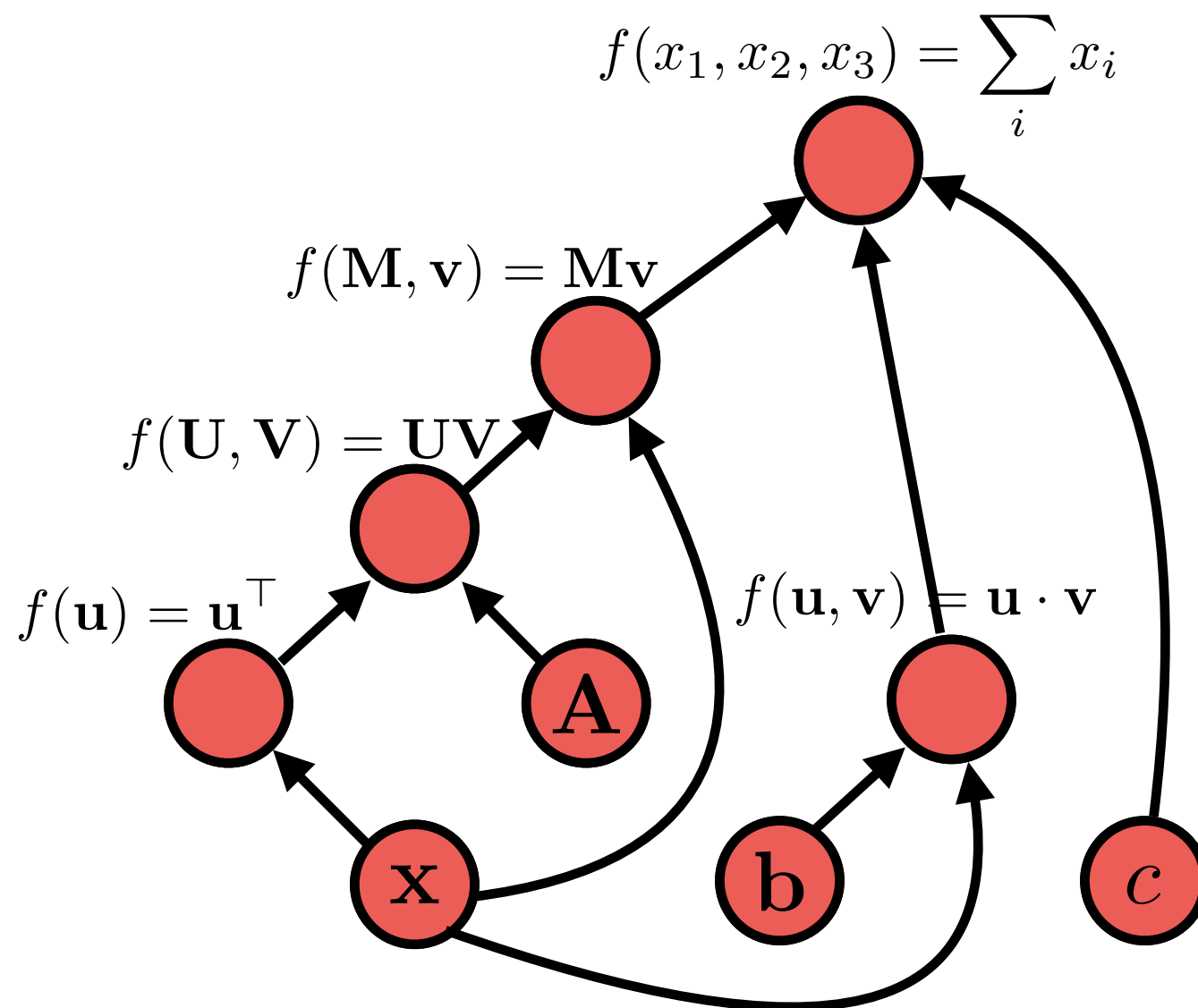


# Algorithms (2)

- **Back-propagation:**
  - Process examples in reverse topological order
  - Calculate the derivatives of the parameters with respect to the final value  
(This is usually a “loss function”, a value we want to minimize)
- **Parameter update:**
  - Move the parameters in the direction of this derivative  
$$W -= \alpha * dl/dW$$

# Back Propagation

graph:



# Concrete Implementation Examples

# Neural Network Frameworks

**PYTORCH**

Developed by FAIR/Meta

Most widely used in NLP

Favors dynamic execution

More flexibility

Most vibrant ecosystem

  
TensorFlow



Developed by Google

Used in some NLP projects

Favors definition+compilation

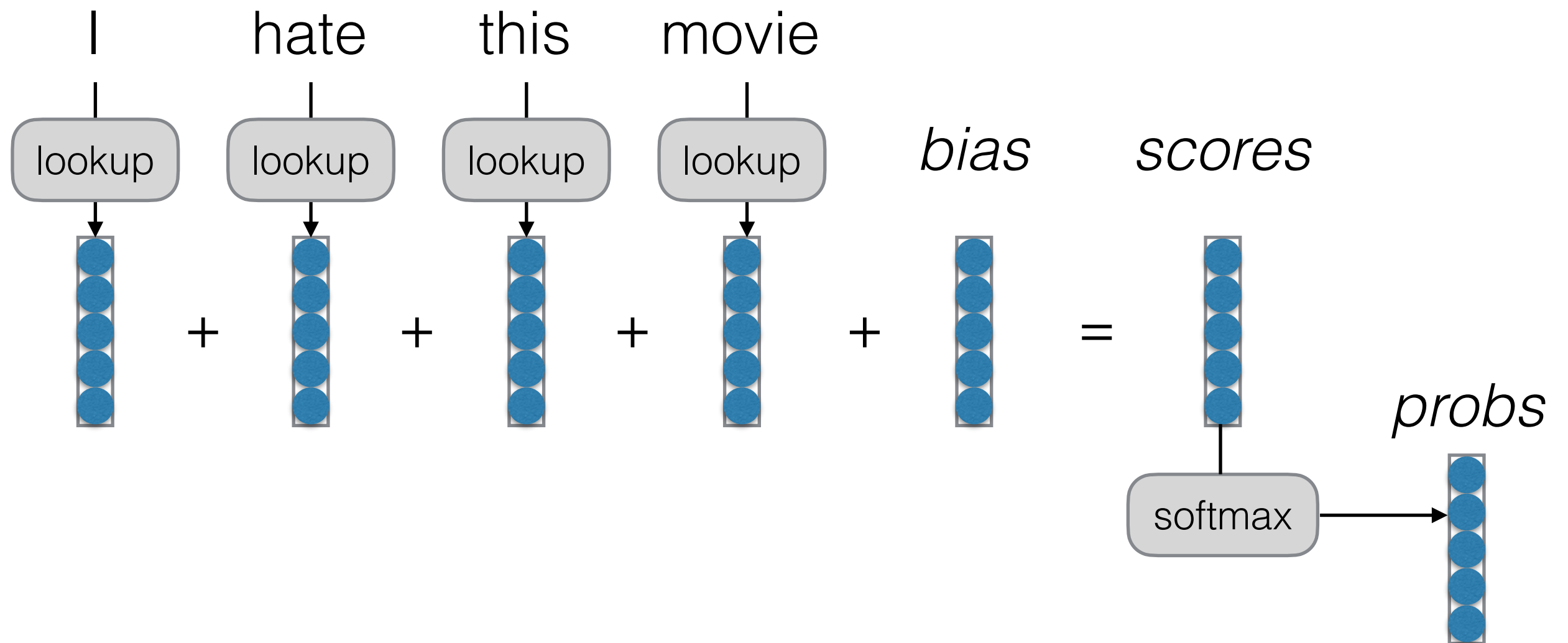
Conceptually simple parallelization



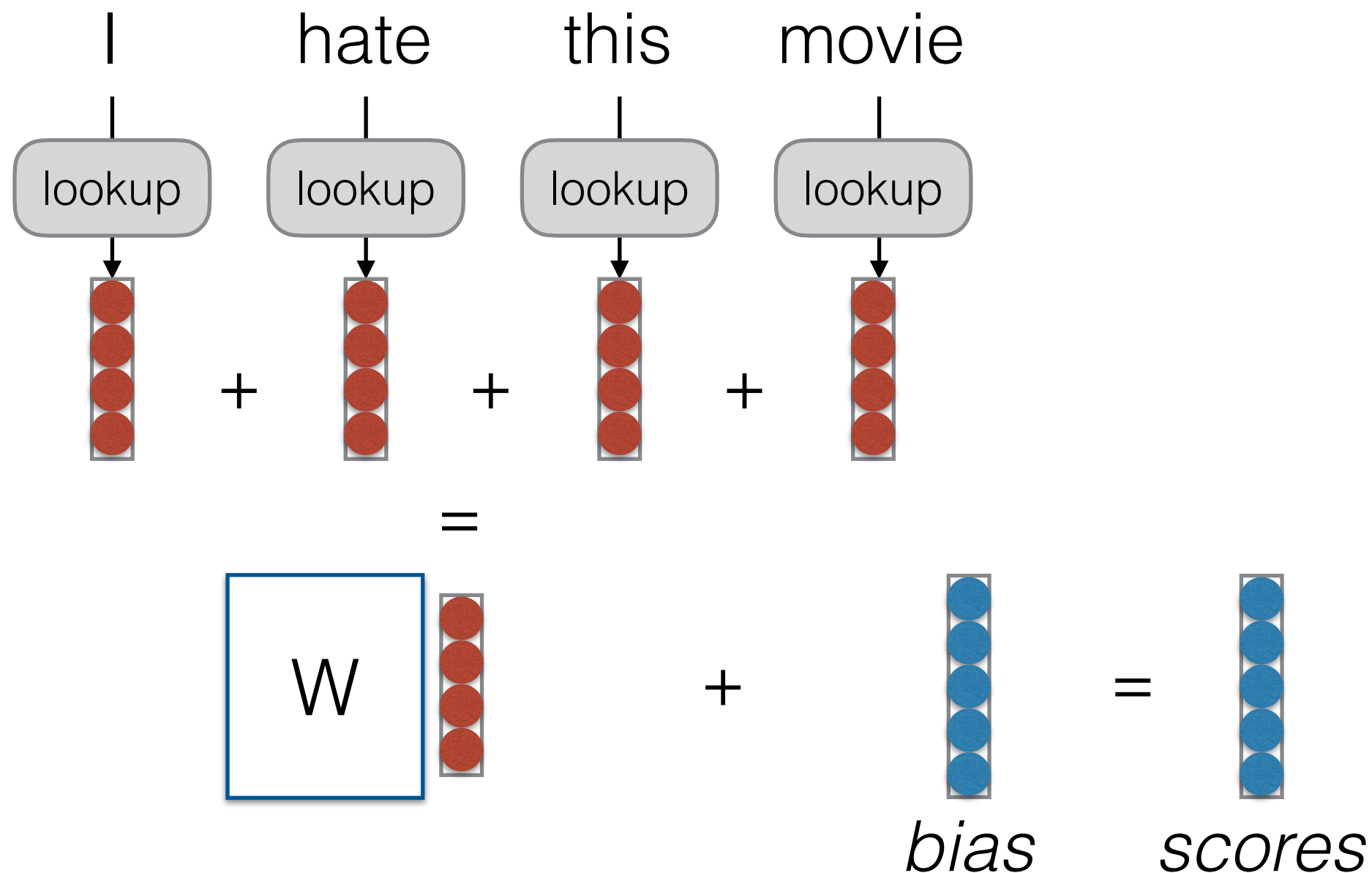
# Basic Process in Neural Network Frameworks

- 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**

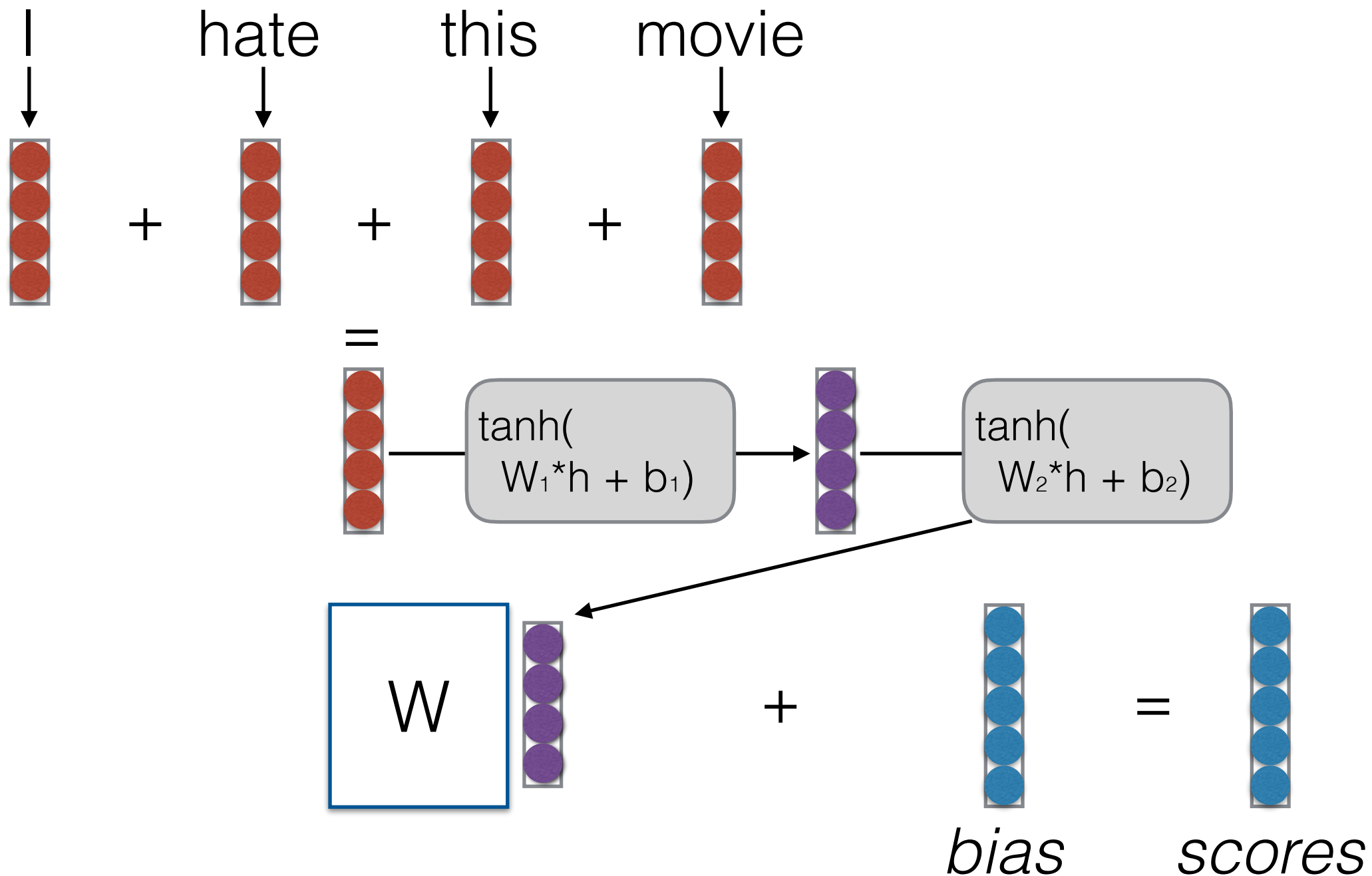
# Bag of Words (BOW)



# Continuous Bag of Words (CBOW)



# Deep CBOW



# A Few More Important Concepts

# A Better Optimizer: Adam

- Most standard optimization option in NLP and beyond
- Considers rolling average of gradient, and momentum

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad \text{Momentum}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t \odot g_t \quad \text{Rolling Average of Gradient}$$

- Correction of bias early in training

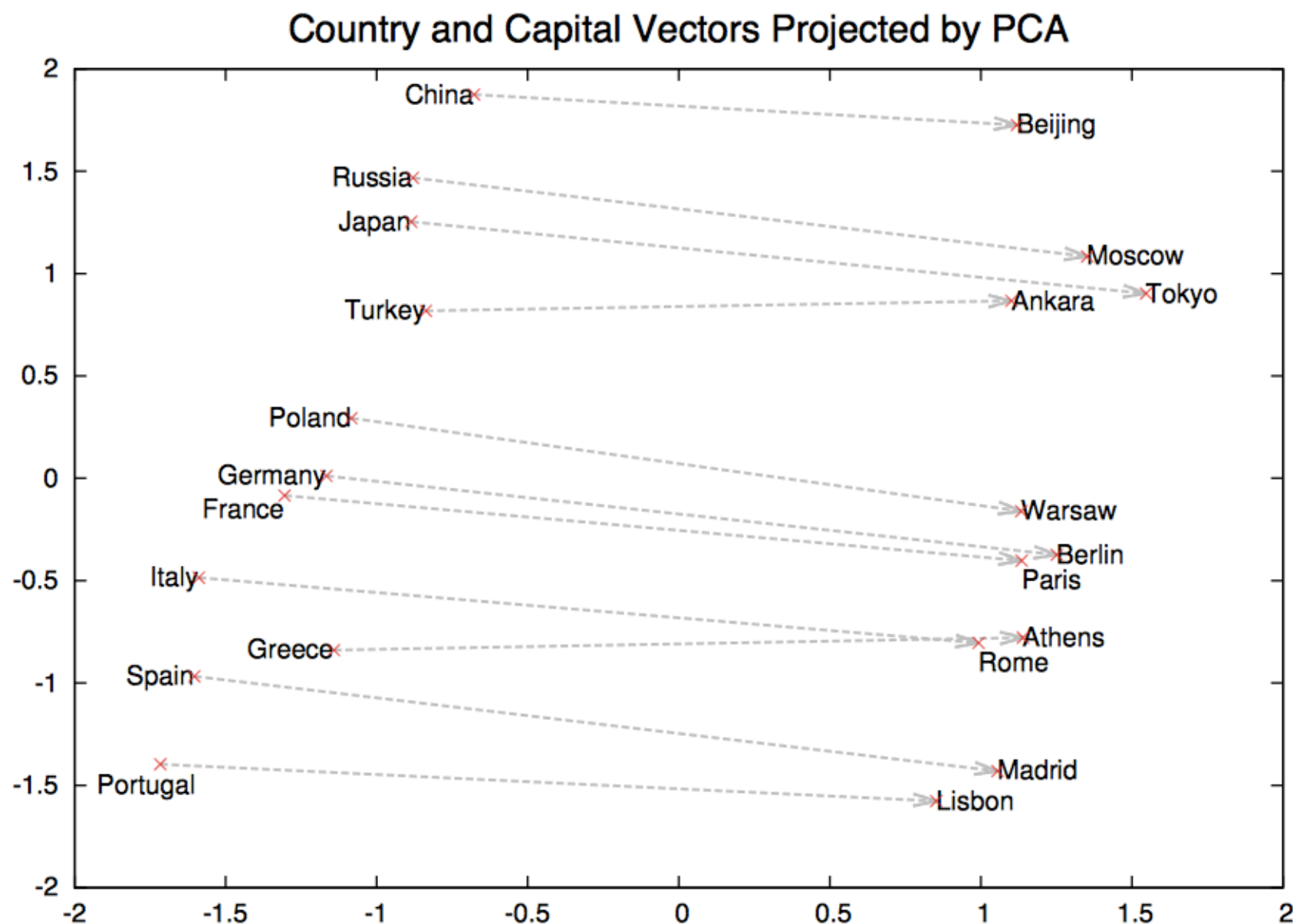
$$\hat{m}_t = \frac{m_t}{1 - (\beta_1)^t} \quad \hat{v}_t = \frac{v_t}{1 - (\beta_2)^t}$$

- Final update

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

# Visualization of Embeddings

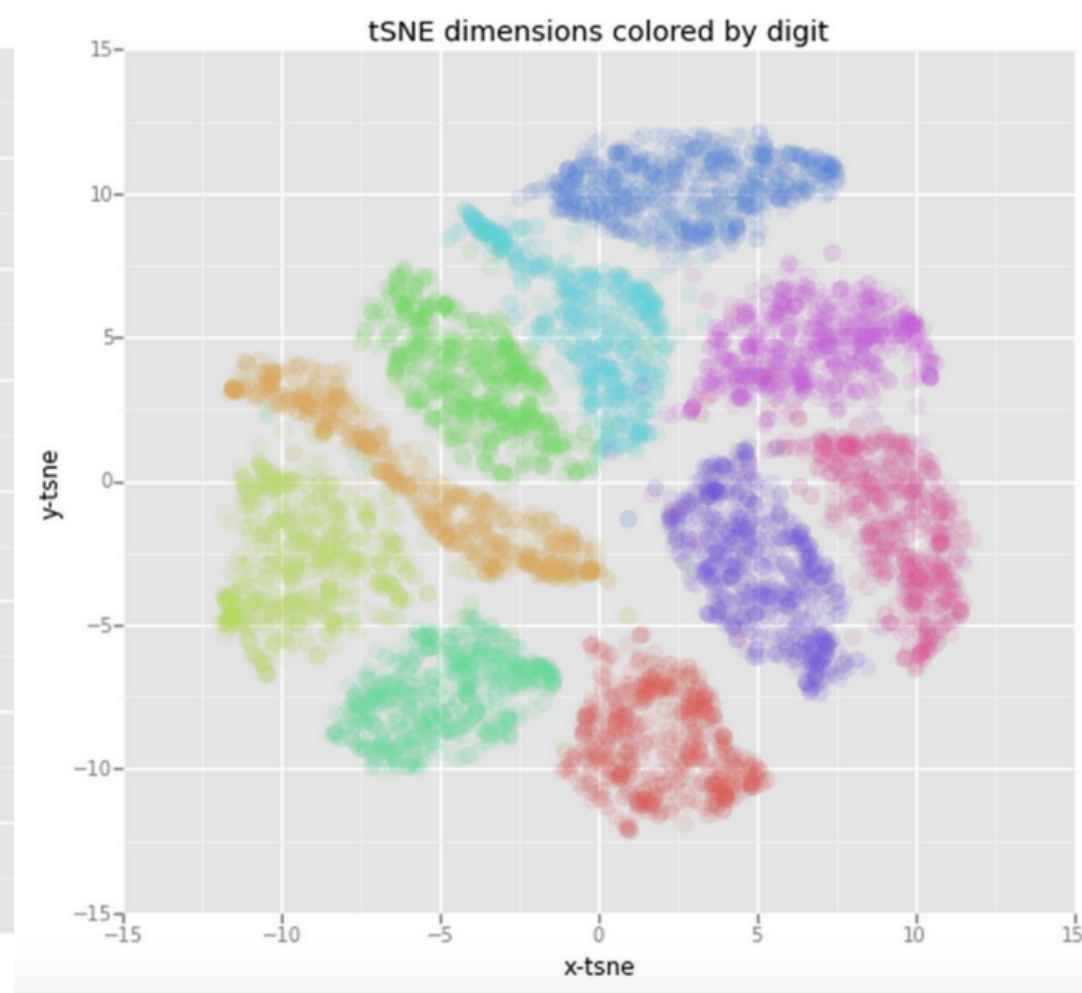
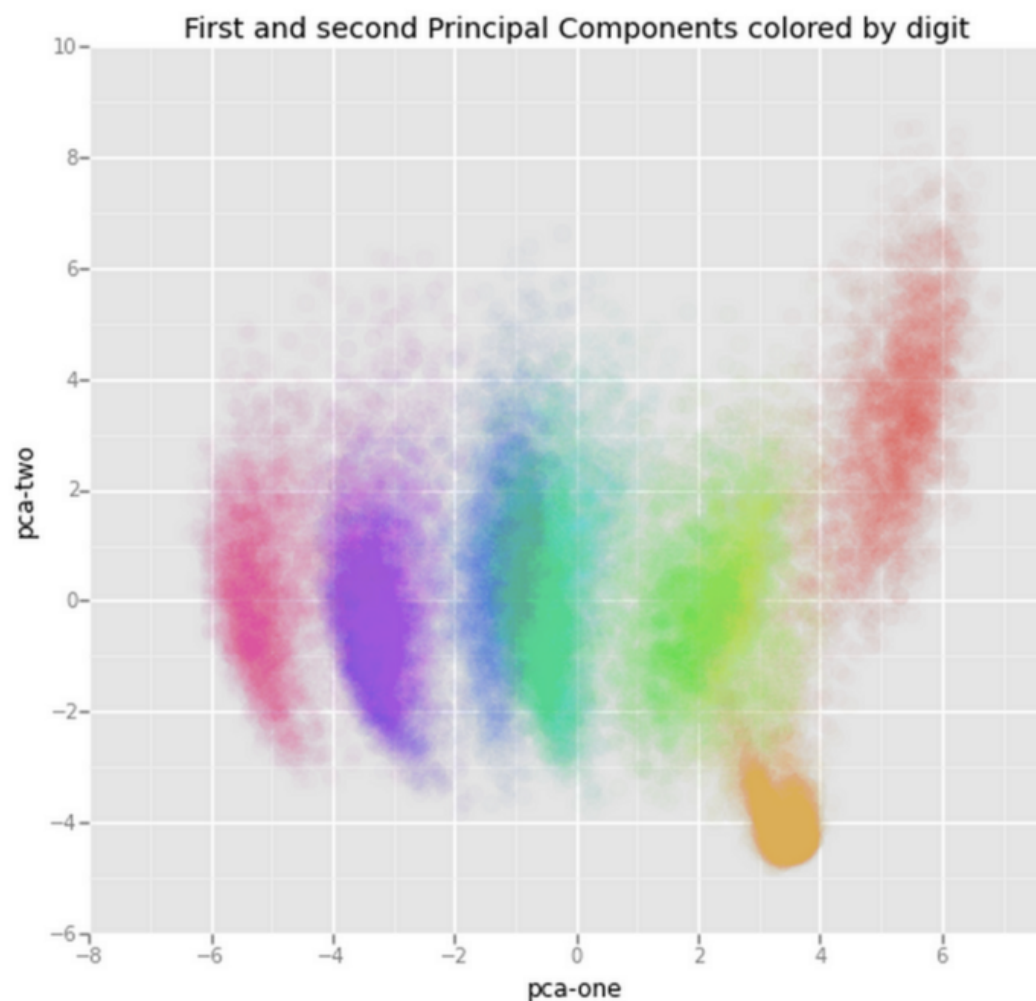
- Reduce high-dimensional embeddings into 2/3D for visualization (e.g. Mikolov et al. 2013)



# Non-linear Projection

- Non-linear projections group things that are close in high-dimensional space
- e.g. SNE/t-SNE (van der Maaten and Hinton 2008) group things that give each other a high probability according to a Gaussian

PCA



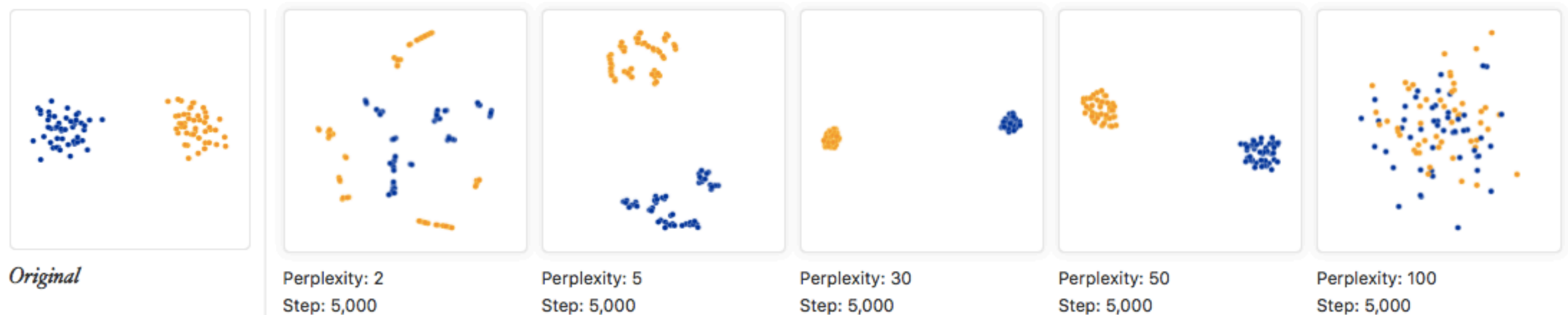
t-SNE

(Image credit: Derksen 2016)

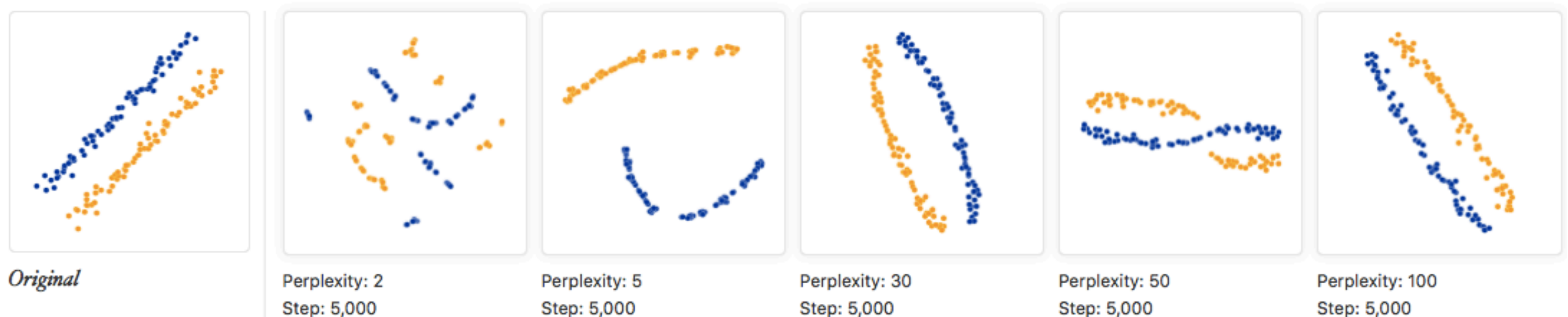


# t-SNE Visualization can be Misleading! (Wattenberg et al. 2016)

- Settings matter



- Linear correlations cannot be interpreted



# Any Questions?

(sequence models in next class)