CS11-711 Advanced NLP

Learned Representations

Sean Welleck

Carnegie Mellon University

https://cmu-l3.github.io/anlp-fall2025/

https://github.com/cmu-l3/anlp-fall2025-code

Slides adapted from Graham Neubig's Fall 2024 course

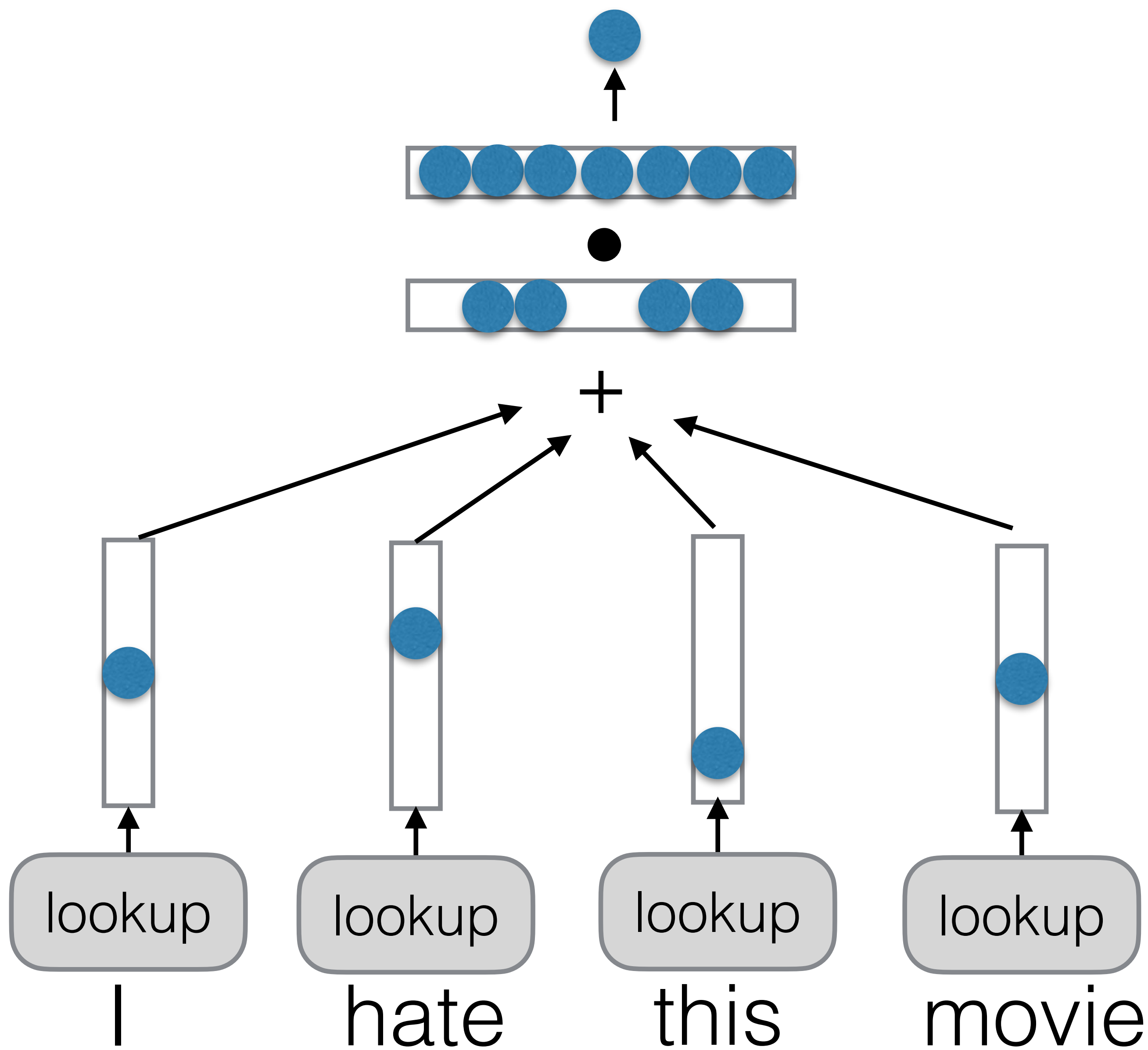
Recap

- Goal: learn a good scoring function $s_\theta(x, y)$
  - => good probabilistic models $p_\theta(y \mid x) \propto s_\theta(x, y)$
- Three key ingredients
  - **Modeling/Parameterization**: how s_θ (or p_θ) is implemented (e.g., the architecture)
  - **Learning**: setting the parameters θ using supervision
  - **Inference**: making a decision after learning
- We saw an example *classification* model based on:
  - Bag-of-words and word identities
  - Structured perceptron learning
  - A simple inference algorithm

Today's lecture

- We will still focus on classification: $g(x) \rightarrow \{1,2,\ldots,K\}$
- We will go over fundamentals that underlie any state-of-the-art NLP system:
  - Continuous representations of subwords
  - Parameterization based on neural networks
  - Learning by optimizing a loss function with back propagation and gradient descent

Recap: Bag of Words (BoW)



Features: sum of 1-hot vectors
Weights: learned

Bag of Words: Symptoms

- 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**

the companies are expanding

↓

the **compan _ies** are **expand _ing**

- Benefits:
  - **Share parameters** between subwords
  - Reduce parameter size, **save compute+memory**

Core problem: tokenization

- Map text into a sequence of discrete **tokens** from a **vocabulary**

$$x_1, \ldots, x_T \quad x_t \in \mathscr{V}$$

$\mathbf{x}$

the companies are expanding

the, companies, are, expanding

the, compan ,_ies, are, expand, _ing

t, h, e, c, o, m, p, a, n, ….

- We want a vocabulary $\mathscr{V}$ that is:
    - **Expressive**: represent any text (English, Japanese, code, …)
    - **Efficient**
        - **Not too large:** larger vocabulary means more parameters to learn/store
        - **Not too small:** smaller vocabulary means longer inputs

Core problem: tokenization

- Demo: https://tiktokenizer.vercel.app/

Idea 1: UTF-8

- Tokenize text as UTF-8 bytes

元気ですか。Hello!

Unicode string

```
utf = "元気ですか。Hello!".encode("utf-8")
print([x for x in utf])
```

✓ 0.0s

UTF-8

(Vocabulary = 256 byte choices)

```
[229, 133, 131, 230, 176, 151, 227, 129, 167, 227, 129, 153, 227, 129, 139, 227, 128, 130, 72, 101, 108, 108, 111, 33]
```

- **Expressive:** any Unicode string (Japanese, English, Latex, …)
- **Vocabulary is too small**: sequences are very long (inefficient)

Idea 2: Byte Pair Encoding

- **Key idea**: merge the most common token pairs into new tokens
  - Start with a base vocabulary (e.g., UTF-8) and a training set
  - Repeat:
    - Find the token pair that occurs most often
    - Introduce a new token and replace the token pair

```
training_text = """Hello, world!
Here is some example text to test
the BPE algorithm. It is not very
interesting, but it will do the job.
"""
```

```
pair: ('e', ' ') freq: 5
merging ('e', ' ') into a new token 256

pair: ('t', ' ') freq: 5
merging ('t', ' ') into a new token 257

pair: ('e', 'r') freq: 3
merging ('e', 'r') into a new token 258

pair: ('t', 'h') freq: 3
merging ('t', 'h') into a new token 259

pair: ('l', 'l') freq: 2
merging ('l', 'l') into a new token 260
```

Practical tools: tiktoken

- Load pre-existing OpenAI vocabularies (e.g., GPT-2, GPT-4)

- Tokenize and decode text

⌛ **tiktoken**

tiktoken is a fast BPE tokeniser for use with OpenAI's models.

```python
# !pip install tiktoken
import tiktoken

enc = tiktoken.get_encoding("gpt2")
print(enc.encode("Hello, こんにちは"))

enc = tiktoken.get_encoding("cl100k_base")
print(enc.encode("Hello, こんにちは"))
```

✓ 0.0s

```
[15496, 11, 23294, 241, 22174, 28618, 2515, 94, 31676]
[9906, 11, 220, 90115]
```

Practical tools: SentencePiece

google / sentencepiece

- Also supports *training* a tokenizer
- Uses *Unicode* as the base vocabulary
    - *byte_fallback=True*: tokenize as UTF-8 bytes when a Unicode character is out-of-vocabulary

```
ids = sp.encode("hello, こんにちは マラソ マラソン marathon")
print(ids)

print([sp.id_to_piece(idx) for idx in ids])
```

```
[1298, 295, 1339, 1353, 1333, 1534, 1457, 1366, 1793, 1373, 1333, 329, 1407, 584, 964]
['_he', 'll', 'o', ',', '_', 'こ', 'ん', 'に', 'ち', 'は', '_', 'マラ', 'ソ', '_マラソン', '_marathon']
```

Subword Considerations

- **Vocabulary depends on the BPE training data**:
    - Under-represented languages: merged less, hence longer sequences
    - *Work-around*: upsample under-represented languages
- **Inconsistent numbers**: 123 -> “123” vs. 927 -> “92” “7”
    - *Work-around*: Hand-defined rules, e.g. never group digits together

Recap

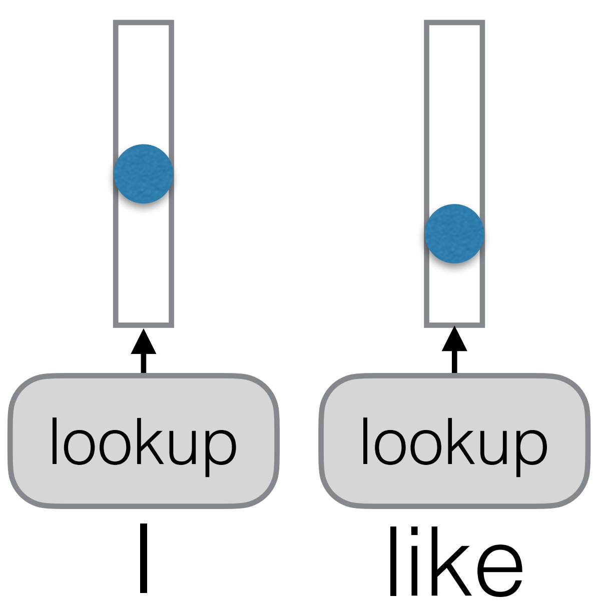
- Tokenization and subword models
    - Represent sequences as tokens determined based on frequency
- **Next**: Token embeddings

Continuous Word Embeddings

Basic Idea

- Previously: **one-hot** vectors (*sparse*)
- Continuous embeddings: *dense* vectors in $\mathbb{R}^{d_{emb}}$

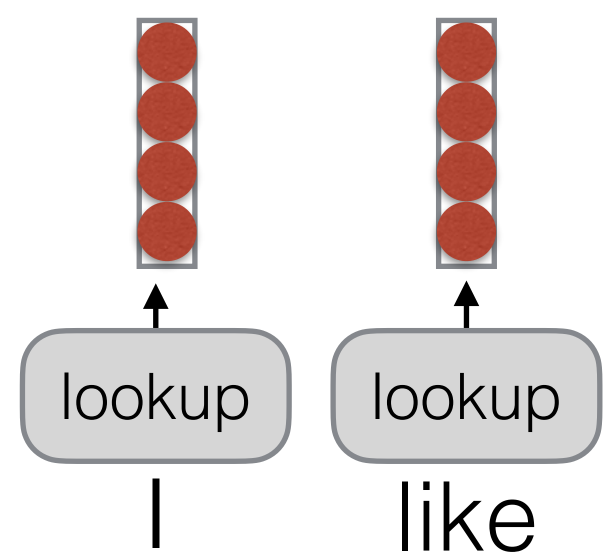
<u>One-hot</u>



$$x_t : [0, \ldots 1, \ldots, 0] \in \{0,1\}^V$$

V: vocabulary size

<u>Dense</u>

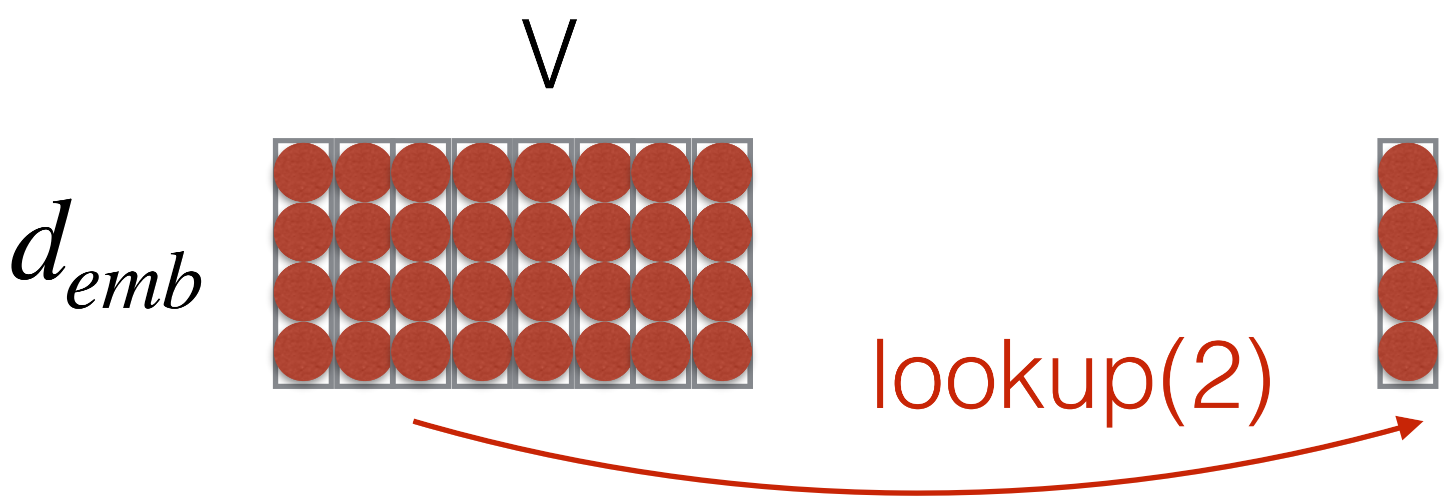


$$x_t : [0.2, -1.3, \ldots, 0.6] \in \mathbb{R}^{d_{emb}}$$

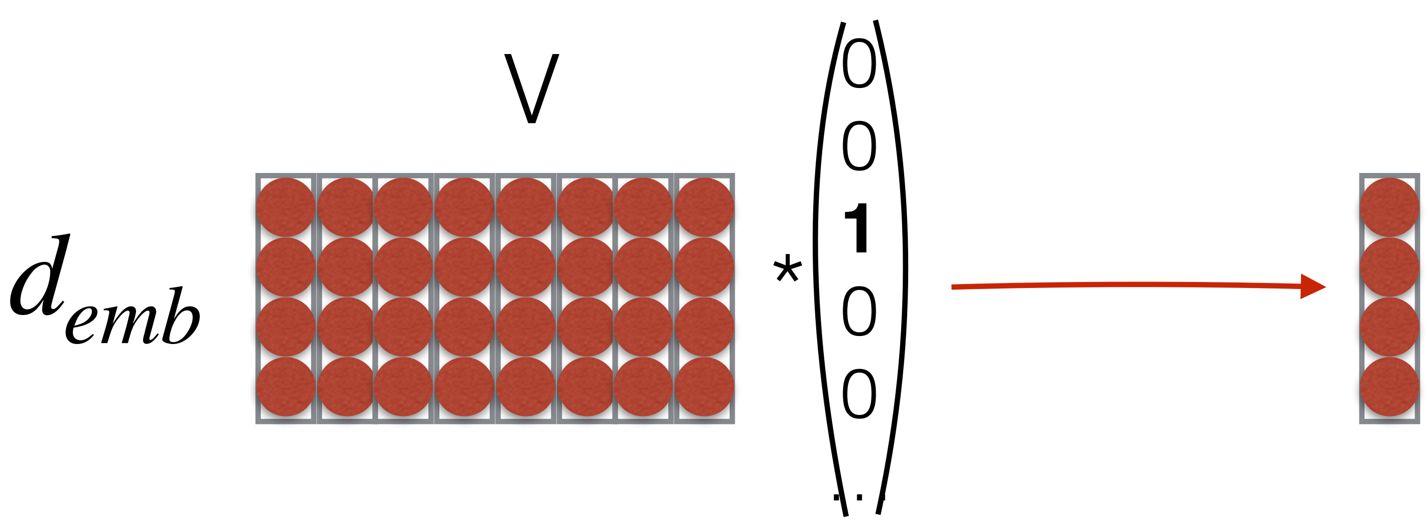
d_{emb}: "embedding dimension"

Embedding Layer

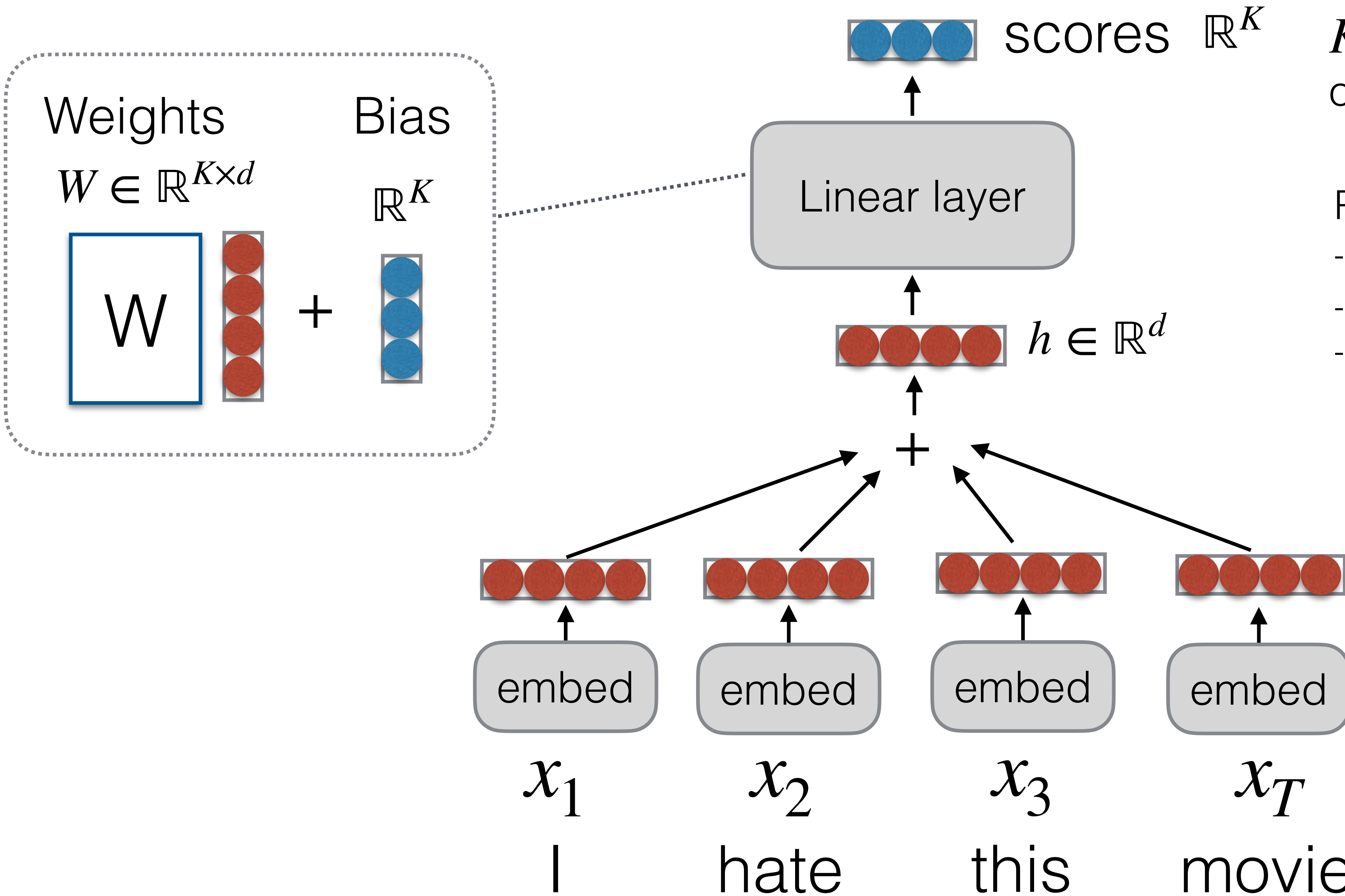
- Embedding layer: matrix with a row/column for each vocabulary token. “Lookup”: select a row/column.



- Equivalent to multiplying by a one-hot vector



Continuous Bag of Words (CBoW)



K: number of output classes

Parameters θ

- Embedding layer
- Weights W
- Bias

In Code

```python
class Embedding(nn.Module):
    def __init__(self, vocab_size, emb_size):
        super(Embedding, self).__init__()
        self.weight = nn.Parameter(torch.randn(vocab_size, emb_size))
        self.vocab_size = vocab_size

    def forward(self, x):
        xs = torch.nn.functional.one_hot(x, num_classes=self.vocab_size).float()
        return torch.matmul(xs, self.weight)
```

In practice, implemented in libraries (e.g., nn.Embedding)

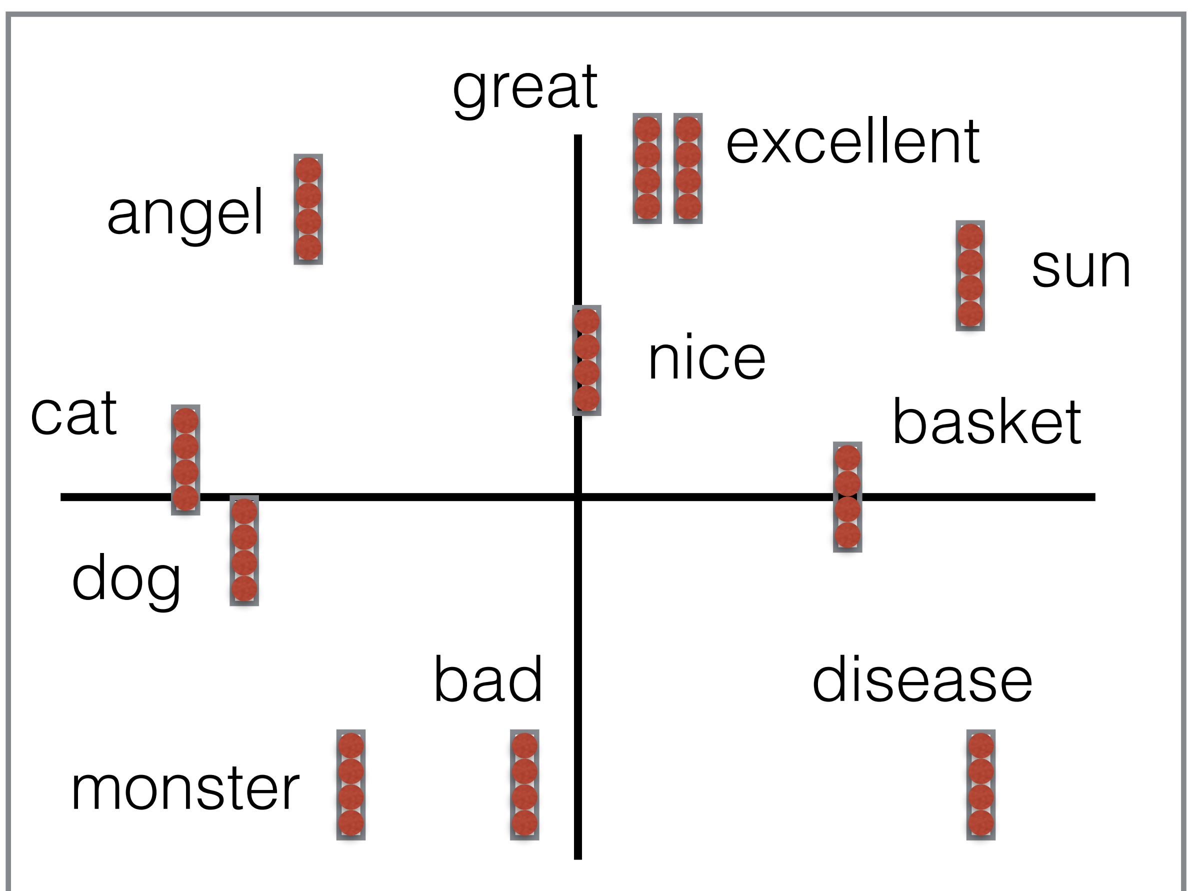
In Code

```python
class CBoW(torch.nn.Module):
    def __init__(self, vocab_size, num_labels, emb_size):
        super(CBoW, self).__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size)
        self.output_layer = nn.Linear(emb_size, num_labels)

    def forward(self, tokens):
        emb = self.embedding(tokens)    # [len(tokens) x emb_size]
        emb_sum = torch.sum(emb, dim=0) # [emb_size]
        h = emb_sum.view(1, -1)         # [1 x emb_size]
        out = self.output_layer(h)      # [1 x num_labels]
        return out
```

What do Our Vectors Represent?

- No guarantees, but we hope that:
  - Words that are **similar** are **close** in vector space
  - Each vector element is a **feature**



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

Recap

- Tokenization and subword models
    - Represent sequences as tokens determined based on frequency
- Token embeddings
    - Represent tokens as learned continuous vectors
- **Next**: Neural networks

Neural Network Features

Motivation: combination features

I don't love this movie

very good
good
neutral
bad
very bad

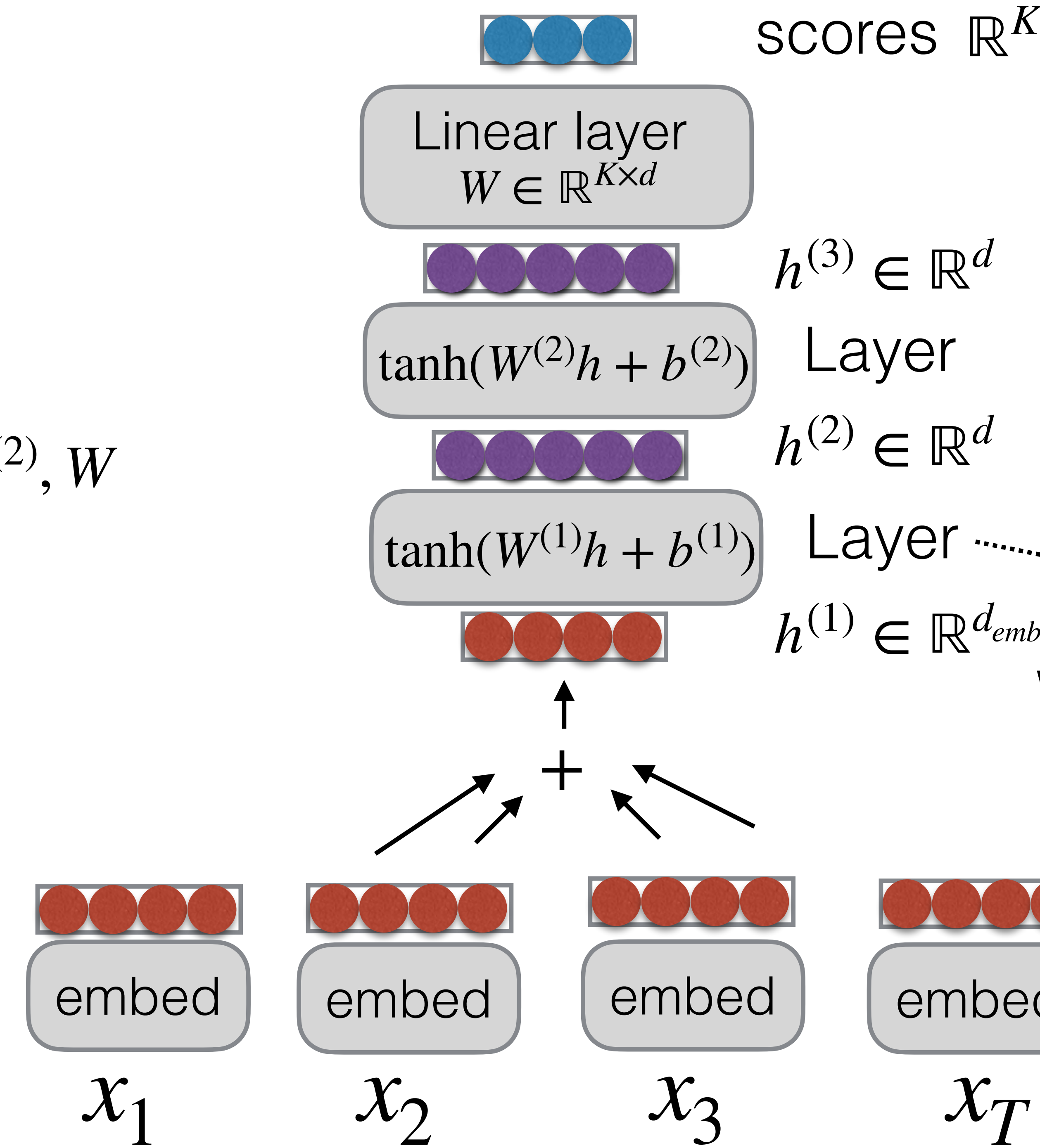
There's nothing I don't love about this movie

very good
good
neutral
bad
very bad

Deep CBoW

K: number of output classes

Parameters θ
- Embedding layer
- Weights $W^{(1)}, W^{(2)}, W$
- Biases

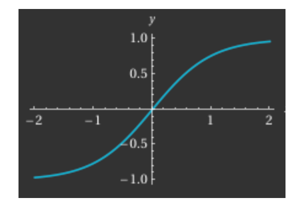
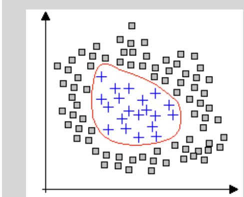


Why tanh?
Why do these layers help?

Nonlinearities

tanh(W*h + b)

- *Activation functions* such as tanh introduce *nonlinearity*
  - Non-linearities allow the neural network to model more complex patterns

W1  W2 ≡ 

Linear | Nonlinear | Linear | Nonlinear

- Without activation functions, stacking matrices collapses to a linear transformation

W1 W2 ≡ W’

Linear | Linear | Linear

Other activation functions: sigmoid, ReLU, GELU, see PyTorch list

Deep CBoW In Code

```
class DeepCBoW(torch.nn.Module):
    def __init__(self, vocab_size, num_labels, emb_size, hid_size):
        super(DeepCBoW, self).__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size)
        self.linear1 = nn.Linear(emb_size, hid_size)     # New addition
        self.output_layer = nn.Linear(hid_size, num_labels)

    def forward(self, tokens):
        emb = self.embedding(tokens)
        emb_sum = torch.sum(emb, dim=0)
        h = emb_sum.view(1, -1)
        h = torch.tanh(self.linear1(h))  # New addition
        out = self.output_layer(h)
        return out
```

(One hidden-layer version)

What do Our Vectors Represent?

- We can learn feature combinations
  - E.g., a node in the second layer might be “feature 1 AND feature 5 are active”
  - E.g. capture things such as “not” AND “hate”
- We can learn nonlinear transformations of the previous layer’s features

Recap

- Tokenization and subword models
    - Represent sequences as tokens determined based on frequency
- Token embeddings
    - Represent tokens as learned continuous vectors
- Neural networks
    - Learn complex, non-linear feature functions
- **Next**: Training neural network models

Training neural network models

Training neural network models

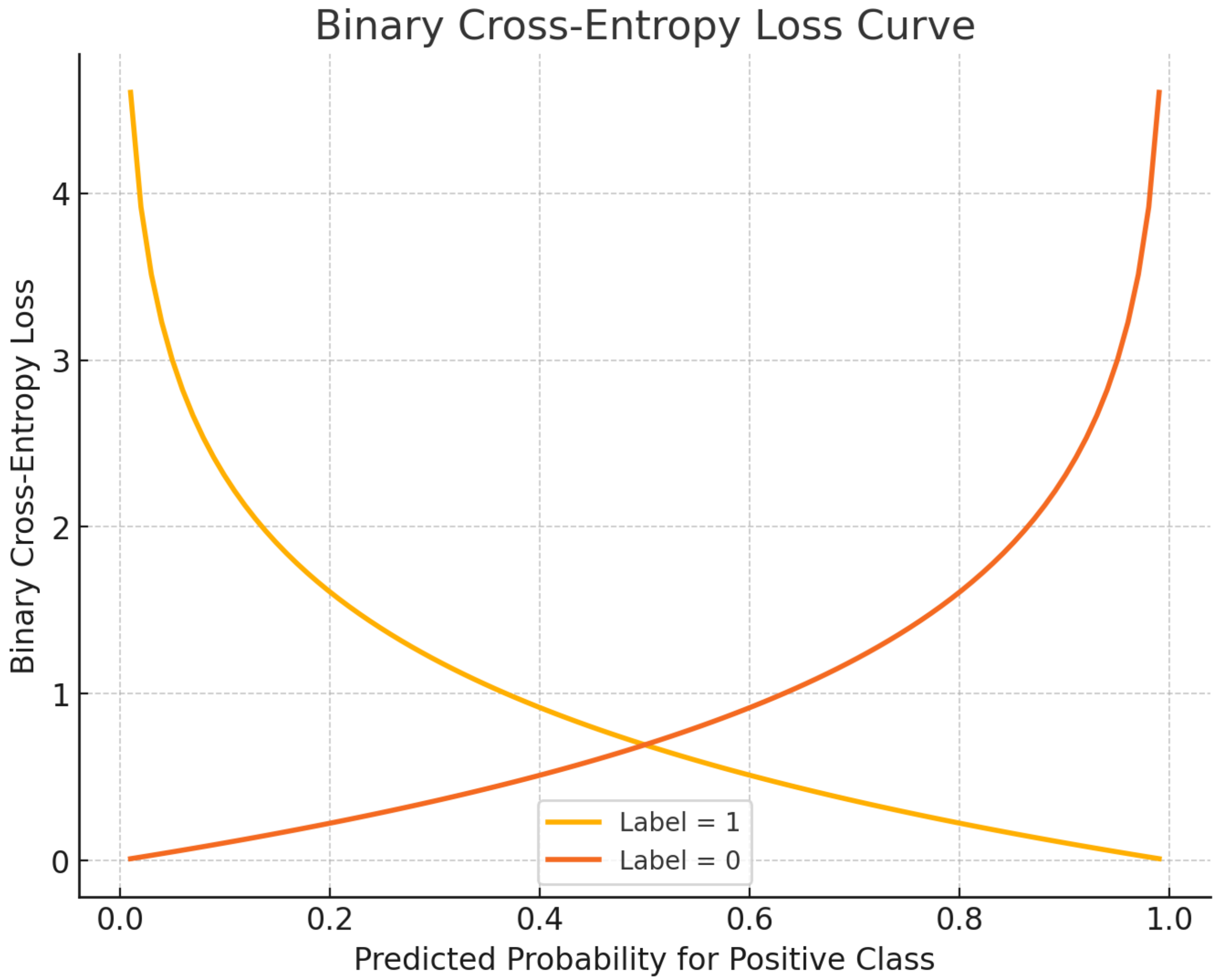
- We use *gradient descent*
  - Write down a *loss function*
  - *Calculate gradients* of the loss function with respect to the parameters
  - Move the parameters in the direction that *reduces the loss function*

Example Loss: Binary Cross entropy

- Example task: classify tweets as positive (1) or negative (0)
  - Model outputs a probability $p \in [0,1]$ for the positive class
    - Use a *sigmoid*:

$$\text{Sigmoid}(s) = \sigma(s) = \frac{1}{1 + \exp(-s)}$$

  - Ground truth label $y \in \{0,1\}$



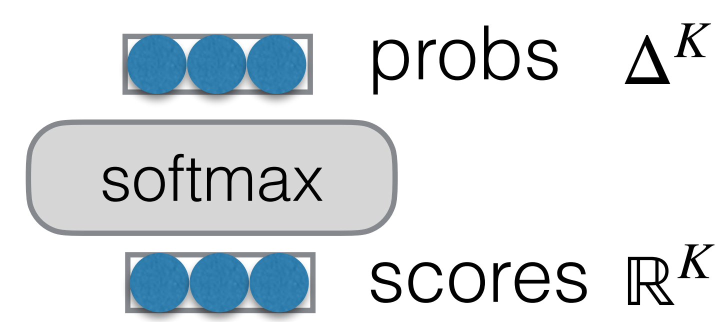
$$L_{\text{BCE}} = -y\log(p) - (1-y)\log(1-p)$$

Cross entropy loss (multi-class)

- Example task: classify tweets as positive (2), neutral (1), or negative (0)

$$L_{CE} = -\sum_{i=1}^{3} y_i \log(p_i)$$

- Given a training example (x, y)
- Model outputs a probability vector
  - E.g. $p = [0.2, 0.5, 0.3]$
- Ground truth label: one-hot vector
  - E.g. $y = [0, 0, 1]$



$$p_i = \frac{\exp(z_i)}{\sum_{j=1}^{3} \exp(z_j)}$$

Cross entropy loss (multi-class)

$$L_{CE} = -\sum_{i=1}^{K} y_i \log(p_i)$$

- Model assigns high probability to correct class:
  - $p_i \approx 1 \implies \log p_i \approx 0 \implies$ **small** loss
- Model assigns low probability to correct class:
  - $p_i \approx 0 \implies \log p_i \approx -\infty \implies$ **large** loss

Where does cross entropy loss come from?

- Minimize the KL Divergence between two distributions:

- $$\min_{p_2} \mathrm{KL}\left(p_1, p_2\right) = \min_{p_2} - \sum_x p_1(x) \log\left(\frac{p_2(x)}{p_1(x)}\right)$$

$$\equiv \min_{p_2} \sum_x - p_1(x) \log p_2(x) + p_1(x) \log p_1(x)$$

$$\equiv \min_{p_2} - \sum_x p_1(x) \log p_2(x)$$

- In our example:

    - $p_1 = [0,0,1]$, and $p_2 = [0.2,0.5,0.3]$

Cross entropy loss (in code)

```
def ce_loss(logits, target):
    log_probs = torch.nn.functional.log_softmax(logits, dim=1)
    loss = -log_probs[:, target]
    return loss
```

Implemented in standard libraries, e.g. nn.CrossEntropyLoss

Training neural network models

- We use *gradient descent*
  - Write down a *loss function*
  - ***Calculate gradients*** **of the loss function with respect to the parameters**
  - Move the parameters in the direction that *reduces the loss function*

Calculating gradients

- $p = \sigma(\underbrace{wx + b}_{z})$, where $\sigma(x) = \dfrac{1}{1 + \exp(-x)}$
- $L = -y \log p - (1-y)\log(1-p)$
- $\dfrac{\partial L}{\partial w} = \dfrac{\partial L}{\partial p}\dfrac{\partial p}{\partial z}\dfrac{\partial z}{\partial w}$
- $\dfrac{\partial L}{\partial p} = -\dfrac{y}{p} + \dfrac{1-y}{1-p}$

  $= \dfrac{p-y}{p(1-p)}$
- $\dfrac{\partial p}{\partial z} = p(1-p)$
- $\dfrac{\partial z}{\partial w} = x$
- Multiplying the three terms, we get $\dfrac{\partial L}{\partial w} = (p-y)x$

Coming up soon:
gradient computation
handled automatically

Training neural network models

- We use *gradient descent*
  - Write down a *loss function*
  - *Calculate gradients* of the loss function with respect to the parameters
  - **Move the parameters in the direction that *reduces the loss function***

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! (e.g., see Ruder 2016 in the references.)

In Code

Loss
Optimizer

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=5e-4)

for EPOCH in range(10):
    random.shuffle(train)
    train_loss = 0.0
    start = time.time()
    model.train()
    for x, y in train:
        x = torch.tensor(x, dtype=torch.long)
        y = torch.tensor([y])
        logits = model(x)
        loss = criterion(logits, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Compute loss

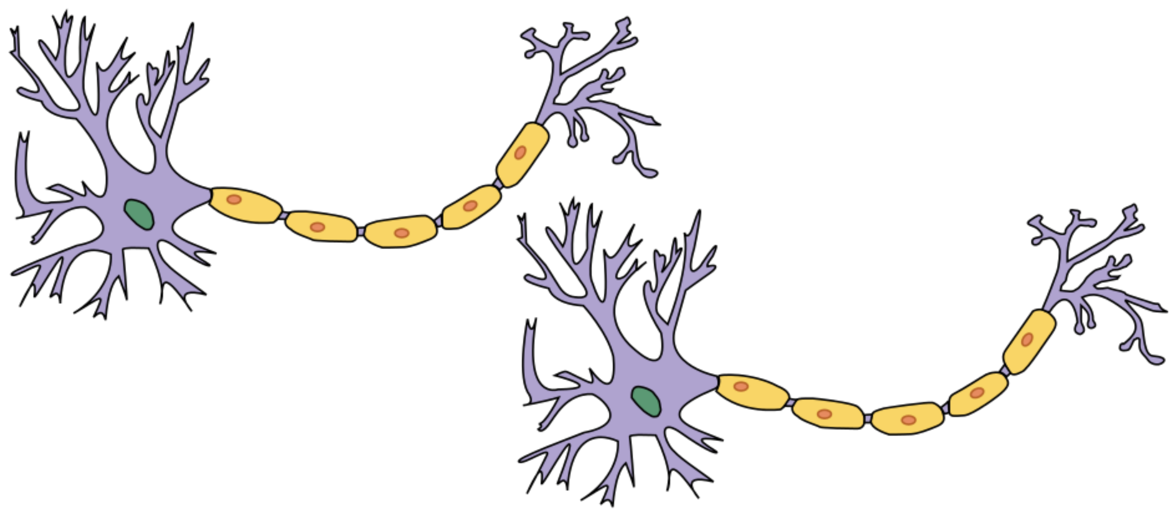
Compute gradients

Update parameters

What is a Neural Net?: Computation Graphs

“Neural” Nets

Original Motivation: Neurons in the Brain



Current Conception: Computation Graphs

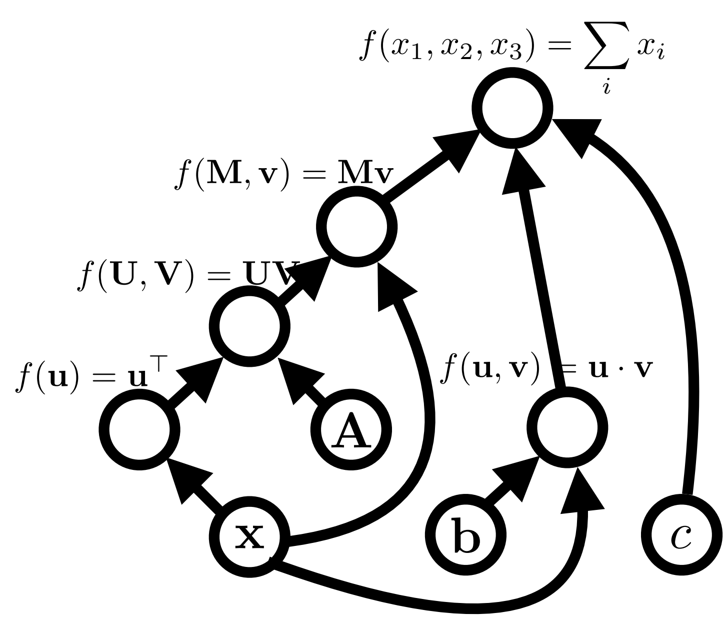


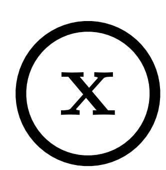
Image credit: Wikipedia

expression:

$\mathbf{x}$

graph:

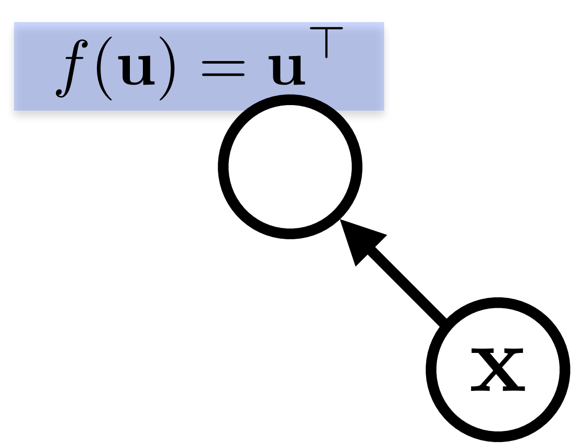
A **node** is a {tensor, matrix, vector, scalar} value



An **edge** represents a function argument. 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 *gradient with respect to each input, here* $\frac{\partial f(\mathbf{u})}{\partial \mathbf{u}}$



$$\frac{\partial F}{\partial \mathbf{u}} = \frac{\partial F}{\partial f(\mathbf{u})} \frac{\partial f(\mathbf{u})}{\partial \mathbf{u}}$$

Incoming gradient

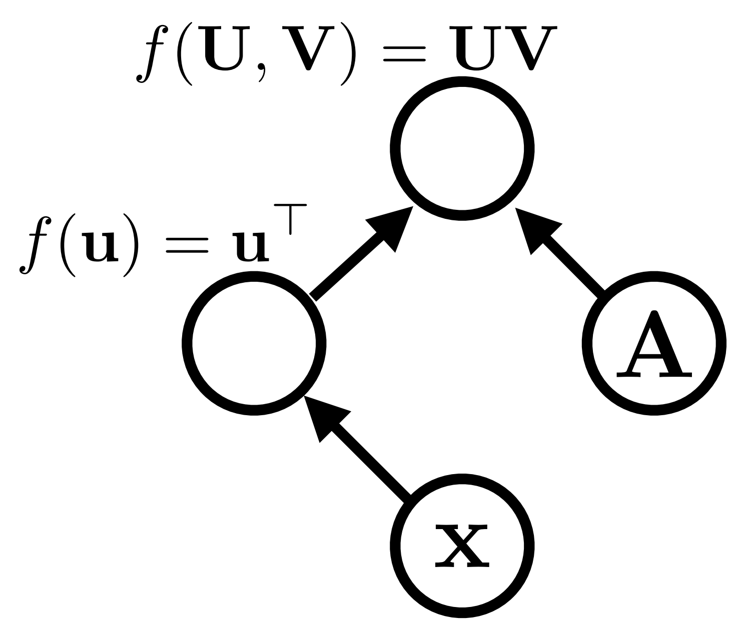
Local Gradient

expression:

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

graph:

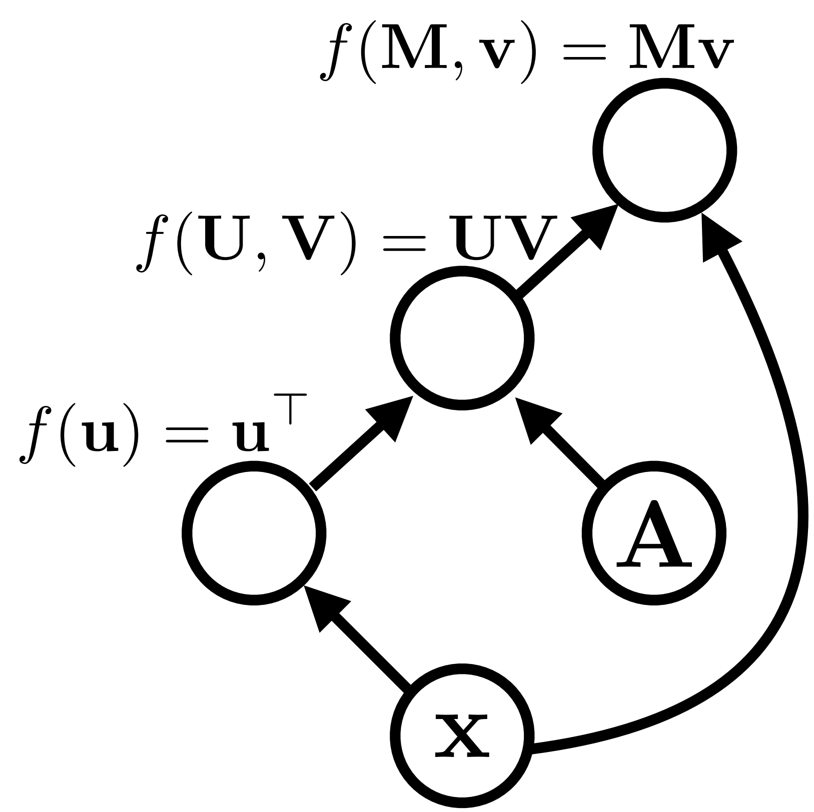
Functions can be nullary, unary,
binary, … *n*-ary. Often they are unary or binary.



expression:

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

graph:

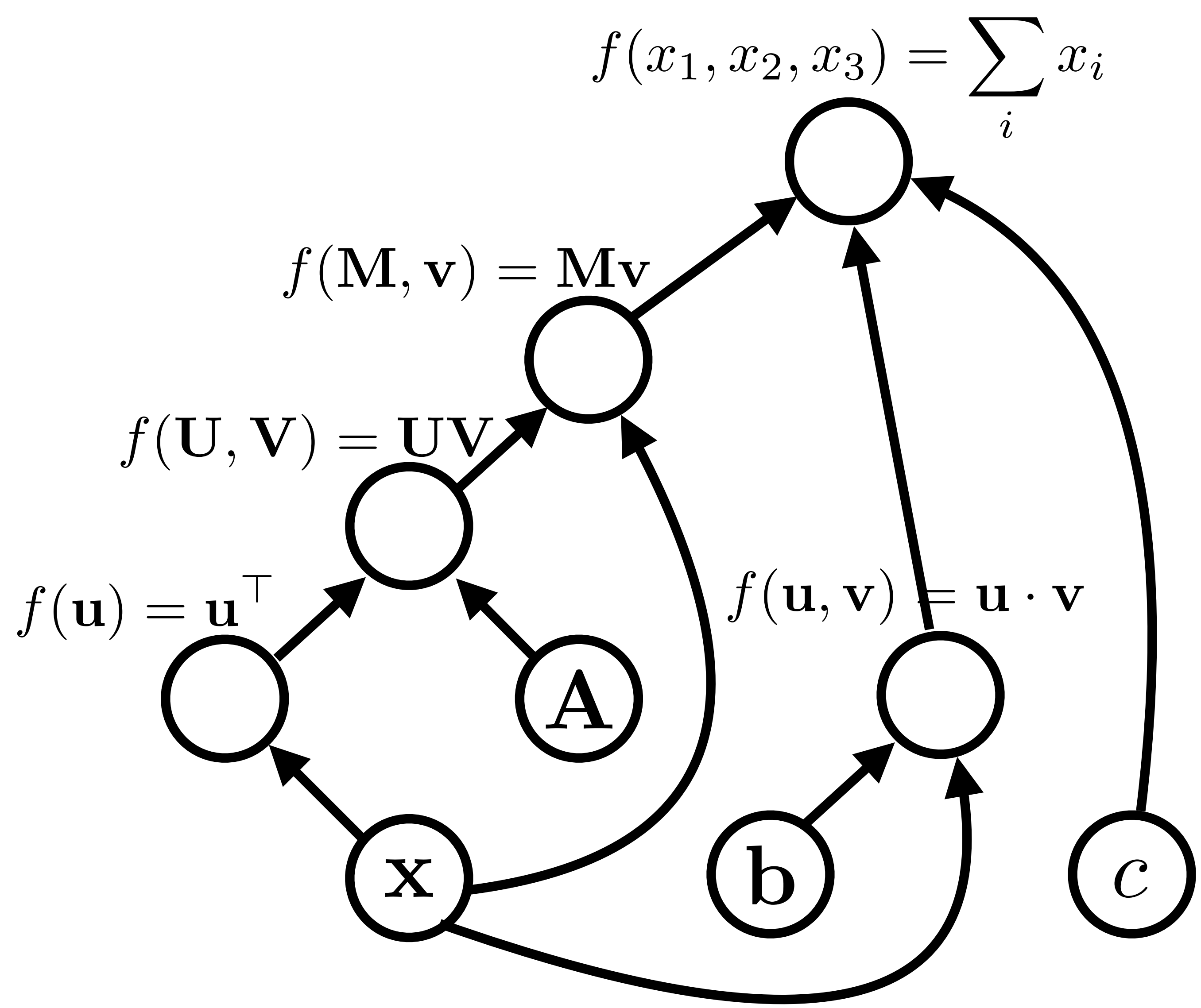


Computation graphs are directed and acyclic (in DyNet)

expression:

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

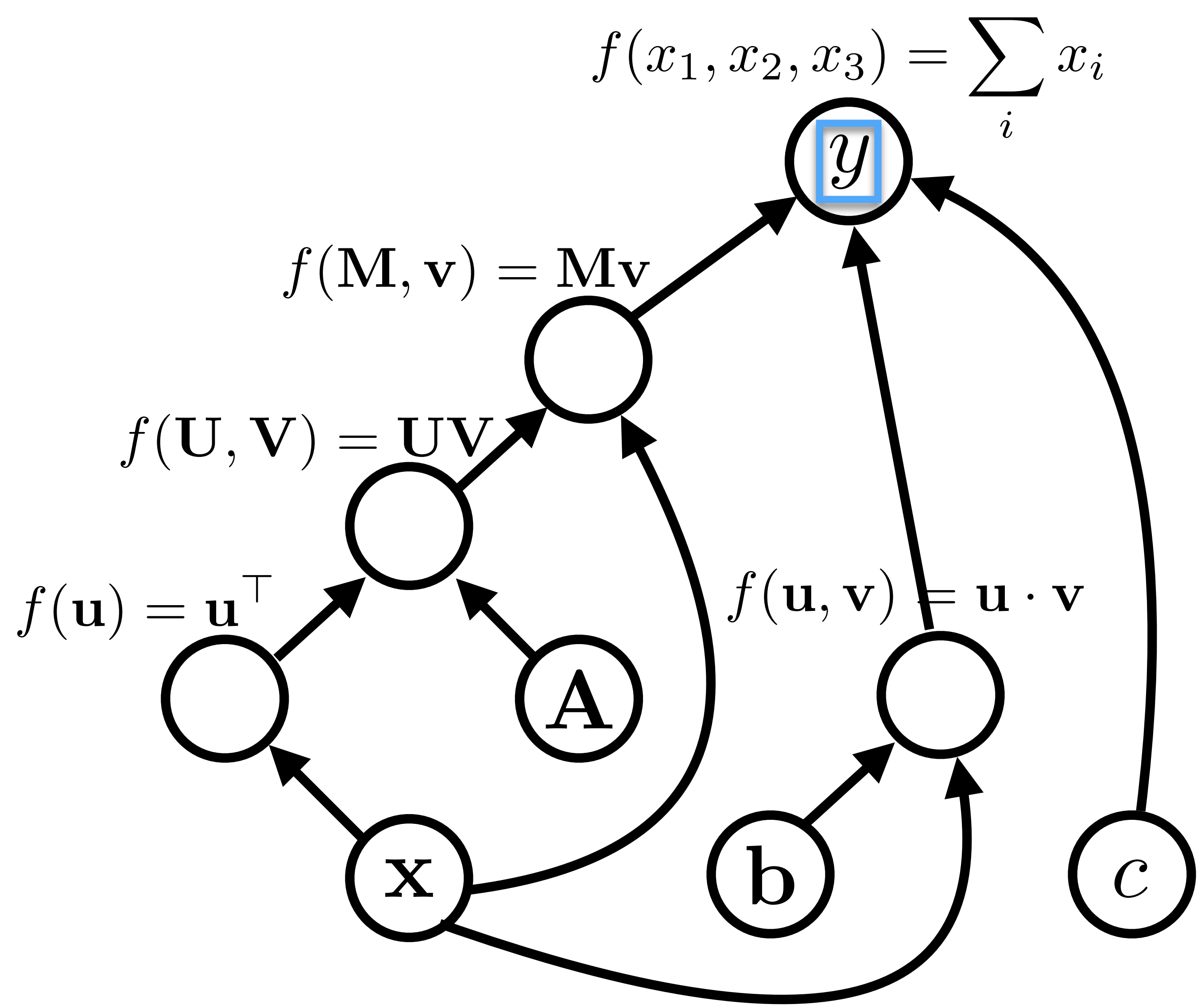
graph:



expression:

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

graph:



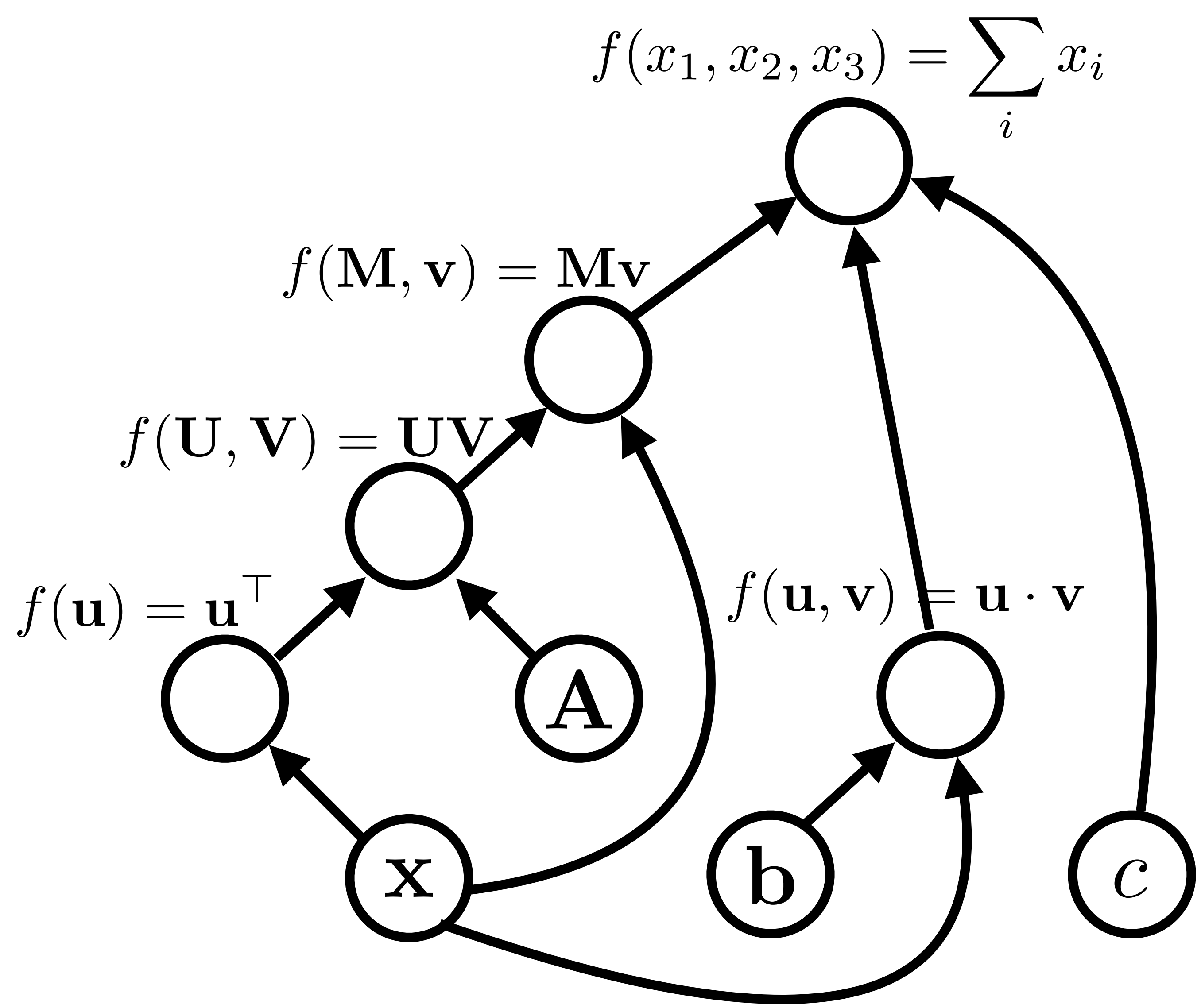
variable names are just labelings of nodes.

Algorithms (1)

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

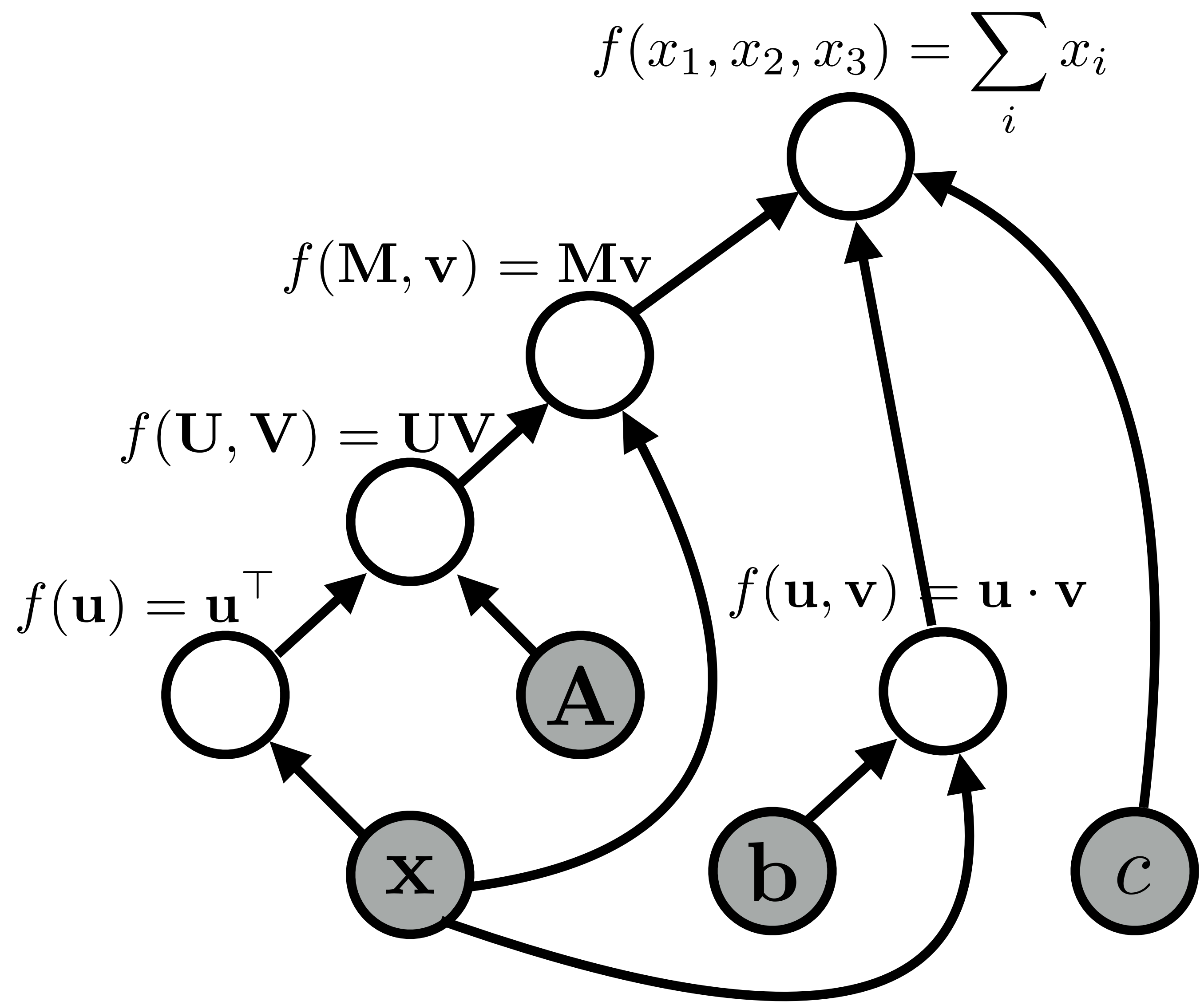
Forward Propagation

graph:



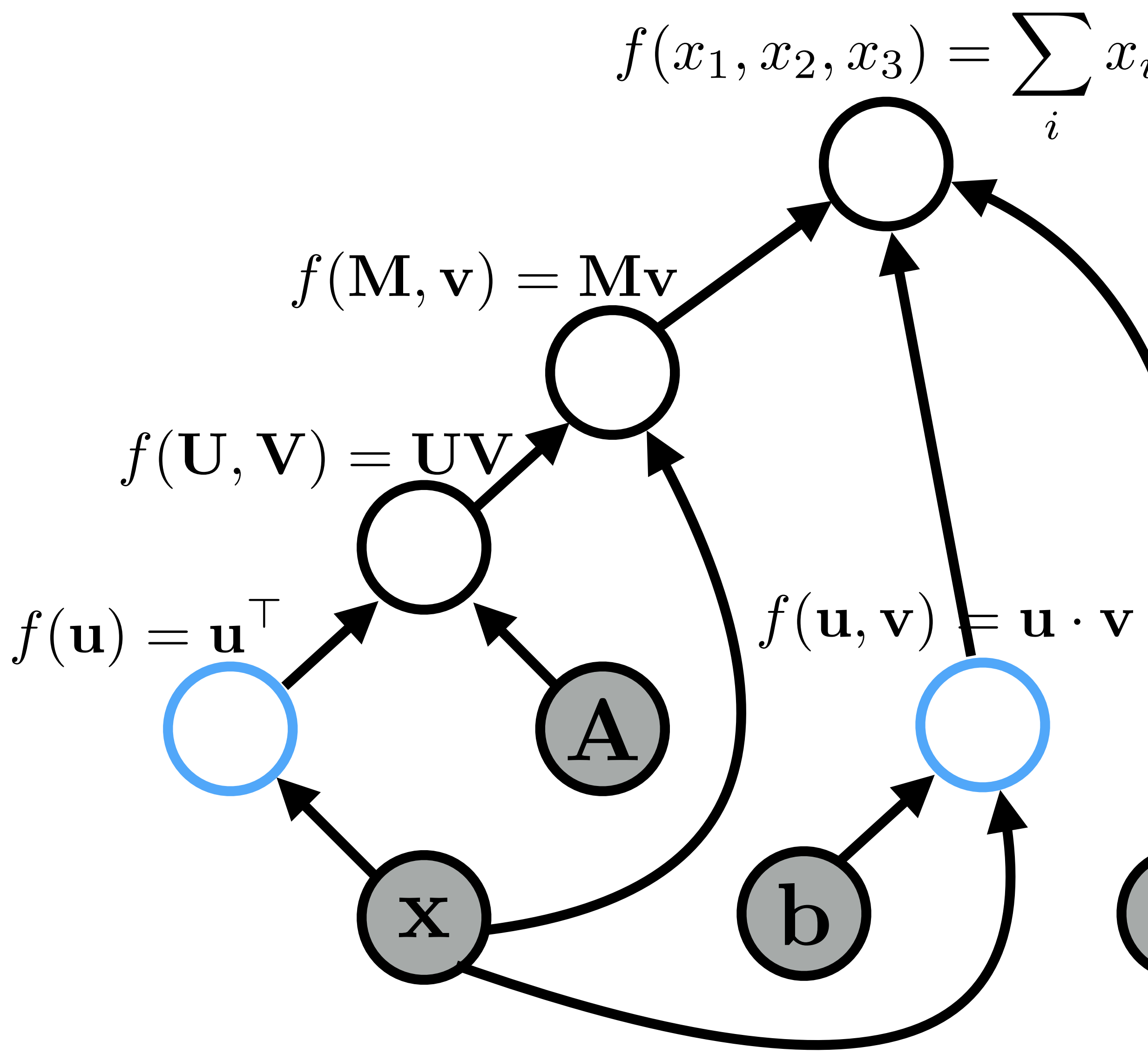
Forward Propagation

graph:



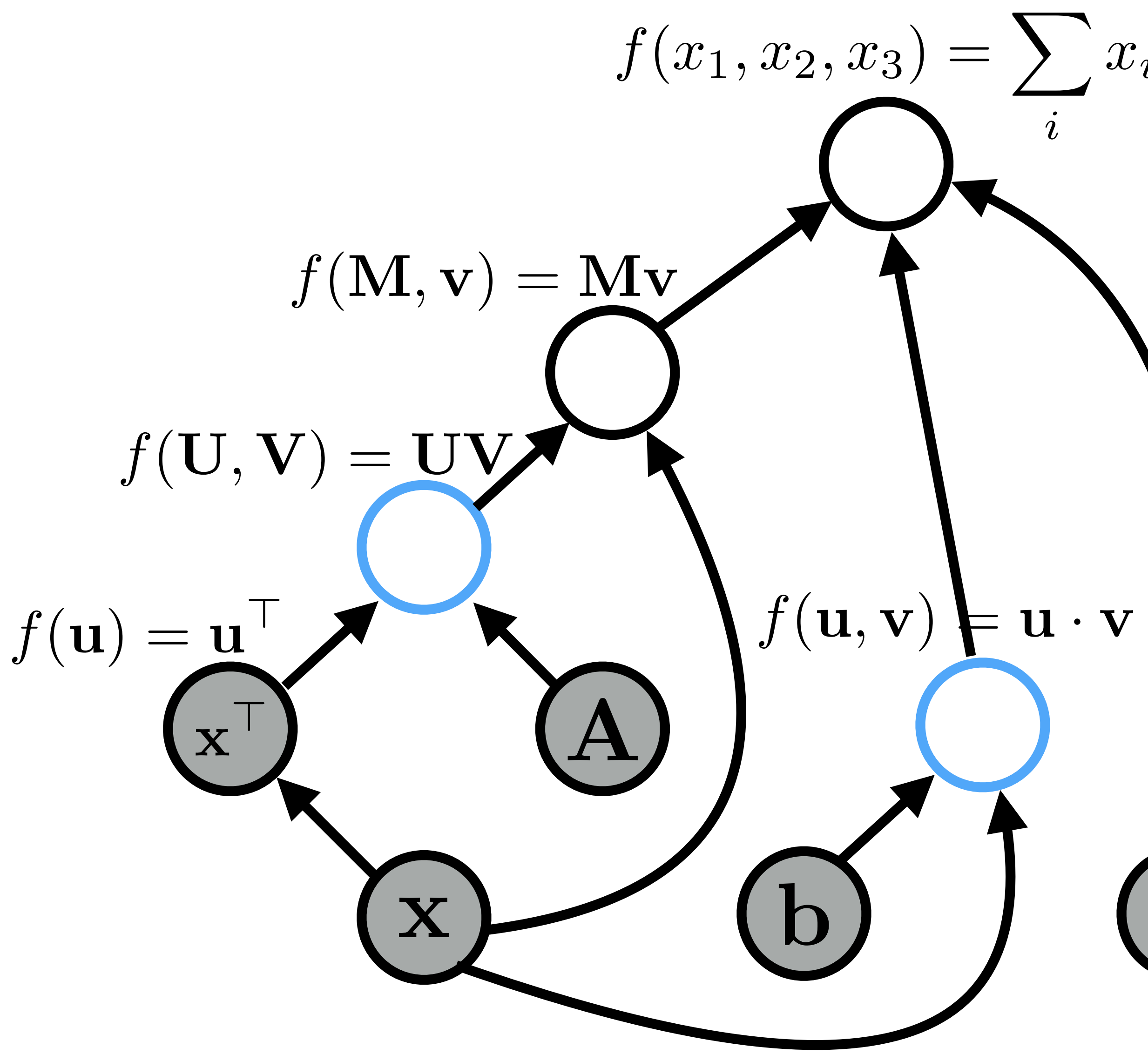
Forward Propagation

graph:



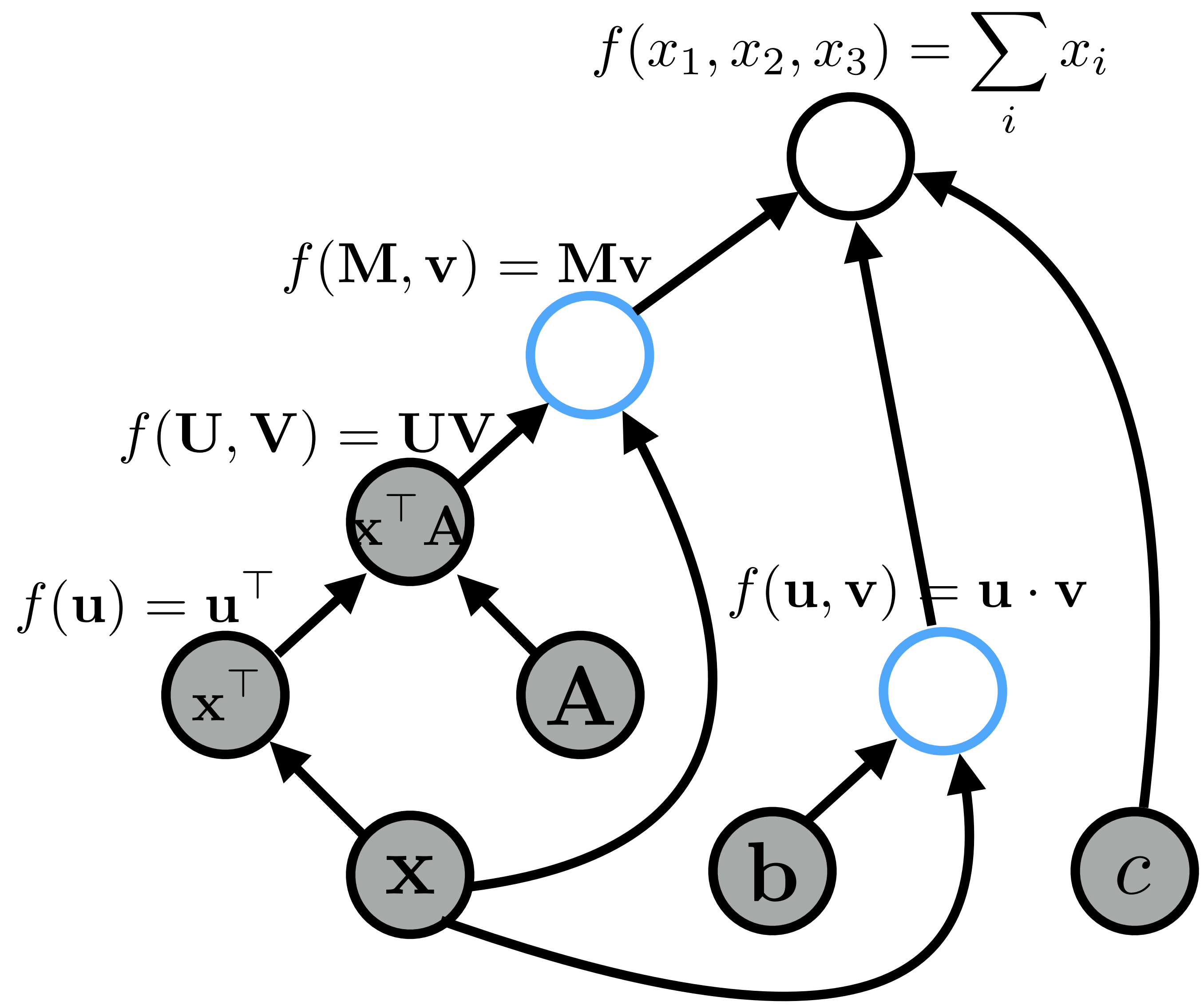
Forward Propagation

graph:



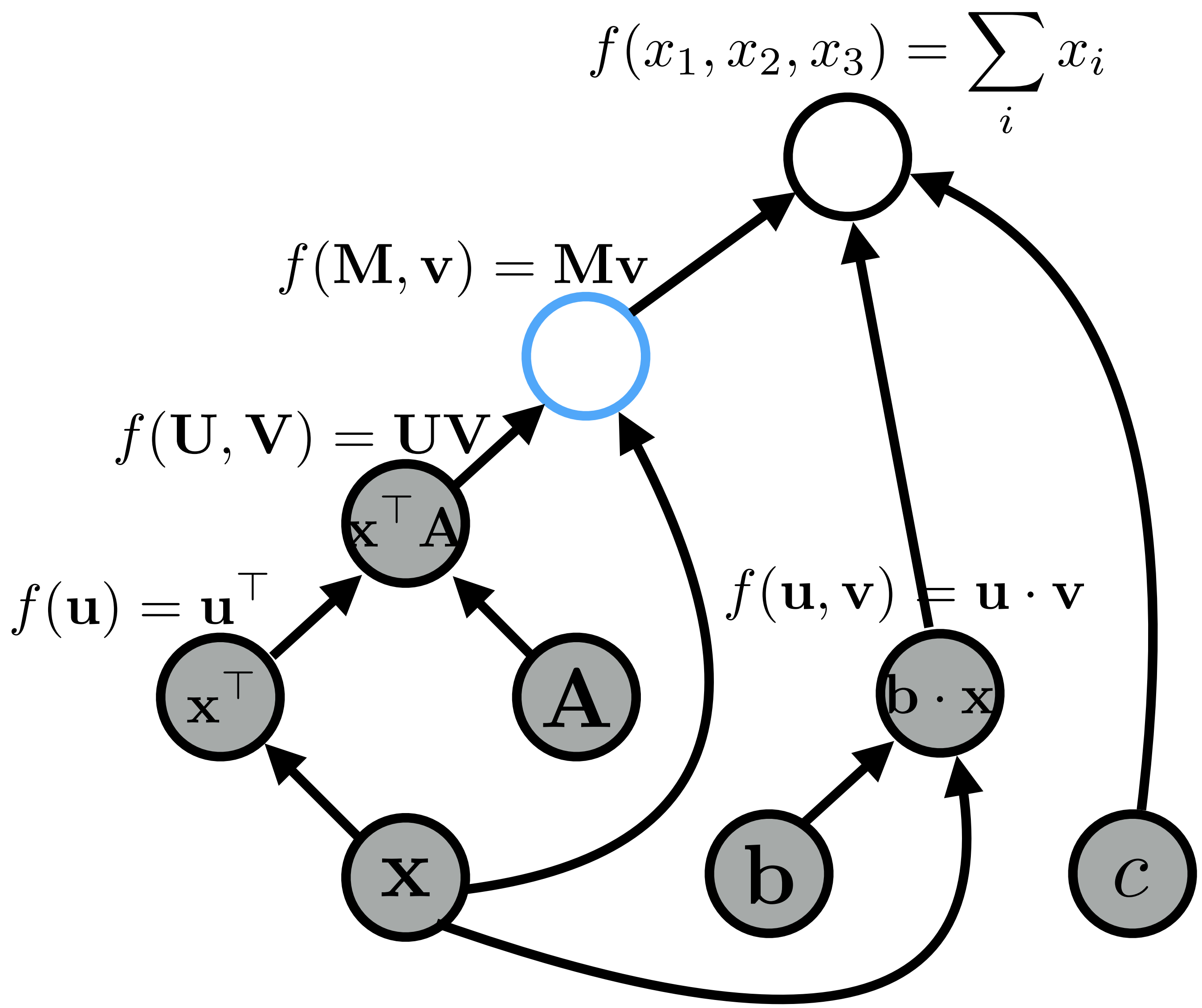
Forward Propagation

graph:



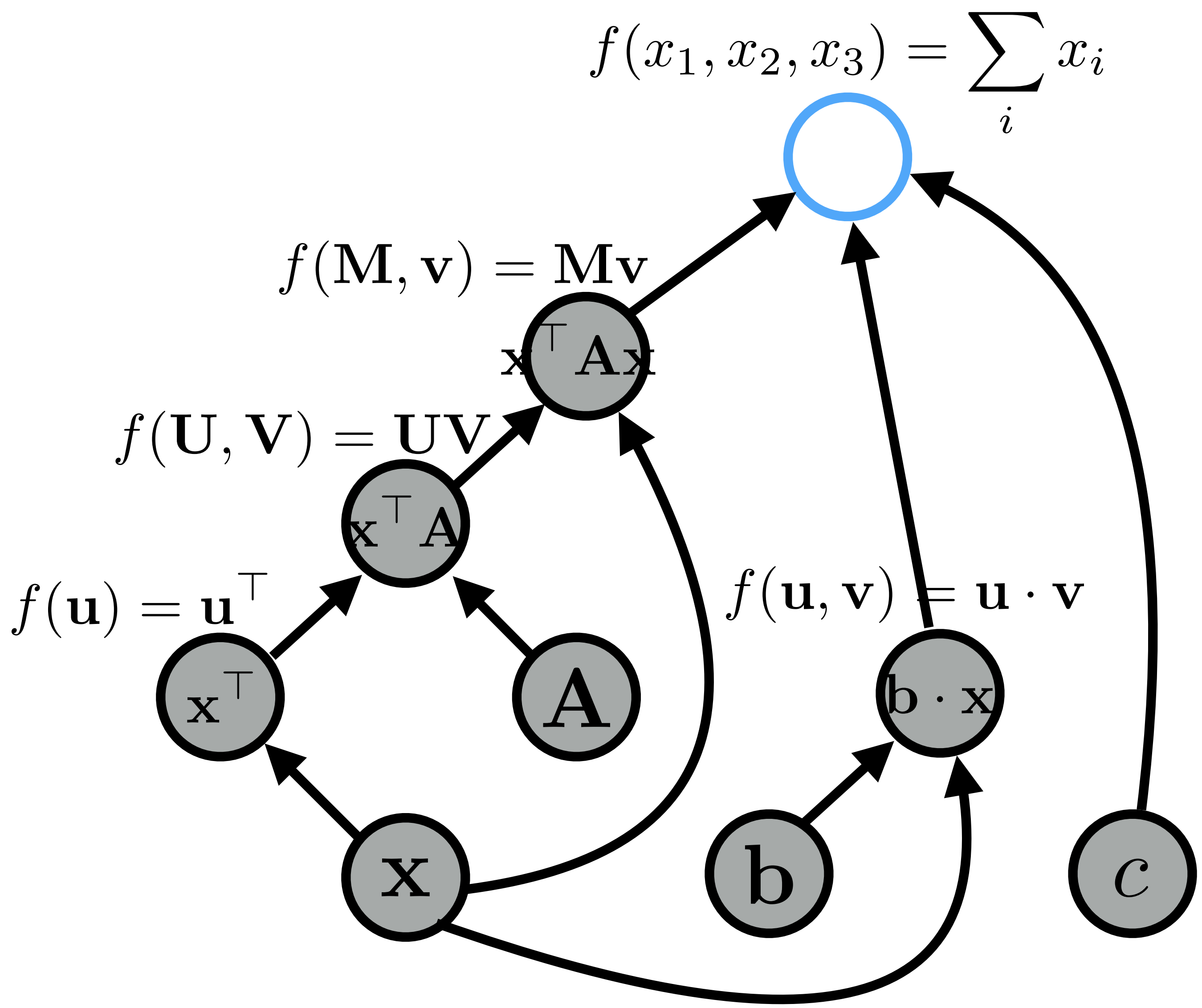
Forward Propagation

graph:



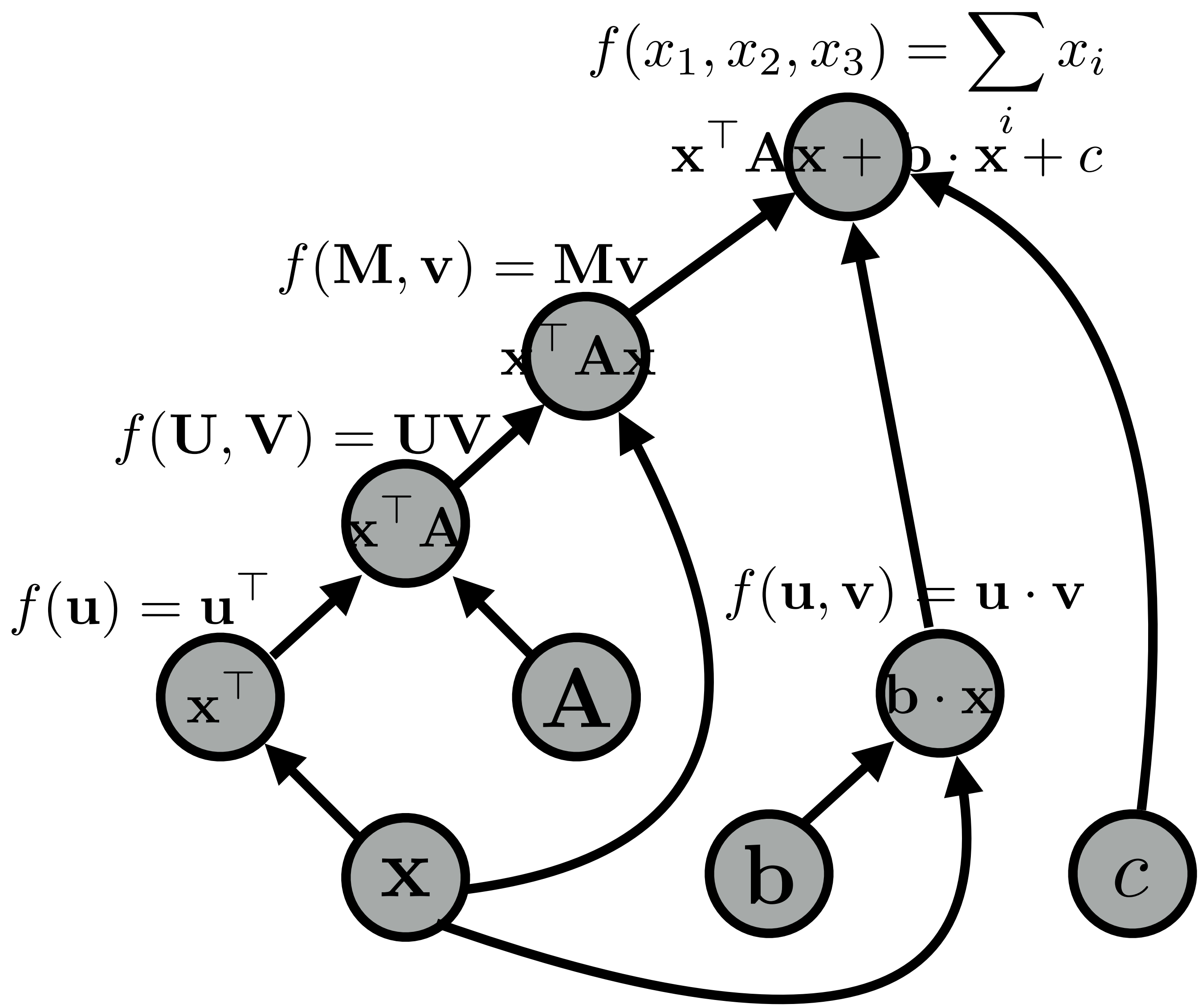
Forward Propagation

graph:



Forward Propagation

graph:

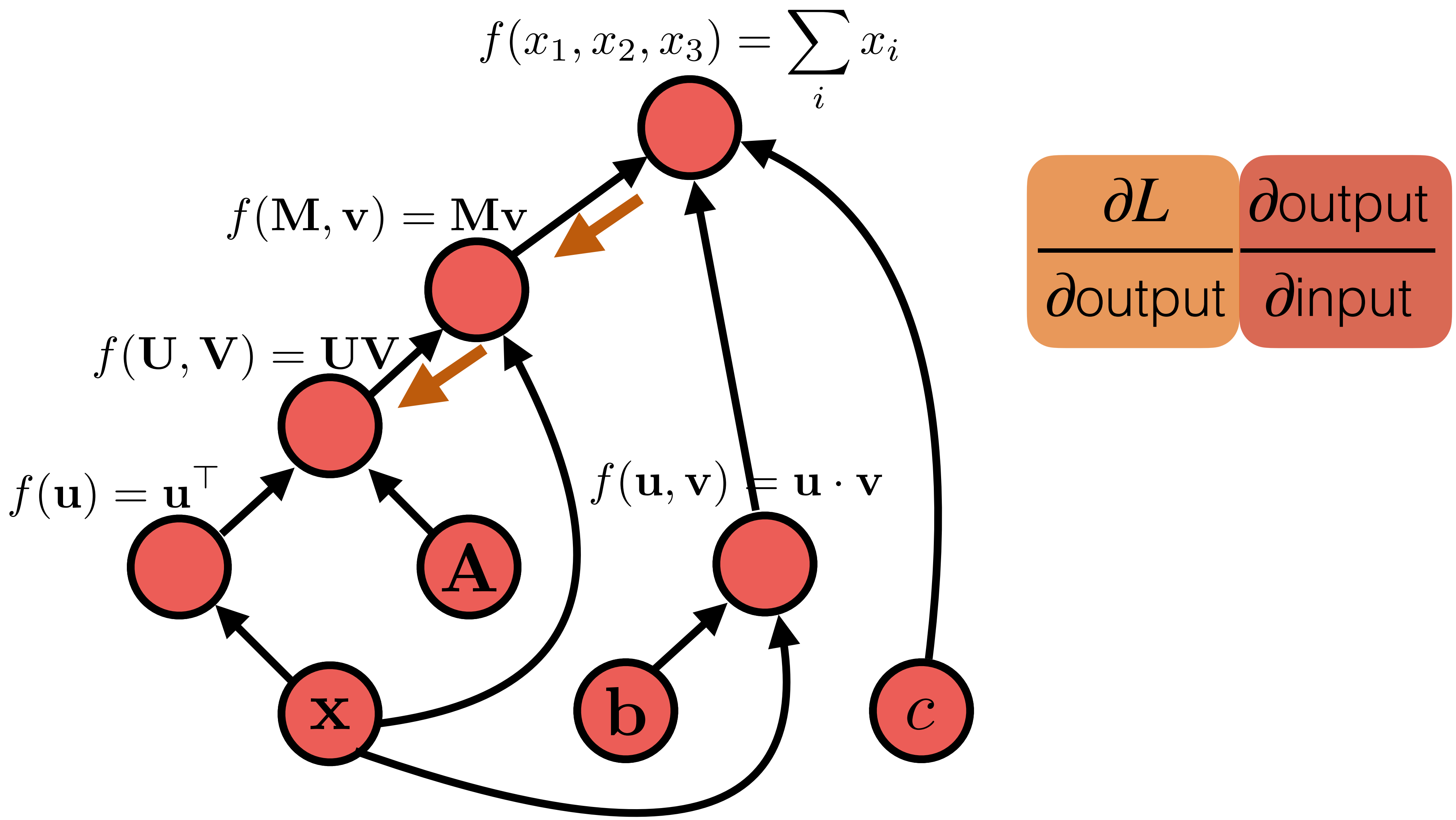


Algorithms (2)

- **Back-propagation:**
  - Process examples in reverse topological order
  - Calculate the gradients of the parameters with respect to the final value (usually a loss function)
- **Parameter update:**
  - Move the parameters in the direction of this gradient
    W -= α * dl/dW

Back Propagation

graph:



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**

Concrete Implementation

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

Code Example

- Classify tweets as positive, negative, or neutral
- BoW, CBoW, DeepCBoW

```
[131]
# Classify an example with our trained model
tweet = "I'm learning so much in advanced NLP!"
tokens = torch.tensor(sp.encode(tweet), dtype=torch.long)
scores = model(tokens)[0].detach()
predict = scores.argmax().item()
label_to_text[predict]

... 'positive'
```

Recap

- Tokenization and subword models
  - Represent sequences as tokens determined based on frequency
- Token embeddings
  - Represent tokens as learned continuous vectors in $\mathbb{R}^d$
- Neural networks
  - Learn complex, non-linear feature functions
- Training a neural network
  - Choose a loss, construct a differentiable graph, take gradients

Thank you!