CS5740: Natural Language Processing

Computation Graphs

Instructor: Yoav Artzi

Some slides from Practical Neural Networks for NLP / Chris Dyer, Yoav Goldberg, Graham Neubig / EMNLP 2016

Computation Graphs

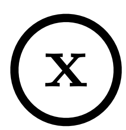
- The descriptive language of deep learning models
- Functional description of the required computation
- Can be instantiated to do two types of computation:
  - Forward computation
  - Backward computation

expression:

$\mathbf{x}$

graph:

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

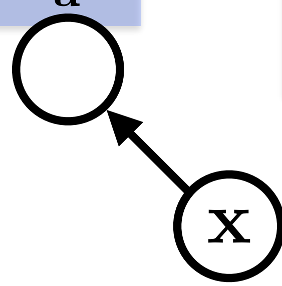


An **edge** represents a function argument (and also data dependency). They are just pointers to nodes.

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* $\frac{\partial \mathcal{F}}{\partial f(\mathbf{u})}$.

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



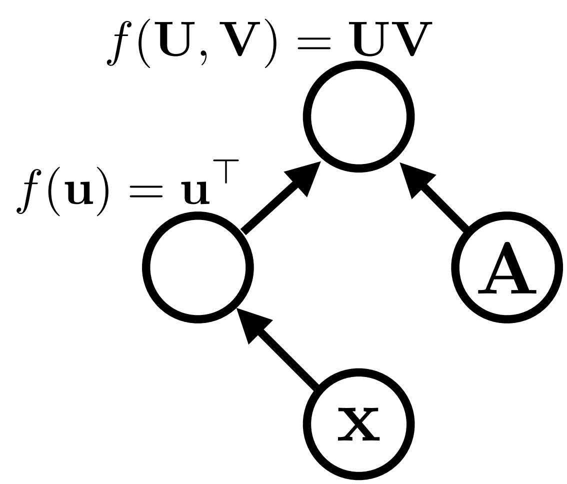
$$\frac{\partial f(\mathbf{u})}{\partial \mathbf{u}} \frac{\partial \mathcal{F}}{\partial f(\mathbf{u})} = \left(\frac{\partial \mathcal{F}}{\partial f(\mathbf{u})} \right)^\top$$

expression:

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

graph:

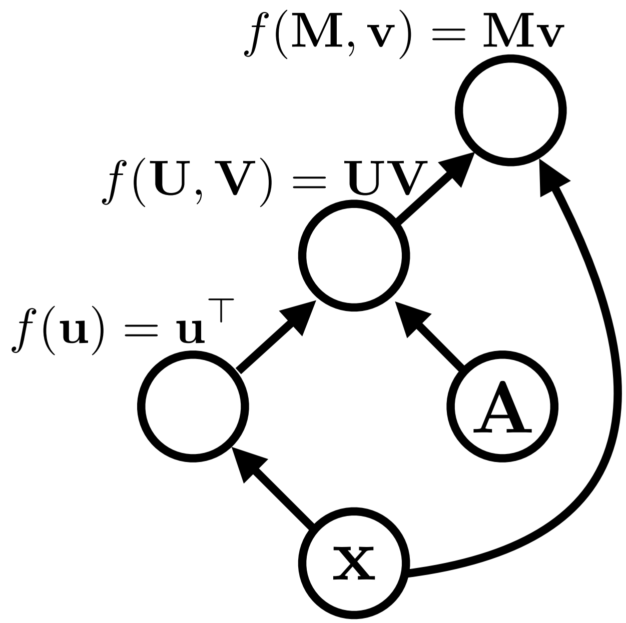
Functions can be nullary, unary, binary, … *n*-ary. Often they are unary or binary.



expression:

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

graph:

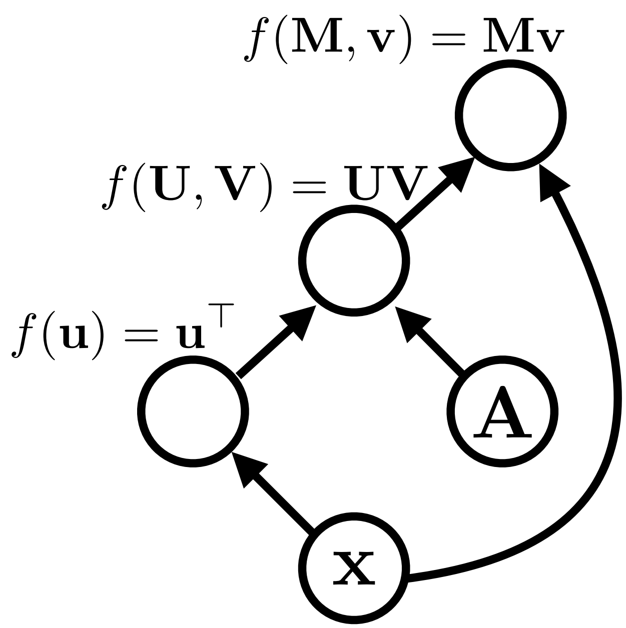


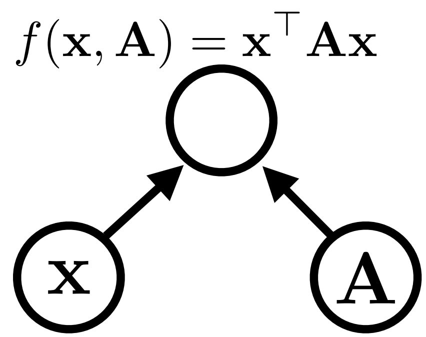
Computation graphs are directed and acyclic (usually)

expression:

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

graph:





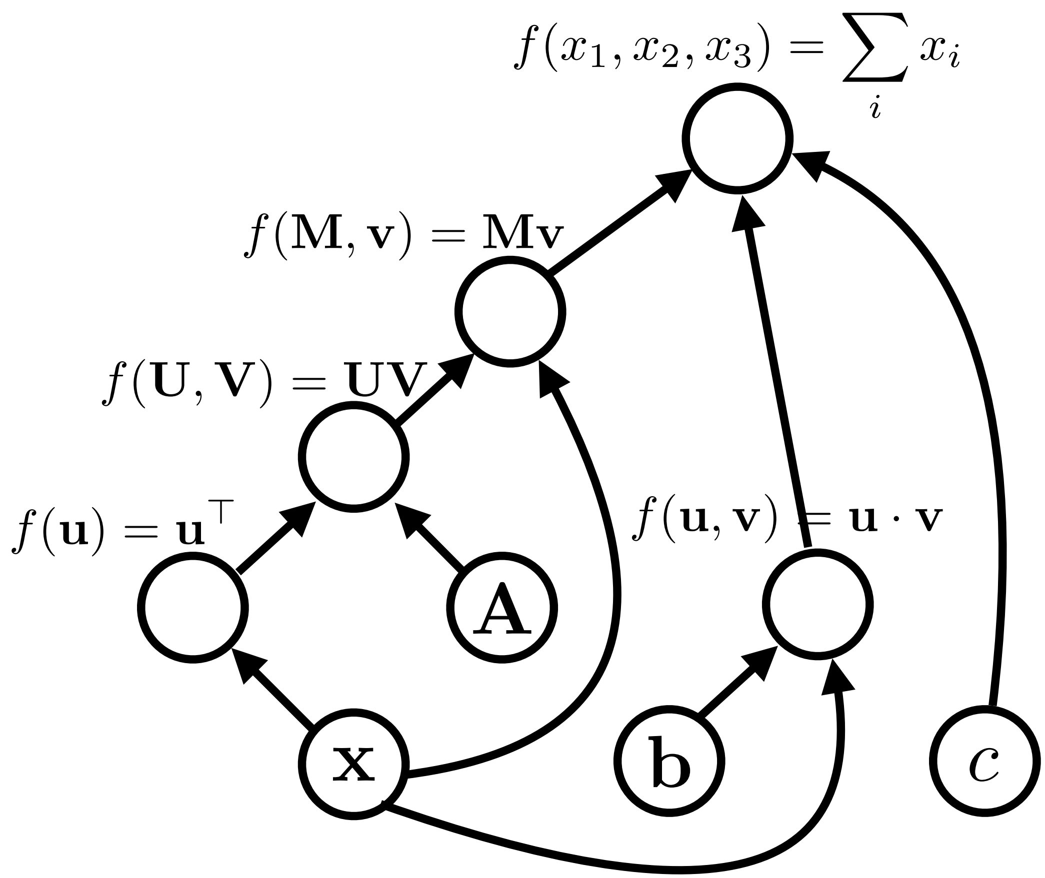
$$\frac{\partial f(\mathbf{x}, \mathbf{A})}{\partial \mathbf{x}} = (\mathbf{A}^\top + \mathbf{A})\mathbf{x}$$

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

expression:

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

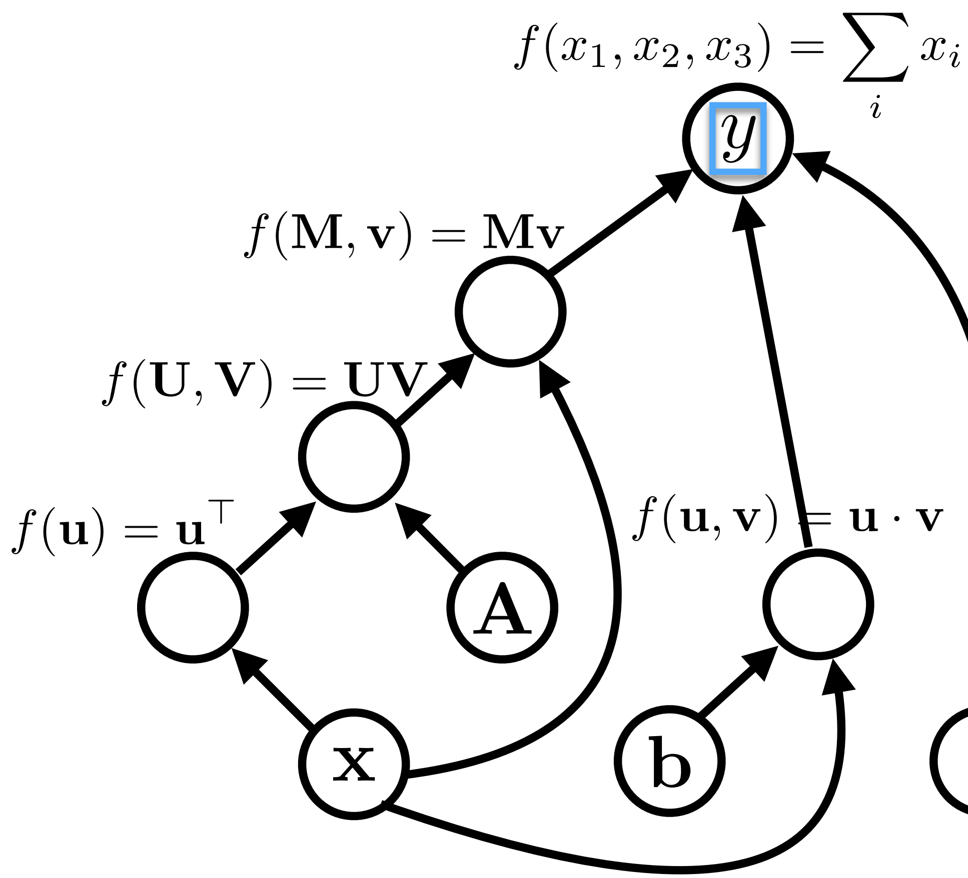
graph:



expression:

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

graph:



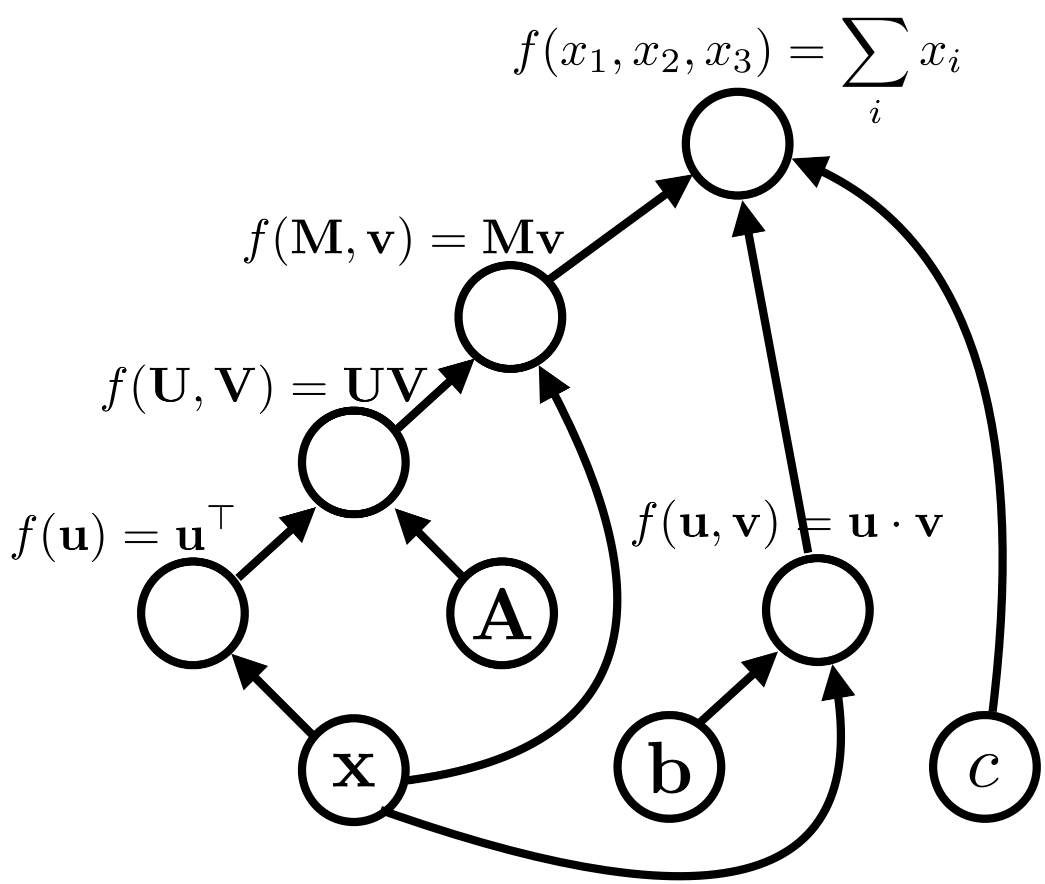
variable names are just labelings of nodes.

Algorithms

- **Graph construction**
- **Forward propagation**
  - Loop over nodes in topological order
    - Compute the value of the node given its inputs
  - *Given my inputs, make a prediction (or compute an "error" with respect to a "target output")*
- **Backward propagation**
  - Loop over the nodes in reverse topological order starting with a final goal node
    - Compute derivatives of final goal node value with respect to each edge's tail node
  - *How does the output change if I make a small change to the inputs?*

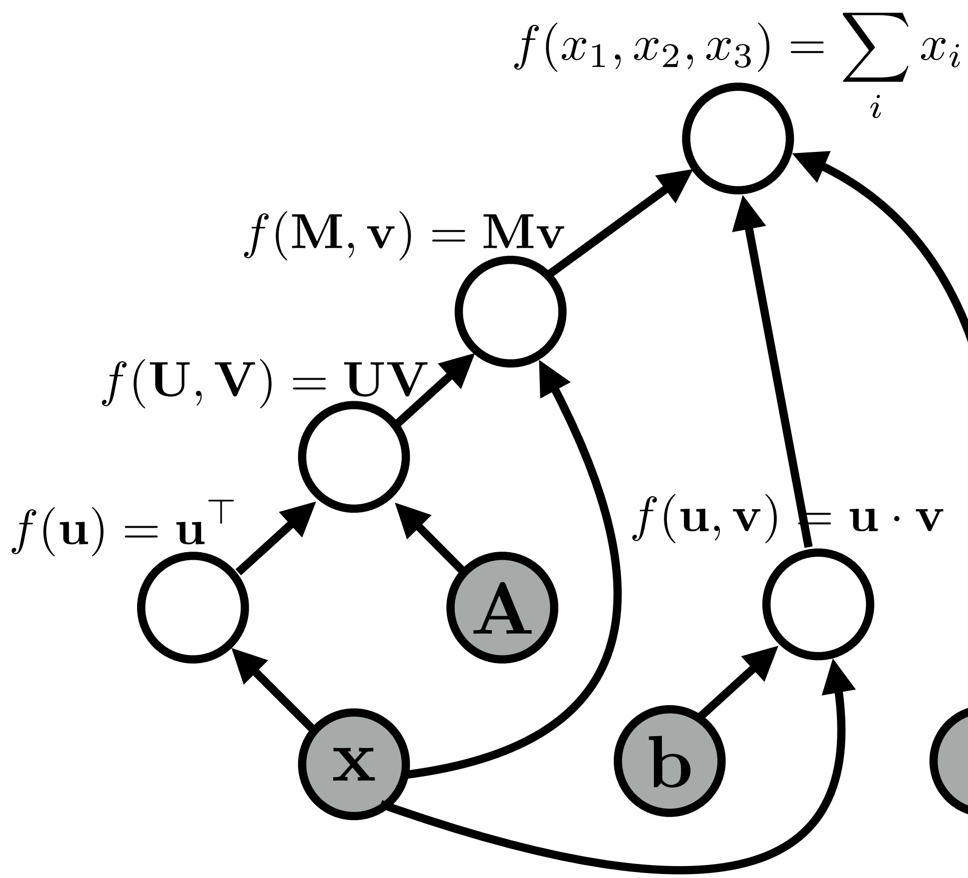
Forward Propagation

graph:



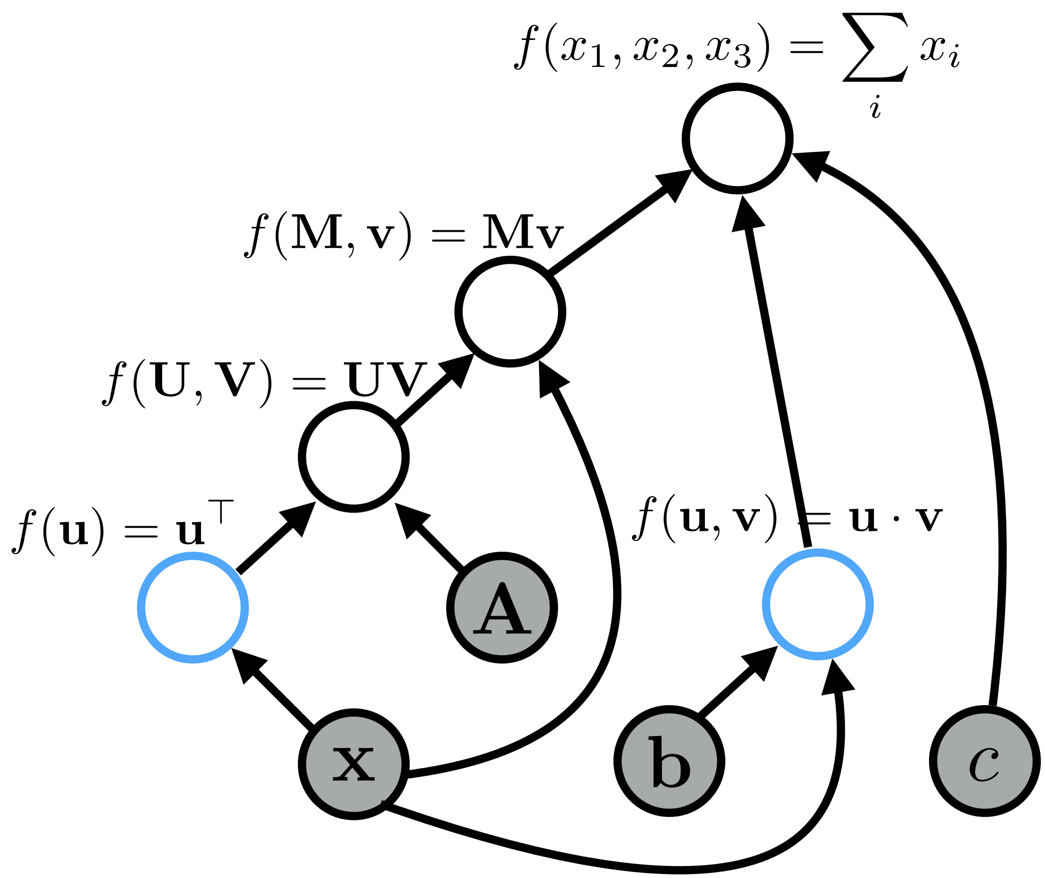
Forward Propagation

graph:



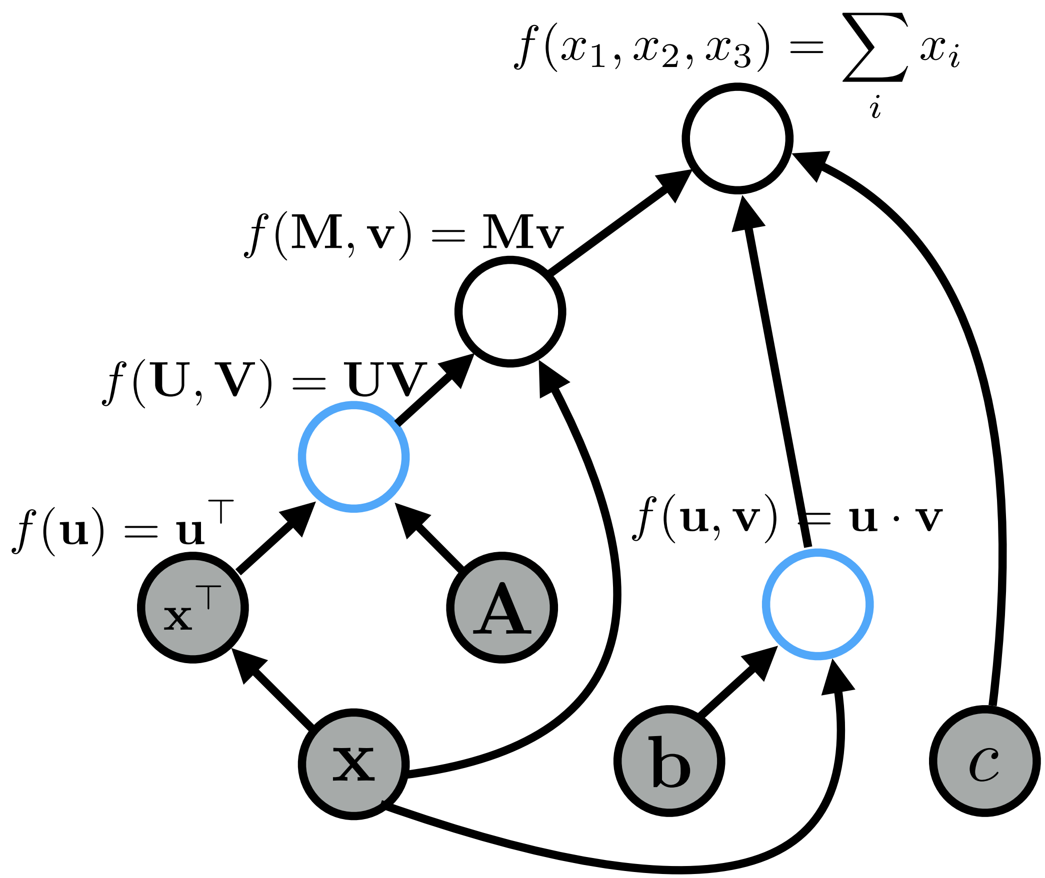
Forward Propagation

graph:



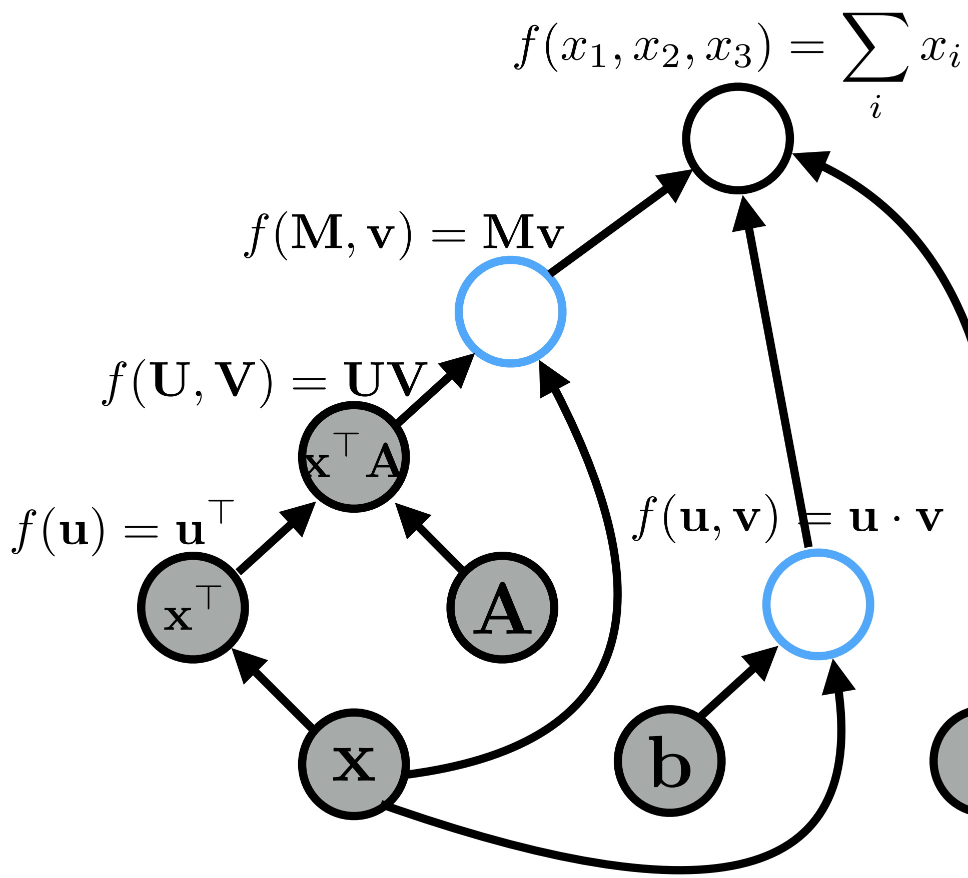
Forward Propagation

graph:



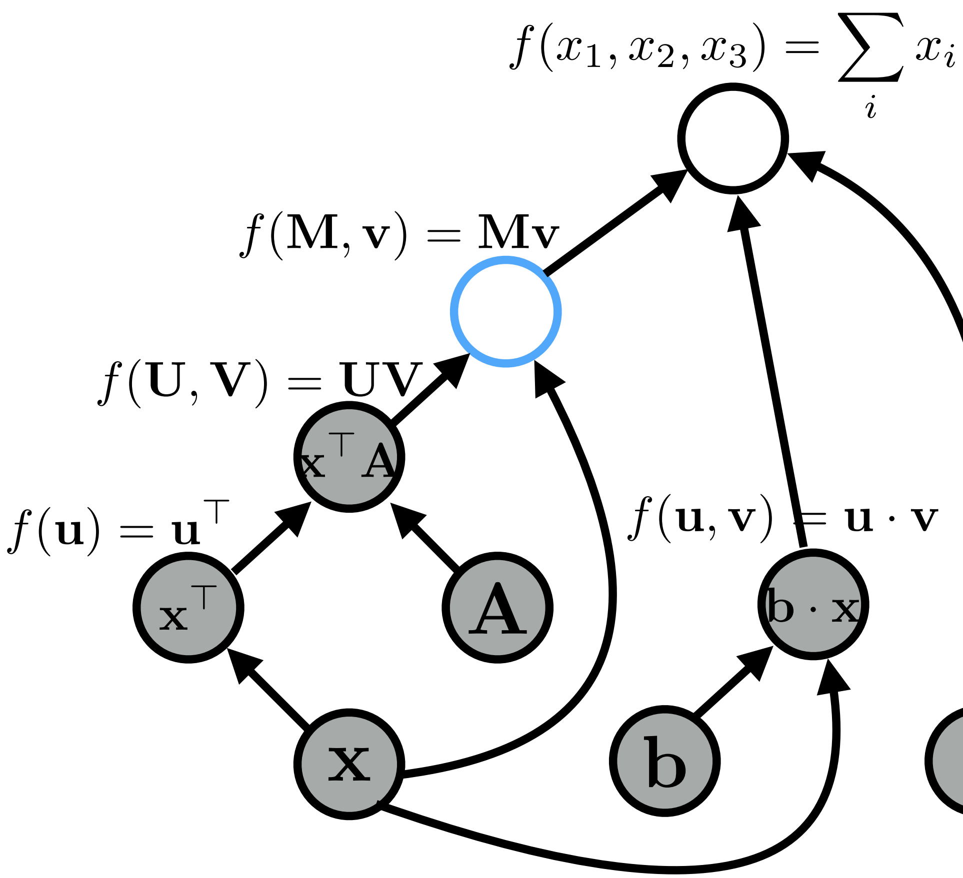
Forward Propagation

graph:



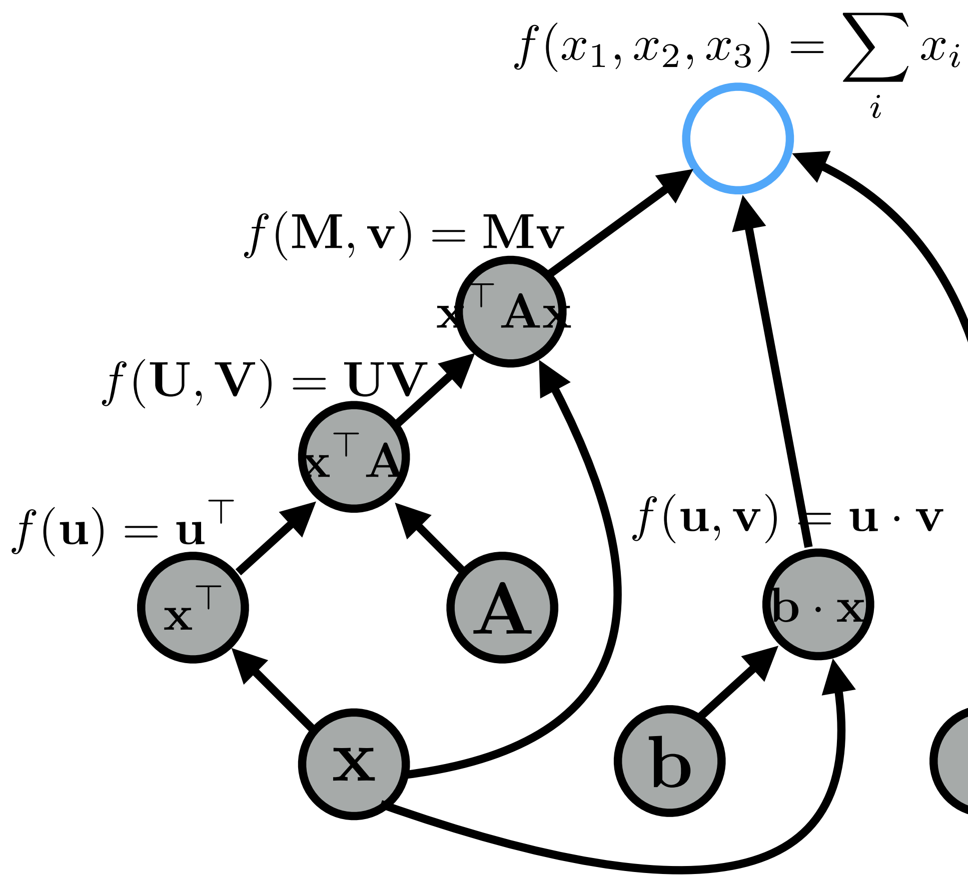
Forward Propagation

graph:



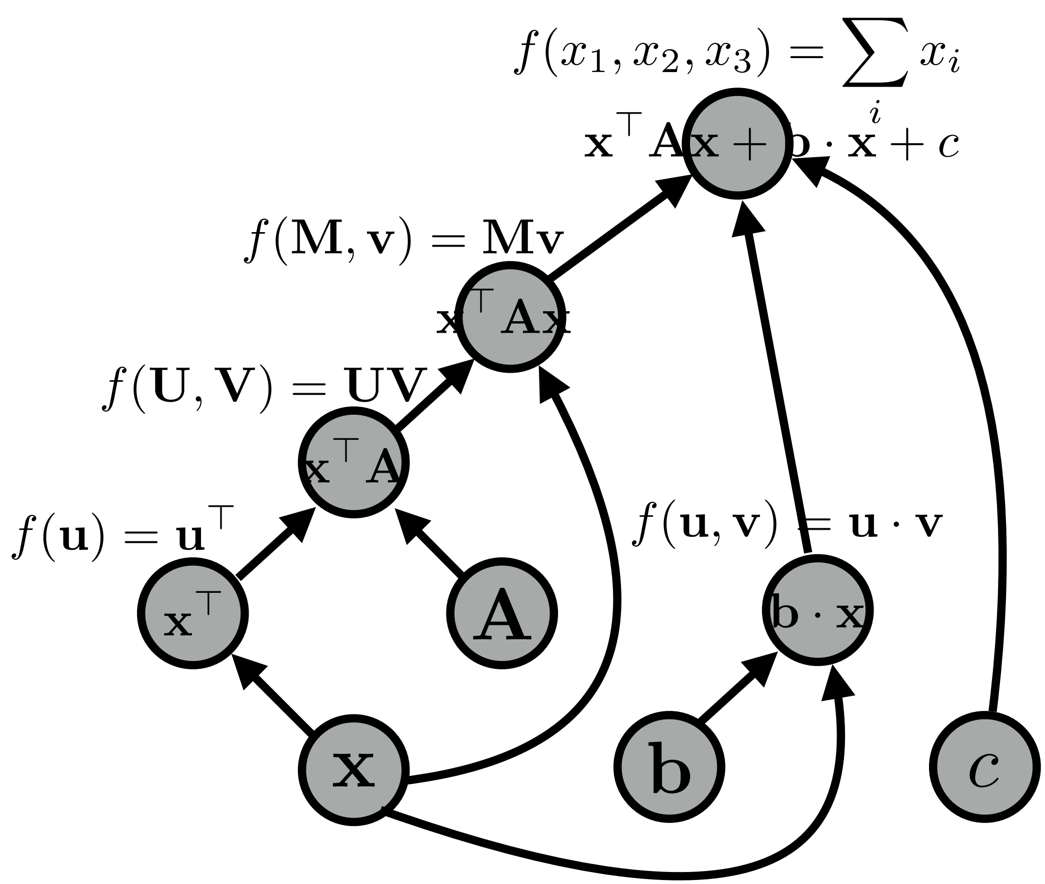
Forward Propagation

graph:



Forward Propagation

graph:



Draw an MLP Computation Graph

$$\mathbf{h}^1 = \sigma([\phi(x_l);\phi(x_r)]\mathbf{W}^1 + \mathbf{b}^1)$$

$$\mathbf{h}^2 = \sigma(\mathbf{h}_1\mathbf{W}^2 + \mathbf{b}^2)$$

$$\mathbf{p} = \mathrm{softmax}(\mathbf{h}^2\mathbf{W}^3 + \mathbf{b}^3)$$

Constructing Graphs: Two Software Models

- **Static declaration**
  - Phase 1: define an architecture
    (maybe with some primitive flow control like loops and conditionals)
  - Phase 2: run a bunch of data through it to train the model and/or make predictions
- **Dynamic declaration**
  - Graph is defined implicitly (e.g., using operator overloading) as the forward computation is executed

Batching

- Two senses to processing your data in batch
    - Computing gradients for more than one example at a time to update parameters during learning
    - Processing examples together to utilize all available resources

Batching

- CPU: made of a small number of cores, so can handle some amount of work in parallel
- GPU: made of thousands of small cores, so can handle a lot of work in parallel
- Process multiple examples together to use all available cores

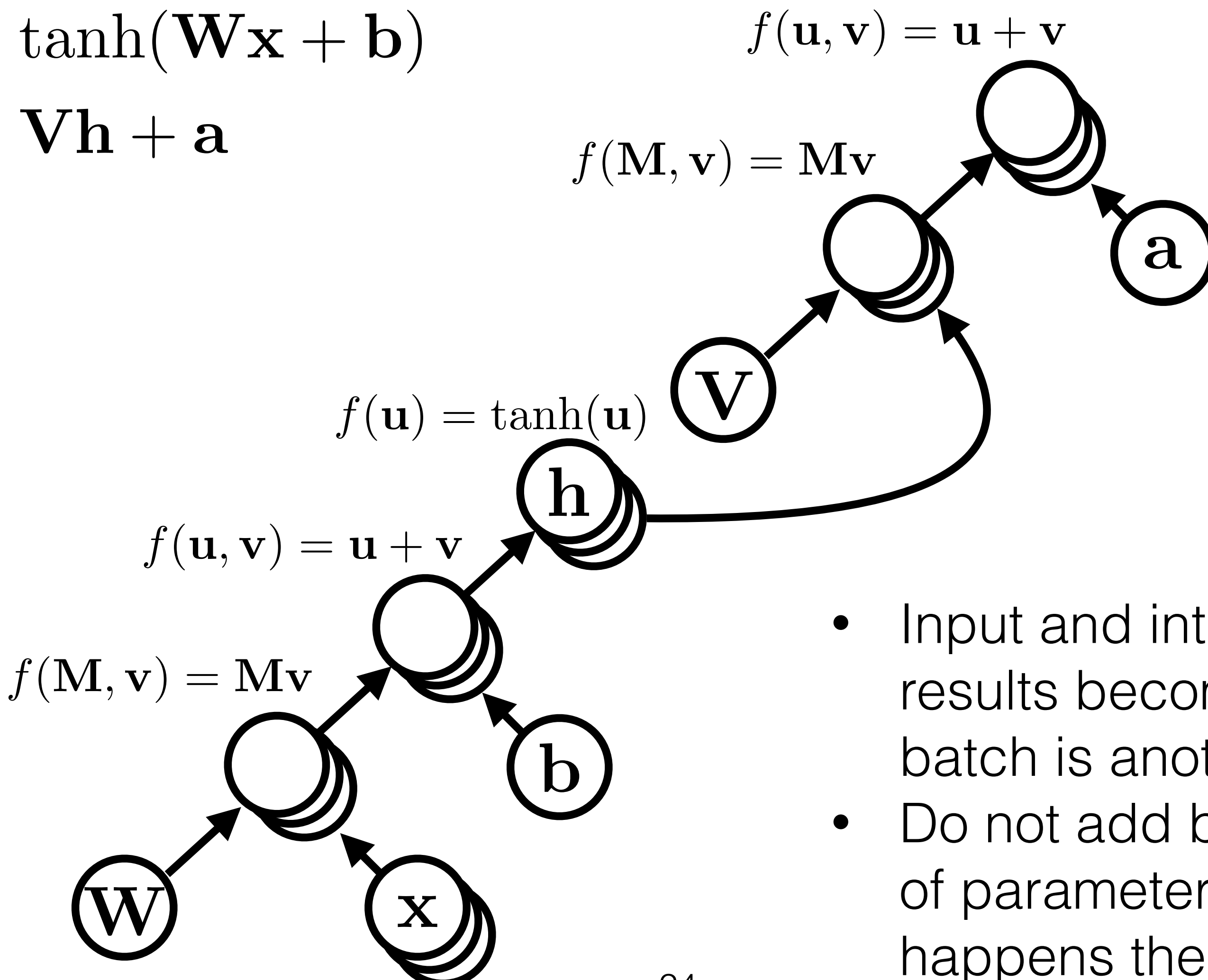
Batching

- Relatively easy when the network looks exactly the same for all examples
- More complex with language data: documents/sentences/words have different lengths
- Frameworks provide different methods to help common cases, but still require work on the developer side
- Key concept is broadcasting: https://pytorch.org/docs/stable/notes/broadcasting.html

The MLP

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\mathbf{y} = \mathbf{V}\mathbf{h} + \mathbf{a}$$



- Input and intermediate results become tensors — batch is another dimension!
- Do not add batch dimension of parameters! What happens then?

Rough notation for illustration only

No batching

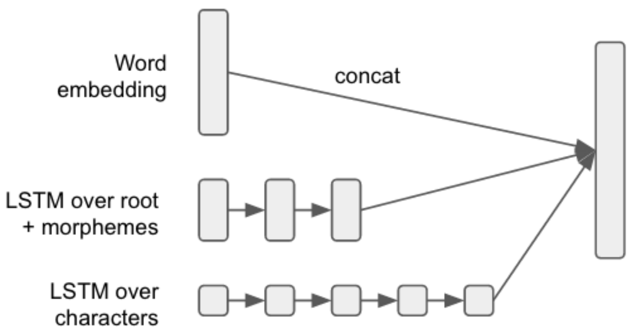
$$
\begin{aligned}
\mathbf{X}^{(j)} &= [x_1, \dots, x_{n^{(j)}}], x_i \in 1, \dots, |\mathcal{V}| \\
\mathbf{a} &= \frac{1}{|\mathbf{X}^{(j)}|} \text{sum}\left(\phi(\mathbf{X}^{(\text{j})})\right) \\
\mathbf{h}_1 &= \sigma(\mathbf{W}_1 \mathbf{a} + \mathbf{b}_1) \\
\mathbf{h}_2 &= \mathbf{W}_2 \mathbf{h}_1 + \mathbf{b}_2 \\
p &= \text{softmax}(\mathbf{h}_2)
\end{aligned}
$$

Batching

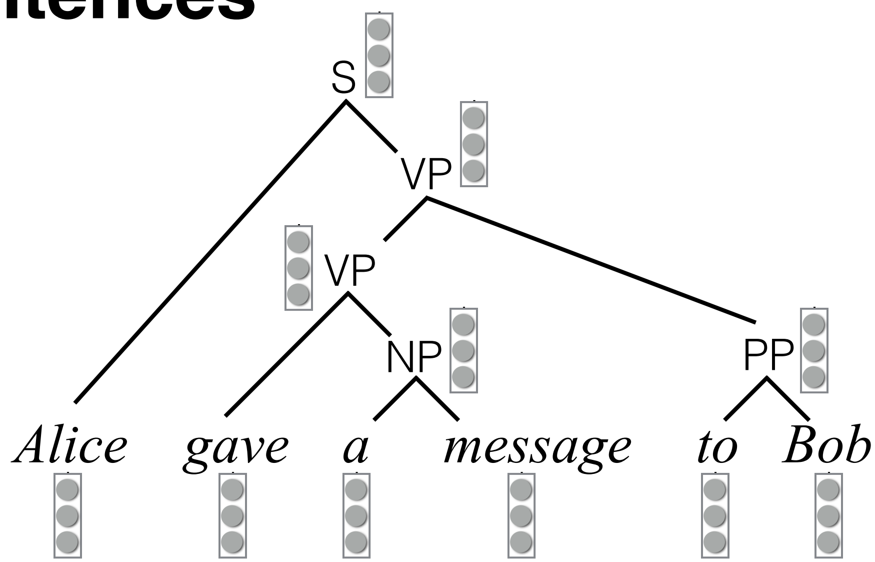
$$
\begin{aligned}
\mathbf{X}'^{(j)} &= [x'_1, \dots, x'_M], x'_i = \begin{cases} x_i & i \leq n^{(j)} \\ 0 & \text{else} \end{cases} \\
\mathbf{B} &= [\mathbf{X}'^{(j)}, \dots, \mathbf{X}'^{(j+B)}] \\
\mathbf{a} &= [\frac{1}{n^{(j)}}, \dots, \frac{1}{n^{(j+B)}}] \text{sum}\left(\phi(\mathbf{B})\right) \\
\mathbf{h}_1 &= \sigma(\mathbf{W}_1 \mathbf{a} + \mathbf{b}_1) \\
\mathbf{h}_2 &= \mathbf{W}_2 \mathbf{h}_1 + \mathbf{b}_2 \\
p &= \text{softmax}(\mathbf{h}_2)
\end{aligned}
$$

Hierarchical Structure

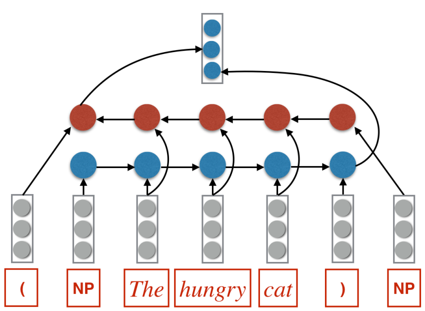
Words



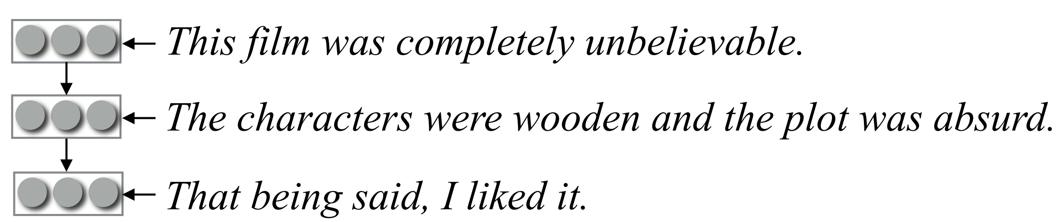
Sentences



Phrases



Documents



Batching with Complex Networks

- Complex networks may include different parts with varying length (more about this later)
- It is complex to batch complete examples this way
- But: you can still batch sub-parts across examples, so you alternate between batched and non-batched computations