Warming Up

Neural Network Basics

Cornell CS 5740: Natural Language Processing
Yoav Artzi, Spring 2023

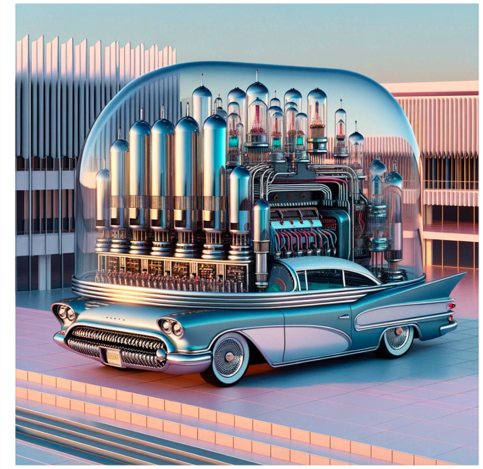
Table of Contents

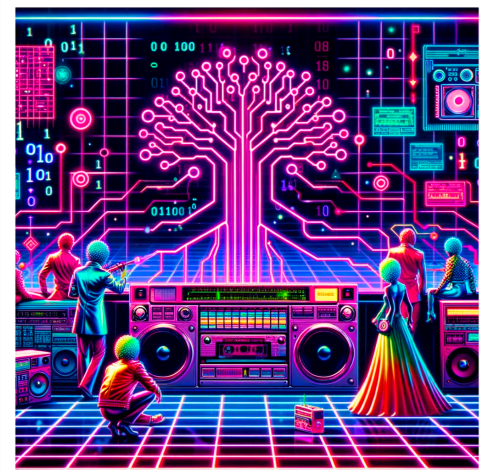
- A very quick introduction to neural networks
- Architecture basics and matrix notation
- Some practical tips
- Computation graphs

Neural Networks

A Little Bit of History

- Neural network algorithms date to the 1980s, and design trace their origin to the 1950s
    - Originally inspired by early neuroscience
- Historically slow, complex, and unwieldy
- Now: term is abstract enough to encompass almost any model – but useful!
- Dramatic shift started around 2013-15 away from MaxEnt (linear, convex) to *neural networks* (non-linear architecture, non-convex)





Neural Networks

The Promise

- Non-neural ML works well because of human-designed representations and input features

- ML becomes just optimizing weights

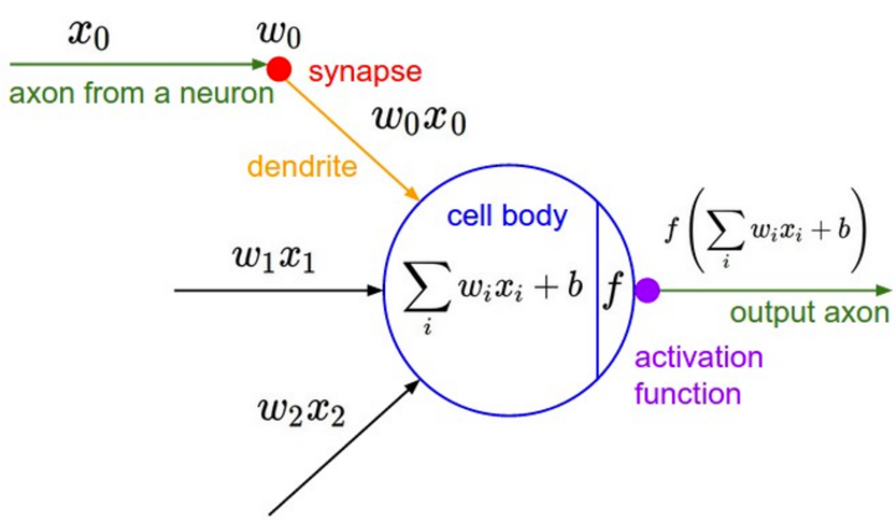
- **Representation learning** attempts to automatically learn good features and representations

- **Deep learning** attempts to learn multiple levels of representation of increasing complexity/abstraction

Building Blocks

The Neuron

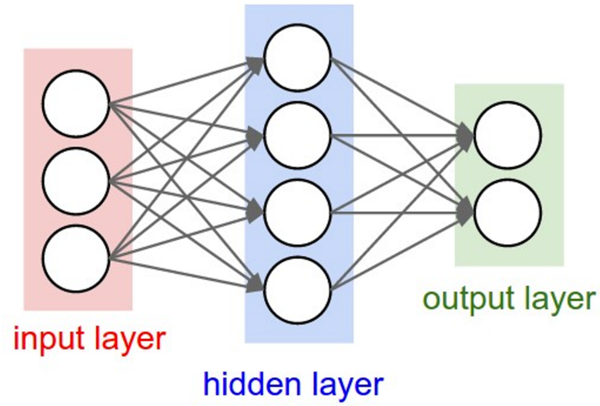
- Neural networks traditionally come with their own terminology baggage
  - Some of it is less common in more recent work
- Parameters:
  - Inputs: x_i
  - Weights: w_i and b
  - Activation function f
- If we drop the activation function, reminds you of something?

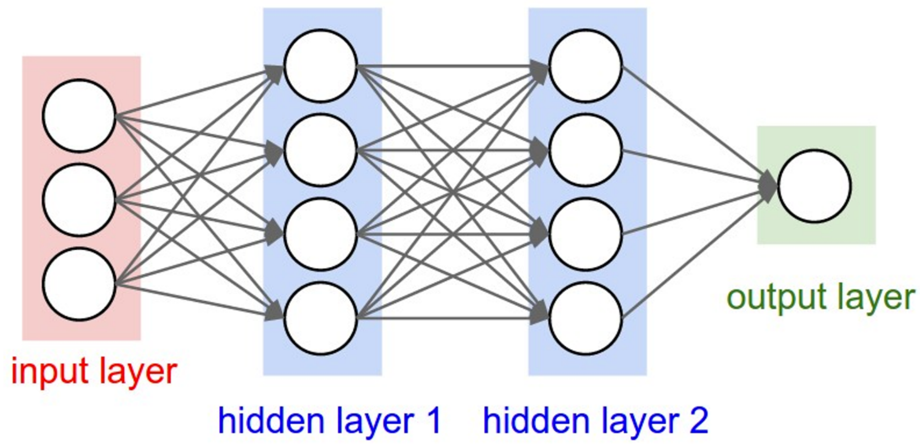


Building Blocks

Hidden Layers

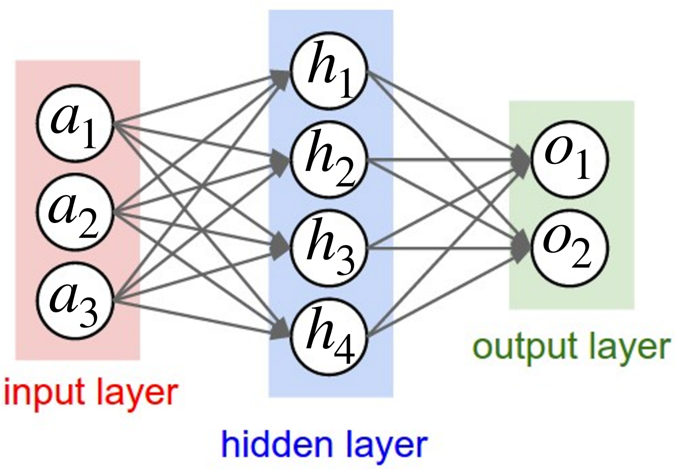
- It gets interesting when you connect and stack neurons
- This modularity is one of the greatest strengths of neural networks
- Input vs. hidden vs. output layers
- The activations of the hidden layers are the learned representation





Building Blocks

Matrix Notation

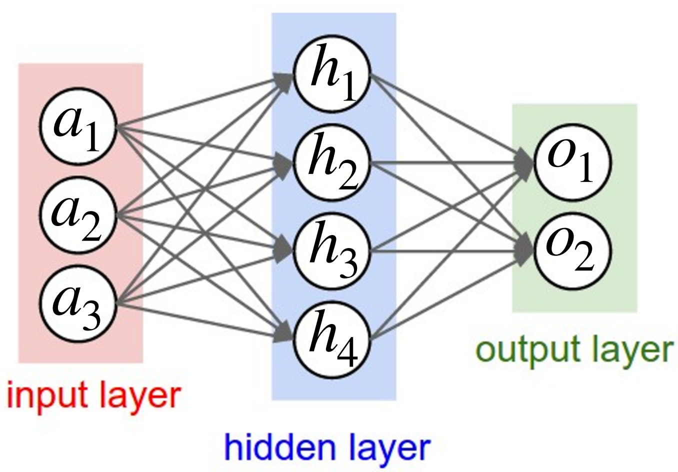


No activation/non-linearity function

Building Blocks

Matrix Notation

$$h_1 = a_1W'_{11} + a_2W'_{21} + a_3W'_{31} + b'_1$$
$$h_2 = a_1W'_{12} + a_2W'_{22} + a_3W'_{32} + b'_1$$
$$h_3 = a_1W'_{13} + a_2W'_{23} + a_3W'_{33} + b'_1$$
$$h_4 = a_1W'_{14} + a_2W'_{24} + a_3W'_{34} + b'_4$$



$$o_1 = h_1W''_{11} + h_2W''_{21} + h_3W''_{31} + h_4W''_{41} + b''_1$$
$$o_2 = h_1W''_{12} + h_2W''_{22} + h_3W''_{32} + h_4W''_{42} + b''_2$$

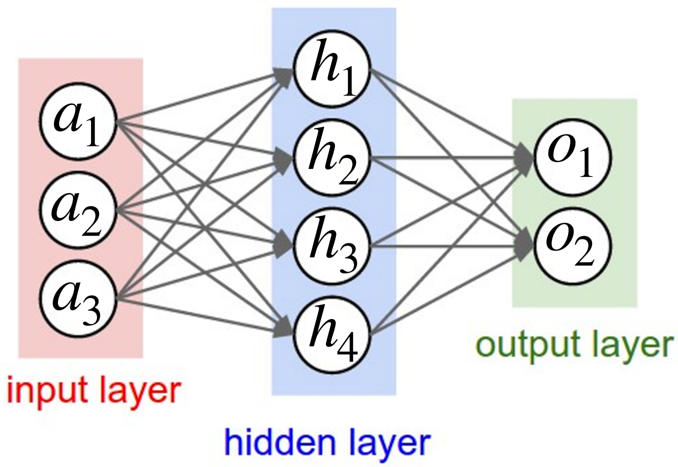
No activation/non-linearity function

Building Blocks

Matrix Notation

$$h_1 = a_1 W'_{11} + a_2 W'_{21} + a_3 W'_{31} + b'_1$$
$$h_2 = a_1 W'_{12} + a_2 W'_{22} + a_3 W'_{32} + b'_1$$
$$h_3 = a_1 W'_{13} + a_2 W'_{23} + a_3 W'_{33} + b'_1$$
$$h_4 = a_1 W'_{14} + a_2 W'_{24} + a_3 W'_{34} + b'_4$$

$$\mathbf{h} = \mathbf{a}\mathbf{W}' + \mathbf{b}'$$
$$\begin{aligned} \mathbf{o} &= \mathbf{h}\mathbf{W}'' + \mathbf{b}'' \\ &= (\mathbf{a}\mathbf{W}' + \mathbf{b}')\mathbf{W}'' + \mathbf{b}'' \end{aligned}$$



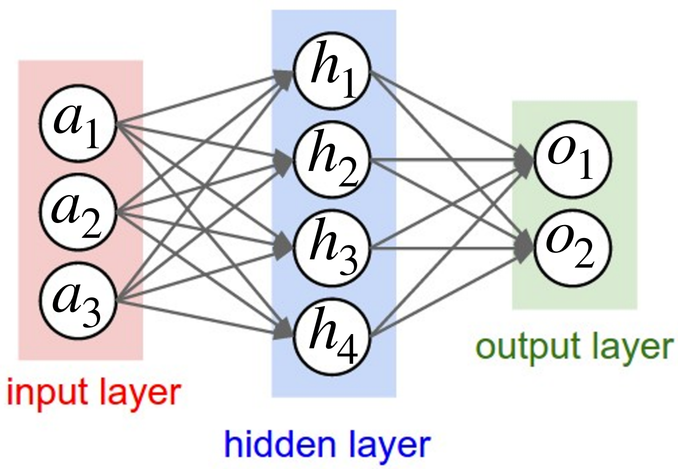
$$o_1 = h_1 W''_{11} + h_2 W''_{21} + h_3 W''_{31} + h_4 W''_{41} + b''_1$$
$$o_2 = h_1 W''_{12} + h_2 W''_{22} + h_3 W''_{32} + h_4 W''_{42} + b''_2$$

No activation/non-linearity function

Building Blocks

Matrix Notation

$$h_1 = a_1 W'_{11} + a_2 W'_{21} + a_3 W'_{31} + b'_1$$
$$h_2 = a_1 W'_{12} + a_2 W'_{22} + a_3 W'_{32} + b'_1$$
$$h_3 = a_1 W'_{13} + a_2 W'_{23} + a_3 W'_{33} + b'_1$$
$$h_4 = a_1 W'_{14} + a_2 W'_{24} + a_3 W'_{34} + b'_4$$



$$o_1 = h_1 W''_{11} + h_2 W''_{21} + h_3 W''_{31} + h_4 W''_{41} + b''_1$$
$$o_2 = h_1 W''_{12} + h_2 W''_{22} + h_3 W''_{32} + h_4 W''_{42} + b''_2$$

$$\mathbf{h} = \mathbf{a}\mathbf{W}' + \mathbf{b}'$$
$$\mathbf{o} = \mathbf{h}\mathbf{W}'' + \mathbf{b}''$$
$$= (\mathbf{a}\mathbf{W}' + \mathbf{b}')\mathbf{W}'' + \mathbf{b}''$$

$\mathbf{a} \in \mathbb{R}^{1\times 3}$

$\mathbf{W}' \in \mathbb{R}^{3\times 4}$

$\mathbf{W}'' \in \mathbb{R}^{4\times 2}$

$\mathbf{b}' \in \mathbb{R}^{1\times 4}$

$\mathbf{b}'' \in \mathbb{R}^{1\times 2}$

Learned

$\mathbf{h} \in \mathbb{R}^{1\times 4}$

$\mathbf{o} \in \mathbb{R}^{1\times 2}$

No activation/non-linearity function

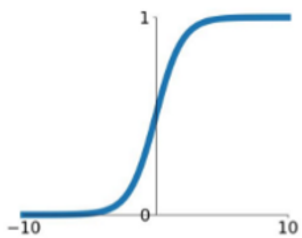
Building Blocks

Activation Functions

Activation (non-linearity) function is an entry-wise function $f: \mathbb{R} \to \mathbb{R}$

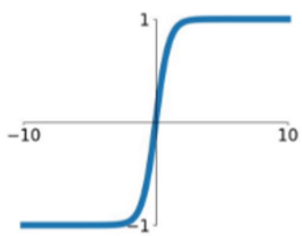
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



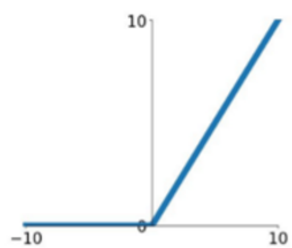
tanh

$\tanh(x)$



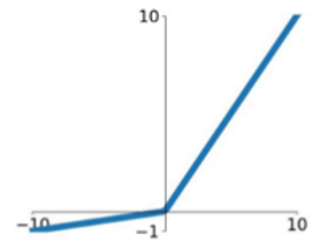
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

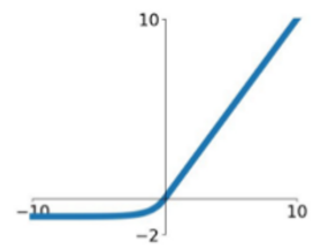


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Building Blocks

Probabilistic Outputs

- What if we want the output to be a probability distribution over possible outputs?
  - So far: output are just real numbers
- Normalize the output activations $\mathbf{o}$ using $\mathrm{softmax}$
- Assume your want a distribution over $y_1, \ldots, y_n$ (i.e., $p(y_i)$)

$$\mathbf{o} = \begin{pmatrix} o_1 \\ o_2 \\ \vdots \\ o_n \end{pmatrix} \qquad y = \mathrm{softmax}(\mathbf{o})$$

$$p(y_i) = \mathrm{softmax}(\mathrm{o_i}) = \frac{\mathrm{e}^{\mathrm{o_i}}}{\sum_{\mathrm{j=1}}^{\mathrm{n}} \mathrm{e}^{\mathrm{o_j}}}$$

- Essentially: (1) make the value positive; and (2) normalize
- Usually: no non-linearity before the $\mathrm{softmax}$

Building Blocks

One-hot Word Representations

- So far, words (and features) are atomic symbols:
  - “hotel”, “conference”, “walking”, “___ing”
- But neural networks take continuous vector inputs
- How can we bridge this gap?
- One-hot vectors

$$\begin{aligned}\text{hotel} &= [0 \quad 0 \quad 0 \quad \cdots 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0] \\ \text{conference} &= [0 \quad 0 \quad 0 \quad \cdots 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0]\end{aligned}$$

- Dimensionality: size of the vocabulary
  - Can be >10M for web-scale corpora
- Problems?

Building Blocks

One-hot Word Representations

- One-hot vectors

$$\text{hotel} = [0 \quad 0 \quad 0 \quad \cdots 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$$
$$\text{conference} = [0 \quad 0 \quad 0 \quad \cdots 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0]$$

- Problems?
  - Information sharing? “hotel” vs. “hotels”

Building Blocks

Word Embeddings

- Each word is represented using a dense low-dimensional vector
  - Low-dimensional << vocabulary size
- If trained well, similar words will have similar vectors
- How to train? What objective to maximize?
  - As part of task training (e.g., supervised training)
  - Pre-training (more on this later)

Training Neural Networks

- No hidden layer → supervised
  - Just like perceptron, but gradient based
- With hidden layers:
  - Latent units → not convex
  - What do we do?
    - Back-propagate the gradient
    - Based on the chain rule
    - About the same, but no guarantees

Neural Bag of Words

- One of the most basic neural models
- Example: sentiment classification
  - Input: text document
  - Classes: very positive, positive, neutral, negative, very negative
- We discussed doing this with a bag-of-words feature-based model
- What would be the neural equivalent?

Neural Bag of Words

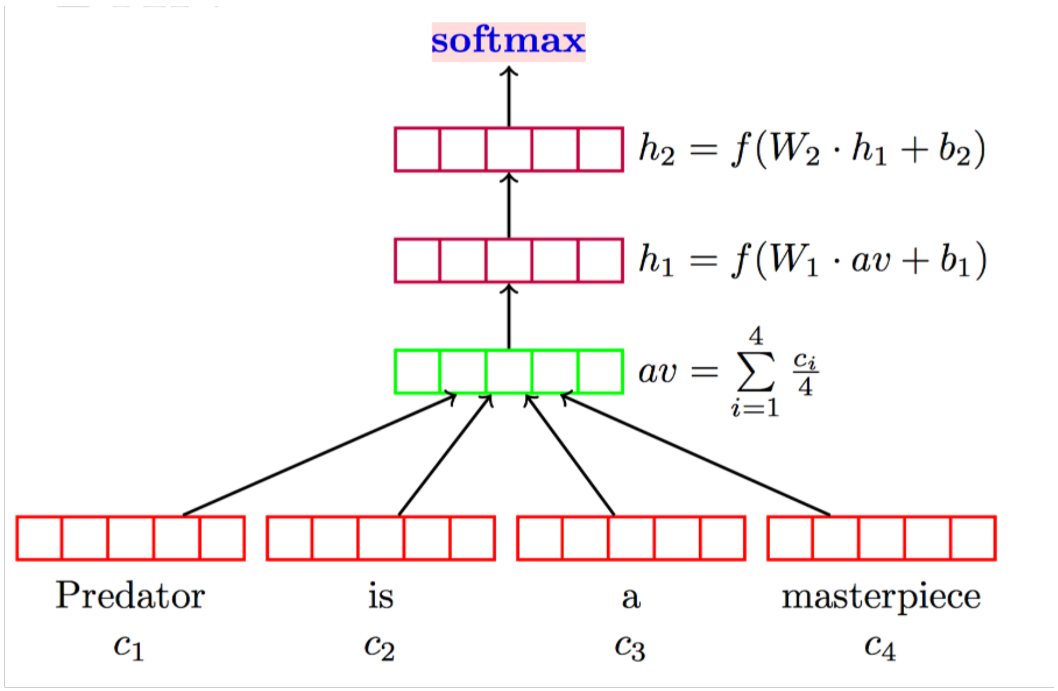
- One of the most basic neural models
- Example: sentiment classification
  - Input: text document
  - Classes: very positive, positive, neutral, negative, very negative
- We discussed doing this with a bag-of-words feature-based model
- What would be the neural equivalent?
  - Concatenate all vectors?

Neural Bag of Words

- One of the most basic neural models
- Example: sentiment classification
  - Input: text document
  - Classes: very positive, positive, neutral, negative, very negative
- We discussed doing this with a bag-of-words feature-based model
- What would be the neural equivalent?
  - Concatenate all vectors?
    - Problem: different documents → different input length
  - Instead: sum, average, etc.

Neural Bag of Words

Deep Averaging Networks (Iyyer et al. 2015)



IMDB Sentiment Analysis

BOW + smoothing + SVM	88.23
NBOW DAN	89.4

*It's not common to put non-linearity before a softmax

Classify Word Pair

- Goal: build a classifier that given a pair of words, classify if they are the full name of a person or not
- The classifier is a multi-layer-perceptron with three layers
- Make a drawing!
- Write the matrix notation, including dimensionality of matrices (choose as you wish, and as needed)
- What are the parameters to be learned

Inputs: x_l, x_r

Input vocabulary: $\mathcal{V}$

Embedding function: $\phi : \mathcal{V} \rightarrow \mathbb{R}^{256}$

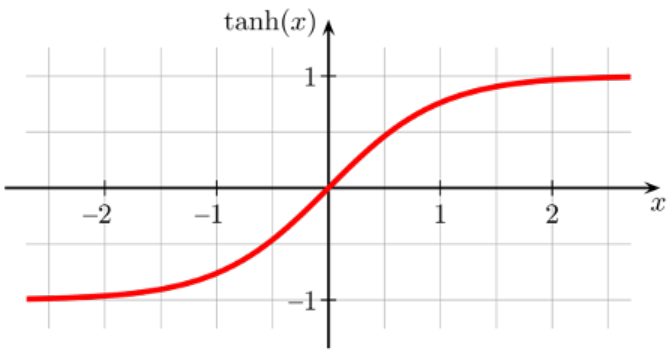
Weight matrices: $\mathbf{W}^1, \mathbf{W}^2, \mathbf{W}^3$

Bias vectors: $\mathbf{b}^1, \mathbf{b}^2, \mathbf{b}^3$

Operations: $2 \times \sigma : \mathbb{R}^* \rightarrow \mathbb{R}^*, 1 \times \mathrm{softmax}$

Practical Tips

- If you control the model (i.e., not using a pre-trained model)
  - Select network structure appropriate for the problem
    - Window vs. recurrent vs. recursive (will discuss throughout the semester)
  - Parameter initialization
  - Model is powerful enough?
    - If not, make it larger
    - Yes, so regularize, otherwise it will overfit
- Gradient checks to identify bugs
  - If you build from scratch
- Know your non-linearity function and its gradient
  - Example $\tanh(x)$
    - $\frac{\partial}{\partial x}\tanh(x) = 1 - \tanh^2(x)$



Practical Tips

Debugging

- Verify value of initial loss when using softmax
- Perfectly fit a single example, then mini-batch, then train
- If learning fails completely, maybe gradients stuck
  - Check learning rate
  - Verify parameter initialization
  - Change non-linearity functions

Practical Tips

Avoid Overfitting

- Very expressive models, can overfit easily
  - It will look great on the training data, but everything else will be terrible
- Some potential cures
  - Reduce model size (but not too much)
  - L1 and L2 regularization
  - Early stopping (e.g., patience)
  - Learning rate scheduling
  - Dropout (Hinton et al. 2012)
    - Randomly set 50% of inputs in each layer to 0

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

$$\mathbf{x}$$

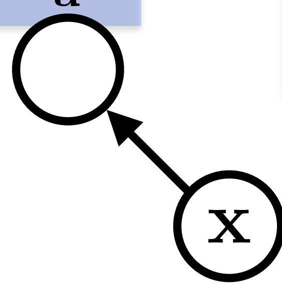
An **edge** represents a function argument (and also data dependency). They are just pointers to nodes.

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

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

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

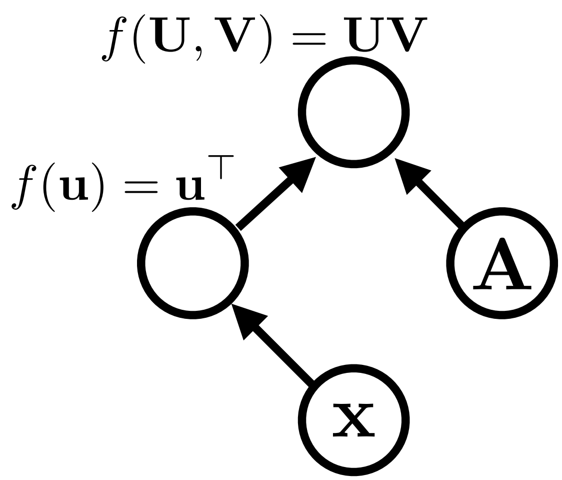


expression:

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

graph:

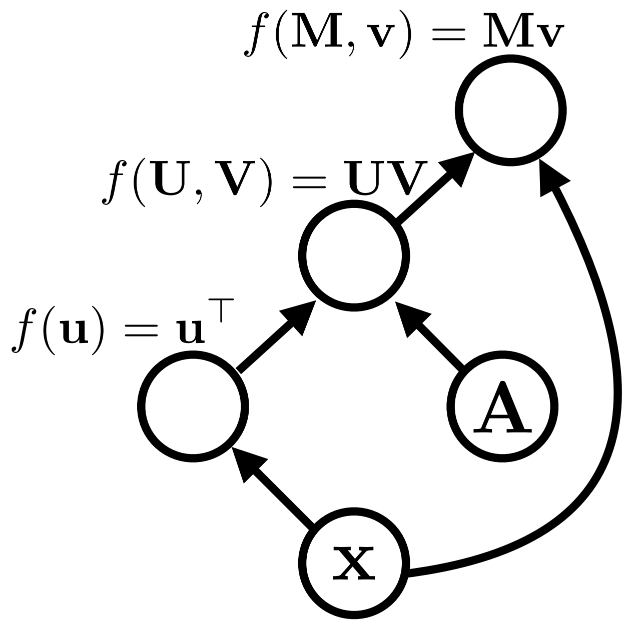
Functions can be nullary, unary, binary, … *n*-ary. Often they are unary or binary.



expression:

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

graph:

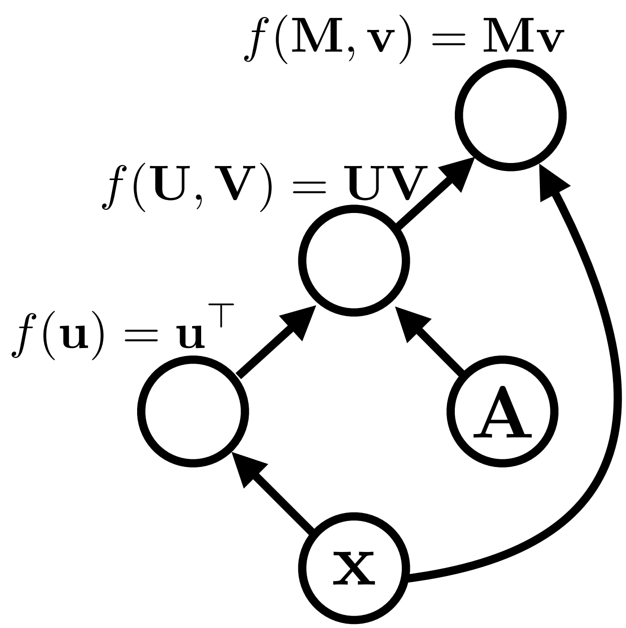


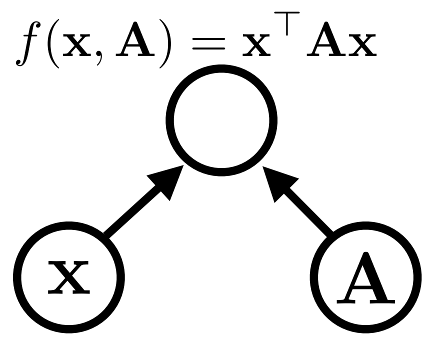
Computation graphs are directed and acyclic (usually)

expression:

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

graph:





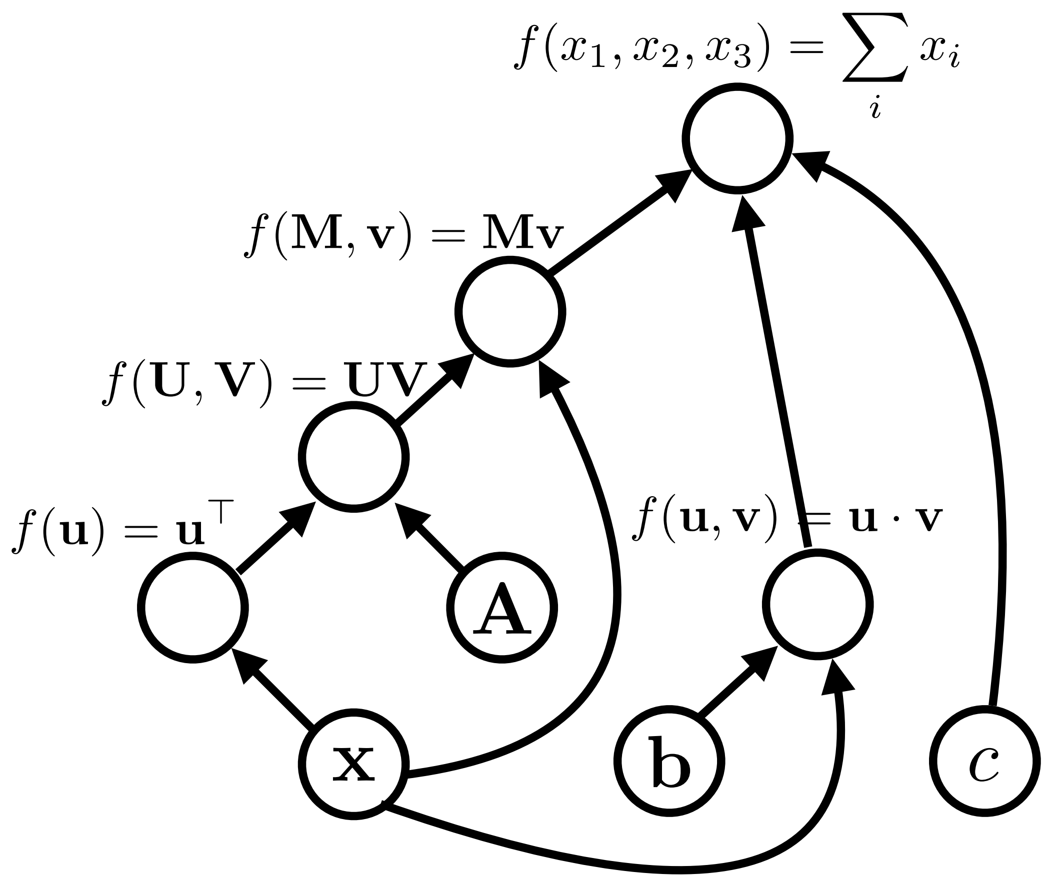
$$\frac{\partial f(\mathbf{x}, \mathbf{A})}{\partial \mathbf{x}} = (\mathbf{A}^\top + \mathbf{A})\mathbf{x}$$

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

expression:

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

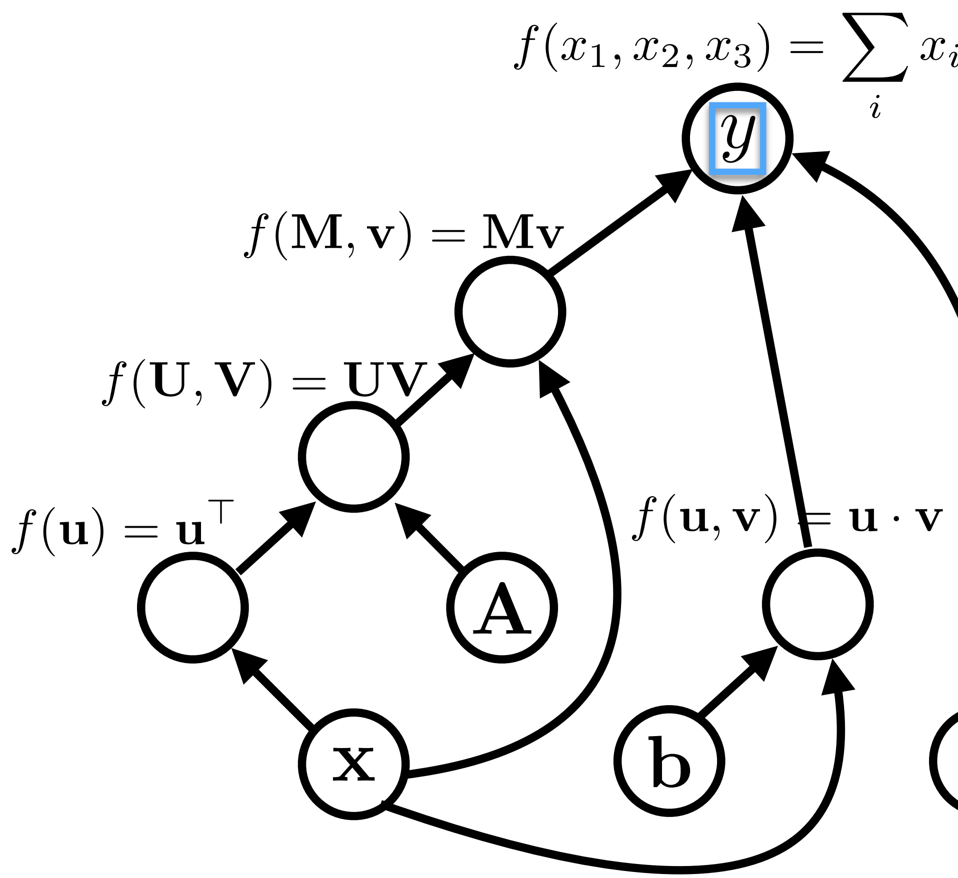
graph:



expression:

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

graph:



variable names are just labelings of nodes.

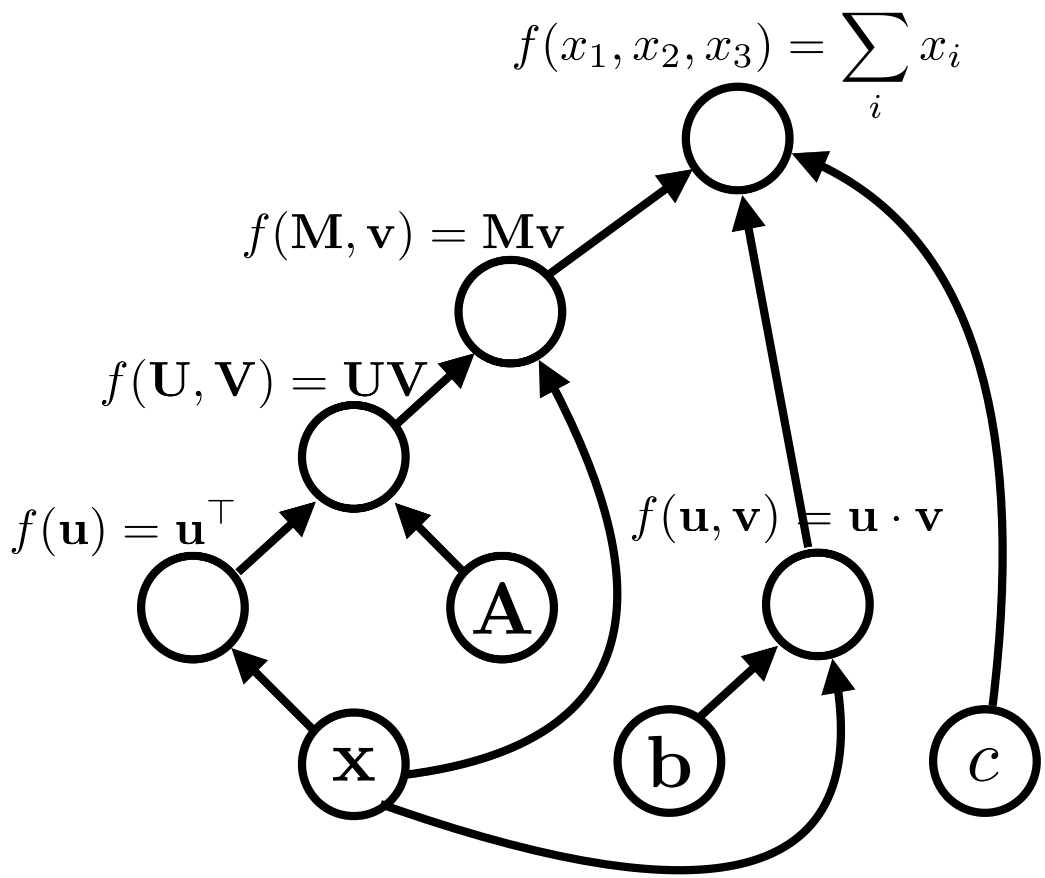
Computation Graphs

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?*

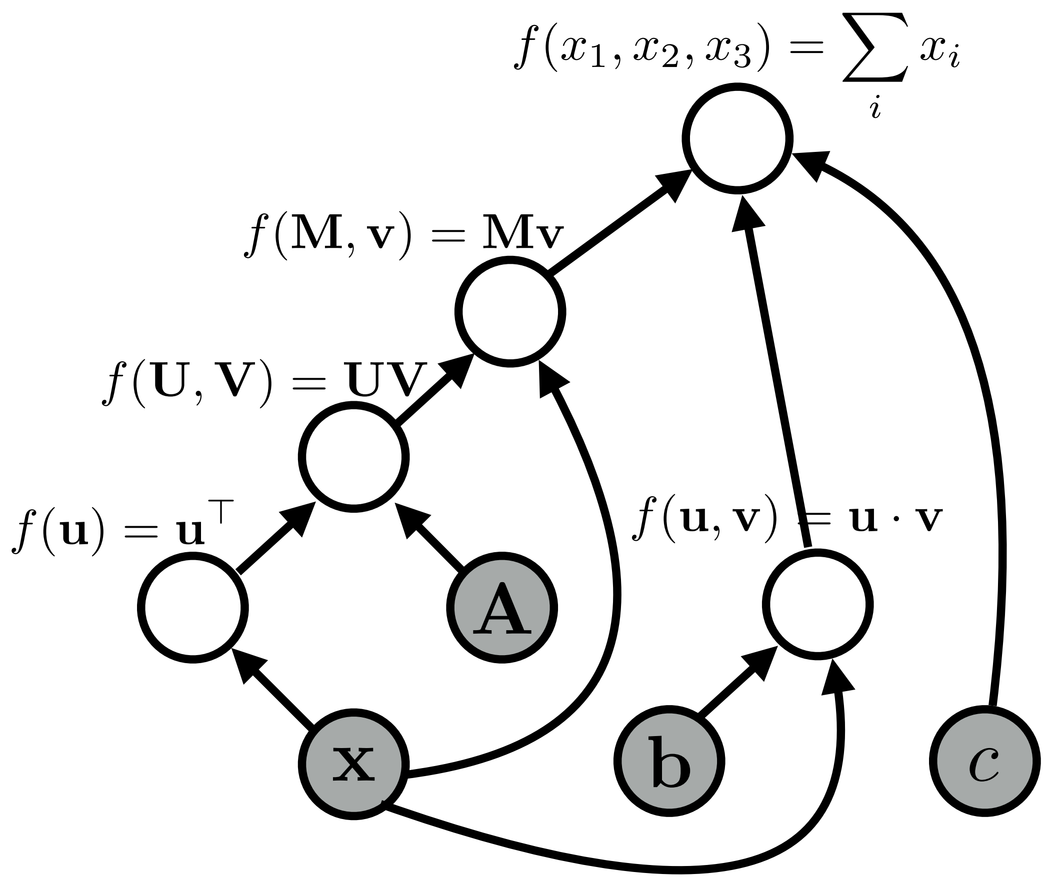
Forward Propagation

graph:



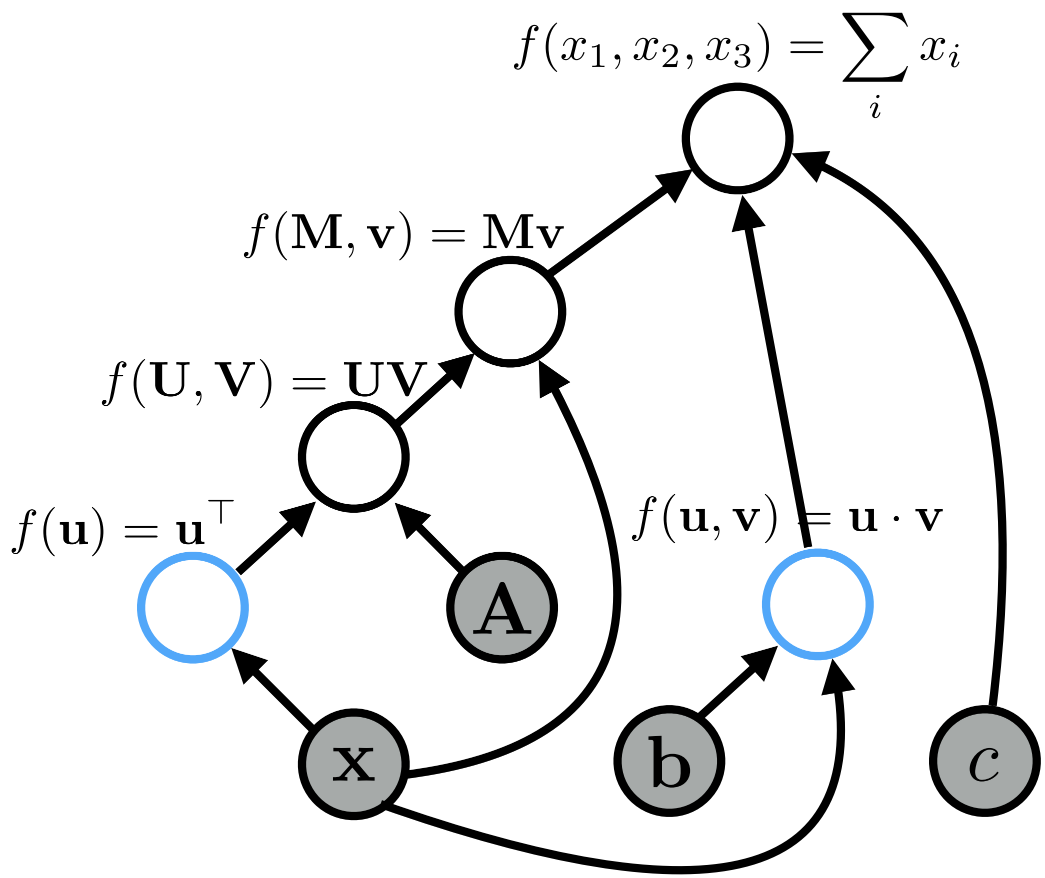
Forward Propagation

graph:



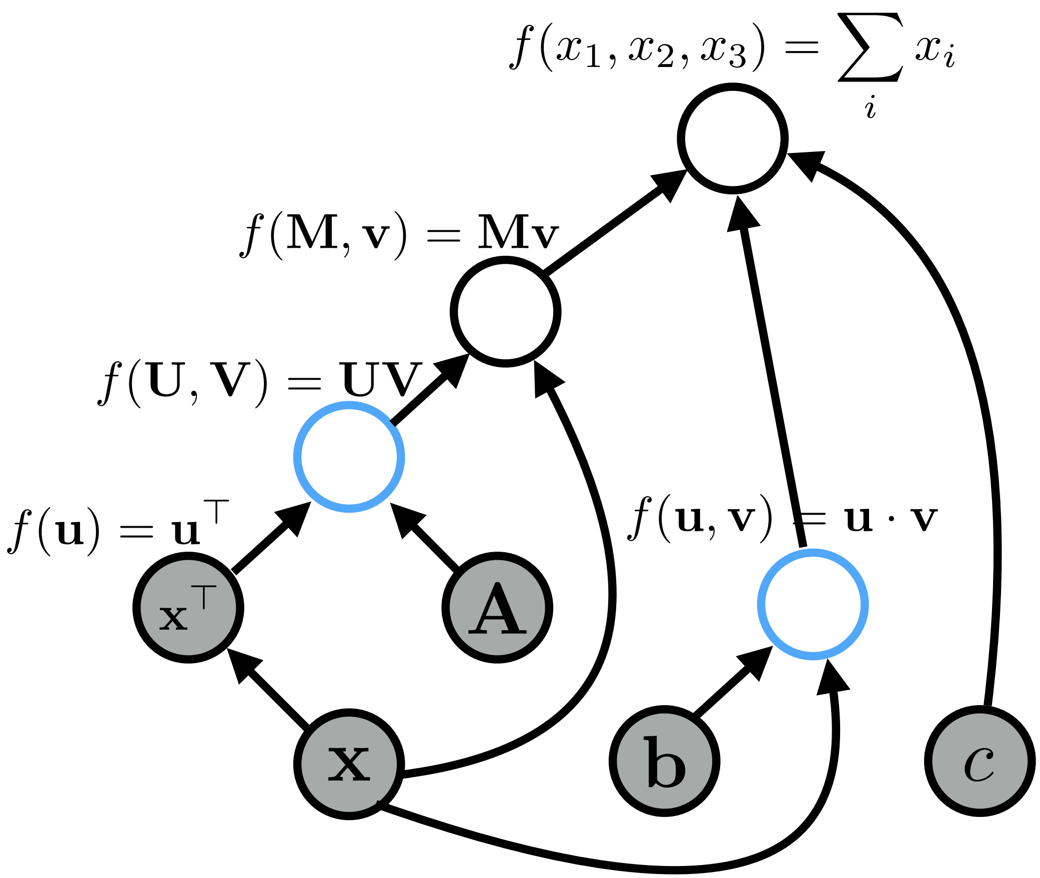
Forward Propagation

graph:



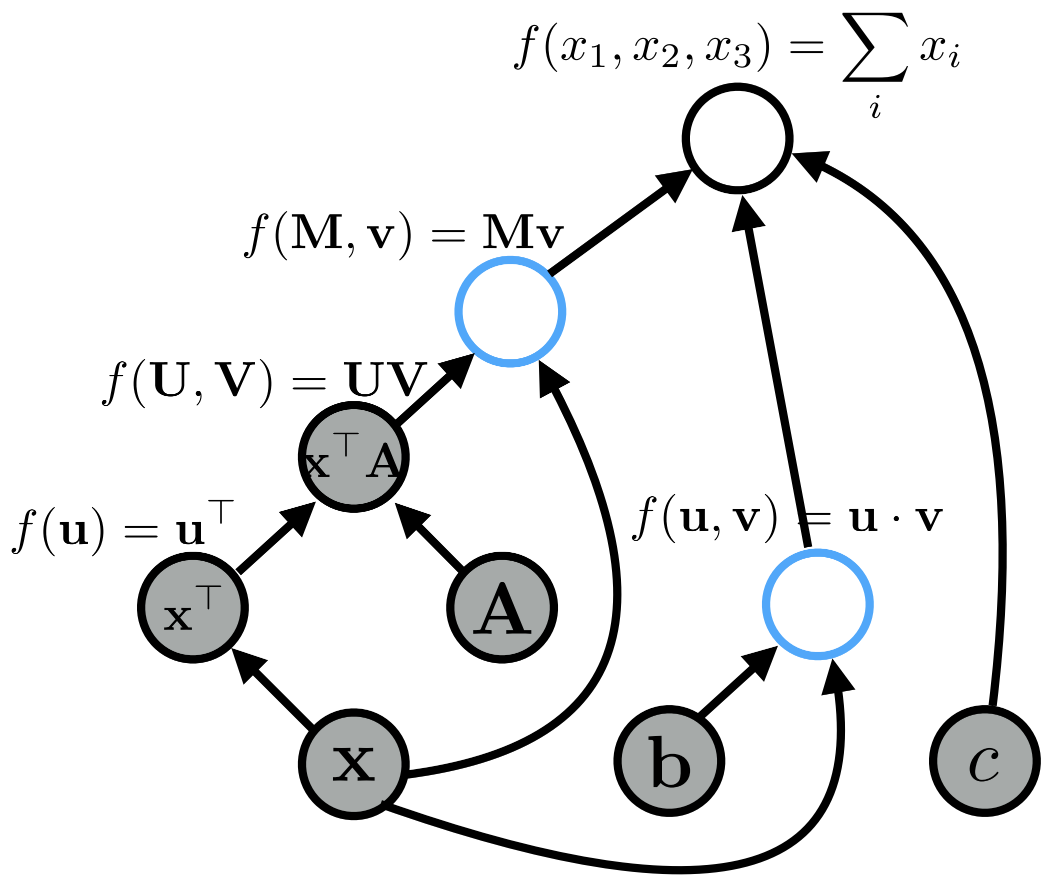
Forward Propagation

graph:



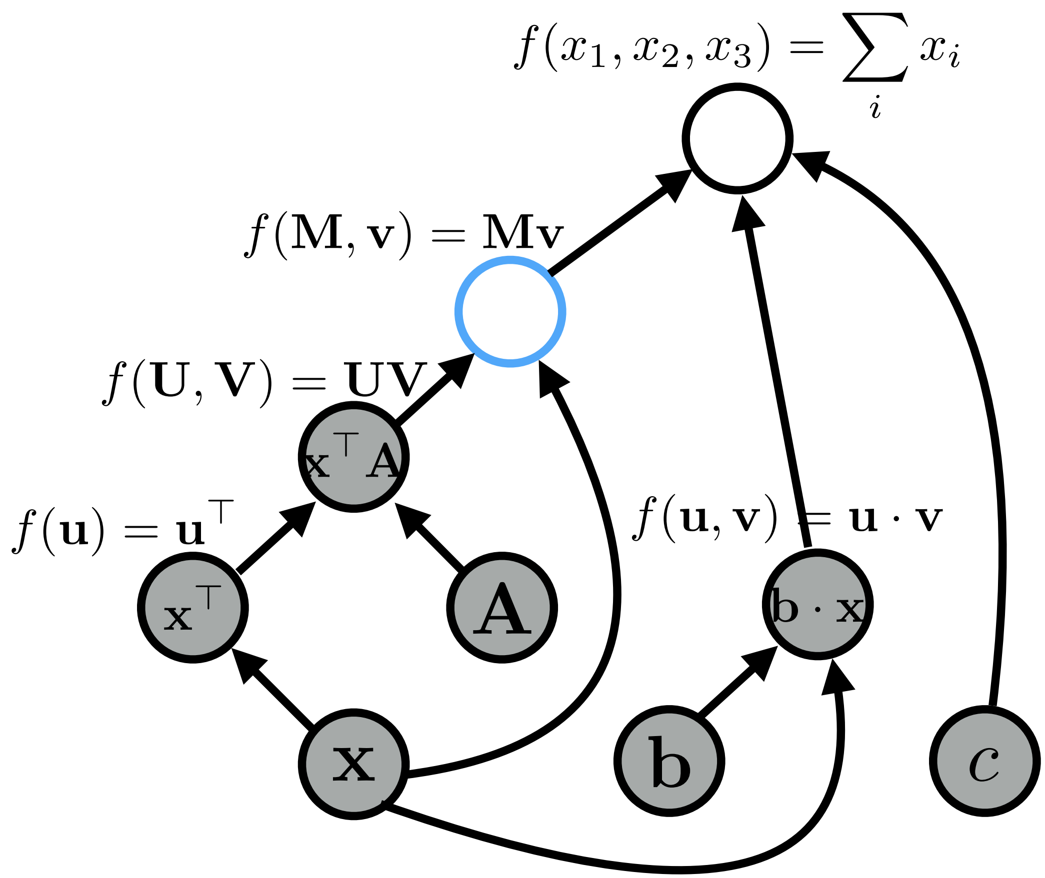
Forward Propagation

graph:



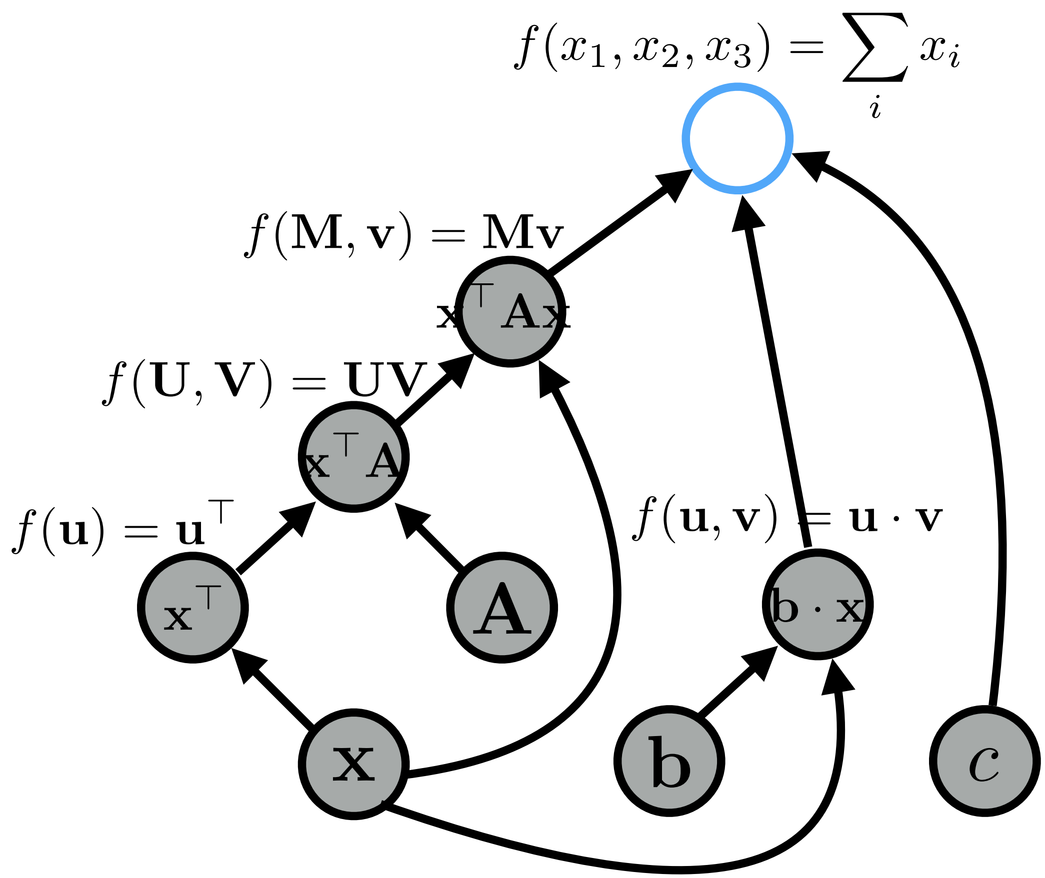
Forward Propagation

graph:



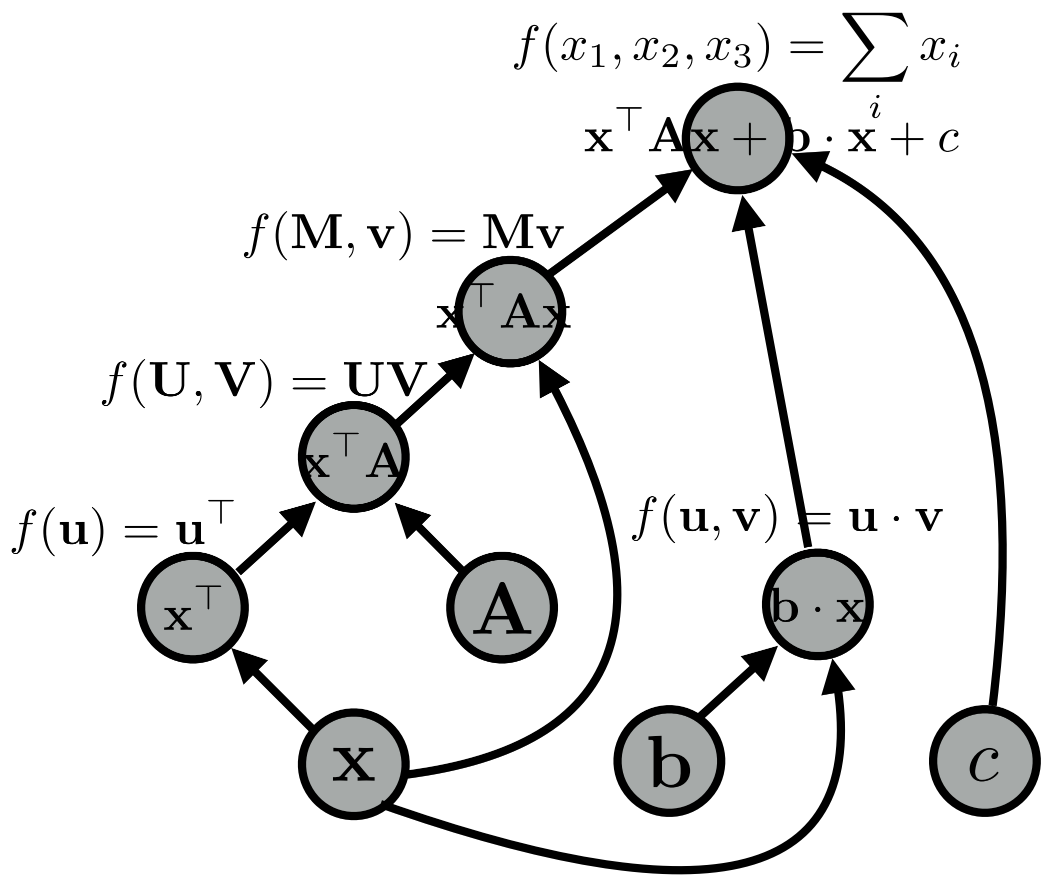
Forward Propagation

graph:



Forward Propagation

graph:



MLP

Draw the 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 (a.k.a define-by-run)
  - Graph is defined implicitly (e.g., using operator overloading) as the forward computation is executed
  - Graph is constructed dynamically
  - This allows incorporating conditionals and loops into the network definitions easily

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
- 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

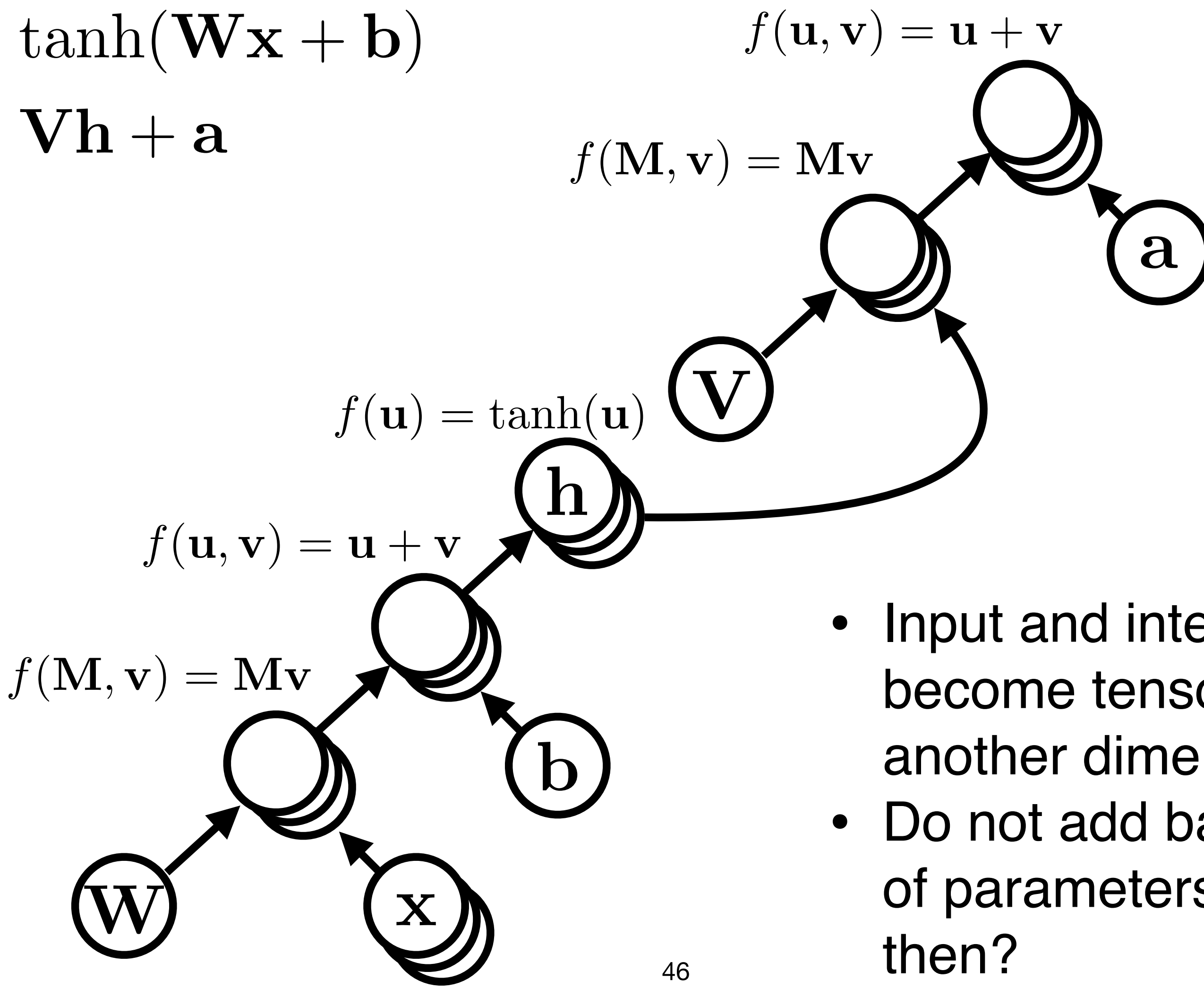
Batching

- 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

Batching

MLP Sketch

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



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

Batching

Rough Notation Sketch

No batching

$$\mathbf{X}^{(j)} = [x_1, \ldots, x_{n^{(j)}}], x_i \in 1, \ldots, |\mathscr{V}|$$

$$\mathbf{a} = \frac{1}{|\mathbf{X}^{(j)}|}\text{sum}\left(\phi(\mathbf{X}^{(\text{j})})\right)$$

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

$$\mathbf{h}_2 = \mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2$$

$$p = \text{softmax}(\mathbf{h}_2)$$

Batching

$$\mathbf{X}'^{(j)} = [x'_1, \ldots, x'_M], x'_i = \begin{cases} x_i & i \leq n^{(j)} \\ 0 & \text{else} \end{cases}$$

$$\mathbf{B} = [\mathbf{X}'^{(j)}, \ldots, \mathbf{X}'^{(j+B)}]$$

$$\mathbf{a} = [\frac{1}{n^{(j)}}, \ldots, \frac{1}{n^{(j+B)}}]\text{sum}\left(\phi(\mathbf{B})\right)$$

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

$$\mathbf{h}_2 = \mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2$$

$$p = \text{softmax}(\mathbf{h}_2)$$

Not accurate notation, for illustration only

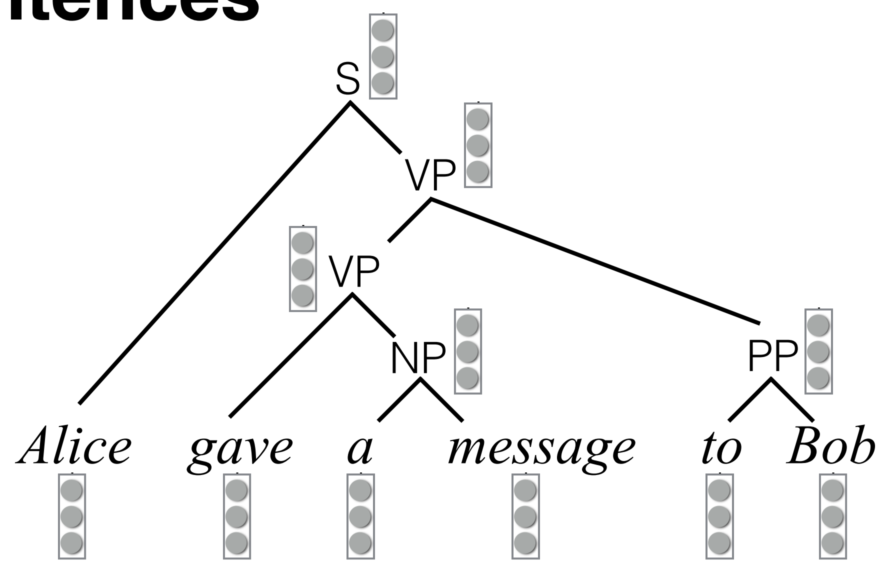
- You have to get certain operations right, such as sum
- But PyTorch's broadcasting sorts out most operations

Batching

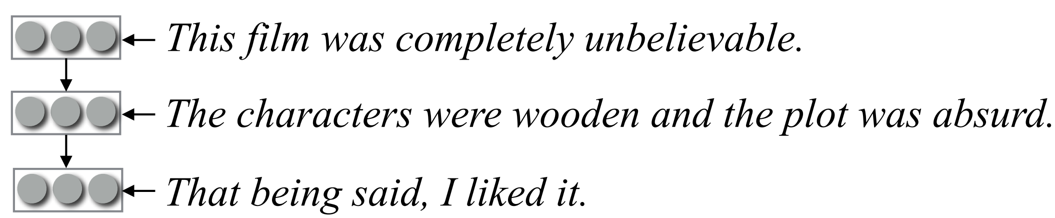
Complex Network Architectures

- Complex networks may include different parts with varying length (more about this later)
- In the extreme, it may be 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

Sentences



Documents



Acknowledgements

We thank the following sources for presentation materials:

- University of Washington CSE 517 by Luke Zettlemoyer
- Berkeley CS 288 by Alane Suhr and Dan Klein (and older versions of the class by Dan Klein)
- Stanford CS 124 by Dan Jurafsky

Computation graph slides were adapted from Practical Neural Networks for NLP / Chris Dyer, Yoav Goldberg, Graham Neubig / EMNLP 2016