CMPT 413/825: Natural Language Processing

Neural Network Basics

Fall 2020
2020-10-07

Adapted from slides from Danqi Chen and Karthik Narasimhan

Announcements

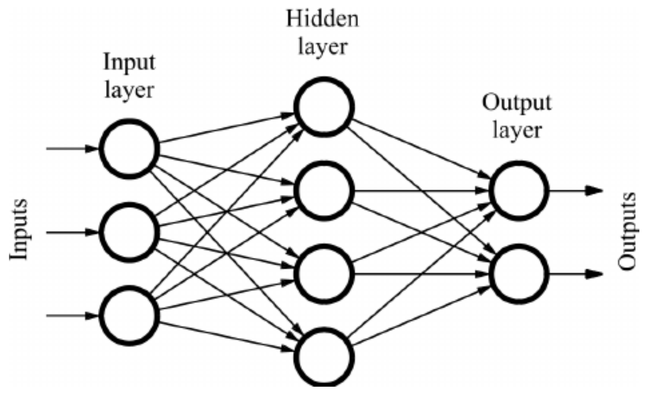
- Short list of some project ideas
  - Think about what you would like your project to be on!
- HW1 grades is out
- HW2 due next week (10/14)
- Video lectures on word vectors (SVD/GloVe) is still being prepared
- Tutorial on pytorch/backpropagation

Topics

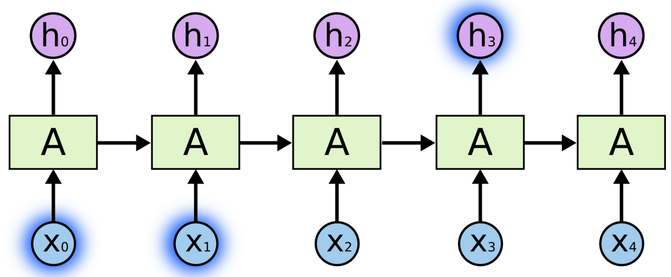
- So far
  - Language Modeling: $P(w_i \mid w_{1:i-1})$
  - Text classification: $P(c \mid d)$
  - Word embeddings: Representing w as a vector
- This week
  - Neural networks for NLP
- Upcoming
  - Sequence modeling
  - Sequence-to-sequence models for text generation
  - Structured prediction (parsing)

Neural networks for NLP

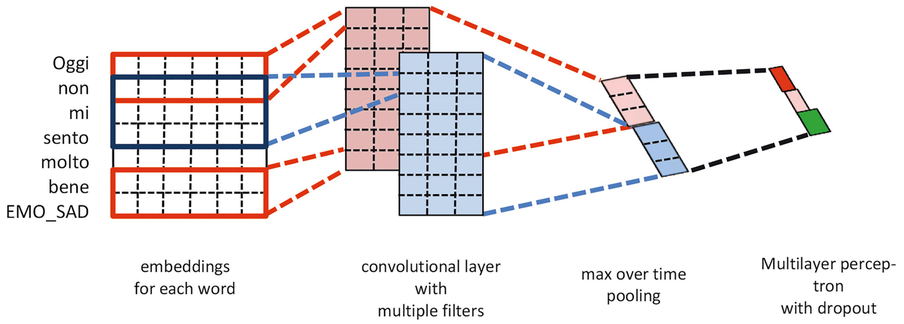
Feed-forward NNs



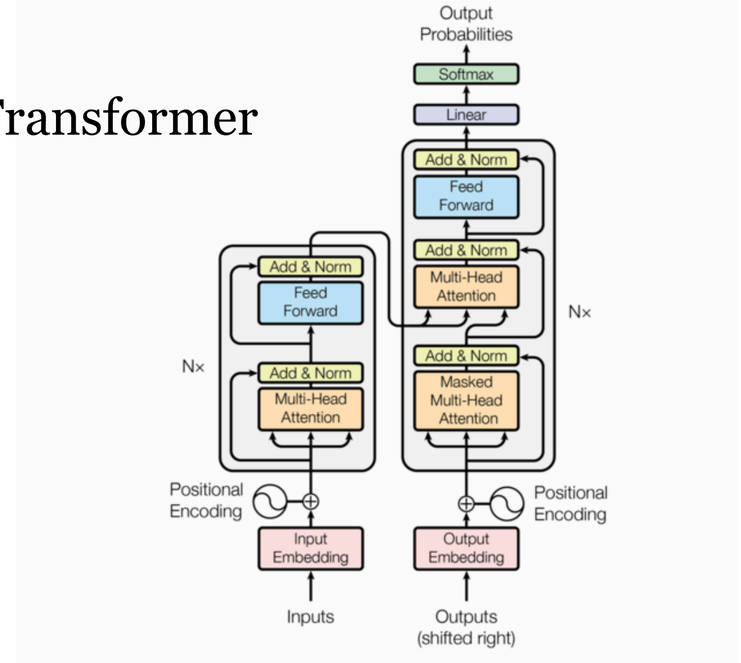
Recurrent NNs



Convolutional NNs



Transformer



Always coupled with word embeddings…

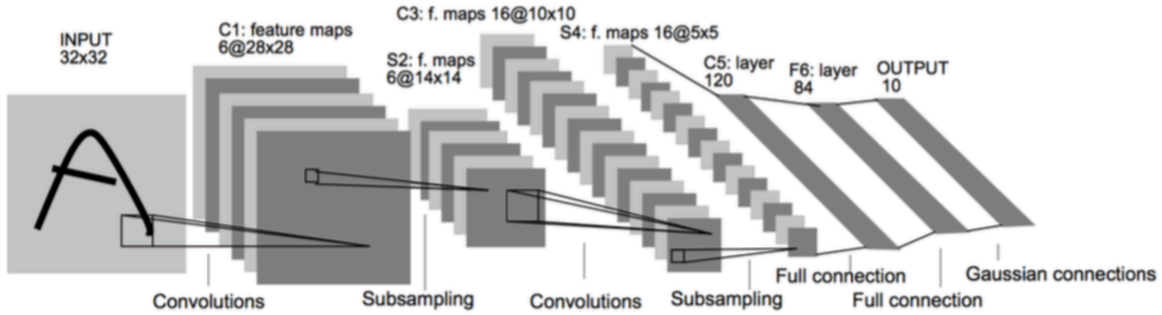
This Lecture

- Feedforward Neural Networks
- Applications
  - Neural Bag-of-Words Models
  - Feedforward Neural Language Models
- The training algorithm: Back-propagation

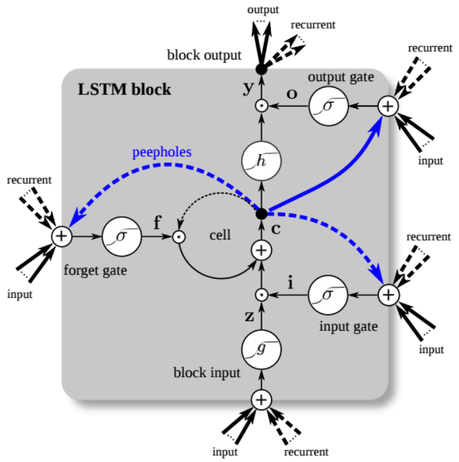
Neural Networks: History

NN "dark ages"

- Rosenblatt's Perceptron (1958)
  - Minsky and Papert (1969) - perceptrons are severely limited
- Neural network algorithms (including backpropagation) date from the 80s
- ConvNets: applied to MNIST by LeCun in 1998



- Long Short-term Memory Networks (LSTMs): Hochreiter and Schmidhuber 1997



- Henderson 2003: neural shift-reduce parser, not SOTA

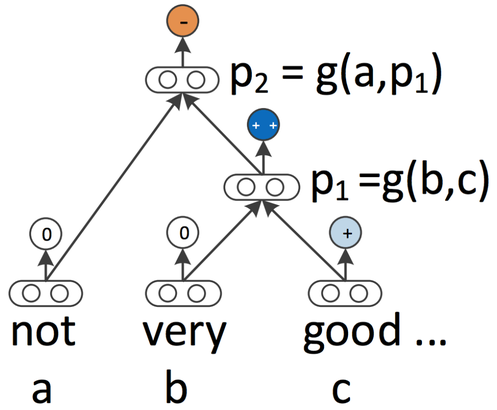
Credits: Greg Durrett

2008-2013: A glimmer of light

- Collobert and Weston 2011: "**NLP (almost) from Scratch**"
  - Feedforward NNs can replace "feature engineering"
  - 2008 version was marred by bad experiments, claimed SOTA but wasn't, 2011 version tied SOTA

- Krizhevskey et al, 2012: AlexNet for ImageNet Classification

- Socher 2011-2014: tree-structured RNNs working okay



Credits: Greg Durrett

2014: Stuff starts working

- Kim (2014) + Kalchbrenner et al, 2014: sentence classification
  - ConvNets work for NLP!

- Sutskever et al, 2014: sequence-to-sequence for neural MT
  - LSTMs work for NLP!

- Chen and Manning 2014: dependency parsing
  - Even feedforward networks work well for NLP!

- 2015: explosion of neural networks for everything under the sun

Credits: Greg Durrett

Why didn't they work before?

- **Datasets too small**: for MT, not really better until you have 1M+ parallel sentences (and really need a lot more)

- **Optimization not well understood**: good initialization, per-feature scaling + momentum (Adagrad/Adam) work best out-of-the-box
  - Regularization: dropout is pretty helpful
  - Computers not big enough: can't run for enough iterations

- Inputs: need **word embeddings** to represent continuous semantics

Credits: Greg Durrett

The “Promise”

- Most NLP works in the past focused on human-designed representations and input features

Var	Definition	Value in Fig. 5.2
x_1	count(positive lexicon) $\in$ doc)	3
x_2	count(negative lexicon) $\in$ doc)	2
x_3	$\begin{cases} 1 & \text{if “no” } \in \text{ doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if “!” } \in \text{ doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(64) = 4.15$

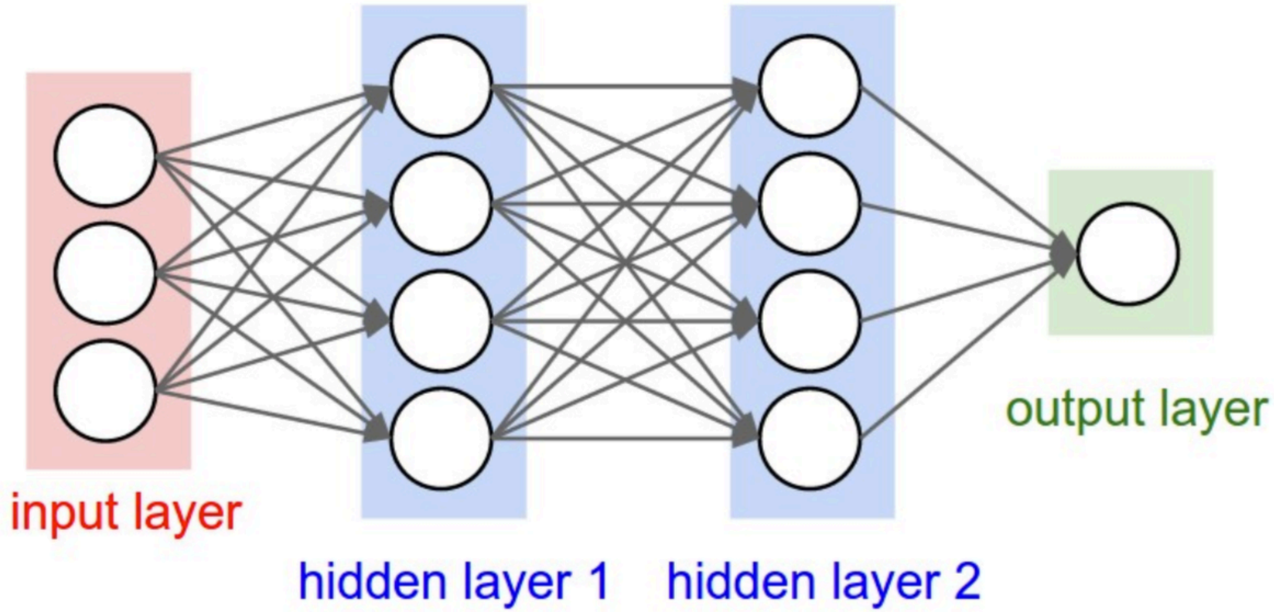
- **Representation learning** attempts to automatically learn good features and representations

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

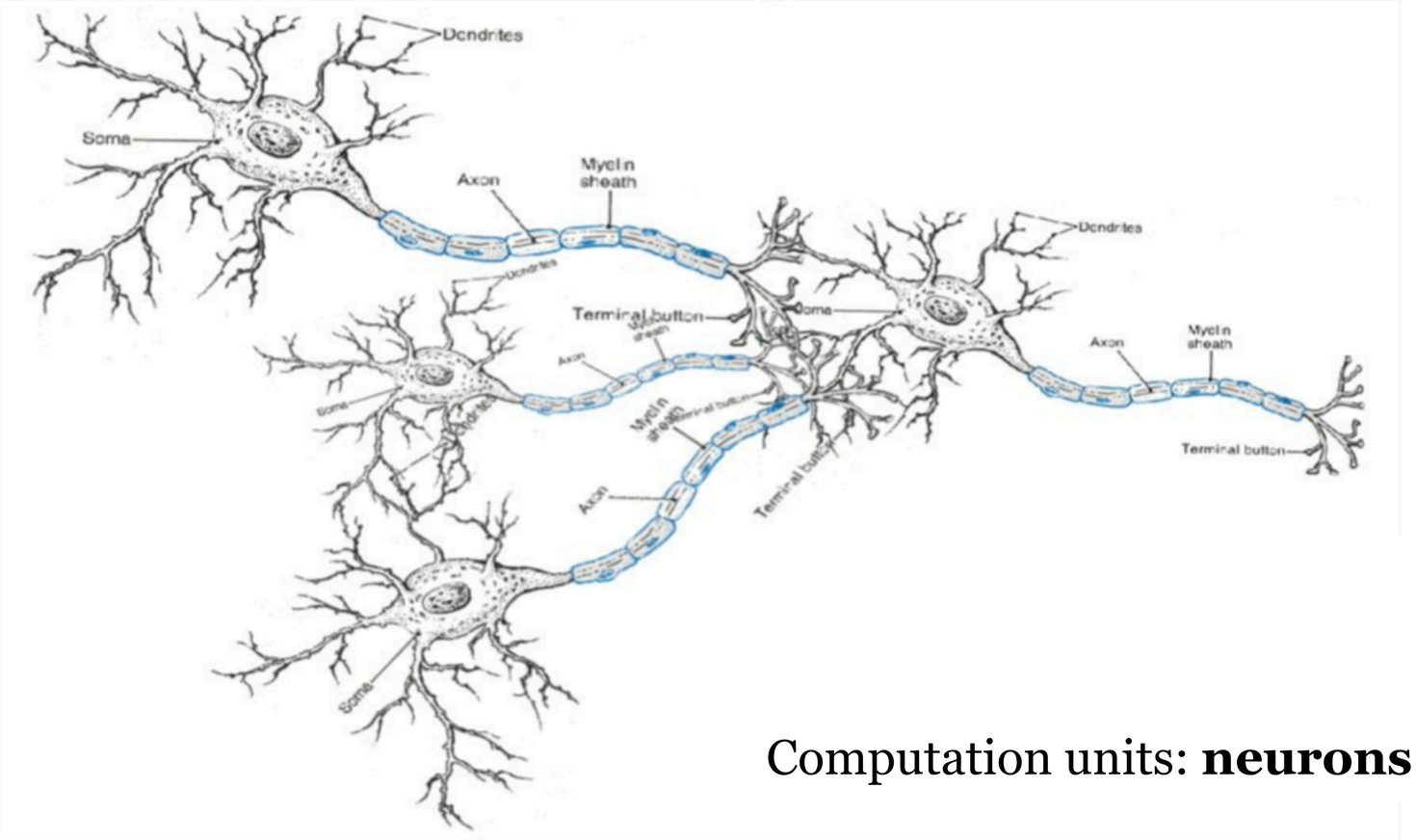
Feed-forward Neural Networks

Feed-forward NNs

- Input: $x_1, \ldots, x_d$
- Output: $y \in \{0,1\}$



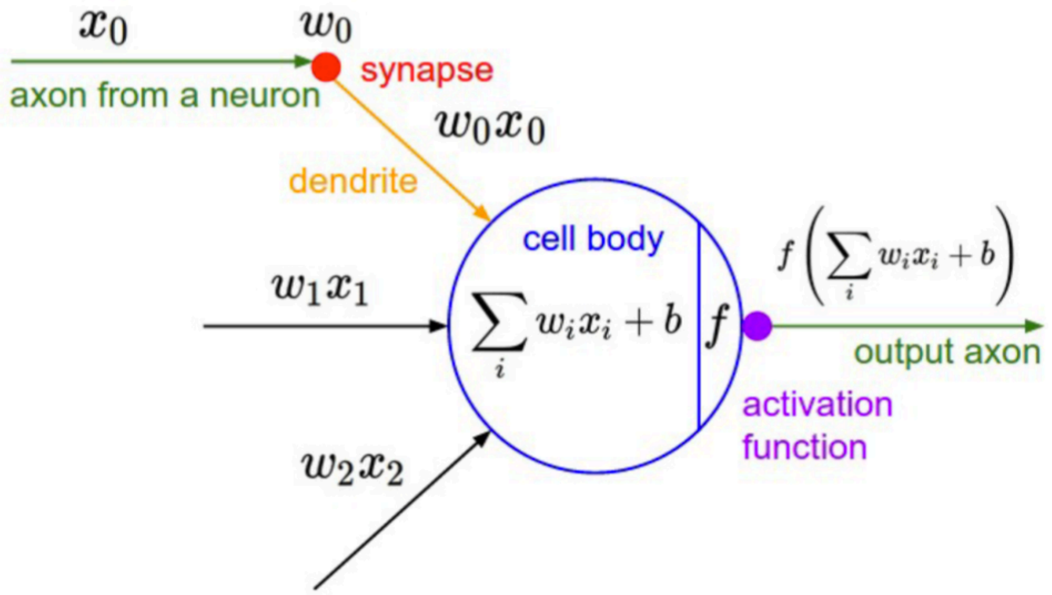
Neural computation



Computation units: **neurons**

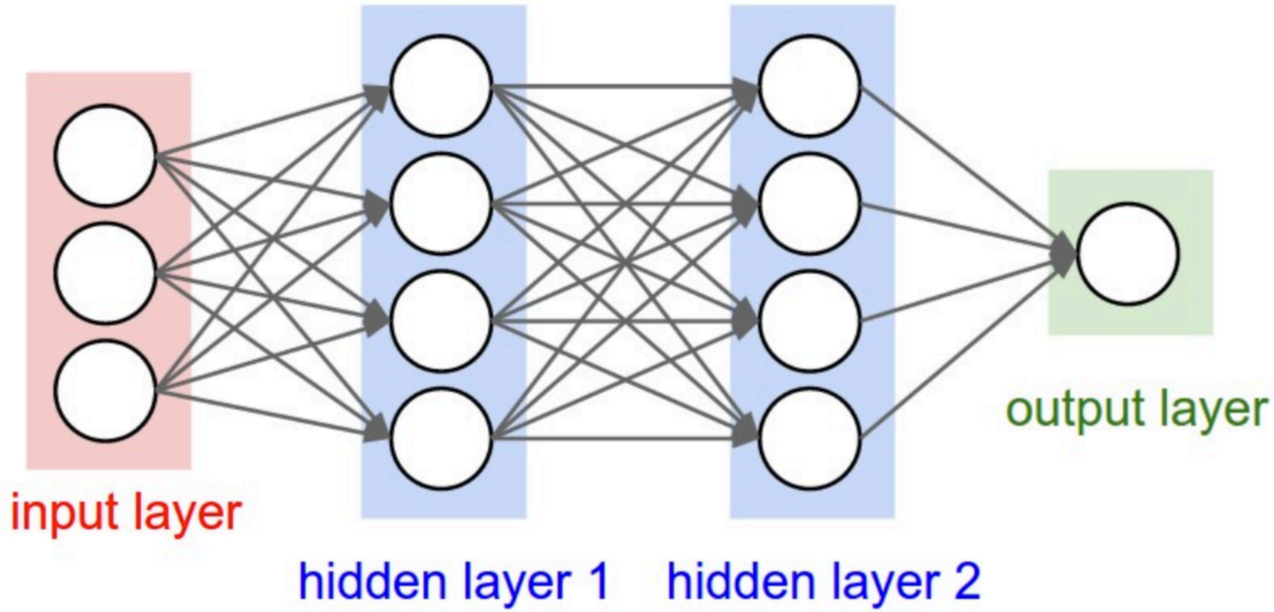
An artificial neuron

- A neuron is a computational unit that has scalar inputs and an output
- Each input has an associated weight.
- The neuron multiples each input by its weight, sums them, applied a **nonlinear function** to the result, and passes it to its output.

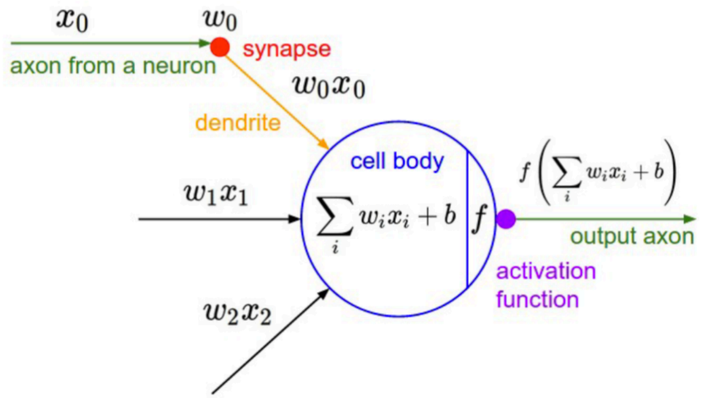


Neural networks

- The neurons are connected to each other, forming a **network**
- The output of a neuron may feed into the inputs of other neurons

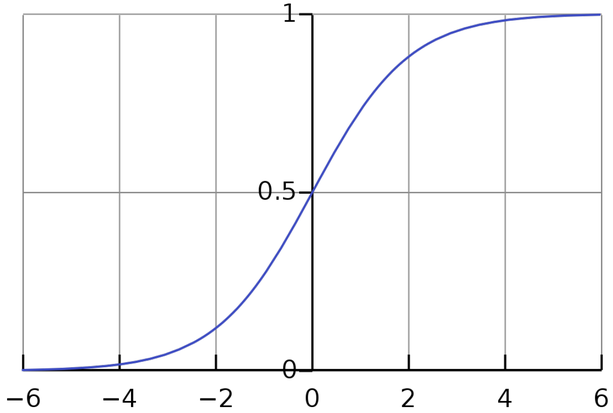


A neuron can be a binary logistic regression unit

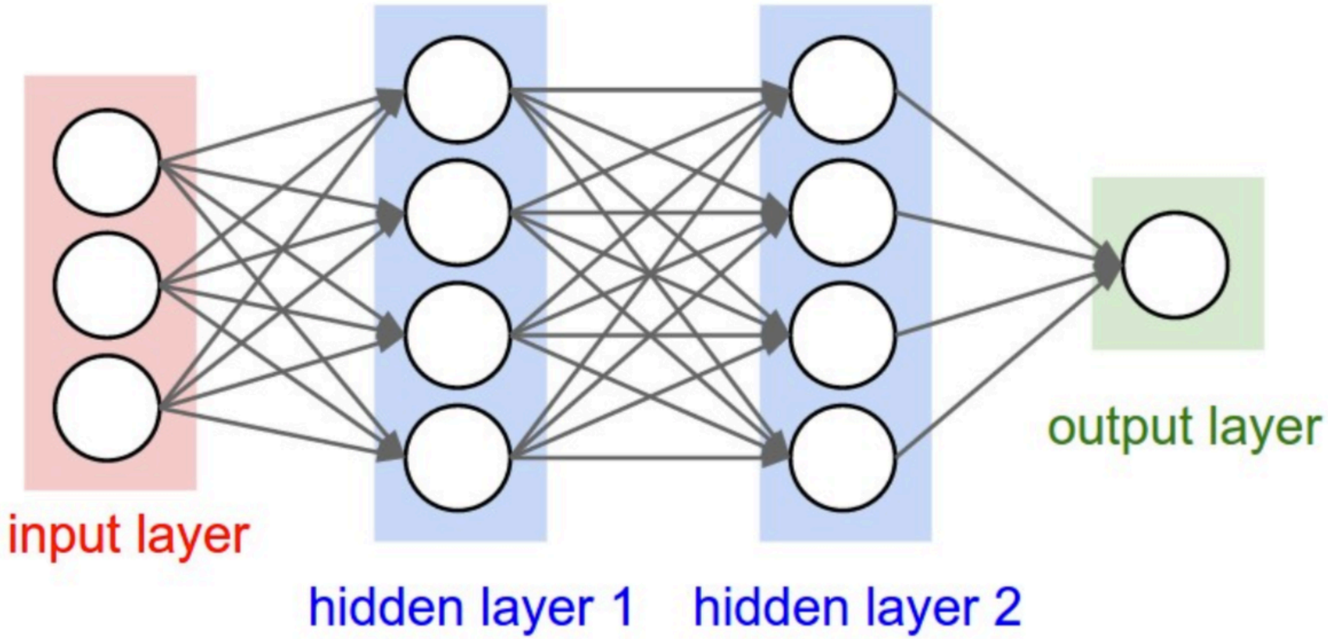


$$f(z) = \frac{1}{1 + e^{-z}}$$

$$h_{\mathbf{w},b}(\mathbf{x}) = f(\mathbf{w}^\intercal \mathbf{x} + b)$$



A neural network
= running several logistic regressions at the same time



- If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs…
- which we can feed into another logistic regression function

Mathematical Notations

- Input layer: $x_1, \ldots, x_d$
- Hidden layer 1: $h_1^{(1)}, h_2^{(1)}, \ldots, h_{d_1}^{(1)}$

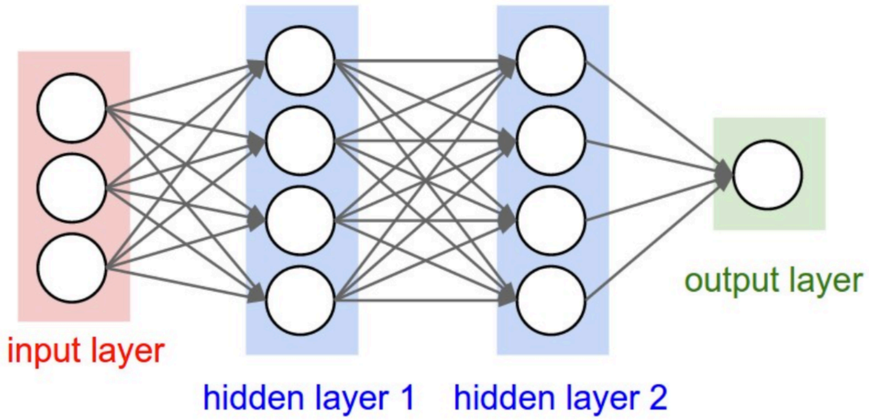
$$h_1^{(1)} = f(W_{1,1}^{(1)} x_1 + W_{1,2}^{(1)} x_2 + \ldots + W_{1,d}^{(1)} x_d + b_1^{(1)})$$
$$h_2^{(1)} = f(W_{2,1}^{(1)} x_1 + W_{2,2}^{(1)} x_2 + \ldots + W_{2,d}^{(1)} x_d + b_2^{(1)})$$
$$\ldots$$

- Hidden layer 2: $h_1^{(2)}, h_2^{(2)}, \ldots, h_{d_2}^{(2)}$

$$h_1^{(2)} = f(W_{1,1}^{(2)} h_1^{(1)} + W_{1,2}^{(2)} h_2^{(1)} + \ldots + W_{1,d_1}^{(2)} h_{d_1}^{(1)} + b_1^{(2)})$$
$$h_2^{(2)} = f(W_{2,1}^{(2)} h_1^{(1)} + W_{2,2}^{(2)} h_2^{(1)} + \ldots + W_{2,d_1}^{(2)} h_{d_1}^{(1)} + b_2^{(2)})$$
$$\ldots$$

- Output layer:

$$y = \sigma(w_1^{(o)} h_1^{(2)} + w_2^{(o)} h_2^{(2)} + \ldots + w_{d_2}^{(o)} h_{d_2}^{(2)} + b^{(o)})$$



Matrix Notations

- Input layer: $\mathbf{x} \in \mathbb{R}^d$
- Hidden layer 1:

$$\mathbf{h}_1 = f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \in \mathbb{R}^{d_1}$$

$$\mathbf{W}^{(1)} \in \mathbb{R}^{d_1 \times d}, \mathbf{b}^{(1)} \in \mathbb{R}^{d_1}$$

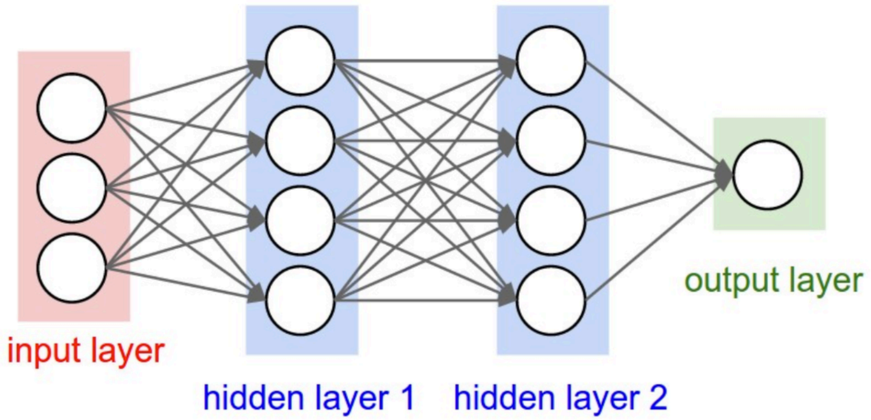
- Hidden layer 2:

$$\mathbf{h}_2 = f(\mathbf{W}^{(2)}\mathbf{h}_1 + \mathbf{b}^{(2)}) \in \mathbb{R}^{d_2}$$

$$\mathbf{W}^{(2)} \in \mathbb{R}^{d_2 \times d_1}, \mathbf{b}^{(2)} \in \mathbb{R}^{d_2}$$

- Output layer:

$$y = \sigma(\mathbf{w}^{(o)} \cdot \mathbf{h}_2 + b^{(o)})$$

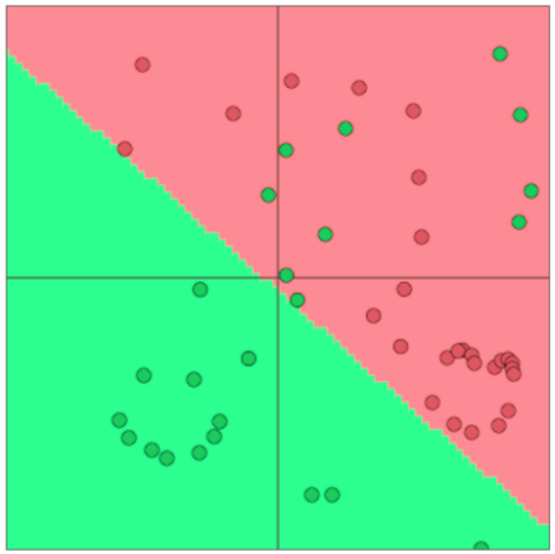


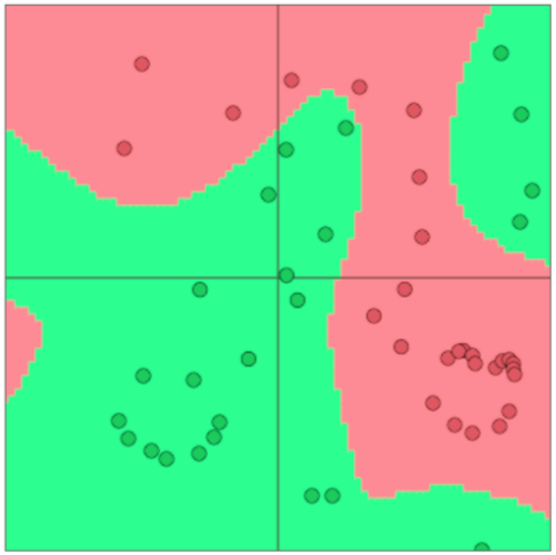
Note: f is applied element-wise

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

Why non-linearities?

- Neural networks can learn much more complex functions and nonlinear decision boundaries





The capacity of the network increases with more hidden units and more hidden layers

What if we remove activation function?

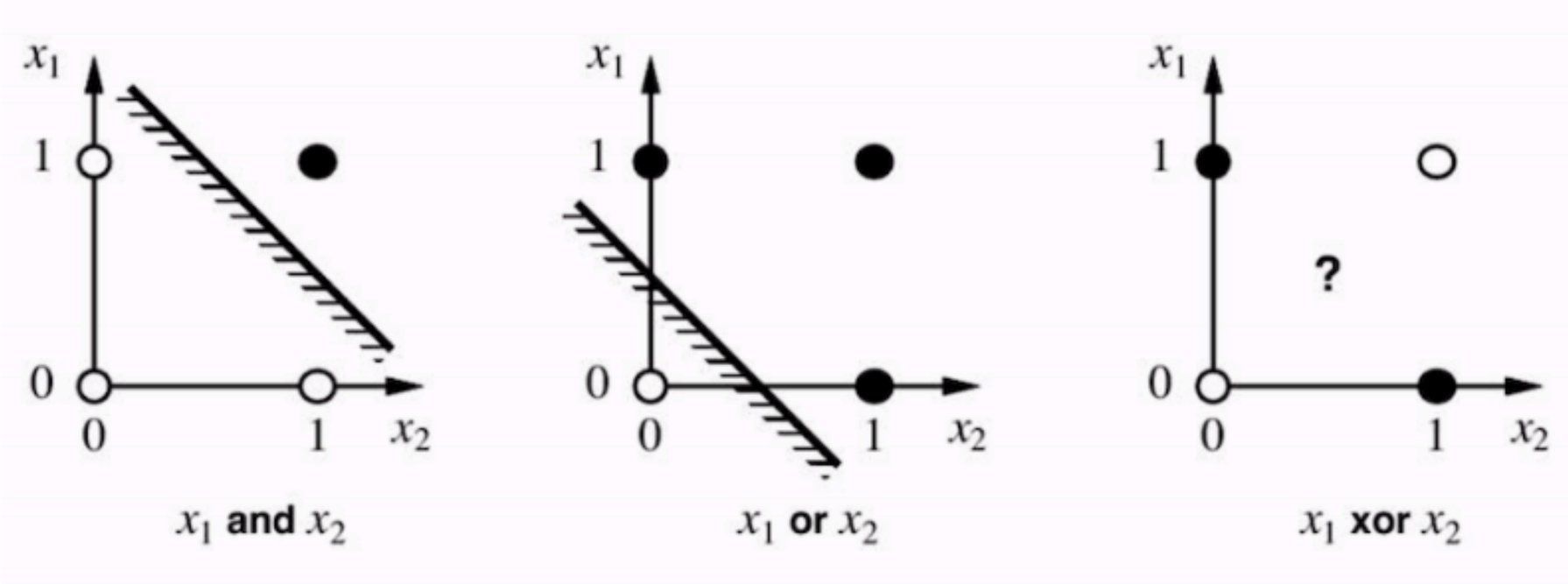
XOR problem

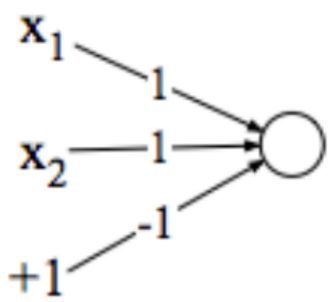
We will assume a training set where each label is in the set $\mathcal{Y} = \{-1, +1\}$

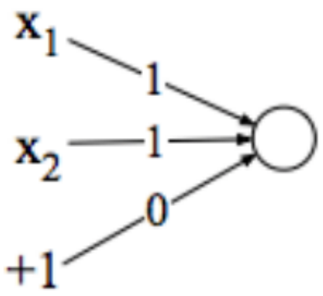
There are four training examples:

$$
\begin{aligned}
x^1 &= [0,0], y^1 = -1 \\
x^2 &= [0,1], y^2 = +1 \\
x^3 &= [1,0], y^3 = +1 \\
x^4 &= [1,1], y^4 = -1
\end{aligned}
$$

Single neuron (perceptron)







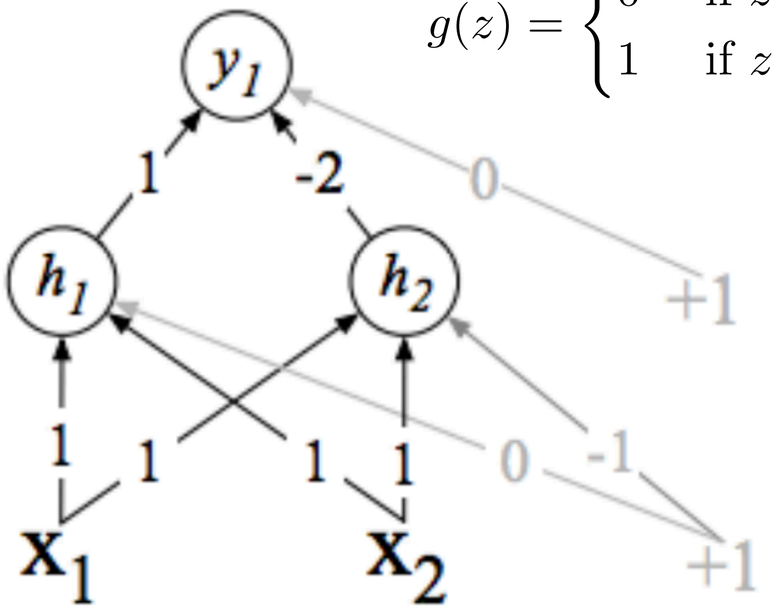
Perceptron can compute **and** and **or**

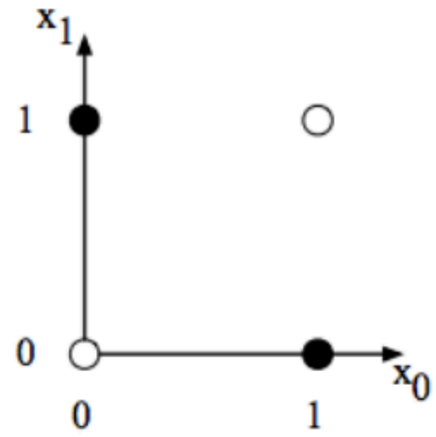
$$y = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

Solution: multiple neurons

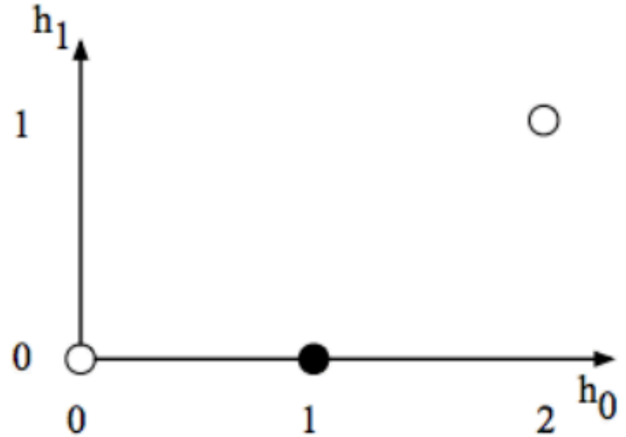
$$y_1 = g(h_1 - 2h_2)$$

$$g(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$





a) The original x space



b) The new h space

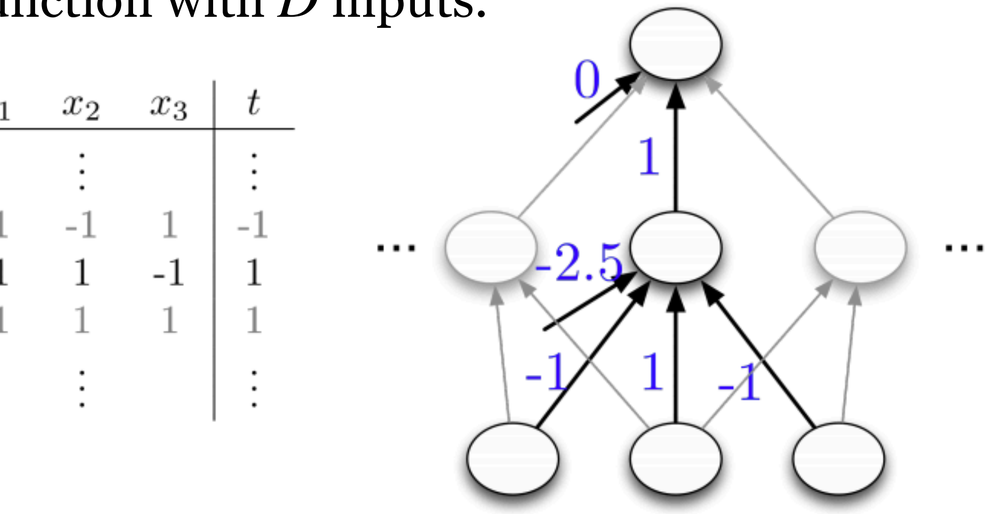
$$h_1 = g(x_1 + x_2)$$

$$h_2 = g(x_1 + x_2 - 1)$$

Expressiveness of neural networks

- Multilayer feed-forward neural nets with nonlinear activation functions are universal approximators
- True for both shallow networks (infinitely wide) and (infinitely) deep networks.
- Consider a network with just 1 hidden layer (with hard threshold activation functions) and a linear output. By having 2^D hidden units, each of which responds to just one input configuration, can model any boolean function with D inputs.

x_1	x_2	x_3	t
	⋮		⋮
-1	-1	1	-1
-1	1	-1	1
-1	1	1	1
	⋮		⋮



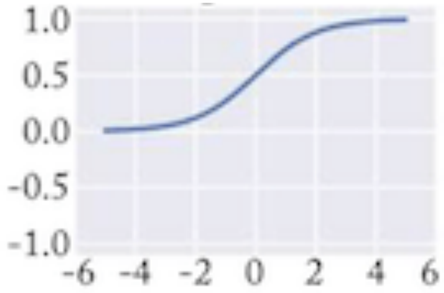
- Deep network can provide a more compact representation

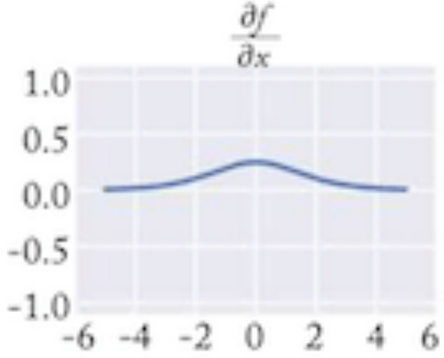
Activation functions

Advantages of ReLU?

sigmoid

$$f(z) = \frac{1}{1 + e^{-z}}$$

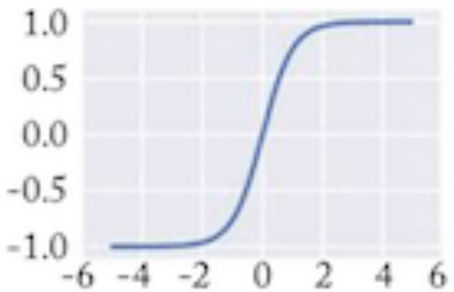


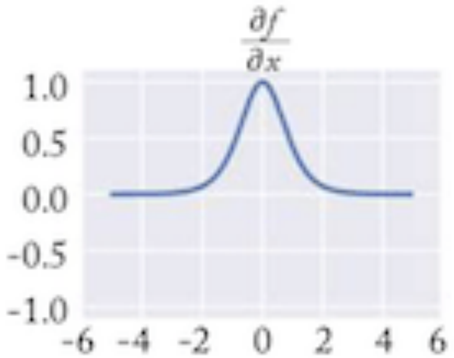


$$f'(z) = f(z) \times (1 - f(z))$$

tanh

$$f(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

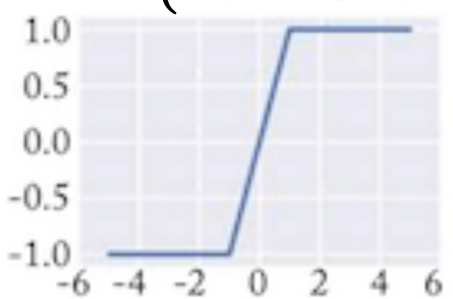


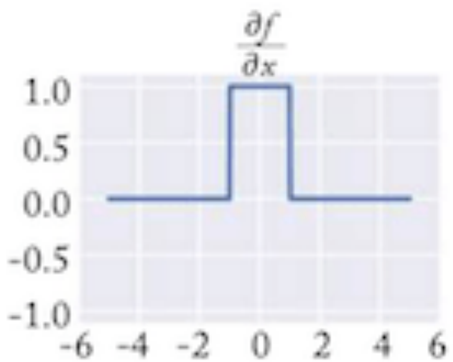


$$f'(z) = 1 - f(z)^2$$

hardtanh
(ramp)

$$f(z) = \begin{cases} 1 & z > 1 \\ z & -1 < z < 1 \\ -1 & z < -1 \end{cases}$$

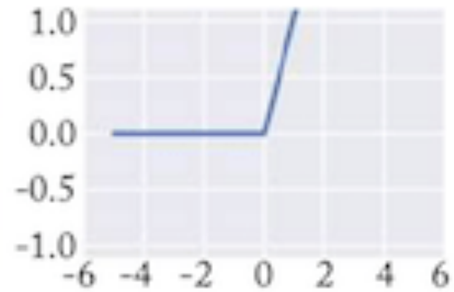


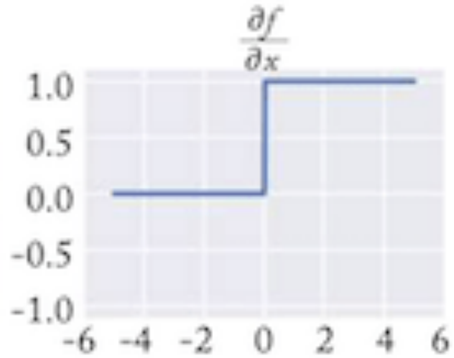


$$f'(z) = \begin{cases} 0 & z > 1 \\ 1 & -1 < z < 1 \\ 0 & z > 1 \end{cases}$$

ReLU
(rectified linear unit)

$$f(z) = \max(0, z)$$





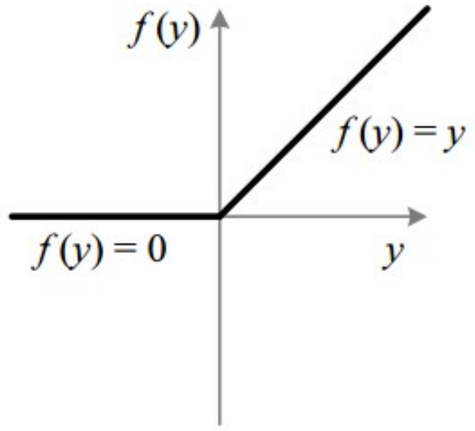
$$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$

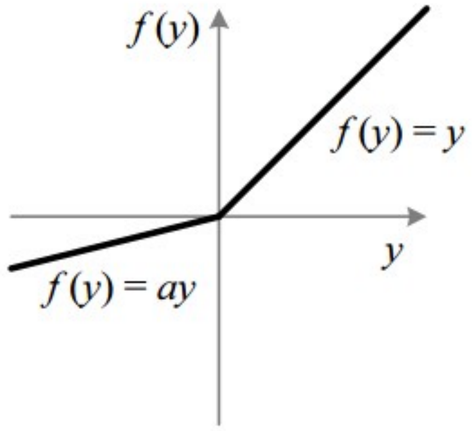
Activation functions

Problems of ReLU? "dead neurons"

Leaky ReLU

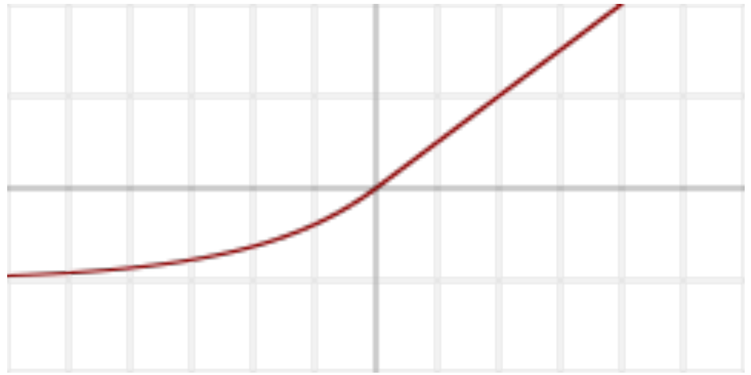
$$f(z) = \begin{cases} z & z \geq 0 \\ 0.01z & z < 0 \end{cases}$$





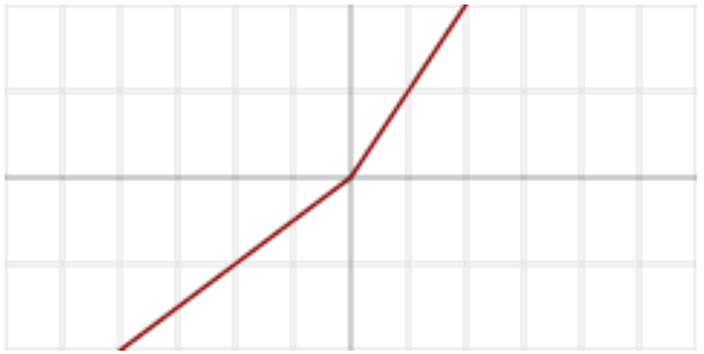
What is the best activation function?

- Depends on the problem!
- ReLU/Leaky ReLU is often a good choice
- Research into families of activation functions



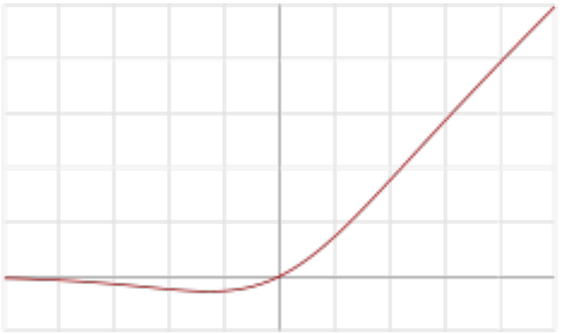
Exponential rectified linear unit (ELU)

$$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$



Parameteric rectified linear unit (PReLU)

$$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$



Swish

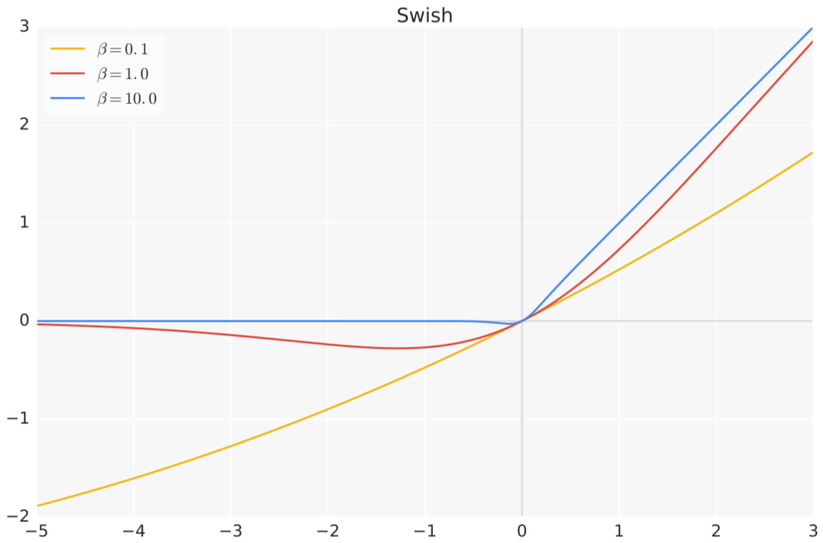
$$f(x) = x\sigma(\beta x) = \frac{x}{1 + e^{-\beta x}}$$

$\beta = 0 \rightarrow f(x) = x/2$

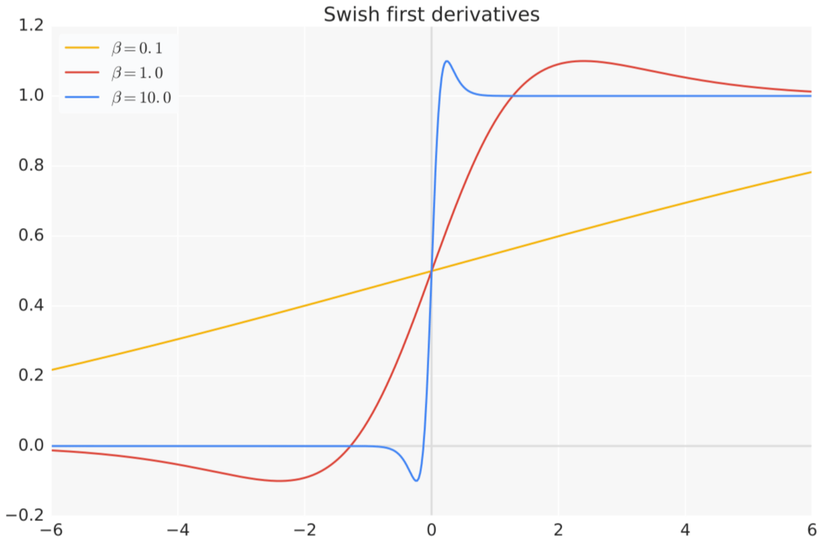
$\beta = \infty \rightarrow f(x) = \text{ReLU}$

α, β can either be constant or trainable parameter

The Swish transfer function



Swish transfer function with different values of β



First derivative of the Swish transfer function

Loss functions

- Binary classification

$$y = \sigma(\mathbf{w}^{(o)} \cdot \mathbf{h}_2 + b^{(o)})$$

$$\mathcal{L}(y, y^*) = -y^* \log y - (1 - y^*) \log (1 - y)$$

- Regression

$$y = \mathbf{w}^{(o)} \cdot \mathbf{h}_2 + b^{(o)}$$

$$\mathcal{L}_{\text{MSE}}(y, y^*) = (y - y^*)^2$$

- Multi-class classification (C classes)

$$y_i = \text{softmax}_i(\mathbf{W}^{(o)}\mathbf{h}_2 + \mathbf{b}^{(o)}) \quad \mathbf{W}^{(o)} \in \mathbb{R}^{C \times d_2}, \mathbf{b}^{(o)} \in \mathbb{R}^C$$

$$\mathcal{L}(y, y^*) = -\sum_{i=1}^{C} y_i^* \log y_i$$

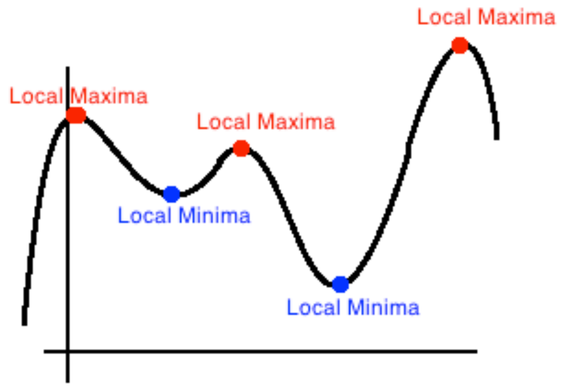
The question again becomes how to compute: $\nabla_\theta \mathcal{L}(\theta)$

$$\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \mathbf{w}^{(o)}, b^{(o)}\}$$

Optimization

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta)$$

- Logistic regression is convex: one global minimum
- Neural networks are non-convex and not easy to optimize
- A class of more sophisticated “adaptive” optimizers that scale the parameter adjustment by an accumulated gradient.
  - **Adam**
  - Adagrad
  - RMSprop
  - …

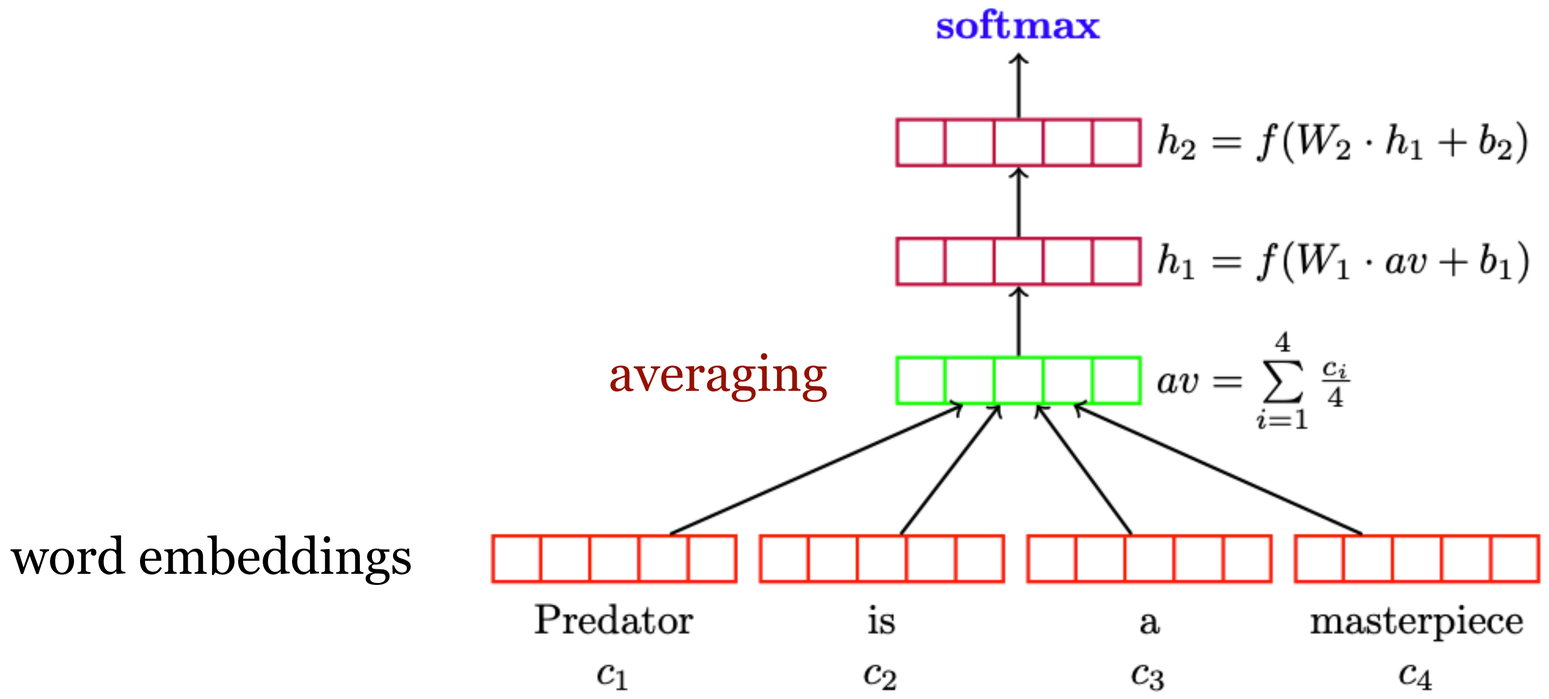


(Ruder 2016): An overview of gradient descent optimization algorithms (https://ruder.io/optimizing-gradient-descent/)

Applications

Neural Bag-of-Words (NBOW)

- Deep Averaging Networks (DAN) for Text Classification



(Iyyer et 2015): Deep Unordered Composition Rivals Syntactic Methods for Text Classification

Word embeddings: re-train or not?

- Word embeddings can be treated as parameters too!

$$\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \mathbf{w}^{(o)}, b^{(o)}, \mathbf{E}_{emb}\}$$

- When the training set is small, don't re-train word embeddings (think of them as features!).

Why?

- Most cases: initialize word embeddings using pre-trained ones (word2vec, Glove) and re-train them for the task

"good" vs "bad"

- When you have enough data, you can just randomly initialize them and train from scratch (e.g. machine translation)

Neural Bag-of-Words (NBOW)

Default: Models are initialized with GloVE embeddings

RAND: Randomly initialized word vectors

Initializing with GloVe is better than random

Rotten Tomatoes Movie Review

Stanford Sentiment Treebank

Model	RT	SST fine	SST bin	IMDB	Time (s)
DAN-ROOT	—	46.9	85.7	—	**31**
DAN-RAND	77.3	45.4	83.2	88.8	136
DAN	80.3	47.7	86.3	89.4	136
NBOW-RAND	76.2	42.3	81.4	88.9	91
NBOW	79.0	43.6	83.6	89.0	91
BiNB	—	41.9	83.1	—	—
NBSVM-bi	79.4	—	—	91.2	—

2 hidden layers better than just averaging

DAN: 2 hidden layers

NBOW: no hidden layer

Bigram Naive Bayes

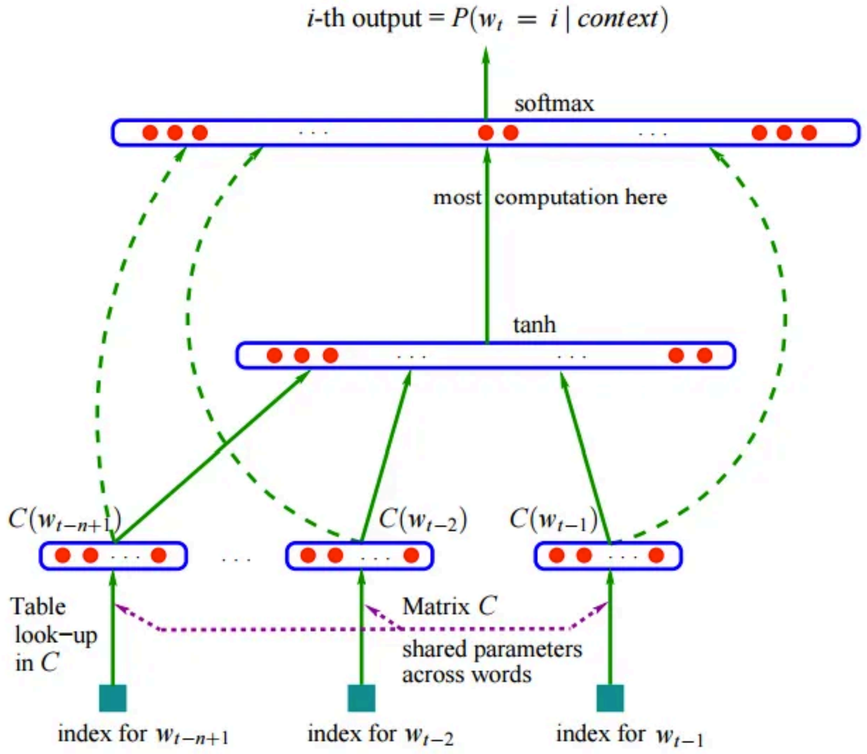
Interpolation of NB and SVM

Memory Intensive ($O(|V|^2)$ features)

(Iyyer et 2015): Deep Unordered Composition Rivals Syntactic Methods for Text Classification

Feedforward Neural LMs

- N-gram models: $P(\text{mat}|\text{the cat sat on the})$



- Input layer (context size n = 5):

$$\mathbf{x} = [\mathbf{e}_{\text{the}}; \mathbf{e}_{\text{cat}}; \mathbf{e}_{\text{sat}}; \mathbf{e}_{\text{on}}; \mathbf{e}_{\text{the}}] \in \mathbb{R}^{dn}$$

concatenation

- Hidden layer

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b}) \in \mathbb{R}^{h}$$

- Output layer (softmax)

$$\mathbf{z} = \mathbf{U}\mathbf{h} \in \mathbb{R}^{|V|}$$

$$P(w = i \mid \text{context}) = softmax_i(\mathbf{z})$$

(Bengio et 2003): A Neural Probabilistic Language Model

Backpropagation

How to compute gradients?

Backpropagation

- It's taking derivatives and applying chain rule!
- We'll **re-use** derivatives computed for higher layers in computing derivatives for lower layers so as to minimize computation
- Good news is that modern automatic differentiation tools did all for you!
    - Implementing backprop by hand is like programming in assembly language.

Deriving gradients for Feedforward NNs

Input: $\mathbf{x}$ $\quad \mathbf{x} \in \mathbb{R}^d$

$\mathbf{h}_1 = \tanh(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$ $\quad \mathbf{W}_1 \in \mathbb{R}^{d_1 \times d}$ $\quad \mathbf{b}_1 \in \mathbb{R}^{d_1}$

$\mathbf{h}_2 = \tanh(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$ $\quad \mathbf{W}_2 \in \mathbb{R}^{d_2 \times d_1}$ $\quad \mathbf{b}_2 \in \mathbb{R}^{d_2}$

$y = \sigma(\mathbf{w}^\intercal \mathbf{h_2} + b)$ $\quad \mathbf{w} \in \mathbb{R}^{d_2}$

$\mathcal{L}(y, y^*) = -y^* \log y - (1 - y^*) \log (1 - y)$

$$\frac{\partial L}{\partial \mathbf{w}} = \; ? \qquad \frac{\partial L}{\partial b} = \; ?$$

$$\frac{\partial L}{\partial \mathbf{W}_2} = \; ? \qquad \frac{\partial L}{\partial \mathbf{b}_2} = \; ?$$

$$\frac{\partial L}{\partial \mathbf{W}_1} = \; ? \qquad \frac{\partial L}{\partial \mathbf{b}_1} = \; ?$$

Deriving gradients for Feedforward NNs

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 \qquad \mathbf{h}_1 = \tanh(\mathbf{z}_1)$$

$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2 \qquad \mathbf{h}_2 = \tanh(\mathbf{z}_2)$$

$$y = \sigma(\mathbf{w}^\intercal \mathbf{h_2} + b)$$

Forward Propagation

$$\mathcal{L}(y, y^*) = -y^* \log y - (1 - y^*) \log (1 - y)$$

$$\frac{\partial \mathcal{L}}{\partial b} = y - y^* \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{w}} = (y - y^*)\mathbf{h}_2 \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{h}_2} = (y - y^*)\mathbf{w}$$

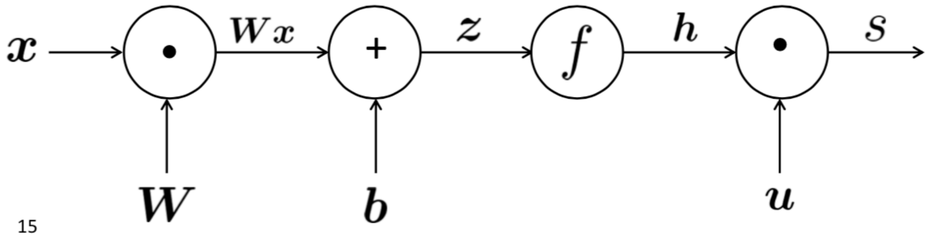
Backward Propagation

$$\frac{\partial L}{\partial \mathbf{z}_2} = (1 - \mathbf{h}_2^2) \circ \frac{\partial L}{\partial \mathbf{h}_2}$$

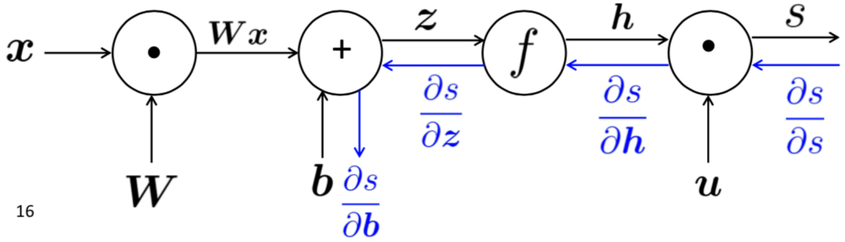
$$\frac{\partial L}{\partial \mathbf{W}_2} = \frac{\partial L}{\partial \mathbf{z}_2}\mathbf{h}_1^\intercal \qquad \frac{\partial L}{\partial \mathbf{b}_2} = \frac{\partial L}{\partial \mathbf{z}_2} \qquad \frac{\partial L}{\partial \mathbf{h}_1} = \mathbf{W}_2^\intercal \frac{\partial L}{\partial \mathbf{z}_2}$$

$$\frac{\partial L}{\partial \mathbf{z}_1} = (1 - \mathbf{h}_1^2) \circ \frac{\partial L}{\partial \mathbf{h}_1} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{z_1}}\mathbf{x}^\intercal \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{z_1}}$$

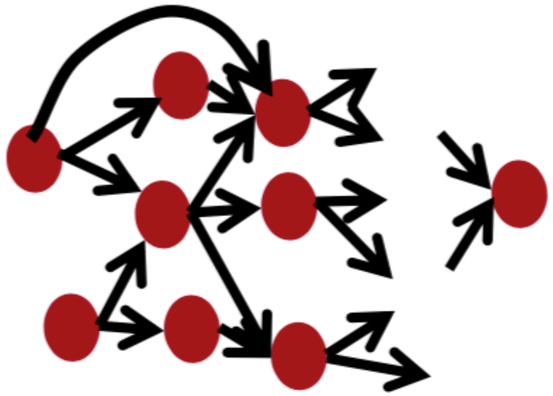
Computational graphs

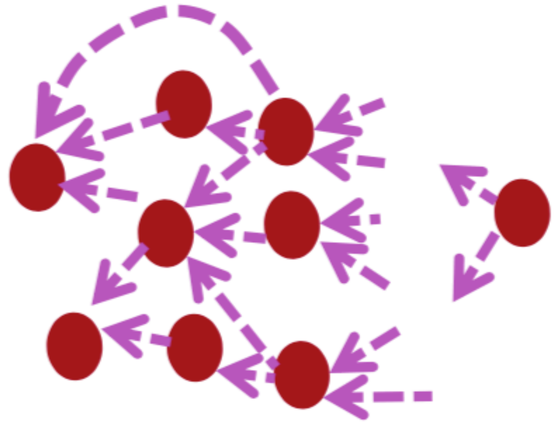


15



16



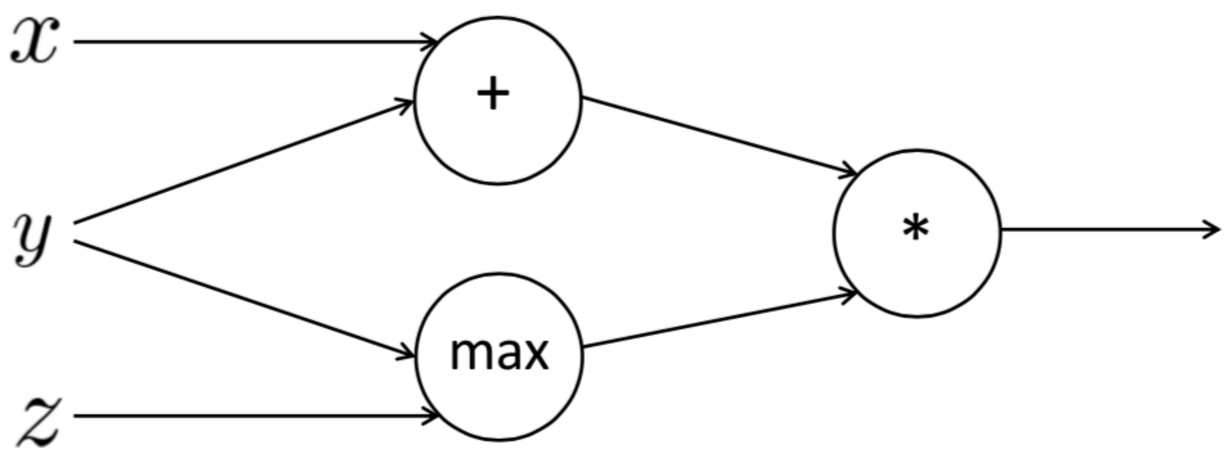


Credits: Chris Manning

An example

$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$



Compute the gradients yourself!

Backpropagation in general computational graph

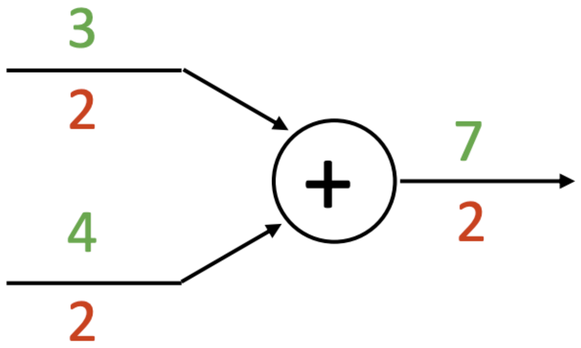
- Forward propagation: visit nodes in topological sort order
  - Compute value of node given predecessors
- Backward propagation:
  - Initialize output gradient as 1
  - Visit nodes in reverse order and compute gradient wrt each node using gradient wrt successors

$$\frac{\partial L}{\partial x} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial y_i} \frac{\partial y_i}{\partial x}$$

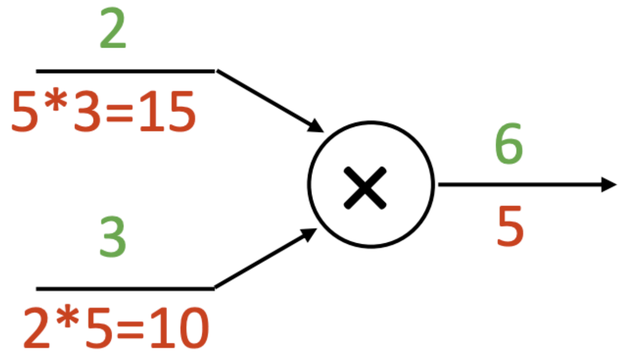
$$\{y_1, \ldots, y_n\} = \text{ successors of } x$$

Patterns in gradient flow

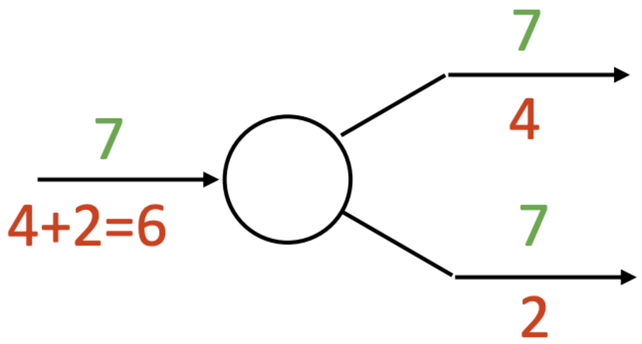
add gate: gradient distributor



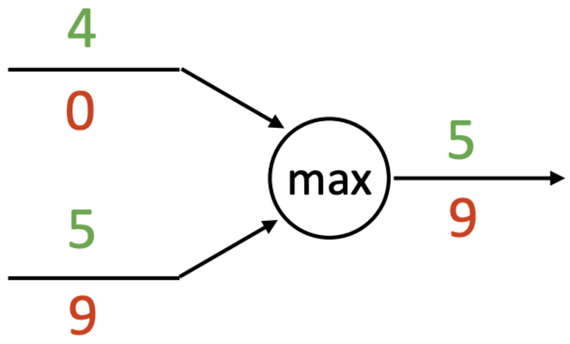
mul gate: "swap multiplier"



copy gate: gradient adder



max gate: gradient router

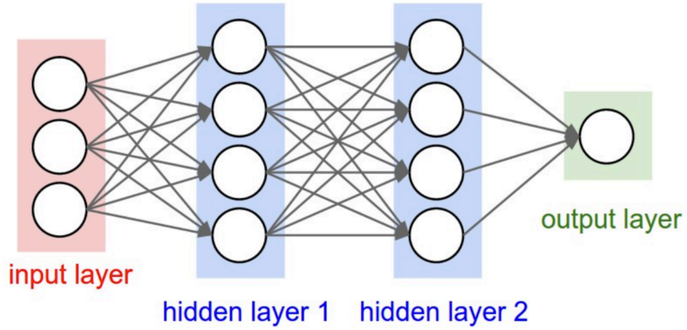


Credits: Justin Johnson

Designing classifiers
with neural networks

Feature design: partly eliminated, partly rolled into network design

- Input **features**: $f(x) \rightarrow [f_1, f_2, \ldots, f_m]$
  - Need to determine features
- Output: estimate $P(y = c \,|\, x)$ for each class c
  - Need to model $P(y = c \,|\, x)$ with a **family of functions**



Neural Networks
figure out architecture

Still need to figure out loss function

- Train phase: Learn **parameters** of model to minimize loss function
  - Need **Loss function** and **Optimization** algorithm

General methods using auto-differentiation

- Test phase: Apply parameters to predict class given a new input

Rise of deep-learning frameworks

Pytorch, TensorFlow, Keras, Theano, …

Provide frequently used components that can be connected together

- Easy to build complex models
  - Connect up neural building blocks
  - Mix and match selection of loss functions, regularizers, and optimizers
- Optimize using auto-differentiation, no need to hand code optimizers for specific models
- Deals with numerical stability issues
- Deals with efficient computation (e.g. using GPUs)
- Provides (some) experiment logging and visualization tools

No longer need to code all the pieces of your model and optimizer by yourself

Allows researchers and developers to focus on

- Modeling the problem
- Designing the network