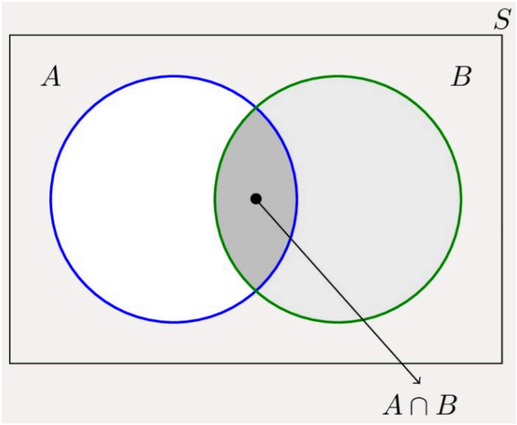
CMPT 413/713: Natural Language Processing

Neural Network Basics

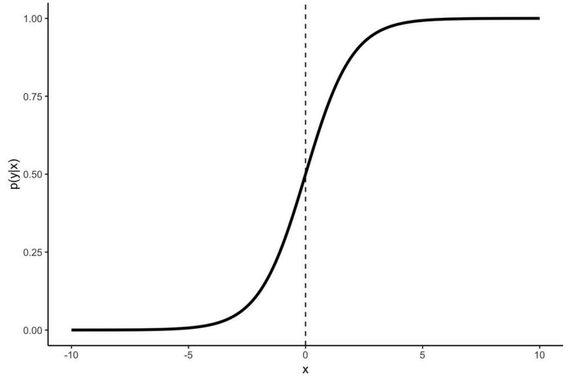
Spring 2024
2024-01-22

Adapted from slides from Danqi Chen and Karthik Narasimhan

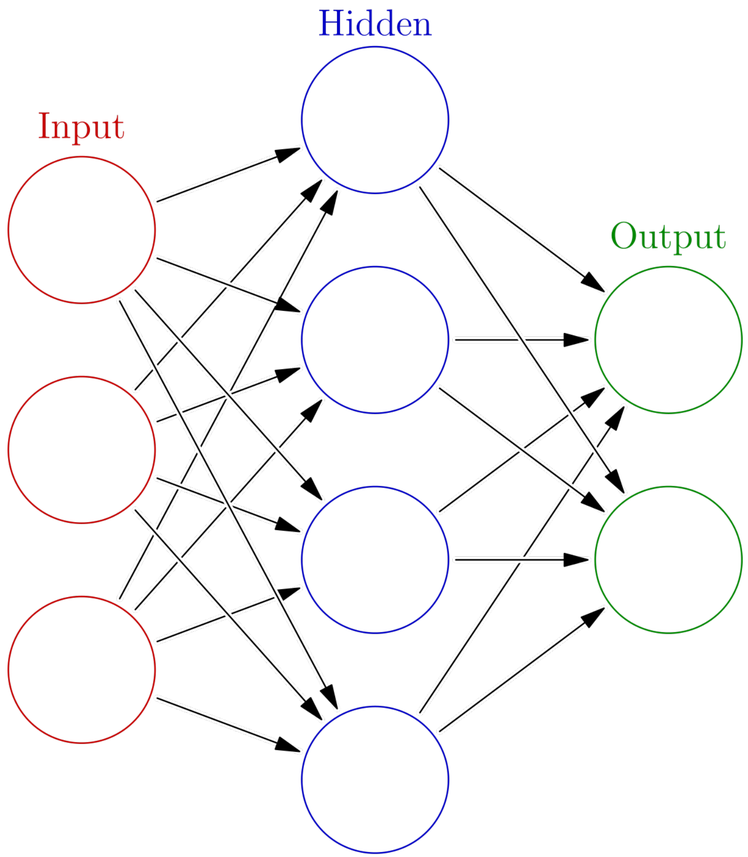
Types of supervised classifiers



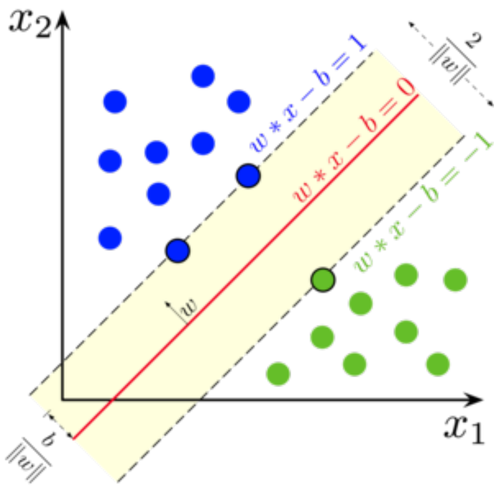
Naive Bayes



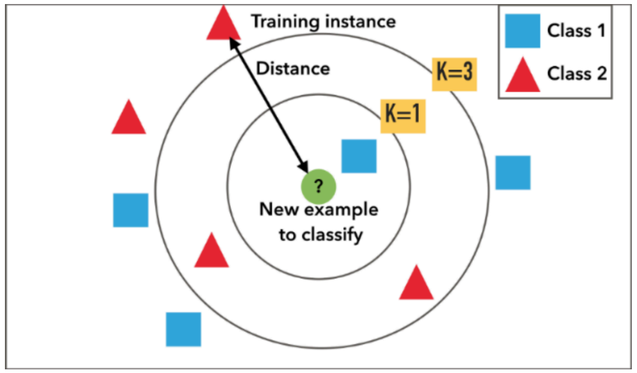
Logistic regression



Neural networks



Support vector machines



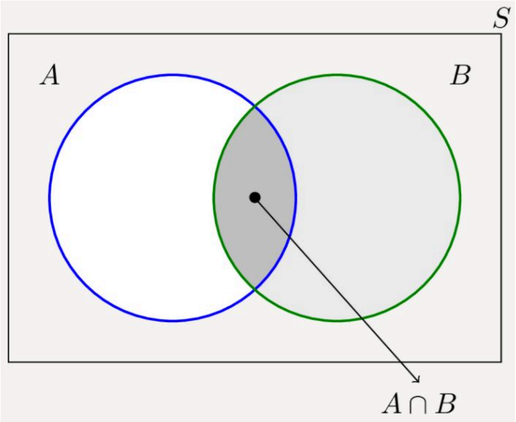
k-nearest neighbors

Multinomial Naive Bayes

- Generative model that assumes input features are conditionally independent given the class:

$$P(y \mid x) = \frac{P(y)P(x \mid y)}{P(x)} = \frac{P(y)\prod_j P(f_j(x) \mid y)}{P(x)}$$

Multinomial refers to the distribution of the features



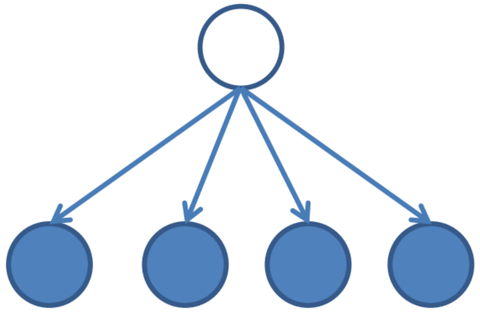
- Parameters are the class priors $P(y)$ and $P(f_j \mid y)$

- Maximum likelihood parameters are easy to estimate (learn)

- Use Laplace smoothing to regularize

$$P(f_j \mid y) = \frac{\text{count}(f_j, y) + \alpha}{\text{count}(y) + m \times \alpha}$$

For BOW model, f_j are the words and $m = |V|$



Generative model

Model joint probability $P(x, y)$

Logistic Regression

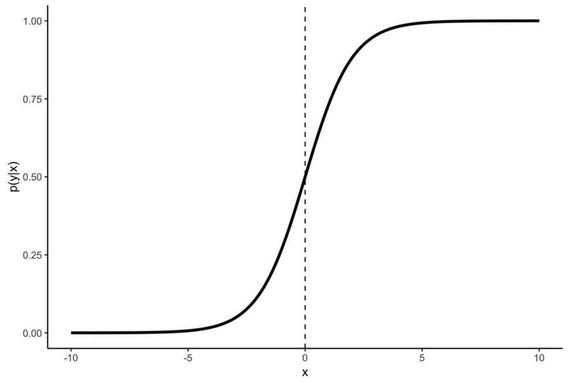
- Models $P(y \mid x)$ as logistic function (or softmax for >2 classes)

$$P(y = c \mid x) = \frac{\exp(\overbrace{\mathbf{w}_c \cdot \mathbf{f}_c(x, c)}^{\text{Score } s_c = \mathbf{w}_c \cdot \mathbf{f}_c(x,c)})}{\sum_{c' \in C} \exp(\mathbf{w}_{c'} \cdot \mathbf{f}_{c'}(x, c'))}$$

Generalization of the logistic function

Parameters are weights $\mathbf{w}$

Features are hand designed



- Does not have independence assumption

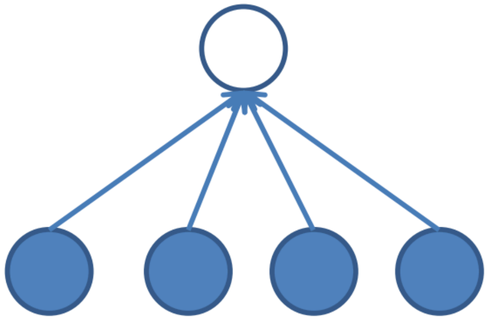
- Train to maximize conditional log-probability

- Maximizing conditional log-probability = Minimizing Cross-Entropy loss

Loss for one training sample (x_i, y_i)

$$L_{\mathrm{CE}_i} = -\log\left(\frac{\exp(s_{y_i})}{\sum_{k \in C} \exp(s_k)}\right)$$

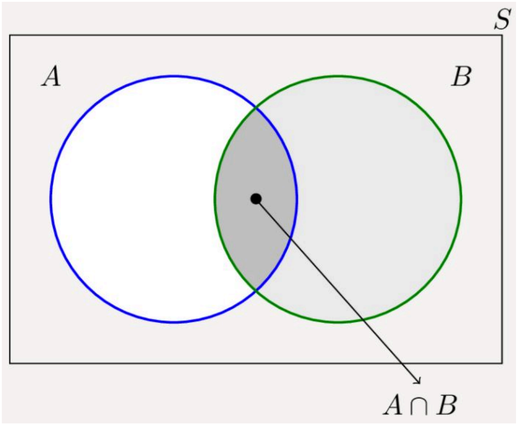
- Also known as Log-linear/MaxEnt model for classification



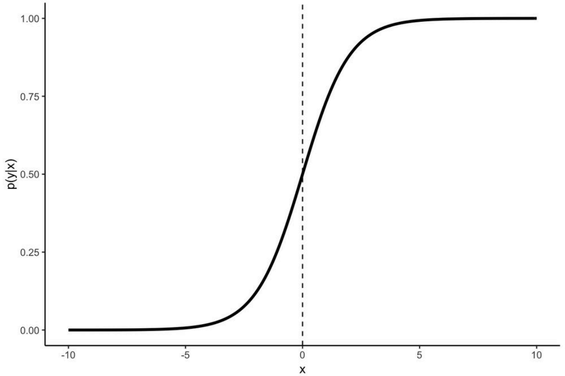
Discriminative model

Model decision boundary directly

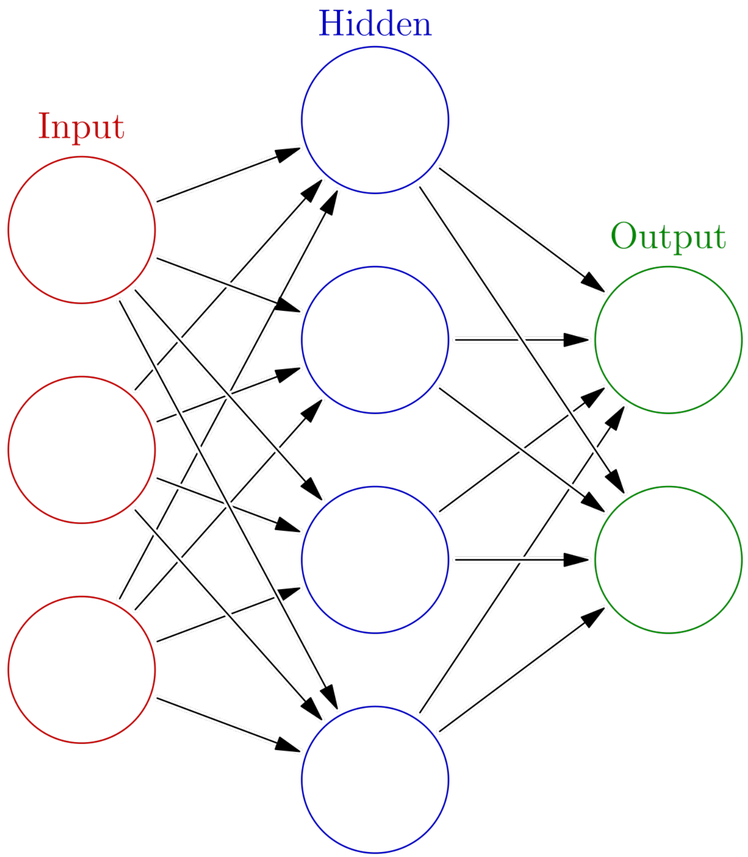
Types of supervised classifiers



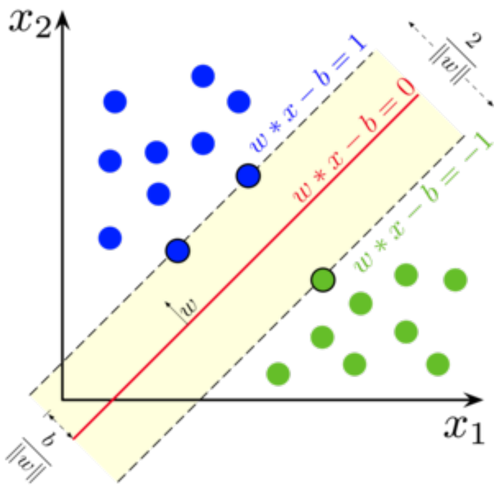
Naive Bayes



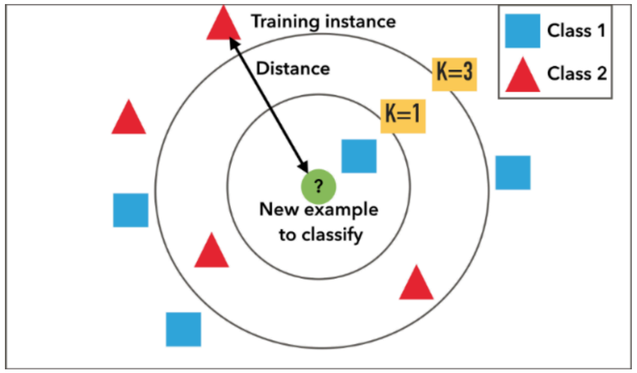
Logistic regression



Neural networks



Support vector machines

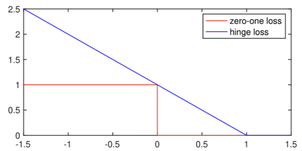


k-nearest neighbors

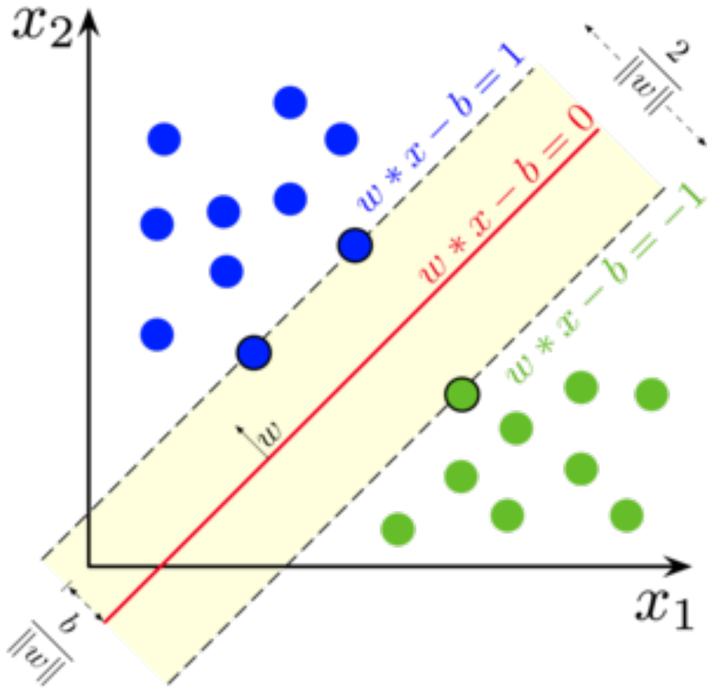
SVM (Support Vector Machines)

- Margin Classifier
  - Find separating hyperplane that maximizes margin
- Linear Classifier: $\hat{y} = \text{sign}(\underbrace{\mathbf{w} \cdot \mathbf{f}(x) - b}_{\text{score } s})$ Binary classes: +1, -1
  - Use kernel trick to transform input into different space
- Soft-margin SVM: find weights $\mathbf{w}$, b to minimize

$$\frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y_i(\mathbf{w} \cdot \mathbf{f}(x_i) - b)) + \lambda \|\mathbf{w}\|^2$$



Hinge/max-margin loss (upper bound on 0-1 loss)



- Multiclass hinge loss

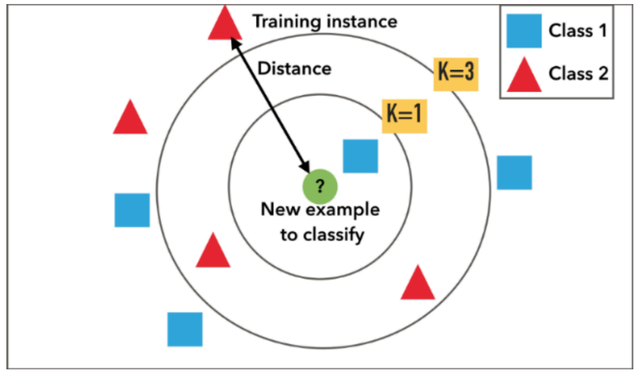
Loss for one training sample (x_i, y_i)

$$L_{\text{Hinge}_i} = \sum_{k \neq y_i} \max(0, s_k - s_{y_i} + 1)$$

$s_k = \mathbf{w}_k \cdot \mathbf{f}(x_i, k)$ $\quad$ $s_{y_i} = \mathbf{w}_{y_i} \cdot \mathbf{f}(x_i, y_i)$

kNN (Nearest Neighbors)

- Requires distance / similarity metric
- Find k nearest neighbors and take majority vote as label
- Non-linear model - very jagged decision boundaries
- Nothing to do during training
  - Need to keep all training data for inference
- Can be expensive (memory + runtime) at inference time
  - Use appropriate nearest neighbor (ANN) algorithms to speed up search for nearest neighbors in high dimensions



K = 3

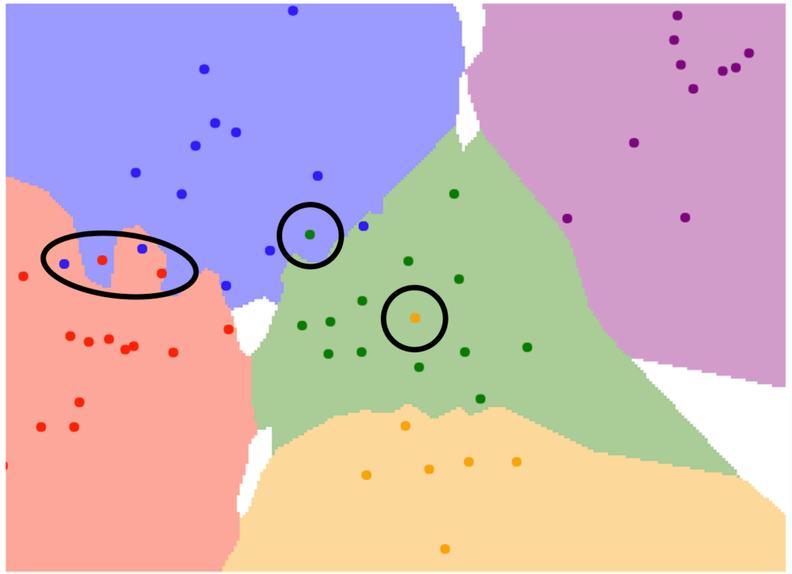
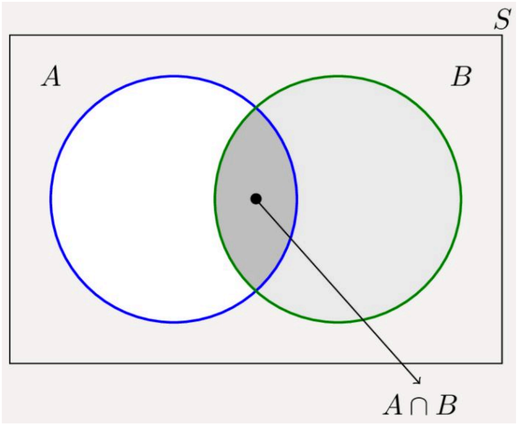
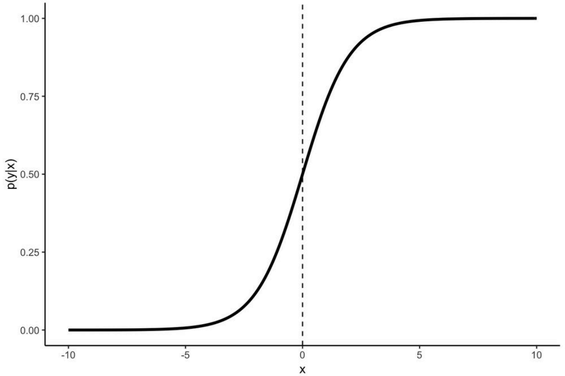


Image Credits: Justin Johnson

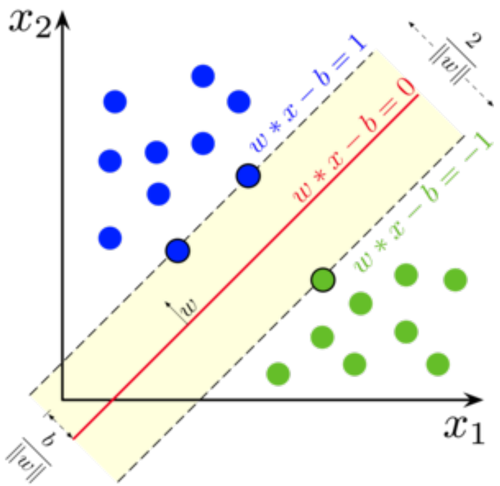
Types of supervised classifiers



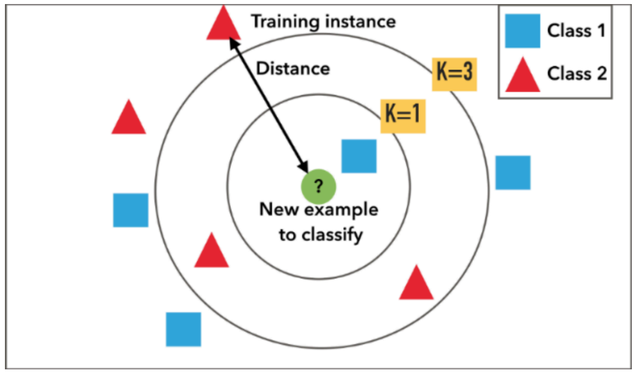
Naive Bayes



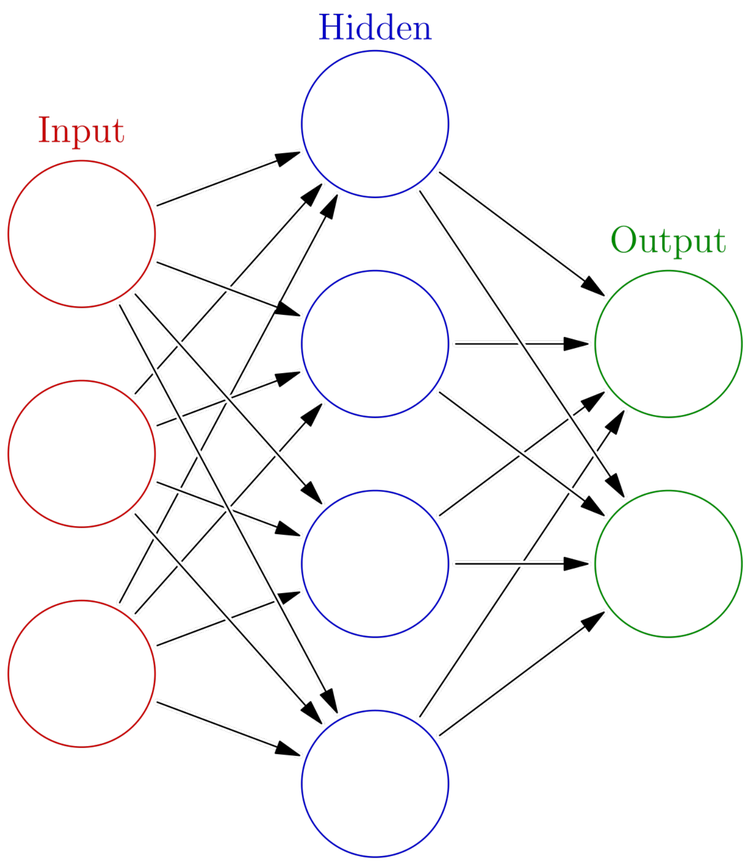
Logistic regression



Support vector machines



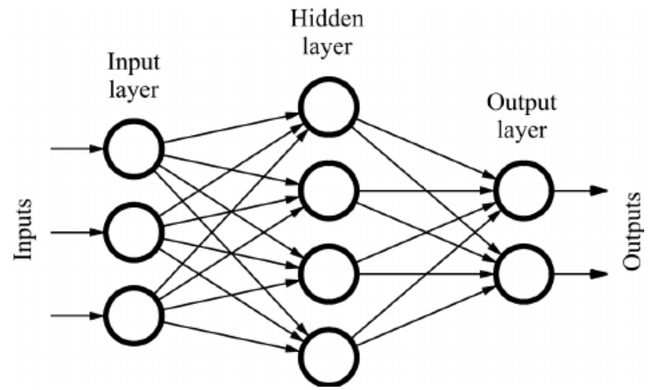
k-nearest neighbors



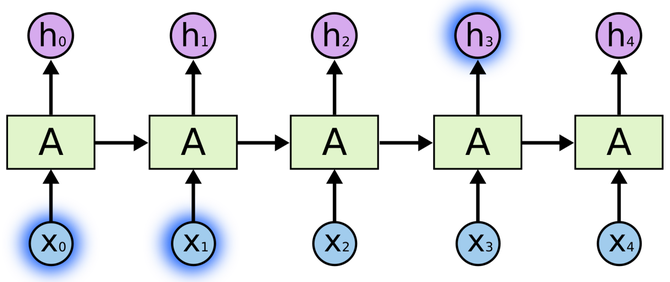
Neural networks

Neural networks for NLP

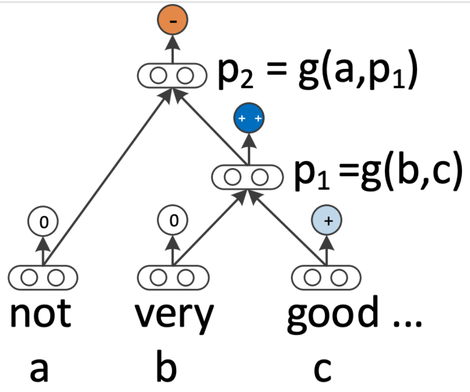
Feed-forward NNs



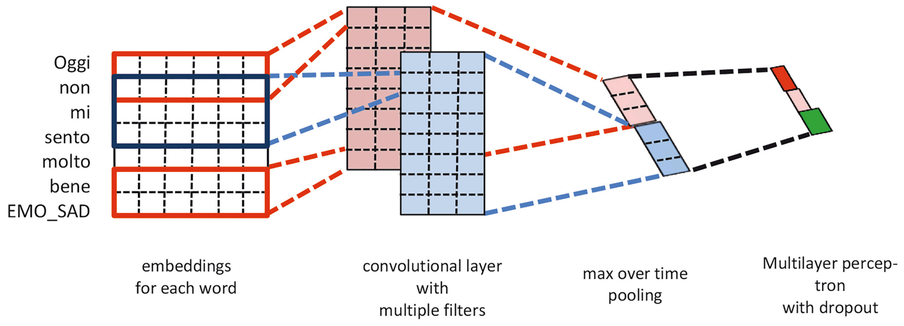
Recurrent NNs



Recursive NNs

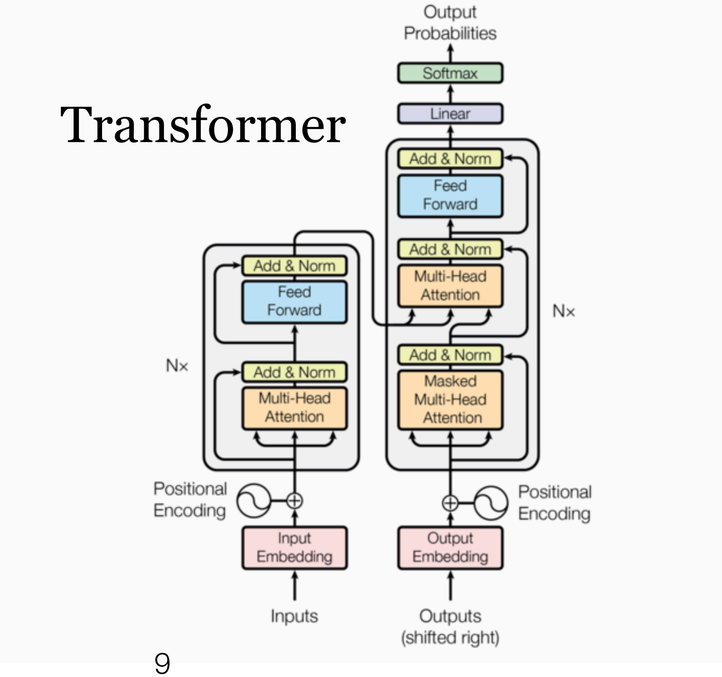


Convolutional NNs

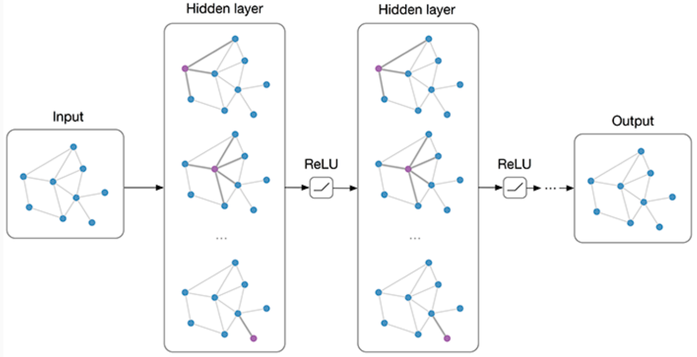


Always coupled with word embeddings...

Transformer

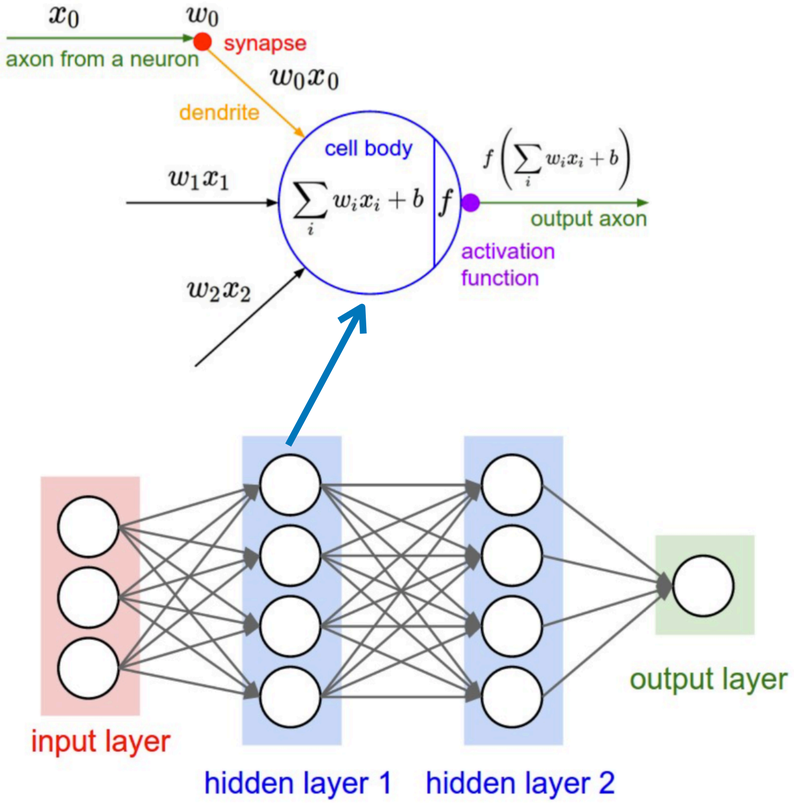


Graph NNs



Neural Networks

- Hookup neurons (computation units with non-linear activation functions) into networks
- Universal function approximations (for arbitrary width or depth)
- Mix-and-match different loss functions and optimizers



Different Loss Functions

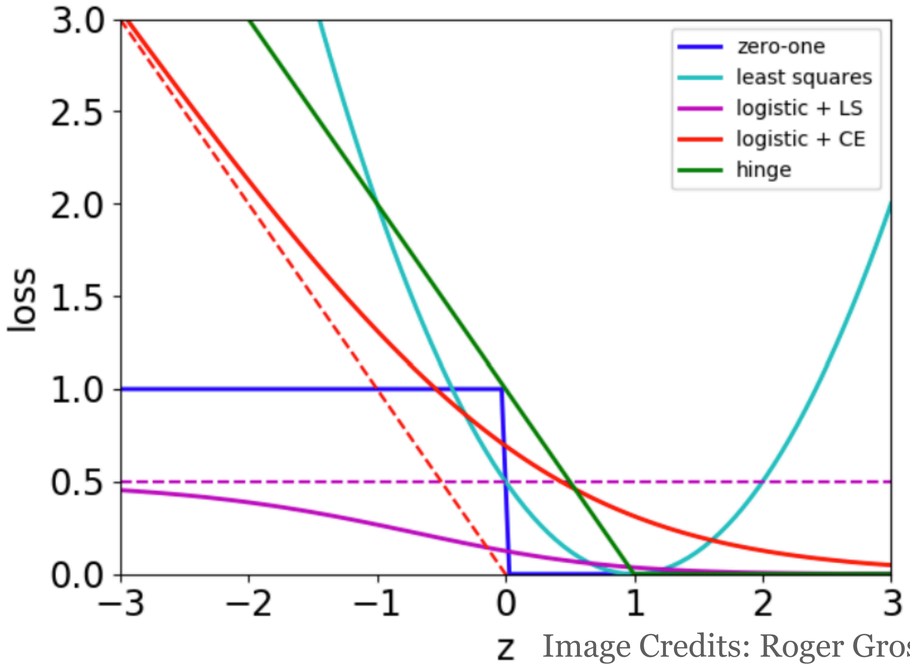
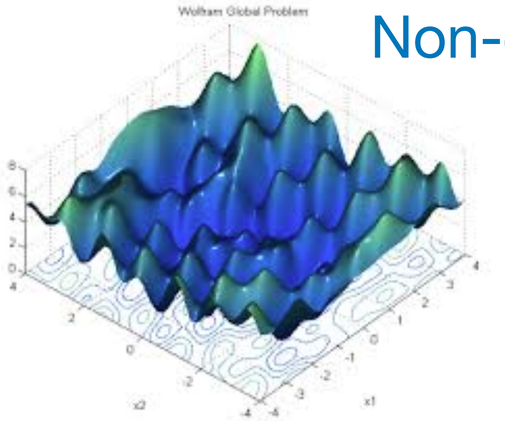
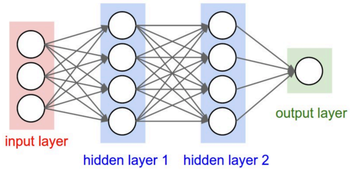


Image Credits: Roger Grosse

Non-convex



Summary of the different models

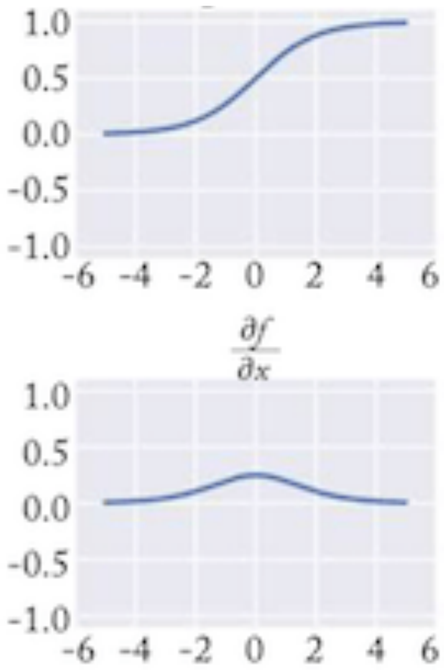
	Features	Function Family	Linear	Parameters	Loss	Optimization
Multinomial Naive Bayes	Typically, hand designed features	$\propto P(y)P(\mathbf{f}(x) \mid y)$	Y	*P(y)*, *P(f_j|y)*	0-1 loss	Estimate probabilities with counts
Logistic Regression		$\frac{\exp(\mathbf{w}_c \cdot \mathbf{f}_c(x,c))}{\sum_{c' \in C} \exp(\mathbf{w}_{c'} \cdot \mathbf{f}_{c'}(x,c'))}$ bias rolled into weights	Y	Weights **w**	Cross Entropy	Convex
SVM		$\mathbf{w} \cdot \mathbf{f}(x)$	Y	Weights **w**	Hinge	Convex
kNN	Need distance/similarity metric		N	No optimization of parameters (but there are hyperparameters)		
Neural Network	Learned from input		N	Network parameters	You choose!	Non-convex Auto-differentiation with SGD + friends

Activation functions

Activation functions

sigmoid

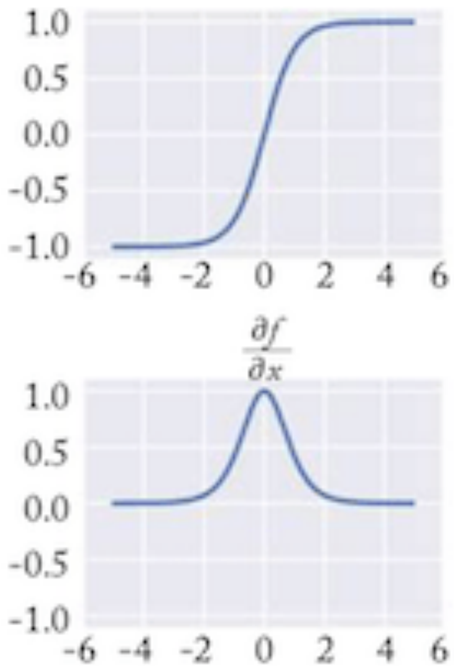
$$f(z) = \frac{1}{1 + e^{-z}}$$



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

Zero centered
tanh

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

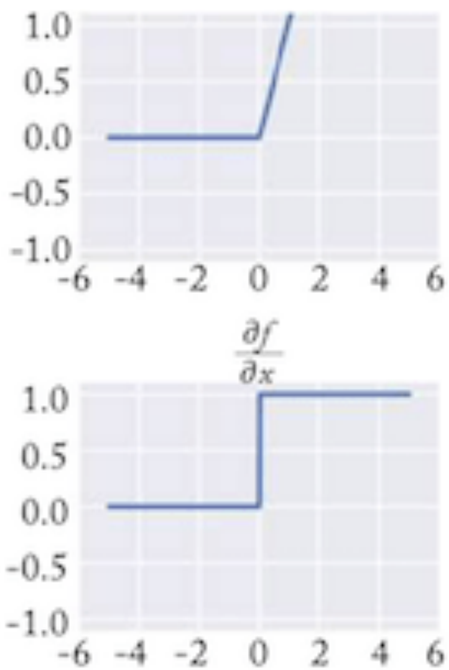


$$f'(z) = 1 - f(z)^2$$

Advantages of ReLU?

ReLU
(rectified linear unit)

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



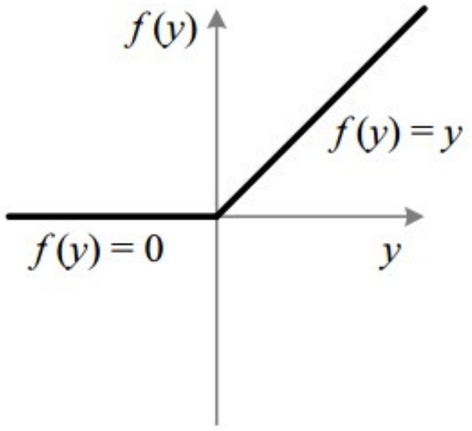
$$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$

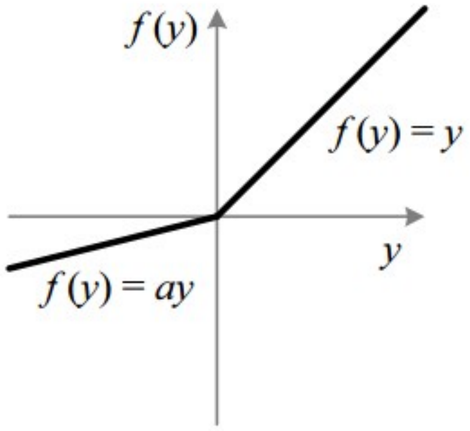
Activation functions

Problems of ReLU? "dead neurons"

Leaky ReLU

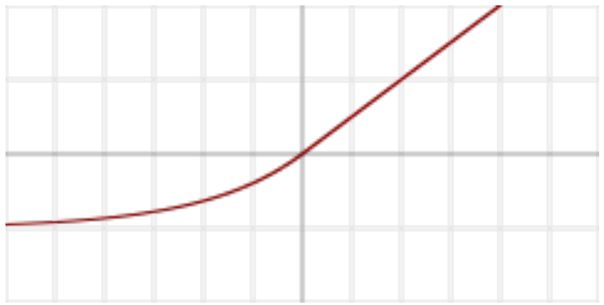
$$f(z) = \begin{cases} z & z \geq 0 \\ 0.01z & z < 0 \end{cases}$$





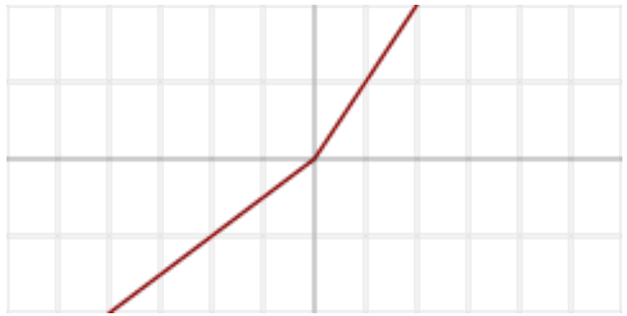
What is the best activation function?

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



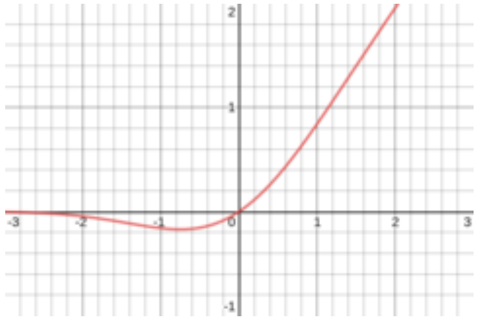
Exponential rectified linear unit (ELU)

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

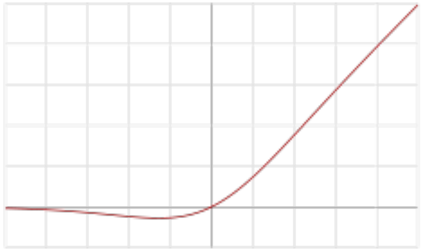


Parameteric rectified linear unit (PReLU)

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



$$\frac{1}{2}x + (1 + \text{erf}(\frac{x}{\sqrt{2}}))$$



Swish

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

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

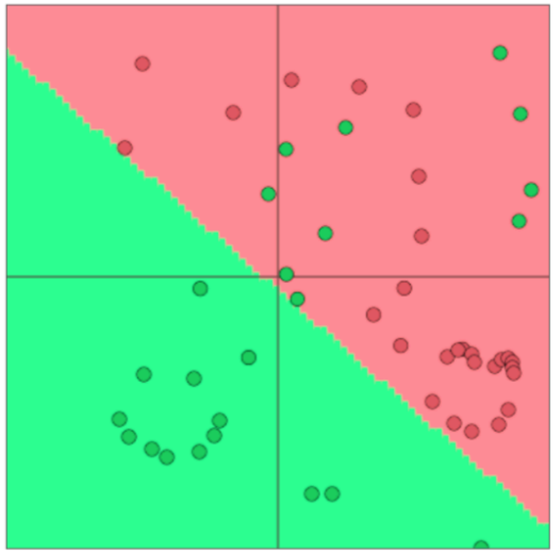
$\beta = \infty \rightarrow f(x) = \text{ReLU}$

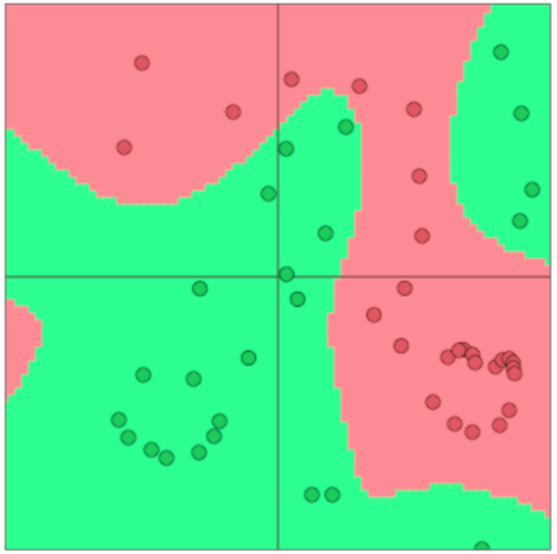
α, β can either be constant or trainable parameter

Expressiveness of neural networks

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?

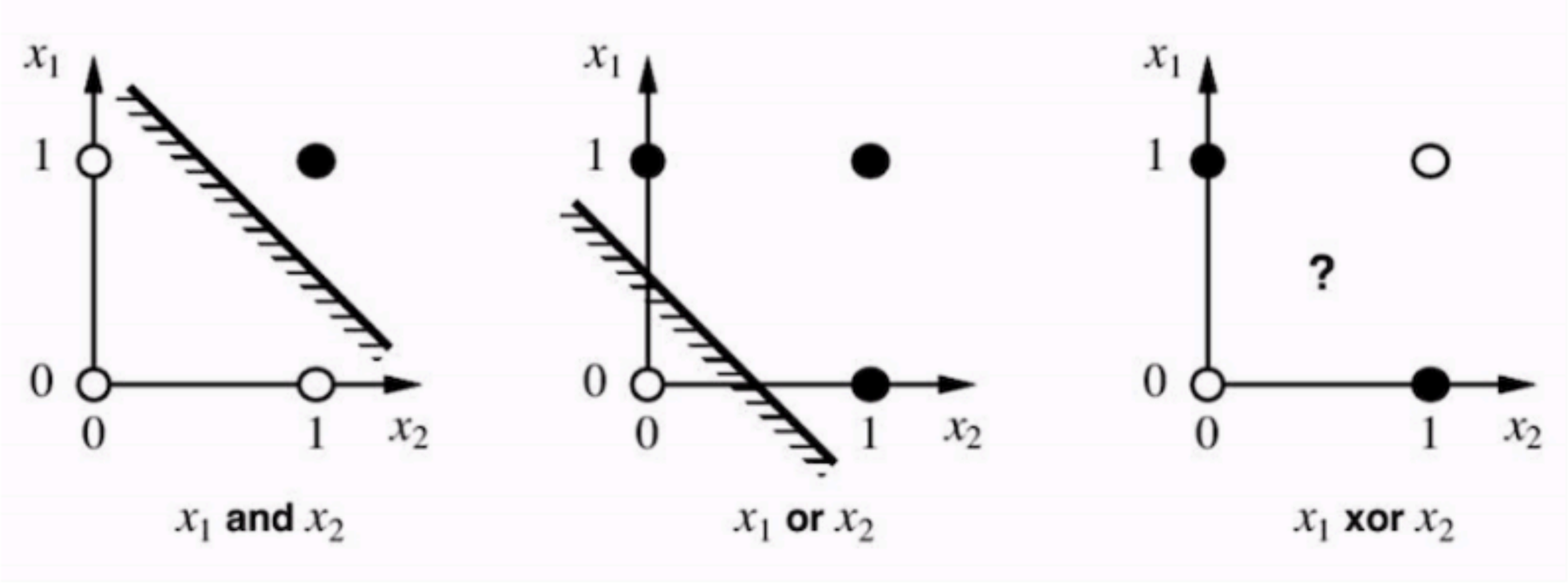
XOR problem

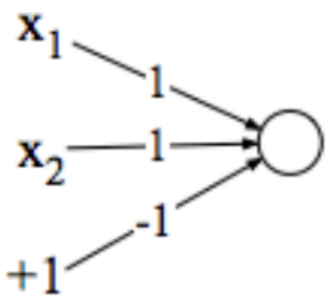
We will assume a training set where each label is in the set $\mathcal{Y} = \{-1, +1\}$

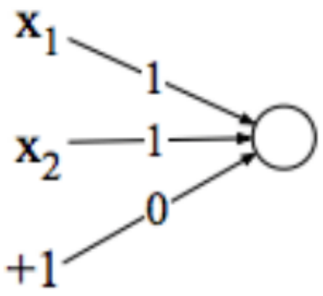
There are four training examples:

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

Single neuron (perceptron)







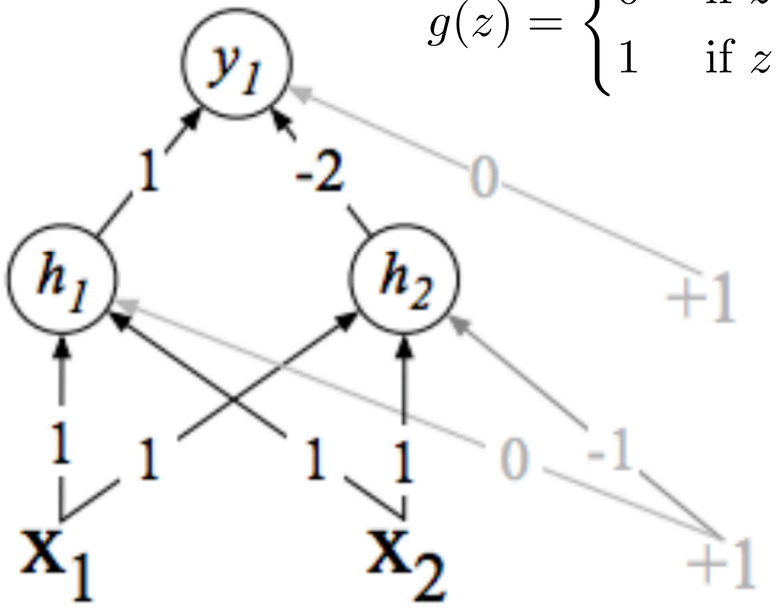
Perceptron can compute **and** and **or**

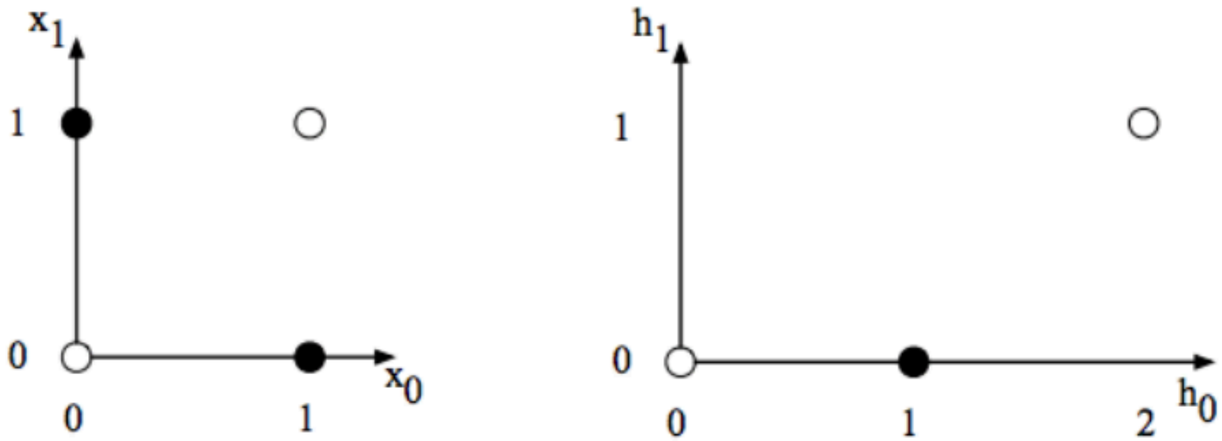
$$y = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

Multiple neurons for XOR

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

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





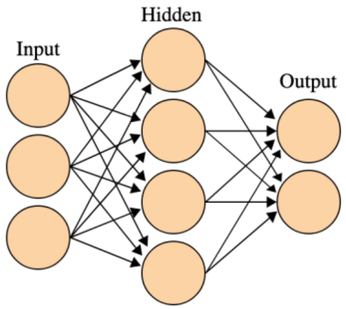
a) The original x space

b) The new h space

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

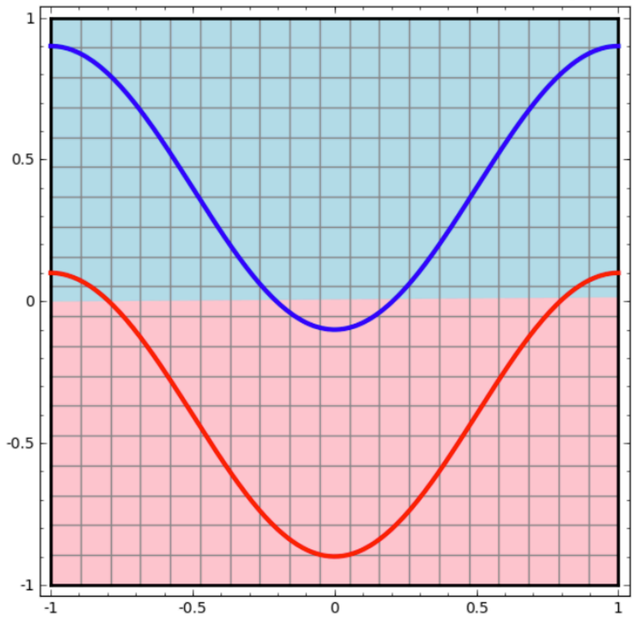
$$h_2 = g(x_1 + x_2 - 1)$$

Why nonlinearities



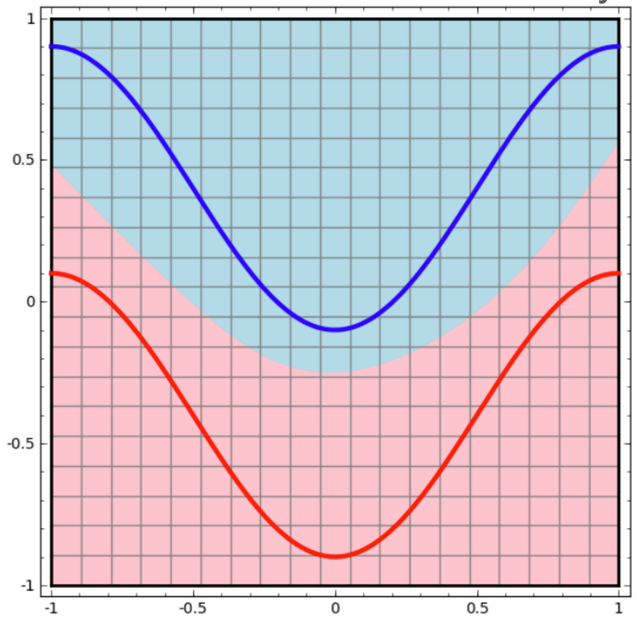
Learn to classify whether points should belong to the blue curve or red curve

Linear decision boundary



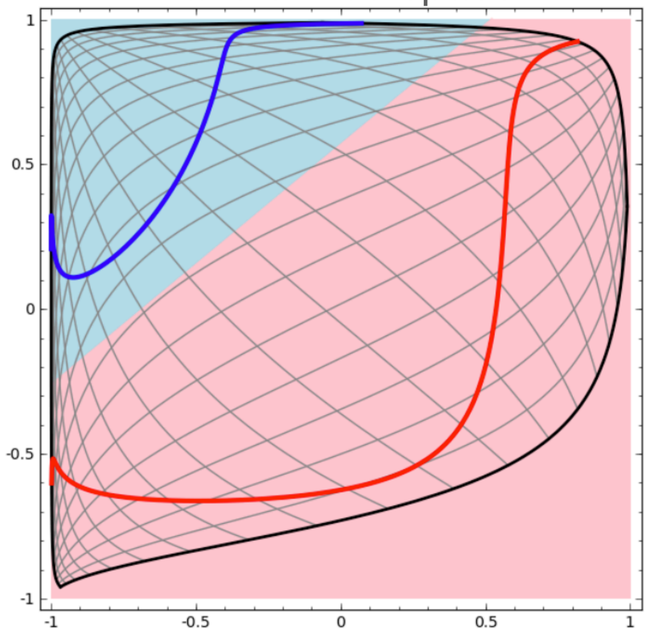
Linear classifier

Non-linear decision boundary



Neural Networks

Linear decision boundary
in transformed space



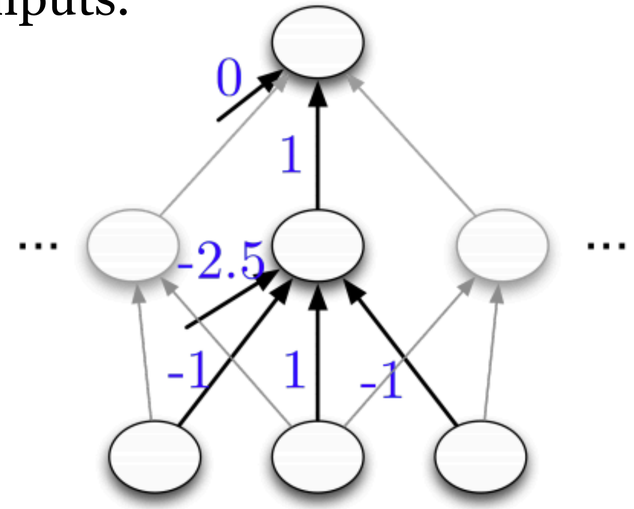
Hidden representations
are linearly separable!

https://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

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

Math notation for neural networks

Mathematical Notations

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

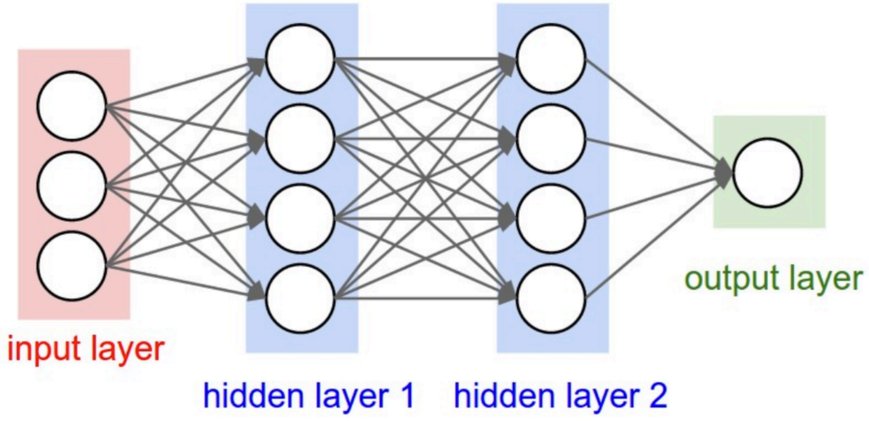
$$h_1^{(1)} = f(W_{1,1}^{(1)} x_1 + W_{1,2}^{(1)} x_2 + \ldots + W_{1,d}^{(1)} x_d + b_1^{(1)})$$
$$h_2^{(1)} = f(W_{2,1}^{(1)} x_1 + W_{2,2}^{(1)} x_2 + \ldots + W_{2,d}^{(1)} x_d + b_2^{(1)})$$
$$\ldots$$

- 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:

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

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

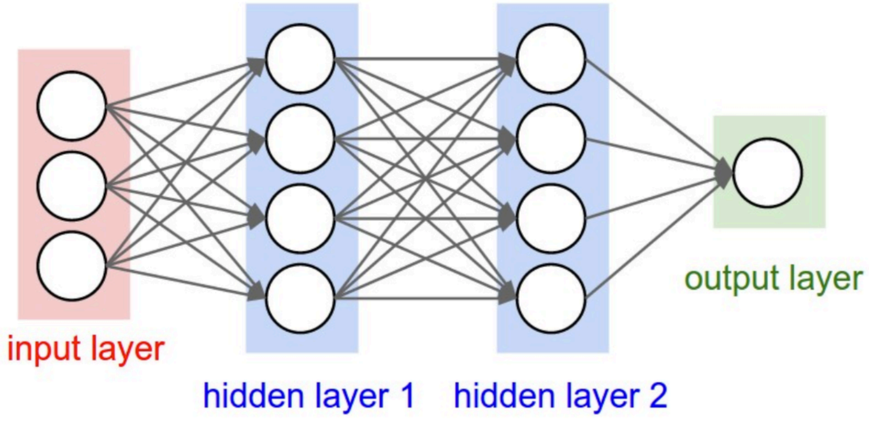
- Hidden layer 2:

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

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

- Output layer:

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



Note: f is applied element-wise

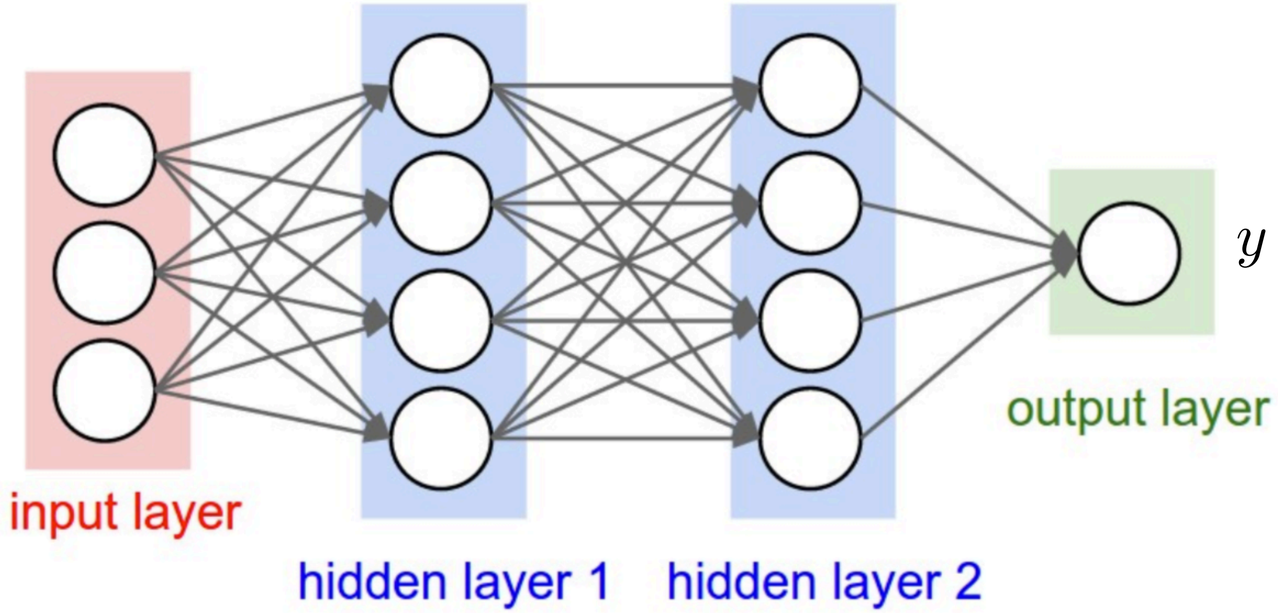
$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

Matrix Notation

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

Input

$\mathbf{x} \in \mathbb{R}^d$



Output

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

$\mathbf{w}^{(o)} \in \mathbb{R}^{d_2}$

$b^{(o)} \in \mathbb{R}$

$\mathbf{h}^{(1)} = \tanh(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$ $\quad$ $\mathbf{h}^{(2)} = \tanh(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$

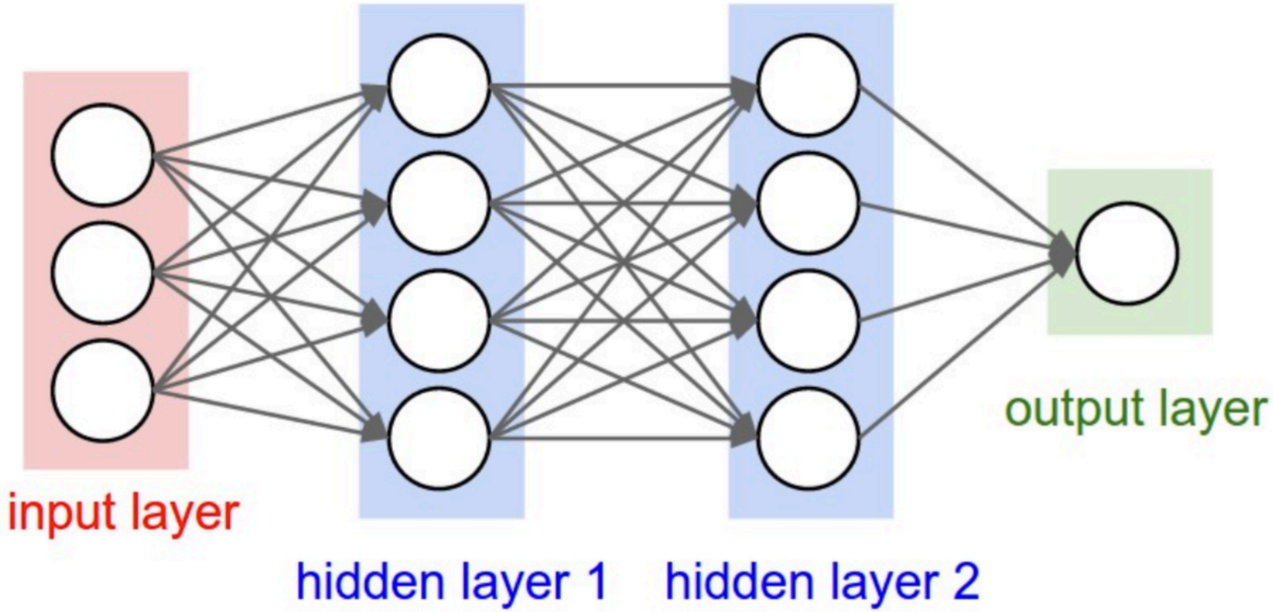
$\mathbf{W}^{(1)} \in \mathbb{R}^{d_1 \times d}$ $\quad$ $\mathbf{b}^{(1)} \in \mathbb{R}^{d_1}$ $\quad$ $\mathbf{W}^{(2)} \in \mathbb{R}^{d_2 \times d_1}$ $\quad$ $\mathbf{b}^{(2)} \in \mathbb{R}^{d_2}$

Matrix Notation

- Learn parameters $\theta = \{\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \mathbf{w}, b\}$

Input

$\mathbf{x} \in \mathbb{R}^d$



Output

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

$\mathbf{w} \in \mathbb{R}^{d_2}$

$\mathbf{h}_1 = \tanh(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$ $\quad$ $\mathbf{h}_2 = \tanh(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$

$\mathbf{W}_1 \in \mathbb{R}^{d_1 \times d}$ $\quad$ $\mathbf{b}_1 \in \mathbb{R}^{d_1}$ $\quad$ $\mathbf{W}_2 \in \mathbb{R}^{d_2 \times d_1}$ $\quad$ $\mathbf{b}_2 \in \mathbb{R}^{d_2}$

Training neural networks

Need loss functions and optimization algorithm

Loss functions

- Binary classification

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

Ground truth label

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

- Regression

$$y = \mathbf{w} \cdot \mathbf{h}_2 + b$$

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

- Multi-class classification (C classes)

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

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

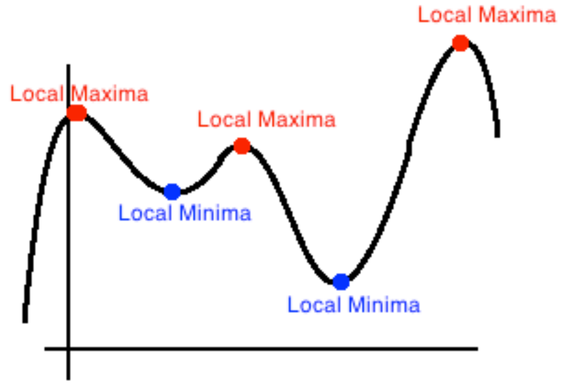
How find parameters to minimize $\mathcal{L}(\theta)$

$$\theta = \{\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \mathbf{w}, b\}$$

Optimization

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

- Optimize using Stochastic Gradient



- Logistic regression is convex: one global minimum
- Neural networks are non-convex and not easy to optimize

- Improvements to SGD: a class of more sophisticated "adaptive" optimizers that scale the parameter adjustment by an accumulated gradient.
  - **AdamW** / Adam
  - Adagrad
  - ...

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

Standard SGD

- Simple SGD: update with only gradients

$$g_t = \nabla_{\theta_{t-1}} L(\theta_{t-1})$$

Gradient of Loss

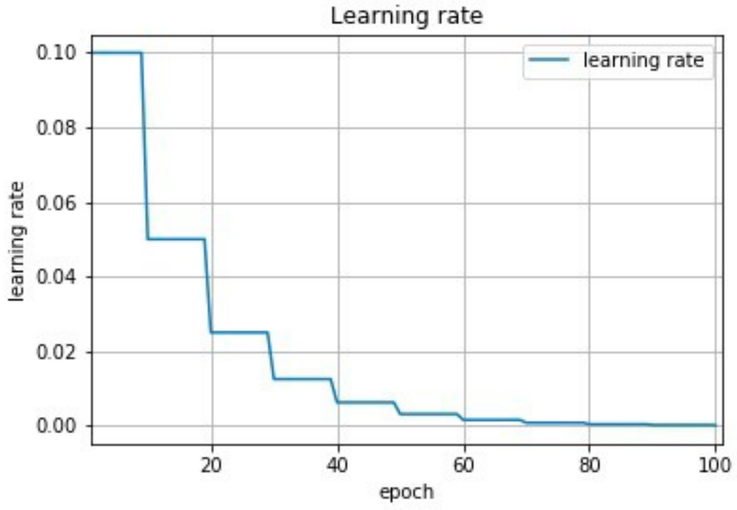
$$\theta_t = \theta_{t-1} - \underline{\eta} g_t$$

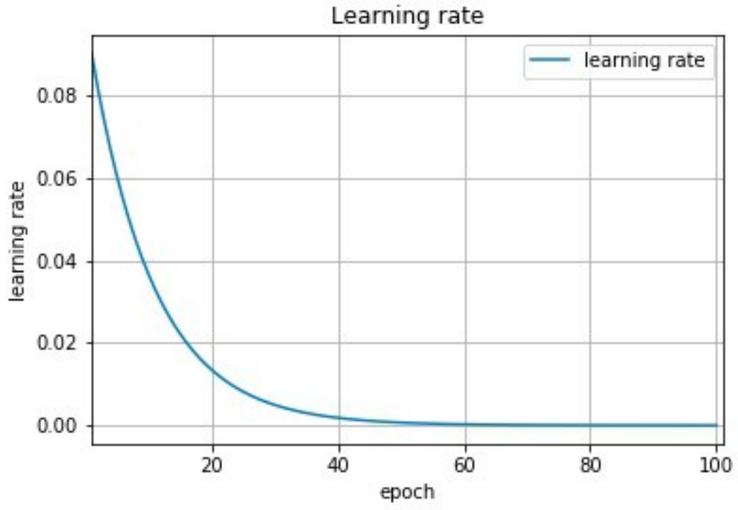
Learning Rate

Credits: Graham Neubig

Using SGD

- Decay learning rate





- Mini-batch (update after seeing m samples)
  - Less variance than pure SGD
  - More efficient than updating the weights after every sample
  - Randomize/shuffle samples in each mini-batch

There is a cost to updating weights

Optimization

- Standard/Vanilla SGD: update with only gradients
- Momentum: update w/ running average of gradients
  - Prevent instability from sudden changes
- Adagrad: different learning rate for each parameter
  - update down weighs high variance values
- Adam: update w/ running average of gradient, down weights by running average of variance
- AdamW: improve version of Adam when using L2 regularization

More details: TA tutorial on optimization

Blog: https://ruder.io/optimizing-gradient-descent/

Adapted from slide by Graham Neubig

Optimization

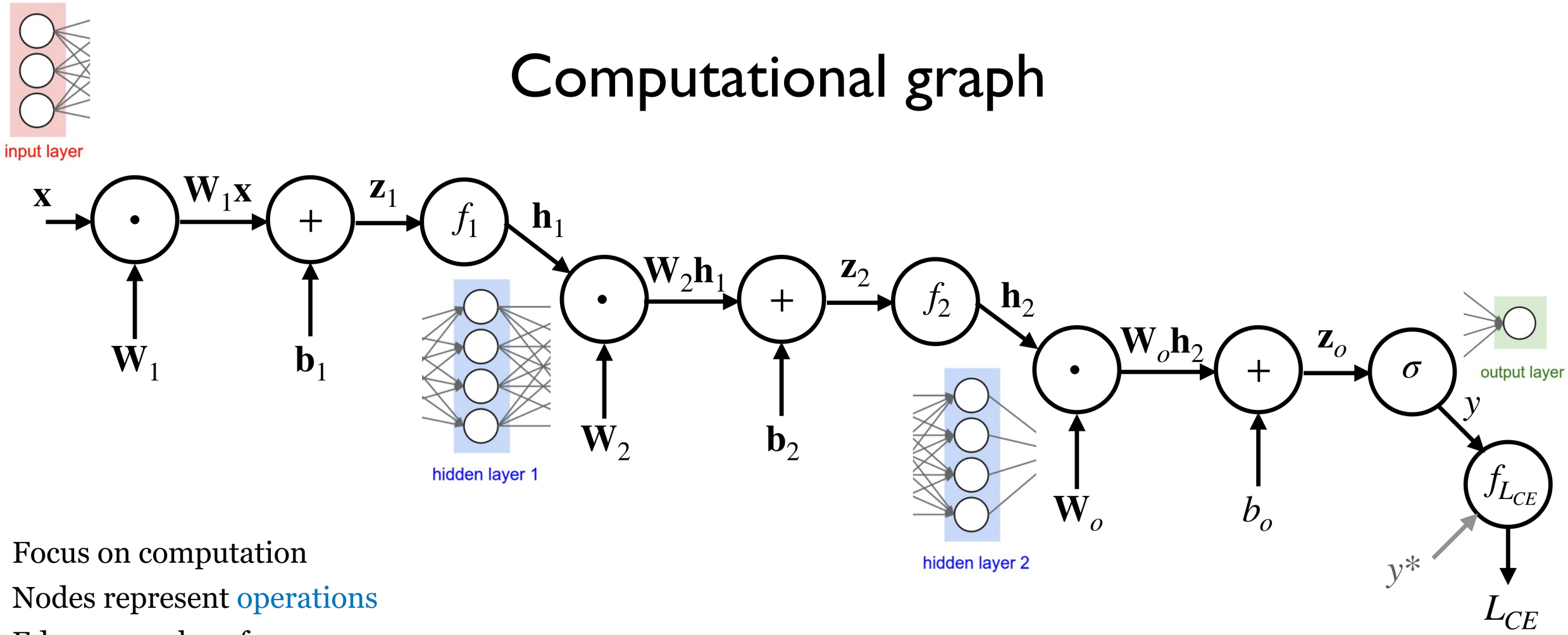
Gradient Descent

$$\theta^{(t+1)} = \theta^{(t)} - \eta \underline{\nabla_\theta \mathcal{L}(\theta)}$$

How to compute the gradient?

Neural networks as computational graphs

Computational graph



Focus on computation

Nodes represent operations

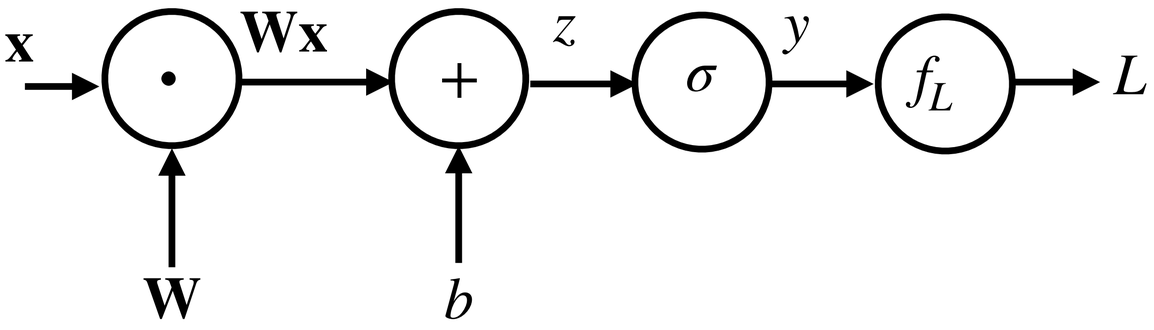
Edges are values from one operator to the next

Forward pass: compute function value

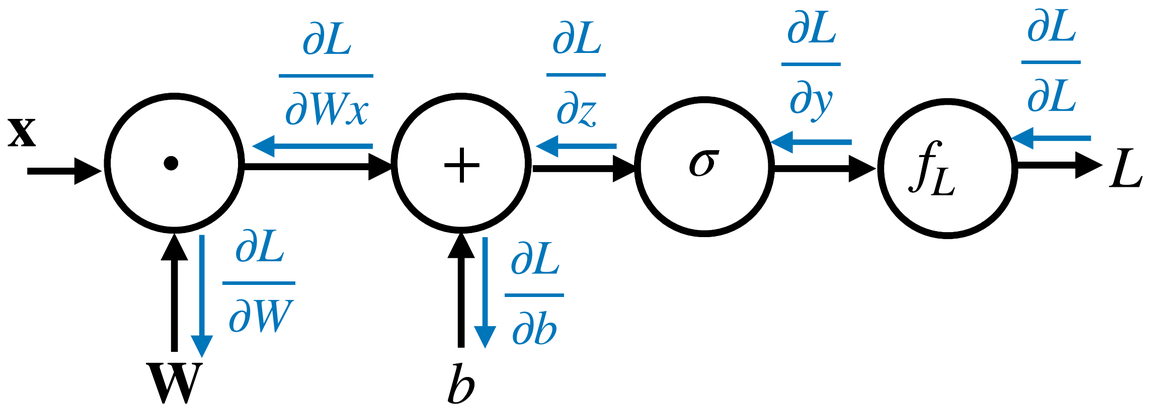
Backward pass: gradient using chain rule

Simplified example

Forward pass: compute function value



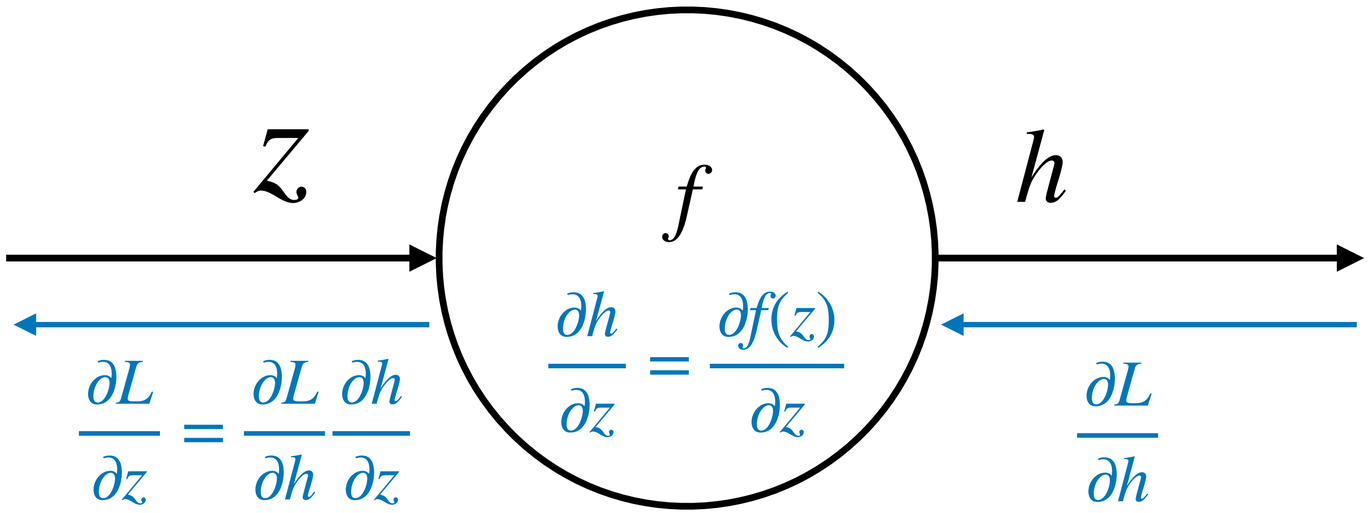
Backward pass: gradient using chain rule



- Good news is that modern automatic differentiation tools did all for you!
- Implementing backprop by hand is like programming in assembly language.

Backpropagation: single node

$$h = f(z)$$

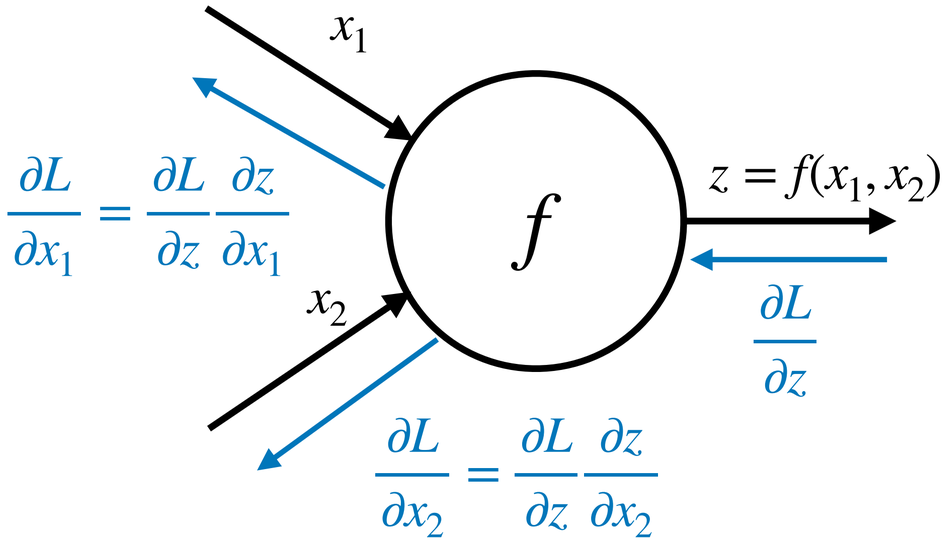


Downstream gradient = Local gradient × Upstream gradient

Gradient wrt input

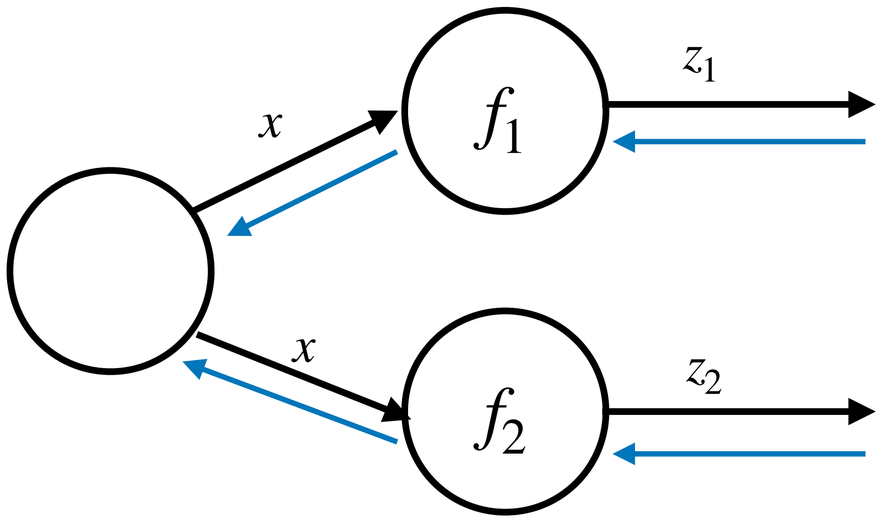
Backpropagation

Multiple inputs



Compute gradients for each input

Multiple output branches



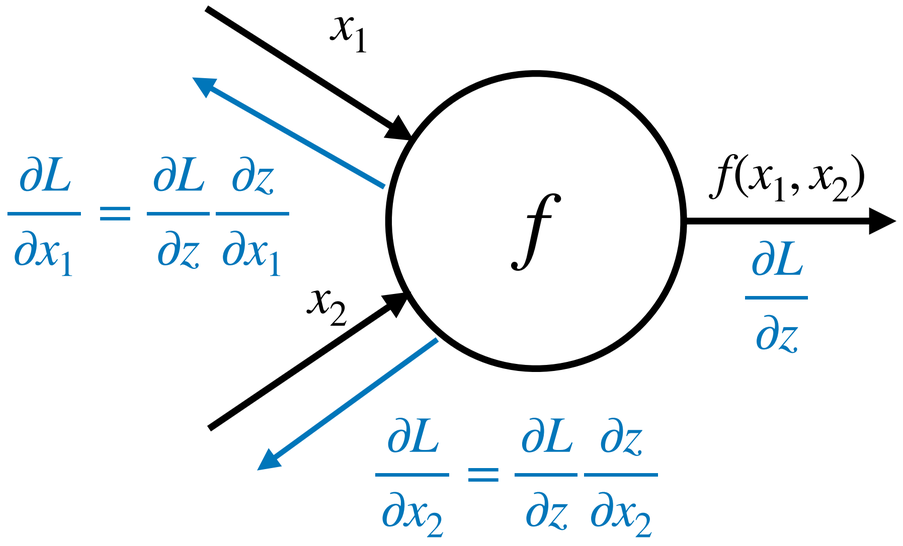
Sum gradients of branches

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

$$\{z_1, \ldots, z_n\} = \text{ successors of } x$$

Backpropagation API

Multiple inputs



Each node (operator) implement local forward/backward API

- forward(inputs)

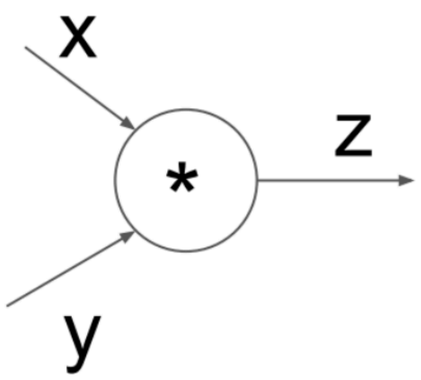
  $f(x_1, \ldots, x_k)$

- backward(upstream gradient)

  $$\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_k}$$

- Chain them together to form computation graph
- Reuse derivatives to minimize computation

Example: MultiplyGate



(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

http://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes03-neuralnets.pdf

Credits: Chris Manning (Stanford cs224n)

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

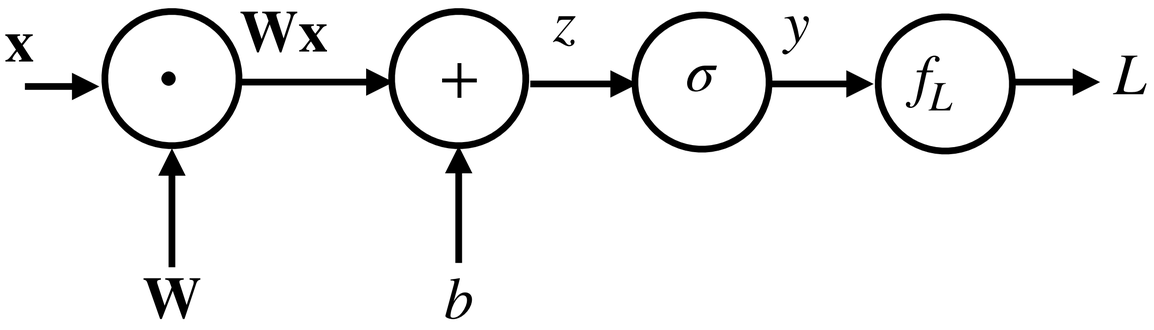
$$\frac{\partial L}{\partial x} = \sum_{i=1}^{n} \frac{\partial L}{\partial z_i} \frac{\partial z_i}{\partial x}$$

$$\{z_1, \ldots, z_n\} = \text{ successors of } x$$

For more details see https://colah.github.io/posts/2015-08-Backprop/

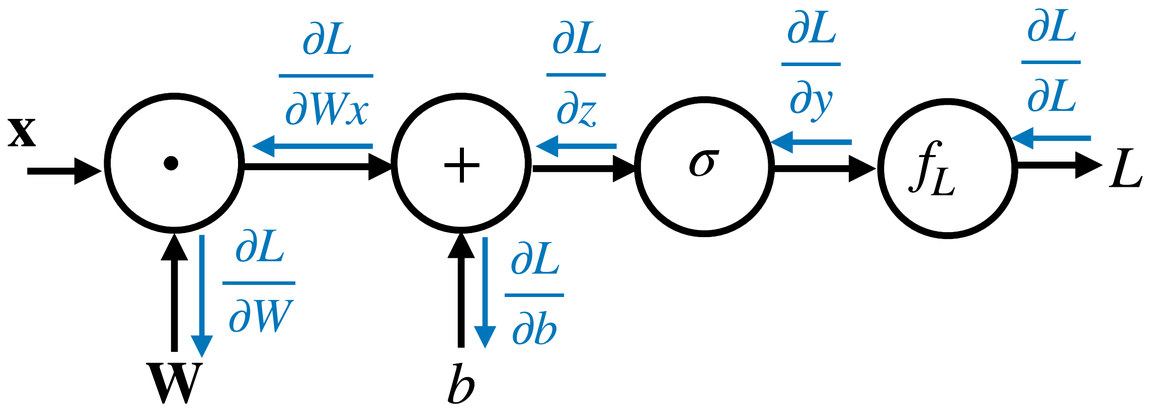
Simplified example

Forward pass: compute function value



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

Backward pass: gradient using chain rule



Chain rule

$$\frac{\partial L}{\partial L} = 1$$

$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial L}\frac{\partial f_L(y, y^*)}{\partial y}$$

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z} = \frac{\partial L}{\partial y}\frac{\partial \sigma(z)}{\partial z}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial b} = \frac{\partial L}{\partial z}\frac{\partial (\mathbf{W}\mathbf{x} + b)}{\partial b}$$

$$\frac{\partial L}{\partial \mathbf{W}\mathbf{x}} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial \mathbf{W}\mathbf{x}} = \frac{\partial L}{\partial z}\frac{\partial (\mathbf{W}\mathbf{x} + b)}{\partial \mathbf{W}\mathbf{x}}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{W}\mathbf{x}}\frac{\partial \mathbf{W}\mathbf{x}}{\partial \mathbf{W}}$$

Local derivatives

$$\frac{\partial f_L(y, y^*)}{\partial y} = y - y^*$$

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z))$$

$$\frac{\partial (\mathbf{W}\mathbf{x} + b)}{\partial b} = 1$$

$$\frac{\partial (\mathbf{W}\mathbf{x} + b)}{\partial \mathbf{W}\mathbf{x}} = 1$$

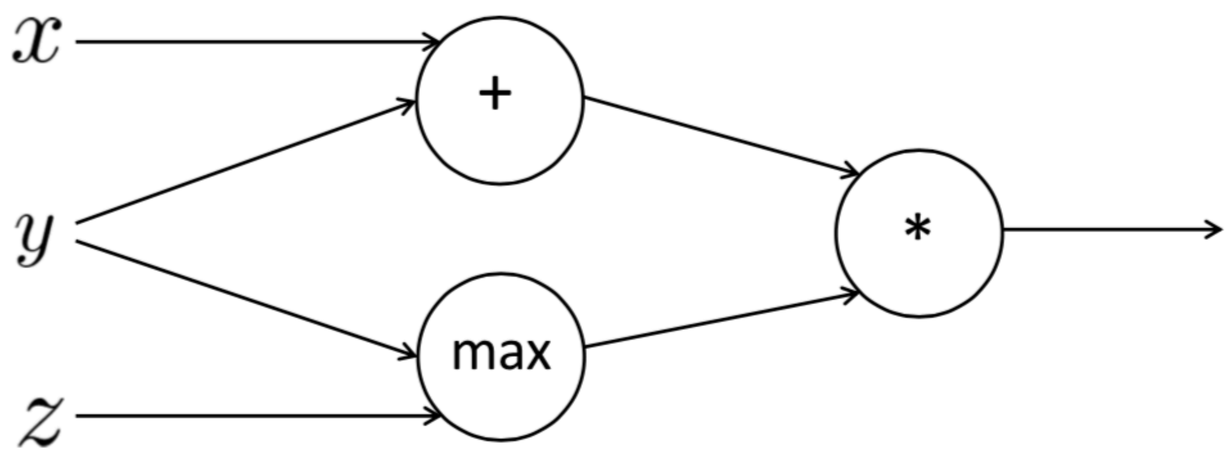
$$\frac{\partial \mathbf{W}\mathbf{x}}{\partial \mathbf{W}} = \mathbf{x}^T$$

An example

Try to compute the gradients yourself!

$$\boxed{\begin{aligned} &f(x, y, z) = (x + y)\max(y, z) \\ &x = 1, y = 2, z = 0 \end{aligned}}$$

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



http://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes03-neuralnets.pdf

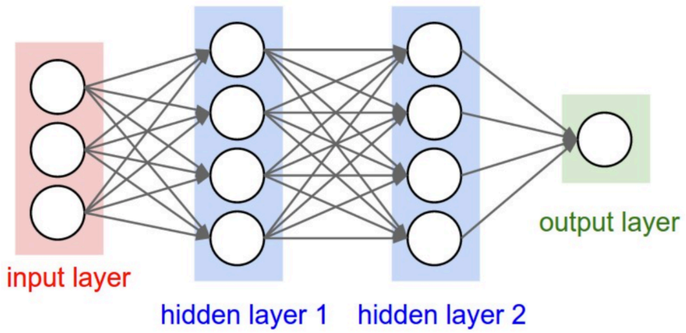
Summary of backpropagation

- 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.

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 $P(y = c \mid x)$ for each class c
  - Need to model $P(y = c \mid x)$ 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. batching, 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

Resources

- There is a lot more to learn about neural networks!
  - TA Tutorials
    - Optimization
    - Pytorch and backpropagation
    - Debugging Tips and Tricks
  - Classes: CMPT 728 (Deep Learning)

Resources

- Deep learning books
  - Courville, Goodfellow, and Bengio: https://www.deeplearningbook.org/
  - Yoav Goldberg's Primer on NN models for NLP: http://u.cs.biu.ac.il/~yogo/nnlp.pdf
- Classes from other universities
  - Stanford CS231n notes: https://cs231n.github.io/
  - University of Toronto: https://csc413-2020.github.io/ (Jimmy Ba and Roger Grosse)
  - University of Michigan: https://web.eecs.umich.edu/~justincj/teaching/eecs498/FA2020/ (Justin Johnson)