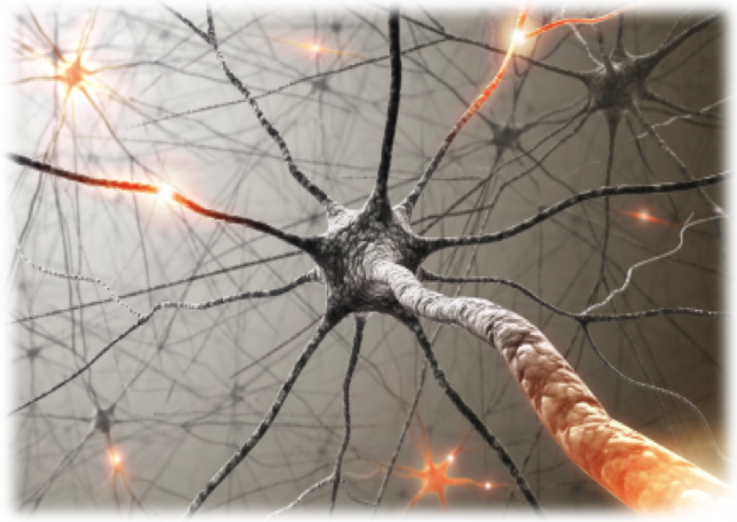
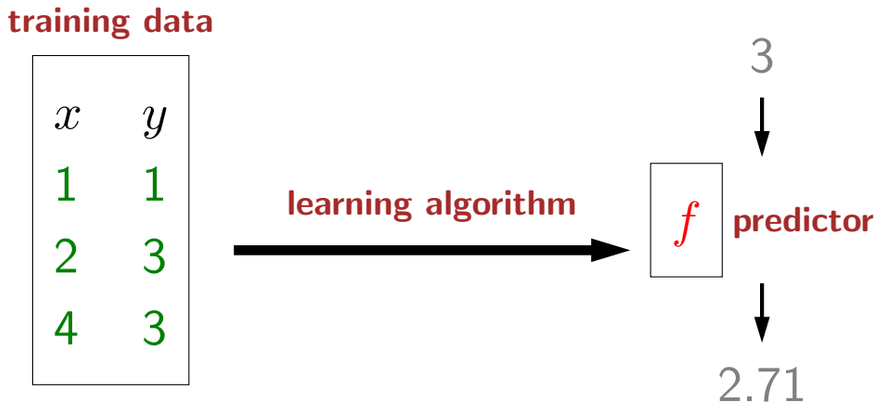
Machine learning: non-linear features



Linear regression



Which predictors are possible?

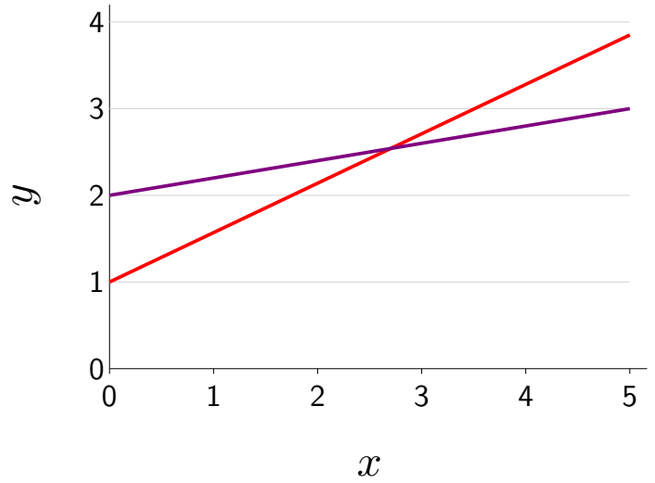
Hypothesis class

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^d\}$$

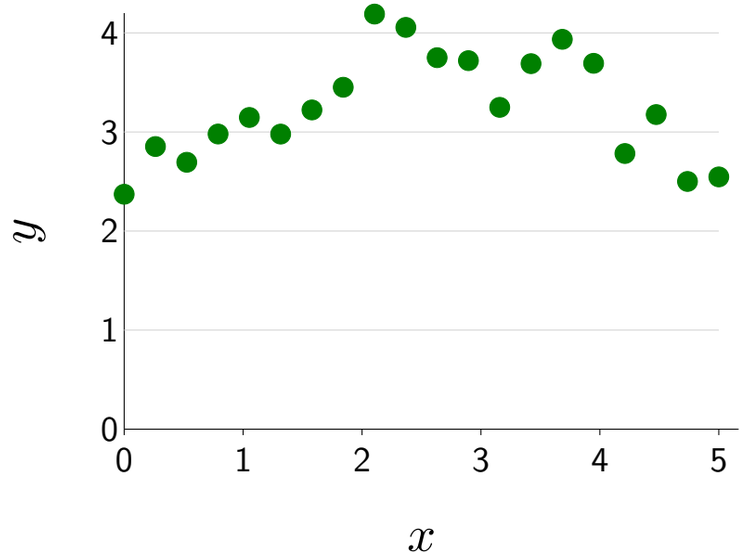
$\phi(x) = [1, x]$

$f(x) = [1, 0.57] \cdot \phi(x)$

$f(x) = [2, 0.2] \cdot \phi(x)$



More complex data



How do we fit a non-linear predictor?

Quadratic predictors

$$\phi(x) = [1, x, x^2]$$

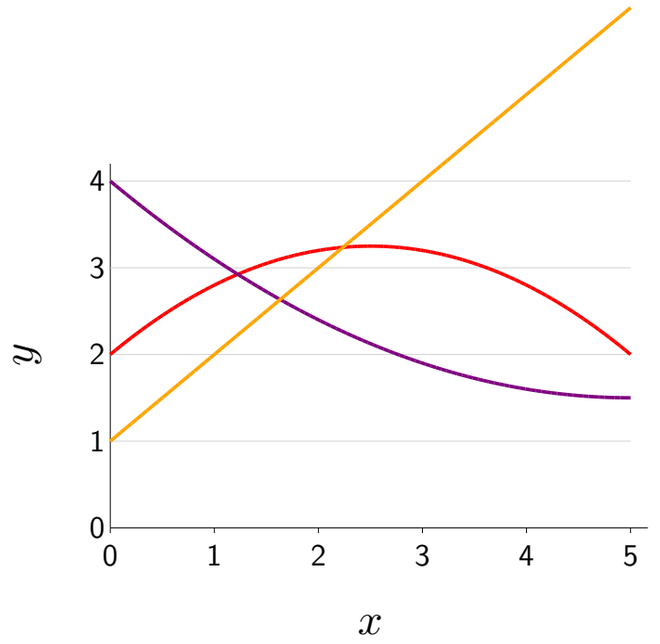
Example: $\phi(3) = [1, 3, 9]$

$f(x) = [2, 1, -0.2] \cdot \phi(x)$

$f(x) = [4, -1, 0.1] \cdot \phi(x)$

$f(x) = [1, 1, 0] \cdot \phi(x)$

$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^3\}$



Non-linear predictors just by changing ϕ

Piecewise constant predictors

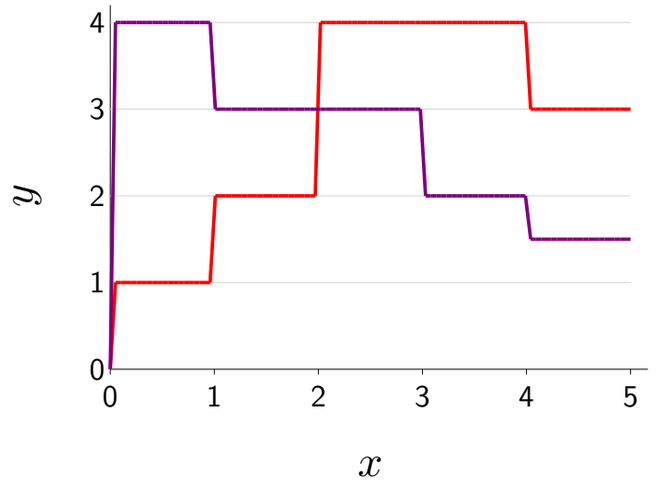
$$\phi(x) = [\mathbf{1}[0 < x \leq 1], \mathbf{1}[1 < x \leq 2], \mathbf{1}[2 < x \leq 3], \mathbf{1}[3 < x \leq 4], \mathbf{1}[4 < x \leq 5]]$$

Example: $\phi(2.3) = [0, 0, 1, 0, 0]$

$$f(x) = [1, 2, 4, 4, 3] \cdot \phi(x)$$

$$f(x) = [4, 3, 3, 2, 1.5] \cdot \phi(x)$$

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^5\}$$



Expressive non-linear predictors by partitioning the input space

Predictors with periodicity structure

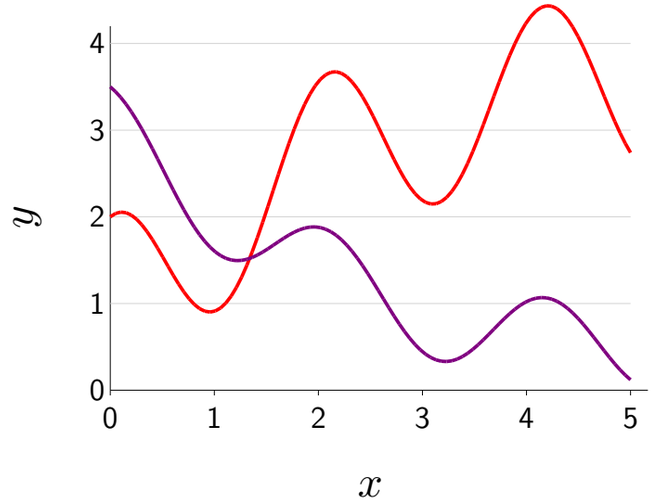
$$\phi(x) = [1, x, x^2, \cos(3x)]$$

Example: $\phi(2) = [1, 2, 4, 0.96]$

$f(x) = [1, 1, -0.1, 1] \cdot \phi(x)$

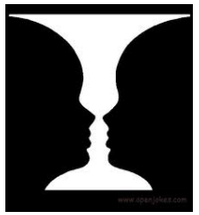
$f(x) = [3, -1, 0.1, 0.5] \cdot \phi(x)$

$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^4\}$



Just throw in any features you want

Linear in what?



Prediction:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

Linear in $\mathbf{w}$? Yes

Linear in $\phi(x)$? Yes

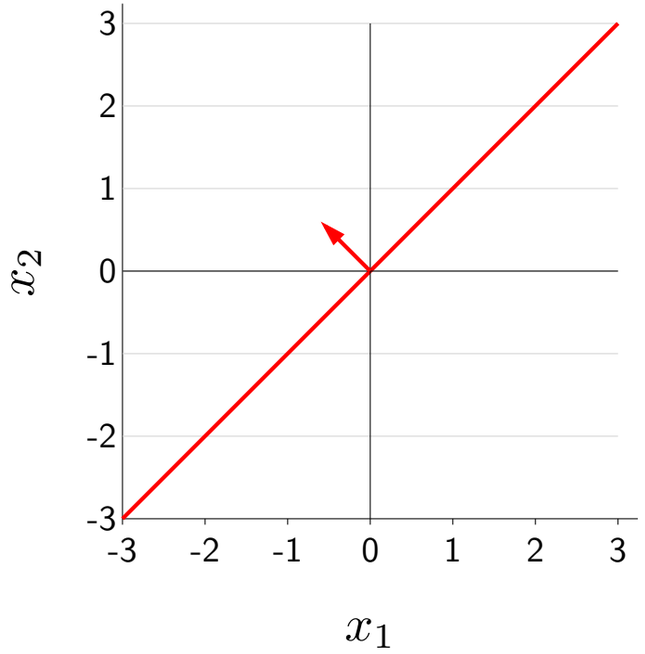
Linear in x? No!

Key idea: non-linearity

- Expressivity: score $\mathbf{w} \cdot \phi(x)$ can be a **non-linear** function of x
- Efficiency: score $\mathbf{w} \cdot \phi(x)$ always a **linear** function of $\mathbf{w}$

Linear classification

$$\phi(x) = [x_1, x_2]$$
$$f(x) = \text{sign}([-0.6, 0.6] \cdot \phi(x))$$



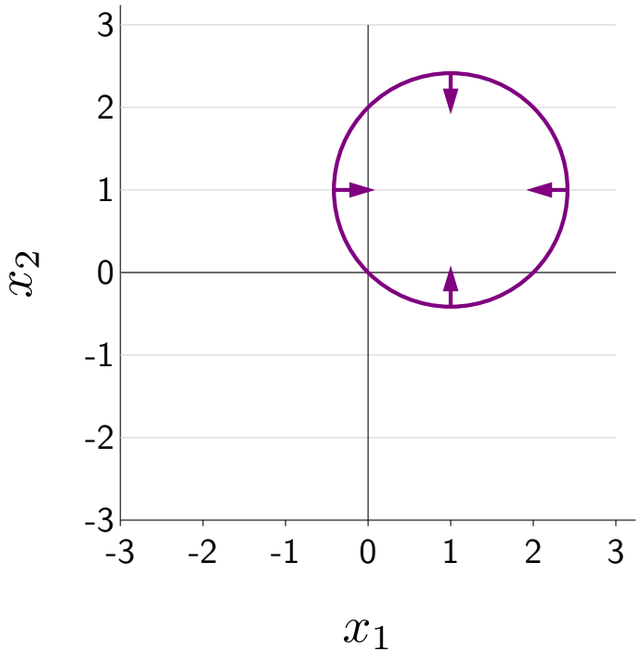
Decision boundary is a line

Quadratic classifiers

$$\phi(x) = [x_1, x_2, x_1^2 + x_2^2]$$
$$f(x) = \text{sign}([2, 2, -1] \cdot \phi(x))$$

Equivalently:

$$f(x) = \begin{cases} 1 & \text{if } \{(x\_1 - 1)^2 + (x\_2 - 1)^2 \leq 2\} \\ -1 & \text{otherwise} \end{cases}$$

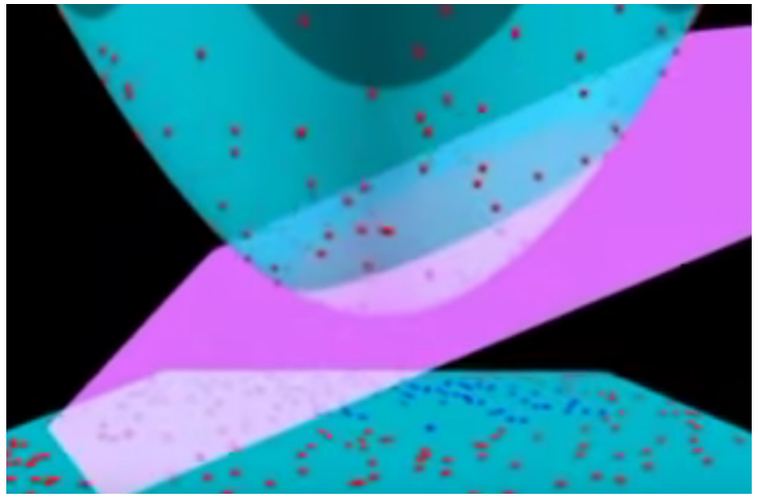


Decision boundary is a circle

Visualization in feature space

Input space: $x = [x_1, x_2]$, decision boundary is a circle

Feature space: $\phi(x) = [x_1, x_2, x_1^2 + x_2^2]$, decision boundary is a line

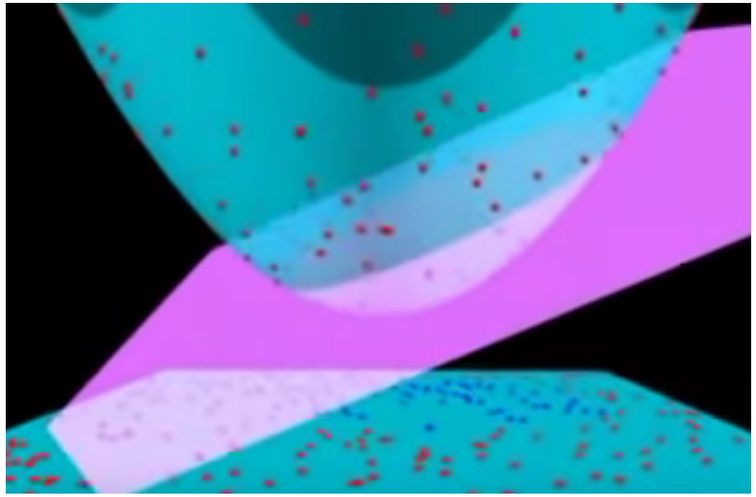


Summary

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

linear in $\mathbf{w}, \phi(x)$

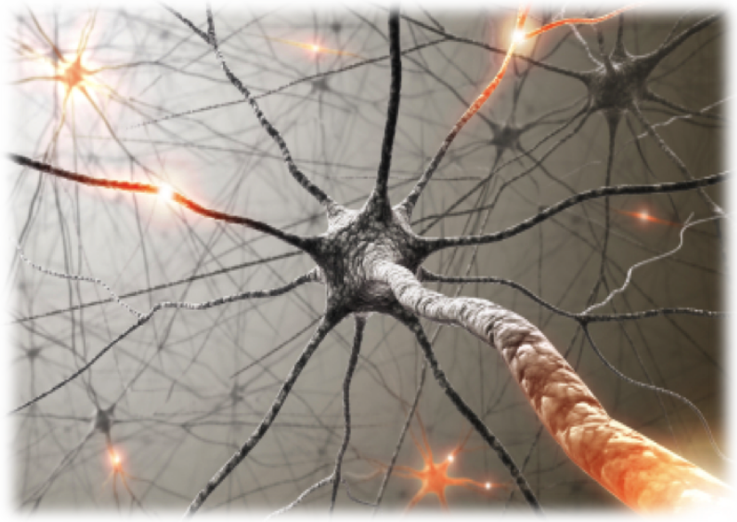
non-linear in x



- Regression: non-linear predictor, classification: non-linear decision boundary
- Types of non-linear features: quadratic, piecewise constant, etc.

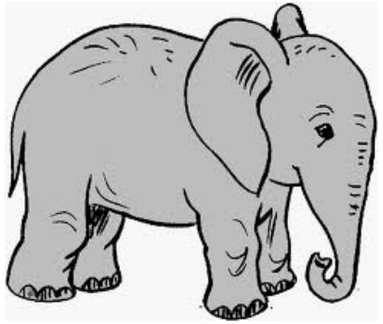
Non-linear predictors with linear machinery

Machine learning: stochastic gradient descent



Gradient descent is slow

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$



Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

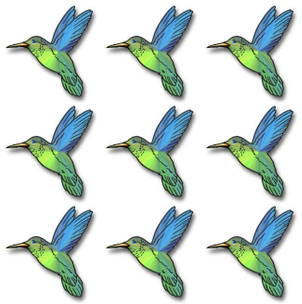
For $t = 1, \ldots, T$:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$

Problem: each iteration requires going over all training examples — expensive when have lots of data!

Stochastic gradient descent

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$



Algorithm: stochastic gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

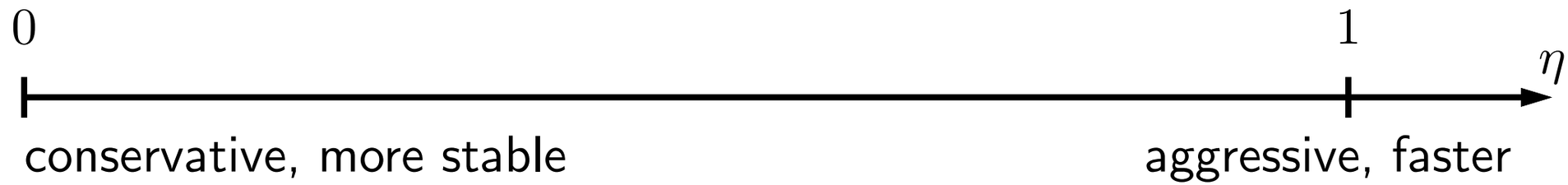
  For $(x, y) \in \mathcal{D}_{\text{train}}$:

    $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$

Step size

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$$

Question: what should η be?

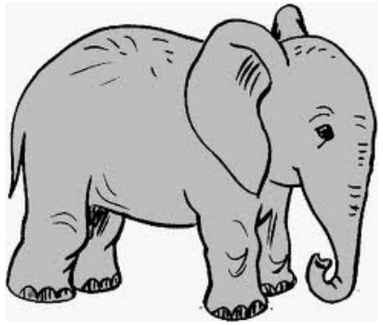


Strategies:

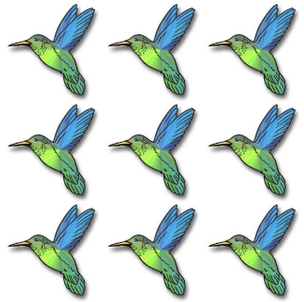
- Constant: $\eta = 0.1$
- Decreasing: $\eta = 1/\sqrt{\#\text{ updates made so far}}$

Summary

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$



gradient descent

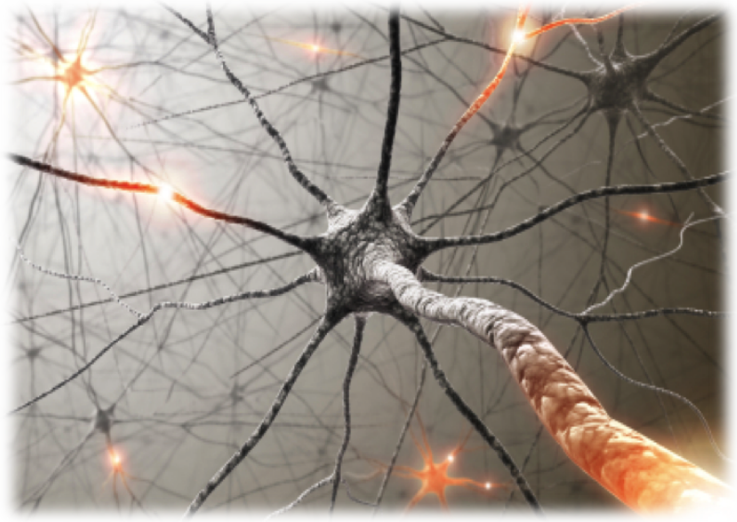


stochastic gradient descent

Key idea: stochastic updates

It's not about **quality**, it's about **quantity**.

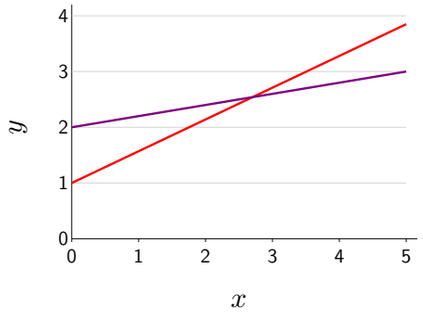
Machine learning: neural networks



Non-linear predictors

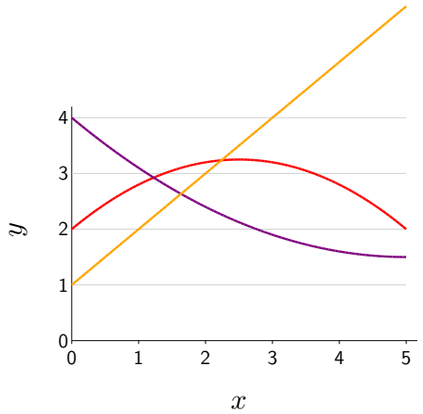
Linear predictors:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x), \ \phi(x) = [1, x]$$



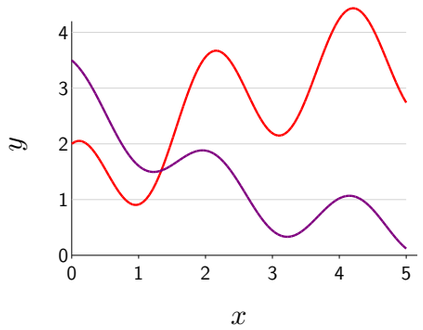
Non-linear (quadratic) predictors:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x), \ \phi(x) = [1, x, x^2]$$



Non-linear neural networks:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \sigma(\mathbf{V}\phi(x)), \ \phi(x) = [1, x]$$



Motivating example

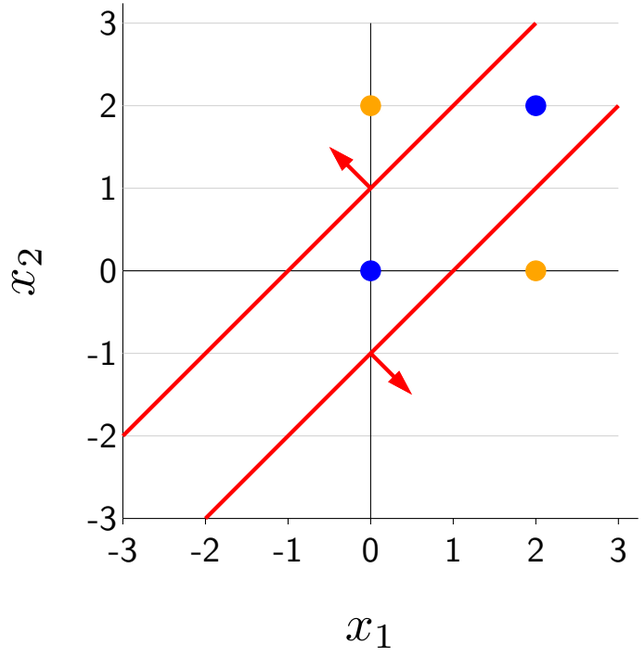
Example: predicting car collision

Input: positions of two oncoming cars $x = [x_1, x_2]$

Output: whether safe ($y = +1$) or collide ($y = -1$)

Unknown: safe if cars sufficiently far: $y = \text{sign}(|x_1 - x_2| - 1)$

x_1	x_2	y
0	2	1
2	0	1
0	0	-1
2	2	-1



Decomposing the problem

Test if car 1 is far right of car 2:

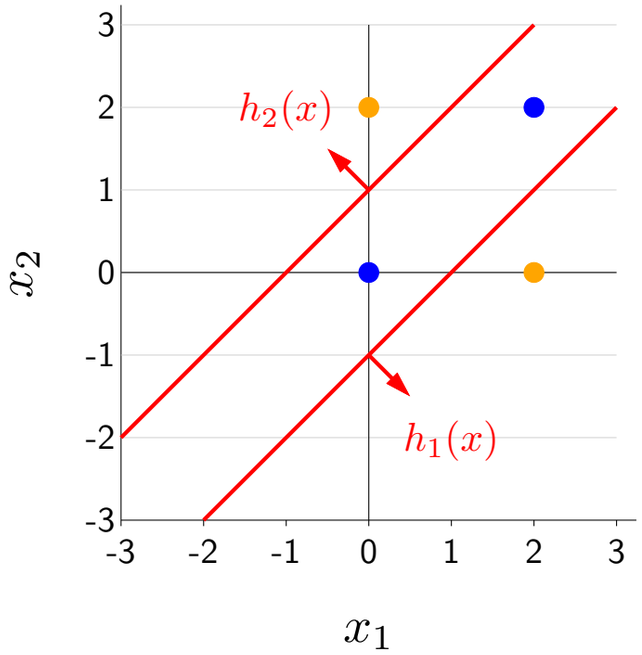
$$h_1(x) = \mathbf{1}[x_1 - x_2 \geq 1]$$

Test if car 2 is far right of car 1:

$$h_2(x) = \mathbf{1}[x_2 - x_1 \geq 1]$$

Safe if at least one is true:

$$f(x) = \mathsf{sign}(h_1(x) + h_2(x))$$



x	$h_1(x)$	$h_2(x)$	$f(x)$
$[0, 2]$	0	1	$+1$
$[2, 0]$	1	0	$+1$
$[0, 0]$	0	0	-1
$[2, 2]$	0	0	-1

Rewriting using vector notation

Intermediate subproblems:

$$h_1(x) = \mathbf{1}[x_1 - x_2 \geq 1] = \mathbf{1}[[-1, +1, -1] \cdot [1, x_1, x_2] \geq 0]$$

$$h_2(x) = \mathbf{1}[x_2 - x_1 \geq 1] = \mathbf{1}[[-1, -1, +1] \cdot [1, x_1, x_2] \geq 0]$$

$$\mathbf{h}(x) = \mathbf{1}\left[\begin{bmatrix} -1 & +1 & -1 \\ -1 & -1 & +1 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \geq 0\right]$$

Predictor:

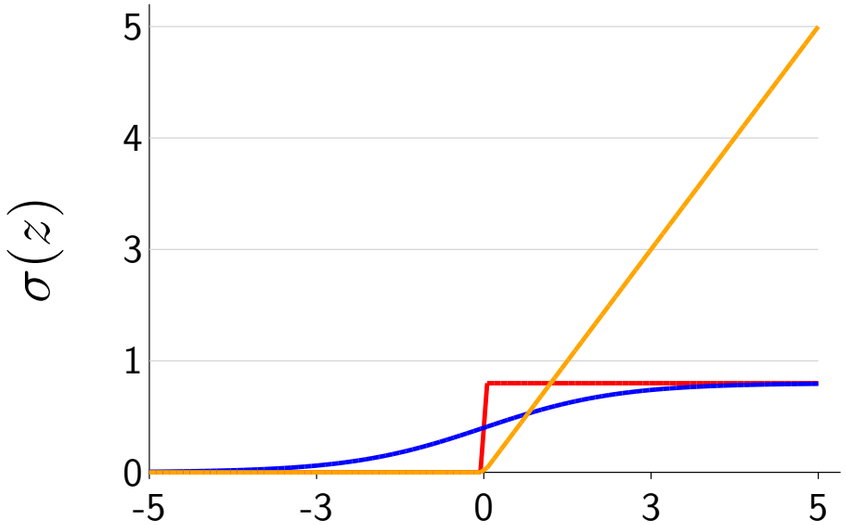
$$f(x) = \text{sign}(h_1(x) + h_2(x)) = \text{sign}([1, 1] \cdot \mathbf{h}(x))$$

Avoid zero gradients

Problem: gradient of $h_1(x)$ with respect to $\mathbf{v}_1$ is 0

$$h_1(x) = \mathbf{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0]$$

Solution: replace with an **activation function** σ with non-zero gradients



$$z = \mathbf{v}_1 \cdot \phi(x)$$

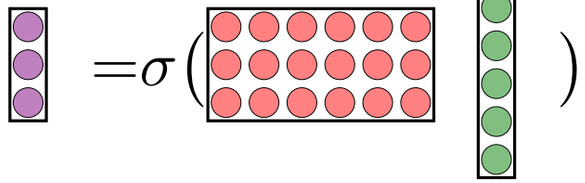
- Threshold: $\mathbf{1}[z \geq 0]$
- Logistic: $\frac{1}{1+e^{-z}}$
- ReLU: $\max(z, 0)$

$$h_1(x) = \sigma(\mathbf{v}_1 \cdot \phi(x))$$

Two-layer neural networks

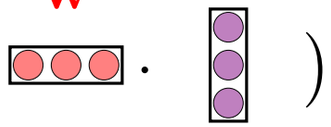
Intermediate subproblems:

$$\mathbf{h}(x) = \sigma\left(\mathbf{V}\ \phi(x)\right)$$



Predictor (classification):

$$f_{\mathbf{V},\mathbf{w}}(x) = \text{sign}\left(\mathbf{w} \cdot \mathbf{h}(x)\right)$$



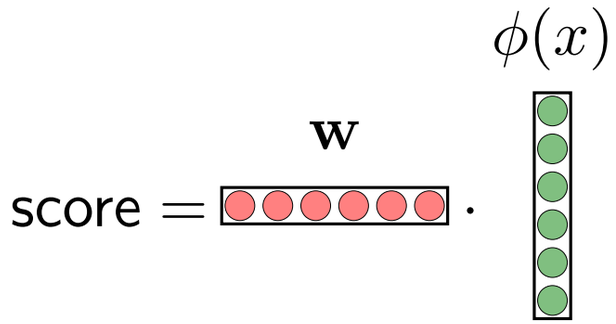
Interpret $\mathbf{h}(x)$ as a learned feature representation!

Hypothesis class:

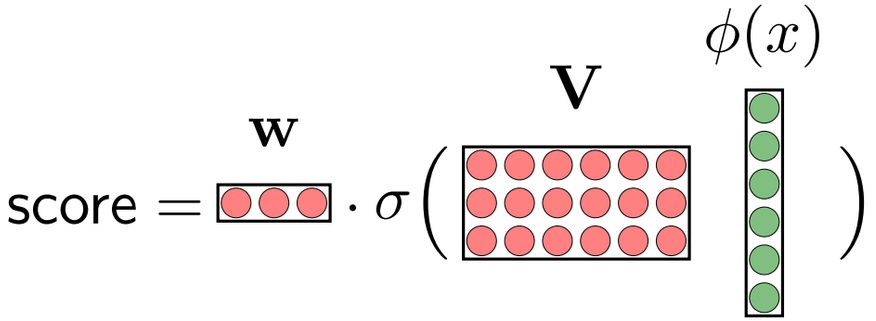
$$\mathcal{F} = \{f_{\mathbf{V},\mathbf{w}} : \mathbf{V} \in \mathbb{R}^{k \times d}, \mathbf{w} \in \mathbb{R}^k\}$$

Deep neural networks

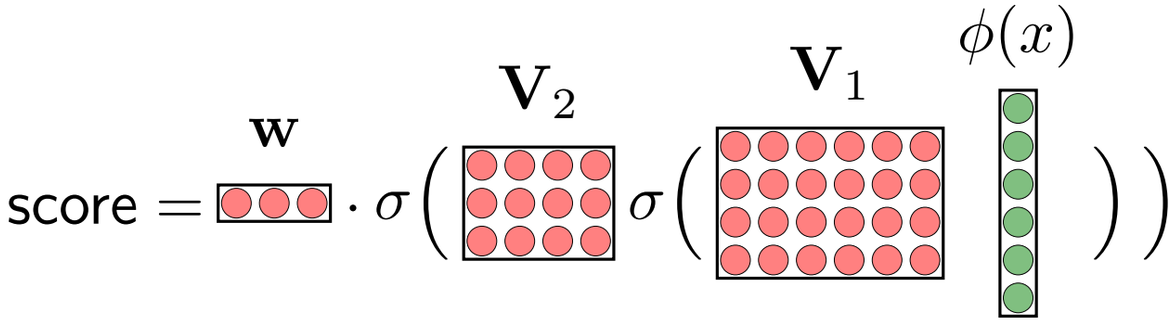
1-layer neural network:



2-layer neural network:

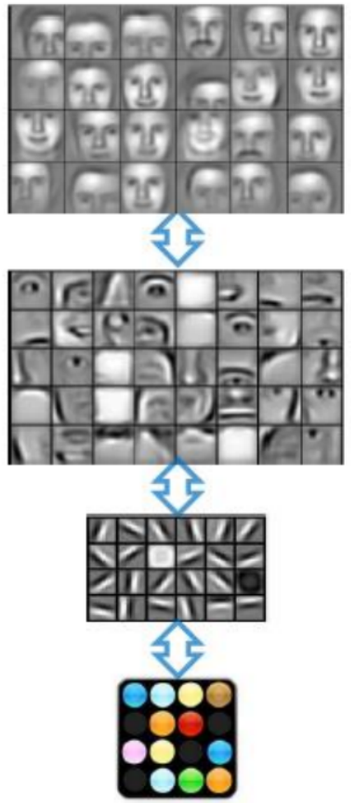


3-layer neural network:



[figure from Honglak Lee]

Layers represent multiple levels of abstractions



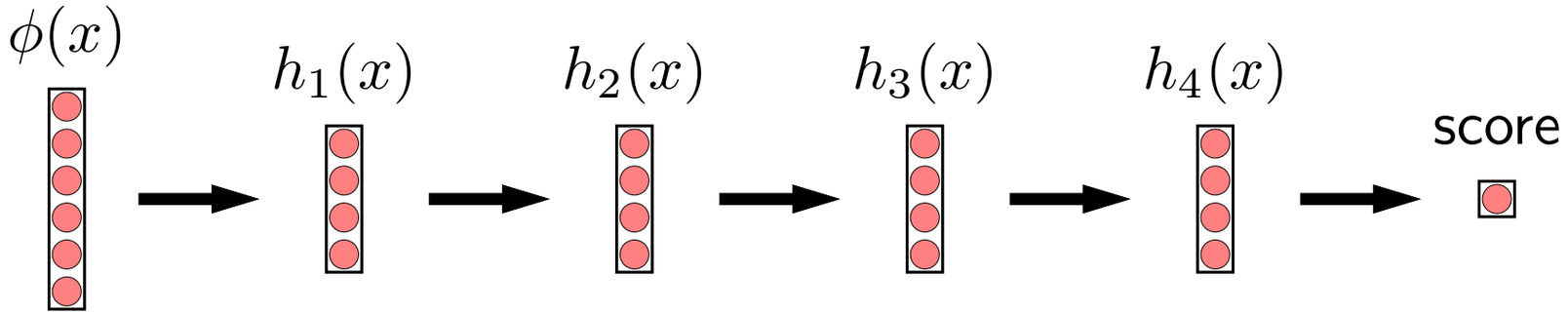
3rd layer
"Objects"

2nd layer
"Object parts"

1st layer
"Edges"

Pixels

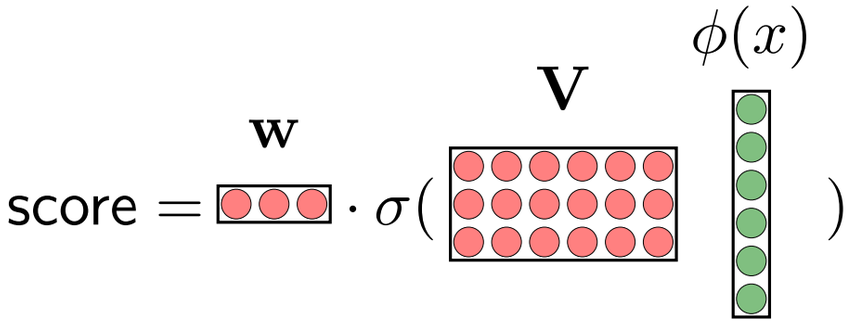
Why depth?



Intuitions:

- Multiple levels of abstraction
- Multiple steps of computation
- Empirically works well
- Theory is still incomplete

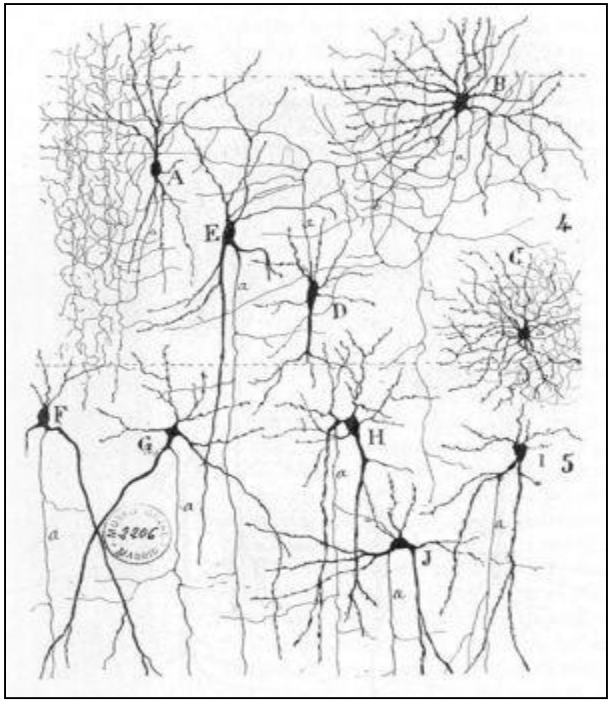
Summary



- Intuition: decompose problem into intermediate parallel subproblems
- Deep networks iterate this decomposition multiple times
- Hypothesis class contains predictors ranging over weights for all layers
- Next up: learning neural networks

An Introduction to Neural Networks

The origin: To make a computer "think" and "learn", maybe we should start by trying to understand how the human brain think and learn.



Connectionist model: Complicated functions can be described by a set of interconnected simple units.

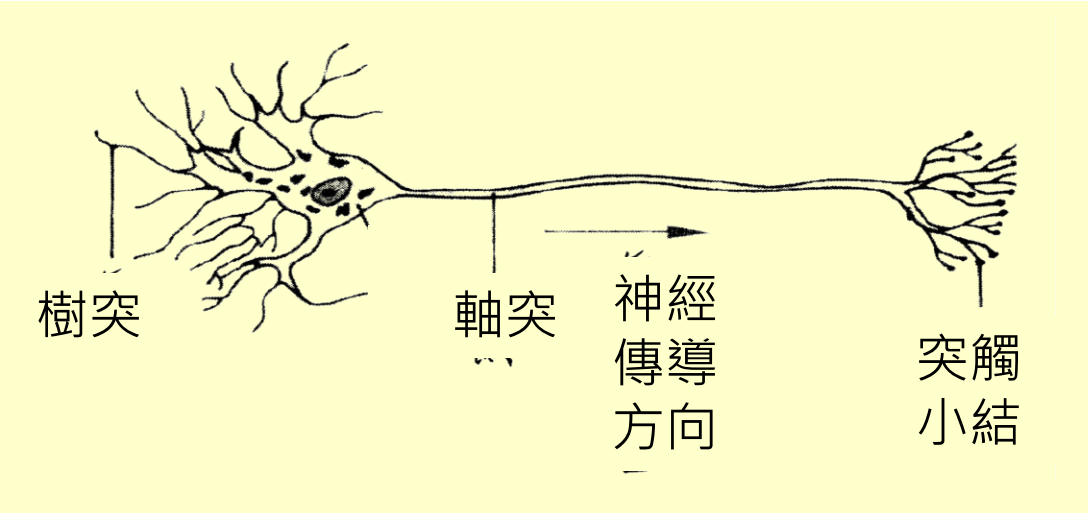
From "Texture of the Nervous System of Man and the Vertebrates" by Santiago Ramón y Cajal. The figure illustrates the diversity of neuronal morphologies in the auditory cortex.

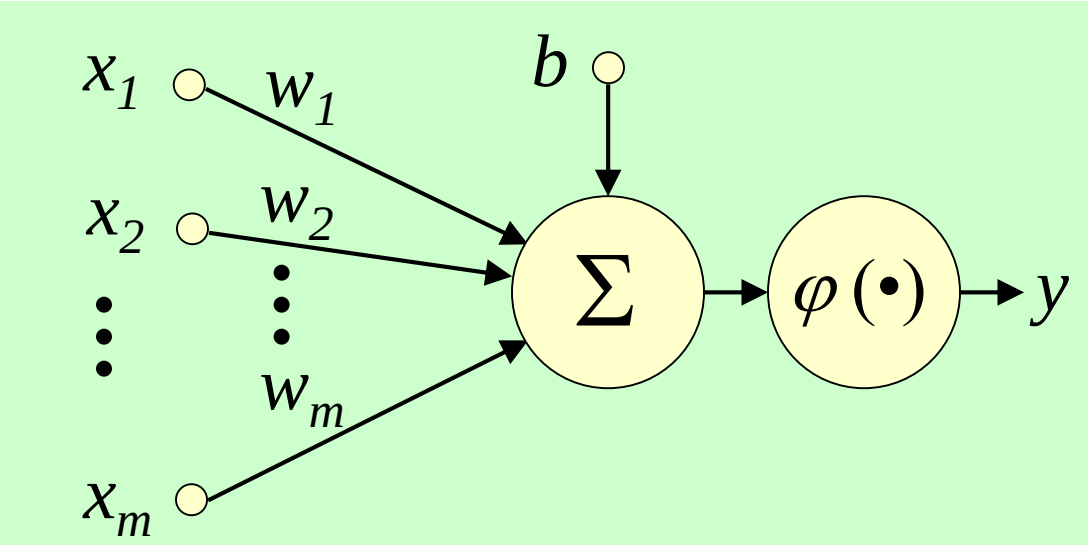
This is a model that spans cognitive psychology, neuroscience, philosophy, and computer science.

Biological vs. Artificial Neurons

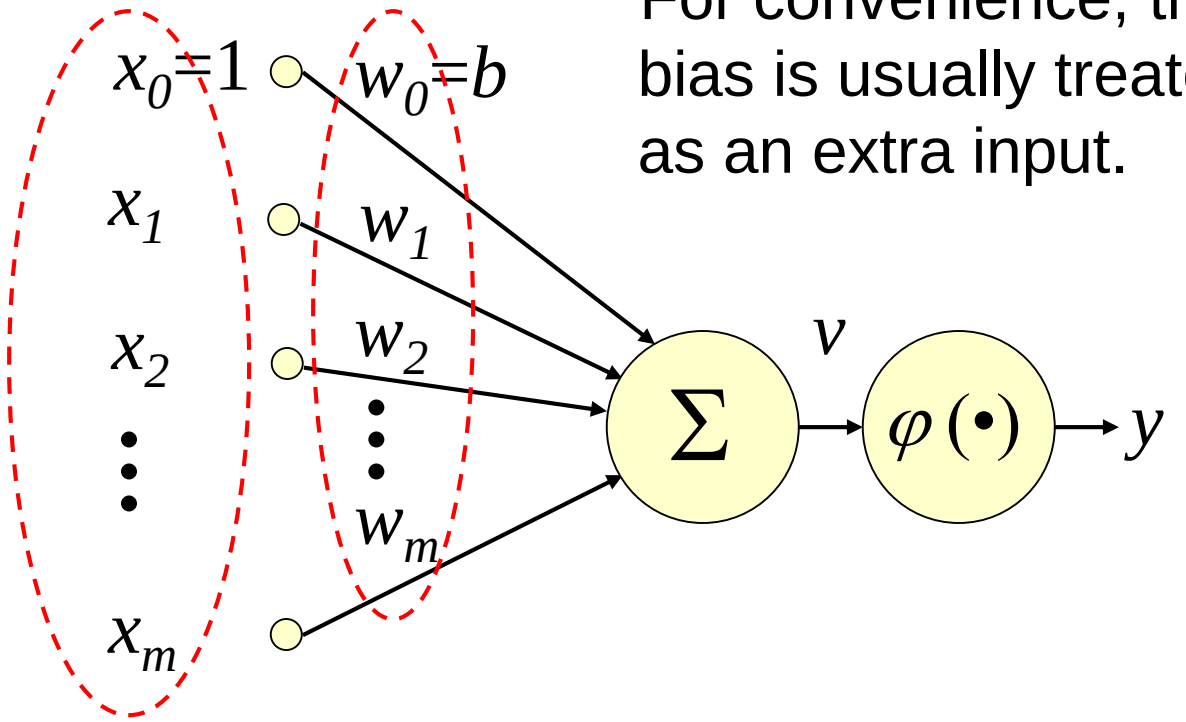
Important components:

- Inputs
- Synaptic weights
- Bias
- Summing function
- Activation function
- Output





Important Notations



For convenience, the bias is usually treated as an extra input.

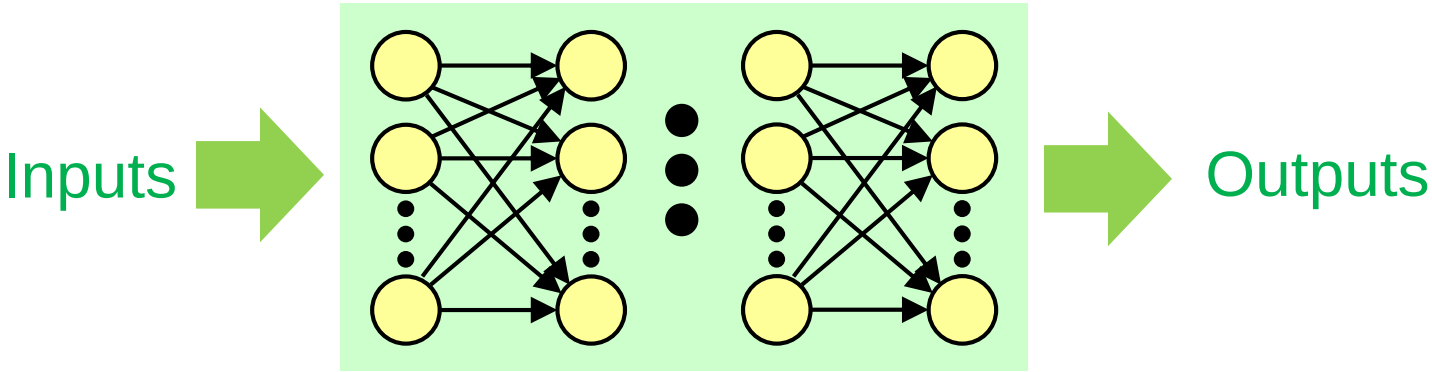
$\boldsymbol{x}$: The (column) vector of all the inputs

$\boldsymbol{w}$: The (column) vector of all the synaptic weights of a neuron

The combined input: $v = \boldsymbol{w}^T \boldsymbol{x}$

Neural Network Architectures

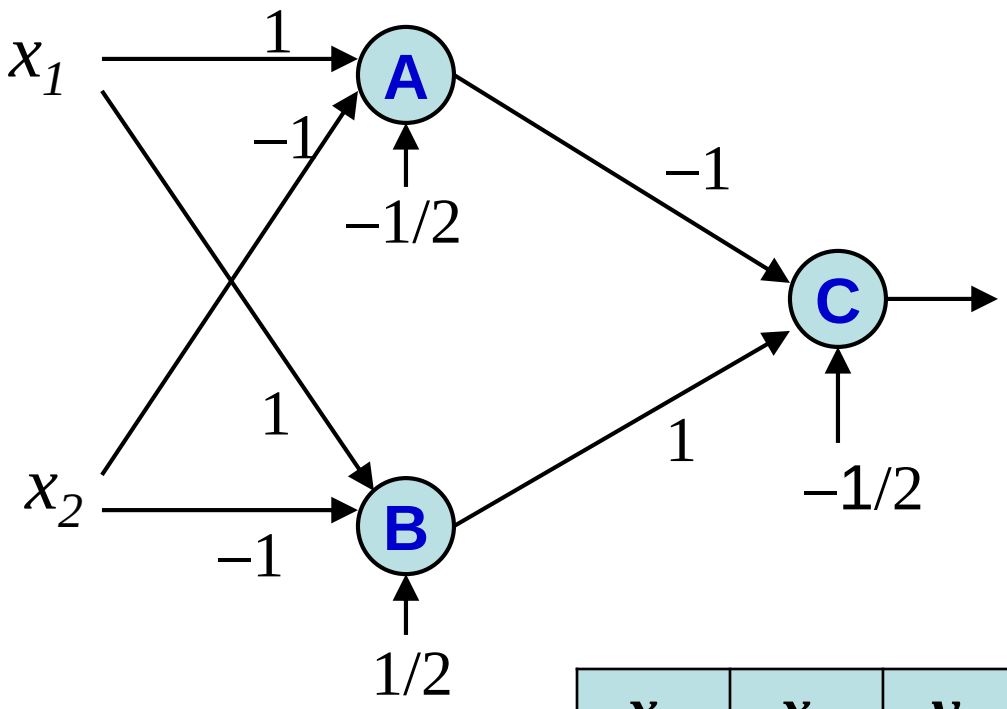
An artificial neural network is made of a directed graph of a set of neurons:



Two main categories of neural network architectures:

- Feedforward NNs: Signals flow in one direction only.
- Recurrent NNs: Both feedforward and feedback links ➔ Have the effect of "state memory" ➔ More suitable for processing sequence/dynamic data. (Examples: language, speech, handwriting, etc.)

A Simple Feedforward NN

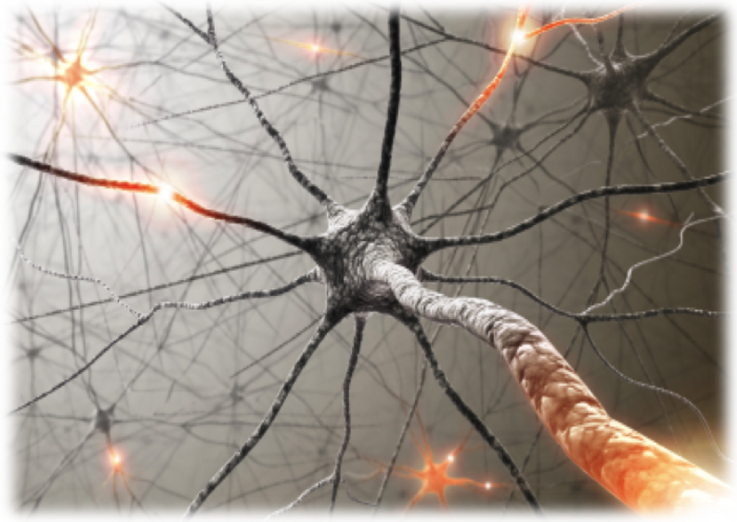


Activation function:

$$\varphi(v) = \begin{cases} 1 & \text{for } v > 0 \\ 0 & \text{otherwise} \end{cases}$$

x_1	x_2	v_A	y_A	v_B	y_B	v_C	y_C
0	0	−1/2	0	1/2	1	1/2	1
1	0	1/2	1	3/2	1	−1/2	0
0	1	−3/2	0	−1/2	0	−1/2	0
1	1	−1/2	0	1/2	1	1/2	1

Machine learning: backpropagation



Motivation: regression with four-layer neural networks

Loss on one example:

$$\text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}_3 \sigma(\mathbf{V}_2 \sigma(\mathbf{V}_1 \phi(x)))) - y)^2$$

Stochastic gradient descent:

$$\mathbf{V}_1 \leftarrow \mathbf{V}_1 - \eta \nabla_{\mathbf{V}_1} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{V}_2 \leftarrow \mathbf{V}_2 - \eta \nabla_{\mathbf{V}_2} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

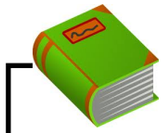
$$\mathbf{V}_3 \leftarrow \mathbf{V}_3 - \eta \nabla_{\mathbf{V}_3} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

How to get the gradient without doing manual work?

Computation graphs

$$\text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}_3 \sigma(\mathbf{V}_2 \sigma(\mathbf{V}_1 \phi(x)))) - y)^2$$



Definition: computation graph

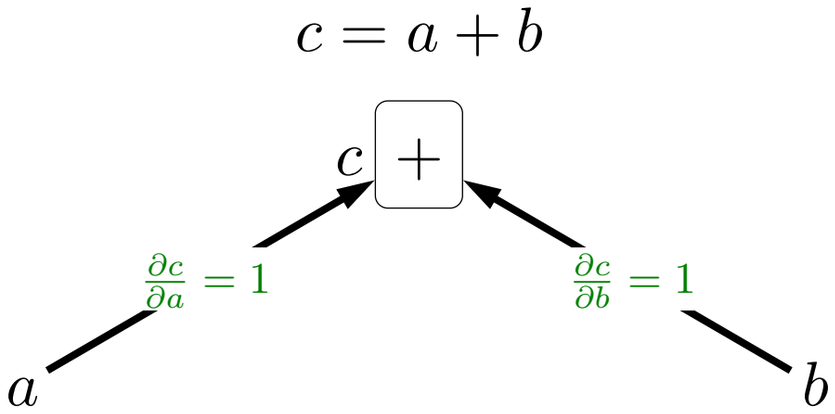
A directed acyclic graph whose root node represents the final mathematical expression and each node represents intermediate subexpressions.

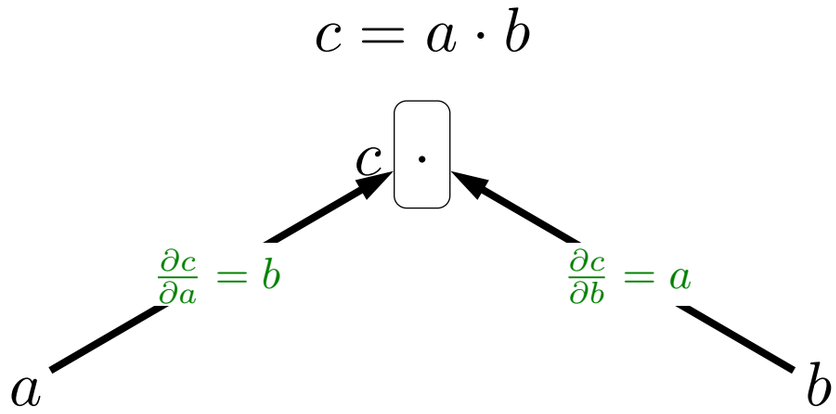
Upshot: compute gradients via general **backpropagation** algorithm

Purposes:

- Automatically compute gradients (how TensorFlow and PyTorch work)
- Gain insight into modular structure of gradient computations

Functions as boxes

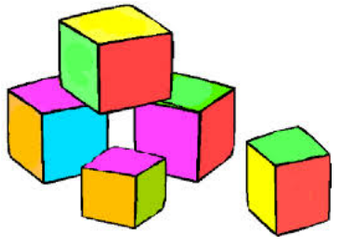




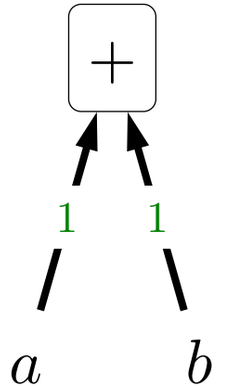
$$(a + \epsilon) + b = c + 1\epsilon$$
$$a + (b + \epsilon) = c + 1\epsilon$$

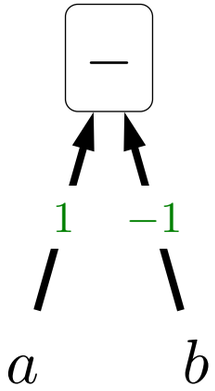
$$(a + \epsilon)b = c + b\epsilon$$
$$a(b + \epsilon) = c + a\epsilon$$

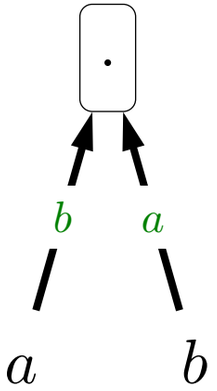
Gradients: how much does c change if a or b changes?

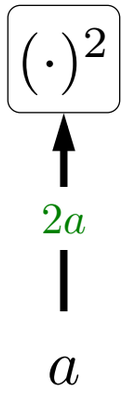


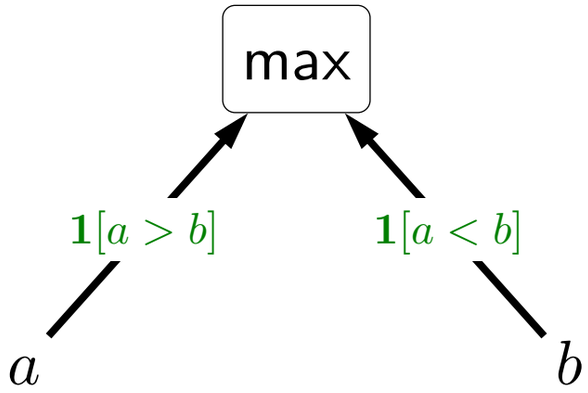
Basic building blocks

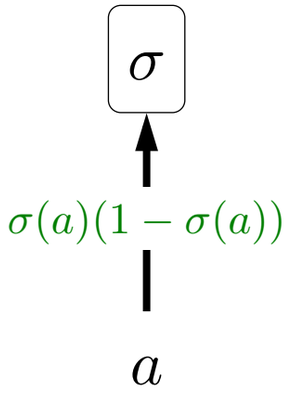




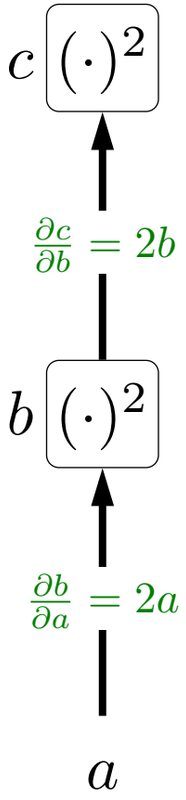








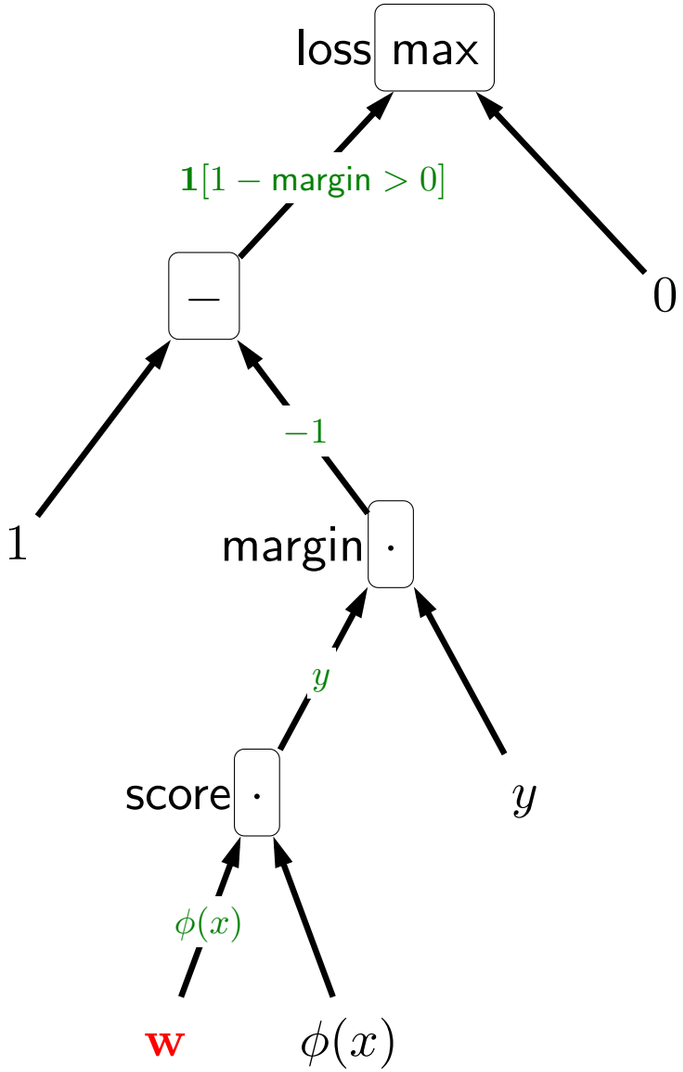
Function composition



Chain rule:

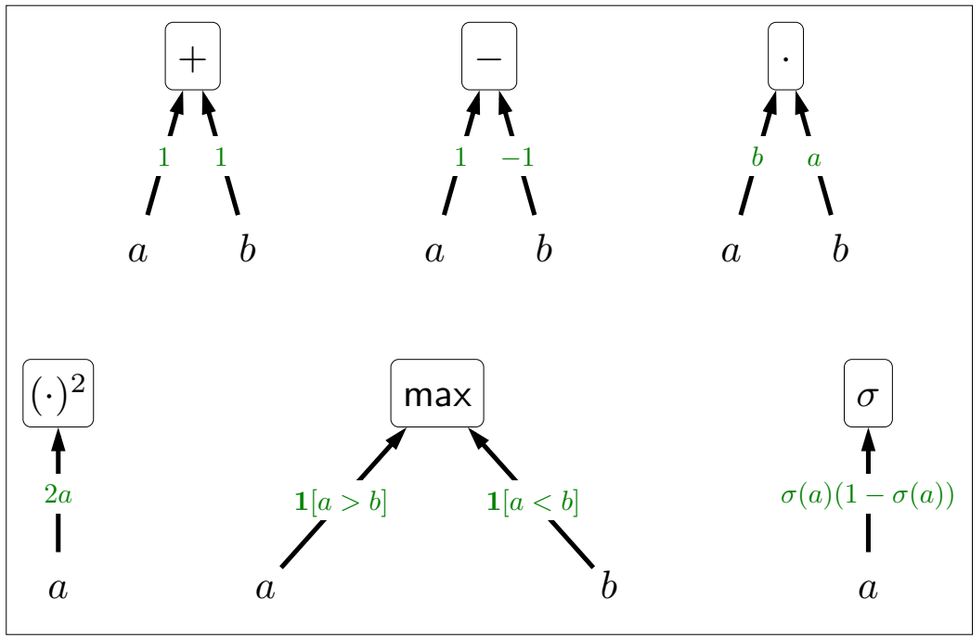
$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b}\frac{\partial b}{\partial a} = (2b)(2a) = 4a^3$$

Linear classification with hinge loss

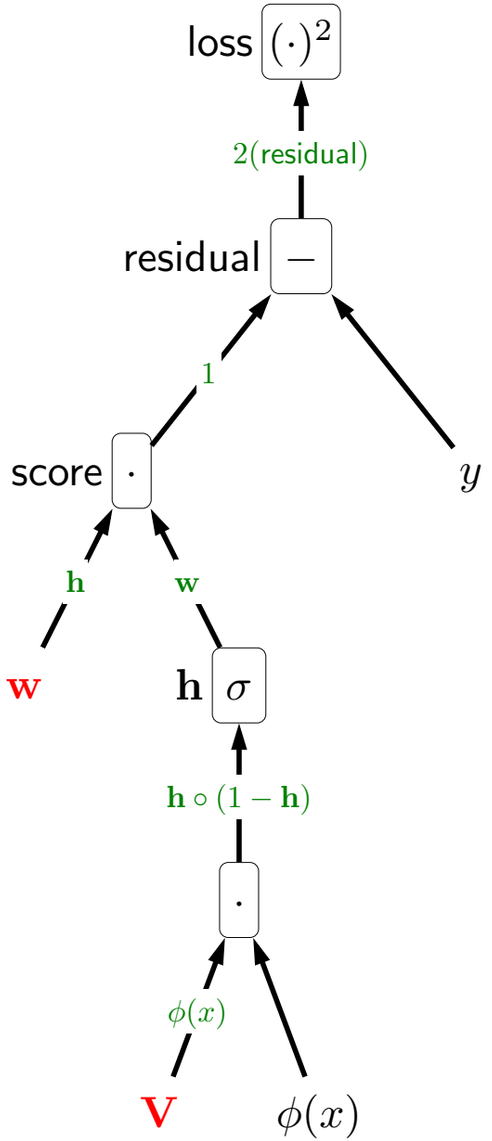


$$\text{Loss}(x, y, \mathbf{w}) = \max\{1 - \mathbf{w} \cdot \phi(x)y, 0\}$$

$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w}) = -\mathbf{1}[\text{margin} < 1]\phi(x)y$$



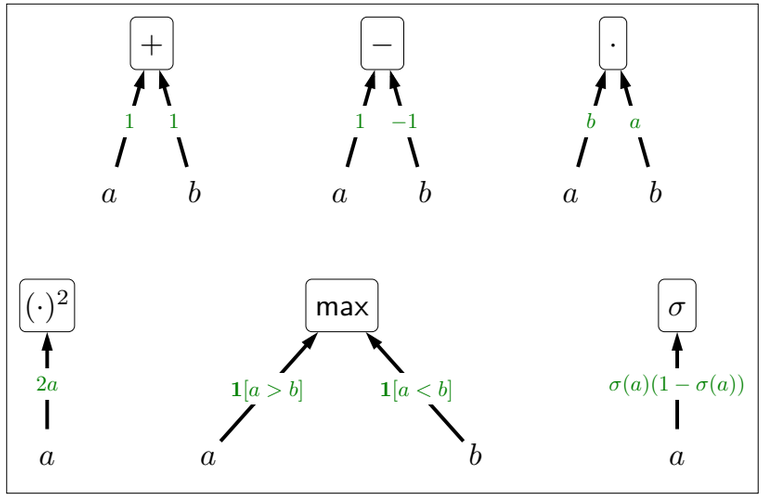
Two-layer neural networks



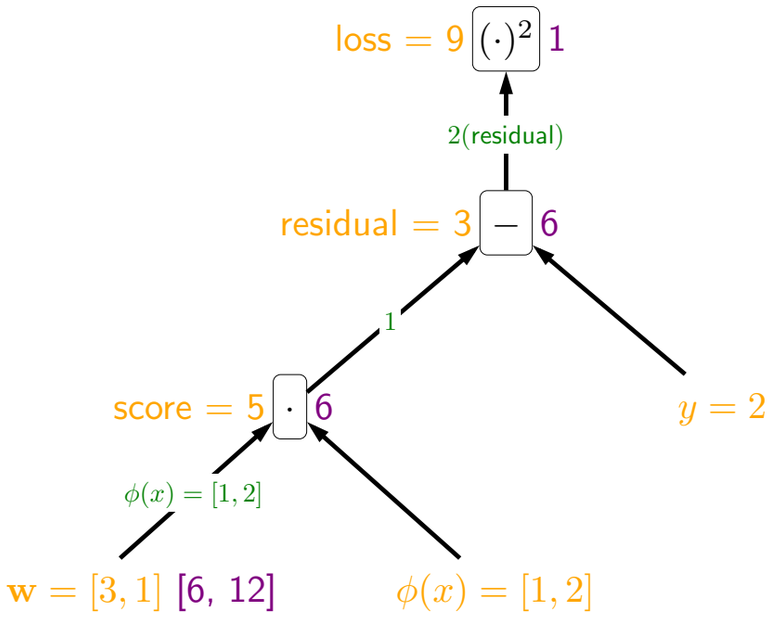
$$\text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}\phi(x)) - y)^2$$

$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2(\text{residual})\mathbf{h}$$

$$\nabla_{\mathbf{V}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2(\text{residual})\mathbf{w} \circ \mathbf{h} \circ (1 - \mathbf{h})\phi(x)^\top$$



Backpropagation

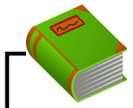


$$\text{Loss}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$$

$\mathbf{w} = [3, 1], \phi(x) = [1, 2], y = 2$

backpropagation

$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w}) = [6, 12]$



Definition: Forward/backward values

Forward: f_i is value for subexpression rooted at i

Backward: $g_i = \frac{\partial \text{loss}}{\partial f_i}$ is how f_i influences loss

Algorithm: backpropagation algorithm

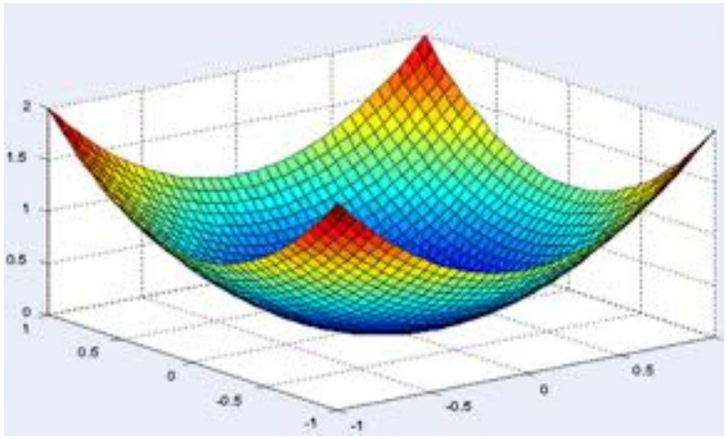
Forward pass: compute each f_i (from leaves to root)

Backward pass: compute each g_i (from root to leaves)

A note on optimization

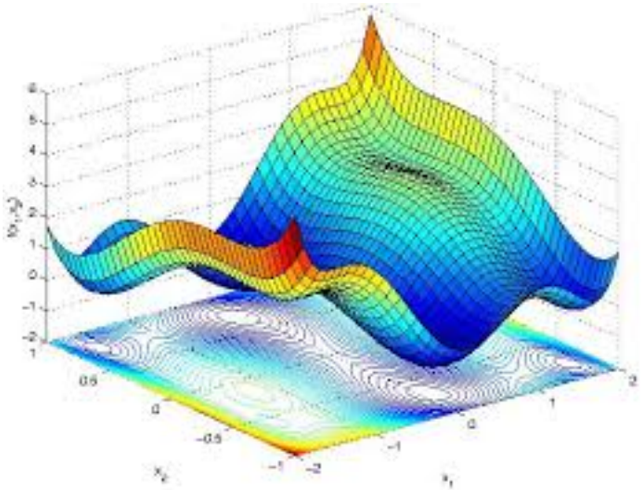
$$\min_{\mathbf{V},\mathbf{w}} \text{TrainLoss}(\mathbf{V}, \mathbf{w})$$

Linear predictors



(convex)

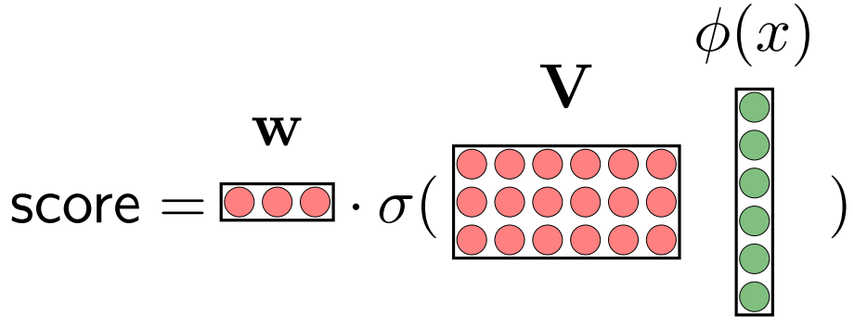
Neural networks

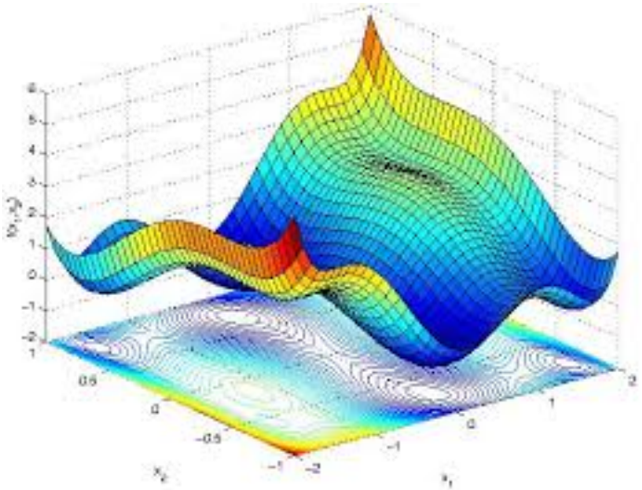


(non-convex)

Optimization of neural networks is in principle hard

How to train neural networks

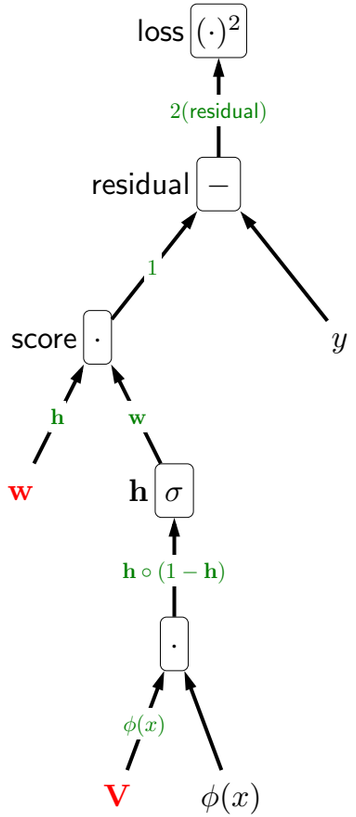




- Careful initialization (random noise, pre-training)
- Overparameterization (more hidden units than needed)
- Adaptive step sizes (AdaGrad, Adam)

Don't let gradients vanish or explode!

Summary



- Computation graphs: visualize and understand gradients
- Backpropagation: general-purpose algorithm for computing gradients