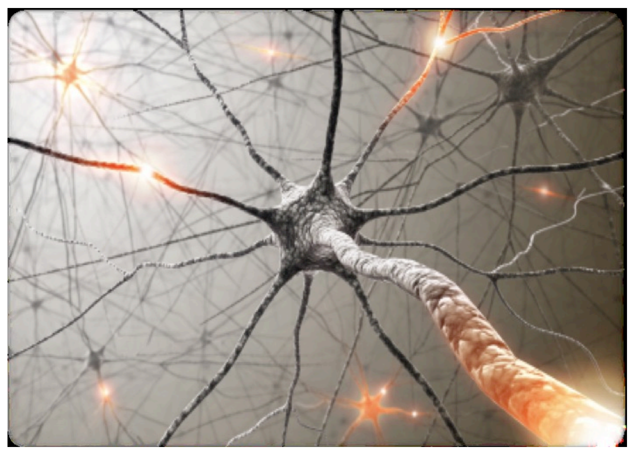
05 Neural Networks

Basics and BP



Roadmap

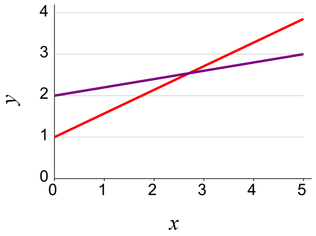
- Neural network basics

- Gradients without tears - BP algorithm

Non-linear predictors

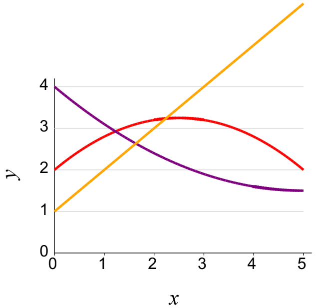
Linear predictors:

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



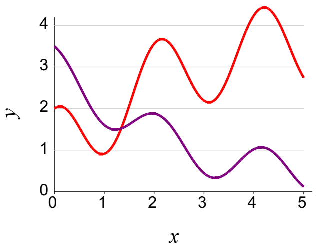
Non-linear (quadratic) predictors:

$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \varphi(x), \quad \varphi(x) = [1, x, {\color{red}x^2}]$



Non-linear neural networks:

$f_{\mathbf{w}}(x) = \mathbf{w} \cdot {\color{red}\sigma(\mathbf{V}\varphi(x))}, \quad \varphi(x) = [1, x]$



Motivating example

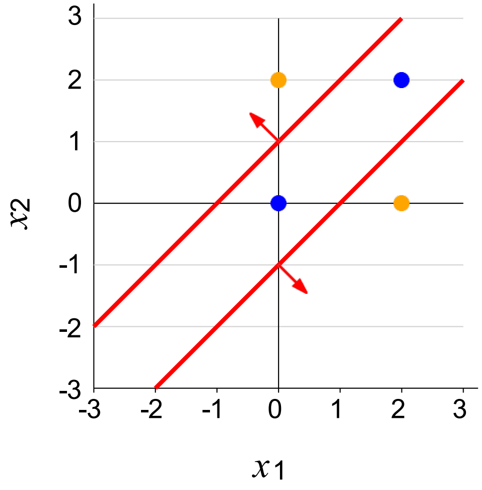
Example : predicting car collision

Input: position 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:

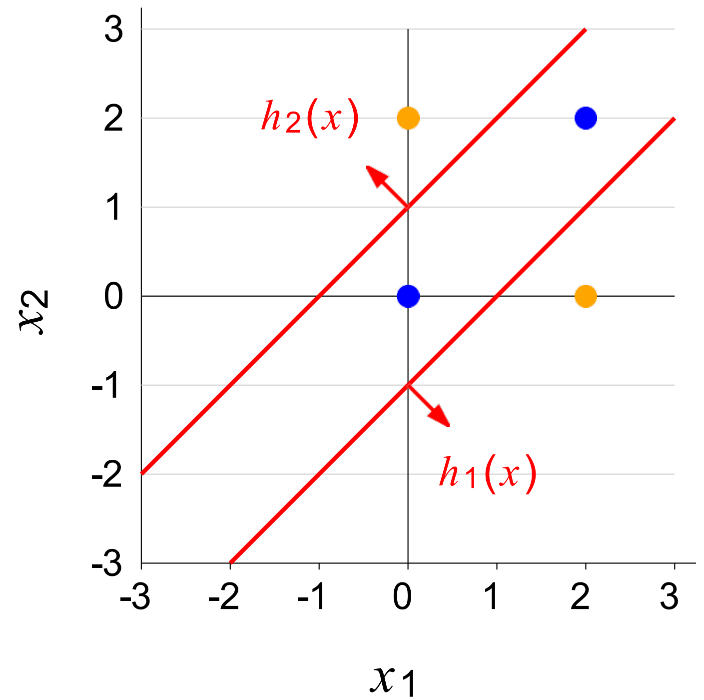
$$h_1(x) = \mathbf{1}[x_1 - x_2 \geq 1]$$

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:

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



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

Learning strategy

Define: $\phi(x) = [1, x_1, x_2]$

Intermediate hidden sub-problems:

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

$$h_2 = \mathbf{1}[\mathbf{v}_2 \cdot \phi(x) \geq 0] \qquad \mathbf{v}_2 = [-1, -1, +1]$$

Final prediction:

$$f_{V,\mathbf{w}}(x) = \text{sign}(w_1 h_1 + w_2 h_2) \qquad \mathbf{w} = [1, 1]$$

Key idea: minimize training loss

Goal: learn both hidden sub-problems $V = [\mathbf{v}_1, \mathbf{v}_2]$ and combination weights $\mathbf{w} = [w_1, w_2]$

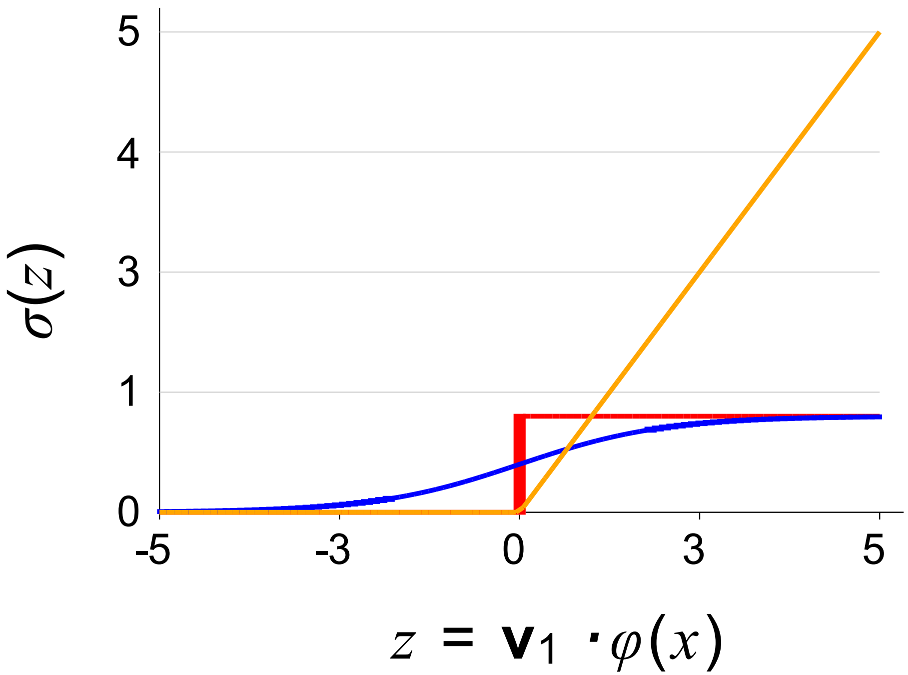
Avoid zero gradients

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

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

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

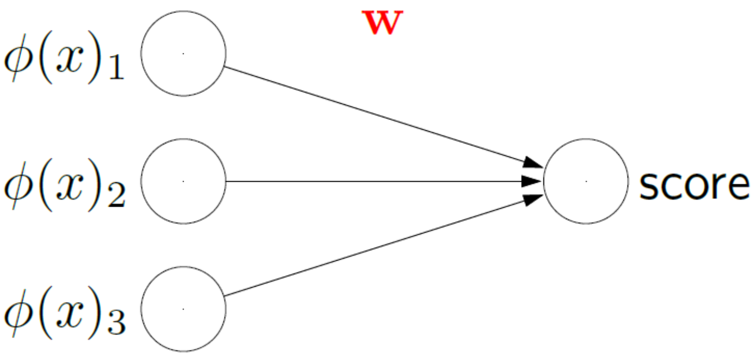
- Threshold: $\mathbf{1}[z \geq 0]$
- Logistic: $\frac{1}{1+e^{-z}}$
- ReLU: $\max(z, 0)$



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

Review: Linear predictors

Linear predictor:

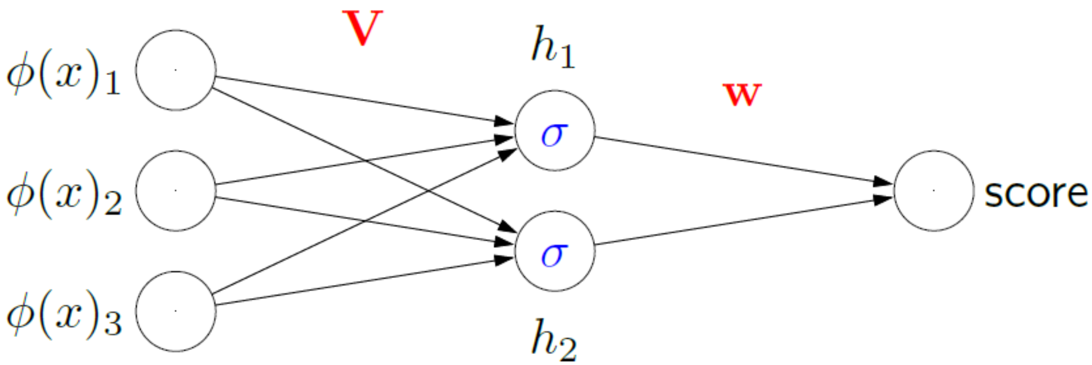


Output:

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

Two-layer Neural networks

Neural network:



Intermediate hidden units:

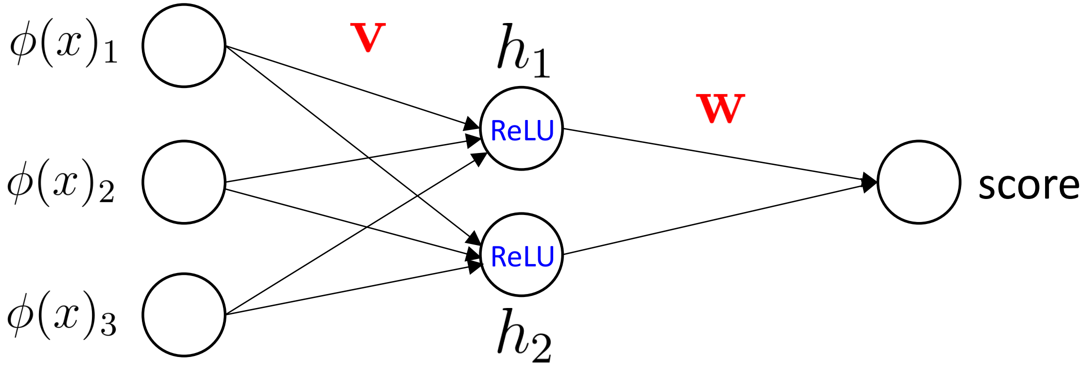
$$h_j = \sigma(\mathbf{v}_j \cdot \phi(x)) \quad \sigma(z) = (1 + e^{-z})^{-1}$$

Output:

$$\text{score} = \mathbf{w} \cdot [h_1, h_2]$$

Two-layer Neural networks

Neural network:



Intermediate hidden units:

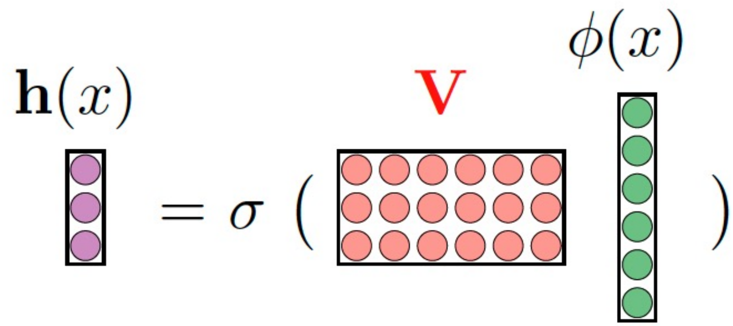
$$h_j = \text{ReLU}(\mathbf{v}_j \cdot \phi(x)) \quad \text{ReLU}(z) = \max(z, 0)$$

Output:

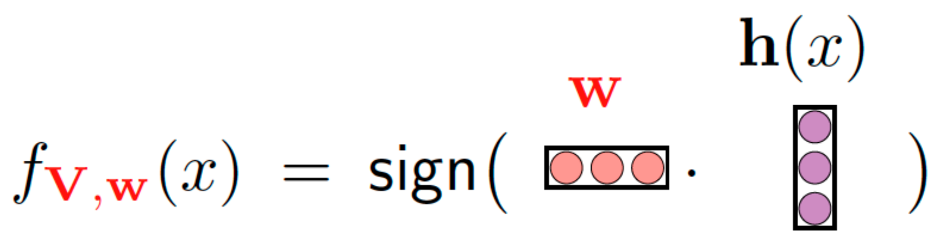
$$\text{score}(\phi(x); \mathbf{v}, \mathbf{w}) = \mathbf{w} \cdot [h_1, h_2]$$

Two-layer Neural networks

Intermediate hidden units:



Predictor (classification):



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

Hypothesis class:

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

Two-layer Neural networks

Interpretation: intermediate hidden units as learned features of a linear predictor

Key idea: feature learning

Before: apply linear predictor on manually specify features

$$\phi(x)$$

Now: apply linear predictor on automatically learned features

$$h(x) = [h_1(x), \ldots, h_k(x)]$$

Graph View of Neural Networks

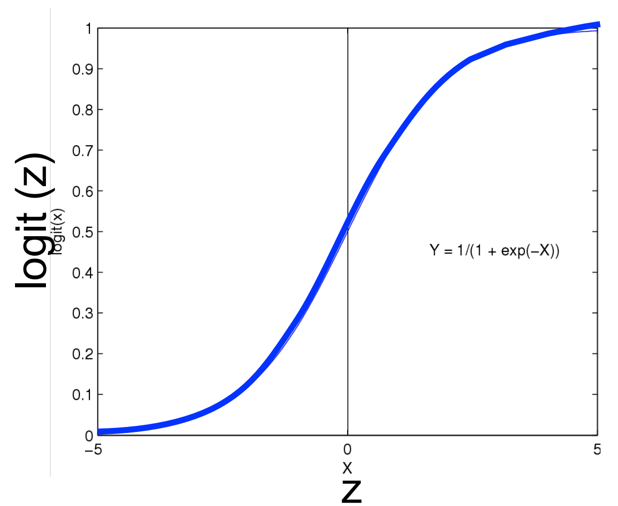
Review: Logistic Regression

Assumes the following functional form for P(Y|X):

$$P(Y=1|X) = \frac{1}{1+\exp(-(w_0+\sum_i w_i X_i))}$$

Logis1c function applied to a linear function of the data

Logistic function (or Sigmoid): $\dfrac{1}{1+exp(-z)}$



Graph View of Neural Networks

Review: Logistic Regression is a Linear Classifier!

Assumes the following functional form for P(Y|X):

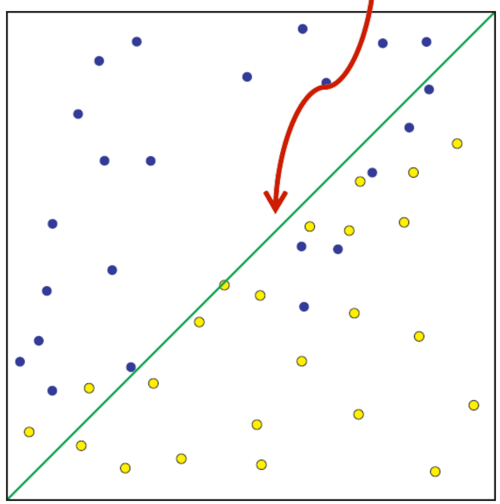
$$P(Y=1|X) = \frac{1}{1+\exp(-(w_0 + \sum_i w_i X_i))}$$

$$w_0 + \sum_i w_i X_i = 0$$

Decision boundary:

$$P(Y=0|X) \overset{0}{\underset{1}{\gtrless}} P(Y=1|X)$$

$$0 \overset{0}{\underset{1}{\gtrless}} w_0 + \sum_i w_i X_i$$



(Linear Decision Boundary)

Graph View of Neural Networks

Review: Training Logistic Regression

- **How to learn the parameters $w_0, w_1, \ldots w_d$?**
- Training Data $\{(X^{(j)}, Y^{(j)})\}_{j=1}^{n}$ $X^{(j)} = (X_1^{(j)}, \ldots, X_d^{(j)})$
- Maximum (Conditional) Likelihood Estimates

$$\widehat{\mathbf{w}}_{MCLE} = \arg\max_{\mathbf{w}} \prod_{j=1}^{n} P(Y^{(j)} \mid X^{(j)}, \mathbf{w})$$

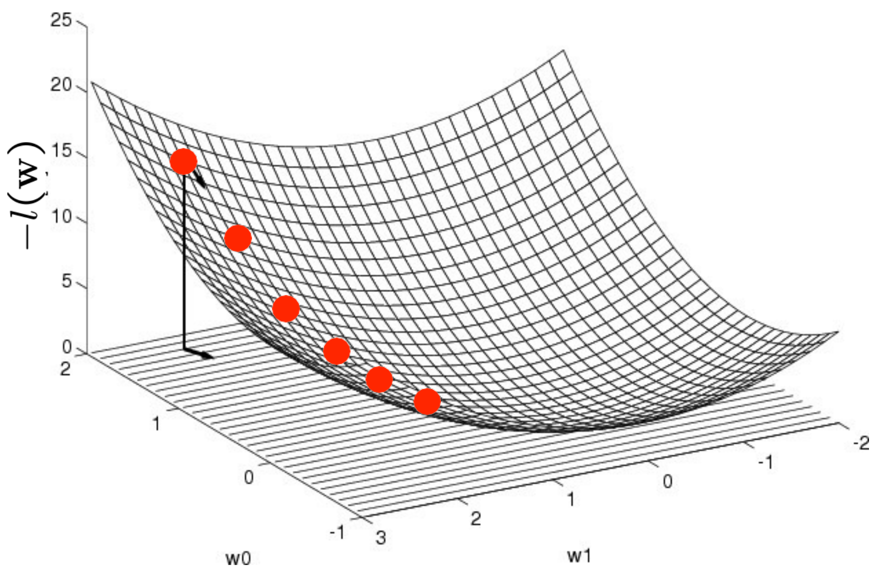
Discriminative philosophy – Don't waste effort learning P(X), focus on P(Y|X) – that's all that matters for classification!

Graph View of Neural Networks

Review: Training Logistic Regression

- Max Conditional log--likelihood = Min Negative Conditional log--likelihood
- Negative Conditional log--likelihood is a convex function

Gradient Descent (convex)



Gradient:

$$\nabla_{\mathbf{w}} l(\mathbf{w}) = [\frac{\partial l(\mathbf{w})}{\partial w_0}, \ldots, \frac{\partial l(\mathbf{w})}{\partial w_d}]'$$

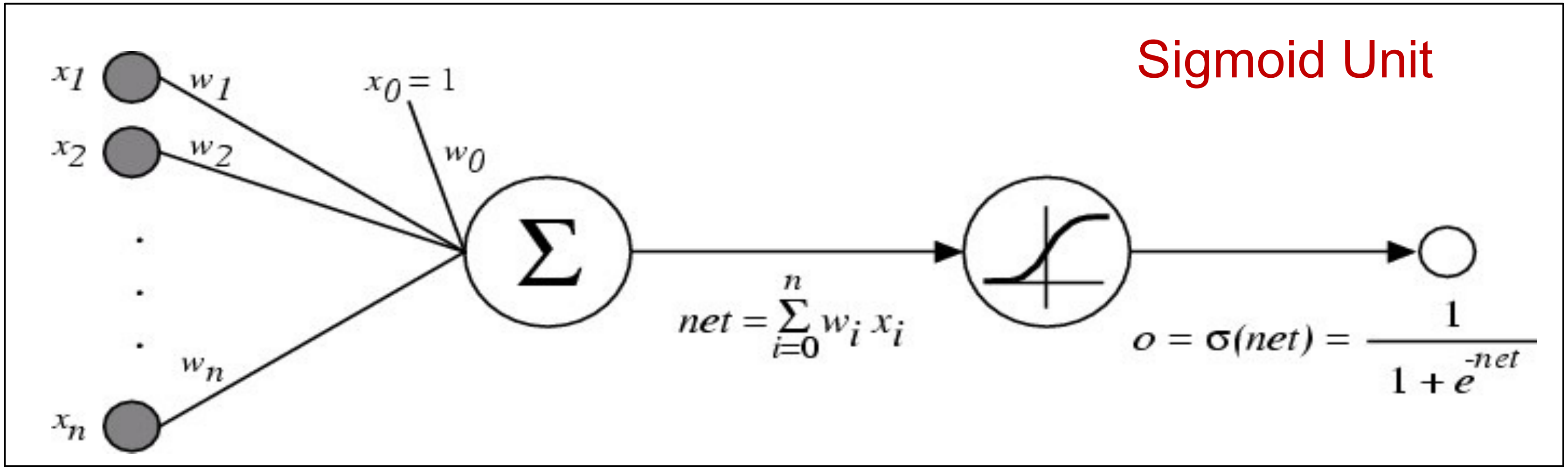
Update rule: **Learning rate, η>0**

$$\Delta \mathbf{w} = \eta \nabla_{\mathbf{w}} l(\mathbf{w})$$

$$w_i^{(t+1)} \leftarrow w_i^{(t)} - \eta \left.\frac{\partial l(\mathbf{w})}{\partial w_i}\right|_t$$

Graph View of Neural Networks
Logistic function as a Graph

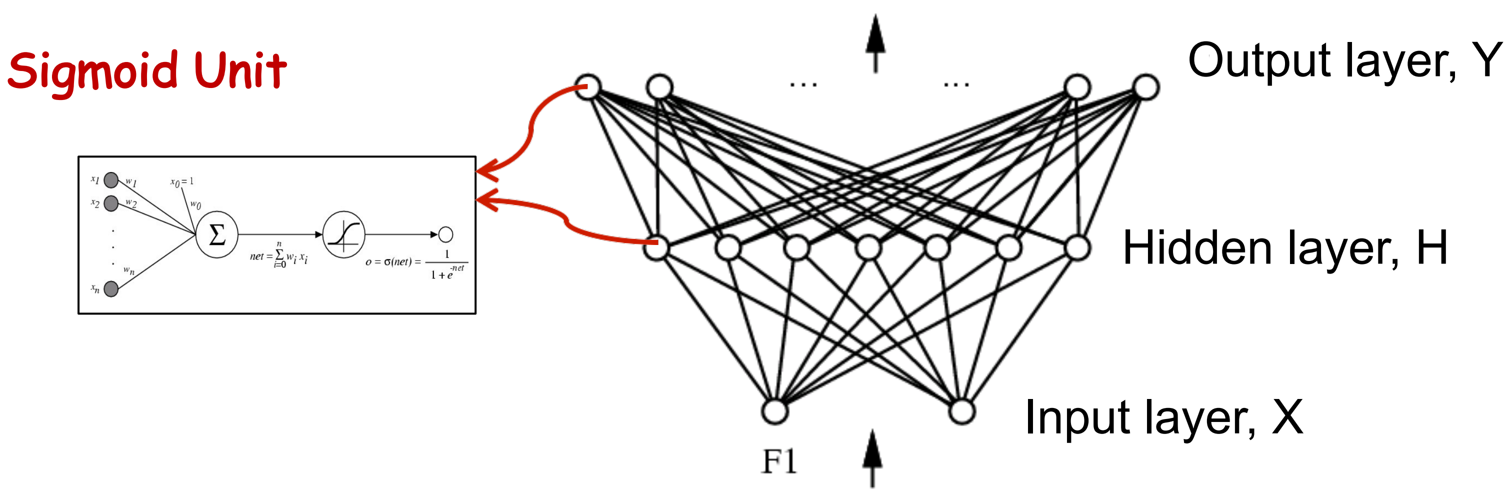
$$\text{Output, } o(\mathbf{x}) = \sigma(w_0 + \sum_i w_i X_i) = \frac{1}{1 + \exp(-(w_0 + \sum_i w_i X_i))}$$



Graph View of Neural Networks
Neural Networks to learn f: X→Y

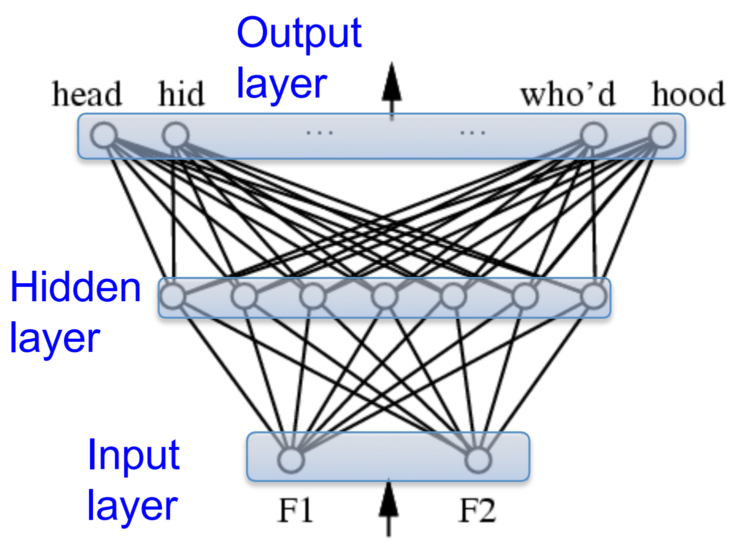
- f can be a **non-linear** function
- X (vector of) continuous and/or discrete variables
- Y (**vector** of) continuous and/or discrete variables

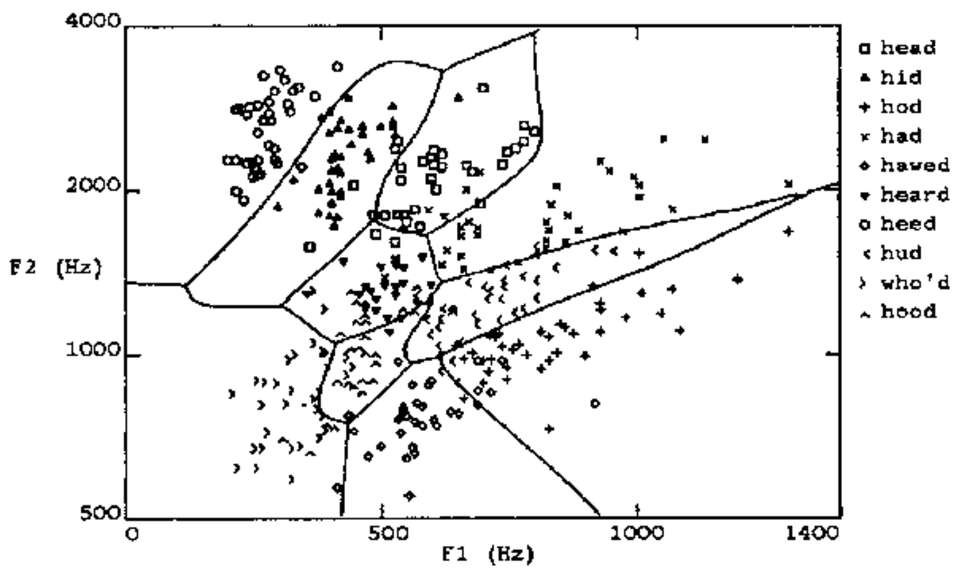
- Neural networks -- Represent f by *network* of logistic/sigmoid units:



Graph View of Neural Networks
Multilayer networks of sigmoid units

Neural Network trained to distinguish vowel sounds using 2 formants (features)



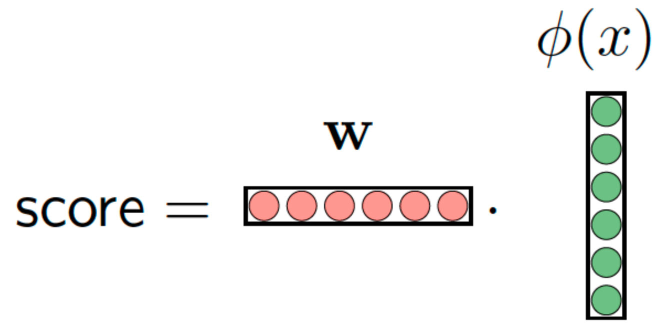


Two layers of logistic units

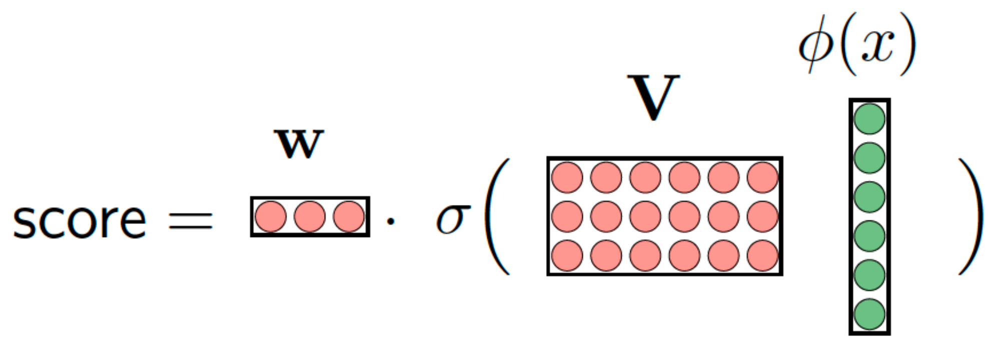
Highly non-linear decision surface

Deep neural networks

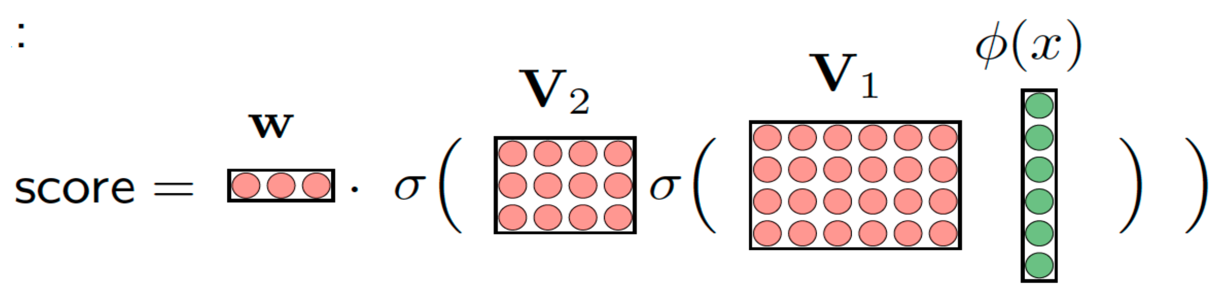
1-layer neural network:



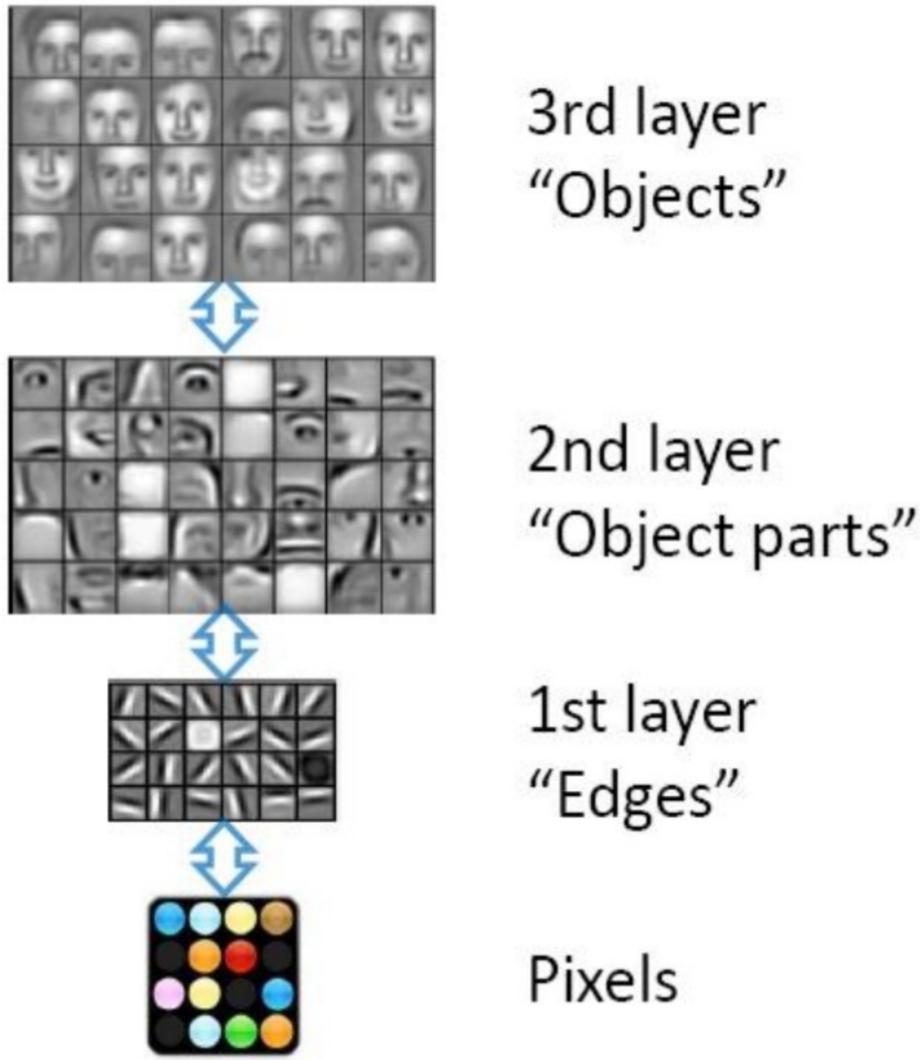
2-layer neural network:



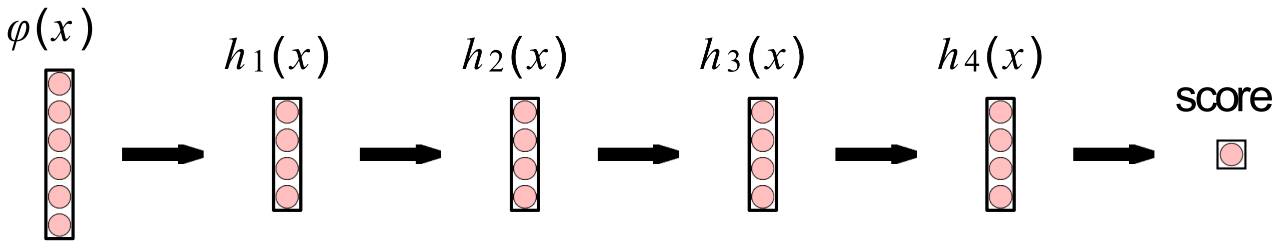
3-layer neural network:



Layers represent multiple levels of abstractions



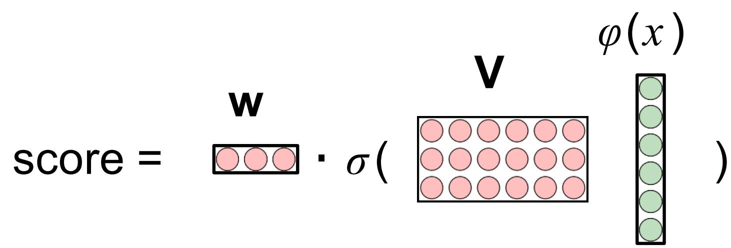
Why depth?



Intuitions:

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

Summary so far



- 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

Roadmap

- Neural network basics
- Gradients without tears - BP algorithm

Motivation: loss minimization

Optimization problem:

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

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

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

$$f_{\mathbf{V},\mathbf{w}}(x) = \sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x))$$

Goal: compute gradient $\text{Loss}(x, y, \mathbf{v}, \mathbf{w}) = (y - f_{\mathbf{v},\mathbf{w}}(x))^2$

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

Motivation: regression with four-layer neural networks

Loss on one example:

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

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?

Approach

- Mathematically: just grind through the chain rule

- Next: visualize the computation using a computation graph

- Advantages:
  - Avoid long equations
  - Reveal structure of computations (modularity, efficiency, dependencies)

Computation graphs

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

Definition: computation graph

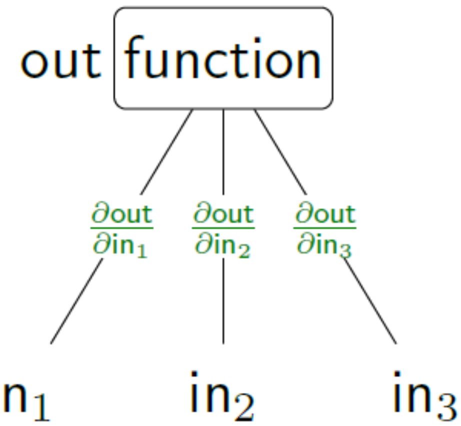
A directed acyclic graph whose root node represents the final mathematical expression, and each node represents intermediate subexpressions.

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

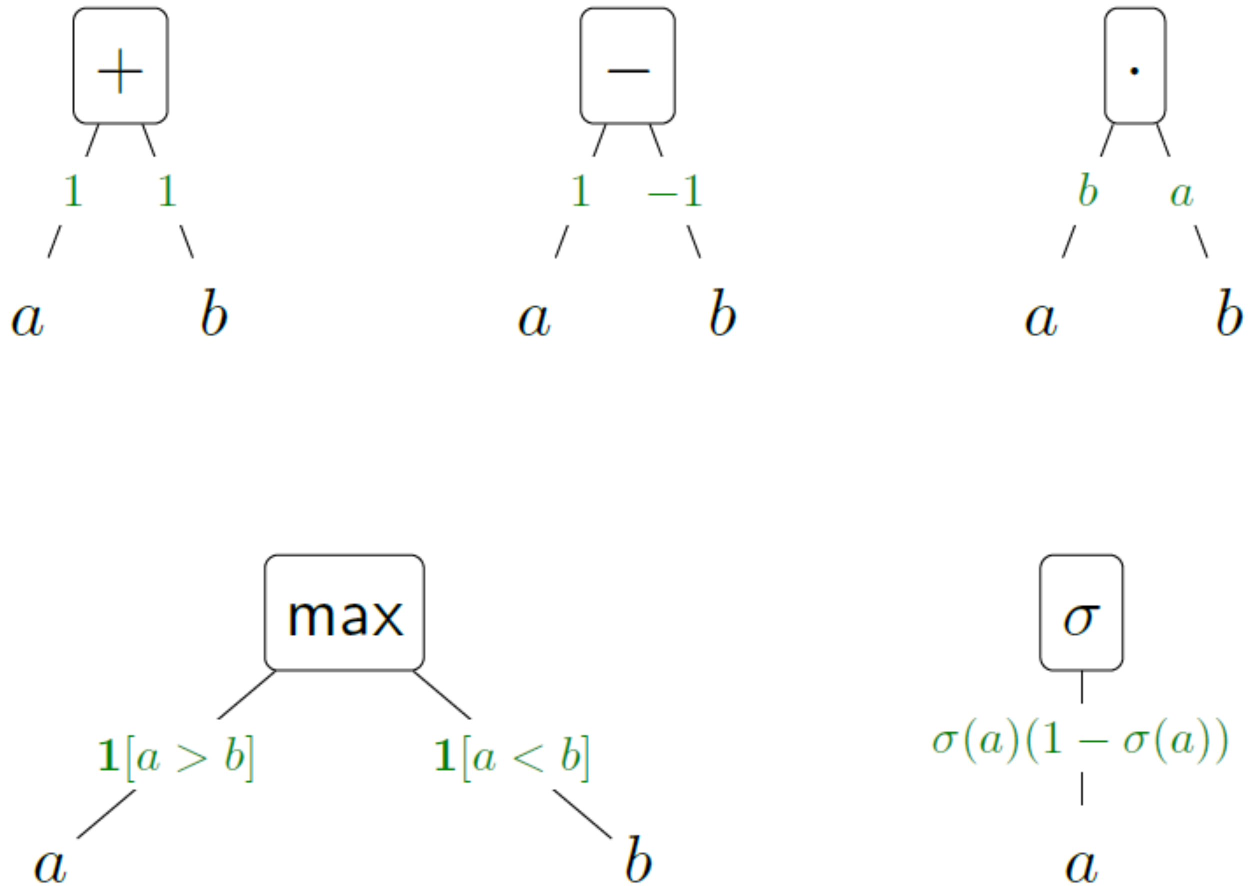


Partial derivatives (gradients): how much does the output change if an input changes?

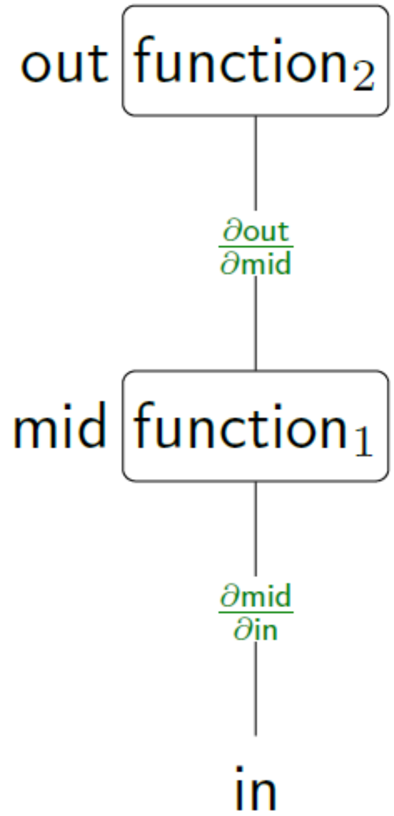
Example:

$$2\text{in}_1 + (\text{in}_2 + \epsilon)\text{in}_3 = \text{out} + \text{in}_3\epsilon$$

Basic building blocks



Composing functions



Chain rule:

$$\frac{\partial \text{out}}{\partial \text{in}} = \frac{\partial \text{out}}{\partial \text{mid}} \frac{\partial \text{mid}}{\partial \text{in}}$$

Binary classification with hinge loss

- Hinge loss:

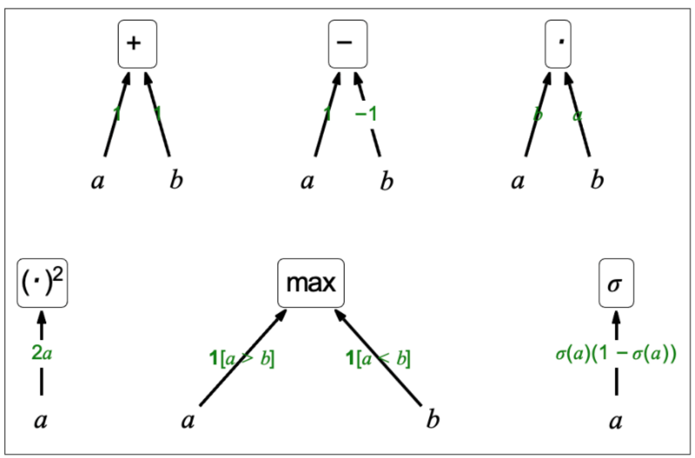
$$\mathrm{Loss}(x, y, \mathbf{w}) = \max\{1 - \mathbf{w} \cdot \phi(x) y, 0\}$$

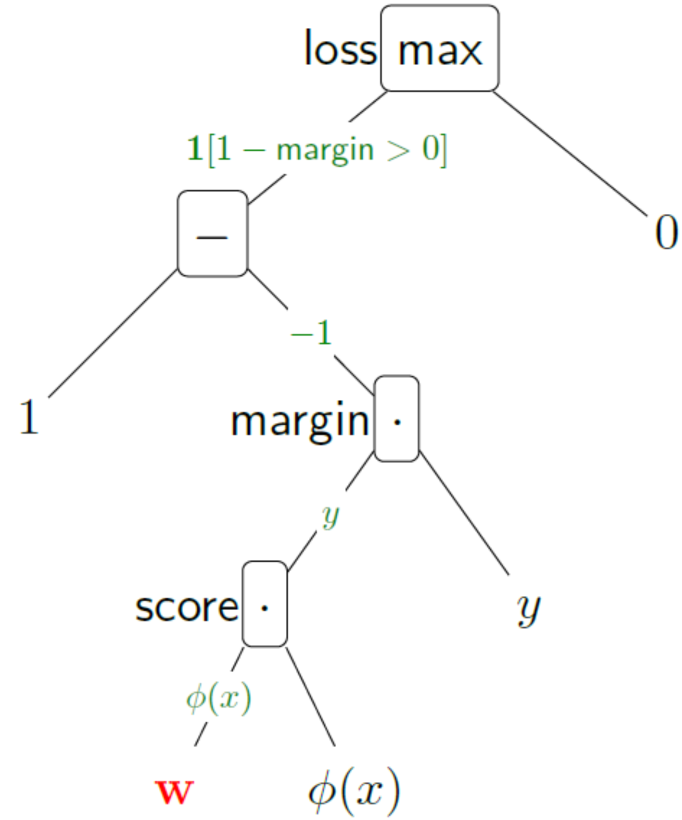
- Compute:

$$\frac{\partial \mathrm{Loss}(x, y, \mathbf{w})}{\partial \mathbf{w}}$$

Binary classification with hinge loss

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





Gradient: multiply the edges

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

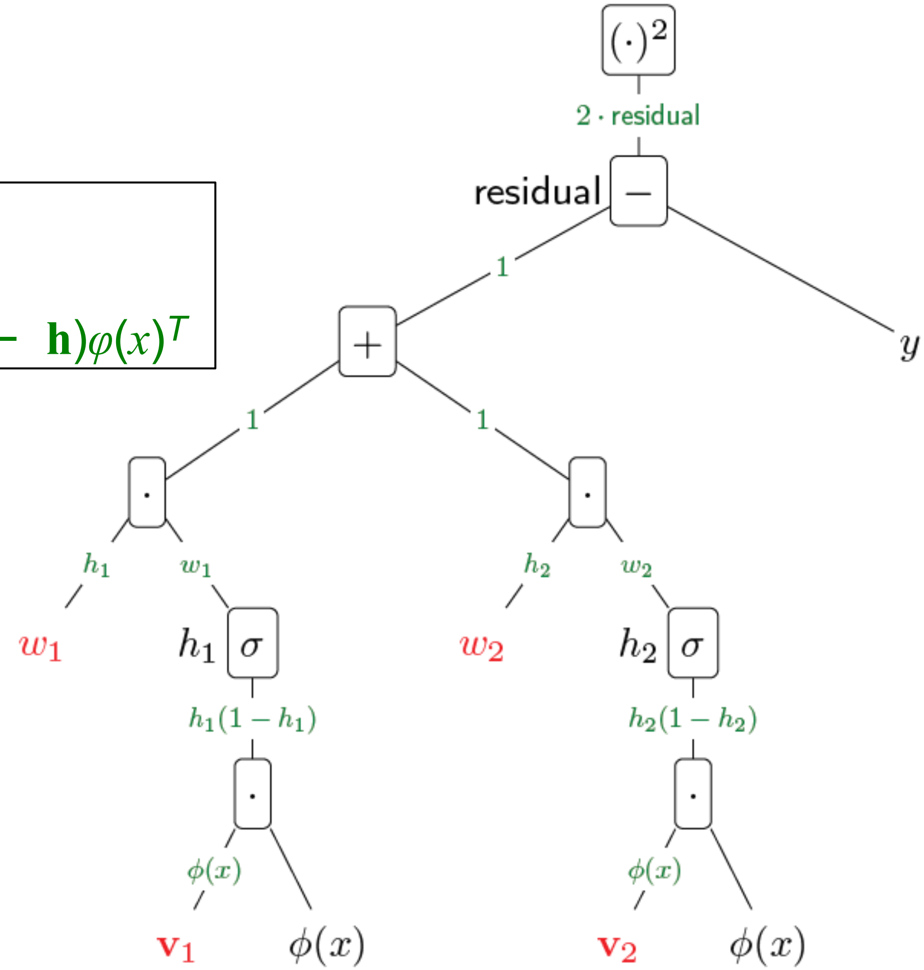
Two-layer Neural network

$$\text{Loss}(x, y, \mathbf{w}) = \left(\sum_{j=1}^{k} w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y\right)^2$$

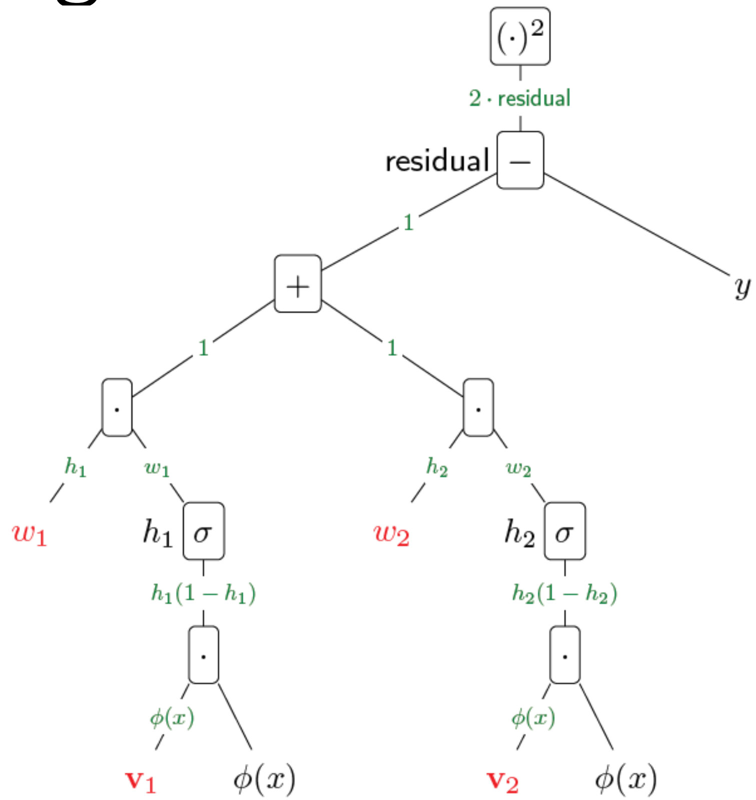
$$\text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}\varphi(x)) - y)^2$$

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

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



Backpropagation

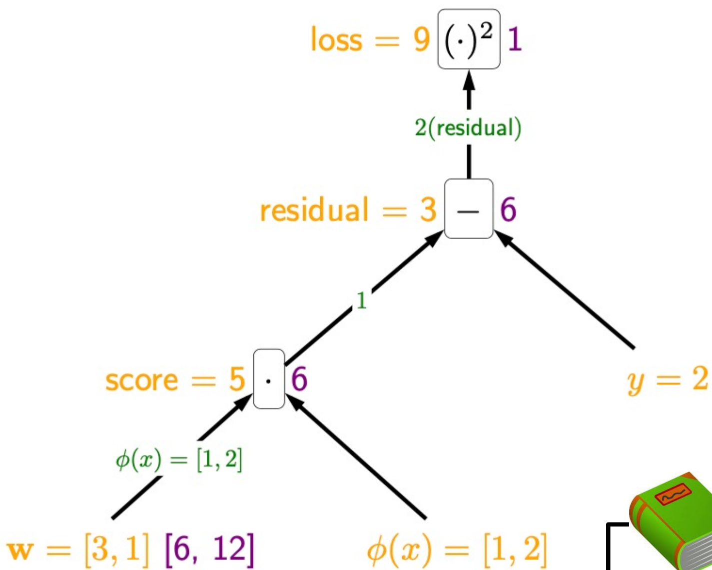


Definition: forward/backward values

Forward: f_i is value for subexpression rooted at i

Backward: $g_i = \dfrac{\partial \text{out}}{\partial f_i}$ is how f_i influences output

Backpropagation



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

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

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

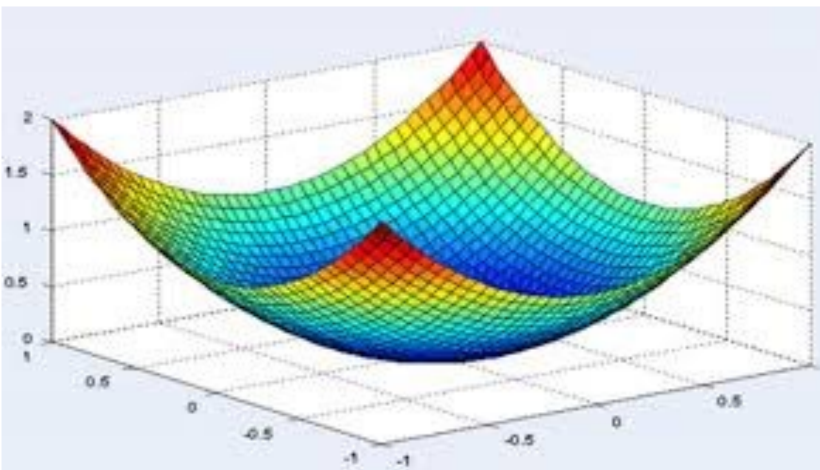
Forward pass: compute each f_i (from leaves to root)

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

A note on optimization

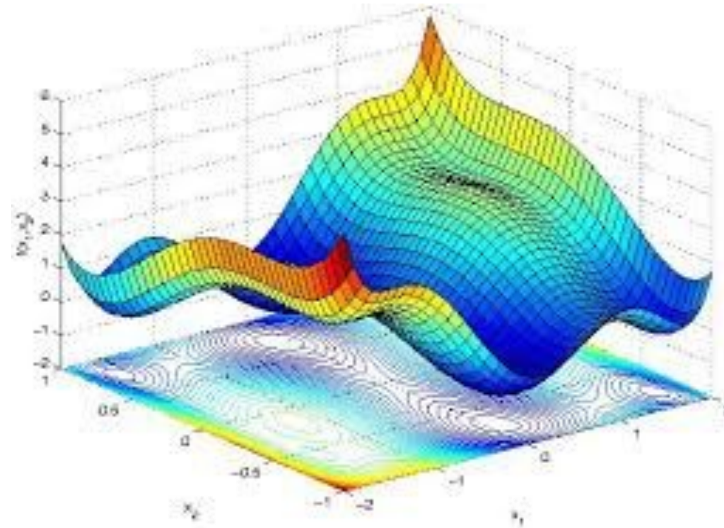
$$\min_{\mathbf{V},\mathbf{w}} \text{TrainLoss}(\mathbf{V}, \mathbf{w})$$

Linear predictors



(convex)

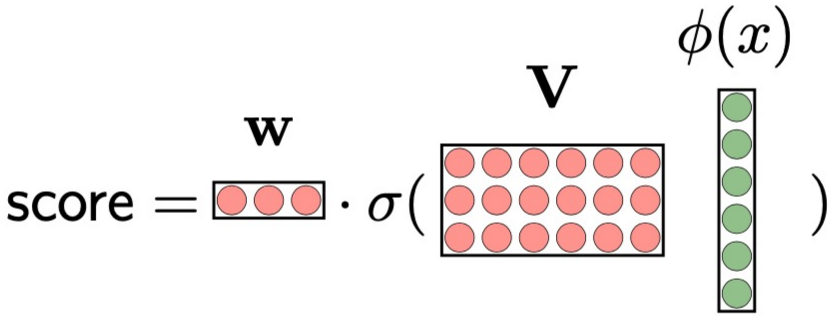
Neural networks

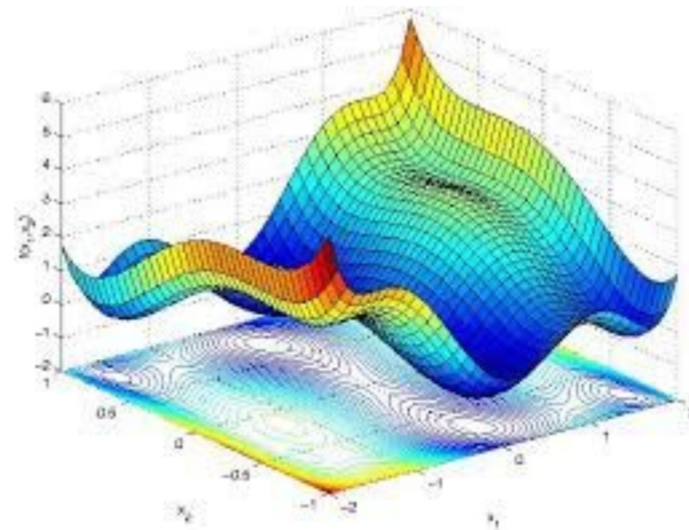


(non-convex)

Optimization of neural networks is in principle hard

How to train neural networks



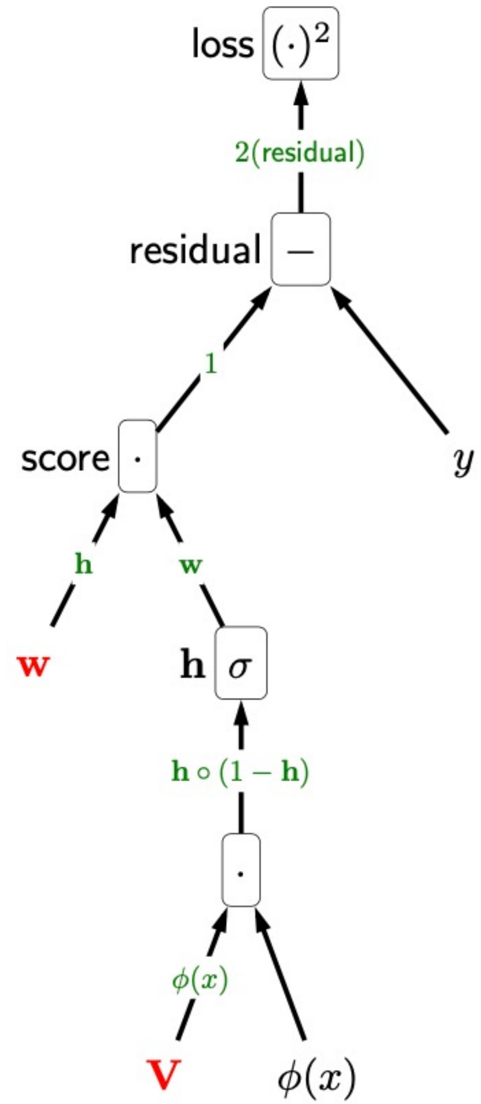


- Careful initialization (random noise, pre-training)
- Overparameterization (more hidden units than needed)
- Adaptive step sizes (AdaGrad, Adam)

Don’t let gradients vanish or explode!

Summary of learners

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



Summary of learners

- Linear predictors: combine raw features
  - prediction is fast, easy to learn, weak use of features

- Neural networks: combine learned features
  - prediction is fast, hard to learn, powerful use of features