Mathematics of Deep Learning
Lecture 1: Introduction

Prof. Predrag R. Jelenković
Time: Tuesday 4:10-6:40pm
602 Hamilton Hall

Dept. of Electrical Engineering
Columbia University , NY 10027, USA
Office: 812 Schapiro Research Bldg.
Phone: (212) 854-8174
Email: predrag@ee.columbia.edu
URL: http://www.ee.columbia.edu/∼predrag

Math of Deep Learning: Brief Description

- **Deep Learning**: In recent years, deep learning methods have achieved unparalleled success in various application areas of machine learning.
- **Emerging Mathematical Understanding**: However, the theoretical understanding of why deep learning works well remains limited. This course will cover some of the emerging mathematical aspects and understanding of deep learning methods.
- **The preliminary list of topics** include the results on:
  - Expressive (approximation) power of neural networks
  - Depth separation results: Can deep neural networks of depth $(d+1)$ express functions much more efficiently in terms of the number of neurons and parameters compared to networks of depth d?
  - What classes of functions can deep neural nets approximate well?

Math of Deep Learning: Brief Description

- **The preliminary list of topics** include the results on:
  - Wide over-parametrized networks
  - Connection between wide neural nets and Kernels: Neuro Tangent Kernels (NTK)
  - Global versus local optimality
  - Convergence properties of training for wide nets:
    - Is training converging to global min?
    - Is training converging far or close to the initial NTK?
  - Generalization error: Deep neural networks have a lot of parameters. How come they are not overfitting?
    - Basic generalization concepts from machine learning theory will be covered
  - Deep residual networks
  - Deep generative probabilistic models, e.g., deep Boltzmann machines (time permitting)
  - Etc.

Math of Deep Learning: Course Logistics

Prerequisites: Solid undergraduate knowledge of multivariate calculus, linear algebra and probability/statistics.

Textbook: No textbook (on mathematics of deep learning exists). Research papers and lecture notes will be used.

Textbook on theory of machine learning

[UML] Shai Shalev-Shwartz and Shai Ben-David, Understanding Machine Learning: From Theory to Algorithms, Cambridge University Press, 2014. (Click blue text for free pdf.)

Acknowledgement. I am grateful to Shai Shalev-Shwartz for sharing the lecture notes that are related to the book. Occasionally, modified parts of these notes will be used.

Textbook on deep learning

[DL] I. Goodfellow and Y. Bengio and A. Courville, Deep Learning, MIT Press, 2016.

Introductory books to statistical/machine learning

[ESL] Hastie, T., Tibshirani, R. and Friedman, J. The Elements of Statistical Learning: Data Mining, Inference and Prediction, 2nd Edition. Springer, 2009.

[ISL] James, G., Witten, D. Hastie, T. and Tibshirani, R., An Introduction to Statistical Learning, Springer, 2014.

Math of Deep Learning: Course Logistics

Grading: Participation ($\leq 15\%$) + Final Project ($\geq 85\%$) (tentative). Maybe some limited amount of homework.

Programming: The experimental part of the project can be implemented in R or Python

Final Research Project:

- Done in groups of 4
- Can be mathematical or experimental, or mix between the two
- Deliverables: presentation + paper + code (for numerics)
- The paper should include:
  - **Survey** on a topic related to the course
  - **Research** part: math and/or experimental
    Experiments need to study properties of deep learning networks (instead of focusing on solving a particular problem training a particular data set)
    Might need to write code from scratch

Math of Deep Learning: Course Characteristics

- Research oriented: starting with some classical papers from late 80s and early 90s, we will mostly cover recent research papers
- Organize the material around important questions/themes (Good questions are the most important component of research)
- Some expected difficulties
  - No book $\rightarrow$ less structure
  - Non-uniform notation: Different communities use different terminology
- Advanced background topics in mathematics and machine learning theory will be covered as needed, e.g.:
  - Math: functional analysis, approximation theory, optimization,..
  - Probability: concentration inequalities
  - ML theory: PAC learning, VC dimension, Rademacher complexity, ...

Programming in R or Python

R computing platform:

- Language for statistical computing and learning
- Free software
- Download
  - R from http://cran.r-project.org/
  - RStudio, an Integrated Development Environment for R, from http://www.rstudio.com/products/rstudio/download/
  - Deep learning software - Tensor Flow in R
- Resources
  - R for beginners
  - Quick-R
  - Cookbook for R
  - R for Data Science
  - Try R

General Learning Framework: Supervised or Unsupervised

- Supervised learning: there is an input-output relationship

$$\boxed{Y = f(X)}$$

  - f - unknown
  - Training data (observations): $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
  - Objectives:
    - Learn/estimate $\hat{f}$ from training data
    - Inference/prediction: Use $\hat{f}$ to predict outcomes on unseen/new data
  - Two problems:
    - Regression: Y is quantitative
    - Classification: Y is categorical

- Unsupervised learning: Just X, no output Y (no labels)
  Typical problems:
  - Distribution estimation: learn the distribution/density, $p(x)$
    - Generative modeling: first estimate the distribution, then use it for other learning problems, e.g., QDA/LDA
  - Dimensionality reduction, e.g., PCA
  - Clustering and, in general, any data mining, e.g., data association, etc.

Statistical Learning: What Does It Involve?

Supervised learning

$$\boxed{Y = f(X)}$$

Problem: Estimate f from training data $\{(x_i, y_i)\}$

Areas involved:

- Approximation theory - for picking a class of approximation functions
- Optimization - for fitting the training data
- Computing - fitting and testing
- Probability and Statistics - estimation of testing error

Interesting Question: What is the difference between classical programming and statistical/machine learning?

- Classical Programming: f is an algorithm designed by a person
- Statistical Learning: f is discovered through examples by training

Why Is Learning Difficult?: Curse of Dimensionality

Why is it hard to estimate a function in high dimensions?

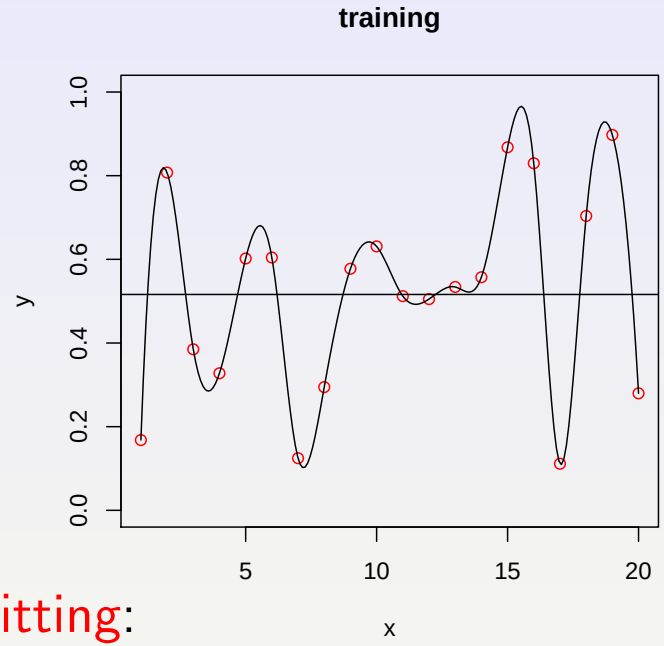
- How do we estimate a density of one dimensional X on $[0, 1]$?
  Question: Say, we want to learn $p(x) : [0, 1] \to \mathbb{R}$
  - Solution: split $[0, 1]$ in 100 bins of size $\epsilon = 0.01$, get about 1000 samples of X and plot a histogram
    This should be a pretty good estimate of $p(x)$
- Suppose X is supported on 100 dimensional cube $[0, 1]^{100}$
  Learn density $p(x) : [0, 1]^{100} \to \mathbb{R}$
  - The preceding solution: splitting $[0, 1]^{100}$ in bins of size 0.01^{100} would require more than

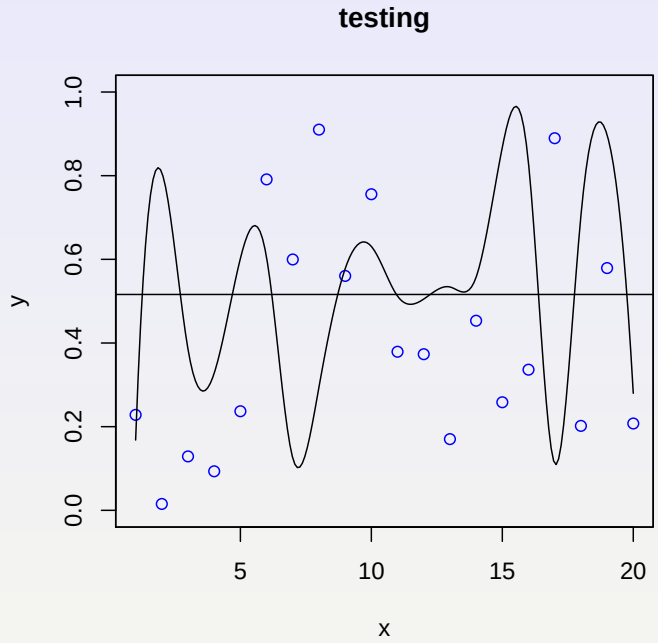
    10^{200} samples (!)

    This is usually referred to as curse of dimensionality
- Many real world problems are high dimensional
  Images $> 10^6$ dimensions; gene expression data $> 10,000$
- Only hope: existence of low dimensional structure

Overfitting Problem

One can fit infinitely many functions through a finite set of points, but





Overfitting:

- Low training error does not imply low testing error
- More complicated models not always better
  - Less interpretability
  - More difficult to train
- Mystery: Deep is learning highly flexible, possibly millions of parameters, but usually doesn't overfit.
- **John von Neumann elephant quote**: "With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."

General Supervised Learning Setup

Supervised learning problem can be formulated as (say, $x_i \in \mathbb{R}^p, y_i \in \mathbb{R}^m$)

$$\hat{f}^* = \arg\min_{\hat{f} \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, \hat{f}(x_i)) + \lambda R(\hat{f}) \tag{1}$$

- $\ell : \mathbb{R}^{p+m} \to \mathbb{R}$ - loss function, and empirical risk/loss is defined as $\boldsymbol{x} = (x_1, \ldots, x_p), \boldsymbol{y} = (y_1, \ldots, y_m)$

  $$\bar{L}(\boldsymbol{x}, \boldsymbol{y}) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, \hat{f}(x_i))$$

  Hence, Equation (1) is called Empirical Risk Minimization (ERM)

- $\mathcal{H}$ - Hypothesis class (class of approximation functions)
  Desirable properties of $\mathcal{H}$:
  - Rich/versatile, yields accurate predictions, easy to train (e.g., problem (1) is convex), interpretable/simple, etc.
- $\lambda R(\hat{f}) \in \mathbb{R}^+$ - regularizer/penalty, shrinkage term
  - $\lambda R(\hat{f})$ - shrinking $\mathcal{H}$: if $\lambda_2 > \lambda_1 \to \mathcal{H}_{\lambda_2} \subset \mathcal{H}_{\lambda_1}$
  - Should prevent overfitting

Linear Examples: Regression and Classification

Regularized Linear Regression (say $y \in \mathbb{R}$)

- $\mathcal{H}$: set off all p-dimensional linear functions
  $\hat{f}(x) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$
- Quadratic loss: $\ell(y_i, \hat{f}(x_i)) = (y_i - \hat{f}(x_i))^2$
- Typical penalties
  - Ridge - l_2 norm: $R(\hat{f}) =< \hat{f}, \hat{f} >= \|f\|^2 = \sum \beta_j^2$
  - LASSO: l_1 norm (basis pursuit): $R(\hat{f}) = \sum |\beta_j|$

Support Vector Classifier: Separate two classes by a hyperplane,
$y_i \in \{-1, 1\}$

$$\min_{\beta_0, \boldsymbol{\beta}} \sum_{i=1}^{n} \max\left[0, 1 - y_i(\beta_0 + x_{i1}\beta_1 + \cdots + x_{ip}\beta_p)\right] + \lambda \sum_{j=0}^{p} \beta_j^2$$

$\ell(x_i, y_i) = \max\left[0, 1 - y_i(\beta_0 + x_{i1}\beta_1 + \cdots + x_{ip}\beta_p)\right]$ is called **hinge loss**

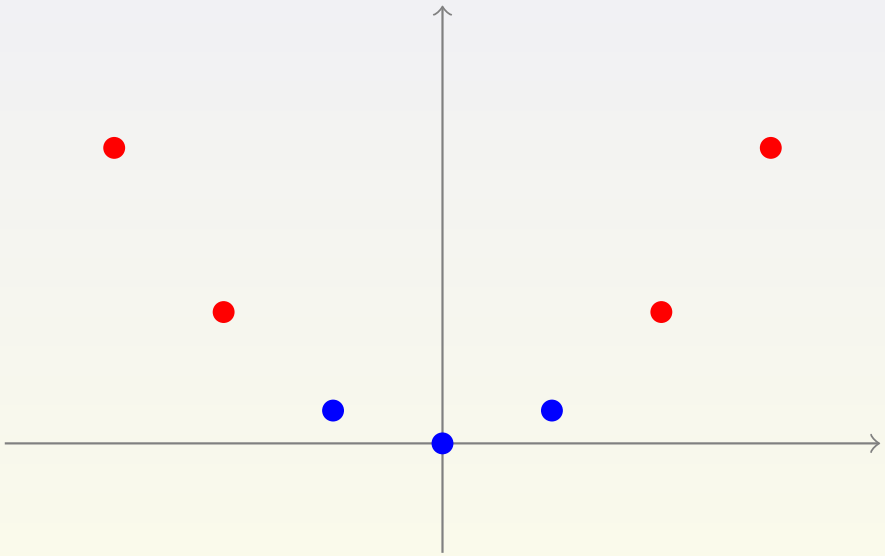
- What if the function is not linear?
  This is most likely the case.

Nonlinear Functions: Transform/Expand Features

Example: Classifying red and blue point on a real line

- Can't be separated by a single point (hyperplane)

- Feature transformation/basis expansion: $x \rightarrow (x, x^2)$
- Now, the points are separable by a single hyperplane (!)



- Feature engineering: $x \rightarrow \phi(x)$
- Well-developed theory: Reproducing Kernel Hilbert Spaces (RKHS)
- Problem: How do we find feature functions/kernels?

Where Does Deep Learning Fits in This Framework?

- Rich $\mathcal{H}$: Provides a versatile parametric class of functions
  - Universal function approximation: depth helps improves expressiveness
  - However, it is difficult to train: non-convex optimization
  - Often produces accurate predictions in practice, but difficult to understand and interpret
- Automatic extraction of feature maps
  - Traditional Feature Engineering approach: expert constructs feature mapping $\phi : \mathcal{X} \to \Phi$. Then, apply machine learning to find a linear predictor on $\phi(\mathbf{x})$.
  - "Deep learning" approach: neurons in hidden layers can be thought of as feature maps that are being learned automatically from the data
  - Shallow neurons corresponds to low level features, while deep neurons correspond to high level features

Parametric Supervised Learning

- $\mathcal{H}$ is a parametric class of functions $f(w, x), w \in \mathcal{W}, w = (w_1, \ldots, w_k)$
  - Examples: polynomials, or other linear combinations of basis, neural networks
- Finding $f \in \mathcal{H}$ is equivalent to finding $w \in \mathcal{W}$

Hence, our general supervised learning problem can be formulated in terms of w as (say, $x_i \in \mathbb{R}^p, y_i \in \mathbb{R}^m$,$\ell : \mathbb{R}^{p+m} \to \mathbb{R}$ - loss function)

$$\hat{w} = \arg\min_{w \in \mathcal{W}} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(w, x_i)) + \lambda R(w) \tag{2}$$

How do we solve the preceding problem?

- When the problem is convex and we are lucky, we can find an explicit $\hat{w}$ by solving (e.g., generalized (Kernel) ridge regression)

$$\frac{\partial}{\partial w_i} \left(\frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(w, x_i)) + \lambda R(w) \right) = 0, \qquad i = 1, \ldots, k.$$

Parametric Supervised Learning

Let us denote the objective function

$$F(w) := \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(w, x_i)) + \lambda R(w)$$

In general, we use the following numerical algorithm:

1. Initialization: Pick an initial value w_0, possibly random
2. Iteration: Keep updating w in small steps Δw

$$w_{n+1} = w_n + \Delta w,$$

such that $F(w_{n+1}) < F(w_n)$.
Stoping criteria: $F(w_n) - F(w_{n+1}) < \epsilon$.

For the preceding procedure to find a local minimum

- We need to find a direction where F has a maximum decrease/steepest descent
- Avoid getting stuck on a flat surfaces, say flat saddle point

If F is convex, we can find a global minimum.

Recall Multi Calc: Gradient and Directional Derivatives

- Let $f(x) : \mathbb{R}^n \to \mathbb{R}, x = (x_1, \ldots, x_n)$
- **Gradient** is a vector of partial derivatives (assuming they exist)

$$\nabla f(x) := \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

- Let $u = (u_1, \ldots, u_n)$ be a unit vector ($\|u\|_2^2 = \sum u_i^2 = 1$)
- **Directional derivative** of $f(x)$ in direction u is

$$D_u f(x) := \frac{d}{dt} f(x+ut) = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} u_i = \nabla f(x) \cdot u = \|\nabla f(x)\|_2 \cos\theta,$$

  where $y \cdot z = \langle y, z \rangle$ is the dot product and θ is the angle between $\nabla f(x)$ and u.
- Hence, $-\|\nabla f(x)\|_2 \leq D_u f(x) \leq \|\nabla f(x)\|_2$, i.e.,
  $\nabla f(x)$: direction of steepest ascent/max increase of $f(x)$
  $-\nabla f(x)$: direction of steepest descent/max decrease of $f(x)$
  $\nabla f(x)$ is perpendicular to level curves $f(x) = c$ (prove this)

Path of Gradient Descent

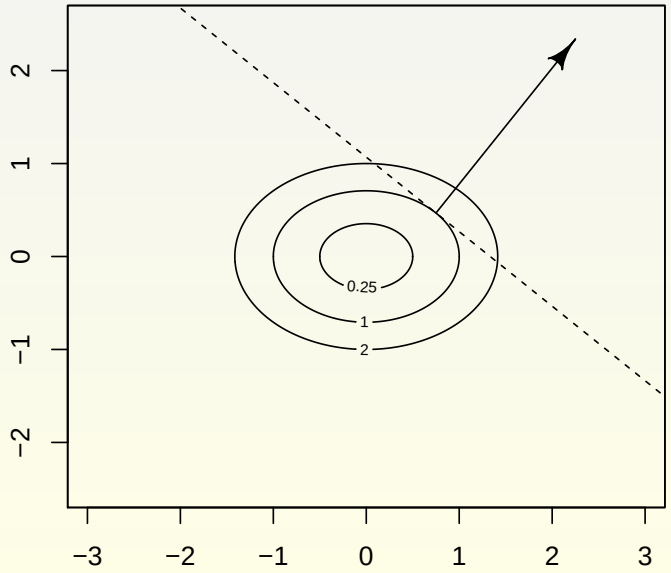
- $\mathbf{x}(t)$ - path of steepest gradient descent given initial condition $\mathbf{x}(0)$ is given by an ODE

$$\frac{d\mathbf{x}(t)}{dt} = -\eta \nabla f(\mathbf{x}(t)), \tag{3}$$

where η is the rate/speed of descent, a.k.a. learning rate. Note that $x'(t)$ is tangent to the curve $x(t)$, and thus parallel to $\nabla f(\mathbf{x}(t))$.

- Example: $f(\mathbf{x}) = ax_1^2 + bx_2^2, a, b > 0, \Rightarrow$
$\nabla f(\mathbf{x}) = (2ax_1, 2bx_2) \Rightarrow x_1'(t) = -2\eta a x_1(t), x_2'(t) = -2\eta b x_2(t)$

$$x_1(t) = x_1(0)e^{-2\eta a t}, \qquad x_2(t) = x_2(0)e^{-2\eta b t}$$



Gradient Descent Algorithm

GD Algorithm is a discrete linear approximation to Equation (3)

$$\frac{\mathbf{x}(t+\Delta t)-\mathbf{x}(t)}{\Delta t} \approx \frac{d\mathbf{x}(t)}{dt} = -\eta \nabla f(\mathbf{x}(t))$$

or equivalently (with a small abuse of notation)

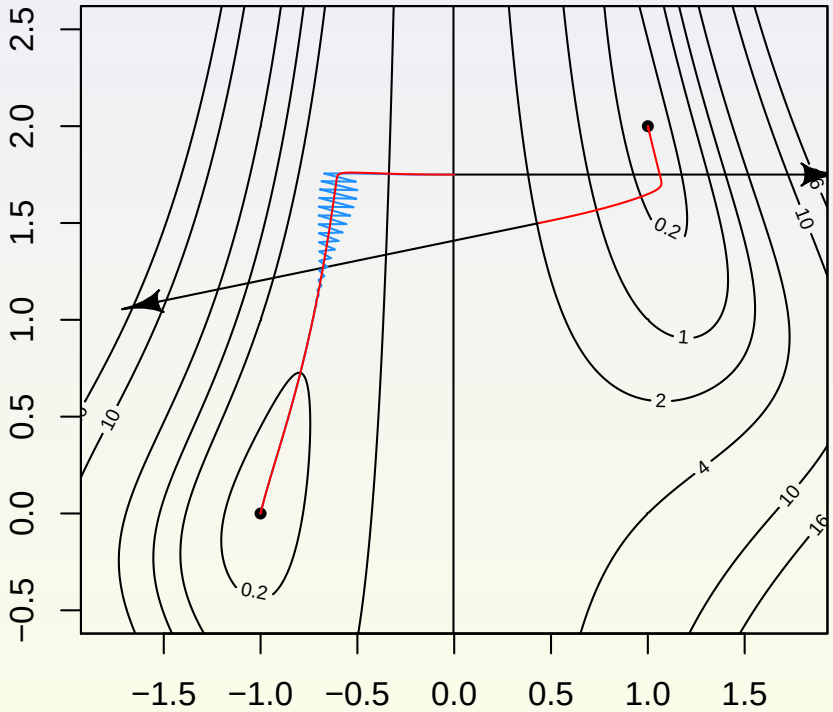
$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta_t \nabla f(\mathbf{x}^{(t)})$$

- Hence, after initialization at $\mathbf{x}^{(0)}$, the GD Algorithm follows the preceding iteration to a local minimum
- Stopping criterion (could be): $|f(\mathbf{x}^{(t+1)}) - f(\mathbf{x}^{(t)})| < \epsilon$

Adaptive GD (AdaGrad): modifies the learning rate η_t in each iteration t

More Interesting Landscape

- $f(x) = (x_1^2 - 1)^2 + (x_1^2 x_2 - x_1 - 1)^2$
- Gradient
$$\nabla f(x) = \begin{bmatrix} 4x_1(x_1^2 - 1) + 2(2x_1x_2 - 1)(x_1^2x_2 - x_1 - 1) \\ 2x_1^2(x_1^2x_2 - x_1 - 1) \end{bmatrix}$$
- Oscillations in "narrow valleys"

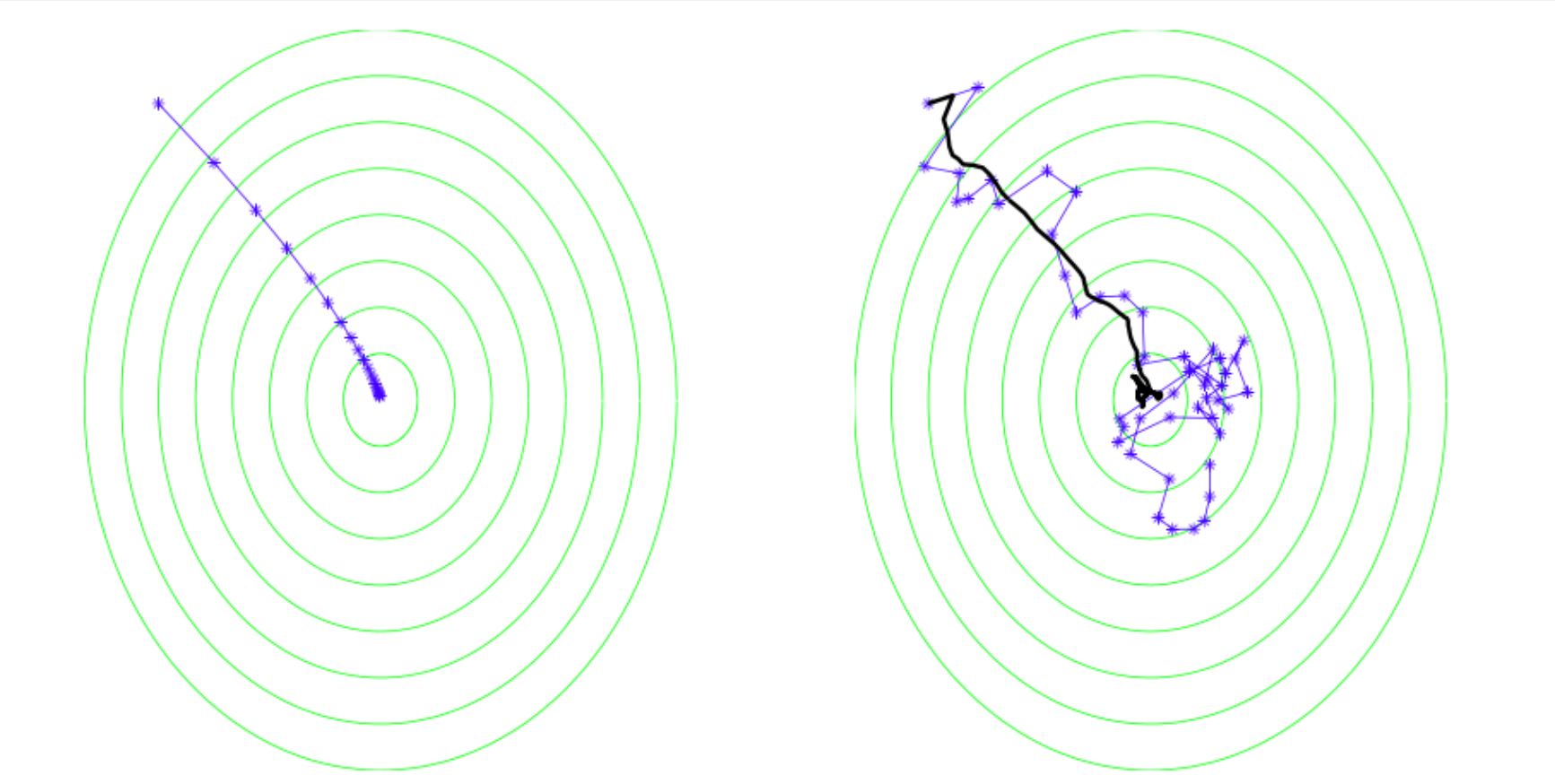


Motivation for momentum: remembers/averages previous Δx

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta_t \nabla f(\mathbf{x}^{(t)}) + \mu_t(\mathbf{x}^{(t)} - \mathbf{x}^{(t-1)})$$

Stochastic Gradient Descent

- Dates back to Robbins and Monroe (1951).
- Stochastic gradient is an unbiased estimator of the gradient
- Stochastic versus regular gradient descent



Stochastic Gradient Descent

- Stochastic approximation of gradient descent
- Function (typically encountered in learning)

$$f(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x)$$

- Computationally expensive gradient for large n
- Approximation: pick a random subset $\mathcal{S} \in [1, n]$

$$\nabla f(x) \approx \tilde{\nabla} f(x) = \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \nabla f_i(x)$$

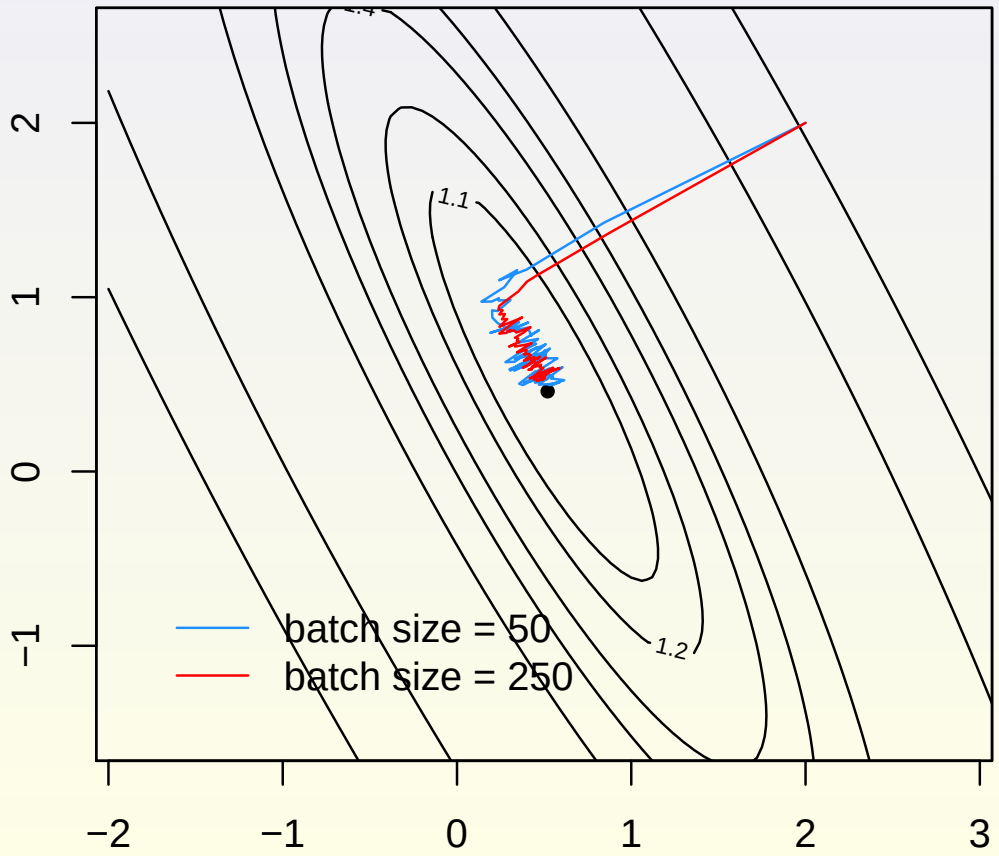
- $\mathcal{S}$ - called batch/mini-batch; $\tilde{\nabla}$ - stochastic gradient
- Example
  - n scalar data points $x_1, x_2, \cdots, x_n$
  - objective

$$\min_c \frac{1}{n} \sum_{i=1}^{n} (x_i - c)^2$$

SGD Example: Linear Regression

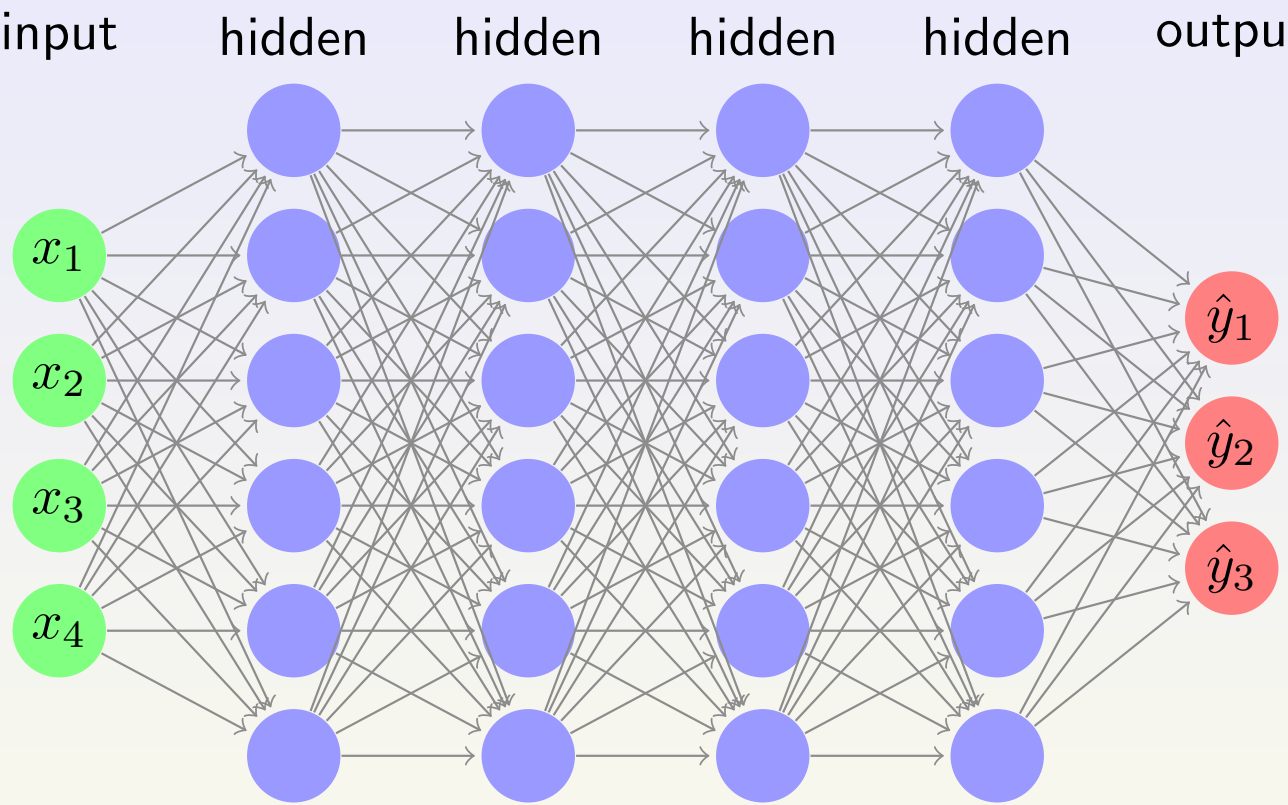
- Data: $(x_i, y_i)_{i=1}^n$, $n = 10^5$
- Loss function: $L(\beta, x, y) = \frac{1}{n}\sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_i)^2$
- Stochastic gradient

$$\nabla_\beta \hat{L}(\beta, x, y) = \frac{2}{|\mathcal{S}|}\sum_{i\in\mathcal{S}} \begin{bmatrix} (\beta_0 + \beta_1 x_i - y_i) \\ x_i(\beta_0 + \beta_1 x_i - y_i) \end{bmatrix}$$



Deep Neural Networks, a.k.a. Multilayer Perceptrons

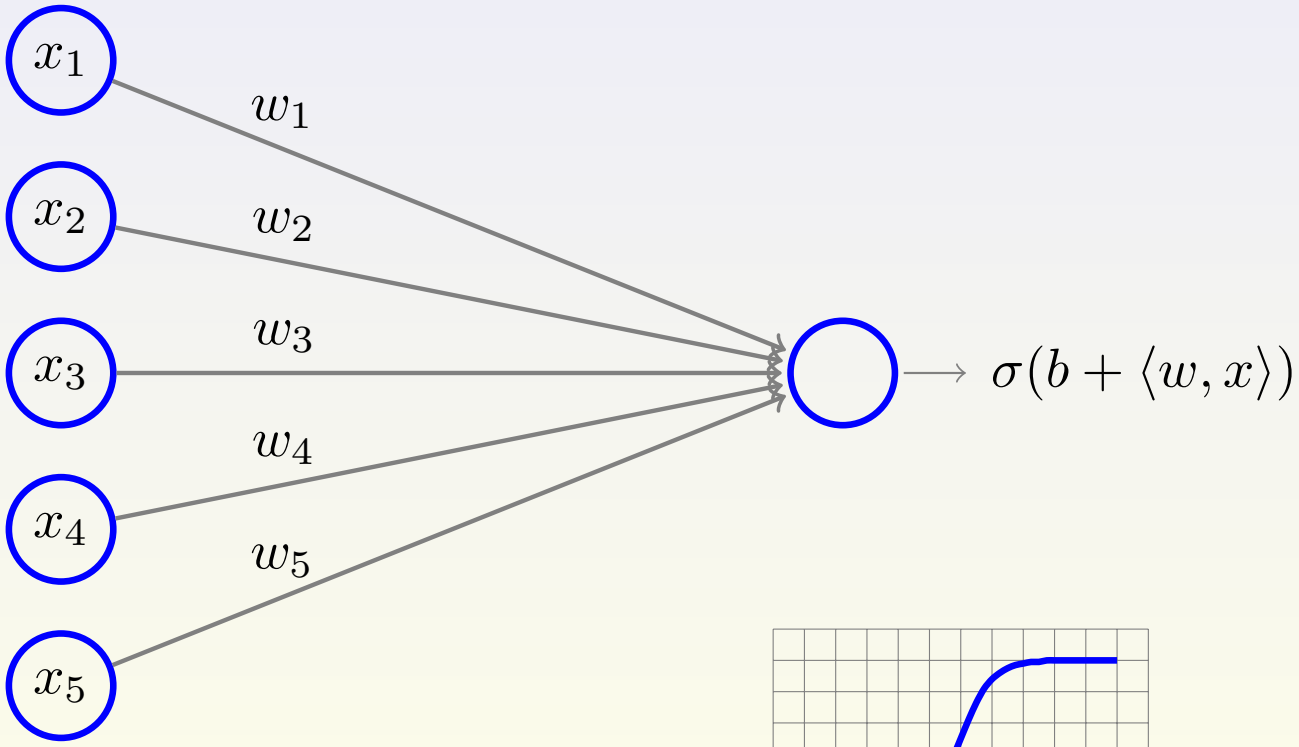
- Feed-forward network



- Designed to mimic the function of neurons
- Blue nodes: activation functions/neurons
- Depth $=$ lengths of a longest path
- Deep network: depth ≥ 3
- Very successful in solving practical problems

A Single Artificial Neuron

- A single neuron function: $\mathbf{x} \mapsto \sigma(b + \langle \mathbf{w}, \mathbf{x} \rangle)$, where $\sigma : \mathbb{R} \to \mathbb{R}$ is called the activation function of the neuron. Inner/dot product: $\langle \mathbf{w}, \mathbf{x} \rangle = \sum x_i w_i$.
- More compact notation $\langle \tilde{\mathbf{w}}, \tilde{\mathbf{x}} \rangle$, where $\tilde{\mathbf{x}} = (1, \mathbf{x}), \tilde{\mathbf{w}} = (b, \mathbf{w})$



- E.g., σ is a sigmoidal function

Common Activation Functions/Perceptrons

- step function

$$\sigma(x) = 1_{\{x>0\}} \qquad \sigma'(x) = 0,\ x \neq 0$$

- logistic

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

- rectified linear unit (ReLU)

$$\sigma(x) = \max\{x, 0\} \qquad \sigma'(x) = 1_{\{x>0\}},\ x \neq 0$$

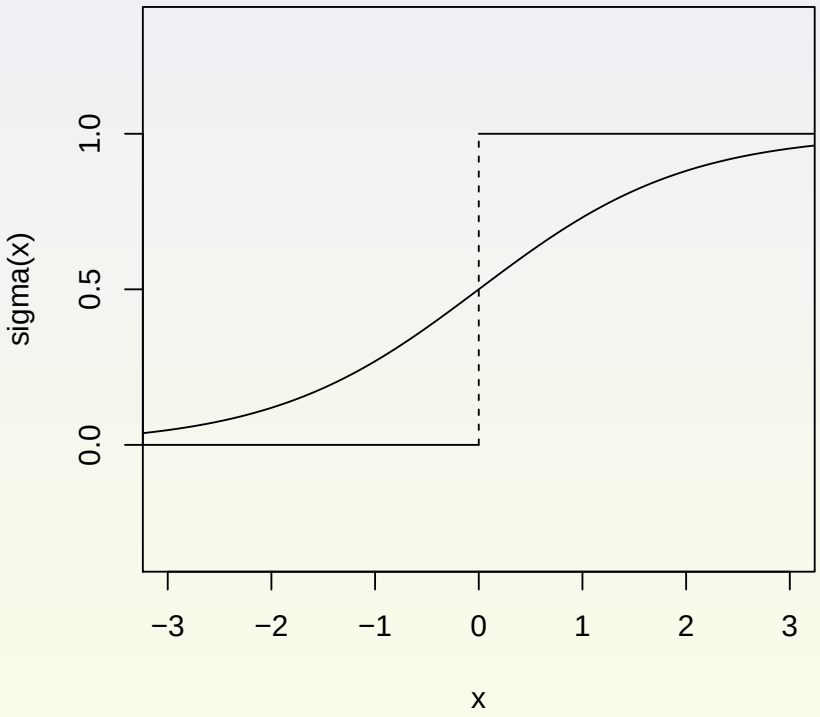
- soft-plus

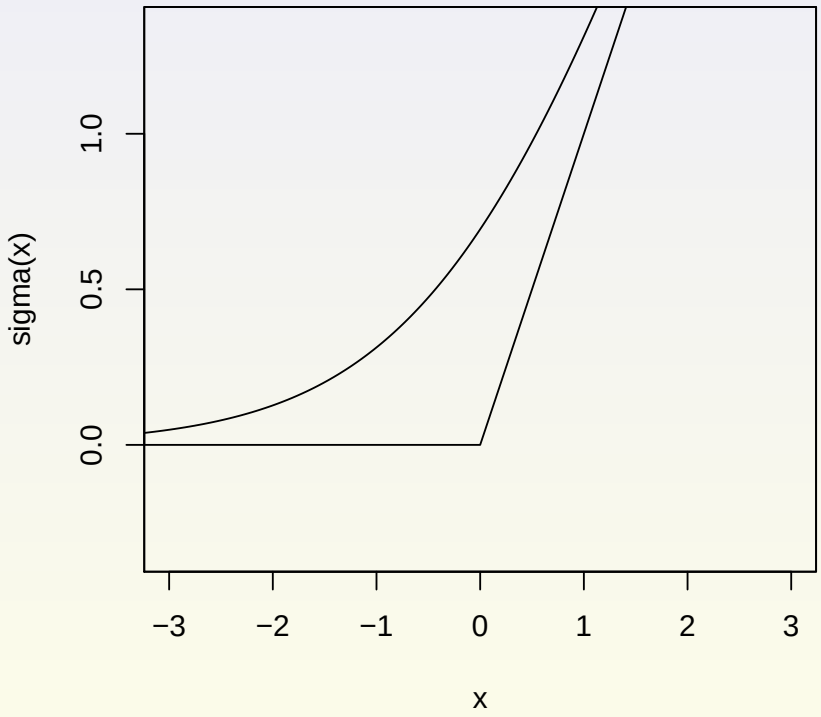
$$\sigma(x) = \log(1+e^{x}) \qquad \sigma'(x) = \frac{1}{1+e^{-x}}$$

Comparing Activation Functions

Logistic and soft-plus have continuous derivatives in comparison to step function and ReLU

- Positive derivative avoids vanishing gradient





Neural Network Notation: Graph Notation

This is a graph notation from the [UML] book; equivalently, we'll also use a matrix notation.

- A neural network is obtained by connecting many neurons
- We focus on feedforward networks, formally defined by a directed acyclic graph $G = (V, E)$
- Input nodes: nodes with no incoming edges
- Output nodes: nodes without outgoing edges
- Weights: $w[e] : E \to \mathbb{R}, W = \{w[e] : e \in E\}$
- Calculation in the network: each neuron (node) receives **input**

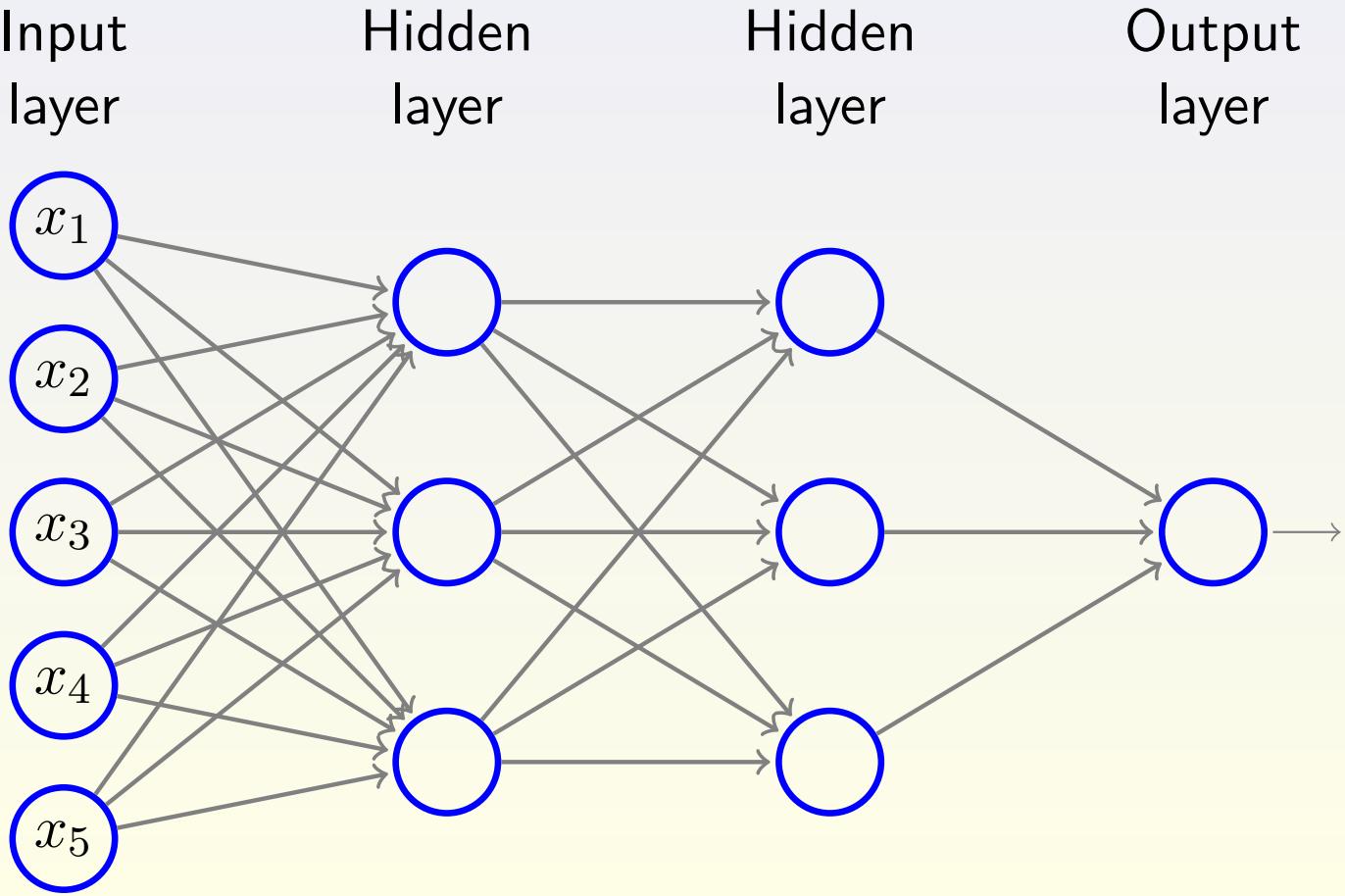
$$a[v] = \sum_{u:u \to v \in E} w[u \to v] o[u]$$

and yields **output**

$$o[v] = \sigma(a[v])$$

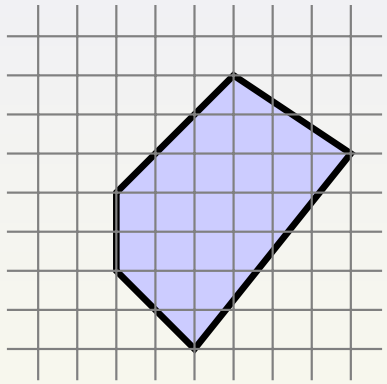
Multilayer Neural Networks

- Neurons are organized in layers: $V = \biguplus_{t=0}^{T} V_t$, and edges are only between adjacent layers
- Neural network specified by (V, E, σ, W)
- Example of a multilayer neural network of depth 3 and size 6



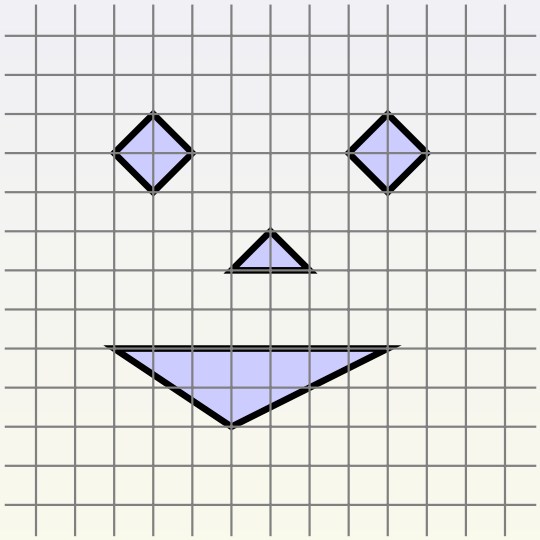
Example

- Single neuron is a binary half-space classifier: $\text{sign}(w \cdot x + b)$
- 2 hidden layer networks can express intersection of halfspaces



Example

- Networks with 3 hidden layers can express unions of intersection of halfspaces



How to train neural network?

- Neural nets: excellent hypothesis class, but difficult to train
- Main technique: Stochastic Gradient Descent (SGD)
- Not convex, no guarantees, can take a long time, but:
  - Often still works fine, finds a good solution
  - With some luck:)

Stochastic Gradient Descent (SGD) for Neural Networks

Common Training Ideas:

- **Random initialization**: rule of thumb, $w[u \to v] \sim U[-c, c]$ where $c = \sqrt{3/|\{(u', v) \in E\}|}$ (or small Gaussian instead of $U[-c, c]$)
- **SGD**: Update step with Nesterov's momentum: Initialize $\theta = 0$ and:

$$\theta_{t+1} = \mu_t \theta_t - \eta_t \tilde{\nabla} L(w_t + \mu_t \theta_t)$$
$$w_{t+1} = w_t + \theta_{t+1}$$

  where:
  μ_t is momentum parameter (e.g. $\mu_t = 0.9$ for all t)
  η_t is learning rate (e.g. $\eta_t = 0.01$ for all t)
  $\tilde{\nabla} L$ is an estimate of the gradient of L based on a small set of random examples (often called a "minibatch")

- Efficient gradient calculation: Backpropagation

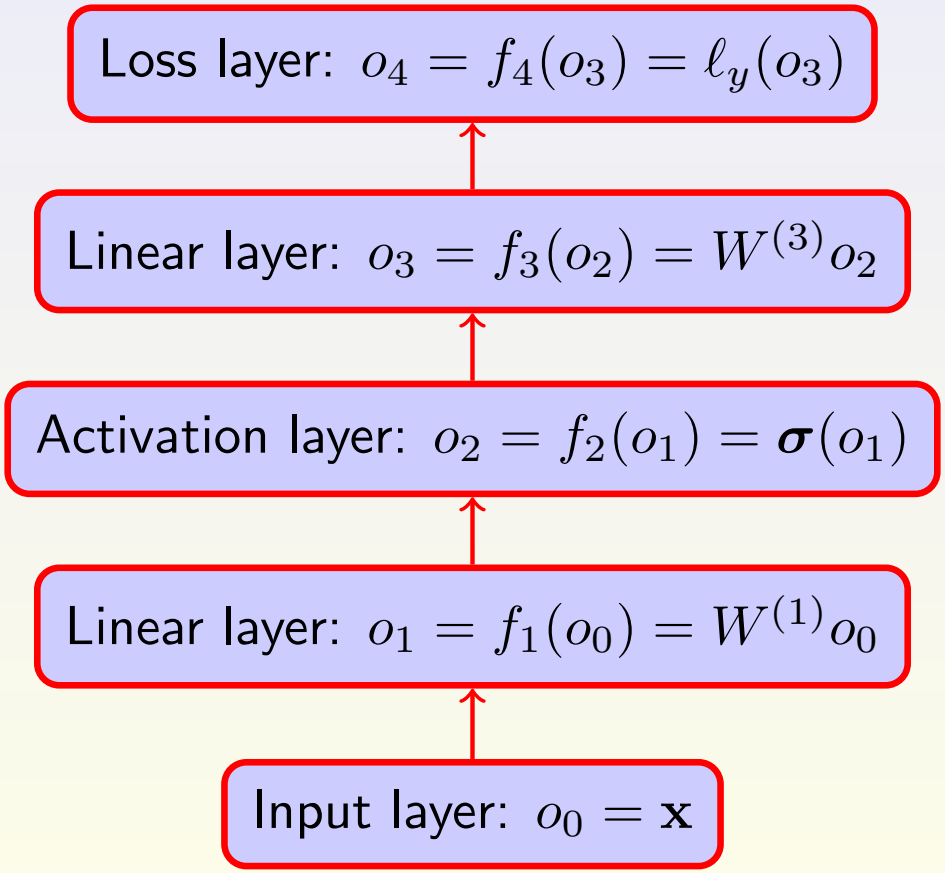
Back-Propagation

- The back-propagation algorithm is an efficient way to calculate $\nabla \ell(h_w(x), y)$ using the chain rule
- Let $\mathbf{f}(\mathbf{w}) : \mathbb{R}^n \to \mathbb{R}^m$, $\mathbf{f} = (f_1, \ldots f_m)$, $f_i(\mathbf{w}) : \mathbb{R}^n \to \mathbb{R}$
- $J_{\mathbf{w}}(\mathbf{f})$ - Jacobian of $\mathbf{f}(\mathbf{w})$ is the $m \times n$ matrix whose i, j element is the partial derivative of $\partial f_i(\mathbf{w}) / \partial w_j$
  i.e., ith row of $J_{\mathbf{w}}(\mathbf{f})$ is equal to $\nabla f_i(\mathbf{w})$
  Examples
  - If $\mathbf{f}(\mathbf{w}) = A\mathbf{w}$ then $J_{\mathbf{w}}(\mathbf{f}) = A$.
  - If $\boldsymbol{\sigma} : \mathbb{R}^n \to \mathbb{R}^n$ is element-wise application of $\sigma : \mathbb{R} \to \mathbb{R}$ then $J_{\boldsymbol{\theta}}(\boldsymbol{\sigma}) = \mathrm{diag}((\sigma'(\theta_1), \ldots, \sigma'(\theta_n)))$.
- Chain rule:

$$J_{\mathbf{w}}(\mathbf{f} \circ \mathbf{g}) = J_{g(\mathbf{w})}(\mathbf{f}) J_{\mathbf{w}}(\mathbf{g})$$

Back-Propagation

Let $\ell_y : \mathbb{R}^k \to \mathbb{R}$ be the loss function at the output layer.
It's convenient to describe the network as a sequence of simple layer functions:



Back-Propagation

- Can write $\ell(h_{\mathbf{w}}, (\mathbf{x}, y)) = (f_{T+1} \circ \ldots \circ f_3 \circ f_2 \circ f_1)(\mathbf{x})$
- Denote $F_t = f_{T+1} \circ \ldots \circ f_{t+1}$ and $\delta_t = J_{o_t}(F_t)$, then

$$\begin{aligned}\delta_t &= J_{o_t}(F_t) = J_{o_t}(F_{t-1} \circ f_{t+1}) \\ &= J_{f_{t+1}(o_t)}(F_{t-1}) J_{o_t}(f_{t+1}) = J_{o_{t+1}}(F_{t-1}) J_{o_t}(f_{t+1}) \\ &= \delta_{t+1} J_{o_t}(f_{t+1})\end{aligned}$$

- Note that

$$J_{o_t}(f_{t+1}) = \begin{cases} W^{(t+1)} & \text{for linear layer} \\ \operatorname{diag}(\boldsymbol{\sigma}'(o_t)) & \text{for activation layer} \end{cases}$$

- Using the chain rule again we obtain

$$J_{W^{(t)}}(\ell(h_{\mathbf{w}}, (\mathbf{x}, y))) = \delta_t o_{t-1}^\top$$

Back-Propagation: Pseudo-code

Forward:

- set $o_0 = \mathbf{x}$ and for $t = 1, 2, \ldots, T$ set

$$o_t = f_t(o_{t-1}) = \begin{cases} W^{(t)} o_{t-1} & \text{for linear layer} \\ \boldsymbol{\sigma}(o_{t-1}) & \text{for activation layer} \end{cases}$$

Backward:

- set $\delta_{T+1} = \nabla \ell_y(o_T)$ and for $t = T, T-1, \ldots, 1$ set

$$\delta_t = \delta_{t+1} J_{o_t}(f_{t+1}) = \delta_{t+1} \cdot \begin{cases} W^{(t+1)} & \text{for linear layer} \\ \operatorname{diag}(\boldsymbol{\sigma}'(o_t)) & \text{for activation layer} \end{cases}$$

- For linear layers, set the gradient w.r.t. the weights in $W^{(t)}$ to be the elements of the matrix $\delta_t o_{t-1}^\top$

Matrix Notation

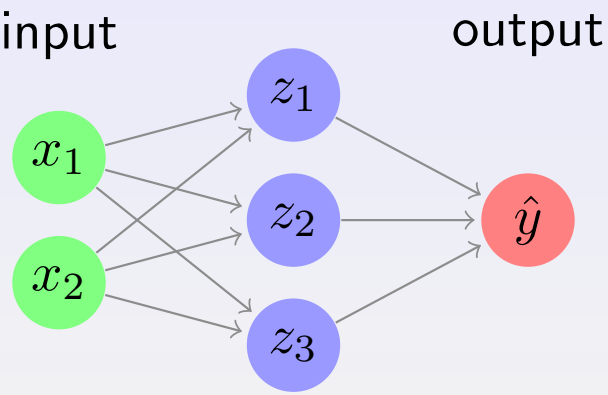
- Vector multiplication:
  - $w, x \in \mathbb{R}^m$
  - $w^\top x = w \cdot x = \sum_{i=1}^{m} w_i x_i$
- Single neuron: $\sigma(w^\top x)$
- First layer
  - different weights for different neurons
  - same input x
  - matrix notation

$$\begin{bmatrix} w_1^\top \\ \vdots \\ w_k^\top \end{bmatrix} x = Wx = \begin{bmatrix} w_1^\top x \\ \vdots \\ w_k^\top x \end{bmatrix}$$

  - matrix of layer weights W
  - layer output

$$f(Wx) = \begin{bmatrix} \sigma(w_1^\top x) \\ \vdots \\ \sigma(w_k^\top x) \end{bmatrix}$$

Example: Simple network

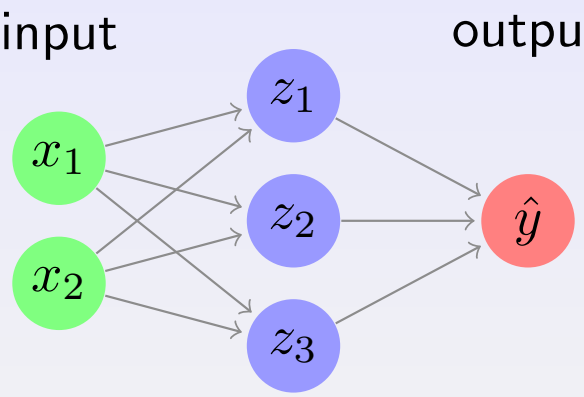


- Network: $\boldsymbol{x} \mapsto \hat{y} = f_2(W^2 f_1(W^1 \boldsymbol{x}))$, where

$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} \\ w^1_{21} & w^1_{22} \\ w^1_{31} & w^1_{32} \end{bmatrix} \quad \text{and} \quad W^2 = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \end{bmatrix}$$

- FF network: $\boldsymbol{x} \mapsto f_k(W^k f_{k-1}(\cdots f_2(W^2 f_1(W^1 \boldsymbol{x}))\cdots))$

Example: Simple network



- Network: $\boldsymbol{x} \mapsto \hat{y} = f_2(W^2 f_1(W^1 \boldsymbol{x}))$
- Training:

$$\min_{\text{weights } W^1, W^2} L = \sum_{i=1}^{n} \ell(y_i, \hat{y}_i)$$

  - gradient descent
  - data points $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$
  - evaluate gradient of $f_2(W^2 f_1(W^1 \boldsymbol{x}_i))$
  - gradient evaluated at fixed data points $\boldsymbol{x}_i$

Backpropagation

- Simple network: Chain rule

$$\frac{\partial L}{\partial w_{21}^1} = \sum_{i=1}^{n} \frac{\partial \ell(y_i, \hat{y}_i)}{\partial \hat{y}_i} f_2'(W^2 f_1(W^1 \boldsymbol{x}_i))\, w_{12}^2\, f_1'(w_{21}^1 x_{i1} + w_{22}^1 x_{i2})\, x_{i1}$$

$$\frac{\partial L}{\partial w_{13}^2} = \sum_{i=1}^{n} \frac{\partial \ell(y_i, \hat{y}_i)}{\partial \hat{y}_i} f_2'(W^2 f_1(W^1 \boldsymbol{x}_i))\, f_1(w_{31}^1 x_{i1} + w_{32}^1 x_{i2})$$

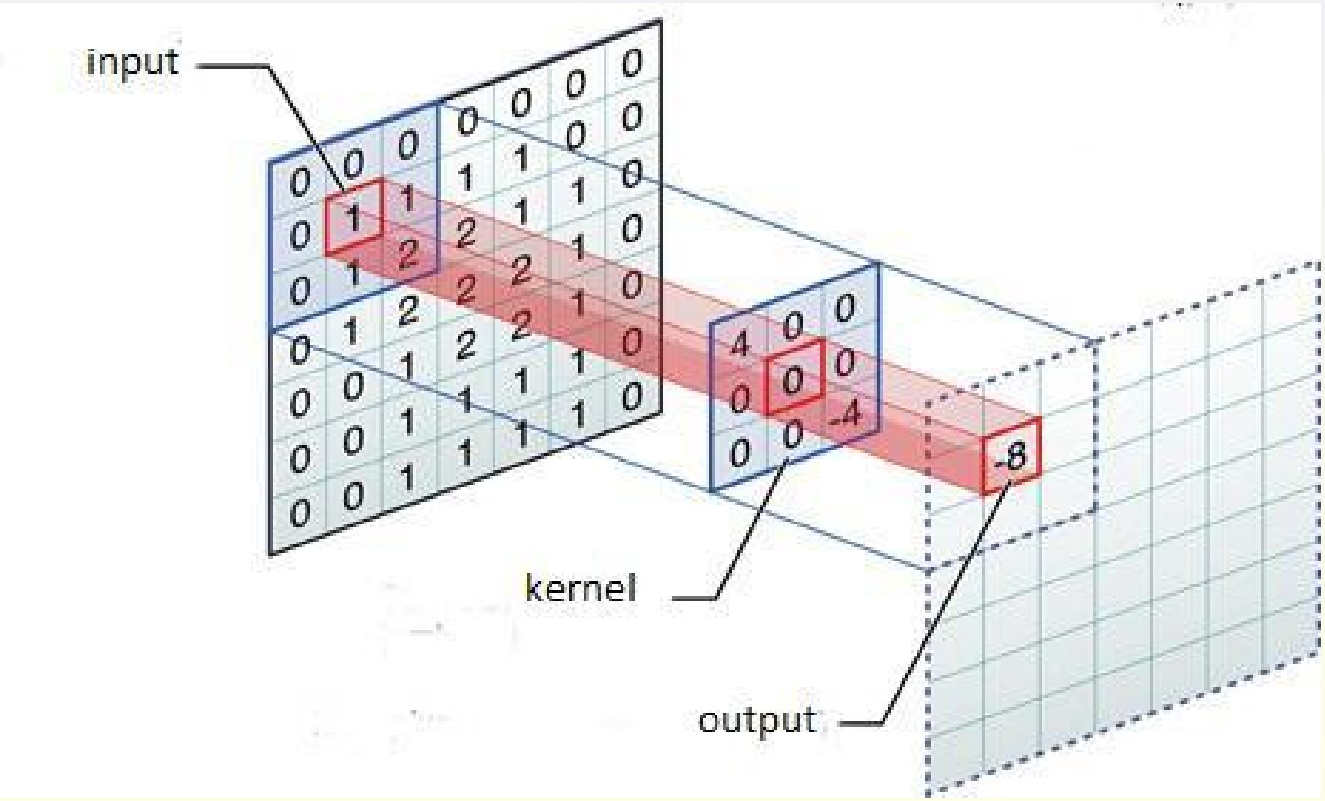
- Backpropagation = efficient implementation of chain rule

Detour: Convolutional Networks

Designed for computer vision problems

Main ideas:

- Convolutional layers: reuse the same weights on different patches of the image
- Pooling layers: decrease image resolution (good for translation invariance, higher level features, and runtime)



Neural Networks: Recent Trends

- ReLU activation: $\sigma(a) = \max\{0, a\}$. This helps convergence, but do not hurt expressiveness
- Very large networks: often, the number of parameters is very large, even much larger than the number of examples. This might lead to overfitting, which is (partially) avoided by many types of regularization
- Regularization: besides norm regularization, early stopping of SGD also serves as a regularizer
- Dropout: this is another form of regularization, in which some neurons are "muted" at random during training
- Weight sharing (convolutional networks)
- SGD tricks: momentum, Nesterov's acceleration, other forms of second order approximation
- Training on GPUs!

Historical Remarks

- 1940s-70s:
  - Inspired by learning/modeling the brain (Pitts, Hebb, and others)
  - Perceptron Rule (Rosenblatt), Multilayer perceptron (Minksy and Papert)
  - Backpropagation (Werbos 1975)
- 1980s – early 1990s:
  - Practical Back-prop (Rumelhart et al 1986) and SGD (Bottou)
  - Initial empirical success
- 1990s-2000s:
  - Lost favor to implicit linear methods: SVM, Boosting
- 2006 –:
  - Regain popularity because of unsupervised pre-training (Hinton, Bengio, LeCun, Ng, and others)
  - Computational advances and several new tricks allow training HUGE networks. Empirical success leads to renewed interest
  - 2012: Krizhevsky, Sustkever, Hinton: significant improvement of state-of-the-art on imagenet dataset (object recognition of 1000 classes), without unsupervised pre-training

Next Lecture: Expressive Power of Neural Nets

Approximation theory perspective

Papers: We will start with

- 2-layer (1 hidden) networks are universal approximators:
  - Approximations by superpositions of sigmoidal functions, by Cybenko, 1989.
  - Approximation capabilities of multilayer feedforward networks, by Hornik, 1991.
  - Universal approximation bounds for super- positions of a sigmoidal function, by Barron 1993.
- Depth separation results: Can deep neural networks of depth $(d+1)$ express functions much more efficiently in terms of the number of neurons compared to networks of depth d?
  - Representation Benefits of Deep Feedforward Networks, by Telgarsky, 2015.
  - The Power of Depth for Feedforward Neural Networks, by Eldan and Shamir, 2016
- More papers to follow...

Reading/browsing:

[UML] Chapter 14: Stochastic Gradient Descent
Chapter 20: Neural Networks

[DL] Chapter 6: Deep Feedforward Networks

Software: Download R, R Studio and TensorFlow
Or, equivalent for Python

Good book on optimization

Convex Optimization, Stephen Boyd and Lieven Vandenberghe, Cambridge University Press, 2004. (Click blue text for free pdf.)