

Mathematical Background

Vahid Tarokh
ECE685D
Fall 2025

Quick Review of Linear Algebra

Introduction

- In order to design and implement deep networks we need to know
 - Basics of Linear Algebra
 - Basics of Multivariable Calculus
 - Basics of Probability
- For writing research papers, you may need to know more.
- Here, we will include some of the background for completeness, but will not teach them all.
- Source: [Dive into Deep Learning](#)
 - Professor Smola's Slides
 - Professor David Carlson's Slides
 - Professor Lawrence Caron's slides
 - Professor Ruslan Salakhutdinov's slides (available online)

Scalars

- Simple operations

$$c = a + b$$

$$c = a \cdot b$$

$$c = \sin a$$

- Length

$$|a| = \begin{cases} a & \text{if } a > 0 \\ -a & \text{otherwise} \end{cases}$$

$$|a + b| \leq |a| + |b|$$

$$|a \cdot b| = |a| \cdot |b|$$

Vectors



- Simple operations

$$c = a + b \quad \text{where } c_i = a_i + b_i$$

$$c = \alpha \cdot b \quad \text{where } c_i = \alpha b_i$$

$$c = \sin a \quad \text{where } c_i = \sin a_i$$

- Length

Definition of a vector space

Definition of norm

$$\|a\|_2 = \left[\sum_{i=1}^m a_i^2 \right]^{\frac{1}{2}}$$

$$\|a\| \geq 0 \text{ for all } a$$

$$\|a + b\| \leq \|a\| + \|b\|$$

$$\|a \cdot b\| = |a| \cdot \|b\|$$

Vectors



$$c = a + b$$



$$c = \alpha \cdot b$$

Mathematician's 'parallel for all do'

Vectors



- Dot product

$$a^\top b = \sum_i a_i b_i$$

- Orthogonality

$$a^\top b = \sum_i a_i b_i = 0$$

(e.g. if we have two vectors that are orthogonal with a third, their linear combination is it, too)

Matrices



- Simple operations

$$C = A + B \quad \text{where } C_{ij} = A_{ij} + B_{ij}$$

$$C = \alpha \cdot B \quad \text{where } C_{ij} = \alpha B_{ij}$$

$$C = \sin A \quad \text{where } C_{ij} = \sin A_{ij}$$

- Functional Analysis 101

vector = function, matrix = linear operator

most theorems work sort-of in infinite dimensional spaces

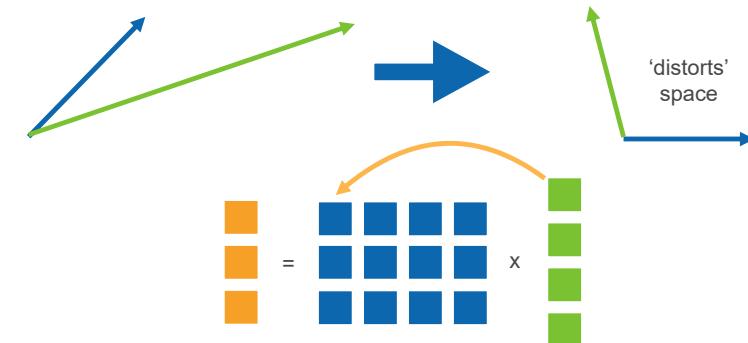
Matrices

- Multiplications (matrix vector)

$$c = Ab \text{ where } c_i = \sum_j A_{ij} b_j$$



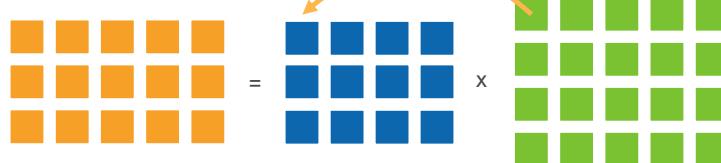
Matrices



Matrices

- Multiplications (matrix matrix)

$$C = AB \text{ where } C_{ik} = \sum_j A_{ij} B_{jk}$$



Matrices

- Norms

$$c = A \cdot b \text{ hence } \|c\| \leq \|A\| \cdot \|b\|$$

- Choices depending on how to measure length of b and c

- A Popular norm

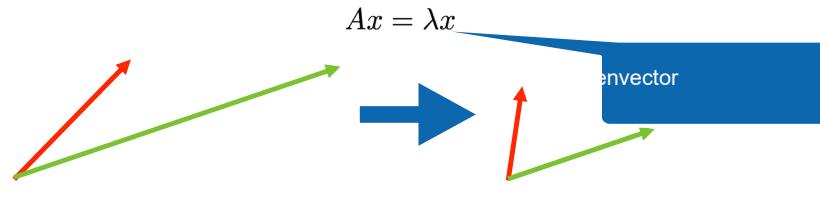
- Frobenius norm

$$\|A\|_{\text{Frob}} = \left[\sum_{ij} A_{ij}^2 \right]^{\frac{1}{2}}$$

Matrices

- Eigenvectors and eigenvalue

- Vectors that aren't changed by the matrix



- For symmetric matrices we can always find this

Special Matrices

- Symmetric, antisymmetric $A_{ij} = A_{ji}$ and $A_{ij} = -A_{ji}$



- Non-negative definite

$\|x\|^2 = x^\top x \geq 0$ generalizes to $x^\top Ax \geq 0$

(all non-negative eigenvalues)

Special Matrices

- Orthogonal Matrices

- All rows of the matrix are orthogonal to each other
- All rows of the matrix have unit length

$$U \text{ with } \sum_j U_{ij} U_{kj} = \delta_{ik}$$

Show that
 $U^\top U = \mathbf{1}$

- Rewrite in matrix form

$$UU^\top = \mathbf{1}$$

Show that P is
orthogonal

- Permutation Matrices

$$P \text{ where } P_{ij} = 1 \text{ if and only if } j = \pi(i)$$

Multidimensional Arrays

N-dimensional Array Examples

N-dimensional array, short for ndarray, is the main data structure for machine learning and neural networks

0-d (scalar)



1.0

A class label

1-d (vector)



[1.0, 2.7, 3.4]

A feature vector

2-d (matrix)



[[1.0, 2.7, 3.4]
[5.0, 0.2, 4.6]
[4.3, 8.5, 0.2]]
[[3.2, 5.7, 3.4]
[5.4, 6.2, 3.2]
[4.1, 3.5, 6.2]]]

A example-by-
feature matrix

ND Array Examples, cont

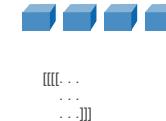
3-d



[[[0.1, 2.7, 3.4]
[5.0, 0.2, 4.6]
[4.3, 8.5, 0.2]]]
[[3.2, 5.7, 3.4]
[5.4, 6.2, 3.2]
[4.1, 3.5, 6.2]]]]

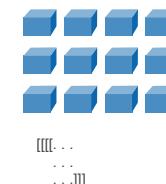
A RGB image
(width x height
x channels)

4-d



A batch of
RGB images
(batch-size x time
x width x height
x channels)

5-d



A batch of videos
(batch-size x time x
width x height x
channels)

Access Elements

An element: [1, 2]

0	1	2	3
1	5	6	7
2	9	10	11
3	13	14	15

A row: [1, :]

0	1	2	3
1	1	2	3
2	5	6	7
3	9	10	11

A column: [1, :]

0	1	2	3
1	1	2	3
2	5	6	7
3	9	10	11

0 1 2 3

0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

Review of Basic Probability

Probability

Space of events X

- server working; slow response; server broken
- income of the user (e.g. \$95,000)
- query text for search (e.g. "statistics tutorial")

Probability axioms

$$\Pr(X) \in [0, 1], \Pr(\mathcal{X}) = 1$$
$$\Pr(\cup_i X_i) = \sum_i \Pr(X_i) \text{ if } X_i \cap X_j = \emptyset$$

Example queries

- $\Pr(\text{server working}) = 0.999$
- $\Pr(90,000 < \text{income} < 100,000) = 0.1$

discrete

continuous

What you must know

- Definitions of random variable and random vector
- Conditional probability, Independence, and dependence
- Law of Total Probability
- Definition of PMF, PDF, and CDF
- Generation of an arbitrary random variable with a given pdf from the uniform random variable
- PMF and PDF of transforms of random vectors
- Mathematical Expectation
- Variance, Covariance, correlation, etc.
- Multivariable Calculus and Lagrange's multiplier method

(In)dependence

Independence

- Login behavior of two users (approximately)
- Disk crash in different colos (approximately)

independent events

$$\Pr(x, y) = \Pr(x) \cdot \Pr(y)$$

Everywhere

- Emails
- Queries
- News stream / Buzz / Tweets
- IM communication
- Russian Roulette

$$\Pr(x, y) \neq \Pr(x) \cdot \Pr(y)$$

Weak Low of Large Numbers

Let X_1, X_2, \dots, X_n be i.i.d. random variable with mean μ . Then for any $\epsilon > 0$

$$P[|(X_1 + X_2 + \dots + X_n)/n - \mu| > \epsilon] \rightarrow 0$$

as $n \rightarrow \infty$.

Order Statistics

For X_1, X_2, \dots, X_n iid random variables X_k is the k th smallest X , usually called the k th order statistic.

$X_{(1)}$ is therefore the smallest X and

$$X_{(1)} = \min(X_1, \dots, X_n)$$

Similarly, $X_{(n)}$ is the largest X and

$$X_{(n)} = \max(X_1, \dots, X_n)$$

Order Statistics (density of maximum)

For X_1, X_2, \dots, X_n iid continuous random variables with pdf f and cdf F the density of the maximum is

$$\begin{aligned} P(X_{(n)} \in [x, x + \epsilon]) &= P(\text{one of the } X\text{'s} \in [x, x + \epsilon] \text{ and all others} < x) \\ &= \sum_{i=1}^n P(X_i \in [x, x + \epsilon] \text{ and all others} < x) \\ &= nP(X_1 \in [x, x + \epsilon] \text{ and all others} < x) \\ &= nP(X_1 \in [x, x + \epsilon])P(\text{all others} < x) \\ &= nP(X_1 \in [x, x + \epsilon])P(X_2 < x) \cdots P(X_n < x) \\ &= nf(x)\epsilon F(x)^{n-1} \\ f_{(n)}(x) &= nf(x)F(x)^{n-1} \end{aligned}$$

Order Statistics (density of minimum)

For X_1, X_2, \dots, X_n iid continuous random variables with pdf f and cdf F the density of the minimum is

$$\begin{aligned} P(X_{(1)} \in [x, x + \epsilon]) &= P(\text{one of the } X\text{'s} \in [x, x + \epsilon] \text{ and all others} > x) \\ &= \sum_{i=1}^n P(X_i \in [x, x + \epsilon] \text{ and all others} > x) \\ &= nP(X_1 \in [x, x + \epsilon] \text{ and all others} > x) \\ &= nP(X_1 \in [x, x + \epsilon])P(\text{all others} > x) \\ &= nP(X_1 \in [x, x + \epsilon])P(X_2 > x) \cdots P(X_n > x) \\ &= nf(x)\epsilon(1 - F(x))^{n-1} \\ f_{(1)}(x) &= nf(x)(1 - F(x))^{n-1} \end{aligned}$$

Order Statistics (density of k-th)

For X_1, X_2, \dots, X_n iid continuous random variables with pdf f and cdf F the density of the k th order statistic is

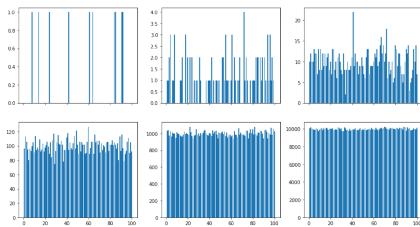
$$\begin{aligned} P(X_{(k)} \in [x, x + \epsilon]) &= P(\text{one of the } X\text{'s} \in [x, x + \epsilon] \text{ and exactly } k-1 \text{ of the others} < x) \\ &= \sum_{i=1}^n P(X_i \in [x, x + \epsilon] \text{ and exactly } k-1 \text{ of the others} < x) \\ &= nP(X_1 \in [x, x + \epsilon] \text{ and exactly } k-1 \text{ of the others} < x) \\ &= nP(X_1 \in [x, x + \epsilon])P(\text{exactly } k-1 \text{ of the others} < x) \\ &= nP(X_1 \in [x, x + \epsilon]) \left(\binom{n-1}{k-1} P(X < x)^{k-1} P(X > x)^{n-k} \right) \\ f_{(k)}(x) &= nf(x) \left(\binom{n-1}{k-1} F(x)^{k-1} (1 - F(x))^{n-k} \right) \end{aligned}$$

Uniform Distribution

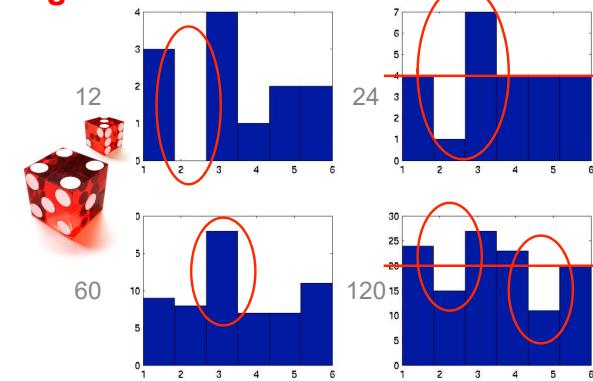
- Constant within an interval, zero outside

$$p(x) = \frac{1}{U - L} \text{ if } L \leq x \leq U$$

- Useful for initializing parameters or for load distribution



Tossing a Fair Dice



Euler's Gamma Function

For $x > 0$ The Euler's gamma function is defined as:

$$\Gamma(x) \equiv \int_0^{\infty} u^{x-1} e^{-u} du.$$

- The value $\Gamma(n) = (n - 1)!$ When $n > 0$ is a positive integer.
- Show that $\Gamma(x) = (x - 1)\Gamma(x - 1)$ for $x > 1$.
- Calculate $\Gamma(3/2)$
- Calculate $\Gamma(1/2)$

Beta Distribution

- We define a distribution for a parameter $\mu \in [0, 1]$

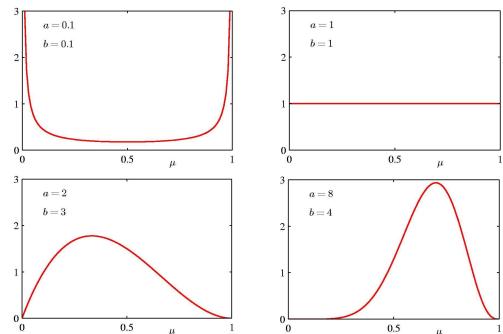
$$\begin{aligned}\text{Beta}(\mu|a, b) &= \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \mu^{a-1} (1-\mu)^{b-1} \\ \mathbb{E}[\mu] &= \frac{a}{a+b} \\ \text{var}[\mu] &= \frac{ab}{(a+b)^2(a+b+1)}\end{aligned}$$

where the Euler's gamma function is defined as:

$$\Gamma(x) \equiv \int_0^{\infty} u^{x-1} e^{-u} du.$$

and ensures that the Beta distribution is normalized.

Beta Distribution



Relationship Between Beta and Uniform

Let $X_1, X_2, \dots, X_n \stackrel{iid}{\sim} \text{Unif}(0, 1)$ then the density of $X_{(n)}$ is given by

$$f_{(k)}(x) = nf(x) \binom{n-1}{k-1} F(x)^{k-1} (1-F(x))^{n-k}$$

$$= \begin{cases} n \binom{n-1}{k-1} x^{k-1} (1-x)^{n-k} & \text{if } 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

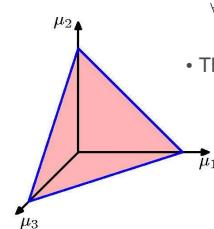
This is an example of the Beta distribution where $r = k$ and $s = n - k + 1$.

$$X_{(k)} \sim \text{Beta}(k, n - k + 1)$$

Dirichlet Distribution

- Consider a distribution over the K-dimensional simplex, subject to constraints:

$$\forall k : \mu_k \geq 0 \quad \text{and} \quad \sum_{k=1}^K \mu_k = 1$$



- The Dirichlet distribution is defined as:

$$\text{Dir}(\boldsymbol{\mu}|\boldsymbol{\alpha}) = \frac{\Gamma(\alpha_0)}{\Gamma(\alpha_1) \cdots \Gamma(\alpha_K)} \prod_{k=1}^K \mu_k^{\alpha_k-1}$$

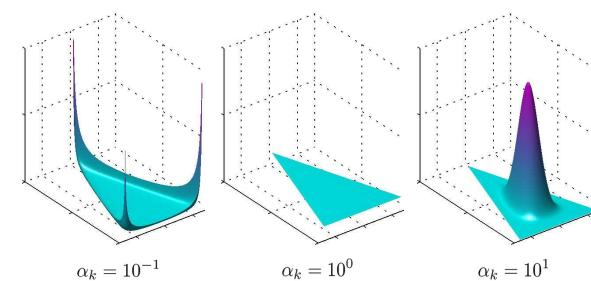
$$\alpha_0 = \sum_{k=1}^K \alpha_k$$

where $\alpha_1, \dots, \alpha_k$ are the parameters of the distribution, and $\Gamma(x)$ is the gamma function.

- The Dirichlet distribution is confined to a simplex as a consequence of the constraints.

Dirichlet Distribution

- Plots of the Dirichlet distribution over three variables.



Generation of Dirichlet from Beta

- Consider a Stick of length 1.

Simulate a random variate $X_j \sim Beta(\alpha_j, \sum_{i=j+1}^k \alpha_i)$, where $j = 1, \dots, k-1$. When $j = 1$, we have $X_1 \sim Beta(\alpha_1, \sum_{i=2}^k \alpha_i)$. The first piece of the stick has length $1 \cdot X_1$, such that the length of the remaining stick is $1 - X_1$. Also, set $Y_1 = X_1$.

Generation of Dirichlet from Beta

When $j = 2$, we have $X_2 \sim Beta(\alpha_2, \sum_{i=3}^k \alpha_i)$. The second piece of the stick has length $(1 - X_1)X_2$, such that the length of the remaining stick is $(1 - X_1) - (1 - X_1)X_2 = (1 - X_1)(1 - X_2)$. Also, set $Y_2 = (1 - X_1)X_2$.

⋮
Continue in this way.

Generation of Dirichlet from Beta

When $j = k-1$, we have $X_{k-1} \sim Beta(\alpha_{k-1}, \alpha_k)$. The $(k-1)^{th}$ piece of the stick has length $X_{k-1} \prod_{j=1}^{k-2} (1 - X_j)$, such that the length of the remaining stick is $\prod_{j=1}^{k-1} (1 - X_j)$. Also, set $Y_{k-1} = X_{k-1} \prod_{j=1}^{k-2} (1 - X_j)$. Note that the k^{th} piece of the stick has length $\prod_{j=1}^{k-1} (1 - X_j)$ and set $Y_k = \prod_{j=1}^{k-1} (1 - X_j)$. We can conclude that $(Y_1, \dots, Y_k) \sim Dir(\alpha_1, \dots, \alpha_k)$.

Source: Bela A Frigyik, Amol Kapila, and Maya R Gupta. Introduction to the Dirichlet distribution and related processes. Technical report, UWEETR-2010-0006, 2010

Entropy

The [entropy](#) of a d -dimensional random vector $\mathbf{X} := [X_1 \ \dots \ X_d]^T$ is defined by the expectation of the self information

$$H(\mathbf{X}) := \mathbb{E}_{\mathbf{X}} \left[\log \frac{1}{p(\mathbf{X})} \right] = \sum_{\mathbf{x} \in \mathcal{X}_1 \times \dots \times \mathcal{X}_d} p(\mathbf{x}) \log \frac{1}{p(\mathbf{x})} = H(X_1, \dots, X_d).$$

The [conditional entropy](#) of X given Y is defined by

$$H(X|Y) := \sum_{y \in \mathcal{Y}} p(y) H(X|Y=y) = \sum_{x \in \mathcal{X}, y \in \mathcal{Y}} p(x, y) \log \frac{1}{p_{X|Y}(x|y)}.$$

Kullback-Leibler Divergence

Let $p(\cdot)$ and $q(\cdot)$ are two p.m.f.'s of a random variable X . The relative entropy between p and q is $D(p||q) := \mathbb{E}_p \left[\log \frac{p(X)}{q(X)} \right]$
(the subscript "p" denotes that the expectation is taken over the distribution p .)

Please note that KL divergence is NOT symmetric: $D(p||q) \neq D(q||p)$.

Important Results:

$D(p||q) \geq 0$, with equality iff $p(x) = q(x)$ for all $x \in \mathcal{X}$.

Maximum Likelihood Estimator (MLE)

- Let $p_*(\cdot)$ be the true data generating distribution
- $E_*(\cdot)$ be expectation w.r.t. $p_*(\cdot)$
- Suppose that iid samples (observations)
 y_1, y_2, \dots, y_n are given.
- Let $p \equiv p_\theta$ for $\theta \in \Theta$ denote our guesses for $p_*(\cdot)$
- MLE: Choose the value of $\theta \in \Theta$ that achieves the maximum of

$$\frac{\sum_1^n \log p(y_i)}{n}$$

Maximum Likelihood Estimator (MLE)

- Why is it so popular?
 - It is an elementary function of the probability density function
 - Intimate relation with KL-divergence
 - Notice that

$$-\frac{\sum_1^n \log p(y_i)}{n} \rightarrow E_{p^*} [-\log p(y)]$$

- Minimizing

$$-\frac{\sum_1^n \log p(y_i)}{n}$$

is asymptotically equivalent to minimizing

$$E_* \{-\log p(y)\} = D_{KL}(p_*||p) + H(p_*)$$

or equivalently

$$D_{KL}(p_*||p).$$

Bernoulli Distribution

- Consider a single binary random variable $x \in \{0, 1\}$.
- For example, x can describe the outcome of flipping a coin:
Coin flipping: heads = 1, tails = 0.
- The probability of $x=1$ will be denoted by the parameter ¹, so that:

$$p(x=1|\mu) = \mu \quad 0 \leq \mu \leq 1.$$
- The probability distribution, known as Bernoulli distribution, can be written as:

$$\begin{aligned} \text{Bern}(x|\mu) &= \mu^x (1-\mu)^{1-x} \\ \mathbb{E}[x] &= \mu \\ \text{var}[x] &= \mu(1-\mu) \end{aligned}$$

Parameter Estimation

- Suppose we observed a data $\mathcal{D} = \{x_1, \dots, x_N\}$
- We can construct the likelihood function, which is a function of μ .

$$p(\mathcal{D}|\mu) = \prod_{n=1}^N p(x_n|\mu) = \prod_{n=1}^N \mu^{x_n} (1-\mu)^{1-x_n}$$

- Equivalently, we can maximize the log of the likelihood function:

$$\ln p(\mathcal{D}|\mu) = \sum_{n=1}^N \ln p(x_n|\mu) = \sum_{n=1}^N \{x_n \ln \mu + (1-x_n) \ln(1-\mu)\}$$

- Note that the likelihood function depends on the N observations x_n only through the sum

$\sum_n x_n$ ← Sufficient Statistic

Parameter Estimation

- Suppose we observed a data $\mathcal{D} = \{x_1, \dots, x_N\}$

$$\ln p(\mathcal{D}|\mu) = \sum_{n=1}^N \ln p(x_n|\mu) = \sum_{n=1}^N \{x_n \ln \mu + (1-x_n) \ln(1-\mu)\}$$

- Setting the derivative of the log-likelihood function w.r.t μ to zero, we obtain:

$$\mu_{\text{ML}} = \frac{1}{N} \sum_{n=1}^N x_n = \frac{m}{N}$$

where m is the number of heads.

Binomial Distribution

- We can also work out the distribution of the number m of observations of $x=1$ (e.g. the number of heads).
- The probability of observing m heads given N coin flips and a parameter μ is given by:

$$p(m \text{ heads}|N, \mu) = \text{Bin}(m|N, \mu) = \binom{N}{m} \mu^m (1-\mu)^{N-m}$$

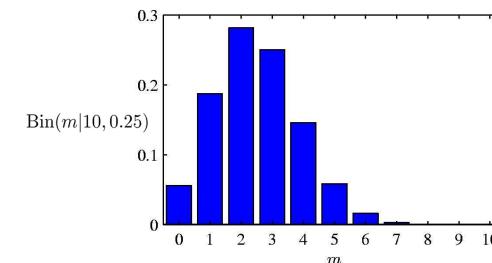
- The mean and variance can be easily derived as:

$$\mathbb{E}[m] \equiv \sum_{m=0}^N m \text{Bin}(m|N, \mu) = N\mu$$

$$\text{var}[m] \equiv \sum_{m=0}^N (m - \mathbb{E}[m])^2 \text{Bin}(m|N, \mu) = N\mu(1-\mu)$$

Example

- Histogram plot of the Binomial distribution as a function of m for $N=10$ and $\mu = 0.25$.



Multinomial Variables

- Consider a random variable that can take on one of K possible mutually exclusive states (e.g. roll of a dice).
- We will use so-called 1-of-K encoding scheme.
- If a random variable can take on K=6 states, and a particular observation of the variable corresponds to the state $x_3=1$, then \mathbf{x} will be represented as:

$$\text{1-of-K coding scheme: } \mathbf{x} = (0, 0, 1, 0, 0, 0)^T$$

- If we denote the probability of $x_k=1$ by the parameter μ_k , then the distribution over \mathbf{x} is defined as:

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{k=1}^K \mu_k^{x_k} \quad \forall k : \mu_k \geq 0 \quad \text{and} \quad \sum_{k=1}^K \mu_k = 1$$

Multinomial Variables

- Multinomial distribution can be viewed as a generalization of Bernoulli distribution to more than two outcomes.

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{k=1}^K \mu_k^{x_k}$$

- It is easy to see that the distribution is normalized:

$$\sum_{\mathbf{x}} p(\mathbf{x}|\boldsymbol{\mu}) = \sum_{k=1}^K \mu_k = 1$$

and

$$\mathbb{E}[\mathbf{x}|\boldsymbol{\mu}] = \sum_{\mathbf{x}} p(\mathbf{x}|\boldsymbol{\mu}) \mathbf{x} = (\mu_1, \dots, \mu_K)^T = \boldsymbol{\mu}$$

Maximum Likelihood Estimation

- Suppose we observed a data $\mathcal{D} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$
- We can construct the likelihood function, which is a function of $\boldsymbol{\mu}$.

$$p(\mathcal{D}|\boldsymbol{\mu}) = \prod_{n=1}^N \prod_{k=1}^K \mu_k^{x_{nk}} = \prod_{k=1}^K \mu_k^{(\sum_n x_{nk})} = \prod_{k=1}^K \mu_k^{m_k}$$

- Note that the likelihood function depends on the N data points only through the following K quantities:

$$m_k = \sum_n x_{nk}, \quad k = 1, \dots, K.$$

which represents the number of observations for $x_k=1$.

- These are called the sufficient statistics for this distribution.

Maximum Likelihood Estimation

$$p(\mathcal{D}|\boldsymbol{\mu}) = \prod_{n=1}^N \prod_{k=1}^K \mu_k^{x_{nk}} = \prod_{k=1}^K \mu_k^{(\sum_n x_{nk})} = \prod_{k=1}^K \mu_k^{m_k}$$

- To find a maximum likelihood solution for $\boldsymbol{\mu}$, we need to maximize the log-likelihood taking into account the constraint that $\sum_k \mu_k = 1$

- Forming the Lagrangian:

$$\sum_{k=1}^K m_k \ln \mu_k + \lambda \left(\sum_{k=1}^K \mu_k - 1 \right)$$

$$\mu_k = -m_k/\lambda \quad \mu_k^{\text{ML}} = \frac{m_k}{N} \quad \lambda = -N$$

which is the fraction of observations for which $x_k=1$.

Multinomial Distribution

- We can construct the joint distribution of the quantities $\{m_1, m_2, \dots, m_k\}$ given the parameters $\boldsymbol{\mu}$ and the total number N of observations:

$$\text{Mult}(m_1, m_2, \dots, m_K | \boldsymbol{\mu}, N) = \binom{N}{m_1 m_2 \dots m_K} \prod_{k=1}^K \mu_k^{m_k}$$

$$\mathbb{E}[m_k] = N\mu_k$$

$$\text{var}[m_k] = N\mu_k(1 - \mu_k)$$

$$\text{cov}[m_j m_k] = -N\mu_j\mu_k$$

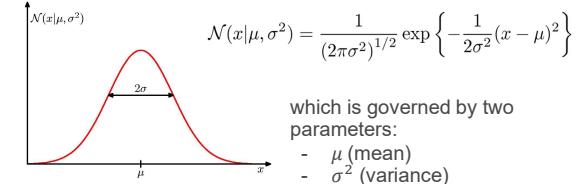
- The normalization coefficient is the number of ways of partitioning N objects into K groups of size m_1, m_2, \dots, m_K .

- Note that

$$\sum_k m_k = N.$$

Gaussian Univariate Distribution

- In the case of a single variable x , the Gaussian distribution takes form:



- The Gaussian distribution satisfies:

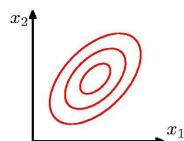
$$\mathcal{N}(x|\mu, \sigma^2) > 0$$

$$\int_{-\infty}^{\infty} \mathcal{N}(x|\mu, \sigma^2) dx = 1$$

Multivariate Gaussian Distribution

- For a D -dimensional vector \mathbf{x} , the Gaussian distribution takes form:

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right\}$$



which is governed by two parameters:

- $\boldsymbol{\mu}$ is a D -dimensional mean vector.
- $\boldsymbol{\Sigma}$ is a D by D covariance matrix. and $|\boldsymbol{\Sigma}|$ denotes the determinant of $\boldsymbol{\Sigma}$.

- Note that the covariance matrix is a symmetric positive definite matrix.

Central Limit Theorem

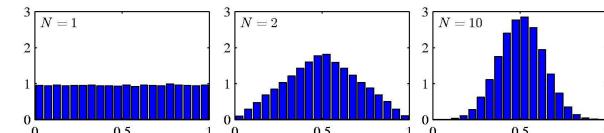
- The distribution of the sum of N i.i.d. random variables becomes increasingly Gaussian as N grows.

- Consider N variables, each of which has a uniform distribution over the interval $[0, 1]$.

- Let us look at the distribution over the mean:

$$\frac{x_1 + x_2 + \dots + x_N}{N}$$

- As N increases, the distribution tends towards a Gaussian distribution.



Moments of the Gaussian Distribution

- The expectation of \mathbf{x} under the Gaussian distribution:

$$\begin{aligned}\mathbb{E}[\mathbf{x}] &= \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma|^{1/2}} \int \exp\left\{-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})\right\} \mathbf{x} d\mathbf{x} \\ &= \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma|^{1/2}} \underbrace{\int \exp\left\{-\frac{1}{2}\mathbf{z}^T \Sigma^{-1} \mathbf{z}\right\} (\mathbf{z} + \boldsymbol{\mu}) d\mathbf{z}}_{\text{The term in } z \text{ in the factor } (z+1) \text{ will vanish by symmetry.}}\end{aligned}$$

$$\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu}$$

Moments of the Gaussian Distribution

- The second order moments of the Gaussian distribution:

$$\mathbb{E}[\mathbf{x}\mathbf{x}^T] = \boldsymbol{\mu}\boldsymbol{\mu}^T + \Sigma$$

- The covariance is given by:

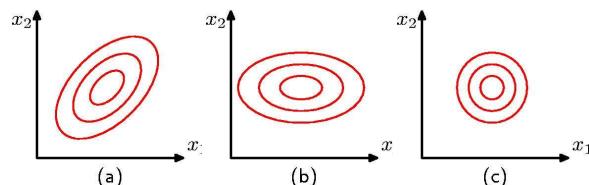
$$\text{cov}[\mathbf{x}] = \mathbb{E}[(\mathbf{x} - \mathbb{E}[\mathbf{x}])(\mathbf{x} - \mathbb{E}[\mathbf{x}])^T] = \Sigma$$

$$\begin{array}{c} \xrightarrow{\quad} \\ \mathbb{E}[\mathbf{x}] = \boldsymbol{\mu} \end{array}$$

- Because the parameter matrix $\boldsymbol{\Sigma}$ governs the covariance of \mathbf{x} under the Gaussian distribution, it is called the covariance matrix.

Moments of the Gaussian Distribution

- Contours of constant probability density:



Covariance matrix is of general form.

Diagonal, axis-aligned covariance matrix.

Spherical (proportional to identity) covariance matrix.

Partitioned Gaussian Distribution

- Consider a D-dimensional Gaussian distribution: $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$

- Let us partition \mathbf{x} into two disjoint subsets \mathbf{x}_a and \mathbf{x}_b :

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{pmatrix} \quad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{pmatrix} \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{pmatrix}$$

- In many situations, it will be more convenient to work with the precision matrix (inverse of the covariance matrix):

$$\boldsymbol{\Lambda} \equiv \boldsymbol{\Sigma}^{-1} \quad \boldsymbol{\Lambda} = \begin{pmatrix} \boldsymbol{\Lambda}_{aa} & \boldsymbol{\Lambda}_{ab} \\ \boldsymbol{\Lambda}_{ba} & \boldsymbol{\Lambda}_{bb} \end{pmatrix}$$

- Note that $\boldsymbol{\Lambda}_{aa}$ is not given by the inverse of $\boldsymbol{\Sigma}_{aa}$.

Marginal Distribution

- It turns out that the marginal distribution is also a Gaussian distribution:

$$\begin{aligned} p(\mathbf{x}_a) &= \int p(\mathbf{x}_a, \mathbf{x}_b) d\mathbf{x}_b \\ &= \mathcal{N}(\mathbf{x}_a | \boldsymbol{\mu}_a, \boldsymbol{\Sigma}_{aa}) \end{aligned}$$

- For a marginal distribution, the mean and covariance are most simply expressed in terms of partitioned covariance matrix.

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{pmatrix} \quad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{pmatrix} \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{pmatrix}$$

Conditional Distribution

- It turns out that the conditional distribution is also a Gaussian distribution:

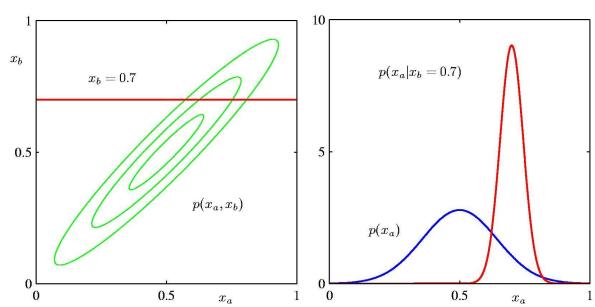
$$p(\mathbf{x}_a | \mathbf{x}_b) = \mathcal{N}(\mathbf{x}_a | \boldsymbol{\mu}_{a|b}, \boldsymbol{\Sigma}_{a|b})$$

Covariance does
not depend on \mathbf{x}_b .

$$\begin{aligned} \boldsymbol{\Sigma}_{a|b} &= \boldsymbol{\Lambda}_{aa}^{-1} = \boldsymbol{\Sigma}_{aa} - \boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}\boldsymbol{\Sigma}_{ba} \\ \boldsymbol{\mu}_{a|b} &= \boldsymbol{\Sigma}_{a|b} \{ \boldsymbol{\Lambda}_{aa}\boldsymbol{\mu}_a - \boldsymbol{\Lambda}_{ab}(\mathbf{x}_b - \boldsymbol{\mu}_b) \} \\ &= \boldsymbol{\mu}_a - \boldsymbol{\Lambda}_{aa}^{-1}\boldsymbol{\Lambda}_{ab}(\mathbf{x}_b - \boldsymbol{\mu}_b) \\ &= \boldsymbol{\mu}_a + \boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}(\mathbf{x}_b - \boldsymbol{\mu}_b) \end{aligned}$$

Linear function
of \mathbf{x}_b .

Conditional and Marginal Distributions



Maximum Likelihood Estimation

- To find a maximum likelihood estimate of the mean, we set the derivative of the log-likelihood function to zero:

$$\frac{\partial}{\partial \boldsymbol{\mu}} \ln p(\mathbf{X} | \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^N \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}) = 0$$

and solve to obtain:

$$\boldsymbol{\mu}_{ML} = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n.$$

- Similarly, we can find the ML estimate of $\boldsymbol{\Sigma}$:

$$\boldsymbol{\Sigma}_{ML} = \frac{1}{N} \sum_{n=1}^N (\mathbf{x}_n - \boldsymbol{\mu}_{ML})(\mathbf{x}_n - \boldsymbol{\mu}_{ML})^T.$$

Maximum Likelihood Estimation

- Evaluating the expectation of the ML estimates under the true distribution, we obtain:

$$\begin{aligned}\mathbb{E}[\mu_{\text{ML}}] &= \mu && \text{Unbiased estimate} \\ \mathbb{E}[\Sigma_{\text{ML}}] &= \frac{N-1}{N}\Sigma. && \text{Biased estimate}\end{aligned}$$

- Note that the maximum likelihood estimate of Σ is biased.

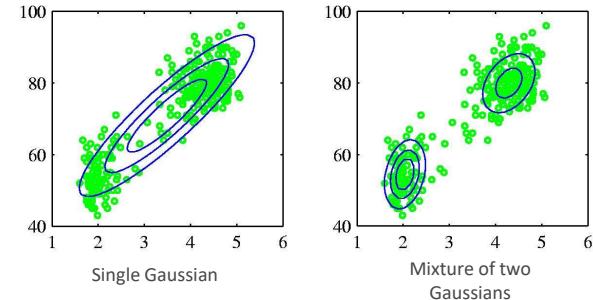
- We can correct the bias by defining a different estimator:

$$\tilde{\Sigma} = \frac{1}{N-1} \sum_{n=1}^N (\mathbf{x}_n - \mu_{\text{ML}})(\mathbf{x}_n - \mu_{\text{ML}})^T.$$

Mixture of Gaussians

- When modeling real-world data, Gaussian assumption may not be appropriate.

- Consider the following example: Old Faithful Dataset



Mixture of Gaussians

- We can combine simple models into a complex model by defining a superposition of K Gaussian densities of the form:

$$p(\mathbf{x}) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)$$

Component

Mixing coefficient

$$\forall k : \pi_k \geq 0 \quad \sum_{k=1}^K \pi_k = 1$$

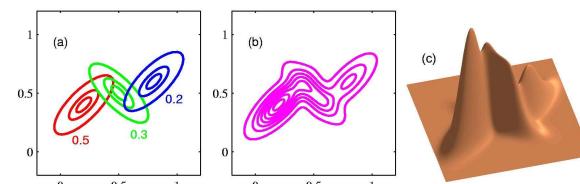
$K=3$

A graph showing three red Gaussian curves representing components, and a blue curve representing the total mixture probability density $p(x)$ as a sum of these components.

- Note that each Gaussian component has its own mean μ_k and covariance Σ_k . The parameters π_k are called mixing coefficients.
- More generally, mixture models can comprise linear combinations of other distributions.

Mixture of Gaussians

- Illustration of a mixture of 3 Gaussians in a 2-dimensional space:



(a) Contours of constant density of each of the mixture components, along with the mixing coefficients

(b) Contours of marginal probability density $p(\mathbf{x}) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)$

(c) A surface plot of the distribution $p(\mathbf{x})$.

Maximum Likelihood Estimation

- Given a dataset D, we can determine model parameters by maximizing the log-likelihood function:

$$\ln p(\mathbf{X}|\pi, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^N \ln \left\{ \underbrace{\sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}_{\text{Log of a sum: no closed form solution}} \right\}$$

- Solution:** use standard, iterative, numeric optimization methods or the Expectation Maximization algorithm.

The Exponential Distribution

The family of exponential distribution provides probability models that are very widely used.

Definition

X is said to have an **exponential distribution** with parameter $\lambda > 0$ if the pdf of X is

$$f(x; \lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The Gamma Distribution

Definition

A continuous random variable X is said to have a **gamma distribution** if the pdf of X is

$$f(x; \alpha, \beta) = \begin{cases} \frac{1}{\beta^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-x/\beta} & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where the parameters α and β satisfy $\alpha > 0, \beta > 0$. The **standard gamma distribution** has $\beta = 1$.

We may be lazy and write $\text{Gam}(x | \alpha, \beta)$ for the above pdf instead.

Generation of Dirichlet from Gamma

- Take $Y_1 \sim \text{Gam}(x | \alpha_1, \beta), Y_2 \sim \text{Gam}(x | \alpha_2, \beta), \dots, Y_n \sim \text{Gam}(x | \alpha_n, \beta)$,

Let $V = \sum_1^n Y_i$.

Let

$$X_i = \frac{Y_i}{V}$$

- Then (X_1, \dots, X_n) are distributed according to Dirichlet with parameters $\alpha_1, \dots, \alpha_n$.

Student's t-Distribution

- Consider Student's t-Distribution

$$\begin{aligned}
 p(x|\mu, a, b) &= \int_0^\infty \mathcal{N}(x|\mu, \tau^{-1}) \text{Gam}(\tau|a, b) d\tau \\
 &= \int_0^\infty \mathcal{N}(x|\mu, (\eta\lambda)^{-1}) \text{Gam}(\eta|\nu/2, \nu/2) d\eta \\
 &= \frac{\Gamma(\nu/2 + 1/2)}{\Gamma(\nu/2)} \left(\frac{\lambda}{\pi\nu} \right)^{1/2} \left[1 + \frac{\lambda(x - \mu)^2}{\nu} \right]^{-\nu/2 - 1/2} \\
 &= \text{St}(x|\mu, \lambda, \nu)
 \end{aligned}$$

Infinite mixture
of Gaussians

where

$$\lambda = a/b \quad \eta = \tau b/a \quad \nu = 2a.$$

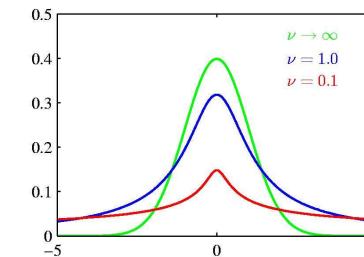
Sometimes called
the precision
parameter.

Degrees of
freedom

Student's t-Distribution

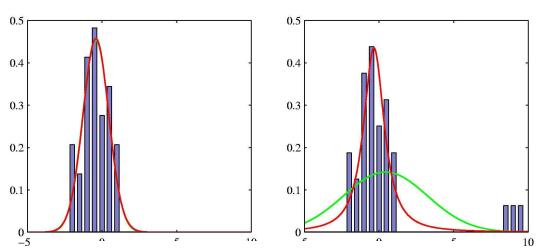
- Setting $\nu = 1$ recovers Cauchy distribution
- The limit $\nu \rightarrow \infty$ corresponds to a Gaussian distribution.

	$\nu = 1$	$\nu \rightarrow \infty$
$\text{St}(x \mu, \lambda, \nu)$	Cauchy	$\mathcal{N}(x \mu, \lambda^{-1})$



Student's t-Distribution

- Robustness to outliers: Gaussian vs. t-Distribution.



Student's t-Distribution

- The multivariate extension of the t-Distribution of dimension D :

$$\begin{aligned}
 \text{St}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}, \nu) &= \int_0^\infty \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, (\eta\boldsymbol{\Lambda})^{-1}) \text{Gam}(\eta|\nu/2, \nu/2) d\eta \\
 &= \frac{\Gamma(D/2 + \nu/2)}{\Gamma(\nu/2)} |\boldsymbol{\Lambda}|^{1/2} \left[1 + \frac{\Delta^2}{\nu} \right]^{-D/2 - \nu/2}
 \end{aligned}$$

where $\Delta^2 = (\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Lambda} (\mathbf{x} - \boldsymbol{\mu})$

- Properties:

$$\begin{aligned}
 \mathbb{E}[\mathbf{x}] &= \boldsymbol{\mu}, & \text{if } \nu > 1 \\
 \text{cov}[\mathbf{x}] &= \frac{\nu}{(\nu - 2)} \boldsymbol{\Lambda}^{-1}, & \text{if } \nu > 2 \\
 \text{mode}[\mathbf{x}] &= \boldsymbol{\mu}
 \end{aligned}$$

The Exponential Family

- The exponential family of distributions over \mathbf{x} is defined to be a set of distributions of the form:

$$p(\mathbf{x}|\boldsymbol{\eta}) = h(\mathbf{x})g(\boldsymbol{\eta}) \exp\{\boldsymbol{\eta}^T \mathbf{u}(\mathbf{x})\}$$

where

- $\boldsymbol{\eta}$ is the vector of natural parameters
- $\mathbf{u}(\mathbf{x})$ is the vector of sufficient statistics

- The function $g(\boldsymbol{\eta})$ can be interpreted as the coefficient that ensures that the distribution $p(\mathbf{x}|\boldsymbol{\eta})$ is normalized:

$$g(\boldsymbol{\eta}) \int h(\mathbf{x}) \exp\{\boldsymbol{\eta}^T \mathbf{u}(\mathbf{x})\} d\mathbf{x} = 1$$

Bernoulli Distribution

- The Bernoulli distribution is a member of the exponential family:

$$\begin{aligned} p(x|\mu) &= \text{Bern}(x|\mu) = \mu^x(1-\mu)^{1-x} \\ &= \exp\{x \ln \mu + (1-x) \ln(1-\mu)\} \\ &= (1-\mu) \exp\left\{\ln\left(\frac{\mu}{1-\mu}\right)x\right\} \end{aligned}$$

- Comparing with the general form of the exponential family:

$$p(\mathbf{x}|\boldsymbol{\eta}) = h(\mathbf{x})g(\boldsymbol{\eta}) \exp\{\boldsymbol{\eta}^T \mathbf{u}(\mathbf{x})\}$$

we see that

$$\boldsymbol{\eta} = \ln\left(\frac{\mu}{1-\mu}\right) \quad \text{and so} \quad \mu = \sigma(\boldsymbol{\eta}) = \underbrace{\frac{1}{1 + \exp(-\boldsymbol{\eta})}}_{\text{Logistic sigmoid}}.$$

Bernoulli Distribution

- The Bernoulli distribution is a member of the exponential family:

$$\begin{aligned} p(x|\mu) &= \text{Bern}(x|\mu) = \mu^x(1-\mu)^{1-x} \\ &= \exp\{x \ln \mu + (1-x) \ln(1-\mu)\} \\ &= (1-\mu) \exp\left\{\ln\left(\frac{\mu}{1-\mu}\right)x\right\} \end{aligned}$$

$$p(\mathbf{x}|\boldsymbol{\eta}) = h(\mathbf{x})g(\boldsymbol{\eta}) \exp\{\boldsymbol{\eta}^T \mathbf{u}(\mathbf{x})\}$$

- The Bernoulli distribution can therefore be written as:

$$p(x|\boldsymbol{\eta}) = \sigma(-\boldsymbol{\eta}) \exp(\boldsymbol{\eta}x)$$

where

$$\begin{aligned} u(x) &= x \\ h(x) &= 1 \\ g(\boldsymbol{\eta}) &= 1 - \sigma(\boldsymbol{\eta}) = \sigma(-\boldsymbol{\eta}). \end{aligned}$$

Multinomial Distribution

- The Multinomial distribution is a member of the exponential family:

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{k=1}^M \mu_k^{x_k} = \exp\left\{\sum_{k=1}^M x_k \ln \mu_k\right\} = h(\mathbf{x})g(\boldsymbol{\mu}) \exp\{\boldsymbol{\mu}^T \mathbf{u}(\mathbf{x})\}$$

where $\mathbf{x} = (x_1, \dots, x_M)^T$ $\boldsymbol{\mu} = (\mu_1, \dots, \mu_M)^T$

and

$$\begin{aligned} \boldsymbol{\mu}_k &= \ln \mu_k \\ \mathbf{u}(\mathbf{x}) &= \mathbf{x} \\ h(\mathbf{x}) &= 1 \\ g(\boldsymbol{\mu}) &= 1. \end{aligned}$$

NOTE: The parameters $\boldsymbol{\mu}_k$ are not independent since the corresponding $\boldsymbol{\mu}_k$ must satisfy $\sum_{k=1}^M \mu_k = 1$.

- In some cases it will be convenient to remove the constraint by expressing the distribution over the M-1 parameters.

Multinomial Distribution

- The Multinomial distribution is a member of the exponential family:

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{k=1}^M \mu_k^{x_k} = \exp \left\{ \sum_{k=1}^M x_k \ln \mu_k \right\} = h(\mathbf{x})g(\boldsymbol{\eta}) \exp (\boldsymbol{\eta}^T \mathbf{u}(\mathbf{x}))$$

- Let $\mu_M = 1 - \sum_{k=1}^{M-1} \mu_k$

- This leads to:

$$\eta_k = \ln \left(\frac{\mu_k}{1 - \sum_{j=1}^{M-1} \mu_j} \right) \quad \text{and} \quad \mu_k = \underbrace{\frac{\exp(\eta_k)}{1 + \sum_{j=1}^{M-1} \exp(\eta_j)}}_{\text{Softmax function}}.$$

- Here the parameters η_k are independent.

- Note that:

$$0 \leq \mu_k \leq 1 \quad \text{and} \quad \sum_{k=1}^{M-1} \mu_k \leq 1.$$

Multinomial Distribution

- The Multinomial distribution is a member of the exponential family:

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{k=1}^M \mu_k^{x_k} = \exp \left\{ \sum_{k=1}^M x_k \ln \mu_k \right\} = h(\mathbf{x})g(\boldsymbol{\eta}) \exp (\boldsymbol{\eta}^T \mathbf{u}(\mathbf{x}))$$

- The Multinomial distribution can therefore be written as:

$$p(\mathbf{x}|\boldsymbol{\mu}) = h(\mathbf{x})g(\boldsymbol{\eta}) \exp (\boldsymbol{\eta}^T \mathbf{u}(\mathbf{x}))$$

where

$$\boldsymbol{\eta} = (\eta_1, \dots, \eta_{M-1}, 0)^T$$

$$\mathbf{u}(\mathbf{x}) = \mathbf{x}$$

$$h(\mathbf{x}) = 1$$

$$g(\boldsymbol{\eta}) = \left(1 + \sum_{k=1}^{M-1} \exp(\eta_k) \right)^{-1}.$$

Gaussian Distribution

- The Gaussian distribution can be written as:

$$\begin{aligned} p(x|\mu, \sigma^2) &= \frac{1}{(2\pi\sigma^2)^{1/2}} \exp \left\{ -\frac{1}{2\sigma^2}(x-\mu)^2 \right\} \\ &= \frac{1}{(2\pi\sigma^2)^{1/2}} \exp \left\{ -\frac{1}{2\sigma^2}x^2 + \frac{\mu}{\sigma^2}x - \frac{1}{2\sigma^2}\mu^2 \right\} \\ &= h(x)g(\boldsymbol{\eta}) \exp \{ \boldsymbol{\eta}^T \mathbf{u}(x) \} \end{aligned}$$

where

$$\boldsymbol{\eta} = \begin{pmatrix} \mu/\sigma^2 \\ -1/2\sigma^2 \end{pmatrix} \quad h(\mathbf{x}) = (2\pi)^{-1/2}$$

$$\mathbf{u}(x) = \begin{pmatrix} x \\ x^2 \end{pmatrix} \quad g(\boldsymbol{\eta}) = (-2\eta_2)^{1/2} \exp \left(\frac{\eta_1^2}{4\eta_2} \right).$$

ML for the Exponential Family

- Recall the Exponential Family:

$$p(\mathbf{x}|\boldsymbol{\eta}) = h(\mathbf{x})g(\boldsymbol{\eta}) \exp \{ \boldsymbol{\eta}^T \mathbf{u}(\mathbf{x}) \}$$

- From the definition of the normalizer $g(\cdot)$:

$$g(\boldsymbol{\eta}) \int h(\mathbf{x}) \exp \{ \boldsymbol{\eta}^T \mathbf{u}(\mathbf{x}) \} d\mathbf{x} = 1$$

- We can take a derivative w.r.t \cdot :

$$\underbrace{\nabla g(\boldsymbol{\eta}) \int h(\mathbf{x}) \exp \{ \boldsymbol{\eta}^T \mathbf{u}(\mathbf{x}) \} d\mathbf{x}}_{1/g(\boldsymbol{\eta})} + \underbrace{g(\boldsymbol{\eta}) \int h(\mathbf{x}) \exp \{ \boldsymbol{\eta}^T \mathbf{u}(\mathbf{x}) \} \mathbf{u}(\mathbf{x}) d\mathbf{x}}_{\mathbb{E}[\mathbf{u}(\mathbf{x})]} = 0$$

- Thus

$$-\nabla \ln g(\boldsymbol{\eta}) = \mathbb{E}[\mathbf{u}(\mathbf{x})]$$

ML for the Exponential Family

- Recall the Exponential Family:

$$p(\mathbf{x}|\boldsymbol{\eta}) = h(\mathbf{x})g(\boldsymbol{\eta}) \exp\{\boldsymbol{\eta}^T \mathbf{u}(\mathbf{x})\}$$

- We can take a derivative w.r.t $\boldsymbol{\eta}$:

$$\nabla g(\boldsymbol{\eta}) \underbrace{\int h(\mathbf{x}) \exp\{\boldsymbol{\eta}^T \mathbf{u}(\mathbf{x})\} d\mathbf{x}}_{1/g(\boldsymbol{\eta})} + g(\boldsymbol{\eta}) \underbrace{\int h(\mathbf{x}) \exp\{\boldsymbol{\eta}^T \mathbf{u}(\mathbf{x})\} \mathbf{u}(\mathbf{x}) d\mathbf{x}}_{\mathbb{E}[\mathbf{u}(\mathbf{x})]} = 0$$

- Thus

$$-\nabla \ln g(\boldsymbol{\eta}) = \mathbb{E}[\mathbf{u}(\mathbf{x})]$$

- Note that the covariance of $\mathbf{u}(\mathbf{x})$ can be expressed in terms of the second derivative of $g(\cdot)$, and similarly for the higher moments.

ML for the Exponential Family

- Suppose we observed i.i.d $d\mathbf{x} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$.

- We can construct the log-likelihood function, which is a function of the natural parameter $\boldsymbol{\eta}$.

$$p(\mathbf{x}|\boldsymbol{\eta}) = h(\mathbf{x})g(\boldsymbol{\eta}) \exp\{\boldsymbol{\eta}^T \mathbf{u}(\mathbf{x})\}$$

$$p(\mathbf{X}|\boldsymbol{\eta}) = \left(\prod_{n=1}^N h(\mathbf{x}_n) \right) g(\boldsymbol{\eta})^N \exp\left\{ \boldsymbol{\eta}^T \sum_{n=1}^N \mathbf{u}(\mathbf{x}_n) \right\}.$$

- Therefore we have

$$-\nabla \ln g(\boldsymbol{\eta}_{\text{ML}}) = \frac{1}{N} \sum_{n=1}^N \mathbf{u}(\mathbf{x}_n)$$

Sufficient Statistic

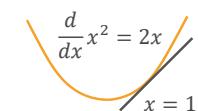
Review of Multivariable Calculus

Review Scalar Derivative

y	a	x^n	$\exp(x)$	$\log(x)$	$\sin(x)$
$\frac{dy}{dx}$	0	nx^{n-1}	$\exp(x)$	$\frac{1}{x}$	$\cos(x)$

a is not a function of x

Derivative is the slope of the tangent line

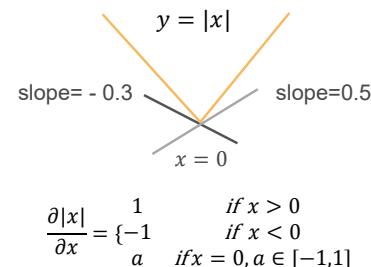


The slope of the tangent line is 2

y	$u + v$	uv	$y = f(u), u = g(x)$
$\frac{dy}{dx}$	$\frac{du}{dx} + \frac{dv}{dx}$	$\frac{du}{dx}v + \frac{dv}{dx}u$	$\frac{dy}{du} \frac{du}{dx}$

Subderivative

Extend derivative to non-differentiable cases



Another example:

$$\frac{\partial}{\partial x} \max(x, 0) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \\ a & \text{if } x = 0, a \in [0, 1] \end{cases}$$

Gradients

Generalize derivatives into vectors

		Vector
Scalar	x	\mathbf{x}
Scalar	y	$\frac{\partial y}{\partial x}$
Vector	\mathbf{y}	$\frac{\partial \mathbf{y}}{\partial x}$

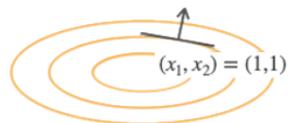
$\frac{\partial y}{\partial \mathbf{x}}$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \frac{\partial y}{\partial \mathbf{x}} = \left[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \dots, \frac{\partial y}{\partial x_n} \right]$$

y	x
$\frac{\partial y}{\partial x}$	$\frac{\partial y}{\partial \mathbf{x}}$
\mathbf{y}	$\frac{\partial \mathbf{y}}{\partial x}$

$$\frac{\partial}{\partial \mathbf{x}} x_1^2 + 2x_2^2 = [2x_1, 4x_2]$$

Direction $(2, 4)$, perpendicular to the contour lines



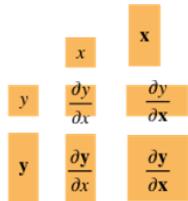
Examples

$$\begin{array}{c|cccc} y & a & au & \text{sum}(\mathbf{x}) & \|\mathbf{x}\|^2 \\ \hline \frac{\partial y}{\partial \mathbf{x}} & \mathbf{0}^T & a \frac{\partial u}{\partial \mathbf{x}} & \mathbf{1}^T & 2\mathbf{x}^T \end{array} \quad \begin{array}{l} a \text{ is not a function of } \mathbf{x} \\ \mathbf{0} \text{ and } \mathbf{1} \text{ are vectors} \end{array}$$

$$\begin{array}{c|ccc} y & u + v & uv & \langle \mathbf{u}, \mathbf{v} \rangle \\ \hline \frac{\partial y}{\partial \mathbf{x}} & \frac{\partial u}{\partial \mathbf{x}} + \frac{\partial v}{\partial \mathbf{x}} & \frac{\partial u}{\partial \mathbf{x}} v + \frac{\partial v}{\partial \mathbf{x}} u & \mathbf{u}^T \frac{\partial \mathbf{v}}{\partial \mathbf{x}} + \mathbf{v}^T \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \end{array}$$

$\partial \mathbf{y} / \partial x$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \quad \frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} \\ \frac{\partial y_2}{\partial x} \\ \vdots \\ \frac{\partial y_m}{\partial x} \end{bmatrix}$$

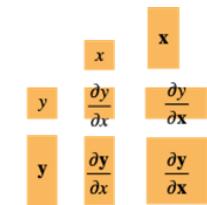


$\partial \mathbf{y} / \partial \mathbf{x}$ is a row vector, while $\frac{\partial \mathbf{y}}{\partial x}$ is a column vector

It is called numerator-layout notation. The reversed version is called denominator-layout notation

 $\partial \mathbf{y} / \partial \mathbf{x}$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$



$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial \mathbf{x}} \\ \frac{\partial y_2}{\partial \mathbf{x}} \\ \vdots \\ \frac{\partial y_m}{\partial \mathbf{x}} \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1}, \frac{\partial y_1}{\partial x_2}, \dots, \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1}, \frac{\partial y_2}{\partial x_2}, \dots, \frac{\partial y_2}{\partial x_n} \\ \vdots \\ \frac{\partial y_m}{\partial x_1}, \frac{\partial y_m}{\partial x_2}, \dots, \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Examples

$$\mathbf{y} \mid \mathbf{a} \quad \mathbf{x} \quad \mathbf{A}\mathbf{x} \quad \mathbf{x}^T \mathbf{A}$$

$$\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m, \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times n}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \mid \mathbf{0} \quad \mathbf{I} \quad \mathbf{A} \quad \mathbf{A}^T$$

\mathbf{a}, \mathbf{a} and \mathbf{A} are not functions of \mathbf{x}
 $\mathbf{0}$ and \mathbf{I} are matrices

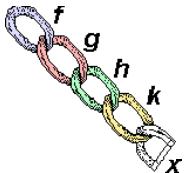
$$\mathbf{y} \mid a\mathbf{u} \quad \mathbf{A}\mathbf{u} \quad \mathbf{u} + \mathbf{v}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \mid a \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \quad \mathbf{A} \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \quad \frac{\partial \mathbf{u}}{\partial \mathbf{x}} + \frac{\partial \mathbf{v}}{\partial \mathbf{x}}$$

Generalize to Matrices

	Scalar	Vector	Matrix
Scalar	x (1,)	\mathbf{x} (n, 1)	\mathbf{X} (n, k)
Vector	y (1,)	$\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ (1, n)	$\frac{\partial \mathbf{y}}{\partial \mathbf{X}}$ (k, n)
Matrix	\mathbf{Y} (m, l)	$\frac{\partial \mathbf{Y}}{\partial \mathbf{x}}$ (m, l)	$\frac{\partial \mathbf{Y}}{\partial \mathbf{X}}$ (m, l, n)

Chain Rule



Generalize to Vectors

Chain rule for scalars:

$$y = f(u), u = g(x) \quad \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x}$$

Generalize to vectors straightforwardly

$$\begin{aligned} \frac{\partial y}{\partial \mathbf{x}} &= \frac{\partial y}{\partial u} \frac{\partial u}{\partial \mathbf{x}} & \frac{\partial y}{\partial \mathbf{x}} &= \frac{\partial y}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{x}} & \frac{\partial \mathbf{y}}{\partial \mathbf{x}} &= \frac{\partial \mathbf{y}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \\ (1, n) \quad (1,) &(1, n) & (1, n) \quad (1, k) &(k, n) & (m, n) \quad (m, k) &(k, n) \end{aligned}$$



Example 1

Assume $\mathbf{x}, \mathbf{w} \in \mathbb{R}^n, \quad y \in \mathbb{R}$

$$z = (\langle \mathbf{x}, \mathbf{w} \rangle - y)^2$$

Compute $\frac{\partial z}{\partial \mathbf{w}}$

$$a = \langle \mathbf{x}, \mathbf{w} \rangle$$

$$b = a - y$$

$$z = b^2$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{x}}$$

$$\begin{aligned} \frac{\partial z}{\partial \mathbf{w}} &= \frac{\partial z}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial \mathbf{w}} \\ &= \frac{\partial b^2}{\partial b} \frac{\partial a - y}{\partial a} \frac{\partial \langle \mathbf{x}, \mathbf{w} \rangle}{\partial \mathbf{w}} \\ &= 2b \cdot 1 \cdot \mathbf{x}^T \\ &= 2(\langle \mathbf{x}, \mathbf{w} \rangle - y) \mathbf{x}^T \end{aligned}$$

Example 2

Assume $\mathbf{X} \in \mathbb{R}^{m \times n}, \quad \mathbf{w} \in \mathbb{R}^n, \quad \mathbf{y} \in \mathbb{R}^m$

$$z = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$$

Compute $\frac{\partial z}{\partial \mathbf{w}}$

$$\begin{aligned} \mathbf{a} &= \mathbf{X}\mathbf{w} \\ \mathbf{b} &= \mathbf{a} - \mathbf{y} \\ z &= \|\mathbf{b}\|^2 \end{aligned}$$

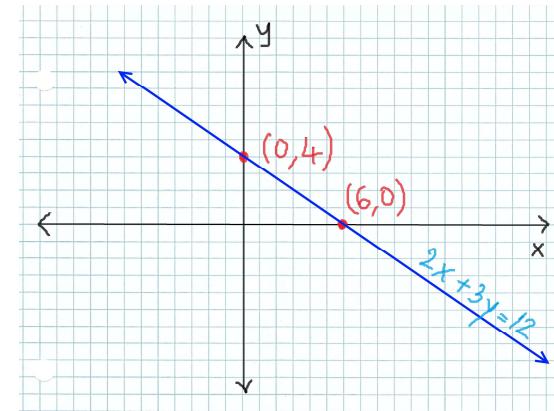
$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{x}}$$

$$\begin{aligned} \frac{\partial z}{\partial \mathbf{w}} &= \frac{\partial z}{\partial \mathbf{b}} \frac{\partial \mathbf{b}}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{w}} \\ &= \frac{\partial \|\mathbf{b}\|^2}{\partial \mathbf{b}} \frac{\partial \mathbf{a} - \mathbf{y}}{\partial \mathbf{a}} \frac{\partial \mathbf{X}\mathbf{w}}{\partial \mathbf{w}} \\ &= 2\mathbf{b}^T \times \mathbf{I} \times \mathbf{X} \\ &= 2(\mathbf{X}\mathbf{w} - \mathbf{y})^T \mathbf{X} \end{aligned}$$

Linear and Logistic Regression, Classification

Vahid Tarokh
ECE685D, Fall 2025

Linear Methods



A Simplified Model

Assumption 1

The key factors impacting y are denoted by x_1, x_2, x_3

Assumption 2

The value of y is a weighted sum over the key factors

$$y = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_0$$

Weights and bias are determined later.

Linear Least Squares

Given a vector of d -dimensional inputs $\mathbf{x} = (x_1, x_2, \dots, x_d)^T$, we want to predict the target (response) using the linear model:

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_d x_d = w_0 + \sum_{j=1}^d w_j x_j.$$

The term w_0 is the intercept, or often called bias term. It will be convenient to include the constant variable 1 in \mathbf{x} and write:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{x}^T \mathbf{w}.$$

Observe a **training set** consisting of N observations

$$\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N)^T,$$

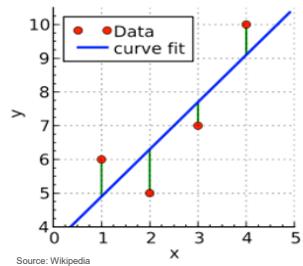
together with the corresponding target values

$$\mathbf{t} = (t_1, t_2, \dots, t_N)^T.$$

Note that \mathbf{X} is an $N \times (d+1)$ matrix.

Linear Least Squares

One option is to minimize **the sum of the squares of the errors** between the predictions $y(\mathbf{x}_n, \mathbf{w})$ for each data point \mathbf{x}_n and the corresponding real-valued targets t_n .

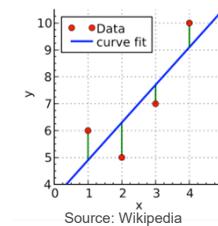


Loss function: sum-of-squared error function:

$$\begin{aligned} E(\mathbf{w}) &= \frac{1}{2} \sum_{n=1}^N (\mathbf{x}_n^T \mathbf{w} - t_n)^2 \\ &= \frac{1}{2} (\mathbf{X}\mathbf{w} - \mathbf{t})^T (\mathbf{X}\mathbf{w} - \mathbf{t}). \end{aligned}$$

Linear Least Squares

If $\mathbf{X}^T \mathbf{X}$ is nonsingular, then the unique solution is given by:

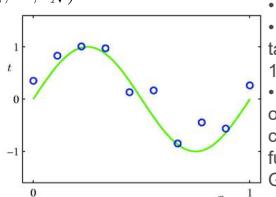


optimal weights
 $\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{t}$
vector of target values
the design matrix has one input vector per row

- At an arbitrary input \mathbf{x}_0 , the prediction is $y(\mathbf{x}_0, \mathbf{w}) = \mathbf{x}_0^T \mathbf{w}^*$.
- The entire model is characterized by $d+1$ parameters \mathbf{w}^* .

Example: Polynomial Curve Fitting

Consider observing a **training set** consisting of N 1-dimensional observations: $\mathbf{x} = (x_1, x_2, \dots, x_N)^T$, together with corresponding real-valued targets: $\mathbf{t} = (t_1, t_2, \dots, t_N)^T$.



- The green plot is the true function
- The training data was generated by $\sin(2\pi x)$, taking x_n spaced uniformly between [0 1].
- The target set (blue circles) was obtained by first computing the corresponding values of the sin function, and then adding a small Gaussian noise.

Goal: Fit the data using a polynomial function of the form:

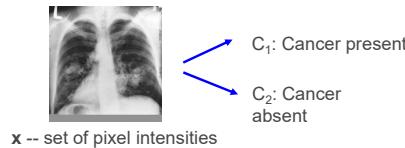
$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \dots + w_M x^M = \sum_{j=0}^M w_j x^j.$$

Note: the polynomial function is a nonlinear function of x , but it is a linear function of the coefficients \mathbf{w} ! **Linear Models**.

Classification

Classification

- The goal of classification is to assign an input \mathbf{x} into one of K discrete classes C_k , where $k=1,..,K$.
- Typically, each input is assigned only to one class.
- Example:** The input vector \mathbf{x} is the set of pixel intensities, and the output variable t will represent the presence of cancer, class C_1 , or absence of cancer, class C_2 .



Linear Classification

- The goal of classification is to assign an input \mathbf{x} into one of K discrete classes C_k , where $k=1,..,K$.
- The input space is divided into decision regions whose boundaries are called **decision boundaries** or **decision surfaces**.
- We will consider **linear models for classification**. Remember, in the simplest linear regression case, **the model is linear in parameters**:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{x}^T \mathbf{w} + w_0.$$

adaptive parameters fixed nonlinear function:
activation function

- For classification, we need to predict discrete class labels, or posterior probabilities that lie in the range of (0,1), so we use a nonlinear function.

Linear Classification

$$y(\mathbf{x}, \mathbf{w}) = f(\mathbf{x}^T \mathbf{w} + w_0).$$

- The **decision surfaces** correspond to $y(\mathbf{x}, \mathbf{w}) = \text{const}$, so that $\mathbf{x}^T \mathbf{w} + w_0 = \text{const}$, and hence the **decision surfaces are linear functions of \mathbf{x} , even if the activation function is nonlinear**.
- This class of models is called **generalized linear models**.
- Note that these models are no longer linear in parameters, due to the presence of nonlinear activation function.
- This leads to more complex analytical and computational properties, compared to linear regression.
- Note that we can make a **fixed nonlinear transformation of the input variables** using a vector of basis functions $\phi(\mathbf{x})$, as we did for regression models.

Notation

- In the case of two-class problems, we can use the binary representation for the target value $t \in \{0,1\}$ such that $t=1$ represents the **positive class** and $t=0$ represents the **negative class**.
 - We can interpret the value of t as the **probability of the positive class**, and the output of the model can be represented as the probability that the model assigns to the positive class.
- If there are K classes, we use a **1-of- K encoding scheme**, in which t is a vector of length K containing a single 1 for the correct class and 0 elsewhere.
 - For example, if we have $K=5$ classes, then an input that belongs to class 2 would be given a target vector:

$$t = (0, 1, 0, 0, 0)^T.$$

- We can interpret a vector t as a vector of class probabilities.

Three Approaches to Classification

- **First approach:** Construct a **discriminant function** that directly maps each input vector to a specific class.
- **Second approach:** Model the **decision regions** and then use this to make optimal decisions.
- There are two alternative approaches:
 - **Discriminative Approach:** Model $p(\mathcal{C}_k|\mathbf{x})$, directly, for example by representing them as parametric models, and optimize for parameters using the training set (e.g. logistic regression).
 - **Generative Approach:** Model class conditional densities $p(\mathbf{x}|\mathcal{C}_k)$ together with the prior probabilities $p(\mathcal{C}_k)$ for the classes. Infer posterior probability using Bayes' rule:

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})}.$$
- For example, we could fit multivariate Gaussians to the input vectors of each class. Given a test vector, we see under which Gaussian the test vector is most probable.

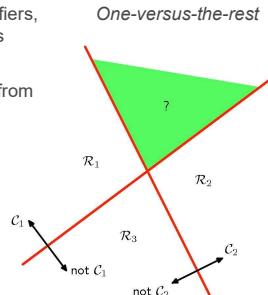
Discriminant Functions

- Consider: $y(\mathbf{x}) = \mathbf{x}^T \mathbf{w} + w_0$.
 - Assign \mathbf{x} to C_1 if $y(\mathbf{x}) \geq 0$, and class C_2 otherwise.
 - Decision boundary: $y(\mathbf{x}) = 0$.
 - If two points \mathbf{x}_A and \mathbf{x}_B lie on the decision surface, then:

$$y(\mathbf{x}_A) = y(\mathbf{x}_B) = 0, \quad \mathbf{w}^T(\mathbf{x}_A - \mathbf{x}_B) = 0.$$
 - \mathbf{w} is orthogonal to the decision surface.
 - If \mathbf{x} is a point on the decision surface, then: $\frac{\mathbf{w}^T \mathbf{x}}{\|\mathbf{w}\|} = -\frac{w_0}{\|\mathbf{w}\|}$.
 - Hence w_0 determines the location of the decision surface.
-

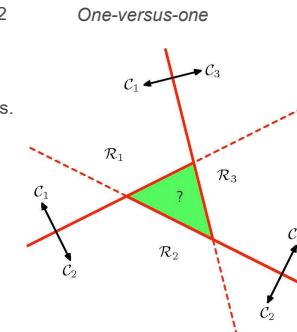
Multiple Classes

- Consider the extension of linear discriminants to $K > 2$ classes.
- One option is to use $K-1$ classifiers, each of which solves a two class problem:
 - Separate points in class \mathcal{C}_k from points not in that class.
- There are regions in input space that are ambiguously classified.



Multiple Classes

- Consider the extension of linear discriminants to $K > 2$ classes.
- An alternative is to use $K(K-1)/2$ binary discriminant functions.
 - Each function discriminates between two particular classes.
- Similar problem of ambiguous regions.



Simple Solution

- Use K linear discriminant functions of the form:
 $y_k(\mathbf{x}) = \mathbf{x}^T \mathbf{w}_k + w_{k0}$, where $k = 1, \dots, K$.
- Assign \mathbf{x} to class C_k , if $y_k(\mathbf{x}) > y_j(\mathbf{x}) \forall j \neq k$ (pick the max).
- This is guaranteed to give decision boundaries that are singly connected and convex.

- For any two points that lie inside the region R_k :

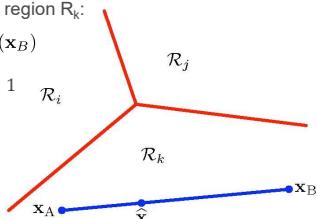
$$y_k(\mathbf{x}_A) > y_j(\mathbf{x}_A) \text{ and } y_k(\mathbf{x}_B) > y_j(\mathbf{x}_B)$$

implies that for any positive $0 < \alpha < 1$

$$y_k(\alpha \mathbf{x}_A + (1 - \alpha) \mathbf{x}_B) >$$

$$y_j(\alpha \mathbf{x}_A + (1 - \alpha) \mathbf{x}_B)$$

due to linearity of the discriminant functions.



The Perceptron Algorithm

- We now consider another example of a linear discriminant model.
- Consider the following generalized linear model of the form

$$y(\mathbf{x}) = f(\mathbf{w}^T \phi(\mathbf{x}))$$

where nonlinear activation function $f(\cdot)$ is given by a step function:

$$f(a) = \begin{cases} +1 & a \geq 0 \\ -1 & a < 0 \end{cases}$$

and \mathbf{x} is transformed using a fixed nonlinear transformation $\phi(\mathbf{x})$.

- Hence we have a two-class model.

The Perceptron Algorithm

- A natural choice of error function would be the total number of misclassified examples (but hard to optimize, discontinuous).

- We will consider an alternative error function.

- First, note that:

- Patterns \mathbf{x}_n in Class C_1 should satisfy:

$$\mathbf{w}^T \phi(\mathbf{x}_n) > 0$$

- Patterns \mathbf{x}_n in Class C_2 should satisfy:

$$\mathbf{w}^T \phi(\mathbf{x}_n) < 0$$

- Using the target coding $t \in \{-1, +1\}$, we see that we would like all patterns to satisfy:

$$\mathbf{w}^T \phi(\mathbf{x}_n) t_n > 0$$

Error Function

- Using the target coding $t \in \{-1, +1\}$, we see that we would like all patterns to satisfy:

$$\mathbf{w}^T \phi(\mathbf{x}_n) t_n > 0$$

- The error function is therefore given by:

$$E_P(\mathbf{w}) = - \sum_{n \in M} \mathbf{w}^T \phi(\mathbf{x}_n) t_n$$

M denotes the set of all misclassified patterns

- The error function is linear in \mathbf{w} in regions of \mathbf{w} space where the example is misclassified.

- The error function is piece-wise linear ([show this](#)).

Error Function

- We can use gradient descent. Given a misclassified example, the change in weight is given by:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla E_p(\mathbf{w}) = \mathbf{w}^t + \eta \phi(\mathbf{x}_n) t_n,$$

where η is the learning rate.

- Since the perceptron function $y(\mathbf{x}) = f(\mathbf{w}^T \phi(\mathbf{x}))$ is unchanged if we multiply \mathbf{w} by a constant, we set $\eta = 1$.

- Note that the contribution to the error from a misclassified example will be reduced:

$$\begin{aligned} -\mathbf{w}^{(t+t)T} \phi(\mathbf{x}_n) t_n &= -\mathbf{w}^{(t)T} \phi(\mathbf{x}_n) t_n - (\phi(\mathbf{x}_n) t_n)^T (\phi(\mathbf{x}_n) t_n) \\ &< -\mathbf{w}^{(t)T} \phi(\mathbf{x}_n) t_n \end{aligned}$$

Always positive

Error Function

- Note that the contribution to the error from a misclassified example will be reduced:

$$\begin{aligned} -\mathbf{w}^{(t+t)T} \phi(\mathbf{x}_n) t_n &= -\mathbf{w}^{(t)T} \phi(\mathbf{x}_n) t_n - (\phi(\mathbf{x}_n) t_n)^T (\phi(\mathbf{x}_n) t_n) \\ &< -\mathbf{w}^{(t)T} \phi(\mathbf{x}_n) t_n \end{aligned}$$

Always positive

- However, the change in \mathbf{w} may cause some previously correctly classified points to be misclassified.

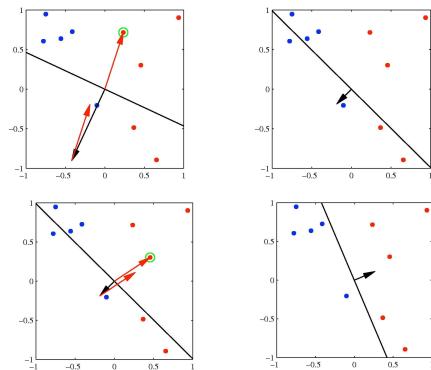
- No convergence guarantees in general.**

- If there exists an exact solution (if the training set is linearly separable), then the perceptron learning algorithm is guaranteed to find an exact solution in finite number of steps.

- The perceptron does not provide probabilistic outputs, nor does it generalize readily to K>2 classes.

Illustration of Convergence

- Convergence of the perceptron learning algorithm



Three Approaches to Classification

- Construct a **discriminant function** that directly maps each input vector to a specific class.

- Model the conditional probability distribution $p(\mathcal{C}_k | \mathbf{x})$, and then use this distribution to make optimal decisions.

- There are two alternative approaches:

- Discriminative Approach:** Model $p(\mathcal{C}_k | \mathbf{x})$, directly, for example by representing them as parametric models, and optimize for parameters using the training set (e.g. logistic regression).

- Generative Approach:** Model class conditional densities $p(\mathbf{x} | \mathcal{C}_k)$ together with the prior probabilities $p(\mathcal{C}_k)$ for the classes. Infer posterior probability using Bayes' rule:

$$p(\mathcal{C}_k | \mathbf{x}) = \frac{p(\mathbf{x} | \mathcal{C}_k) p(\mathcal{C}_k)}{p(\mathbf{x})}.$$

We will consider next.

Probabilistic Generative Models

- Model class conditional densities $p(\mathbf{x}|\mathcal{C}_k)$ separately for each class, as well as the **class priors** $p(\mathcal{C}_k)$.
- Consider the case of two classes. The posterior probability of class \mathcal{C}_1 is given by:

$$\begin{aligned} p(\mathcal{C}_1|\mathbf{x}) &= \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1) + p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} \\ &= \frac{1}{1 + \exp(-a)} = \sigma(a), \end{aligned}$$

where we defined:

$$a = \ln \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} = \ln \frac{p(\mathcal{C}_1|\mathbf{x})}{1 - p(\mathcal{C}_1|\mathbf{x})},$$

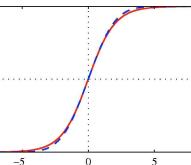
which is known as the **logit function**. It represents the log of the ratio of probabilities of two classes, also known as the **log-odds**.

Sigmoid Function

- The posterior probability of class \mathcal{C}_1 is given by:

$$\begin{aligned} p(\mathcal{C}_1|\mathbf{x}) &= \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1) + p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} \\ &= \frac{1}{1 + \exp(-a)} = \sigma(a), \end{aligned}$$

Logistic sigmoid function



- The term **sigmoid** means **S-shaped**: it maps the whole real axis into (0 1).

- It satisfies:

$$\sigma(-a) = 1 - \sigma(a), \quad \frac{d}{da}\sigma(a) = \sigma(a)(1 - \sigma(a)).$$

Softmax Function

- For case of $K>2$ classes, we have the following **multi-class generalization**:

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{\sum_j p(\mathbf{x}|\mathcal{C}_j)p(\mathcal{C}_j)} = \frac{\exp(a_k)}{\sum_j \exp(a_j)}, \quad a_k = \ln[p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)].$$

- This **normalized exponential** is also known as the **softmax function**, as it represents a **smoothed version of the "max"** function:

if $a_k \gg a_j, \forall j \neq k$, then $p(\mathcal{C}_k|\mathbf{x}) \approx 1, p(\mathcal{C}_j|\mathbf{x}) \approx 0$.

- We now look at some specific forms of class conditional distributions.

Example of Continuous Inputs

- Assume that the input vectors **for each class are from a Gaussian distribution**, and all classes share the same covariance matrix:

$$p(\mathbf{x}|\mathcal{C}_k) = \frac{1}{(2\pi)^{D/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)\right).$$

- For the case of two classes, the posterior is logistic function:

$$p(\mathcal{C}_k|\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x} + w_0),$$

where we have defined:

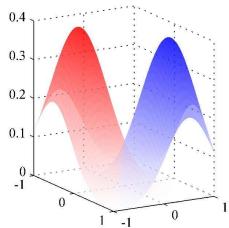
$$\mathbf{w} = \Sigma^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2),$$

$$w_0 = -\frac{1}{2}\boldsymbol{\mu}_1^T \Sigma^{-1} \boldsymbol{\mu}_1 + \frac{1}{2}\boldsymbol{\mu}_2^T \Sigma^{-1} \boldsymbol{\mu}_2 + \ln \frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)}.$$

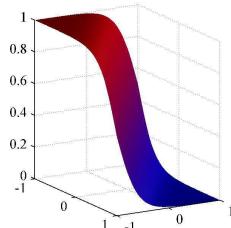
- The **quadratic terms in \mathbf{x} cancel** (due to the assumption of common covariance matrices).

- This leads to a **linear function of \mathbf{x}** in the argument of logistic sigmoid. Hence **the decision boundaries are linear in input space**.

Example of Two Gaussian Models



Class-conditional densities for two classes



The corresponding posterior probability $p(\mathcal{C}_1|x)$, given by the sigmoid function of a linear function of \mathbf{x} .

Case of K Classes

- For the case of K classes, the posterior is a softmax function:

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{\sum_j p(\mathbf{x}|\mathcal{C}_j)p(\mathcal{C}_j)} = \frac{\exp(a_k)}{\sum_j \exp(a_j)},$$

$$a_k = \mathbf{w}_k^T \mathbf{x} + w_{k0},$$

where, similar to the 2-class case, we have defined:

$$\mathbf{w}_k = \Sigma^{-1} \boldsymbol{\mu}_k,$$

$$w_{k0} = -\frac{1}{2} \boldsymbol{\mu}_k^T \Sigma^{-1} \boldsymbol{\mu}_k + \ln p(\mathcal{C}_k).$$

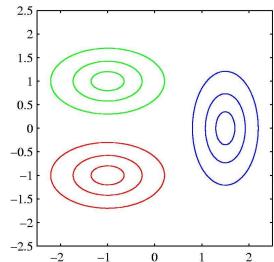
- Again, the decision boundaries are linear in input space.

- If we allow each class-conditional density to have its own covariance, we will obtain quadratic functions of \mathbf{x} .

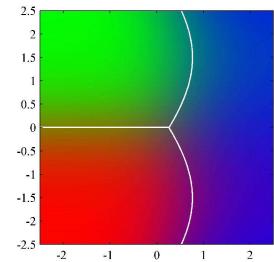
- This leads to a **quadratic discriminant**.

Quadratic Discriminant

The decision boundary is linear when the covariance matrices are the same and quadratic when they are not.



Class-conditional densities for three classes



The corresponding posterior probabilities for three classes.

Maximum Likelihood Solution

- Consider the case of two classes, each having a Gaussian class-conditional density with shared covariance matrix.

- We observe a dataset $\{\mathbf{x}_n, t_n\}$, $n = 1, \dots, N$.

- Here $t_n=1$ denotes class C_1 , and $t_n=0$ denotes class C_2 .

- Also denote $p(\mathcal{C}_1) = \pi$, $p(\mathcal{C}_2) = 1 - \pi$.

- The likelihood function takes form:

$$p(\mathbf{t}, \mathbf{X} | \pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \Sigma) = \prod_{n=1}^N \left[\pi \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_1, \Sigma) \right]^{t_n} \left[(1 - \pi) \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_2, \Sigma) \right]^{1-t_n}.$$

↑ ↑
Data points from class C_1 . Data points from class C_2 .

- As usual, we will maximize the log of the likelihood function.

Maximum Likelihood Solution

$$p(\mathbf{t}, \mathbf{X} | \pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \boldsymbol{\Sigma}) = \prod_{n=1}^N \left[\pi \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_1, \boldsymbol{\Sigma}) \right]^{t_n} \left[(1 - \pi) \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_2, \boldsymbol{\Sigma}) \right]^{1-t_n}.$$

- Maximizing the respect to π , we look at the terms of the log-likelihood functions that depend on π :

$$\sum_n [t_n \ln \pi + (1 - t_n) \ln(1 - \pi)] + \text{const.}$$

Differentiating, we get:
 $\pi = \frac{1}{N} \sum_{n=1}^N t_n = \frac{N_1}{N_1 + N_2}.$

- Maximizing the respect to $\boldsymbol{\mu}_1$, we look at the terms of the log-likelihood functions that depend on $\boldsymbol{\mu}_1$:

$$\sum_n t_n \ln \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_1, \boldsymbol{\Sigma}) = -\frac{1}{2} \sum_n t_n (\mathbf{x}_n - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_1) + \text{const.}$$

Differentiating, we get:
 $\boldsymbol{\mu}_1 = \frac{1}{N_1} \sum_{n=1}^N t_n \mathbf{x}_n.$ And similarly:
 $\boldsymbol{\mu}_2 = \frac{1}{N_2} \sum_{n=1}^N (1 - t_n) \mathbf{x}_n.$

Maximum Likelihood Solution

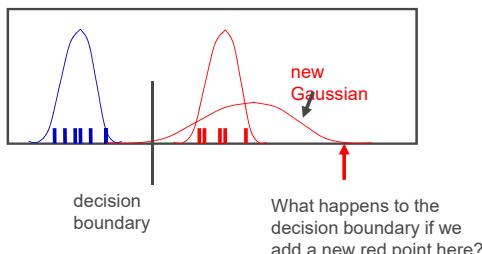
$$p(\mathbf{t}, \mathbf{X} | \pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \boldsymbol{\Sigma}) = \prod_{n=1}^N \left[\pi \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_1, \boldsymbol{\Sigma}) \right]^{t_n} \left[(1 - \pi) \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_2, \boldsymbol{\Sigma}) \right]^{1-t_n}.$$

- Maximizing the respect to $\boldsymbol{\Sigma}$:

$$\begin{aligned} & -\frac{1}{2} \sum_n t_n \ln |\boldsymbol{\Sigma}| - \frac{1}{2} \sum_n t_n (\mathbf{x}_n - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_1) \\ & -\frac{1}{2} \sum_n (1 - t_n) \ln |\boldsymbol{\Sigma}| - \frac{1}{2} \sum_n (1 - t_n) (\mathbf{x}_n - \boldsymbol{\mu}_2)^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_2) \\ & = -\frac{N}{2} \ln |\boldsymbol{\Sigma}| - \frac{N}{2} \text{Tr}(\boldsymbol{\Sigma}^{-1} \mathbf{S}). \end{aligned}$$

- Here we defined:
 $\mathbf{S} = \frac{N_1}{N_1} \mathbf{S}_1 + \frac{N_2}{N_1} \mathbf{S}_2,$
 $\mathbf{S}_1 = \frac{1}{N_1} \sum_{n \in \mathcal{C}_1} (\mathbf{x}_n - \boldsymbol{\mu}_1)(\mathbf{x}_n - \boldsymbol{\mu}_1)^T,$
 $\mathbf{S}_2 = \frac{1}{N_2} \sum_{n \in \mathcal{C}_2} (\mathbf{x}_n - \boldsymbol{\mu}_2)(\mathbf{x}_n - \boldsymbol{\mu}_2)^T.$
- Using standard results (see HW) for a Gaussian distribution we have: $\boldsymbol{\Sigma} = \mathbf{S}.$
- Maximum likelihood solution represents a weighted average of the covariance matrices associated with each of the two classes.

Example



- For generative fitting, the red mean moves rightwards but the decision boundary moves leftwards! If you believe the data is Gaussian, this is reasonable.

Three Approaches to Classification

- Construct a **discriminant function** that directly maps each input vector to a specific class.

- Model the conditional probability distribution $p(\mathcal{C}_k | \mathbf{x})$, and then use this distribution to make optimal decisions.

- There are two approaches:

- **Discriminative Approach:** Model $p(\mathcal{C}_k | \mathbf{x})$, directly, for example by representing them as parametric models, and optimize for parameters using the training set (e.g. logistic regression).

- **Generative Approach:** Model class conditional densities $p(\mathbf{x} | \mathcal{C}_k)$ together with the prior probabilities $p(\mathcal{C}_k)$ for the classes. Infer posterior probability using Bayes' rule:

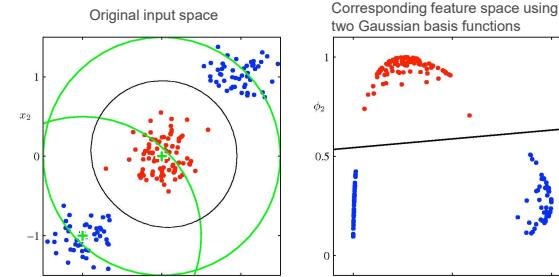
$$p(\mathcal{C}_k | \mathbf{x}) = \frac{p(\mathbf{x} | \mathcal{C}_k) p(\mathcal{C}_k)}{p(\mathbf{x})}.$$

We will consider next.

Fixed Basis Functions

- So far, we have considered classification models that work directly in the input space.
- All considered algorithms are equally applicable if we first make a fixed nonlinear transformation of the input space using vector of basis functions $\phi(\mathbf{x})$.
- Decision boundaries will be linear in the feature space ϕ , but would correspond to nonlinear boundaries in the original input space \mathbf{x} .
- Classes that are linearly separable in the feature space $\phi(\mathbf{x})$ need not be linearly separable in the original input space.

Linear Basis Function Models



- We define two Gaussian basis functions with centers shown by green crosses, and with contours shown by the green circles.
- Linear decision boundary (right) is obtained using logistic regression, and corresponds to nonlinear decision boundary in the input space (left, black curve).

Logistic Regression

Logistic Regression

- Let us look at the two-class classification problem.
- We have seen that the posterior probability of class C_1 can be written as a sigmoid function:

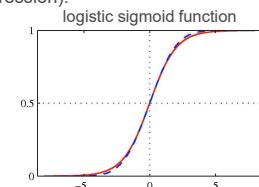
$$p(C_1|\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x})} = \sigma(\mathbf{w}^T \mathbf{x}),$$

where $p(C_2|\mathbf{x}) = 1 - p(C_1|\mathbf{x})$, and we omit the bias term for clarity.

- This model is known as **logistic regression** (although this is a model for classification rather than regression).

Note that for generative models, we would first determine the class conditional densities and class-specific priors, and then use Bayes' rule to obtain the posterior probabilities.

Here we model $p(C_k|\mathbf{x})$ directly.



Logistic Regression

- We observed a training dataset $\{\mathbf{x}_n, t_n\}$, $n = 1, \dots, N$; $t_n \in \{0, 1\}$.
- Maximize the probability of getting the label right, so the likelihood function takes form:

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}) = \prod_{n=1}^N \left[y_n^{t_n} (1 - y_n)^{1-t_n} \right], \quad y_n = \sigma(\mathbf{w}^T \mathbf{x}_n).$$

- Taking the negative log of the likelihood, we can define the **cross-entropy error function** (that we want to minimize):

$$E(\mathbf{w}) = -\ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}) = -\sum_{n=1}^N \left[t_n \ln y_n + (1 - t_n) \ln(1 - y_n) \right] = \sum_{n=1}^N E_n.$$

- Differentiating and using the chain rule:

$$\frac{d}{dy_n} E_n = \frac{y_n - t_n}{y_n(1 - y_n)}, \quad \frac{d}{d\mathbf{w}} y_n = y_n(1 - y_n)\mathbf{x}_n, \quad \boxed{\frac{d}{da}\sigma(a) = \sigma(a)(1 - \sigma(a))}.$$

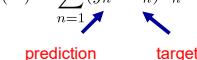
$$\frac{d}{d\mathbf{w}} E_n = \frac{dE_n}{dy_n} \frac{dy_n}{d\mathbf{w}} = (y_n - t_n)\mathbf{x}_n.$$

- Note that the factor involving the derivative of the logistic function cancelled.

ML for Logistic Regression

- We therefore obtain:

$$\nabla E(\mathbf{w}) = \sum_{n=1}^N (y_n - t_n)\mathbf{x}_n.$$



- This takes exactly the same form as the **gradient of the sum-of-squares error function** for the linear regression model.
- Unlike in linear regression, there is **no closed form solution**, due to nonlinearity of the logistic sigmoid function.
- The **error function** can be optimized using standard gradient-based (or more advanced) optimization techniques.
- Easy to adapt to the **online learning setting**.

Multiclass Logistic Regression

- For the multiclass case, we represent posterior probabilities by a softmax transformation of linear functions of input variables:

$$p(C_k|\mathbf{x}) = y_k(\mathbf{x}) = \frac{\exp(\mathbf{w}_k^T \mathbf{x})}{\sum_j \exp(\mathbf{w}_j^T \mathbf{x})}.$$

- Unlike in generative models, here we will use maximum likelihood to determine parameters of this discriminative model directly.

- As usual, we observed a dataset $\{\mathbf{x}_n, t_n\}$, $n = 1, \dots, N$, where we use 1-of-K encoding for the target vector \mathbf{t}_n .

- So if \mathbf{x}_n belongs to class C_k , then \mathbf{t} is a binary vector of length K containing a single 1 for element k (the correct class) and 0 elsewhere.

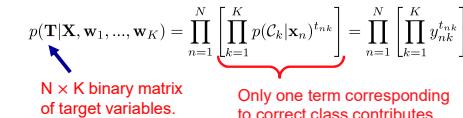
- For example, if we have K=5 classes, then an input that belongs to class 2 would be given a target vector:

$$\mathbf{t} = (0, 1, 0, 0, 0)^T.$$

Multiclass Logistic Regression

- We can write down the likelihood function:

$$p(\mathbf{T}|\mathbf{X}, \mathbf{w}_1, \dots, \mathbf{w}_K) = \prod_{n=1}^N \left[\prod_{k=1}^K p(C_k|\mathbf{x}_n)^{t_{nk}} \right] = \prod_{n=1}^N \left[\prod_{k=1}^K y_{nk}^{t_{nk}} \right]$$



where $y_{nk} = p(C_k|\mathbf{x}_n) = \frac{\exp(\mathbf{w}_k^T \mathbf{x}_n)}{\sum_j \exp(\mathbf{w}_j^T \mathbf{x}_n)}$.

- Taking the negative logarithm gives the **cross-entropy entropy function** for multi-class classification problem:

$$E(\mathbf{w}_1, \dots, \mathbf{w}_K) = -\ln p(\mathbf{T}|\mathbf{X}, \mathbf{w}_1, \dots, \mathbf{w}_K) = -\sum_{n=1}^N \left[\sum_{k=1}^K t_{nk} \ln y_{nk} \right].$$

- Taking the gradient:

$$\nabla E_{\mathbf{w}_j}(\mathbf{w}_1, \dots, \mathbf{w}_K) = \sum_{n=1}^N (y_{nj} - t_{nj})\mathbf{x}_n.$$

Special Case of Softmax

- If we consider a softmax function for two classes:

$$p(\mathcal{C}_1|\mathbf{x}) = \frac{\exp(a_1)}{\exp(a_1) + \exp(a_2)} = \frac{1}{1 + \exp(-(a_1 - a_2))} = \sigma(a_1 - a_2).$$

- So the **logistic sigmoid is just a special case of the softmax function** that avoids using redundant parameters:
 - Adding the same constant to both a_1 and a_2 has no effect.
 - The over-parameterization of the softmax is because probabilities must add up to one.

From Logistic Regression to Feed-Forward Neural Networks

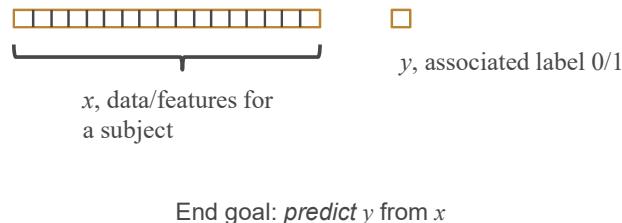
Vahid Tarokh
ECE685D, Fall 2025

Introduction

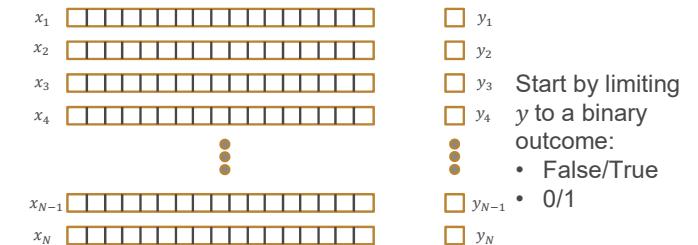
- We will next discuss logistic regression and the construction of neural Neural Networks.
- Important Note: Source of some of my slides (with great appreciation and acknowledgements)
 - Professor David Carlson Slides
 - Professor Alex Smola's slides (available online)
 - Professor Ruslan Salakhutdinov's slides (available online)
 - Professor Hugo Larochelle's class on Neural Networks

Logistic Regression

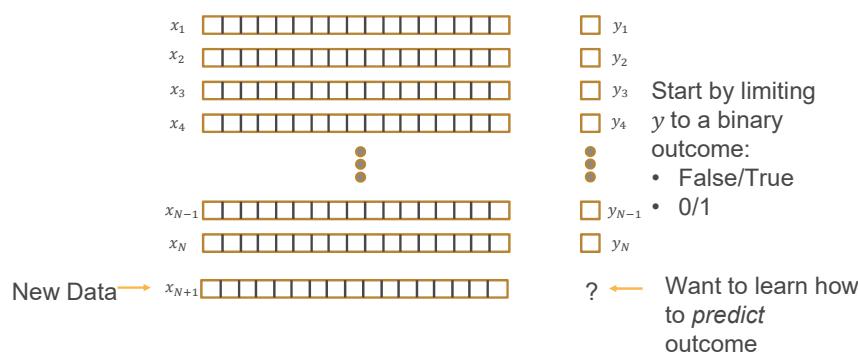
Learning a Predictive Model Based on Labeled Data



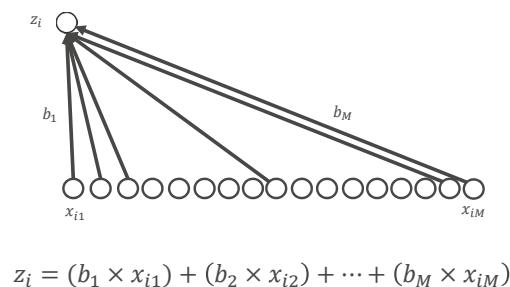
Training Set (Historical Data)



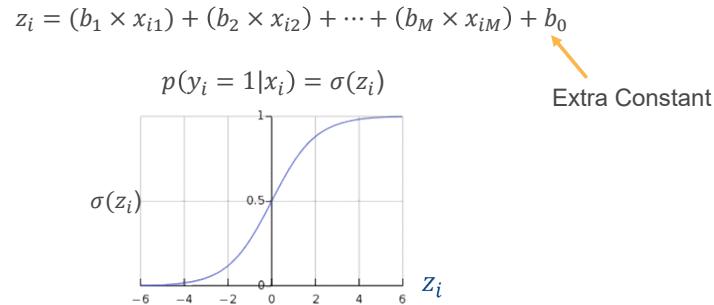
Making Predictions



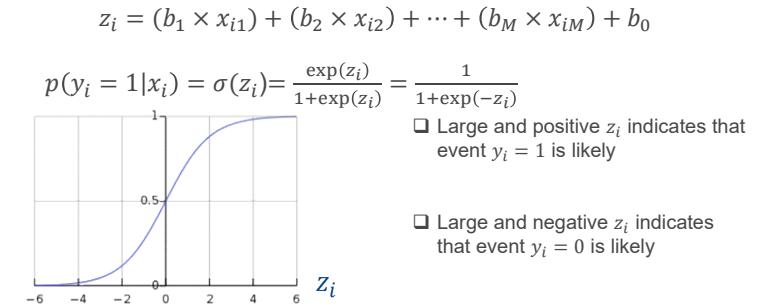
Linear Predictive Model



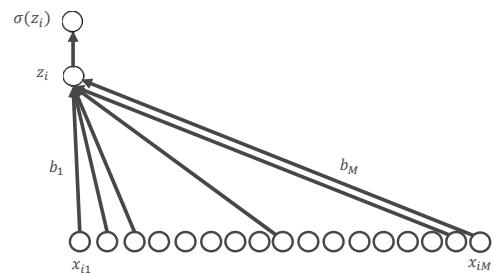
Convert to a Probability



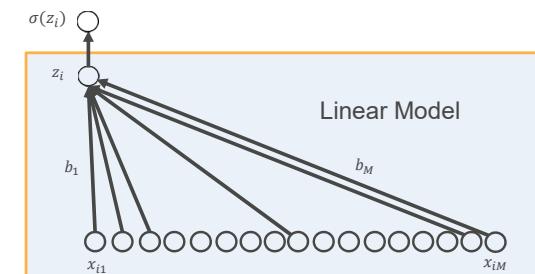
Convert to a Probability



Logistic Regression

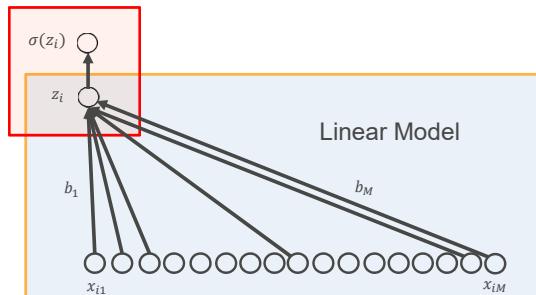


Logistic Regression



Logistic Regression

Convert to Probability



What do the parameters and model mean?

AN EXAMPLE

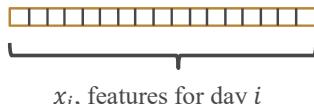
Example

Outcome:

- $y_i = 1$, it rains on day i ;
- $y_i = 0$, it does not rain on day i

Features:

- On day i what is the $\{cloud\ cover, humidity, temperature, air\ pressure, \dots\}$



y_i , did it rain on day i

x_i , features for day i

Example

Outcome: $y_i = 1$, it rains on day i ; $y_i = 0$, it does not rain on day i

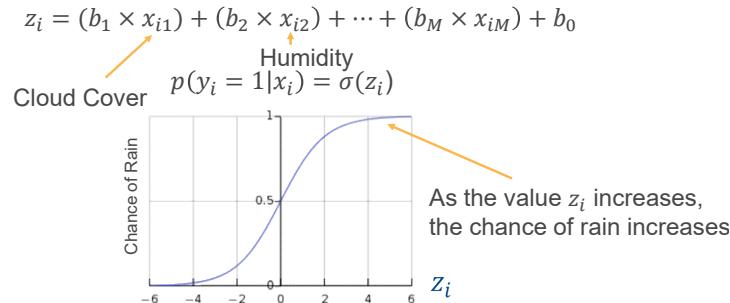
Features: On day i what is the
 $\{1: cloud\ cover, 2: humidity, 3: temperature, \dots\}$

$$z_i = (b_1 \times x_{i1}) + (b_2 \times x_{i2}) + \dots + (b_M \times x_{iM}) + b_0$$

Cloud Cover Humidity

- If cloud cover is positively related to rainfall, b_1 should be positive

Impact on the Sigmoid Function



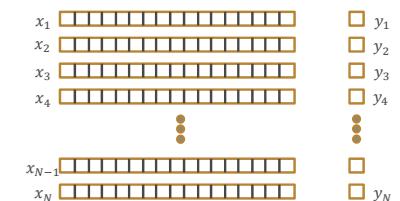
Building the Training Set

Need to learn the parameters

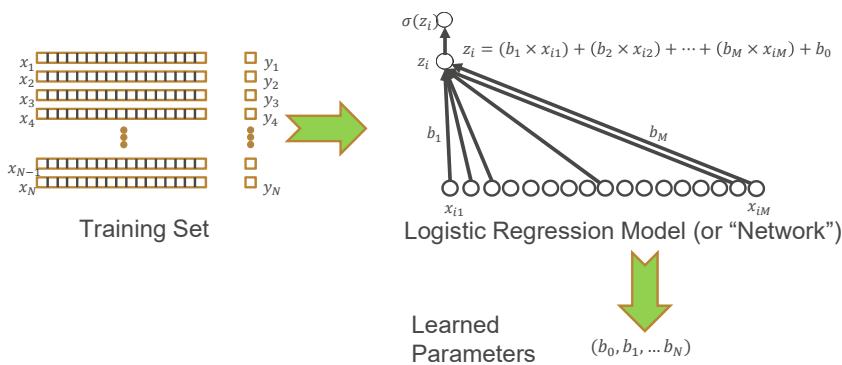
Requires *training data*

Record data from N days

- Capture features: {cloud cover, humidity, temperature, ...}
- Did it rain?

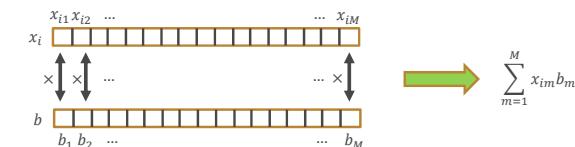


Learning Model Parameters



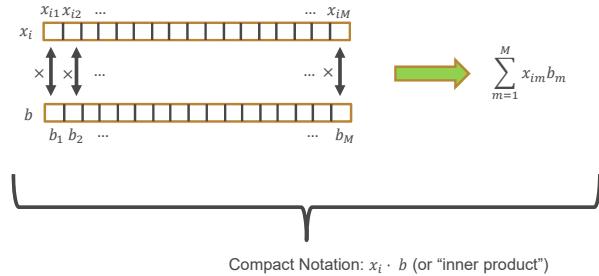
Interpretation of Logistic Regression

$$z_i = (b_1 \times x_{i1}) + (b_2 \times x_{i2}) + \dots + (b_M \times x_{iM}) + b_0$$



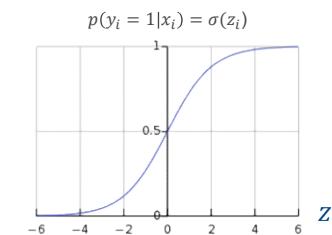
Interpretation of Logistic Regression

$$z_i = (b_1 \times x_{i1}) + (b_2 \times x_{i2}) + \dots + (b_M \times x_{iM})$$



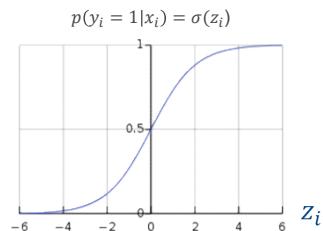
Interpretation of Logistic Regression

$$z_i = b_0 + (b_1 \times x_{i1}) + (b_2 \times x_{i2}) + \dots + (b_M \times x_{iM})$$



Interpretation of Logistic Regression

$$z_i = b_0 + (b_1 \times x_{i1}) + (b_2 \times x_{i2}) + \dots + (b_M \times x_{iM})$$



- ❑ May think of vector b as a template or filter (will visualize to make clear)
- ❑ If x_i is aligned/matched with b , then the sum will be larger
- ❑ The parameter b_0 is a bias to correct for class prevalences

Artificial Neurons

Artificial Neuron

- Neuron pre-activation (or input activation):

$$a(\mathbf{x}) = b + \sum_i w_i x_i = b + \mathbf{w}^\top \mathbf{x}$$

- Neuron output activation:

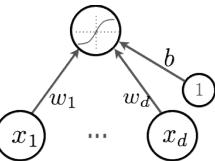
$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$

where

w are the weights (parameters)

b is the bias term

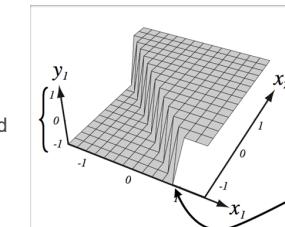
$g(\cdot)$ is called the activation function



Artificial Neuron

- Output activation of the neuron:

$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$



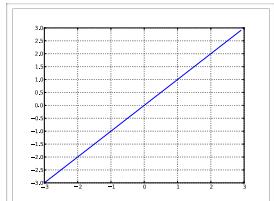
Range is determined by $g(\cdot)$
Bias only changes the position of the ridge
(from Pascal Vincent's slides)

Activation Function

- Linear activation function:

- No nonlinear transformation
- No input squashing

$$g(a) = a$$

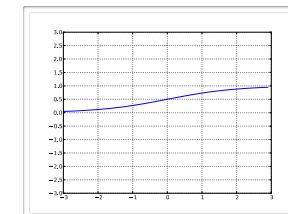


Activation Function

- Sigmoid activation function:

- Squashes the neuron's output between 0 and 1
- Always positive
- Bounded
- Strictly Increasing

$$g(a) = \text{sigm}(a) = \frac{1}{1+\exp(-a)}$$



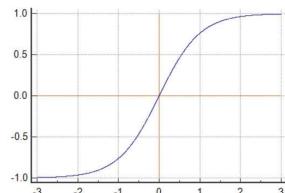
Does this ring a bell?

Activation Function

- Hyperbolic tangent ("tanh") activation function:

- Squashes the neuron's activation between -1 and 1
- Can be positive or negative
- Bounded
- Strictly increasing (wrong plot)

$$g(a) = \tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} = \frac{\exp(2a) - 1}{\exp(2a) + 1}$$

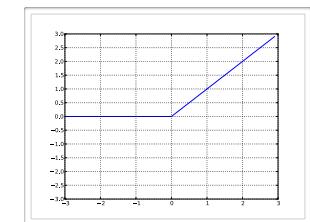


Activation Function

- Rectified linear (ReLU) activation function:

- Bounded below by 0 (always non-negative)
- Tends to produce units with sparse activities
- Not upper bounded
- Strictly increasing

$$g(a) = \text{reclin}(a) = \max(0, a)$$

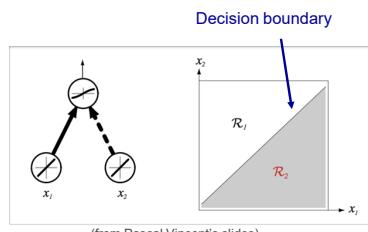


Decision Boundary of a Neuron

- Binary classification:

- With sigmoid, one can interpret neuron as estimating $p(y = 1|x)$
- Interpret as a **logistic classifier**

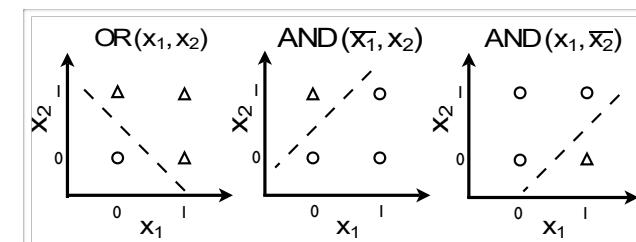
- If activation is greater than 0.5, predict 1
- Otherwise predict 0



Same idea can be applied to a $\tanh()$ activation

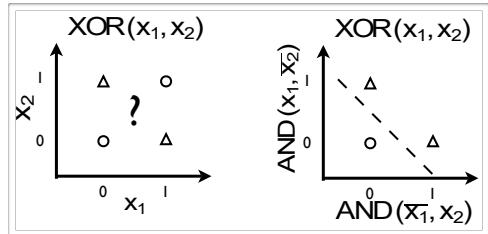
Capacity of a Single Neuron

- Can solve linearly separable problems.



Capacity of a Single Neuron

- Can not solve non-linearly separable problems.



- Need to transform the input into a better representation.
- Remember **basis functions**!

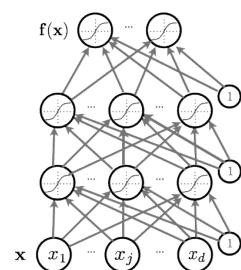
Feed-Forward Neural Nets

Feedforward Neural Networks

- ▶ How neural networks predict

$f(\mathbf{x})$ given an input \mathbf{x} :

- Forward propagation
- Types of units
- Capacity of neural networks



- ▶ How to train neural nets:

- Loss function
- Back-propagation with gradient descent

- ▶ More recent techniques:

- Dropout
- Batch normalization
- Unsupervised Pre-training

Single Hidden Layer Neural Net

- Hidden layer pre-activation:

$$\mathbf{a}(\mathbf{x}) = \mathbf{b}^{(1)} + \mathbf{W}^{(1)}\mathbf{x}$$
$$(a(\mathbf{x})_i = b_i^{(1)} + \sum_j W_{i,j}^{(1)} x_j)$$

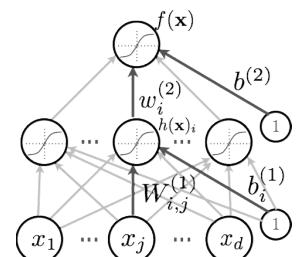
- Hidden layer activation:

$$\mathbf{h}(\mathbf{x}) = \mathbf{g}(\mathbf{a}(\mathbf{x}))$$

- Output layer activation:

$$f(\mathbf{x}) = o\left(b^{(2)} + \mathbf{w}^{(2)\top} \mathbf{h}^{(1)} \mathbf{x}\right)$$

↓
Output activation
function



Softmax Activation Function

- Remember multi-way classification:
 - We need multiple outputs (1 output per class)
 - We need to estimate conditional probability: $p(y = c|x)$
 - Discriminative Learning

- Softmax activation function at the output

$$\mathbf{o}(\mathbf{a}) = \text{softmax}(\mathbf{a}) = \left[\frac{\exp(a_1)}{\sum_c \exp(a_c)} \cdots \frac{\exp(a_C)}{\sum_c \exp(a_c)} \right]^\top$$

- strictly positive
- sums to one

- Predict class with the highest estimated class conditional probability.

Multilayer Neural Net

- Consider a network with L hidden layers.

- layer pre-activation for $k > 0$

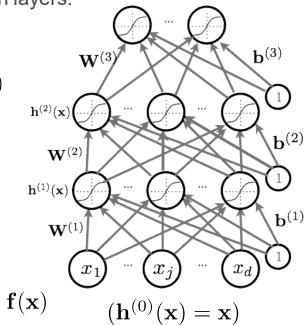
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

- hidden layer activation from 1 to L:

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

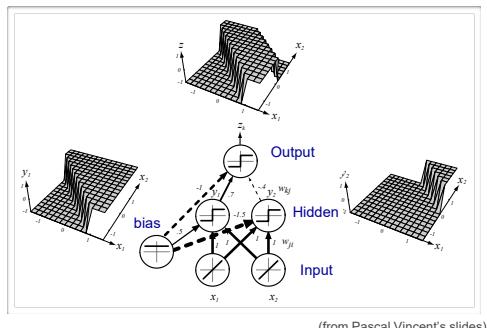
- output layer activation ($k=L+1$):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$

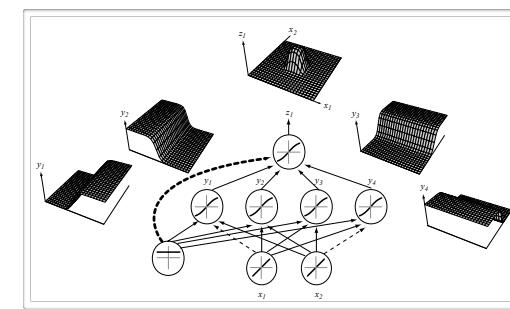


Capacity of Neural Nets

- Consider a single layer neural network

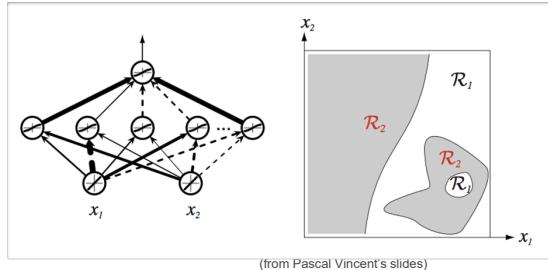


- Consider a single layer neural network



Capacity of Neural Nets

- Consider a single layer neural network



Universal Approximation

The key result is due to great mathematicians Kolmogorov and Arnold (very difficult to prove) established in 1956.

Any continuous function of m inputs can be represented **exactly** by a small (polynomial sized) two-layer network.

$$f(x_1, \dots, x_m) = \sum_{i=1}^{2m+1} g_i \left(\sum_{j=1}^m h_{i,j}(x_j) \right)$$

Where g_i and $h_{i,j}$ are continuous scalar-to-scalar functions.

Universal Approximation

A much more trivial result to prove is:

For any (possibly discontinuous) $f : [0, 1]^m \rightarrow \mathbb{R}$ we have

$$f(x_1, \dots, x_m) = g \left(\sum_i h_i(x_i) \right)$$

for (discontinuous) scalar-to-scalar functions g and h_i .

Proof: Any single real number contains an infinite amount of information.

Select h_i to spread out the digits of its argument so that $\sum_i h_i(x_i)$ contains all the digits of all the x_i .

Universal Approximation

Another relatively straightforward result is due to Cybenko (1989): Any continuous function can be approximated arbitrarily well by a two layer perceptron.

For any continuous $f : [0, 1]^m \rightarrow \mathbb{R}$ and any $\varepsilon > 0$, there exists

$$F(x) = \alpha \cdot \sigma(Wx + \beta)$$

$$= \sum_i \alpha_i \sigma \left(\sum_j W_{i,j} x_j + \beta_i \right)$$

such that for all x in $[0, 1]^m$ we have $|F(x) - f(x)| < \varepsilon$.

Universal Approximation

- Universal Approximation Theorem (Hornik, 1991):
 - “a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units”
- This applies for sigmoid, tanh and many other activation functions.
- However, this does not mean that there is learning algorithm that can find the necessary parameter values.

How to Train Neural Networks

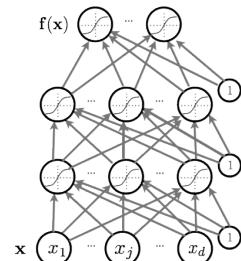
Vahid Tarokh
ECE 685D, Fall 2025

Feedforward Neural Networks

► How neural networks predict

$f(x)$ given an input x :

- Forward propagation
- Types of units
- Capacity of neural networks



► How to train neural nets:

- Loss function
- Back-propagation with gradient descent

► More recent techniques:

- Dropout
- Batch normalization
- Unsupervised Pre-training

Training

• Empirical Risk Minimization:

$$\arg \min_{\theta} \frac{1}{T} \sum_t l(f(\mathbf{x}^{(t)}; \theta), y^{(t)}) + \lambda \Omega(\theta)$$

Loss function Regularizer

• Learning is cast as optimization.

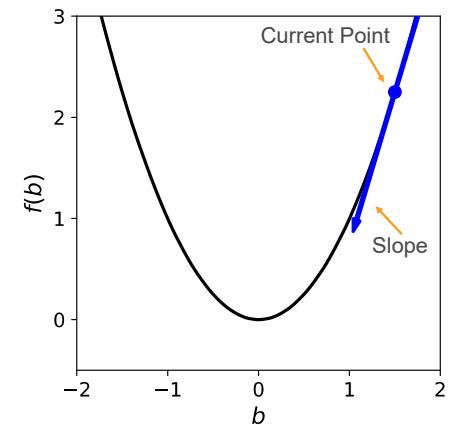
➢ For classification problems, we would like to minimize classification error.

➢ Loss function can sometimes be viewed as a surrogate for what we want to optimize (e.g. upper bound)

GRADIENT DESCENT

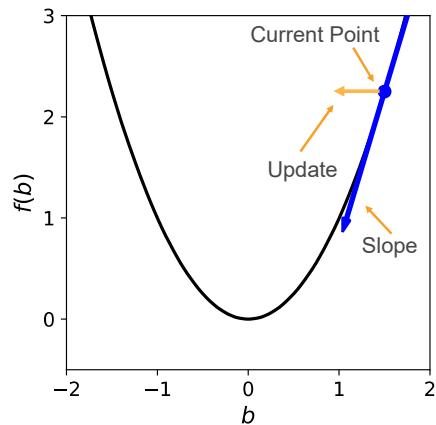
Visualization of Optimization Method

- We want to minimize a mathematical function (i.e. our average loss function)
- One approach is to:
 1. Find the direction pointing "down the hill" (towards a smaller value)
 2. Move a bit in that direction
 3. Repeat 1-2 until satisfied



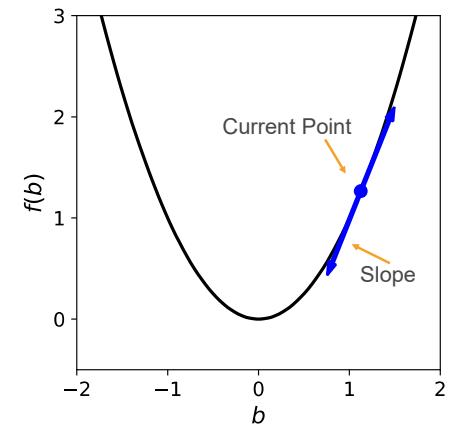
Visualization of Optimization Method

- We want to minimize a mathematical function (i.e. our average loss function)
- One approach is to:
 1. Find the direction pointing "down the hill" (towards a smaller value)
 2. Move a bit in that direction
 3. Repeat 1-2 until satisfied



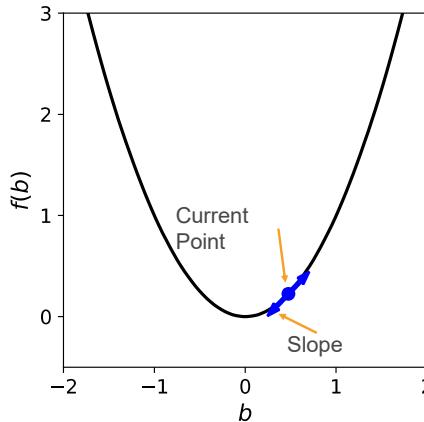
Visualization of Optimization Method

- We want to minimize a mathematical function (i.e. our average loss function)
- One approach is to:
 1. Find the direction pointing "down the hill" (towards a smaller value)
 2. Move a bit in that direction
 3. Repeat 1-2 until satisfied
- This shows the **first** update



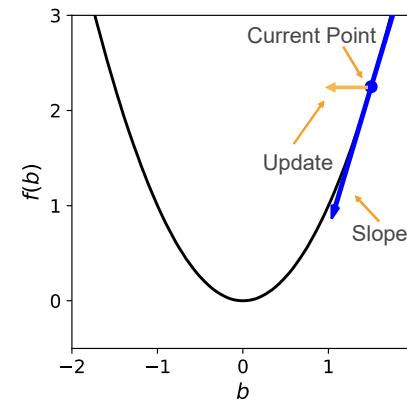
Visualization of Optimization Method

- We want to minimize a mathematical function (i.e. our average loss function)
- One approach is to:
 - Find the direction pointing "down the hill" (towards a smaller value)
 - Move a bit in that direction
 - Repeat 1-2 until satisfied
- This shows the **fourth** update



Mathematical Description of Gradient Descent

- We want to minimize a function $b^* = \arg \min_b f(b)$
- Start at an initial value b^0
- We will run a series of updates to move from b^k to b^{k+1} (i.e. from b^0 to b^1)
- Iteratively run the procedure:
 - Calculate the slope at the current point (For one parameter, this is the derivative. For multiple parameters, this is the *gradient*). $\nabla f(b^k)$. ∇ means gradient or multidimensional slope
 - Move in the direction of the negative gradient with step size α^k : $b^{k+1} = b^k - \alpha^k \nabla f(b^k)$
 - Repeat 1-2 until converged



STOCHASTIC GRADIENT DESCENT

Stochastic Gradient Descent

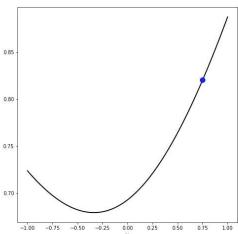
- Perform updates after seeing each example:
 - Initialize: $\theta \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$
 - For $t=1:T$
 - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$
 - $$\Delta = -\nabla_{\theta} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)}) - \lambda \nabla_{\theta} \Omega(\theta)$$
 - $$\theta \leftarrow \theta + \alpha \Delta$$
- Training epoch
= Iteration of all examples

- To train a neural net, we need:

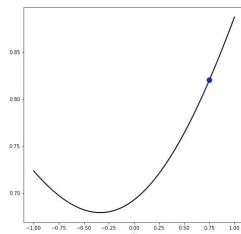
- **Loss function:** $l(f(\mathbf{x}^{(t)}; \theta), y^{(t)})$
- A procedure to **compute gradients**: $\nabla_{\theta} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)})$
- **Regularizer** and its gradient: $\Omega(\theta)$, $\nabla_{\theta} \Omega(\theta)$

Videos for Visualization

Gradient Descent



Stochastic Gradient Descent



Comments on SGD

Stochastic Gradient Descent can update *many more times* than Gradient Descent

Gets ***near*** the solution very quickly

Allows scaling to *big data* (update time doesn't increase with the data size)

In practice, we often use a minibatch, which uses a few data examples to estimate the gradient

Loss Function

- Let us start by considering a classification problem with a softmax output layer.
- We need to estimate: $f(\mathbf{x})_c = p(y=c|\mathbf{x})$
 - We can maximize the log-probability of the correct class given an input: $\log p(y^{(t)} = c|x^{(t)})$
- Alternatively, we can minimize the negative log-likelihood:

$$l(\mathbf{f}(\mathbf{x}), y) = -\sum_c 1_{(y=c)} \log f(\mathbf{x})_c = -\log f(\mathbf{x})_y$$

- This is also known as a **cross-entropy entropy function** for multi-class classification problem (will be discussed more later on).

Stochastic Gradient Descent

- Perform updates after seeing each example:
 - Initialize: $\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$
 - For $t=1:T$
 - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \Delta$$
- Training epoch
Iteration of all examples

- To train a neural net, we need:

- Loss function: $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
- A procedure to compute gradients: $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
- Regularizer and its gradient: $\Omega(\boldsymbol{\theta}), \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$

Multilayer Neural Net: Reminder

- Consider a network with L hidden layers.

- layer pre-activation for $k > 0$

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

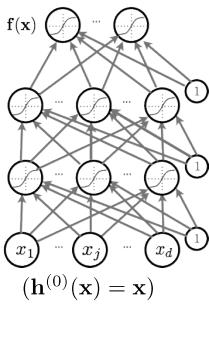
- hidden layer activation from 1 to L:

$$\mathbf{h}^{(k)}(\mathbf{x}) = g(\mathbf{a}^{(k)}(\mathbf{x}))$$

- output layer activation ($k=L+1$):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = o(\mathbf{a}^{(L+1)}(\mathbf{x})) = f(\mathbf{x})$$

Softmax activation function



Gradient Computation

- Loss gradient at output

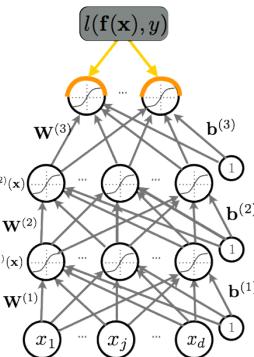
- Partial derivative:

$$\frac{\partial}{\partial f(\mathbf{x})_c} - \log f(\mathbf{x})_y = \frac{-1_{(y=c)}}{f(\mathbf{x})_y}$$

- Gradient:

$$\begin{aligned} \nabla_{f(\mathbf{x})} - \log f(\mathbf{x})_y &= \frac{-1}{f(\mathbf{x})_y} \begin{bmatrix} 1_{(y=0)} \\ \vdots \\ 1_{(y=C-1)} \end{bmatrix} \\ &= \frac{-e(y)}{f(\mathbf{x})_y} \end{aligned}$$

Remember: $f(\mathbf{x})_c = p(y=c|\mathbf{x})$



Gradient Computation

- Loss gradient at output pre-activation

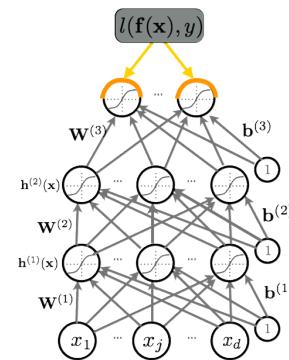
- Partial derivative:

$$\begin{aligned} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} - \log f(\mathbf{x})_y &= - (1_{(y=c)} - f(\mathbf{x})_c) \end{aligned}$$

- Gradient:

$$\nabla_{a^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y = - (e(y) - f(\mathbf{x}))$$

Indicator function



Derivation

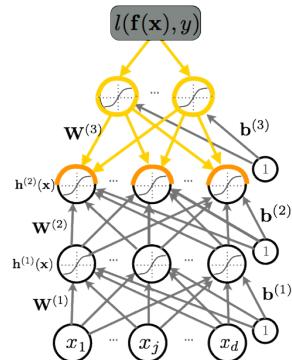
$$\begin{aligned} &\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} - \log f(\mathbf{x})_y \\ &= \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} f(\mathbf{x})_y \\ &= \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y \\ &= \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \frac{\exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} \\ &= \frac{-1}{f(\mathbf{x})_y} \left(\frac{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'}) \cdot \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})^2} - \frac{\exp(a^{(L+1)}(\mathbf{x})_y) \left(\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'}) \right)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})^2} \right) \\ &= \frac{-1}{f(\mathbf{x})_y} \left(\frac{1_{(y=c)} \exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} - \frac{\exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} \frac{\exp(a^{(L+1)}(\mathbf{x})_c)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} \right) \\ &= \frac{-1}{f(\mathbf{x})_y} \left(1_{(y=c)} \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y - \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_c \right) \\ &= \frac{-1}{f(\mathbf{x})_y} (1_{(y=c)} f(\mathbf{x})_y - f(\mathbf{x})_y f(\mathbf{x})_c) \\ &= -(1_{(y=c)} - f(\mathbf{x})_c) \end{aligned}$$

$$\frac{\partial g(x)}{\partial h(x)} = \frac{\partial g(x)}{\partial x} \frac{1}{h(x)} - \frac{g(x)}{h(x)^2} \frac{\partial h(x)}{\partial x}$$

Gradient Computation

- Loss gradient for hidden layers

- This is getting complicated!



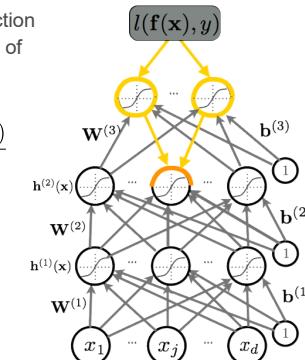
Gradient Computation

- **Chain Rule:** Assume that a function $p(a)$ can be written as a function of intermediate results $q_i(a)$, then:

$$\frac{\partial p(a)}{\partial a} = \sum_i \frac{\partial p(a)}{\partial q_i(a)} \frac{\partial q_i(a)}{\partial a}$$

- We can invoke it by setting:

- a be a hidden unit
- $q_i(a)$ be a pre-activation in the layer above
- $p(a)$ be the loss function



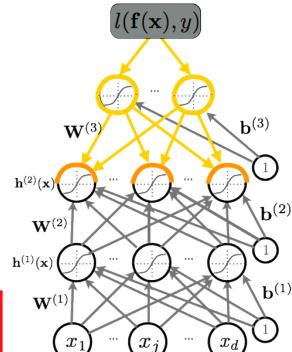
Gradient Computation

- Loss gradient at hidden layers

- Partial derivative:

$$\begin{aligned} & \frac{\partial}{\partial h^{(k)}(\mathbf{x})_j} - \log f(\mathbf{x})_y \\ &= \sum_i \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_i} \frac{\partial a^{(k+1)}(\mathbf{x})_i}{\partial h^{(k)}(\mathbf{x})_j} \\ &= \sum_i \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_i} W_{i,j}^{(k+1)} \end{aligned}$$

Remember:
 $a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$



Gradient Computation

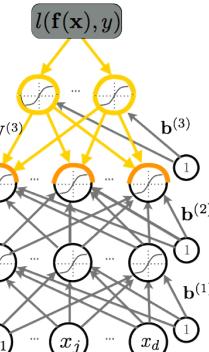
- Loss gradient at hidden layers

- Gradient

$$\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y = \mathbf{W}^{(k+1)}^\top (\nabla_{\mathbf{a}^{(k+1)}(\mathbf{x})} - \log f(\mathbf{x})_y)$$

We already know how to compute that

Remember:
 $a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$



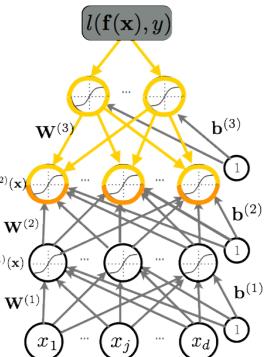
Gradient Computation

- Loss gradient at hidden layers (pre-activation)

- Partial derivative:

$$\begin{aligned} & \frac{\partial}{\partial a^{(k)}(\mathbf{x})_j} - \log f(\mathbf{x})_y \\ &= \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_j} \frac{\partial h^{(k)}(\mathbf{x})_j}{\partial a^{(k)}(\mathbf{x})_j} \\ &= \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_j} g'(a^{(k)}(\mathbf{x})_j) \end{aligned}$$

Remember:
 $h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$



Gradient Computation

- Loss gradient at hidden layers (pre-activation)

- Gradient:

$$\begin{aligned} & \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \\ &= (\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y)^T \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} \mathbf{h}^{(k)}(\mathbf{x}) \\ &= (\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y) \odot [\dots, g'(a^{(k)}(\mathbf{x})_j), \dots] \end{aligned}$$

Let's look at the gradients of activation functions.

Gradient of the activation function

Remember:
 $h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$

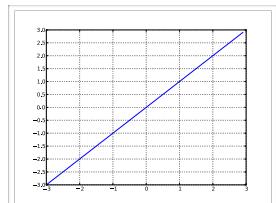
Linear Activation Function Gradient

- Linear activation function:

- Partial derivative

$$g'(a) = 1$$

$$g(a) = a$$



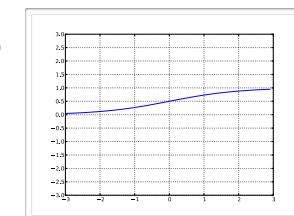
Sigmoid Activation Function Gradient

- Sigmoid activation function:

- Partial derivative

$$g'(a) = g(a)(1 - g(a))$$

$$g(a) = \text{sigm}(a) = \frac{1}{1 + \exp(-a)}$$



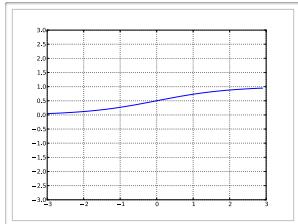
Tanh Activation Function Gradient

- Hyperbolic tangent ("tanh") activation function:

- Partial derivative

$$g(a) = \tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} = \frac{\exp(2a) - 1}{\exp(2a) + 1}$$

$$g'(a) = 1 - g(a)^2$$



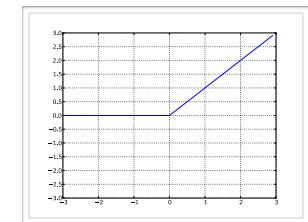
Tanh Activation Function Gradient

- Rectified linear (ReLU) activation function:

- Partial derivative

$$g'(a) = 1_{a>0}$$

$$g(a) = \text{reclin}(a) = \max(0, a)$$



Stochastic Gradient Descent

- Perform updates after seeing each example:

- Initialize: $\theta \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$

- For t=1:T

- for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\theta} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)}) - \lambda \nabla_{\theta} \Omega(\theta)$$

$$\theta \leftarrow \theta + \alpha \Delta$$

$\left. \begin{array}{l} \text{Training epoch} \\ = \\ \text{Iteration of all examples} \end{array} \right\}$

- To train a neural net, we need:

➢ Loss function: $l(f(\mathbf{x}^{(t)}; \theta), y^{(t)})$

➢ A procedure to compute gradients: $\nabla_{\theta} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)})$

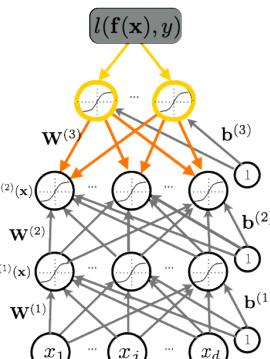
➢ Regularizer and its gradient: $\Omega(\theta), \nabla_{\theta} \Omega(\theta)$

Gradient Computation

- Loss gradient of parameters

- Partial derivative (weights):

$$\begin{aligned} & \frac{\partial}{\partial W_{i,j}^{(k)}} - \log f(\mathbf{x})_y \\ &= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} \frac{\partial a^{(k)}(\mathbf{x})_i}{\partial W_{i,j}^{(k)}} \\ &= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} h_j^{(k-1)}(\mathbf{x}) \end{aligned}$$



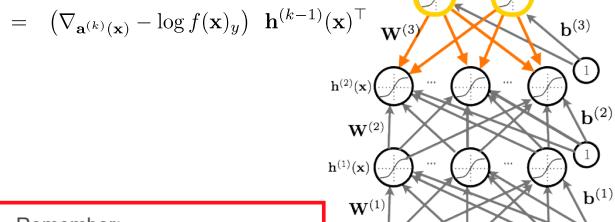
Remember:
 $a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$

Gradient Computation

- Loss gradient of parameters

- Gradient (weights):

$$\nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y$$



Remember:
 $a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$

Gradient Computation

- Loss gradient of parameters

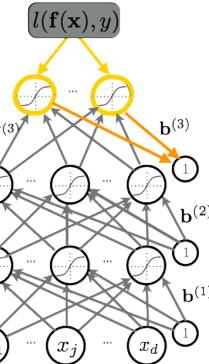
- Partial derivative (biases):

$$\frac{\partial}{\partial b_i^{(k)}} - \log f(\mathbf{x})_y$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} \frac{\partial a^{(k)}(\mathbf{x})_i}{\partial b_i^{(k)}}$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i}$$

Remember:
 $a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$



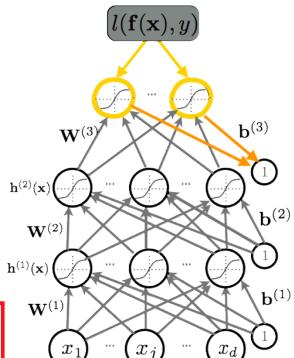
Gradient Computation

- Loss gradient of parameters

- Gradient (biases):

$$\nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y$$

$$= \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$



Remember:
 $a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$

Backpropagation Algorithm

- Perform forward propagation

- Compute output gradient (before activation):

$$\nabla_{\mathbf{a}^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y \iff -(\mathbf{e}(y) - \mathbf{f}(\mathbf{x}))$$

- For k=L+1 to 1

- Compute gradients w.r.t. the hidden layer parameters:

$$\nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y \iff (\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y) \mathbf{h}^{(k-1)}(\mathbf{x})^\top$$

$$\nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y \iff \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

- Compute gradients w.r.t. the hidden layer below:

$$\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \iff \mathbf{W}^{(k)^\top} (\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y)$$

- Compute gradients w.r.t. the hidden layer below (before activation):

$$\nabla_{\mathbf{a}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \iff (\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y) \odot [\dots, g'(a^{(k-1)}(\mathbf{x})_j), \dots]$$

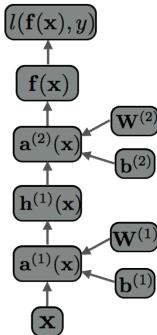
Computational Flow Graph

- Forward propagation can be represented as an acyclic flow graph

- Forward propagation can be implemented in a modular way:

➢ Each box can be an object with an **fprop method**, that computes the value of the box given its children

➢ Calling the fprop method of each box in the right order yields forward propagation



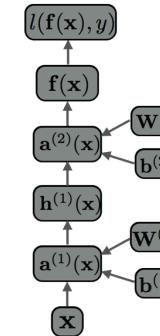
Computational Flow Graph

- Each object also has a **bprop method**

- it computes the gradient of the loss with respect to each child box.

- fprop depends on the fprop output of box's children, while bprop depends on the bprop of box's parents

- By calling bprop in the **reverse order**, we obtain backpropagation



Stochastic Gradient Descent

- Perform updates after seeing each example:

- Initialize: $\theta \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$

- For t=1:T

 - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\theta} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)}) - \lambda \nabla_{\theta} \Omega(\theta)$$

$$\theta \leftarrow \theta + \alpha \Delta$$

Training epoch
 =
 Iteration of all examples

- To train a neural net, we need:

➢ Loss function: $l(f(\mathbf{x}^{(t)}; \theta), y^{(t)})$

➢ A procedure to compute gradients: $\nabla_{\theta} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)})$

➢ Regularizer and its gradient: $\Omega(\theta), \nabla_{\theta} \Omega(\theta)$

Weight Decay

- L^2 regularization:

$$\Omega(\theta) = \sum_k \sum_i \sum_j \left(W_{i,j}^{(k)} \right)^2 = \sum_k \|\mathbf{W}^{(k)}\|_F^2$$

- Gradient:

$$\nabla_{\mathbf{W}^{(k)}} \Omega(\theta) = 2\mathbf{W}^{(k)}$$

- Only applies to weights, not biases (weight decay)

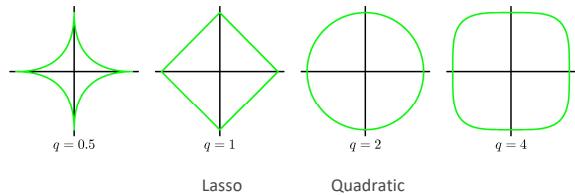
- Can be interpreted as having a Gaussian prior over the weights, while performing MAP estimation.

- We will later look at Bayesian methods.

Other Regularizers

- Using a more general regularizer, we get:

$$\frac{1}{2} \sum_{n=1}^N \{t_n - \mathbf{w}^T \phi(\mathbf{x}_n)\}^2 + \frac{\lambda}{2} \sum_{j=1}^M |w_j|^q$$



L¹ Regularization

- L¹ regularization:

$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j |W_{i,j}^{(k)}|$$

- Gradient:

$$\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = \text{sign}(\mathbf{W}^{(k)})$$

$$\text{sign}(\mathbf{W}^{(k)})_{i,j} = 1_{\mathbf{W}_{i,j}^{(k)} > 0} - 1_{\mathbf{W}_{i,j}^{(k)} < 0}$$

- Only applies to weights, not biases (weight decay)
- Can be interpreted as having a Laplace prior over the weights, while performing MAP estimation.
- Unlike L2, L1 will push some weights to be exactly 0.

Initialization

- Initialize biases to 0
- For weights
 - Can not initialize weights to 0 with tanh activation
 - All gradients would be zero (saddle point)
 - Can not initialize all weights to the same value
 - All hidden units in a layer will always behave the same
 - Need to break symmetry
 - Sample $\mathbf{W}_{i,j}^{(k)}$ from $U[-b, b]$, where

$$b = \frac{\sqrt{6}}{\sqrt{H_k + H_{k-1}}}$$

Sample around 0 and
break symmetry

Size of $\mathbf{h}^{(k)}(\mathbf{x})$

Model Selection

- Training Protocol:
 - Train your model on the **Training Set** $\mathcal{D}^{\text{train}}$
 - For model selection, use **Validation Set** $\mathcal{D}^{\text{valid}}$
 - Hyper-parameter search: hidden layer size, learning rate, number of iterations/epochs, etc.
 - Estimate generalization performance using the **Test Set** $\mathcal{D}^{\text{test}}$
- Remember: Generalization is the behavior of the model on **unseen examples**.

Early Stopping

- To select the number of epochs, stop training when validation set error increases (with some look ahead).



Tricks of the Trade:

- Normalizing your (real-valued) data:
 - for each dimension x_i , subtract its training set mean
 - divide each dimension x_i by its training set standard deviation
 - this can speed up training
- Decreasing the learning rate: As we get closer to the optimum, take smaller update steps:
 - start with large learning rate (e.g. 0.1)
 - maintain until validation error stops improving
 - divide learning rate by 2 and go back to (ii)

Gradient Checking

- To debug your implementation of fprop/bprop, you can compare with a finite-difference approximation of the gradient:

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

- $f(x)$ would be the loss
- x would be a parameter
- $f(x + \epsilon)$ would be the loss if you add ϵ to the parameter
- $f(x - \epsilon)$ would be the loss if you subtract ϵ to the parameter

Debugging on Small Dataset

- If not, investigate the following situations:
 - Are some of the units **saturated**, even before the first update?
 - scale down the initialization of your parameters for these units
 - properly normalize the inputs
 - Is the training error bouncing up and down?
 - decrease the learning rate
- This does not mean that you have computed gradients correctly:
 - You could still overfit with some of the gradients being wrong

Discussion of Optimization Techniques

Vahid Tarokh
ECE 685D, Fall 2025

Introduction

- We will quickly review optimization algorithms and SGD.
- Discuss more why they are preferred more than batch methods
- Discuss mathematical analysis for (and applications to) SGD
- Discuss extensions/variants (RMSprop, ADAGrad, ADAM)
- Important Note: Source of some of my slides (with great appreciation and acknowledgements)
 - Professor David Carlson Slides
 - Professor Ruslan Salakhutdinov's slides (available online).

Optimization Goal

- Have some model or network parameterized by \mathbf{w}
- Goal: given data, find the best \mathbf{w}
- What is the best \mathbf{w} ?
 - ▶ Gives the best prediction (or other metric) on the *true* task
 - ▶ *True* task refers to future, unseen data (i.e. real-world performance)
 - ▶ Quick reminder: often estimate performance with a test set
- Many examples shown so far:
 - ▶ Image recognition
 - ▶ Object detection
 - ▶ Text classification
 - ▶ Etc.

Optimization Goal

- In optimization, this amounts to solving for \mathbf{w}^*
$$\mathbf{w}^* = \arg \min_{\mathbf{W}} \mathbb{E}_{p_{\text{true}}(\mathbf{x}, y)} [\ell(h_{\mathbf{W}}(\mathbf{x}), y)]$$
- $\ell(\cdot, \cdot)$ is the “loss,” can be defined in many ways. Some examples:
 - ▶ squared loss: $\ell(\hat{y}, y) = \|\hat{y} - y\|_2^2$
 - ▶ (binary) cross-entropy loss: $\ell(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$
 - ▶ Negative Log-Likelihood: $\ell(h_{\mathbf{W}}(\mathbf{x}), y) = -\log p_{\mathbf{W}}(\mathbf{x}, y)$
- $h_{\mathbf{W}}(\mathbf{x})$ defines a transformation from the data to the output space – only function that changes when the parameters do (e.g. in logistic regression $h_{\mathbf{W}}(\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x})$)
- $p_{\text{true}}(\mathbf{x}, y)$ is the probability distribution over data (unknown!)

Empirical Risk Minimization

$$\mathbf{w}^* = \arg \min_{\mathbf{w}} \mathbb{E}_{p_{true}(\mathbf{x}, y)} [\ell(h_{\mathbf{w}}(\mathbf{x}), y)]$$

- **Issue:** The probably distribution $p_{true}(\mathbf{x}, y)$ is unknown!
- But we have N data examples $\mathcal{D} = \{\mathbf{x}_n, y_n\} \sim p_{true}(\mathbf{x}, y)$
- Will approximate the above with finite data examples (i.e. “Empirical Risk Minimization” (ERM))

$$\min_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^N \ell(h_{\mathbf{w}}(\mathbf{x}_n), y_n)$$

- ▶ Why does this matter?

Why does using finite data matter?

- Using empirical samples is biased, leads to **overfitting**

Definition (Overfitting)

Overfitting occurs when the learned parameters \mathbf{w} capture random error or noise. This implies the parameters \mathbf{w} are “learning” the noise rather than properties of the data. Mathematically:

$$\mathbb{E}_{p(\mathbf{x}, y)} [\ell(h_{\mathbf{w}}(\mathbf{x}), y)] > \frac{1}{N} \sum_{n=1}^N \ell(h_{\mathbf{w}}(\mathbf{x}_n), y_n)$$

or,

“generalization error” > “training error”

Consequences for Iterative Optimizers

- No benefit to exact optimization, only need “**moderate**” accuracy as fast as possible
- *modus operandi* in big data: use stochastic iterative methods
 - ▶ Less information per iteration, but many many more iterations in the same amount of time
- We assume $\mathbf{w} \in \mathbb{R}^D$, but can easily consider constrained set (most important thing is dimensionality D)
- The above optimization goal is often rewritten for simplicity:

$$\min_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^N f_n(\mathbf{w}), \quad f_n(\mathbf{w}) = \ell(h_{\mathbf{w}}(\mathbf{x}_n), y_n)$$

Binary Logistic Regression

- A canonical model for classification is logistic regression:

$$P(y = 1 | \mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x}), \quad \sigma(z) = \frac{1}{1 + \exp(-z)}$$

- Why is this a good example for optimization?

- ▶ This gives our logistic loss function
- ▶ Can simply derive constants used in convergence analysis (allowing theorems to have *precise* values)

Introduction and Intuition of Stochastic Gradients

(Stochastic) Gradient Descent:

Gradient Descent (GD)

With step size α_k , use updates:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \left[\frac{1}{N} \sum_{i=1}^N \nabla f_i(\mathbf{w}_k) \right]$$

Stochastic Gradient Descent (SGD)

With step size α_k and random index i_k , use updates:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k [\nabla f_{i_k}(\mathbf{w}_k)]$$

Note that $\mathbb{E}_{p(i_k)} [\nabla f_{i_k}(\mathbf{w}_k)] = \left[\frac{1}{N} \sum_{i=1}^N \nabla f_i(\mathbf{w}_k) \right]$ (unbiased). (Underlying i.i.d assumption in our dataset, \mathcal{D})

How expensive are these iterative algorithms?

- Gradient descent takes:
 - $\mathcal{O}(N)$ to estimate gradients
 - $\mathcal{O}(1)$ to update the parameters
- Stochastic gradient descent takes:
 - $\mathcal{O}(1)$ to estimate gradients
 - $\mathcal{O}(1)$ to update the parameters
- What about Newton's method (classical optimization approach)?
 - $\mathcal{O}(N)$ to estimate gradients and $\mathcal{O}(ND^2)$ to estimate Hessian
 - $\mathcal{O}(D^3)$ to update the parameters
- Consider implication for GoogLeNet image classifier on the ImageNet dataset, with $N = 10^6$ and $D \simeq 5 \times 10^6$

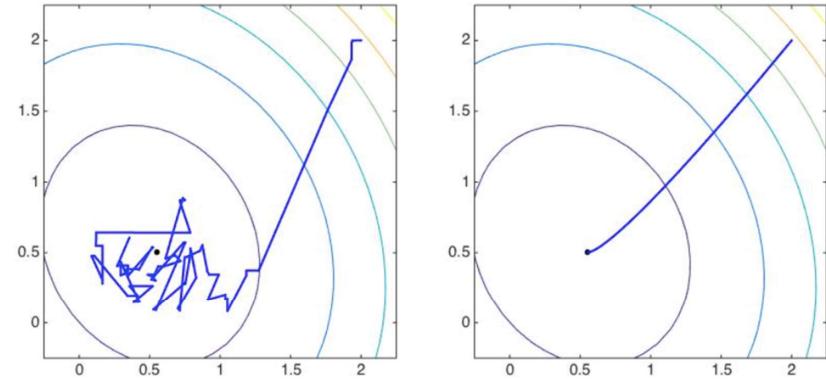


Figure: (Left) Stochastic gradient descent with a fixed step size. (Right) Batch gradient descent.

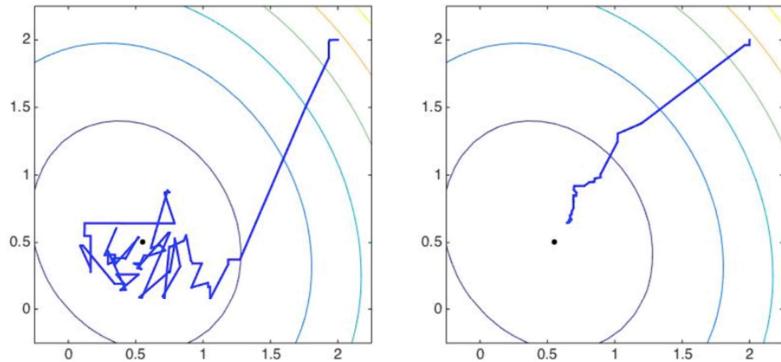


Figure: (Left) Stochastic gradient descent with a fixed step size. (Right) Stochastic gradient descent with diminishing step size.

Necessary Definitions for Convergence Analysis

Definition (Lipschitz-continuous functions)

Given an objective function $F : \mathbb{R}^D \rightarrow \mathbb{R}$, the function is Lipschitz continuous if

$$|F(\mathbf{y}) - F(\mathbf{x})| \leq L_0 \|\mathbf{y} - \mathbf{x}\|_2$$

When will Stochastic Gradient Descent work?

Necessary Definitions for Convergence Analysis

Definition (Lipschitz-continuous objective gradients)

Given an objective function $F : \mathbb{R}^D \rightarrow \mathbb{R}$ is continuously differentiable, the gradient of F is Lipschitz continuous with Lipschitz constant $L > 0$ if

$$\|\nabla F(\mathbf{y}) - \nabla F(\mathbf{x})\|_2 \leq L \|\mathbf{y} - \mathbf{x}\|_2.$$

For continuous second gradients, note $L \geq \max_{\mathbf{x}} \|\nabla^2 F(\mathbf{x})\|_{S_\infty}$

- The matrix $\|\cdot\|_{S_\infty}$ norm is defined as the largest singular value of the matrix (also known as the spectral norm, the Schatten- ∞ norm, and the matrix 2-norm):

$$\|\mathbf{A}\|_{S_\infty} = \max_{\|\mathbf{x}\|_2 \leq 1} \|\mathbf{Ax}\|_2$$

Lipschitz Gradient for Logistic Regression

- Remember the cost function in logistic regression:

$$\min_{\mathbf{w}} F(\mathbf{w}) = \min_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^N -y_n \log(\sigma(\mathbf{w}^T \mathbf{x})) - (1 - y_n) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}))$$

- Hessian is given by

$$\nabla^2 F(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N (\sigma(\mathbf{w}^T \mathbf{x}_n)(1 - \sigma(\mathbf{w}^T \mathbf{x}_n))) \mathbf{x}_n \mathbf{x}_n^T$$

Lipschitz Gradient for Logistic Regression

- First term is bound by:

$$0 \leq (\sigma(\mathbf{w}^T \mathbf{x})(1 - \sigma(\mathbf{w}^T \mathbf{x}))) \leq \frac{1}{4}$$

- Linear algebra allows us to state:

$$\mathbf{0} \preceq \nabla^2 F(\mathbf{w}) \preceq \frac{1}{4N} \sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^T$$

($\mathbf{A} \preceq \mathbf{B}$ means that $\mathbf{B} - \mathbf{A}$ is a positive semidefinite matrix, e.g. $\mathbf{x}^T \mathbf{A} \mathbf{x} \leq \mathbf{x}^T \mathbf{B} \mathbf{x}$ for any \mathbf{x} if matrices are symmetric)

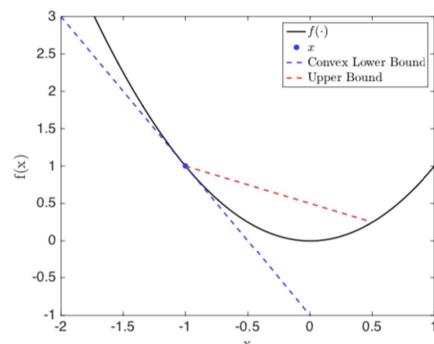
- A simple bound is $L \geq \frac{1}{4} \mathbb{E}_{\mathcal{D}}[\|\mathbf{x}\|_2^2]$

Convexity

- A convex function has two definitions:

$$\begin{aligned} \forall \mathbf{x}, \mathbf{y} \in \mathcal{X}, t \in [0, 1] \\ f(t\mathbf{y} + (1-t)\mathbf{x}) \leq tf(\mathbf{y}) + (1-t)f(\mathbf{x}) \\ f(\mathbf{y}) \geq f(\mathbf{x}) + (\nabla f(\mathbf{x}))^T (\mathbf{y} - \mathbf{x}) \end{aligned}$$

- First definition is more general, gradient doesn't always exist



Strong Convexity

Definition (Strong Convexity):

An objective function $F : \mathbb{R}^d \rightarrow \mathbb{R}$ is strongly convex in that there is a constant $c > 0$ such that

$$F(\mathbf{y}) \geq F(\mathbf{x}) + \nabla F(\mathbf{x})^T (\mathbf{y} - \mathbf{x}) + \frac{1}{2} c \|\mathbf{y} - \mathbf{x}\|_2^2$$

Then F will have a unique minimizer \mathbf{x}_* with function value F_* . If only $c = 0$ holds, then the function is convex.

- This definition covers a significant subset of machine learning problems (e.g. penalized logistic regression, etc.)
- Does not cover models such as LDA, PFA, RBM, SBN, MLP, CNN, RNN, etc.
 - Gives clear theoretical results, and is very useful for intuitions in these problems

Implications

- Convex function implies a local minima is a global minima
- The gradient at the optima (if gradient exists)

$$\nabla f(\mathbf{x}) = \mathbf{0}$$

- Most state-of-the-art machine learning models are **not** convex
- However, very useful for intuition – nonconvex functions are often convex around a local minima

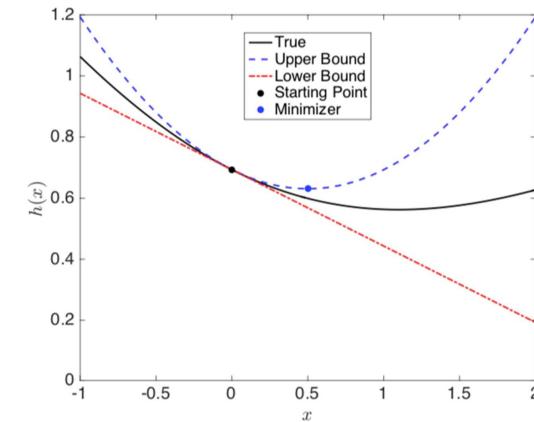
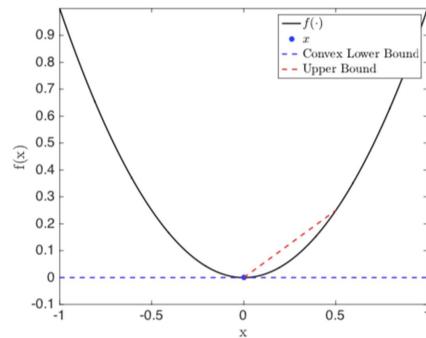


Figure: Examples of the upper bound on logistic loss. The Lipschitz gradient provides a conservative upper bound on the function value.

Consequences for Gradient Methods

Lemma (Decreasing sequence for gradient descent)

Consider a function F with Lipschitz gradient with constant L . For the deterministic (i.e. batch) gradient descent method with iterates

$$\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \alpha \nabla F(\mathbf{w}_k),$$

then the Lipschitz gradient upper bound with this sequence yields

$$\begin{aligned} F(\mathbf{w}_{k+1}) &\leq F(\mathbf{w}_k) - [\nabla F(\mathbf{x})]^T (\alpha \nabla F(\mathbf{w}_k)) + \frac{1}{2} L \|\alpha \nabla F(\mathbf{w}_k)\|_2^2, \\ &\leq F(\mathbf{w}_k) - \alpha \|\nabla F(\mathbf{w}_k)\|_2^2 + \frac{\alpha^2 L}{2} \|\nabla F(\mathbf{w}_k)\|_2^2, \\ &\leq F(\mathbf{w}_k) + \left(\frac{L\alpha^2}{2} - \alpha \right) \|\nabla F(\mathbf{w}_k)\|_2^2. \end{aligned}$$

“Optimal” Step Size

Lemma (Optimal step size for gradient descent)

Consider the upper bound for gradient descent

$$F(\mathbf{w}_{k+1}) \leq F(\mathbf{w}_k) + \left(\frac{L\alpha^2}{2} - \alpha \right) \|\nabla F(\mathbf{w}_k)\|_2^2.$$

The RHS (optimal guaranteed improvement) is minimized at $\alpha = \frac{1}{L}$,

$$F(\mathbf{w}_{k+1}) \leq F(\mathbf{w}_k) - \frac{1}{2L} \|\nabla F(\mathbf{w}_k)\|_2^2.$$

What are the implications of the upper bound?

- If the gradient is nonzero, the next iteration will have a lower function value (sequence is non-increasing)
- If the function has a minimum F^* , will converge to a fixed point (i.e. $\nabla F(\mathbf{w}) = \mathbf{0}$)

Implications

Algorithm (Minibatch Gradient Estimator)

Inputs: Data $\{\mathbf{x}_n, y_n\}_{n=1,\dots,N}$, minibatch size B , parameters \mathbf{w}_k
 Sample B minibatch indices $\{i_1, \dots, i_B\}$
 Return gradient estimate: $\tilde{\mathbf{g}}_k \leftarrow \frac{1}{B} \sum_{m=1}^B \nabla f_{i_m}(\mathbf{w}_k)$

Lemma (Sequence for stochastic gradient)

Consider a function F with Lipschitz gradient with constant L . For the minibatch gradient descent method with iterates

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \tilde{\mathbf{g}}_k,$$

then the sequence satisfies in expectation

$$\mathbb{E}[F(\mathbf{w}_{k+1})] - F(\mathbf{w}_k) \leq -\alpha_k [\nabla F(\mathbf{w}_k)]^T \mathbb{E}[\tilde{\mathbf{g}}_k] + \frac{\alpha_k^2 L}{2} \mathbb{E}[\|\tilde{\mathbf{g}}_k\|_2^2]$$

Consequences for an unbiased estimator

- First, we consider an unbiased gradient estimator (i.e. $\mathbb{E}[\tilde{\mathbf{g}}_k] = \mathbf{g}_k$)
- Assume the variance is bounded (i.e. $\text{var}(\|\tilde{\mathbf{g}}_k\|_2) \leq M$)
- Then the previous lemma reveals that

$$\mathbb{E}[F(\mathbf{w}_{k+1})] - F(\mathbf{w}_k) \leq \left(\frac{\alpha_k^2 L}{2} - \alpha_k \right) \|\mathbf{g}_k\|_2^2 + \frac{\alpha_k^2 L M}{2}$$

- Only difference to deterministic case is the term $\frac{\alpha_k^2 L M}{2}$
- Question: When is a gradient step expected to improve our cost function?

Consequences for an unbiased estimator

- Consider a step size $\alpha_k = \frac{1}{L}$

$$\mathbb{E}[F(\mathbf{w}_{k+1})] - F(\mathbf{w}_k) \leq \frac{1}{2L} (M - \|\mathbf{g}_k\|_2^2)$$

- Whether the function is improved depends on the quantity $M - \|\mathbf{g}_k\|_2^2$
 - Often, M is fairly constant, but $\|\mathbf{g}_k\|_2^2 \rightarrow 0$ at the optimum
- Smaller step sizes allow for further optimization
 - Smaller step sizes take (much) longer if not necessary!

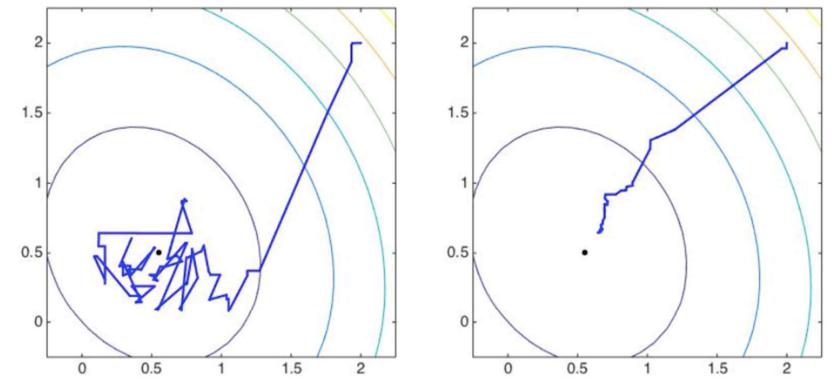


Figure: (Left) Stochastic gradient descent with a fixed step size. (Right) Stochastic gradient descent with diminishing step size.

Summary/Recap

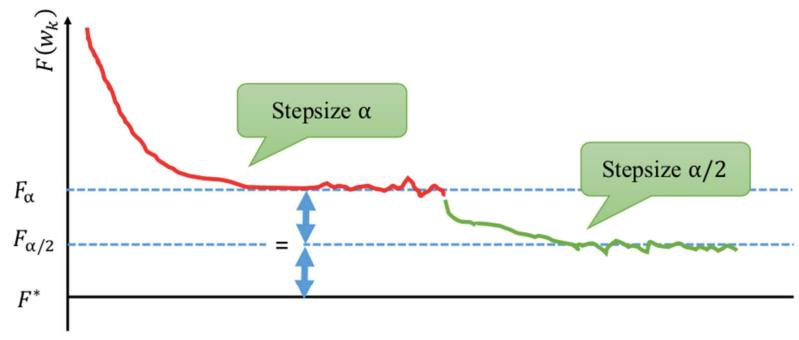


Figure: Reducing the step size allows the objective to reach closer to the optimal value.

- Rigorous analysis depends on constants that can be derived for many standard models
 - ▶ Hard to derive for deep neural networks, but intuition is important for understanding how these methods work
- The variance in the gradient estimator limits how well the function can be optimized
- Reducing the step size *or* increasing the minibatch size allows the optimization algorithm to reach closer to the optimum

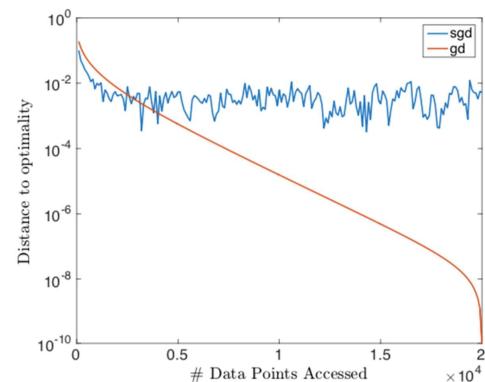


Figure: In the beginning, SGD has an exponential decay, but on small datasets GD will catch up and pass SGD.

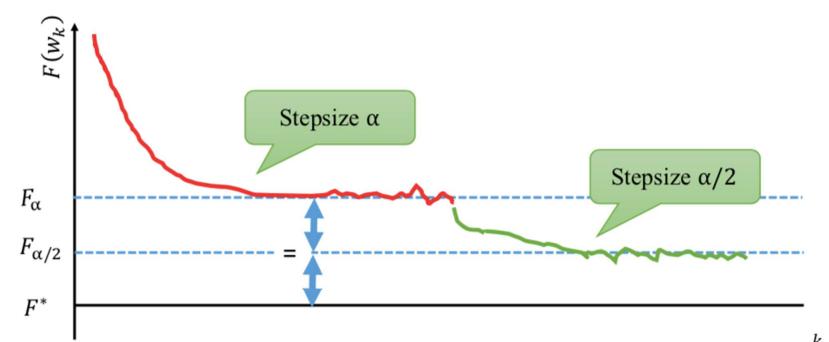


Figure: Reducing the step size allows the objective to reach closer to the optimal value.

SGD with diminishing step sizes

Algorithm (Minibatch Gradient Descent)

```

Inputs:  $\beta, \gamma, \kappa, \mathbf{w}_0$ 
Initialize:  $k \leftarrow 0$ 
for  $k=0, \dots$  do
    Estimate gradient:  $\tilde{\mathbf{g}}_k$ 
    Calculate step size:  $\alpha_k = \beta(\gamma + k)^{-\kappa}$ 
    Update parameters:  $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \alpha_k \tilde{\mathbf{g}}_k$ 
end for

```

Theorem (Convergence of minibatch gradient descent)

For $\kappa = 1$, $\beta \geq \frac{1}{c}$ and $\gamma \geq \beta L$, the expected optimality gap satisfies

$$\mathbb{E}[F(\mathbf{w}_k) - F_*] \leq \frac{\nu}{\gamma + k}, \quad \nu = \max \left\{ (\gamma + 1)(F(\mathbf{w}_1) - F_*), \frac{\beta^2 LM}{2(\beta c - 1)} \right\}$$

- F is assumed to be a strongly convex function with constant c and Lipschitz gradient with constant L

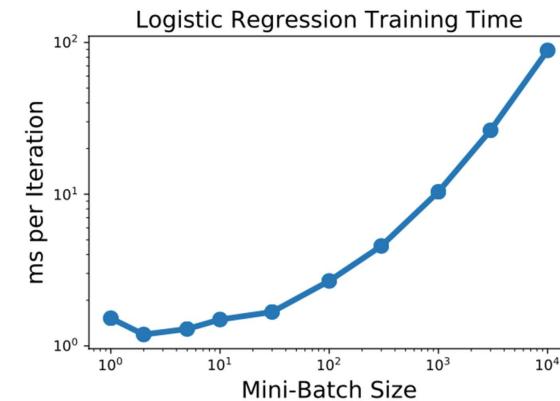
Thoughts and conclusions up to this point in time

- Practically, SGD can dominate batch methods to reasonable accuracy
- The limitations of the stochastic gradient method are:
 - ▶ Constants (and hence step sizes) are usually unknown
 - ▶ Variance in the gradient estimation limits convergence (much of the recent literature and our forthcoming discussion will discuss ways of minimizing variance)
 - ▶ Does not utilize the curvature in space, can be very poorly conditioned.
- So far, we assumed the gradient estimates were unbiased
 - ▶ Often gradients depend on a sequential procedure (RBMs, PFA, variational methods, recurrent neural nets) – gradients are not necessarily unbiased

Improving Accuracy

- One of the fundamental quantities that control the convergence of stochastic gradient is the variance M
- Complex approach: develop algorithmic methods to reduce variance (included in slides, will not get to today)
- Simple approach: increase minibatch size. Typically $M \propto 1/(\text{minibatch size})$
- Problem: Time to estimate the gradient is \propto (minibatch size)
- No theoretical benefit to using minibatch over a single example – in fact, theoretical convergence rate is typically worse.

Empirical Minibatch Timing



Nesterov's Accelerated Gradient Descent:

- Nesterov methods invented originally by Yurii Nesterov is used for acceleration of the first order methods.
- Remember that function F with **only Lipchitz gradient** assumption has sublinear rate $\mathcal{O}(\frac{1}{k})$ convergence.
- Nesterov method establishes a better rate like $\mathcal{O}(\frac{1}{k^2})$.
- In addition to the gradient information from ***the previous iteration***, gradients from other iteration(s) are contributed with appropriate weights to the calculation of the current estimate of the parameter.

Nestrov Acceleration Methods

Nesterov's Accelerated Gradient Descent:

- First define the following sequences:

$$\lambda_0 = 0, \quad \lambda_k = \frac{1 + \sqrt{1 + 4\lambda_{k-1}^2}}{2}, \quad \gamma_k = \frac{1 - \lambda_k}{\lambda_{k+1}}$$

- For $k = 1 \dots$ do (for some initial point $w_1 = t_1$):

$$t_{k+1} = w_k - \frac{1}{\beta} \nabla F(w_k),$$

$$w_{k+1} = (1 - \gamma_k)t_{k+1} + \gamma_k t_k,$$

- The update involves computing of gradient in time steps k and $k + 1$ (using momentum)

Theorem (Nesterov 1983). Let F be a convex and β -smooth function, then the Nesterov's Accelerated Gradient Descent satisfies for all $k > 1$:

$$F(w_k) - F(w_*) \leq \frac{2\beta||w_1 - w_*||_2^2}{k^2}$$

Including Momentum in SGD

Improving Stochastic Methods with Momentum (Acceleration methods)

Algorithm (Minibatch Gradient Descent with Momentum)

```

Inputs:  $\alpha, \beta$ 
Initialize:  $k \leftarrow 0, \mathbf{m}_k$ 
for  $k=0, \dots$  do
    Estimate gradient:  $\tilde{\mathbf{g}}_k$ 
    Update with momentum:  $\mathbf{m}_{k+1} \leftarrow \beta \mathbf{m}_k + \tilde{\mathbf{g}}_k$ 
    Update parameters:  $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \alpha \mathbf{m}_{k+1}$ 
end for

```

- Classically, momentum “dampens” the highly oscillatory terms
- Analysis reduces dependency of $\frac{L}{c}$ (a worst-case condition number of the Hessian) to $\sqrt{\frac{L}{c}}$ for a strongly convex model.

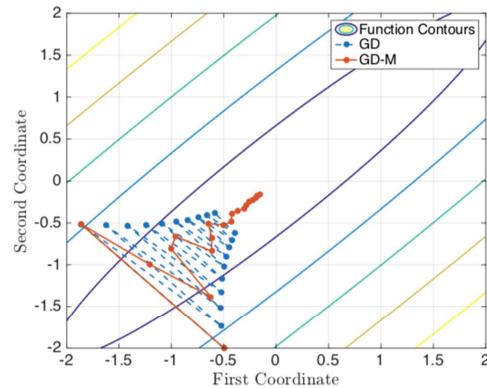


Figure: Effect of using momentum for a skewed 2D Gaussian. Gradient descent bounces back and forth, but using momentum averages out the first dimension and finds the correct path.

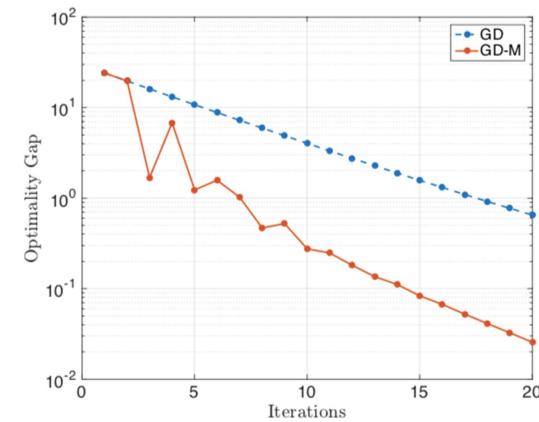


Figure: Effect of using momentum for a skewed 2D Gaussian in the previous figure. Using momentum greatly improves the convergence speed.

Two Viewpoints on Momentum

- Define momentum updates as exponential smoothing

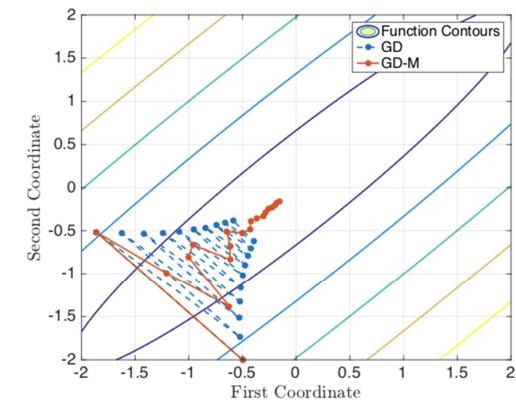
$$\bar{\mathbf{g}} = (1 - \beta) \sum_{i=0}^k \beta^i \tilde{\mathbf{g}}_{k-i}.$$

- Note: $\sum_{i=0}^{\infty} \beta^i = (1 - \beta)^{-1}$
- Then SGD with momentum is implemented with the update

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \bar{\mathbf{g}}.$$

- This can be viewed as:
 - A filter that dampens out transient patterns
 - A bias-variance tradeoff (decrease variance for a small bias) on noisy gradient estimates

Dampening of Transient Patterns



Momentum as De-noising Gradient Estimates

- Examining the term

$$\bar{\mathbf{g}} = (1 - \beta) \sum_{i=0}^k \beta^i \tilde{\mathbf{g}}_{k-i}.$$

- If $\mathbb{E}[\tilde{\mathbf{g}}_k] = \mathbf{g}_k$ and $\text{var}(\|\mathbf{g}_k\|_2) \leq M$, then as $k \rightarrow \infty$

$$\text{var}(\|\bar{\mathbf{g}}_k\|_2) = (1 - \beta)^2 \frac{1}{1 - \beta^2} M = \frac{1 - \beta}{1 + \beta} M.$$

- Decreases variance by a multiplicative factor of $\frac{1 - \beta}{1 + \beta}$
- Downside: introduces bias (i.e. $\mathbb{E}[\bar{\mathbf{g}}_k] \neq \mathbf{g}_k$)

Higher Order Methods

Higher Order Methods

- SGD only uses gradient (first-order) information
- Sometimes the *curvature* is drastically different depending on the direction
 - Recall the momentum example!
- Can change the direction based on curvature
- step size tuning can pose problems, can use curvature to estimate an appropriate step size

Second Order Methods

Definition (Second order approximation)

Consider a second order expansion:

$$q_k(\mathbf{w}) = F(\mathbf{w}_k) + \nabla F(\mathbf{w}_k)^T(\mathbf{w} - \mathbf{w}_k) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_k)^T \hat{B}^{-1}(\mathbf{w} - \mathbf{w}_k).$$

Move \mathbf{w} in the direction that minimizes q_k :

$$\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \alpha_k \hat{B} \nabla F(\mathbf{w}_k).$$

Note: $\alpha_k = 1$ will minimize $q_k(\mathbf{w})$.

Newton's Methods

Algorithm (Newton's method)

Let $\hat{B}^{-2} = \nabla^2 F(\mathbf{w}_k)$ (i.e. the Hessian). Then

$$\begin{aligned}\mathbf{w}_{k+1} &= \arg \min_{\mathbf{w}} q_k(\mathbf{w}) \\ &= \mathbf{w}_k - [\nabla^2 F(\mathbf{w}_k)]^{-1} \nabla F(\mathbf{w}_k)\end{aligned}$$

If certain conditions are satisfied, this approach converges with quadratic convergence $\mathcal{O}(\rho^{k^2})$.

- Convergence is great, but many issues:
 - Gradient typically costs $\mathcal{O}(ND)$
 - Forming the Hessian typically costs $\mathcal{O}(ND^2)$
 - Applying the Hessian to the gradient typically costs $\mathcal{O}(D^3)$
 - Often assumptions are not satisfied, may diverge!
- Can often get a "good enough" solution from SGD by the time Newton runs a single iteration for big data and big models

Practical Approaches

- A large neural network may have $>> 1$ million parameters
 - Neither feasible to calculate Hessian, nor apply it
- Still want to be able to use curvature information
- Common approach is to use a diagonal approximation

$$q_k(\mathbf{w}) = F(\mathbf{w}_k) + \nabla F(\mathbf{w}_k)^T(\mathbf{w} - \mathbf{w}_k) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_k)^T \text{diag}(\mathbf{b})^{-1}(\mathbf{w} - \mathbf{w}_k)$$

with the minimizer with $\alpha_k = 1$ as

$$\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \alpha_k \mathbf{b} \odot \nabla F(\mathbf{w}_k)$$

- Several approaches to developing \mathbf{b}

Consequences of diagonal approximations

- The follow algorithms *do not* improve the convergence rate
- They can improve optimizations constants (provably on regret)
 - ▶ The constants may be dramatically improved
- Biggest reason to use the following methods is their *robustness* to settings
 - ▶ SGD methods are very sensitive to the step size sequences
 - ▶ Many of the so-called “adaptive metric” methods can use the same settings across many models and datasets

Adagrad Algorithm

Algorithm (ADAgrad)

```

Inputs:  $\epsilon, \gamma, \mathbf{w}_0$ 
Initialize:  $k \leftarrow 0, \mathbf{v}_0 \leftarrow \mathbf{0}$ 
for  $k=0, \dots$  do
    Estimate gradient:  $\tilde{\mathbf{g}}_k$ 
    Update sum-of-squares:  $\mathbf{v}_{k+1} \leftarrow \mathbf{v}_k + \tilde{\mathbf{g}}_k \odot \tilde{\mathbf{g}}_k$ 
    Calculate element-wise step sizes:  $\alpha_k = \gamma \mathbf{1} \oslash (\epsilon + \sqrt{\mathbf{v}_{k+1}})$ 
    Update parameters:  $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \alpha_k \odot \tilde{\mathbf{g}}_k$ 
end for

```

- Key idea: sparsely occurring features may be *very* informative – only decrease step size when information relating to that parameter is seen
- *Provably* improves regret bounds compared to online (stochastic) gradient descent

RMSprop Algorithm

Algorithm (RMSprop)

```

Inputs:  $\epsilon, \gamma, \beta, \mathbf{w}_0, \alpha_k$ 
Initialize:  $k \leftarrow 0, \mathbf{v}_0 \leftarrow \mathbf{0}$ 
for  $k=0, \dots$  do
    Estimate gradient:  $\tilde{\mathbf{g}}_k$ 
    Update sum-of-squares:  $\mathbf{v}_{k+1} \leftarrow (1 - \beta)\mathbf{v}_k + \beta(\tilde{\mathbf{g}}_k \odot \tilde{\mathbf{g}}_k)$ 
    Calculate element-wise preconditioner:  $\mathbf{b}_k = \gamma \mathbf{1} \oslash (\epsilon + \sqrt{\mathbf{v}_{k+1}})$ 
    Update parameters:  $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \alpha_k \mathbf{b}_k \odot \tilde{\mathbf{g}}_k$ 
end for

```

- Will converge if $\alpha \rightarrow 0$ with at rate $t^{\frac{1}{2}}$
- Typically a constant step size is used $\alpha_k = \bar{\alpha} \simeq 10^{-3}$
- In practice, has been largely replaced the ADAM optimizer

Comments on Adagrad

- ADAgrad is a great algorithm when features are sparse
- Step sizes can diminish much too quickly (no forgetting—once an element-wise step size goes small, it stays small)
- Intuitive to add a “forgetting” term, especially in deep learning

Comments on RMSprop

- Ad-hoc: originally no convergence guarantee
- No inclusion of momentum
- Adaptive Moments (ADAM) addressed these issues
- RMSprop had significant practical successes, but is less used now

ADAM Algorithm

Algorithm (ADAM)

```

Inputs:  $\alpha_1, \dots, T, \beta_1, \beta_2, \mathbf{w}_0$ 
Initialize:  $k \leftarrow 0, \mathbf{m}_0^{(1)} \leftarrow \mathbf{0}, \mathbf{m}_0^{(2)} \leftarrow \mathbf{0}$ 
for  $k=0, \dots$  do
    Estimate gradient:  $\tilde{\mathbf{g}}_k$ 
    Update first moment:  $\mathbf{m}_{k+1}^{(1)} \leftarrow \beta_1 \mathbf{m}_k^{(1)} + (1 - \beta_1) \tilde{\mathbf{g}}_k$ 
    "Debias" first moment  $\tilde{\mathbf{m}}_{k+1}^{(1)} \leftarrow \mathbf{m}_{k+1}^{(1)} (1 - \beta_1^{(k+1)})^{-1}$ 
    Update second moment:  $\mathbf{m}_{k+1}^{(2)} \leftarrow \beta_2 \mathbf{m}_k^{(2)} + (1 - \beta_2) \tilde{\mathbf{g}}_k \odot \tilde{\mathbf{g}}_k$ 
    "Debias" second moment  $\tilde{\mathbf{m}}_{k+1}^{(2)} \leftarrow \mathbf{m}_{k+1}^{(2)} (1 - \beta_2^{(k+1)})^{-1}$ 
    Update parameters:  $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \alpha_t(\tilde{\mathbf{m}}_{k+1}^{(1)}) \oslash \left( \epsilon + \sqrt{\tilde{\mathbf{m}}_{k+1}^{(2)}} \right)$ 
end for

```

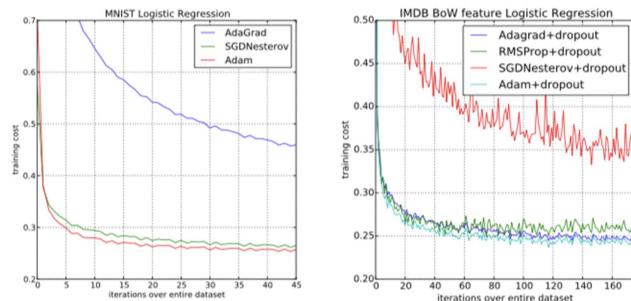


Figure: Comparison of several learning rules on large-scale logistic regression.

Analysis of ADAM Algorithm

- RMSprop inspired ADAM – if $\beta_1 = 0$ then they are very similar algorithms
- Provable regret bound under a decreasing step size
- Can typically use “standard” parameters for many problems, e.g.

$$\alpha_k = \bar{\alpha} = 10^{-3}, \beta_1 = .9, \beta_2 = .999$$

- Reminder: β_1 going higher reduces variance of gradients, but can add bias

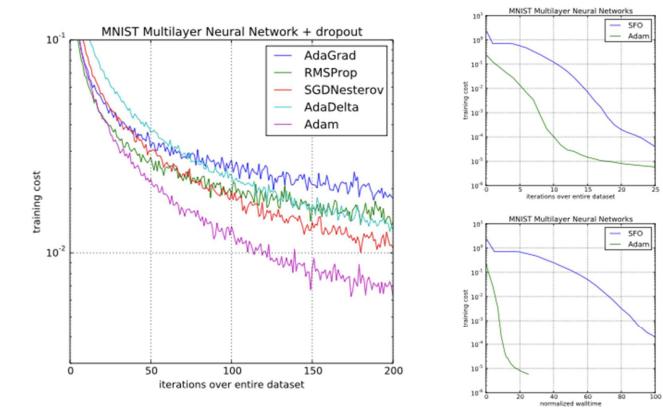


Figure: Comparison of several learning rules on deep neural networks.

Conclusions

- Stochastic gradient descent will converge to the optimum if given enough time
 - Will converge to a fixed point in nonconvex problems as well
- Adding in adaptive higher order information can reduce parameter tuning and account for curvature information
- Stochastic gradient methods are effective at quickly obtaining a reasonable solution
- Machines are learned with stochastic gradient methods
- Many theoretical issues but many practical successes
- Still a very active area of research!
- Many tuning parameters within these algorithms
 - Hopefully their meaning and how to set them is clearer now

Convolutional Neural Networks and Applications to Object Classification

Vahid Tarokh

ECE685D, Fall 2025

Introduction

• We next focus on convolutional neural networks. These have been extremely successful in image classification algorithms.

• **Important Note: Source of some of my slides (with great appreciation and acknowledgements):**

- Dive into Deep Learning
- Professor David Carlson's Slides
- Professor Hugo Larochelle's slides
- Professor Ruslan Salakhutdinov's slides (available online)
 - Some tutorial slides were borrowed from Rob Fergus
<https://sites.google.com/site/deeplearningsummerschool2016/speakers>
 - Marc'Aurelio Ranzato's CVPR 2014 tutorial on Convolutional Nets
<https://sites.google.com/site/lsvrtutorialcvpr14/home/deeplearning>
- Much of the material in this lecture was borrowed from class on NN:
<https://sites.google.com/site/deeplearningsummerschool2016>

Computer Vision

• Design algorithms that can process visual data to accomplish a given task:

- For example, **object recognition**: Given an input image, identify which object it contains

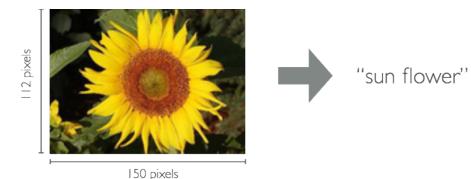


Image Classification

Question: "What is this an image of?"

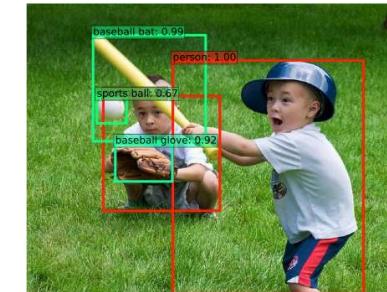
Answer: "95% probability this is a ballplayer"



Object Detection

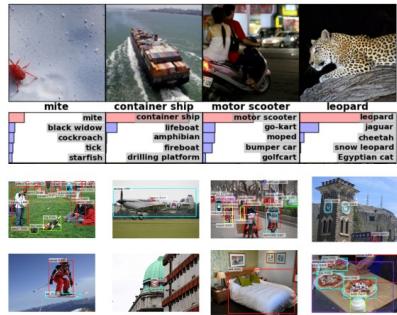
Question: "What are all the objects in this object and where are they?"

Answer:



Classification versus detection

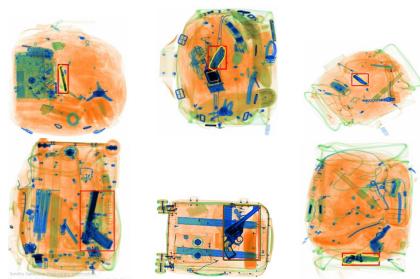
- Very related, but different problems.
- Many aspects will be shared between the two problems
- In detection algorithms, we try to draw a bounding box around the object of interest to locate it within the image.
- There could be many bounding boxes representing different objects of interest within the image and you would not know how many beforehand.



Application: Self-Driving Cars



Application: TSA



Useful Properties

We have some important properties that are useful

- Translation Invariance
- Scale Invariance
- Rotation Invariance

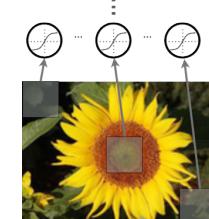
Some are built into the algorithm, and some come from the structure of the dataset

Convolutional Neural Networks

- Our goal is to design neural networks that are specifically adapted for such problems
 - Must deal with very **high-dimensional inputs**: 150×150 pixels = 22500 inputs, or 3×22500 if RGB pixels
 - Can exploit the **2D topology** of pixels (or 3D for video data)
 - Can build on **invariance** to certain variations: translation, illumination, etc.
- **Convolutional networks** leverage these ideas
 - Local connectivity
 - Parameter sharing
 - Convolution
 - Pooling / subsampling hidden units

Local Connectivity

- Use **local connectivity** of hidden units
 - Each hidden unit is connected only to a sub-region (patch) of the input image
 - It is connected to all channels: 1 if grayscale, 3 (R, G, B) if color image
- Why local connectivity?
 - Fully connected layer has **a lot of parameters** to fit, requires a lot of data
 - Spatial correlation is local

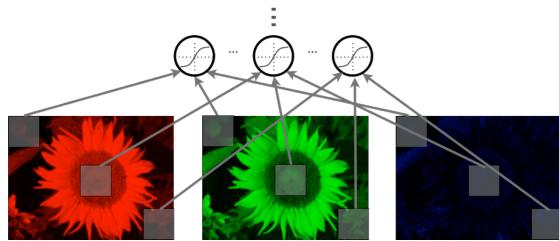


r [] = receptive field

Local Connectivity

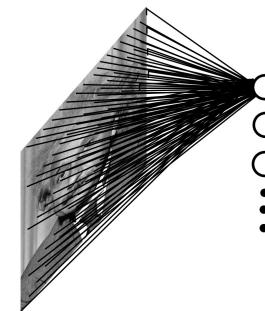
- Units are connected to all channels:

- 1 channel if grayscale image,
- 3 channels (R, G, B) if color image



Local Connectivity

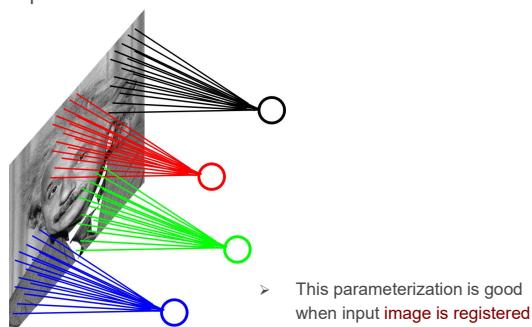
- Example: 200x200 image, 40K hidden units, ~2B parameters!



- Spatial correlation is local
- Too many parameters, will require a lot of training data!

Local Connectivity

- Example: 200x200 image, 40K hidden units, filter size 10x10, 4M parameters!



- This parameterization is good when input image is registered

Convolutional Neural Networks

- Our goal is to design neural networks that are specifically adapted for such problems

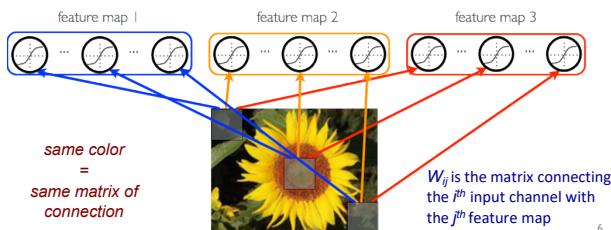
- Must deal with very **high-dimensional** inputs: 150×150 pixels = 22500 inputs, or 3 x 22500 if RGB pixels
- Can exploit the **2D topology** of pixels (or 3D for video data)
- Can build in **invariance** to certain variations: translation, illumination, etc.

- Convolutional networks leverage these ideas

- Local connectivity
- Parameter sharing
- Convolution
- Pooling / subsampling hidden units

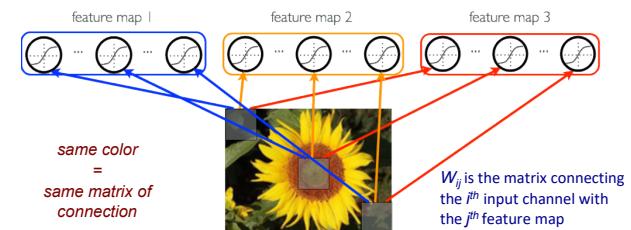
Parameter Sharing

- Share matrix of parameters across some units
 - Units that are organized into the ‘feature map’ share parameters
 - Hidden units within a feature map cover different positions in the image



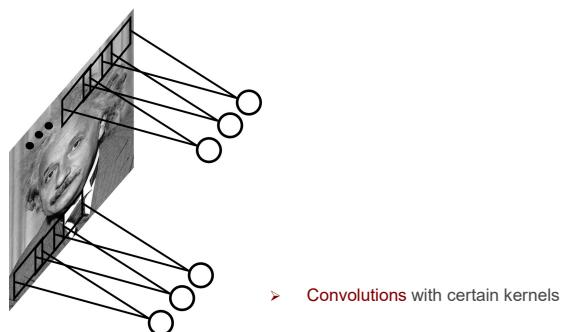
Parameter Sharing

- Why **parameter sharing**?
 - Reduces even more the number of parameters
 - Will extract the same features at every position (**features are “equi-variant”**)



Parameter Sharing

- Share matrix of parameters across certain units

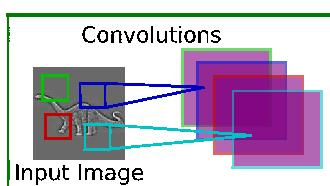


Computer Vision

- Our goal is to design neural networks that are specifically adapted for such problems
 - Must deal with very **high-dimensional** inputs: 150×150 pixels = 22500 inputs, or 3 \times 22500 if RGB pixels
 - Can exploit the **2D topology** of pixels (or 3D for video data)
 - Can build in **invariance** to certain variations: translation, illumination, etc.
- Convolutional networks leverage these ideas
 - Local connectivity
 - Parameter sharing
 - Convolution
 - Pooling / sub-sampling hidden units

Parameter Sharing

- Each feature map forms a 2D grid of features
 - can be computed with a discrete convolution ($*$) of a **kernel matrix** k_j which is the hidden weights matrix W_j with its rows and columns flipped



$$y_j = g_j \tanh\left(\sum_i k_{ij} * x_i\right)$$

- x_i is the i^{th} channel of input
- k_{ij} is the convolution kernel
- g_j is a learned scaling factor
- y_j is the hidden layer

can add bias

Discrete Convolution

- Discrete convolution between **one** kernel (filter) and **one** channel image (matrix; 2-d tensor)
- This is not quite convolution; instead, it is correlation

$$(x * k)_{ij} = \sum_{p,q} x_{i+p,j+q} \cdot k_{p,q}$$

- Example:

$$\begin{array}{ccc} 0 & 80 & 40 \\ 20 & 40 & 0 \\ 0 & 0 & 40 \end{array} * \begin{array}{cc} 0 & 0,25 \\ 0,5 & 1 \end{array} = k$$

x

Discrete Convolution

$$(x * k)_{ij} = \sum_{p,q} x_{i+p,j+q} \cdot k_{p,q}$$

- Example:

$$\begin{array}{ccc} 1 & 0,5 & 80 & 40 \\ 0,25 & 0 & 40 & 0 \\ 0 & 0 & 40 \end{array} * \begin{array}{cc} 0 & 0,25 \\ 0,5 & 1 \end{array} = \tilde{k} = k$$

x

with rows and columns flipped

Discrete Convolution

$$(x * k)_{ij} = \sum_{p,q} x_{i+p,j+q} \cdot k_{p,q}$$

- Example: $1 \times 0 + 0,5 \times 80 + 0,25 \times 20 + 0 \times 40 = 45$

$$\begin{array}{ccc} 1 & 0,5 & 80 & 40 \\ 0,25 & 0 & 40 & 0 \\ 0 & 0 & 40 \end{array} * \begin{array}{cc} 0 & 0,25 \\ 0,5 & 1 \end{array} = 45$$

x

k

Discrete Convolution

$$(x * k)_{ij} = \sum_{p,q} x_{i+p,j+q} \cdot k_{p,q}$$

- Example: $1 \times 80 + 0.5 \times 40 + 0.25 \times 40 + 0 \times 0 = 110$

x k

Discrete Convolution

$$(x * k)_{ij} = \sum_{p,q} x_{i+p,j+q} \cdot k_{p,q}$$

- Example: $1 \times 20 + 0.5 \times 40 + 0.25 \times 0 + 0 \times 0 = 40$

x k

Discrete Convolution

$$(x * k)_{ij} = \sum_{p,q} x_{i+p,j+q} \cdot k_{p,q}$$

- Example: $1 \times 40 + 0.5 \times 0 + 0.25 \times 0 + 0 \times 40 = 40$

x k

Discrete Convolution

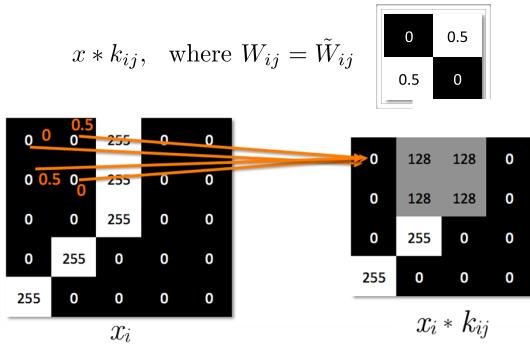
- Pre-activations from channel x_i into feature map y_j can be computed by:
 - getting the convolution kernel where $k_{ij} = W_{ij}$ from the connection matrix W_{ij}
 - applying the correlation $x_i \cdot k_{ij}$
 - We abuse the terminology and notation and refer and use the same notation for convolution and correlation when there is no ambiguity.

- This is equivalent to computing the discrete correlation of x_i with W_{ij}
- Discrete convolution in general form (for f^{th} output filter (kernel), for c^{th} input channel)
- k is a 4-d Tensor which is convolved to the 3-d Tensor input x

$$(x * k)_{fij} = \sum_c \sum_{p,q} x_{c,i+p,j+q} \cdot k_{c,p,q,f}$$

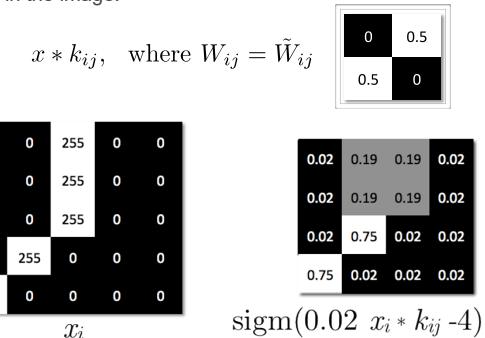
Example

- Illustration:



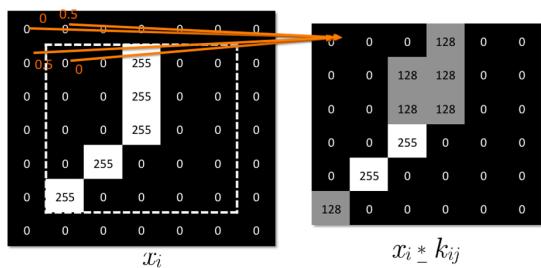
Example

- With a non-linearity, we get a detector of a feature at any position in the image:



Example

- Can use “zero padding” to allow going over the borders (*)

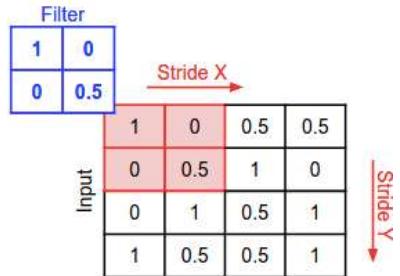


CNN Convolution Parameters

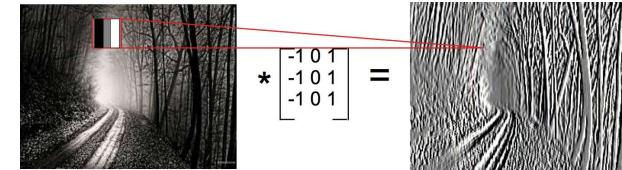
CNN — Parameters

- **Filters:** Represents the amount of filters in a CL.
- **Kernel Size:** Defines the dimensions of the filters.
- **Stride:** Sets the size of the filter shift step.
- **Padding:** defines whether or not there is entry zeroing, influencing the output dimensions:

CNN Convolution Parameters

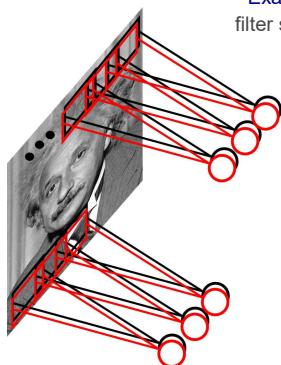


Example



Multiple Feature Maps

- Example: 200x200 image, 100 filters, filter size 10x10, 10K parameters



Convolutional Neural Networks

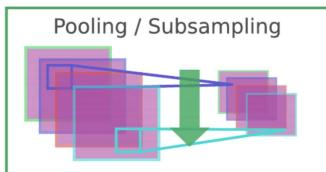
- Our goal is to design neural networks that are specifically adapted for such problems
 - Must deal with very **high-dimensional** inputs: 150×150 pixels = 22500 inputs, or 3×22500 if RGB pixels
 - Can exploit the **2D topology** of pixels (or 3D for video data)
 - Can build in **invariance** to certain variations: translation, illumination, etc.
- Convolutional networks leverage these ideas
 - Local connectivity
 - Parameter sharing
 - Convolution
 - Pooling / subsampling hidden units

Pooling

- Pool hidden units in same neighborhood

- **pooling** is performed in non-overlapping neighborhoods (subsampling)

$$y_{ijk} = \max_{p,q} x_{i,j+p,k+q}$$



Jarret et al. 2009

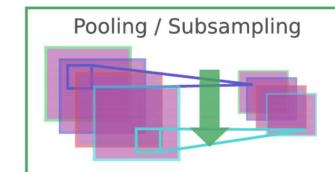
- x_i is the i^{th} channel of input
- $x_{i,j,k}$ is value of the i^{th} feature map at position j,k
- p is vertical index in local neighborhood
- q is horizontal index in local neighborhood
- y_{ijk} is pooled / subsampled layer

Pooling

- Pool hidden units in same neighborhood

- an alternative to “max” pooling is “average” pooling

$$y_{ijk} = \frac{1}{m^2} \sum_{p,q} x_{i,j+p,k+q}$$

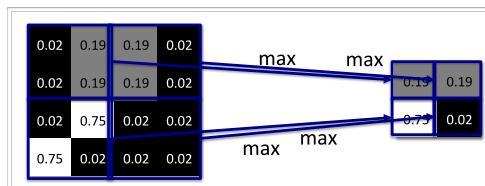


Jarret et al. 2009

- x_i is the i^{th} channel of input
- $x_{i,j,k}$ is value of the i^{th} feature map at position j,k
- p is vertical index in local neighborhood
- q is horizontal index in local neighborhood
- y_{ijk} is pooled / subsampled layer
- m is the neighborhood height/width

Example: Pooling

- Illustration of pooling/subsampling operation

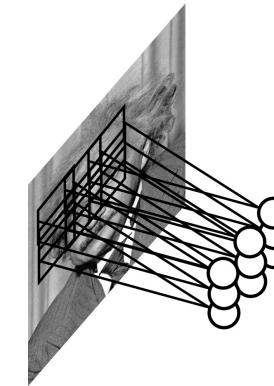


- Why pooling?

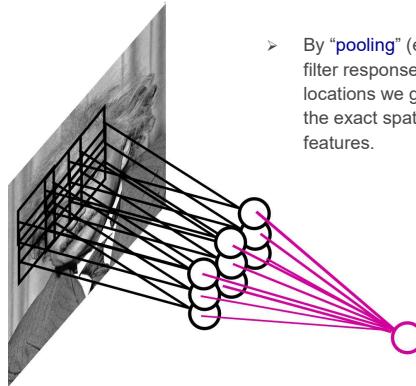
- Introduces invariance to **local translations**
- Reduces the number of hidden units in hidden layer

Example: Pooling

- can we make the detection robust to the exact location of the eye?



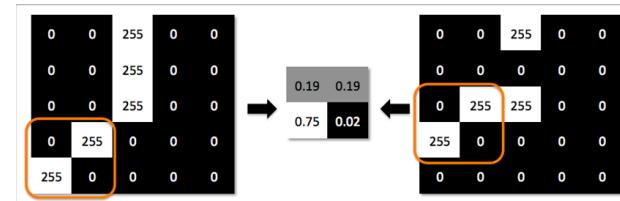
Example: Pooling



- By "pooling" (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

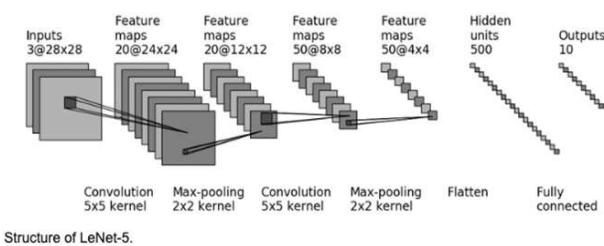
Translation Invariance

- Illustration of local translation invariance
 - both images result in the same feature map after pooling



Convolutional Network

- Convolutional neural network alternates between the convolutional and pooling layers



Structure of LeNet-5.

From Yann LeCun's slides

Convolutional Network

- For classification: Output layer is a regular, fully connected layer with softmax non-linearity
 - Output provides an estimate of the conditional probability of each class
- The network is trained by stochastic gradient descent
 - Backpropagation is used similarly as in a fully connected network
 - We have seen how to pass gradients through element-wise activation function
 - We also need to pass gradients through the convolution operation and the pooling operation

Gradient of Pooling Layer

- Let l be the loss function

- For max pooling operation $y_{ijk} = \max_{p,q} x_{i,j+p,k+q}$, the gradient for x_{ijk} is
 $\nabla_{x_{ijk}} l = 0$, except for $\nabla_{x_{i,j+p',k+q'}} l = \nabla_{y_{ijk}} l$
- where $p', q' = \text{argmax } x_{i,j+p,k+q}$
- In other words, only the “winning” units in layer x get the gradient from the pooled layer
- For the average operation $y_{ijk} = \frac{1}{m^2} \sum_{p,q} x_{i,j+p,k+q}$, the gradient for x_{ijk} is
 $\nabla_x l = \frac{1}{m^2} \text{upsample}(\nabla_y l)$

where you should calculate $\text{upsample}(.)$ as an exercise.

Gradient of Convolutional Layer

The goal is to compute the gradient of the loss function w.r.t. to the weights of the filters in layer h^u and input X^{u-1} given the gradient in the layer h^u .

Remember $h^u = g(w^u * X^{u-1})$, where g is some nonlinear operator (nonlinear activation, pooling...)

Assume you have computed the gradient of loss function, l , up to the **current hidden convolutional layer h^u** , i.e., $\partial_{h_{ijk}^u} \triangleq \nabla_{h_{ijk}^u} l = \frac{\partial l}{\partial h_{ijk}^u}$.

Here h stands for “hidden” layer and has been introduced for the ease of notation. Also, index i denotes the channel number of the current hidden layer h_u . That is, $i = 1, 2, \dots, C_u$ and $u = 1, 2, \dots, L$, where C_u denotes the number of channels in layer u and L is the total number of layers.

For simplicity drop the channel index and the layer number. So, $\partial_{h_{ij}} \triangleq \frac{\partial l}{\partial h_{ij}}$.

All the computations should be done for all the channels and for all the convolutional layers.

Similarly define the gradient w.r.t. to the filter coefficients as $\partial_{w_{ij}} \triangleq \frac{\partial l}{\partial w_{ij}}$

Gradient of Convolutional Layer -- Continue

Now, we establish the gradient operation visually¹.

Assume in forward pass, we have convolved a 3x3 input with a kernel 2x2 which outputs a 2x2 matrix

$$g[X^{u-1}] \quad \begin{matrix} \begin{array}{|c|c|c|} \hline x_{11} & x_{12} & x_{13} \\ \hline x_{21} & x_{22} & x_{23} \\ \hline x_{31} & x_{32} & x_{33} \\ \hline \end{array} & \begin{array}{|c|c|} \hline w_{11} & w_{12} \\ \hline w_{21} & w_{22} \\ \hline \end{array} \end{matrix} = \begin{matrix} \begin{array}{|c|c|} \hline h_{11} & h_{12} \\ \hline h_{21} & h_{22} \\ \hline \end{array} \end{matrix}$$

Filter h^u

We calculate this for $g(z) = z$. Please extend to the general case as an exercise

With the notation from the previous slide, we can write:

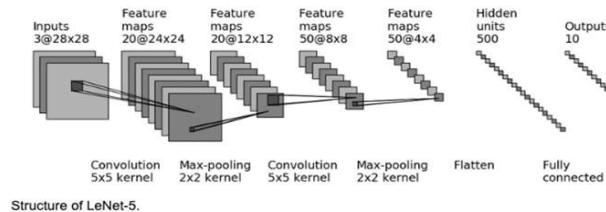
$$\begin{aligned} \partial_{w_{11}} &= X_{11}\partial_{h_{11}} + X_{12}\partial_{h_{12}} + X_{21}\partial_{h_{21}} + X_{22}\partial_{h_{22}} \\ \partial_{w_{12}} &= X_{12}\partial_{h_{11}} + X_{13}\partial_{h_{12}} + X_{22}\partial_{h_{21}} + X_{23}\partial_{h_{22}} \\ \partial_{w_{21}} &= X_{21}\partial_{h_{11}} + X_{22}\partial_{h_{12}} + X_{31}\partial_{h_{21}} + X_{32}\partial_{h_{22}} \\ \partial_{w_{22}} &= X_{22}\partial_{h_{11}} + X_{23}\partial_{h_{12}} + X_{32}\partial_{h_{21}} + X_{33}\partial_{h_{22}} \end{aligned}$$

Gradient of Convolutional Layer -- Continue

- This is our old friend, discrete convolution operator:
 $\partial_W = X * \partial_h$
- Where $\partial_{w_{ij}} = \sum_{p,q} X_{i+p,j+q} \partial_{h_{ij}}$
- Similarly, we can compute the gradient of the loss w.r.t. X (input layer, or X^{u-1}) since we need these gradients in order to propagate the gradient to the layers towards input of the CNN.

Convolutional Network

- Convolutional neural network alternates between the convolutional and pooling layers

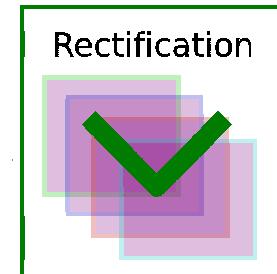


- Need to introduce **other operations** that can improve object recognition.

Rectification

- Rectification layer:** $y_{ijk} = |x_{ijk}|$

- introduces invariance to the sign of the unit in the previous layer
- for instance, loss of information of whether an edge is black-to-white or white-to-black



Local Contrast Normalization

- Perform **local contrast normalization**

$$v_{ijk} = x_{ijk} - \frac{1}{\sum_{ipq} \alpha_{pq} x_{i,j+p,k+q}}$$

Local average

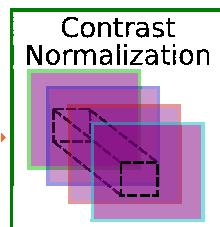
$$y_{ijk} = v_{ijk} / \max(c, \sigma_{jk})$$

Local stdev

$$\sigma_{jk} = \left[\left(\sum_{ipq} \alpha_{pq} v_{i,j+p,k+q}^2 \right)^{1/2} \right], \sum_{pq} \alpha_{pq} = 1$$

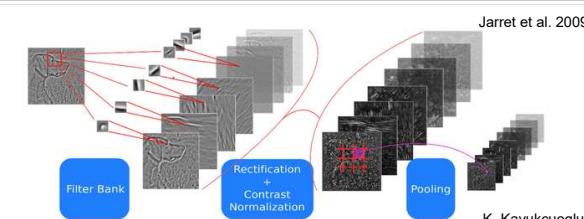
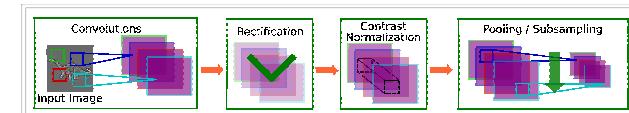
where c is a small constant to prevent division by 0 and $\alpha_{pq} \geq 0$.

- reduces unit's activation if neighbors are also active
- creates competition between feature maps
- scales activations at each layer better for learning



Convolutional Network

- These operations are inserted after the convolutions and before the pooling



Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

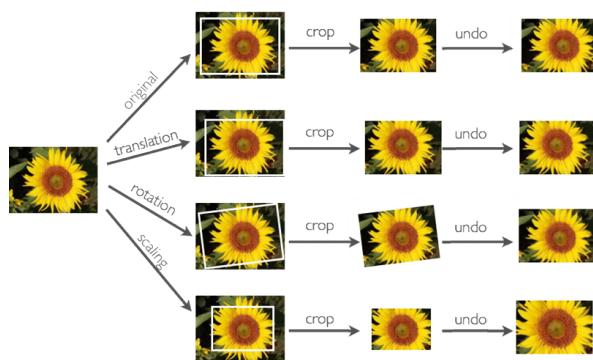
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

Learned linear transformation to adapt to non-linear activation function (γ and β are trained)

Invariance by Dataset Expansion (Augmentation)

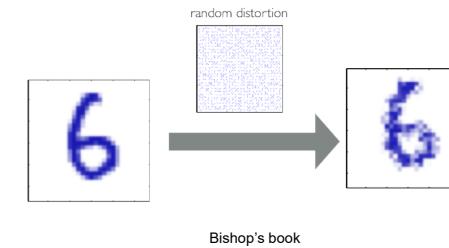
- **Invariances** built-in in convolutional network:
 - **small translations:** due to convolution and max pooling
 - **small illumination changes:** due to local contrast normalization
- It is not invariant to other important variations such as rotations and scale changes
- However, it's easy to artificially generate data with such transformations
 - could use such data as additional training data
 - neural network can potentially learn to be invariant to such transformations

Generating Additional Examples



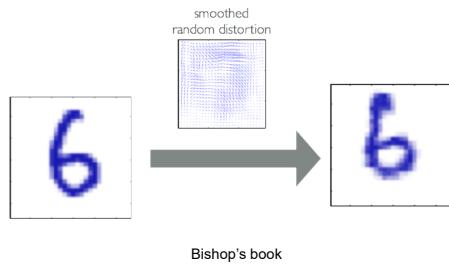
Elastic Distortions

- Can add “elastic” deformations (useful in character recognition)
- We can do this by applying a “distortion field” to the image
 - a distortion field specifies where to displace each pixel value



Elastic Distortions

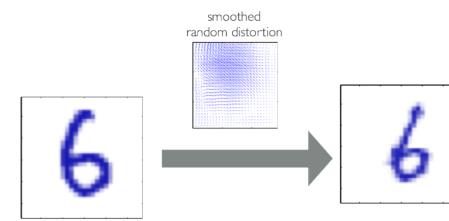
- Can add “elastic” deformations (useful in character recognition)
- We can do this by applying a “distortion field” to the image
 - a distortion field specifies where to displace each pixel value



Bishop's book

Elastic Distortions

- Can add “elastic” deformations (useful in character recognition)
- We can do this by applying a “distortion field” to the image
 - a distortion field specifies where to displace each pixel value



Bishop's book

Conv Nets: Examples

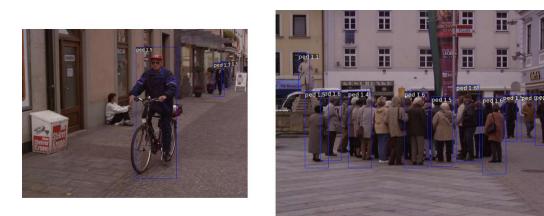
- Optical Character Recognition, House Number and Traffic Sign classification



Ciresan et al. “MCDNN for image classification” CVPR 2012
Wan et al. “Regularization of neural networks using dropconnect” ICML 2013
Goodfellow et al. “Multi-digit number recognition from StreetView..” ICLR 2014
Jaderberg et al. “Synthetic data and ANN for natural scene text recognition” arXiv 2014

Conv Nets: Examples

- Pedestrian detection



Sermanet et al. “Pedestrian detection with unsupervised multi-stage..” CVPR 2013

Conv Nets: Examples

- Object Detection



Sermanet et al. "OverFeat: Integrated recognition, localization" arxiv 2013
Girshick et al. "Rich feature hierarchies for accurate object detection" arxiv 2013
Szegedy et al. "DNN for object detection" NIPS 2013

ImageNet Dataset

- 1.2 million images, 1000 classes

Examples of Hammer

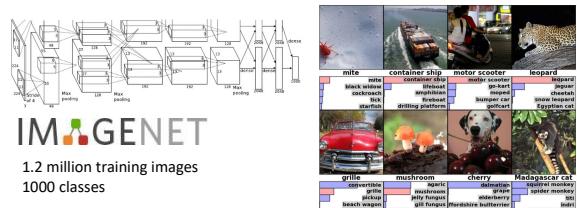


Deng et al. "Imagenet: a large scale hierarchical image database" CVPR 2009

Important Breakthroughs

- Deep Convolutional Nets for Vision (Supervised)

Krizhevsky, A., Sutskever, I. and Hinton, G. E., ImageNet Classification with Deep Convolutional Neural Networks, NeurIPS, 2012.



Architecture

- How can we select the right architecture:

➢ Manual tuning of features is now replaced with the manual tuning of architectures

• Depth

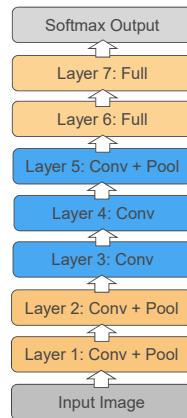
• Width

• Parameter count

AlexNet

- 8 layers total
- Trained on Imagenet dataset [Deng et al. CVPR'09]
- 18.2% top-5 error

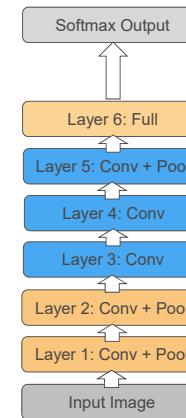
[From Rob Fergus' CIFAR 2016 tutorial]



AlexNet

- Remove top fully connected layer 7
- Drop ~16 million parameters
- Only 1.1% drop in error!

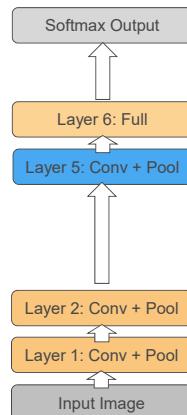
[From Rob Fergus' CIFAR 2016 tutorial]



AlexNet

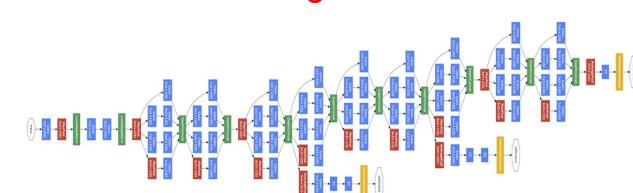
- Let us remove upper feature extractor layers and fully connected:
 - Layers 3,4, 6 and 7
- Drop ~50 million parameters
- **33.5 drop in error!**
- **Depth of the network is the key.**

[From Rob Fergus' CIFAR 2016 tutorial]



GoogLeNet

- 24 layer model that uses so-called inception module.



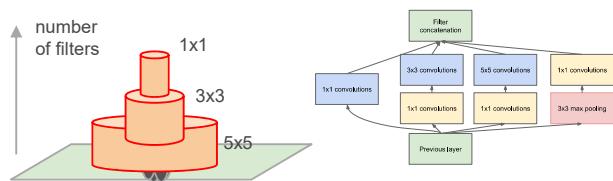
Convolution
Pooling
Softmax
Other

[Going Deep with Convolutions, Szegedy et al., arXiv:1409.4842, 2014]

GoogLeNet

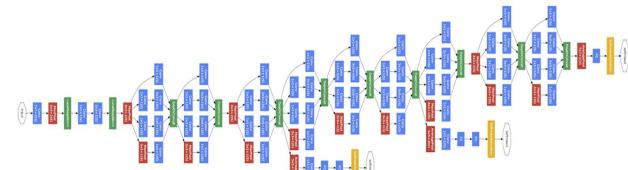
- GoogLeNet inception module:

- Multiple filter scales at each layer
- Dimensionality reduction to keep computational requirements down



[Going Deep with Convolutions, Szegedy et al., arXiv:1409.4842, 2014]

GoogLeNet

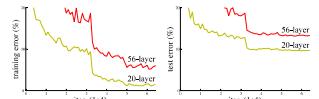


- Width of inception modules ranges from 256 filters (in early modules) to 1024 in top inception modules.
- Can remove fully connected layers on top completely
- Number of parameters is reduced to 5 million
- 6.7% top-5 validation error on ImageNet

[Going Deep with Convolutions, Szegedy et al., arXiv:1409.4842, 2014]

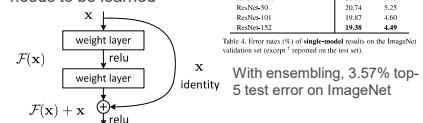
Residual Networks

Really, really deep convolutional nets do not train well,
E.g. CIFAR10:



Key idea: introduce "pass through" into each layer

Thus only residual now
needs to be learned



[He, Zhang, Ren, Sun, CVPR 2016]

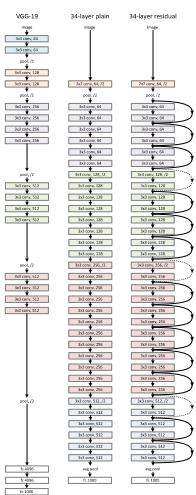


Table 4. Error rates (%) of single-model results on the ImageNet validation set except* reported on the test set.

*Reported on the test set.

With ensembling, 3.57% top-5 test error on ImageNet

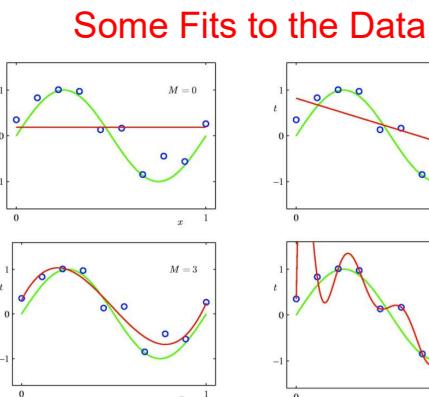
Choosing the Architecture

- Task dependent
- Cross-validation
- [Convolution → pooling]* + fully connected layer
- The more data: the more layers and the more kernels
 - Look at the **number of parameters** at each layer
 - Look at the **number of flops** at each layer
- Computational resources

[From Marc'Aurelio Ranzato, CVPR 2014 tutorial]

Underfitting, Overfitting and Training Tricks

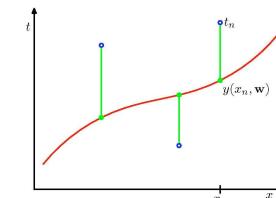
Vahid Tarokh
ECE 685D, Fall 2025



For M=9, we have fitted the training data perfectly.

Example: Polynomial Curve Fitting

- As for the least squares example: we can minimize the sum of the squares of the errors between the predictions $y(x_n, \mathbf{w})$ for each data point x_n and the corresponding target values t_n .



Loss function: sum-of-squared error function:

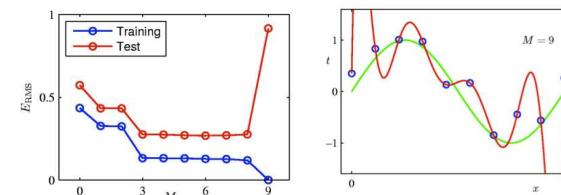
$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^N (y(x_n, \mathbf{w}) - t_n)^2.$$

- Similar to the linear least squares: Minimizing sum-of-squared error function has a unique solution \mathbf{w}^* .
- The model is characterized by $M+1$ parameters \mathbf{w}^* .
- How do we choose M? ! **Model Selection**.

Some Fits to the Data

Overfitting

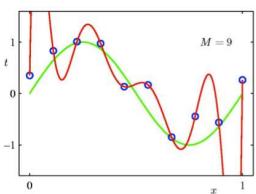
- Consider a separate **test set** containing 100 new data points generated using the same procedure that was used to generate the training data.



- For $M=9$, the training error is zero! The polynomial contains 10 degrees of freedom corresponding to 10 parameters \mathbf{w} , and so can be fitted exactly to the 10 data points.
- However, the test error has become very large. Why?

Overfitting

	$M = 0$	$M = 1$	$M = 3$	$M = 9$
w_0^*	0.19	0.82	0.31	0.35
w_1^*		-1.27	7.99	232.37
w_2^*			-25.43	-5321.83
w_3^*			17.37	48568.31
w_4^*				-231639.30
w_5^*				640042.26
w_6^*				-1061800.52
w_7^*				1042400.18
w_8^*				-557682.99
w_9^*				125201.43

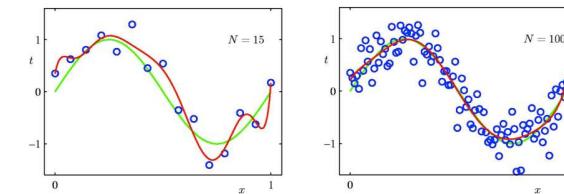


- As M increases, the magnitude of coefficients gets larger.
- For $M=9$, the coefficients have become finely tuned to the data.
- Between data points, the function exhibits large oscillations.

More flexible polynomials with larger M tune to the random noise on the target values.

Varying the Size of the Data

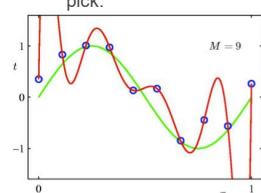
9th order polynomial



- For a given model complexity, the overfitting problem becomes less severe as the size of the dataset increases.
- However, the number of parameters is not necessarily the most appropriate measure of the model complexity.

Generalization

- The goal is achieve good **generalization** by making accurate predictions for new test data that is not known during learning.
- Choosing the values of parameters that minimize the loss function on the training data may not be the best option.
- We would like to model the true regularities in the data and ignore the noise in the data:
 - It is hard to know which regularities are real and which are accidental due to the particular training examples we happen to pick.



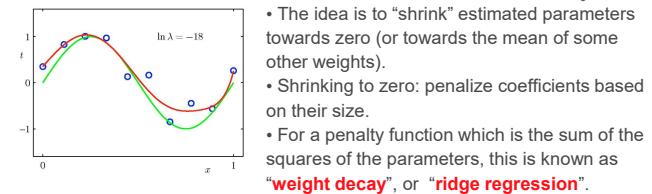
- Intuition:** We expect the model to generalize if it explains the data well given the complexity of the model.
- If the model has as many degrees of freedom as the data, it can fit the data perfectly. But this is not very informative.
- Some theory on how to control model complexity to optimize generalization.

A Simple Way to Penalize Complexity

One technique for controlling over-fitting phenomenon is **regularization**, which amounts to adding a penalty term to the error function.

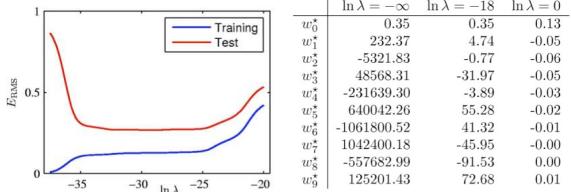
$$\tilde{E}(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \{y(x_n, \mathbf{w}) - t_n\}^2 + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

where $\|\mathbf{w}\| = \mathbf{w}^T \mathbf{w} = w_1^2 + w_2^2 + \dots + w_M^2$ and λ is called the regularization term. Note that we do not penalize the bias term w_0 .



- The idea is to "shrink" estimated parameters towards zero (or towards the mean of some other weights).
- Shrinking to zero: penalize coefficients based on their size.
- For a penalty function which is the sum of the squares of the parameters, this is known as **"weight decay"**, or **"ridge regression"**.

Regularization



Graph of the root-mean-squared training and test errors vs. $\ln \lambda$ for the $M=9$ polynomial.

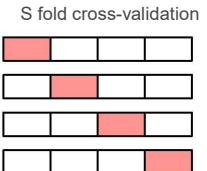
How to choose λ ?

Cross Validation

If the data is plentiful, we can divide the dataset into three subsets:

- **Training Data:** used to fitting/learning the parameters of the model.
- **Validation Data:** not used for learning but for selecting the model, or choosing the amount of regularization that works best.
- **Test Data:** used to get performance of the final model.

For many applications, the supply of data for training and testing is limited. To build good models, we may want to use as much training data as possible. If the validation set is small, we get noisy estimate of the predictive performance.



- The data is partitioned into S groups.
- Then $S-1$ of the groups are used for training the model, which is evaluated on the remaining group.
- Repeat procedure for all S possible choices of the held-out group.
- Performance from the S runs are averaged.

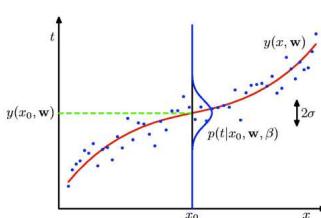
Probabilistic Perspective Of Polynomial Regression

• So far we saw that polynomial curve fitting can be expressed in terms of error minimization. We now view it from probabilistic perspective.

• Suppose that our model arose from a statistical model:

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon,$$

where ϵ is a random error having Gaussian distribution with zero mean, and is independent of \mathbf{x} .



Thus we have:

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}),$$

where β is a precision parameter, corresponding to the inverse variance.

We will use probability distribution and probability density interchangeably. It should be obvious from the context.

Maximum Likelihood

If the data are assumed to be independently and identically distributed (*i.i.d assumption*), the likelihood function takes form:

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \beta) = \prod_{i=1}^N \mathcal{N}(t_n | y(\mathbf{x}_n, \mathbf{w}), \beta^{-1}).$$

It is often convenient to maximize the log of the likelihood function:

$$\ln p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \beta) = -\frac{\beta}{2} \sum_{n=1}^N (y(\mathbf{x}_n, \mathbf{w}) - t_n)^2 + \frac{N}{2} \ln \beta - \frac{N}{2} \ln(2\pi).$$

$\underbrace{\beta}_{\beta E(\mathbf{w})}$

• Maximizing log-likelihood with respect to \mathbf{w} (under the assumption of a Gaussian noise) is equivalent to minimizing the *sum-of-squared error* function.

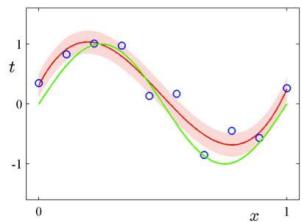
• Determine \mathbf{w}_{ML} by maximizing log-likelihood. Then maximizing w.r.t. β :

$$\frac{1}{\beta_{ML}} = \frac{1}{N} \sum_n (y(\mathbf{x}_n, \mathbf{w}_{ML}) - t_n)^2.$$

Predictive Distribution

Once we determined the parameters \mathbf{w} and β , we can make prediction for new values of \mathbf{x} :

$$p(t|\mathbf{x}, \mathbf{w}_{ML}, \beta_{ML}) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}_{ML}), \beta_{ML}^{-1}).$$



Maximum Likelihood

- As before, assume observations arise from a deterministic function with an additive Gaussian noise:

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon,$$

which we can write as:

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}).$$

- Given observed inputs $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$, and corresponding target values $\mathbf{t} = [t_1, t_2, \dots, t_N]^T$, under i.i.d assumption, we can write down the likelihood function:

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{i=1}^N \mathcal{N}(t_n | \mathbf{w}^T \phi(\mathbf{x}_n), \beta),$$

where $\phi(\mathbf{x}) = (\phi_0(\mathbf{x}), \phi_1(\mathbf{x}), \dots, \phi_{M-1}(\mathbf{x}))^T$.

Maximum Likelihood

Taking the logarithm, we obtain:

$$\begin{aligned} \ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) &= \sum_{i=1}^N \ln \mathcal{N}(t_n | \mathbf{w}^T \phi(\mathbf{x}_n), \beta) \\ &= -\frac{\beta}{2} \sum_{n=1}^N (t_n - \mathbf{w}^T \phi(\mathbf{x}_n))^2 + \frac{N}{2} \ln \beta - \frac{N}{2} \ln(2\pi). \end{aligned}$$

sum-of-squares error function

Differentiating and setting to zero yields:

$$\nabla_{\mathbf{w}} \ln p(\mathbf{t}|\mathbf{w}, \beta) = \beta \sum_{n=1}^N \{t_n - \mathbf{w}^T \phi(\mathbf{x}_n)\} \phi(\mathbf{x}_n)^T = \mathbf{0}.$$

Maximum Likelihood

Differentiating and setting to zero yields:

$$\nabla_{\mathbf{w}} \ln p(\mathbf{t}|\mathbf{w}, \beta) = \beta \sum_{n=1}^N \{t_n - \mathbf{w}^T \phi(\mathbf{x}_n)\} \phi(\mathbf{x}_n)^T = \mathbf{0}.$$

Solving for \mathbf{w} , we get:

$$\mathbf{w}_{ML} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{t}$$

Depends on Data

The Moore-Penrose pseudo-inverse of Φ^\dagger

where Φ is known as the **design matrix**:

$$\Phi = \begin{pmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \cdots & \phi_{M-1}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \cdots & \phi_{M-1}(\mathbf{x}_N) \end{pmatrix}.$$

Sequential Learning

- The training data examples are presented one at a time, and the model parameters are updated after each such presentation (online learning):

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \triangledown E_n$$

weights after seeing training case t+1 learning rate vector of derivatives of the squared error w.r.t. the weights on the training case presented at time t.

- For the case of sum-of-squares error function, we obtain:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \left(t_n - \mathbf{w}^{(t)T} \phi(\mathbf{x}_n) \right) \phi(\mathbf{x}_n).$$

- Stochastic gradient descent:** The training examples are picked at random (dominant technique when learning with very large datasets).
- Care must be taken when choosing learning rate to ensure convergence.

Regularized Least Squares

- Let us consider the following error function:

$$E_D(\mathbf{w}) + \lambda E_W(\mathbf{w})$$

Data term + Regularization term

λ is called the regularization coefficient.

- Using sum-of-squares error function with a quadratic penalization term, we obtain:

$$\frac{1}{2} \sum_{n=1}^N \{t_n - \mathbf{w}^T \phi(\mathbf{x}_n)\}^2 + \frac{\lambda}{2} \mathbf{w}^T \mathbf{w}$$

which is minimized by setting:

Depends on Data

$$\mathbf{w} = (\lambda \mathbf{I} + \Phi^T \Phi)^{-1} \Phi^T \mathbf{t}.$$

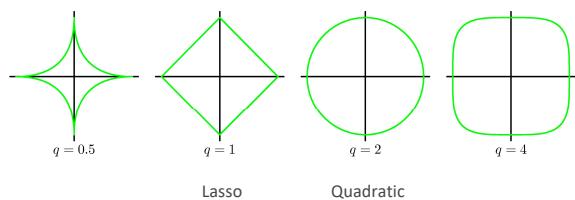
Ridge regression

The solution adds a positive constant to the diagonal of $\Phi^T \Phi$. This makes the problem nonsingular, even if $\Phi^T \Phi$ is not of full rank (e.g. when the number of training examples is less than the number of basis functions).

Other Regularizers

Using a more general regularizer, we get:

$$\frac{1}{2} \sum_{n=1}^N \{t_n - \mathbf{w}^T \phi(\mathbf{x}_n)\}^2 + \frac{\lambda}{2} \sum_{j=1}^M |w_j|^q$$



The LASSO

- Penalize the absolute value of the weights:

$$\mathbf{w}^{lasso} = \underset{\mathbf{w}}{\operatorname{argmin}} \left[\frac{1}{2} \sum_{n=1}^N (t_n - \mathbf{w}^T \phi(\mathbf{x}_n))^2 + \frac{\lambda}{2} \sum_{j=1}^{M-1} |w_j| \right].$$

- For sufficiently large λ , some of the coefficients will be driven to exactly zero, leading to a sparse model.

- The above formulation is equivalent to:

$$\mathbf{w}^{lasso} = \underset{\mathbf{w}}{\operatorname{argmin}} \underbrace{\frac{1}{2} \sum_{n=1}^N (t_n - \mathbf{w}^T \phi(\mathbf{x}_n))^2}_{\text{unregularized sum-of-squares error}}, \text{ subject to } \sum_{j=1}^{M-1} |w_j| \leq \tau.$$

- The two approaches are related using Lagrange multiplies.

- The LASSO solution is a quadratic programming problem: can be solved efficiently.

Review of Inference

Assume that the training examples are drawn **independently** from the set of all possible examples, or from the same underlying distribution $p(\mathbf{x}, t)$.

We also assume that the training examples are **identically distributed** (i.i.d assumption).

Assume that the test samples are drawn in exactly the same way -- i.i.d from the same distribution as the training data.

These assumptions make it unlikely that some strong regularity in the training data will be absent in the test data.

Statistical Decision Theory

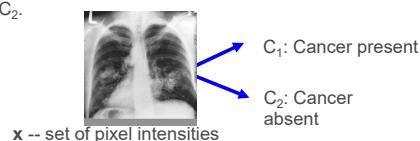
- We now develop a small amount of theory that provides a framework for developing many of the models we consider.
- Suppose we have a real-valued input vector \mathbf{x} and a corresponding target (output) value t with joint probability $p(\mathbf{x}, t)$. distribution:
- Our goal is predict target t given a new value for \mathbf{x} :
 - for regression: t is a real-valued continuous target.
 - for classification: t is a categorical variable representing class labels.

The joint probability distribution $p(\mathbf{x}, t)$ provides a complete summary of uncertainties associated with these random variables.
Determining $p(\mathbf{x}, t)$ from training data is known as the **inference problem**.

Example: Classification

Medical diagnosis: Based on the X-ray image, we would like determine whether the patient has cancer or not.

The input vector \mathbf{x} is the set of pixel intensities, and the output variable t will represent the presence of cancer, class C_1 , or absence of cancer, class C_2 .



Choose t to be binary: $t=0$ correspond to class C_1 , and $t=1$ corresponds to C_2 .

Inference Problem: Determine the joint distribution $p(\mathbf{x}, C_k)$, or equivalently $p(\mathbf{x}, t)$. However, at the end, we must **make a decision** of whether to give treatment to the patient or not.

Example: Classification

Informally: Given a new X-ray image, our goal is to decide which of the two classes that image should be assigned to.

- We could compute conditional probabilities of the two classes, given the input image:

$$p(C_k | \mathbf{x}) = \frac{p(\mathbf{x}, C_k)}{\sum_{k=1}^K p(\mathbf{x}, C_k)} = \frac{p(\mathbf{x}|C_k)p(C_k)}{p(\mathbf{x})}$$

Bayes,
Rule

posterior probability of C_k given observed data.

probability of observed data given C_k

prior probability for class C_k

- If our goal to minimize the probability of assigning \mathbf{x} to the wrong class, then we should choose the class having the highest posterior probability.

Expected Loss

- **Loss Function:** overall measure of loss incurred by taking any of the available decisions.

Suppose that for \mathbf{x} , the true class is C_k , but we assign \mathbf{x} to class j ! incur loss of L_{kj} (k,j element of a loss matrix).

Consider medical diagnosis example: example of a loss matrix:

		Decision	
		cancer	normal
Truth	cancer	0	1000
	normal	1	0

$$\text{Expected Loss: } \mathbb{E}[L] = \sum_k \sum_j \int_{\mathcal{R}_j} L_{kj} p(\mathbf{x}, C_k) d\mathbf{x}$$

Goal is to choose decision regions \mathcal{R}_j as to minimize expected loss.

Regression

Let $\mathbf{x} \in \mathbb{R}^d$ denote a real-valued input vector, and $t \in \mathbb{R}$ denote a real-valued random target (output) variable with joint distribution $p(\mathbf{x}, t)$.

- The decision step consists of finding an estimate $y(\mathbf{x})$ of t for each input \mathbf{x} .

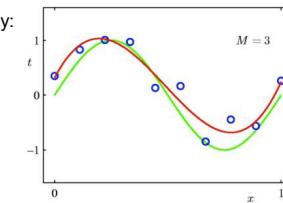
- To quantify what it means to do well or poorly on a task, we need to define a loss (error) function: $L(t, y(\mathbf{x}))$.

- The average, or expected, loss is given by:

$$\mathbb{E}[L] = \int \int L(t, y(\mathbf{x})) p(\mathbf{x}, t) d\mathbf{x} dt.$$

- If we use squared loss, we obtain:

$$\mathbb{E}[L] = \int \int (t - y(\mathbf{x}))^2 p(\mathbf{x}, t) d\mathbf{x} dt.$$



Squared Loss Function

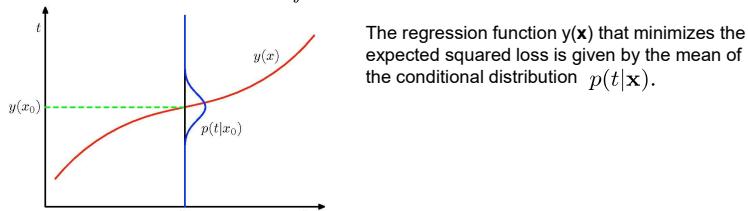
- If we use squared loss, we obtain:

$$\mathbb{E}[L] = \int \int (t - y(\mathbf{x}))^2 p(\mathbf{x}, t) d\mathbf{x} dt.$$

- Our goal is to choose $y(\mathbf{x})$ so as to minimize the expected squared loss.

- The optimal solution (if we assume a completely flexible function) is the conditional average:

$$y(\mathbf{x}) = \int t p(t|\mathbf{x}) dt = \mathbb{E}[t|\mathbf{x}].$$



Squared Loss Function

- If we use squared loss, we obtain:

$$\begin{aligned} (y(\mathbf{x}) - t)^2 &= (y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}] + \mathbb{E}[t|\mathbf{x}] - t)^2 \\ &= (y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}])^2 + 2(y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}])(\mathbb{E}[t|\mathbf{x}] - t) + (\mathbb{E}[t|\mathbf{x}] - t)^2. \end{aligned}$$

- Plugging into expected loss:

$$\mathbb{E}[L] = \underbrace{\int \{y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\}^2 p(\mathbf{x}) d\mathbf{x}}_{\text{expected loss is minimized when } y(\mathbf{x}) = \mathbb{E}[t|\mathbf{x}].} + \underbrace{\int \text{var}[t|\mathbf{x}] p(\mathbf{x}) d\mathbf{x}}_{\text{intrinsic variability of the target values.}}$$

Because it is independent noise, it represents an irreducible minimum value of expected loss.

Other Loss Function

- Simple generalization of the squared loss, called the *Minkowski loss*:

$$\mathbb{E}[L] = \int \int (t - y(\mathbf{x}))^q p(\mathbf{x}, t) d\mathbf{x} dt.$$

- The minimum of $\mathbb{E}[L]$ is given by:

- the conditional mean for $q=2$,
- the conditional median when $q=1$

Bias-Variance Decomposition

- Introducing a regularization term can help us control overfitting. But how can we determine a suitable value of the regularization coefficient?

- Let us examine the expected squared loss function. Remember:

$$\mathbb{E}[L] = \int \{y(\mathbf{x}) - h(\mathbf{x})\}^2 p(\mathbf{x}) d\mathbf{x} + \underbrace{\int \int \{h(\mathbf{x}) - t\}^2 p(\mathbf{x}, t) d\mathbf{x} dt}_{\text{intrinsic variability of the target values: The minimum achievable value of expected loss}}$$

$$h(\mathbf{x}) = \mathbb{E}[t|\mathbf{x}] = \int tp(t|\mathbf{x}) dt.$$

- If we model $h(\mathbf{x})$ using a parametric function $y(\mathbf{x}; \mathbf{w})$, then from a Bayesian perspective, the uncertainty in our model is expressed through the posterior distribution over parameters \mathbf{w} .

- We first look at the frequentist perspective.

Bias-Variance Decomposition

- From a frequentist perspective: we make a point estimate of \mathbf{w}' based on the dataset D .

- We next interpret the uncertainty of this estimate through the following thought experiment:

- Suppose we had a large number of datasets, each of size N , where each dataset is drawn independently from $p(\mathbf{x}, t)$.
- For each dataset D , we can obtain a prediction function $y(\mathbf{x}; D)$.
- Different datasets will give different prediction functions.
- The performance of a particular learning algorithm is then assessed by taking the average over the ensemble of these datasets.

- Let us consider the expression:

$$\{y(\mathbf{x}; D) - h(\mathbf{x})\}^2.$$

- Note that this quantity depends on a particular dataset D .

Bias-Variance Decomposition

- Consider:

$$\{y(\mathbf{x}; D) - h(\mathbf{x})\}^2.$$

- Adding and subtracting the term $\mathbb{E}_D[y(\mathbf{x}; D)]$, we obtain

$$\begin{aligned} & \{y(\mathbf{x}; D) - h(\mathbf{x})\}^2 \\ &= \{y(\mathbf{x}; D) - \mathbb{E}_D[y(\mathbf{x}; D)] + \mathbb{E}_D[y(\mathbf{x}; D)] - h(\mathbf{x})\}^2 \\ &= \{y(\mathbf{x}; D) - \mathbb{E}_D[y(\mathbf{x}; D)]\}^2 + \{\mathbb{E}_D[y(\mathbf{x}; D)] - h(\mathbf{x})\}^2 \\ &\quad + 2\{y(\mathbf{x}; D) - \mathbb{E}_D[y(\mathbf{x}; D)]\}\{\mathbb{E}_D[y(\mathbf{x}; D)] - h(\mathbf{x})\}. \end{aligned}$$

- Taking the expectation over D , the last term vanishes, so we get:

$$\begin{aligned} & \mathbb{E}_D [\{y(\mathbf{x}; D) - h(\mathbf{x})\}^2] \\ &= \underbrace{\{\mathbb{E}_D[y(\mathbf{x}; D)] - h(\mathbf{x})\}^2}_{\text{(bias)}^2} + \underbrace{\mathbb{E}_D [\{y(\mathbf{x}; D) - \mathbb{E}_D[y(\mathbf{x}; D)]\}^2]}_{\text{variance}}. \end{aligned}$$

Bias-Variance Trade-off

$$\text{expected loss} = (\text{bias})^2 + \text{variance} + \text{noise}$$

Average predictions over all datasets differ from the optimal regression function.

Solutions for individual datasets vary around their averages -- how sensitive is the function to the particular choice of the dataset.

Intrinsic variability of the target values.

$$(\text{bias})^2 = \int \{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})] - h(\mathbf{x})\}^2 p(\mathbf{x}) d\mathbf{x}$$

$$\text{variance} = \int \mathbb{E}_{\mathcal{D}} [\{y(\mathbf{x}; \mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]\}]^2 p(\mathbf{x}) d\mathbf{x}$$

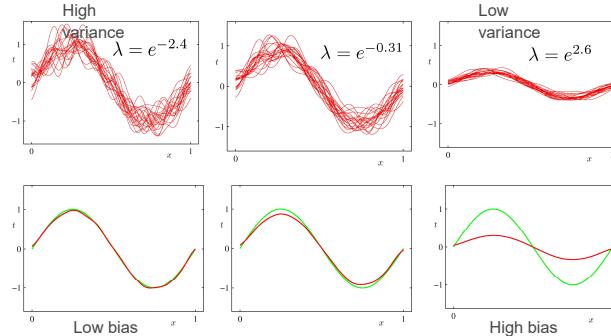
$$\text{noise} = \iint \{h(\mathbf{x}) - t\}^2 p(\mathbf{x}, t) d\mathbf{x} dt$$

- Trade-off between bias and variance: With very flexible models (high complexity) we have low bias and high variance; With relatively rigid models (low complexity) we have high bias and low variance.

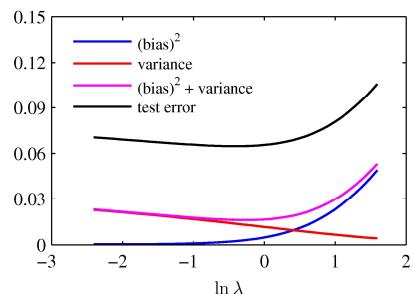
- The model with the **optimal predictive capabilities** has to balance between bias and variance.

Bias-Variance Trade-off

- Consider the sinusoidal dataset. We generate 100 datasets, each containing N=25 points, drawn independently from $h(x) = \sin 2\pi x$.



Bias-Variance Trade-off



From these plots note that over-regularized model (large λ) has high bias, and under-regularized model (low λ) has high variance.

Beating the Bias-Variance Trade-off

- We can reduce the variance by averaging over many models trained on different datasets:

- In practice, we only have a single observed dataset. If we had many independent training sets, we would be better off combining them into one large training dataset.

- Given a standard training set D of size N, we could generate new training sets, of size N, by sampling examples from D uniformly and with replacement.

- This is called **bagging** and it works quite well in practice (**ad hoc**).

- Given enough computation, we could also resort to the Bayesian framework:

- Combine the predictions of many models using the posterior probability of each parameter vector as the combination weight.

Overfitting Issues in Deep Networks

Applications to Deep Networks

Deep nets may have many hidden layers
With limited training data, over-fitting may happen

Methods to reduce overfitting (Regularization)

- Cross-validation set (discussed before)
- Weight regularization, e.g., L_1 and L_2 regularization (discussed before)
- Dropout

In order to understand dropout, we need to first understand bagging.

Bagging (bootstrap aggregating) is a method of averaging over several models to improve generalization. The idea is to train several different models separately, then have all the models vote on the output for test examples.

This is an example of a general strategy in machine learning called model averaging.

- Bagging: average the predictions of all possible settings of the parameters

Bagging

Consider for example a set of k regression models. Suppose that each model makes an error ϵ_i on each example, with the errors drawn from a zero-mean multivariate normal distribution with variances $E(\epsilon_i^2) = \nu$ and co-variances $E(\epsilon_i \epsilon_j) = c$.

Then the error made by the average prediction of all the ensemble models has variance $\frac{\nu}{k} + c \frac{k-1}{k}$

If $c = 0$, the bagging reduces square error by a factor of k . If $c = \nu$, then no gains is achieved.

Typically, bagging involves constructing k different datasets. Each dataset has the same number of examples as the original dataset, but each dataset is constructed by sampling with replacement from the original dataset.

Model i is then trained on dataset i . The differences between which examples are included in each dataset result in differences between the trained models.

Bagging

In bagging:

- The classification probability of the ensemble of neural networks is given by the arithmetic mean of all the corresponding distributions
 - Model i produces the prediction probability as $p^{(i)}(y|x)$
 - Prediction of ensemble of k models is the arithmetic mean $\frac{1}{k} \sum_{i=1}^k p^{(i)}(y|x)$
 - This gives equal weight to ensemble elements predictions
 - It is possible to give unequal weights to ensemble elements giving $\sum_{i=1}^k w_i p^{(i)}(y|x)$ with $w_i \geq 0$ and $\sum_i w_i = 1$.
 - Use geometric mean rather than arithmetic mean of the ensemble member's predicted distributions is *intuitively equivalent* to ensemble log-likelihood optimization.

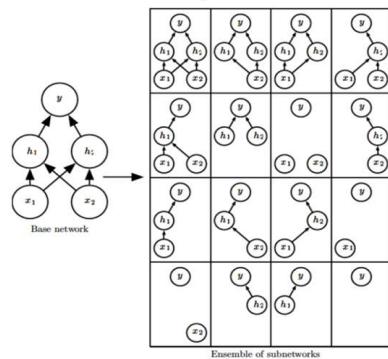
Bagging

Neural networks reach a wide enough variety of solution points that they can often benefit from model averaging even if all the models are trained on the same dataset.

Differences in random initialization, in random selection of mini-batches, or in outcomes of nondeterministic implementations of neural networks are often enough to cause different members of the ensemble to make partially independent errors.

This means that bagging can be very useful in neural networks.

Subnetworks Example



Dropout

Specifically, dropout trains the ensemble consisting of all subnetworks that can be formed by removing non-output units from an underlying base network.

Parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory.

Dropout aims to approximate the above, but with an exponentially large number of neural networks.

Note that in most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero.

Dropout

In bagging, each model is trained to converge on its respective training set.

In dropout, typically most models are not explicitly trained at all.

- A small fraction of the possible subnetworks trained for a single step.

Model parameters for subnetworks are shared. The parameter sharing causes the remaining subnetworks to perform well too.

Dropout

Specifically, to train with dropout, we use a minibatch-based learning algorithm that makes small steps, such as stochastic gradient descent. Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all the input and hidden units in the network.

- The mask for each unit is sampled independently from all the others.
- The probability of sampling a mask value of one (causing a unit to be included) is a hyper-parameter fixed before training begins.
- It is not a function of the current value of the model parameters or the input example.
- Typically, an input unit is included with probability 0.8, and a hidden unit is included with probability 0.5.

We then run forward propagation, back-propagation, and the learning update as usual.

Mathematical Model of Dropout

Denote μ as a mask vector

- which units to include and which ones to be removed

Denote $J(\theta, \mu)$ as the cost of the model prediction

Training with dropout consists of minimizing $\mathbb{E}_\mu J(\theta, \mu)$)

- Expected value contains exponential # of terms

One can get an unbiased estimate of its gradient by sampling values of μ

Mask for dropout training

Denote μ as a mask vector

- each subnetwork is defined by mask vector μ
 - μ determines which units to include and to remove

Hence, each subnetwork outputs a probability distribution $p(y | x, \mu)$.

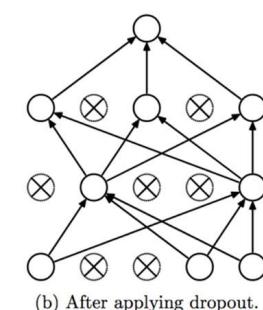
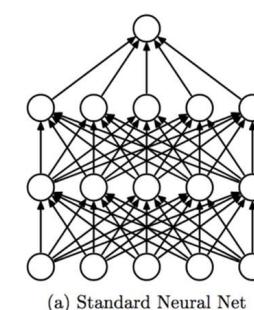
If subnetworks are chosen according to probability/weight $p(\mu)$, then the arithmetic mean **over all masks** is given $\sum_{i=1}^k p^{(i)} (y|x) p(\mu)$

$p(\mu)$ is the distribution used to sample μ at training time

Geometric mean is preferred (average $\log [p^{(i)} (y|x)]$).

This sum may include an exponential number of terms.

Visualization of Drop-out



Dropout

Approximate the sum using sampling

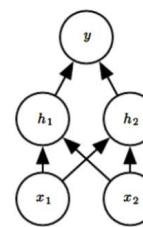
- 1 - By averaging together the output from many masks
 - 10-20 masks are sufficient for good performance
- 2 - Use geometric mean rather than arithmetic mean of the ensemble member's predicted distributions

$$p_{\text{ensemble}}(y|x) = \sqrt[d]{\prod_{\mu} p(y|x, \mu)}$$

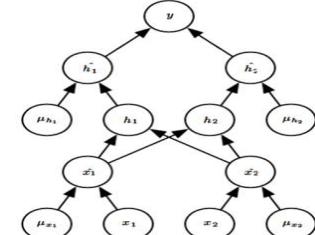
- 3- We have to make sure that none of the models assign probability zero to any event

Forward Propagation with Dropout

Base network



Forward propagation with dropout



μ is a random binary vector (mask) with one entry for each input

The probability of each entry being 1 is a hyper-parameter, usually 0.5 for the hidden layer, and 0.8 for the input.

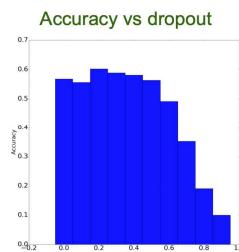
Each unit is multiplied by the corresponding mask μ

Equivalent to randomly selecting one of the subnetworks of previous slide

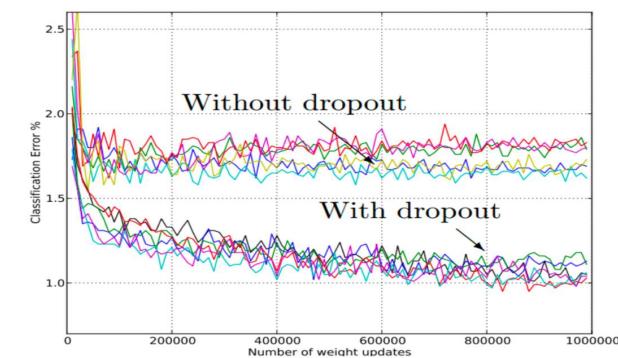
What is the effect of probability of drop

CIFAR-10 test dataset

- Three convolution layers of size 64, 128 and 256
- Followed by two densely connected layers of size 512
- output layer dense layer of size 10



Classification accuracy with or without Dropout



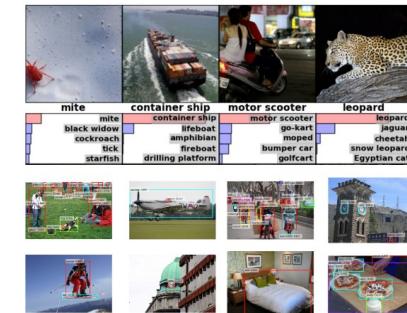
Convolutional Neural Networks

Applications to Object Detection

Vahid Tarokh
ECE685D, Fall 2025

Classification versus detection

- Very related, but different problems.
- Many aspects will be shared between the two problems
- In detection algorithms, we try to draw a bounding box around the object of interest to locate it within the image.
- There could be many bounding boxes representing different objects of interest within the image and you would not know how many beforehand.

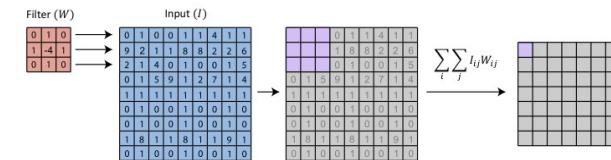


Pieces of a Deep Algorithm

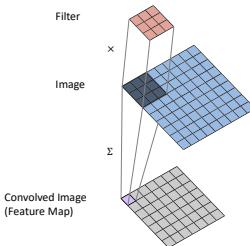
Object detection is very related to image classification. We use similar types of algorithms and related data types.

	Image Classification	Object Detection
Model	CNN Feature Extractor + Softmax Classifier	SSD, Faster R-CNN
Data	Imagenet, CIFAR-10, e.g.	COCO, PASCAL VOC
Scalar loss function (Objective)	CE loss + Regularization	SSD Loss, Faster R-CNN Loss, (both very complicated)
Optimization Algorithm and Hyperparameters	Stochastic Gradient Descent (SGD), Adam, Momentum, etc.	

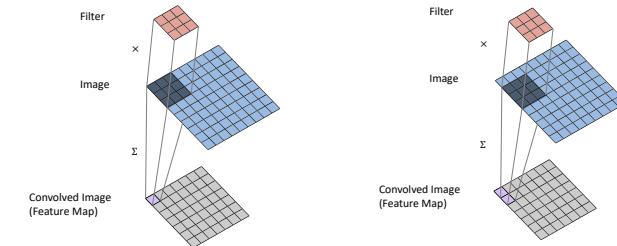
Review: 2D Convolution



Review: 2D Convolution



Review: 2D Convolution

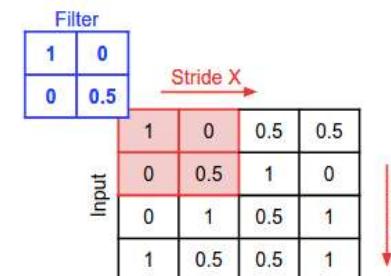


CNN Convolution Parameters

CNN — Parameters

- **Filters:** Represents the amount of filters in a CL.
- **Kernel Size:** Defines the dimensions of the filters.
- **Stride:** Sets the size of the filter shift step.
- **Padding:** defines whether or not there is entry zeroing, influencing the output dimensions:

CNN Convolution Parameters



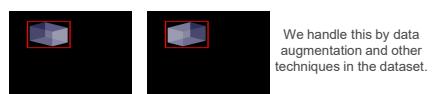
Review of Useful Properties

Recall that we have some important properties that are useful in object detection

- **Translation Invariance**



- **Rotation Invariance**



- **Scale Invariance**



Some are built into the algorithm, and some come from the structure of the dataset

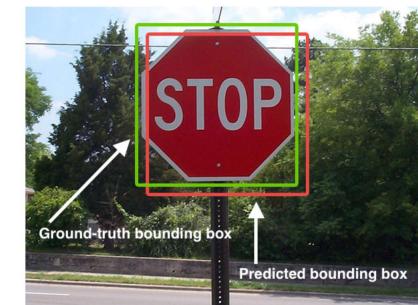
We need a metric to evaluate our model

Classification – did we choose the correct class?

Object detection – is our object correct?

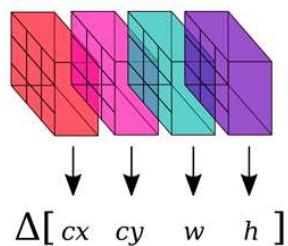
How do we determine if we got the correct location?

We want the bounding box to be good enough.



Detection and Regression

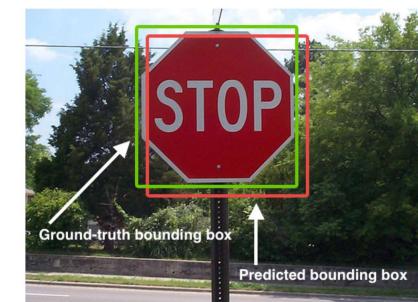
- Finding the bounding box can be thought of as a “**regression problem**”.
- Given the image, we must find the center coordinates, width, and height of the box.
- This means we need to output a 4-dimensional vector corresponding to the box for each object of interest.
- We need to understand what is in the box too.
- Need a measure of how good the box is.



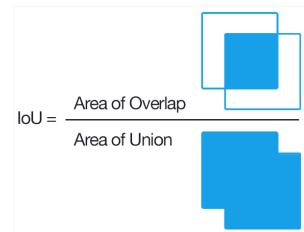
Intersection-over-Union

- The **intersection-over-union** metric is one way of defining this.
- Compute the **Intersection**-the common area covered by the ground-truth bounding box and the predicted bounding box.
- Compute the **Union**-the total area covered by either the ground-truth bounding box and predicted bounding box.

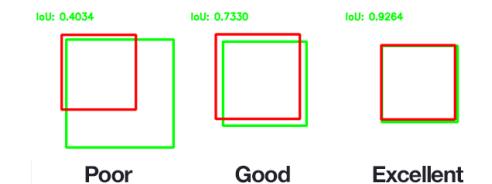
$$\text{IoU} = \frac{\text{Intersection}}{\text{Union}}$$



Intersection-over-Union



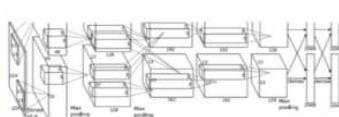
Intersection-over-Union



From Wikipedia

5

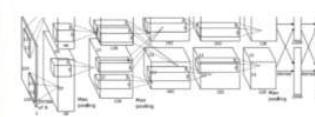
The idea



CAT: (x, y, w, h)

- Images are fed through a convolutional feature extractor.
- Bounding box **regression** and classification occur in one stage relative to a set of anchor boxes.
- This is a new step that we must figure out.

The idea



DUCK: (x, y, w, h)
DUCK: (x, y, w, h)
....

We might not necessarily draw just one bounding box in an object detection case, there could be many bounding boxes representing different objects of interest within the image and **we would not know how many beforehand**.

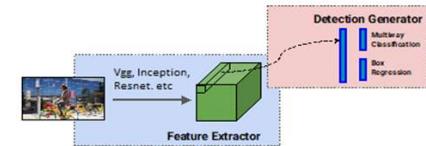
The Major Barrier to Overcome

- The major reason why you cannot proceed with this problem by building a standard convolutional network followed by a fully connected layer is that
- The length of the output layer is variable — not constant, this is because the number of occurrences of the objects of interest is not fixed.
- A naive approach to solve this problem would be to take different regions of interest from an image and use a CNN to classify the presence of the object within that region.
- The problem with this approach is that the objects of interest might have different spatial locations within the image and different aspect ratios. Hence, we would have to select a huge number of regions and this could computationally blow up.
- Therefore, algorithms like R-CNN, YOLO etc., have been developed to find these occurrences and find them fast.

FROM CONVOLUTIONAL FEATURES TO OBJECT DETECTION

Object Detection Algorithm

- Images are fed through a convolutional feature extractor.
- Bounding box regression and classification occur in one stage relative to a set of anchor boxes.
- This is a new step that we have to figure out.



R-CNN

- To bypass the problem of selecting a huge number of regions, Girshick proposed a method where a selective search is used to extract just 2000 regions from the image (These regions are called **region proposals**).
- Therefore, now, instead of trying to classify a huge number of regions, you can just work with 2000 regions.
- These 2000 region proposals are generated using the **selective search algorithm**:
- **Selective Search (High Level Description):**
 1. Generate initial sub-segmentation, we generate many candidate regions
 2. Use greedy algorithm to recursively combine similar regions into larger ones
 3. Use the generated regions to produce the final candidate region proposals

Details are given in

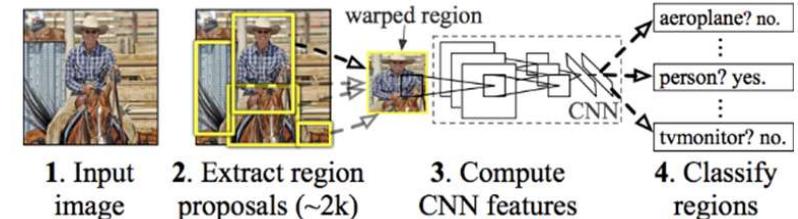
<https://vi.fnwi.uva.nl/isis/publications/2013/UijlingsIJCV2013/UijlingsIJCV2013.pdf>

R-CNN

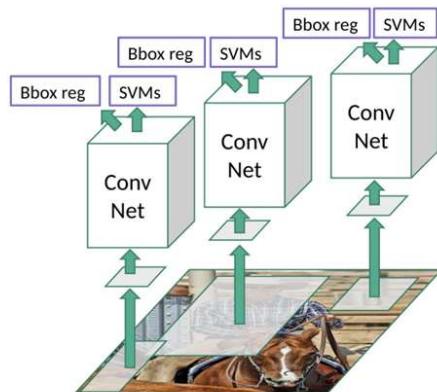
- These 2000 candidate region proposals are warped into a square and fed into a convolutional neural network that produces a 4096-dimensional feature vector as output.
- The CNN acts as a feature extractor and the output dense layer consists of the features extracted from the image and the extracted features are fed into a support vector machine (SVM) to classify the presence of the object within that candidate region proposal.
- In addition to predicting the presence of an object within the region proposals, the algorithm also predicts **four** values which are offset values to increase the precision of the bounding box.
- For example, given a region proposal, the algorithm would have predicted the presence of a person but the face of that person within that region proposal could've been cut in half.
- Therefore, the offset values help in adjusting the bounding box of the region proposal.

R-CNN

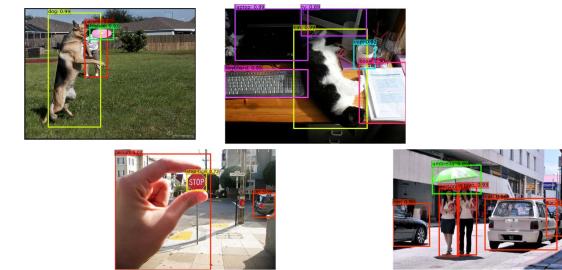
R-CNN: Regions with CNN features



R-CNN



Sample Detections



R-CNN

- It still takes a huge amount of time to train the network as you would have to classify 2000 region proposals per image.
- It cannot be implemented real time as it takes around 47 seconds for each test image.
- The **selective search algorithm** is a **fixed algorithm**. Therefore, no learning is happening at that stage. This could lead to the generation of bad candidate region proposals.

FAST R-CNN

- The approach is similar to the R-CNN algorithm.
- Instead of feeding the region proposals to the CNN, we feed the input image to the CNN to generate a convolutional feature map.
- From the convolutional feature map, we identify the region of proposals and warp them into squares and by using a Region of Interest (RoI) pooling layer we reshape them into a fixed size so that it can be fed into a fully connected layer.

VGG16: Very Deep Convolutional Networks for Large-Scale Image Classification"

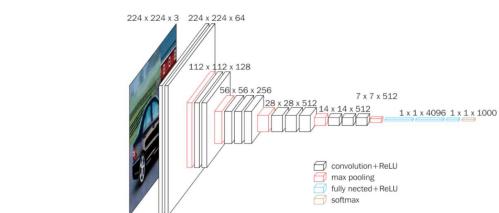
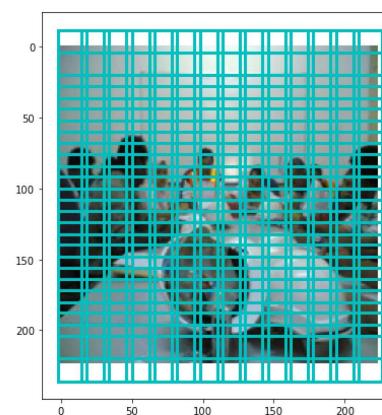


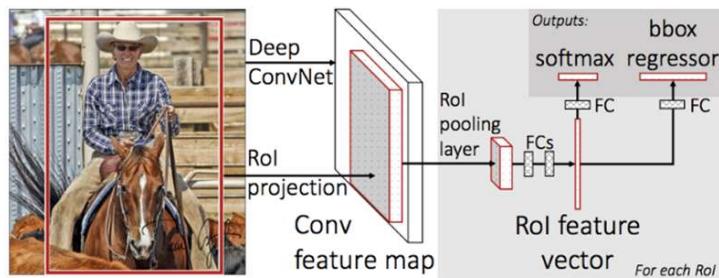
Image copyright Simonyan & Zisserman, 2015

- Input normalized to 224×224 pixels, 3 color channels.
- Last convolutional layer is 14×14 pixels, 512 channels. Call this $\vec{f}[m, n]$, where $\vec{f} \in \mathbb{R}^{512}$, $0 \leq (m, n) \leq 13$.
- Output FCN trained for object recognition: 1000 different object types.

Last convolutional layer contains 196 features



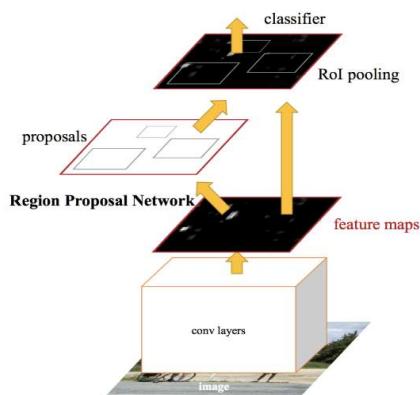
Fast R-CNN



FASTER R-CNN

- Both R-CNN & Fast R-CNN use selective search to find out the region proposals.
- Selective search is a slow and time-consuming process affecting the performance of the network.
- Ren et al, came up with an **object detection algorithm that eliminates the selective search algorithm** and lets the network learn the region proposals.

Faster R-CNN



FASTER R-CNN

- Similar to Fast R-CNN, the image is provided as an input to a convolutional network which provides a convolutional feature map.
- Instead of using selective search algorithm on the feature map to identify the region proposals, **a separate network is used to predict the region proposals**.
- The predicted region proposals are then reshaped using a ROI pooling layer.
- This is then used to classify the image within the proposed region and predict the offset values for the bounding boxes.

VGG16: Very Deep Convolutional Networks for Large-Scale Image Classification"

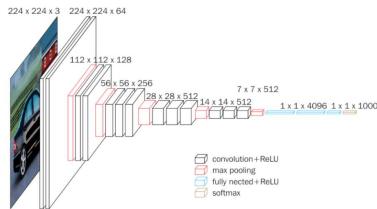


Image copyright Simonyan & Zisserman, 2015

- Input normalized to 224×224 pixels, 3 color channels.
- Last convolutional layer is 14×14 pixels, 512 channels. Call this $\vec{f}[m, n]$, where $\vec{f} \in \mathbb{R}^{512}$, $0 \leq (m, n) \leq 13$.
- Output FCN trained for object recognition: 1000 different object types.

- Faster RCNN assumes that the original image is 1064×1064 pixels, which is then downsampled to the 224×224 -pixel size required as input to VGG16.
- There are 4 layers of max pooling before the last conv layer, so each feature vector in the last conv layer represents

$$\left(2^4 \left(\frac{1064}{224}\right)\right) \times \left(2^4 \left(\frac{1064}{224}\right)\right) = 76 \times 76 \frac{\text{input pixels}}{\text{feature vector}}$$

- The last conv layer contains

$$\left(\frac{224}{2^4}\right) \times \left(\frac{224}{2^4}\right) = 14 \times 14 = 196 \text{ feature vectors.}$$

Last convolutional layer contains 196 features

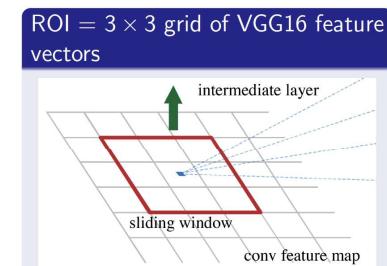
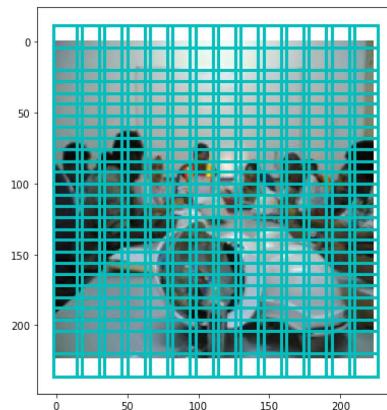


Image copyright Ren, He, Girshick & Sun, 2016

The region proposal network takes, as input, the concatenation of nine neighboring feature vectors from the VGG16 layer:

$$\vec{x}_{m,n} = \begin{bmatrix} \vec{f}[m-1, n-1] \\ \vec{f}[m-1, n] \\ \vdots \\ \vec{f}[m+1, n+1] \end{bmatrix}$$

Notice, we could think of this as another convolutional layer, but Ren et al. treat it as $14 \times 14 = 196$ different FCNs.

Features and Original Image

The $(m, n)^{\text{th}}$ feature vector, $\vec{f}_{m,n}$, covers a particular block of pixels in the input image:

$$(x_{ROI}, y_{ROI}, w_{ROI}, h_{ROI}) = (76n, 76m, 228, 228)$$

- Each $\vec{x}[m, n]$ covers 76×76 input pixels.
- Each $\vec{f}_{m,n}$ is $(3 \cdot 76) \times (3 \cdot 76) = 228 \times 228$.
- $m \rightarrow y$ is the vertical axis, $n \rightarrow x$ horizontal.

Suppose the nearest true object is in rectangle $(x_{REF}, y_{REF}, w_{REF}, h_{REF})$. We want to somehow encode the difference between where we are now $(x_{ROI}, y_{ROI}, w_{ROI}, h_{ROI})$ and where we want to be $(x_{REF}, y_{REF}, w_{REF}, h_{REF})$. Fast RCNN does this using the following target vector, \vec{y}_r , for the neural network:

$$\vec{y}_r = \begin{bmatrix} \frac{x_{REF} - x_{ROI}}{w_{ROI}} \\ \frac{y_{REF} - y_{ROI}}{h_{ROI}} \\ \ln\left(\frac{w_{REF}}{w_{ROI}}\right) \\ \ln\left(\frac{h_{REF}}{h_{ROI}}\right) \end{bmatrix}$$

The neural net is trained to find a \hat{y}_r that is as close as possible to \vec{y}_r (minimum MSE).

Training a bounding box regression network

The network is now trained with two different outputs, \hat{y}_c and \hat{y}_r .

The total loss is

$$\mathcal{L} = \mathcal{L}_c + \mathcal{L}_r$$

where \mathcal{L}_c is BCE for the classifier output:

$$\mathcal{L}_c = -\frac{1}{n} \sum_{i=1}^n (y_{c,i} \ln \hat{y}_{c,i} + (1 - y_{c,i}) \ln(1 - \hat{y}_{c,i}))$$

and \mathcal{L}_r is zero if $y_c = 0$ (no object present), and MSE if $y_c = 1$:

$$\mathcal{L}_r = \frac{1}{2n} \sum_{i=1}^n y_{c,i} \|\vec{y}_{r,i} - \hat{y}_{r,i}\|^2$$

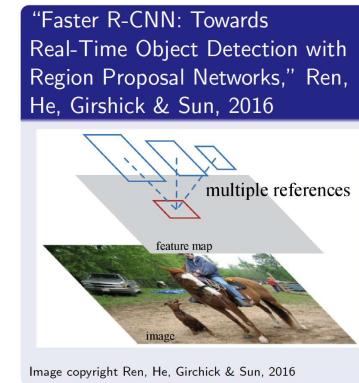
Summary of Approach

- An ROI network has a 4608d input, corresponding to a 3×3 grid of 512d feature vectors from the last conv layer of a VGG16 object recognizer.
- Faster-RCNN defines 9 different anchors centered on each ROI.
- W.r.t. each anchor, we define the classification target $y_c = 1$ if $IOU > 0.7$, otherwise $y_c = 0$.
- If $y_c = 1$, then we define a regression target \vec{y}_r , specifying how much the REF bbox differs from the anchor.

9 anchors per ROI

3 sizes, 3 aspect ratios
The Faster RCNN paper described 9 anchors per ROI:

- 3 different anchor sizes: 128×128 , 256×256 , and 512×512 .
- 3 different aspect ratios: $1 : 2$, $1 : 1$, and $2 : 1$



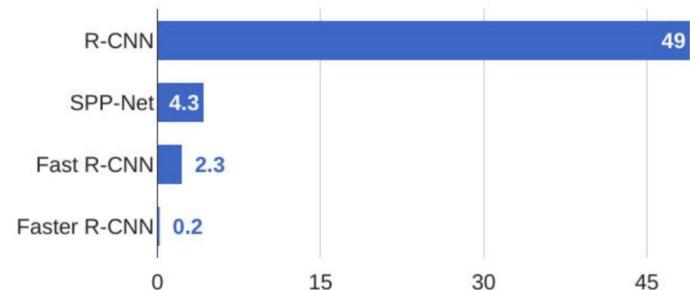
- Each candidate bounding box computes 9 different regression outputs, each of which is a 4-vector (x, y, w, h)
 - The 9 different regression outputs from each bbox are w.r.t. 9 different “anchor” rectangles, each offset from the input ROI. Thus:
- anchor = ROI + known shift
object = anchor + regression

Speeds

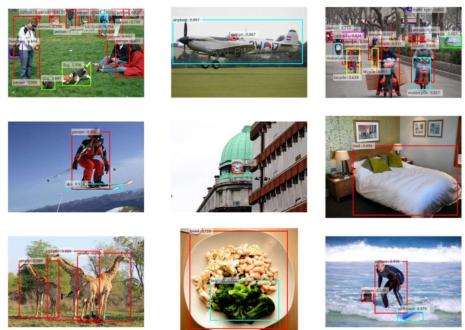
- The ROI is $(x_{ROI}, y_{ROI}, w_{ROI}, h_{ROI})$.
- The anchor is (x_a, y_a, w_a, h_a) .
- The true object is located at $(x_{REF}, y_{REF}, w_{REF}, h_{REF})$.
- The regression target is:

$$\vec{y}_r = \begin{bmatrix} \frac{x_{REF} - x_a}{w_a} \\ \frac{y_{REF} - y_a}{h_a} \\ \ln\left(\frac{w_{REF}}{w_a}\right) \\ \ln\left(\frac{h_{REF}}{h_a}\right) \end{bmatrix}$$

R-CNN Test-Time Speed



Sample Detections



How can this be learned?

- Object detection adds a *lot* of complexity compared to a CNN
- Many practical details become important for effective training
- Please read the papers if you go to implement this yourself

Let's go back to our goals:

We have some important properties that are useful in object detection

- Translation Invariance
- Scale Invariance
- Rotation Invariance

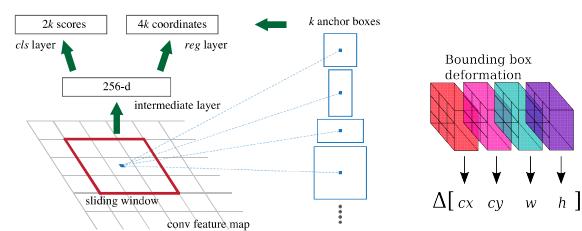
Some are built into the algorithm, and some come from the structure of the dataset

Translation Invariance

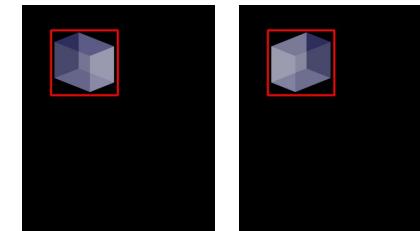


This is addressed with convolutional features...

Translation invariance
comes from the sliding windows (same as conv.)

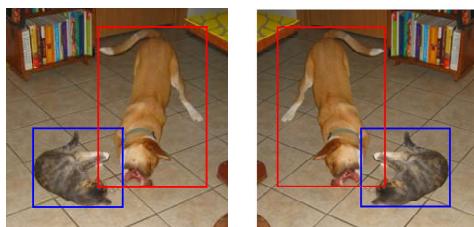


Rotation Invariance



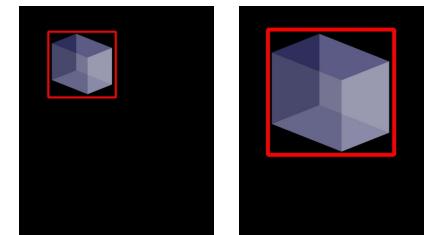
handled by data augmentation and other techniques in the dataset.

Data Augmentation



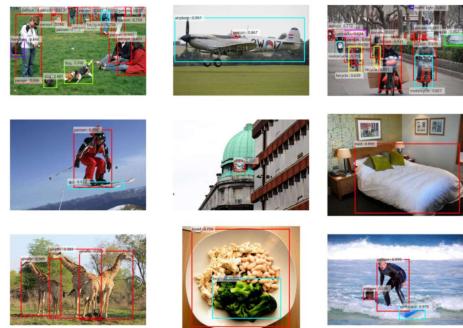
Feed in mirrors, rotations, shifts, etc.

Scale Invariance



handled by data diversity and rescaling feature maps.

Sample Detections



How do we measure performance

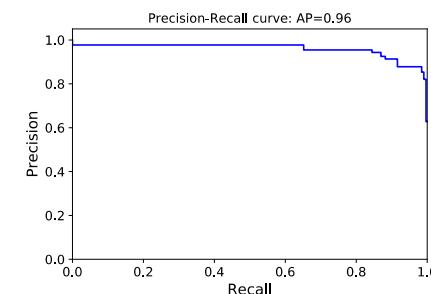
- We use mean average precision (mAP) over all our classes
- Must use a precision-recall metric *because there are so many ways to get things wrong*

Types of Predictions

	Predict Negative	Predict Positive
True Label is Positive	False Negatives (FN)	True Positives (TP)
True Label is Negative	True Negatives (TN)	False Positives (FP)

Precision is given by $(TP)/(TP+FP)$.
Recall is given by $(TP)/(TP+FN)$.

Precision-Recall

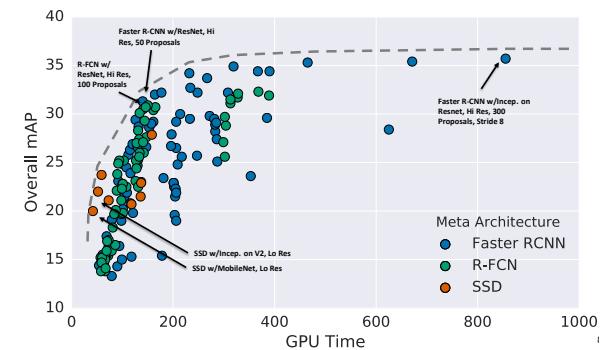


Average Precision is the integral of this curve.

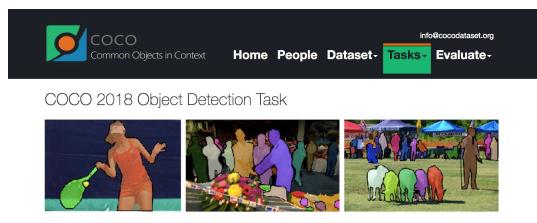
Mean Average Precision

- Average Precision (AP) isn't perfect. Nevertheless, it works well for a single class.
- When we have multiple classes, we can take the "Mean Average Precision," which is the average precision over multiple classes:
- mAP (or MAP) = mean (AP for each class).
- Other common metrics include:
$$F_1 = 2 \frac{(Precision * Recall)}{(Precision + Recall)}$$
- MRR = mean reciprocal rank (useful when only one answer is appropriate out of many)

Performance Tradeoffs



Where is object detection going?



<http://cocodataset.org/#detection-2018>

Resources for training your own object detector

Object detection in PyTorch
<https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Object-Detection>

