Temporal Learning, Modeling and Adaptation

2020

José Dorronsoro
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contents

1 Time Series Review

1.1 Stochastic Processes

Temporal Information Processing?

- First interpretation: the **processing of information that has a time structure**, i.e., time series
- Time series (TS): a time–ordered sequence of scalar or vector values X_t
- The temporal structure determines the behavior of X_t and must be taken into account to model it
- Second interpretation: the **temporal processing of information**
- Even if they do not have a temporal structure, data are (almost always) generated sequentially
- Examples: streaming data, on–line learning
- In both cases we get back to Machine Learning (although with different objectives)
  - Clearly so in on–line learning
  - After some roundabouts in practical TS modelling
- We deal first with time structured data

Time Series Contexts

- Two different TS origins:
  - **Stochastic generation**: the observed TS is a realization of a **stochastic process**
  - **Dynamical system evolution**: the observed TS is the trajectory of the solution of a continuous or discrete dynamical system (DS) from given initial conditions
- Different worlds (stochastic vs deterministic) but sometimes are hard to tell them apart
- Very different tools and perspectives
  - For stochastic TS we worry about **stationarity, ergodicity, spectral densities**; linear models and short term prediction are the main goals
  - For DS we worry about **sinks, sources, stability, attractors, chaos**; the main goal is non–linear long term behavior, often very sensible to initial conditions
- We begin with (general) stochastic processes and then briefly review the (much simpler) ARMA models

Basic Tools

- Probability Space: triplet made up of a set Ω, a subset $\mathcal{A}$ of 2^Ω and a probability P defined in $\mathcal{A}$

- We associate $\mathcal{A}$ with the events to which we can assign a probability
- If Ω is discrete, usually $\mathcal{A} = 2^\Omega$ (i.e., all possible subsets of Ω)

- Two subsets $A, B \in \mathcal{A}$ are **independent** if $P(A \cap B) = P(A)P(B)$
- Random variable: a function $X : \Omega \to \mathbf{R}$
  - As such X is deterministic; it is its arguments that are random
- Mean of X: $\overline{X} = \mu_x = E_P[X]$
- If $\mu_X < \infty$, its variance is $var(X) = E_P[(X - \mu_X)^2] = E_P[X^2] - \mu_X^2$
- The order k moment of X is $E_P[X^k]$

Computing Expectations

- What do we mean by $E_P[X]$?
- In simple cases it is clear:
  - If Ω discrete, $E_P[X] = \sum X(\omega)P(\{\omega\})$
  - If X takes discrete values $\{x_n\}$, $E_P[X] = \sum x_n P(\{\omega : X(\omega) = x_n\})$
- In general, one defines the expectation of X in terms of the **distribution function** of X

$$F(x) = F_X(x) = P(\{\omega : X(\omega) \le x\})$$

  - Clearly $F(x)$ is increasing
  - If x is one–dimensional and F is derivable, $F'(x) = f(x)$ is its **density**
- Then for a general one–dimensional continuous X, $E[X]$ is defined as a **Stieltjes integral** of x with respect to F: $E[X] = \int x dF_X(x) = \int x dF(x)$
  - If F is differentiable, $E[X] = \int x f(x) dx$
  - We can also define $E_X[g(X)] = E[g(X)] = \int g(x) dF(x)$

Joint Distribution Function

- Given two r.v. X, Y their **joint distribution** $F(x, y)$ is defined as

$$\begin{aligned} F(x,y) &= F_{X,Y}(x,y) = P(\{\omega : X(\omega) \le x, Y(\omega) \le y\}) \\ &= P(\{X(\omega) \le x\} \cap \{Y(\omega) \le y\}) \end{aligned}$$

  - Then $\int_{y=-\infty}^{\infty} dF(x,y) = F(x, \infty) = P(\{\omega : X(\omega) \le x\}) = F_X(x)$
  - If $F(x, y)$ is differentiable, $\frac{\partial^2 F}{\partial x \partial y} = f(x, y)$ is the joint density
- We can define $E_{X,Y}[g(X, Y)] = \int g(x, y) dF(x, y)$ also as a Stieltjes integral
  - If $F(x, y)$ differentiable, $\int g(x, y) dF(x, y) = \int g(x, y) f(x, y) dx dy$

- X, Y are said to be **independent** if the subsets $\{x_1 \leq x \leq x_2\}$ and $\{y_1 \leq y \leq y_2\}$ are independent
  - Then $F(x, y) = F_X(x)F_Y(y)$
- Similarly, if $X_1, \dots X_K$ are random variables, we can define $F(x_1, \dots x_K) = P(\{\omega : X_k(\omega) \leq x_k,\ k = 1, \dots, K\})$

Stochastic Processes

- A **stochastic/random process** (SP) is a family $\{X_t\}$ of random variables on a common probability space $(\Omega, \mathcal{A}, P)$
  - We will consider discrete time: $\{t\} = \{\dots, -1, 0, 1, \dots\}$
  - If we fix a $\omega_0 \in \Omega$, a **sample path** or **realization** of a SP is the sequence $\{x_t = X_t(\omega_0)\}$
- Examples of SPs
  - **White noise**: X_t are independent variables with 0 mean and finite variance
  - **Brownian Motion**: $X_0 \equiv 0$, the increments $X_t - X_s$ are independent, $X_t - X_s = N(0, \sqrt{t-s}\ I)$
  - **Markov models**: $\Omega = \{1, \dots, N\}$ is discrete and $P(X_{t+1} = j | X_t = i, X_{t-1} = i_{t-1}, \dots, X_{t-k} = i_{t-k}) = P(X_{t+1} = j | X_t = i)$
    - $p_{ij} = P(X_{t+1} = j | X_t = i)$ is the **transition** matrix
- Kolmogorov's Theorem ensures the existence of underlying SPs
- But if the different X_t behave differently, it will be difficult to say much about them

Stationarity

- We can define the joint distributions $F_\tau(x_1, \dots, x_K)$ for $\tau = (t_1, \dots, t_K)$ of the SP X_t as $F_\tau(x_1, \dots, x_K) = P(\{\omega : X_{t_k}(\omega) \leq x_k,\ k = 1, \dots, K\})$
- The SP X_t is **strictly stationary** (SS) if for all K, τ and h we have for $\tau + h = (t_1 + h, \dots, t_K + h)$
$$F_\tau(x_1, \dots, x_K) = F_{\tau+h}(x_1, \dots, x_K)$$
- If X_t is SS, $\mu_t = \int x dF_t(x) = \int x dF_{t+h}(x) = \mu_{t+h}\ \forall h$
- The **autocovariances** of X_t are
$$\begin{aligned}\gamma(r, s) &= cov(X_r, X_s) = E[(X_r - \mu_r)(X_s - \mu_s)] \\ &= \int (x - \mu_r)(x' - \mu_s) dF_{r,s}(x, x'),\end{aligned}$$
- If X_t is SS, $\gamma(r, s) = \gamma(r + h, s + h) = \gamma(r - s, 0)\ \forall r, s, h,$

Stationarity II

- Also $\gamma(r, s) = \gamma(s, r)$
- SS is very desirable, but perhaps too restrictive
- We relax it to just **stationarity** (S) (or weakly/second order stationarity) if we simply impose
$$\mu_t = \mu, \ \ \gamma(t+h, t) = \gamma(h, 0) \ \ \forall t, h$$
  We just write $\gamma(h)$ instead of $\gamma(h, 0)$
- In particular $\gamma(0) = var[X_t]$ for all t
- Moreover, if $\mu_t = 0$, $\gamma(h) \leq \gamma(0)$
- If we define the **autocorrelations** $\rho(h) = \gamma(h)/\gamma(0)$, we have $\rho(h) \leq 1$

Ergodicity

- The time series $\{x_t\}$ given as a realization of a S SP X_t is the only information that we have if we want to compute the statistics of all X_t
- Ergodicity makes possible to estimate moments of X_t from the time series values
- An intuitive idea is to estimate μ by the mean $\hat{\mu}_k = \frac{1}{2k+1} \sum_{-k}^{k} x_t$, for considering the RV $M_K = \frac{1}{2k+1} \sum_{-k}^{k} X_t$, we have
$$E[M_K] = \frac{1}{2k+1} \sum_{-k}^{k} E[X_t] = \frac{1}{2k+1} \sum_{-k}^{k} \mu = \mu$$
- If the variance σ_k^2 of $\hat{\mu}_k$ tends to 0, $\hat{\mu}_k$ **tends to μ in MSE** and we say that X_t is **mean–ergodic**
- **Slutsky's theorem**: A S SP X_t with covariance $\gamma(k)$ it is mean–ergodic iff $\lim \frac{1}{k+1} \sum_{0}^{k} \gamma(j) = 0$

Covariance Ergodicity

- Assuming $\mu = 0$, the intuitive variance estimation is now
$$\hat{v}_k = \frac{1}{2k+1} \sum_{-k}^{k} x_t^2,$$
  which is the mean of the SP X_t^2
- Now if X_t is a S SP, so is X_t^2 and we can apply again Slutsky's theorem
- The covariance $\gamma^2(k)$ of X_t^2 is $\gamma^2(k) = E[X_k^2 X_0^2] - E[X_0^2]^2$
- The Slutsky's condition for **covariance ergodicity** is now $\lim \frac{1}{k+1} \sum_{0}^{k} \gamma^2(j) = 0$ or, equivalently,
$$\lim \frac{1}{k+1} \sum_{0}^{k} E[X_j^2 X_0^2] = E[X_0^2]^2$$

1.2 Basic TS Models

AR and MA Models

- An **autoregressive** (AR) model of order p is a SP X_t with 0 mean where $X_t = \sum_1^p \phi_j X_{t-j} + \epsilon_t$, with ϵ_t white noise with variance σ^2
- If B denotes the **time delay** operator $BX_t = X_{t-1}$, we can write the above as

$$\epsilon_t = X_t - \sum_1^p \phi_j X_{t-j} = (I - \sum_1^p \phi_j B^j)\ X_t = \Phi(B)\ X_t$$

- A **moving average** (MA) model of order q is a SP X_t where $X_t = \epsilon_t + \sum_1^q \theta_j \epsilon_{t-j}$, with ϵ_t again white noise
- Just as before, we can write a MA (q) SP X_t as

$$X_t = \epsilon_t + \sum_1^q \theta_j \epsilon_{t-j} = (I + \sum_1^q \theta_j B^j)\ \epsilon_t = \Theta(B)\ \epsilon_t$$

Stationarity of AR and MA Models

- It is easy to see that if X_t is a S SP, $Y_t = \sum_0^q \theta_j X_{t-j}$ is also a S SP
- Thus any MA (q) SP is S
- If the series $\sum_0^\infty |\theta_j|$ converges and X_t is a S SP, the SP $Y_t = \sum_0^\infty \theta_j X_{t-j} = \sum_0^\infty \theta_j B^j X_t$ is also S
- What about AR (p) processes?
- If X_t is $AR(1)$ we have $\epsilon_t = X_t - \phi X_{t-1} = (1-\phi B)X_t$ or, formally, $X_t = (1-\phi B)^{-1}\ \epsilon_t = \sum_0^\infty \phi^j B^j\ \epsilon_t$
- If $|\phi| < 1$ the series converges; thus an $AR(1)$ process is S if $|\phi| < 1$ (in fact iff)
- Notice that the root $1/\phi$ of $\Phi(z) = 1 - \phi z$ lies outside the unit circle
- For a general AR (p) SP X_t we have formally $X_t = (\Phi(B))^{-1}\epsilon_t$, and **we can invert $\Phi(B)$ into a convergent series if the polynomial $\Phi(z)$ has all its roots outside the unit circle**
- Thus an AR (p) process X_t is S if(f) $\Phi(z)$ has all its roots outside the unit circle

ARMA Models

- An **ARMA** (p, q) model is a SP X_t s.t.

$$X_t = \sum_1^p \phi_i X_{t-i} + \sum_1^q \theta_j \epsilon_{t-j} + \epsilon_t,$$

  with ϵ_t white noise

- We can rewrite the above as $\epsilon_t + \sum_1^q \theta_j \epsilon_{t-j} = X_t - \sum_1^p \phi_i X_{t-i}$, i.e.

$$\Theta(B)\epsilon_t = \Phi(B)X_t$$

  for some polynomials Φ, Θ

- Formally we have $X_t = \Phi(B)^{-1}\Theta(B)\epsilon_t$, which we can express as

$$X_t = \sum_0^\infty \gamma_j B^j \Theta(B)\epsilon_t = \sum_0^\infty \delta_i B^i \epsilon_t$$

  if $\Phi(z)$ has all its roots outside the unit circle

- Thus an ARMA (p, q) process X_t is S if(f) $\Phi(z)$ has all its roots outside the unit circle

Covariance Functions

- Q1: How easy is to identify an ARMA process?
  - Right now this a too general question
- Q2: are there simple ways to characterize ARMA processes?
  - Yes: through their covariances
- In fact, covariances are sort of a **signature** of S SPs
- **Theorem**: a function $K : Z \to C$ is the autocovariance function of a (possibly complex) S TS iff it is **Hermitian and semi–definite positive**, i.e.,
  - $K(h) = \overline{K(-h)}$ and
  - For any $n \geq 0$ and $a \in C^n$, $a^t K(n) a \geq 0$, where $K(n)$ is the $n \times n$ matrix $K(n)_{ij} = K(i - j)$
- We can thus focus our attention on Hermitian and semi–definite positive functions

Spectral Covariance Representation

- **Riesz–Herglotz Theorem**: a function $\gamma : Z \to C$ is hermitian and semi–definite positive (i.e., an autocovariance function) iff

$$\gamma(h) = \int_{-\pi}^{\pi} e^{ih\omega} dF(\omega)$$

  with F a right–continuous, non decreasing function on $[-\pi, \pi]$ with $F(-\pi) = 0$

- Such an F is called the **spectral distribution** of γ
- If we can write $F(\omega) = \int_{-\pi}^{\omega} f(u)du$, we say that f is the **spectral density** of γ and then $\gamma(h) = \int_{-\pi}^{\pi} e^{ih\omega} f(\omega)d\omega$
- Notice that then $f(\omega) = F'(\omega) \geq 0$

- With (considerable) more work we can arrive at a spectral representation of a S SP X_t

Spectral Densities

- Spectral densities are much easier to handle
- If $\gamma(h)$ is summable (i.e., $\sum_h |\gamma(h)| < \infty$), Fourier series theory implies that

$$f(\omega) = \frac{1}{2\pi} \sum_{-\infty}^{\infty} \gamma(h) e^{ih\omega}$$

- An immediate consequence is that an absolutely summable $\gamma(h)$ is the autocovariance function of a S TS X_t iff $f(\omega) = \sum_{-\infty}^{\infty} \gamma(h) e^{ih\omega} > 0$
- Moreover, if X_t is real, f is symmetric
- Thus, S TSs with "simple" γ should have spectral densities easy to compute
- For instance, if ϵ_t is white noise with variance σ^2, its spectral density is $\sigma^2/2\pi$
- This is also the case for MA and (with more work) AR processes

Autocovariances of MA Processes

- MA processes have the simplest autocovariances:

  If X_t is a zero–mean S SP whose autocovariances verify $\gamma(h) = 0$ if $|h| > q$, then it is a MA(q) process

- More generally, if $\sum_0^\infty |\theta_k| < \infty$,

$$X_t = \sum_0^\infty \theta_k \epsilon_{t-k}$$

  is called a MA(∞) process

- It is relatively easy to check that its autocovariances verify

$$\gamma(h) = \sigma^2 \sum_0^\infty \theta_j \theta_{j+|h|}$$

Autocovariances of AR Processes

- The situation is more complicated for AR(p) processes
- Their covariances cannot be written in closed form unless p is small
- Usually they are all non zero
- To get them, recall that if X_t is AR(p) and $\Phi(z)$ has all its roots outside the unit circle, then we can write $\Phi(B)X_t = \epsilon_t$

- Multiplying both sides by X_{t-k} and taking expectations, we can get recurrence relations for $\gamma(k)$
- Easy exercise: compute them for $X_t = \phi X_{t-1} + \epsilon_t$, $\phi < 1$
- Nevertheless, AR and MA spectral densities are simpler to find

Spect. Densities of AR and MA Processes

- If we have $Y_t = \sum_{-\infty}^{\infty} \psi_j Z_{t-j} = \sum_{-\infty}^{\infty} \psi_j B^j Z_t$, with ψ_j real and Z_t is S with zero mean and spectral density (spd) f_Z, then Y_t is S with spectral distribution

$$F_Y(\omega) = \int_{-\pi}^{\omega} \left| \sum_{-\infty}^{\infty} \psi_j e^{-iju} \right|^2 f_Z(u) du = \int_{-\pi}^{\omega} \left| \sum_{-\infty}^{\infty} \psi_j \left(e^{-iu}\right)^j \right|^2 f_Z(u) du$$

  and $f_Y(\omega) = F_Y'(\omega) = \left| \sum_{-\infty}^{\infty} \psi_j \left(e^{-i\omega}\right)^j \right|^2 f_Z(\omega)$

- Since for an AR(p) X_t we have $\epsilon_t = \Phi(B) X_t$, it follows that

$$\frac{\sigma^2}{2\pi} = |\Phi(e^{-i\omega})|^2 f_X(\omega) \Rightarrow f_X(\omega) = \frac{\sigma^2}{2\pi} \frac{1}{|\Phi(e^{-i\omega})|^2}$$

- And since for a MA(q) X_t we have $X_t = \Theta(B)\epsilon_t$, its spd is $f_X(\omega) = \frac{\sigma^2}{2\pi} |\Theta(e^{-i\omega})|^2$

Spectral Densities of ARMA Processes

- For an ARMA(p, q) X_t we have $Y_t = \Phi(B) X_t = \Theta(B)\epsilon_t$, and putting together the previous equalities, we get

$$f_Y(\omega) = |\Phi(e^{-i\omega})|^2 f_X(\omega) = \frac{\sigma^2}{2\pi} |\Theta(e^{-i\omega})|^2$$

- Working things out we arrive at a rational spectral density

$$f_X(\omega) = \frac{\sigma^2}{2\pi} \frac{|\Theta(e^{-i\omega})|^2}{|\Phi(e^{-i\omega})|^2} = \frac{\sigma^2}{2\pi} \left| \frac{\Theta(e^{-i\omega})}{\Phi(e^{-i\omega})} \right|^2$$

- Since rational functions (and polynomials) are dense in $C([-\pi, \pi])$, a process with a symmetric continuous spd can be approximated in an appropriate sense by ARMA(p, q) or MA(q) processes
- But this has more theoretical than practical interest

ARIMA Models

- In general, time series are not stationary, with a typical instance being SP of the form $X_t = m_t + s_t + Y_t$, with m_t the **trend**, s_t a (periodic) **seasonal** component and Y_t a S SP

- The removal of m_t is problem dependent, with a frequent choice being the application of some power Δ^d of the difference operator $\Delta X_t = \Delta_1 X_t = X_t - X_{t-1}$
- A seasonal component with period S (i.e., $s_t = s_{t+S}$) can be removed applying the operator Δ_S, i.e. $\Delta_S X_t = X_t - X_{t-S}$
- An ARIMA (p, d, q) model is a SP X_t such that $\Delta^d X_t$ is an ARMA (p, q) model
- A seasonal ARIMA $(p, d, q) \times (P, D, Q)_S$ SP X_t can be formally expressed as

$$\Phi(B^S)\Phi(B)\Delta_S^D \Delta^d(X_t) = \Theta(B^S)\Theta(B)\epsilon_t$$

ARX and NARX Models

- Thus things get progressively more complicated and drift towards **system identification**, i.e., to use statistical methods to build mathematical models of dynamical systems from measured data
- Moreover, there may be some other inputs U_t that we may want to incorporate to our model
- In an autoregressive with exogenous inputs (ARX) model we assume that our target X_t has the form

$$X_t = \sum_1^D \phi_j X_{t-j} + \sum_0^{D'} \theta_{j'} U_{t-j'} + \epsilon_t$$

- Issues such as stationarity or ergodicity start to fade ...
- In a non linear ARX (NARX) the target X_t is a non linear function of the $X_{t-j}, U_{t-j'}$:

$$X_t = \Phi(X_{t-1}, \dots, X_{t-D}, U_t, U_{t-1}, \dots, U_{t-D'}) + \epsilon_t$$

- Besides being reasonable by themselves, they also appear when studying dynamical systems

2 Dynamical Systems

2.1 Linear Differential Equations

From AR to Dynamical Systems

- If in an AR process we impose $\epsilon_t = 0$ we are left with the difference equation $x_t = \sum_1^p \alpha_j x_{t-j}$
- In general, we say that a system $X_t \in \mathbf{R}^d$ follows a discrete differential equation if $X(t+1) = x_{t+1} = F(x_t) = F(X(t))$
- In turn writing $X'(t) \simeq X(t+1) - X(t) = F(X(t)) - X(t) = G(X(t))$, we arrive to a system of differential equations $X' = G(X)$

- Differential equation systems have behind a rich (and sometimes difficult) theory
- Basic examples: linear systems

Linear Systems

- An **autonomous linear system** is given by

$$X' = AX, \quad X(0) = x_0 \tag{1}$$

  for a $d \times d$ matrix A
- Basic example: (homogeneous) harmonic oscillator $x'' + ax' + bx = 0$
- Setting $y = x'$ we have $x' = y$, $y' = -bx - ay$; that is, for $X = (x, y)^t$, we have

$$X' = \begin{pmatrix} 0 & 1 \\ -b & -a \end{pmatrix} X = AX$$

- The **exponential** of a matrix B is $e^B = \sum_0^\infty \frac{B^n}{n!}$
- The general solution of (1) when $X(0) = x_0$ is $X(t) = e^{tA} x_0$

The Linearity Principle

- The eigenanalysis of A is the basic tool to study linear systems:
  if $Av_0 = \lambda v_0$, then $V(t) = e^{\lambda t} v_0$ is a solution with $V(0) = v_0$
- Now assume that λ_1, λ_2 are distinct eigenvalues of A with eigenvectors v_1, v_2, and let $x_0 = \alpha_1 v_1 + \alpha_2 v_2$
- Then we can find the solution of $X' = AX$ with $X(0) = x_0$ by writing

$$X(t) = \alpha_1 e^{\lambda_1 t} v_1 + \alpha_2 e^{\lambda_2 t} v_2$$

- This is a particular case of the **Linearity Principle**:

  If $Y_1(t), Y_2(t)$ are solutions of $X' = AX$ and $Y_1(0), Y_2(0)$ are linearly independent, then $X(t) = \alpha Y_1(t) + \beta Y_2(t)$ is the unique solution that satisfies $X(0) = \alpha Y_1(0) + \beta Y_2(0)$
- Simplest linear systems: **planar** systems in $\mathbf{R}^2$

Higher Dimensional Systems

- In principle for any $d \times d$ matrix A, $e^{tA} x_0$ yields the (unique) solution of $X' = AX$ with $X(0) = x_0$
- However the eigenstructure of a general A is more complicated than in the planar case
- Simplest situation: A has d **distinct eigenvalues**
- Then A can be transformed as $M = TAT^{-1}$ into a matrix M made up of a diagonal block and a series of 2–dimensional diagonal blocks $\begin{pmatrix} \alpha & \beta \\ -\beta & \alpha \end{pmatrix}$

- The structure of M when there are repeated eigenvalues is given by the (more complicated) **Jordan form** of A
- However, matrices with distinct eigenvalues are what is to be expected:

  The subset of matrices with d distinct eigenvalues is an **open and dense** *subset of the set of $d \times d$ matrices*

Non Autonomous Linear Systems

- The general form of a non autonomous linear system is $X' = A(t)X$, with $A(t)$ a time varying $d \times d$ matrix
- A general discussion of such systems is not possible; a simpler situation is that of **forced linear system** (FS)

$$X' = AX + G(t),\ X(0) = x_0$$

- The time independent system $X' = AX$ is the **homogeneous equation** (HE)
- If we know a particular solution Z of the FS and X is a solution of the HE, $Y = Z + X$ is another solution of the FS with initial condition $X(0) + Z(0)$
- Conversely, if Y, Z are solutions of the FS, $X = Y - Z$ is a solution of the HE with $X(0) = Y(0) - Z(0)$
- Since $e^{tA}x_0$ gives the general solution of the HE, **it is enough to find a particular solution to the FS**

Variation of Parameters

- Given the FS $X' = AX + G(t),\ X(0) = x_0$, a first try to solve it is to guess a particular solution; this is the method of **undetermined coefficients**
- However such a guess is not usually easy; the method of **variation of parameters** yields a (theoretical) general solution
- The solution of the above FS for $X(0) = x_0$ is given by

$$X(t) = e^{tA}\left(x_0 + \int_0^t e^{-sA}G(s)ds\right)$$

- The difficulty is, of course, to compute the integral!!

2.2 Planar Systems

Planar Systems

- A planar system is an autonomous linear system in $\mathbf{R}^2$, that is a function $X(t) \in \mathbf{R}^2$ such that

$$X' = AX, \quad X(0) = x_0$$

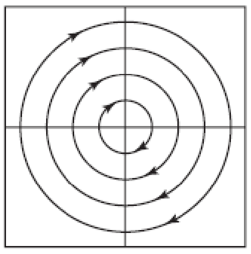
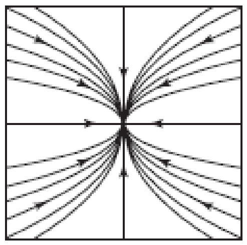
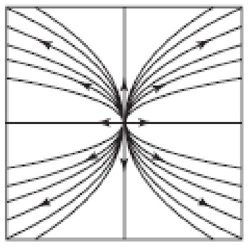
- Solution again given as $X(0) = x_0$ is $X(t) = e^{tA}x_0$
- If $A = \text{diag}\,(\lambda_1, \lambda_2)$ then
$$e^{tA} = \begin{pmatrix} e^{t\lambda_1} & 0 \\ 0 & e^{t\lambda_2} \end{pmatrix}$$
- Now A is a 2×2 matrix and its eigenanalysis is quite simple
- Thus, planar systems can be studied quite exhaustively

Phase Portraits

- The goal in the study of dynamical systems is often to understand their asymptotic behavior
- This is depicted using **phase portraits**
- The phase portrait of a planar system is a picture of a collection of representative solution curves in $\mathbf{R}^2$, which we call the **phase space**, for which a general idea of their evolution can be derived
- **Critical points** (and **equilibrium solutions**) arise when $0 = X' = AX$, with 0 the only critical point if det $A \neq 0$

Examples of Phase Portraits

- Here are some examples associated at particular planar systems

- We can arrive to a complete understanding of the behavior of planar systems mapping the eigenanalysis of A into phase portraits

Planar Systems in Canonical Form

- Eigen values of a 2×2 matrix A: either two distinct real eigenvalues, or two repeated real eigenvalues, or two complex conjugate eigenvalues
- A 2×2 matrix A is in **canonical form** if it has one of the following forms:
$$\begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}, \begin{pmatrix} \alpha & \beta \\ -\beta & \alpha \end{pmatrix}, \begin{pmatrix} \lambda & 1 \\ 0 & \lambda \end{pmatrix}$$
- We will split the analysis of these cases according to the eigenvalues of A:

  - A has two distinct real eigenvalues, i.e., the first matrix with $\lambda_1 \neq \lambda_2$
  - A has two complex conjugate eigenvalues, i.e., the second matrix
  - A has a single repeated real eigenvalue, i.e., the first matrix with $\lambda_1 = \lambda_2$ or the third matrix

- These cases will determine the limit behavior of the solutions of a general $X' = AX$

Real Distinct Eigenvalues I: Saddle Points

- If there are two non zero, distinct real eigenvalues λ_1, λ_2, we have

$$X(t) = \exp\left(t\begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}\right) x_0 = \begin{pmatrix} e^{\lambda_1 t} & 0 \\ 0 & e^{\lambda_2 t} \end{pmatrix} x_0 = \begin{pmatrix} \alpha e^{\lambda_1 t} \\ \beta e^{\lambda_2 t} \end{pmatrix}$$

- There are three important cases: i) $\lambda_1 < 0 < \lambda_2$, ii) $\lambda_1 < \lambda_2 < 0$, iii) $0 < \lambda_1 < \lambda_2$
- In the case $\lambda_1 < 0 < \lambda_2$,
  - The solutions $\alpha e^{\lambda_1 t}$ tend to 0 as $t \to \infty$: they lie in the **stable** line
  - The solutions $\beta e^{\lambda_2 t}$ tend away from 0 as $t \to \infty$: they lie in the **unstable** line
  - The solutions for $\alpha, \beta \neq 0$ tend to $\pm\infty$ getting closer to the unstable line
- The origin is the only equilibrium point, which we call a **saddle point**

Real Distinct Eigenvalues II: Sinks

- When $\lambda_1 < \lambda_2 < 0$, both solutions $\alpha e^{\lambda_1 t}, \beta e^{\lambda_2 t} \to 0$ when $t \to \infty$
- For a general solution $X(t) = \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \alpha e^{\lambda_1 t} \\ \beta e^{\lambda_2 t} \end{pmatrix}$, writing $x(t) = \alpha_1 e^{\lambda_1 t}$, $y(t) = \alpha_2 e^{\lambda_2 t}$, we have

$$\frac{dy}{dx} = \frac{y'}{x'} = \frac{\lambda_2 \alpha_2 e^{\lambda_2 t}}{\lambda_1 \alpha_1 e^{\lambda_1 t}} = \frac{\lambda_2 \alpha_2}{\lambda_1 \alpha_1} e^{(\lambda_2 - \lambda_1)t}$$

  which tends to $\pm\infty$ when $\alpha_2 \neq 0$
- Thus the trajectories **tend to 0 tangentially to the Y axis**
- The origin, again the only equilibrium point, is now called a **sink**

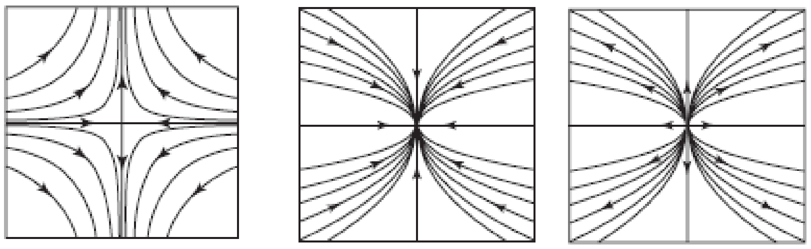
Real Distinct Eigenvalues III: Sources

- When $0 < \lambda_2 < \lambda_1$, a similar analysis yields
  - Both solutions $\alpha e^{\lambda_1 t}, \alpha e^{\lambda_2 t} \to \infty$ when $t \to \infty$
  - Writing as before $x(t) = \alpha_1 e^{\lambda_1 t}$, $y(t) = \alpha_2 e^{\lambda_2 t}$, we have that $\frac{y'}{x'}$ tends to 0 when $\alpha_2 \neq 0$
- Thus the trajectories **tend to ∞ away from 0 tangentially to the Y axis initially and becoming "horizontal"** as $t \to \infty$

- The origin, again the only equilibrium point, is now called a **source**
- When one of the eigenvalues, say $\lambda_1 = 0$, the X–axis defines an equilibrium line
- The other solutions tend to ∞ away from the axis if $\lambda_2 > 0$ or to 0 otherwise

Saddles, Sinks and Sources

- We depict saddles (left), sinks and sources (right) for planar systems in canonical form



From Hirsch et al., *Differential equations dynamical systems and an introduction to chaos.*

Complex Eigenvalues

- When $A = \begin{pmatrix} \alpha & \beta \\ -\beta & \alpha \end{pmatrix}$ the eigenvalues are $\alpha \pm i\beta$
- This yields two real solutions

$$e^{\alpha t}\begin{pmatrix} \cos\beta t \\ -\sin\beta t \end{pmatrix}, \quad e^{\alpha t}\begin{pmatrix} \sin\beta t \\ \cos\beta t \end{pmatrix}$$

  and the general solution

$$X(t) = c_1 e^{\alpha t}\begin{pmatrix} \cos\beta t \\ -\sin\beta t \end{pmatrix} + c_2 e^{\alpha t}\begin{pmatrix} \sin\beta t \\ \cos\beta t \end{pmatrix}$$

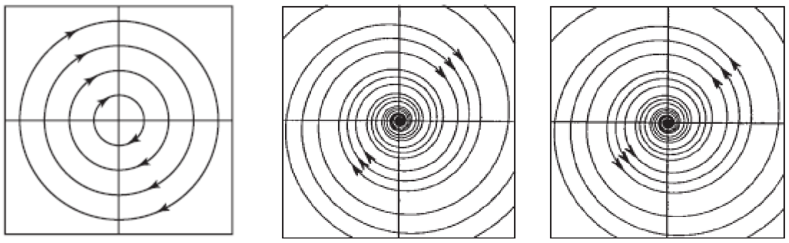
- If $\alpha = 0$, we have $\|X(t)\|^2 = c_1^2 + c_2^2$, i.e., the solutions **cycle** around 0
- If $\alpha \neq 0$, we get spirals that turn towards the origin when $\alpha < 0$ or away from it when $\alpha > 0$

  That is, we get **spiral sinks or sources**

Circles and Spiral Sinks and Sources

- We depict circles (left) and spiral sinks and sources (right) for planar systems in canonical form

Repeated Real Eigenvalue



From Hirsch et al., *Differential equations dynamical systems and an introduction to chaos.*

- We get a single repeated eigenvalue λ when we have

$$A = \begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix} \text{ or } A = \begin{pmatrix} \lambda & 1 \\ 0 & \lambda \end{pmatrix}$$

- In the first case the solutions are $X(t) = e^{\lambda t} V$ for any $V = X(0)$;
  thus the trajectories are straight lines through $(0, 0)$ that either tend to 0 when $\lambda < 0$ or to ∞
- The solutions in the second case are obtained by the method of **undetermined coefficients**: they are assumed of the form $x(t) = \alpha e^{\lambda t} + \mu t e^{\lambda t}$ for some α, μ
- Plugging this into the equation results in

$$X(t) = \alpha e^{\lambda t} \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \mu e^{\lambda t} \begin{pmatrix} t \\ 1 \end{pmatrix}$$

- If $\lambda < 0$ the solutions $\to 0$ as $t \to \infty$; if $\lambda > 0$ they $\to \infty$
- In either case the solutions tend toward or away from $(0, 0)$ in a direction tangent to $(1, 0)$

Changing Coordinates

- Assume we want to solve $X' = AX, X(0) = x_0$ for a general matrix A
- We can find an invertible matrix T such that $M = TAT^{-1}$ is in canonical form
- Let Y be a solution of $Y' = MY$ with $Y(0) = y_0 = Tx_0$; setting $X = T^{-1}Y$ we obtain a solution of $X' = AX$ with $X(0) = T^{-1}y_0$
- Conversely, $Y = TX$ converts solutions of $X' = AX$, $X(0) = x_0$ into solutions of $Y' = TAT^{-1}Y$ with $Y(0) = y_0$
- The transformation T changes the initial coordinates X into the canonical form coordinates $Y = TX$
- Therefore the phase portraits for general planar systems $X' = AX$ can be derived from the phase portraits of canonical form system $Y' = MY$ by applying the T^{-1} coordinate change

- We thus obtain equilibrium points, sinks, sources, cycles or spirals that correspond to appropriate coordinate changes of the ones in canonical form

The Trace–Determinant Plane I

- It can be easily seen that the eigenvalue equation of a 2×2 matrix A is of the form

$$\lambda^2 - \text{ tr } A \; \lambda + \text{ det } A = \lambda^2 - \tau\lambda + \delta = 0$$

with solutions $\lambda_\pm = \frac{1}{2}\left(\tau \pm \sqrt{\tau^2 - 4\delta}\right)$

- As a consequence we have
  - Complex conjugate eigenvalues when $\tau^2 < 4\delta$
  - Real different eigenvalues when $\tau^2 > 4\delta$
  - Real repeated eigenvalues when $\tau^2 = 4\delta$
- The parabola $\delta = \tau^2/4$ separates complex (above) from real (on or below) eigenvalues

The Trace–Determinant Plane II

- The real part of the complex eigenvalues above is $\tau/2$ and, therefore
  - We have a spiral sink if $\tau < 0$ and a spiral source when $\tau > 0$
  - We get a circle when $\tau = 0$
- Below the parabola we have a saddle when $\delta = \lambda_- \lambda_+ < 0$
- When $\delta > 0$, since $|\tau| > \sqrt{\tau^2 - 4\delta}$, we have sign $\lambda_\pm =$ sign τ; thus
  - We get a (real) source point when $\tau > 0$
  - We get a (real) sink point when $\tau < 0$
- One eigenvalue is 0 when $\delta = 0$ but $\tau \neq 0$ while both are 0 if $\delta = \tau = 0$

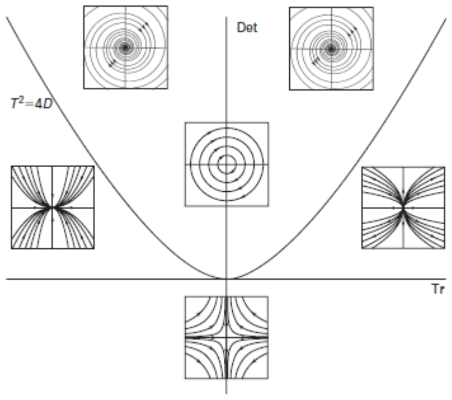
The Trace–Determinant Plane III

- The following plane diagram summarizes the preceding discussion

2.3 Nonlinear Dynamical Systems

Nonlinear Dynamical Systems

- A dynamical system (DS) is a procedure that describes the behavior in time of all points of a given space (Euclidean space, manifolds, ...)
  - They are characterized by the **flow**, a function $\Phi_t : \mathbf{R}^d \to \mathbf{R}^d$ that takes x into $x_t = X(t) = \Phi_t(x)$
  - We call them **discrete or continuous** depending on how we consider time change



From Hirsch et al., *Differential equations dynamical systems and an introduction to chaos.*

- Basic example: $\Phi_t(x)$ being the solution of a system of ordinary differential equations (ODE) $X' = F(X)$
- The behavior of general non linear DSs (NDS) may be quite complicated:
  - Most NDS are impossible to solve analytically
  - Some do not have solutions with a given initial value, some may have infinitely many ones
  - Solutions need not be defined for all time values t as they may tend to ∞ in finite time

Basic Results on ODEs

- **Existence and Uniqueness**: If F is C^1, given t_0 and x_0, there exists an $\epsilon > 0$ and a unique solution $X : (t_0 - \epsilon, t_0 + \epsilon) \to \mathbf{R}^d$ such that $X(t_0) = x_0$
  - Proved by the Picard iteration technique
  - We can show that we have a unique solution defined on a maximal time domain
  - However, the solution may not be defined for all t even for nice F
- **Continuous Dependence of Solutions**: If F is C^1 and $X(t)$ is a solution defined on $[t_0, t_1]$ with $X(t_0) = x_0$, then there is a neighborhood U of x_0 and a constant K such that if $y_0 \in U$, then there is a unique solution $Y(t)$ defined on $[t_0, t_1]$ with $Y(t_0) = y_0$ and for all $t \in [t_0, t_1]$

$$|Y(t) - X(t)| \leq K|y_0 - x_0|e^{K(t-t_0)}$$

  - In particular the flow $\Phi_t(x)$ is continuous in X
- **Continuous Dependence on Parameters**: If $X' = F(X, a)$ and F is C^1 on a and x, the flow $\Phi_t(X, a)$ depends continuously on a

Phenomena on Nonlinear DS

- This is almost as far as the general theory goes: many more tools have been developed but are often applicable only on concrete systems ...
- Moreover new issues and non standard behavior appear: **bifurcations, strange attractor, chaotic systems**, ...
- Chaos (Lorenz): *When the present determines the future, but the approximate present does not approximately determine the future*
- The Lorenz's system opened the way to the consideration of these phenomena

Lorenz's System I

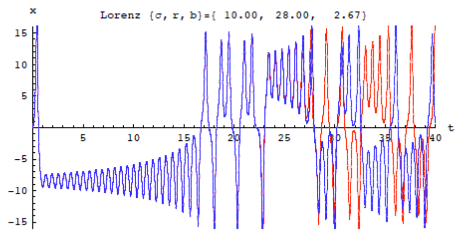
- First goal: a simple approximation to atmospheric flow that models as a two-dimensional fluid cell heated from below and cooled from above
- Further simplified to three independent variables: the rate of convectivity (x; **convection**: the process of heat transfer by a moving fluid), and the horizontal and vertical temperature variation (y and z, respectively)
- Equations: for parameters σ, b, r

$$\dot{x} = \sigma(y - x);\ \dot{y} = r\ x - y - x\ z;\ \dot{z} = x\ y - b\ z$$

- The asymptotic behavior is relatively simple in some cases
  - $r < 1$: all solutions of the Lorenz system tend to the equilibrium point at the origin or
  - $1 < r < r^* = \sigma\left(\frac{\sigma+b+3}{\sigma-b-1}\right)$: the two non–zero equilibrium points $Q_\pm$, i.e., the solutions of $F(Q) = 0$, are sinks

Lorenz's System II

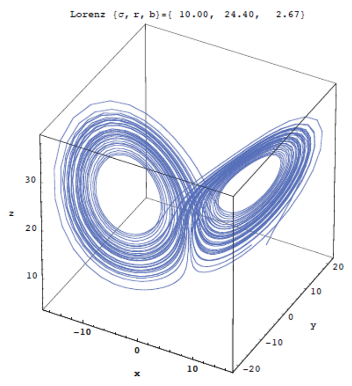
- Lorenz's significant parameters: $\sigma = 10$, $b = 8/3$, $r = 28$
- While being a deterministic system, it is very sensible to very small changes in initial conditions



- Butterfly effect: the flap of a butterfly's wing in Brazil can result in a tornado in Texas

Lorenz's Attractor

- All non–equilibrium solutions tend eventually to the so-called **Lorenz attractor**, roughly speaking an invariant set that "attracts" all nearby solutions



Discrete Dynamical Systems

- The theory of continuous DS focuses on the asymptotic behavior of solutions, assuming of course the system to be known
- If we are interested in (practical) prediction purposes we turn our attention to **discrete** DS (DDS)
- A Discrete Dynamical System is a pair $(\mathcal{X}, T)$ made up of the **state space** $\mathcal{X}$ (i.e., the set of all possible system states, that we assume bounded) and the map $T : \mathcal{X} \to \mathcal{X}$
- Starting at an $x_0 \in \mathcal{X}$ we get a **trajectory** or **orbit** $\{x_0, T(x_0), T^2(x_0), \ldots\}$ of the system
- A way to catch the behaviour of a DDS is to study the asymptotic behavior of orbits
- Long-term system properties are described in terms of **attractors**

Attractors of DDSs

- As for planar systems, the simplest cases are those of attracting points or cycles, but much more complicated attractors are possible
- In broad terms, we say that a compact set A is an **attractor** of a DDS with fundamental neighborhood U if
  - **Invariance**: for all $x \in A$ and all n, $T^n(x) \in A$
  - **Attractivity**: there is an open subset V s.t. $A \subset V$ and if $x \in V$, $T^n(x) \in V$ for all n and $\cap_n T^n(V) = A$
  - **Transitivity**: given any points $y_1, y_2 \in A$ and open neighborhoods U_j of y_j in U, there is a solution curve starting at U_1 and passing through U_2

- When we talk about evolution on the attractor, we actually mean in a neighborhood of the attractor

DS Reconstruction I

- In practice the dynamical system itself is rarely known and its study has to be done from a single orbit
- Moreover, instead of an orbit usually the most we can get is a time series of **measurements**

$$Y = \{y_0, y_1, y_2, \ldots\} = \{f(x_0), f(T(x_0)), f(T^2(x_0)), \ldots\}$$

  derived from a **read out map** $f : \mathcal{X} \to \mathbf{R}$
- In fact we often want a model $g : \mathbf{R}^k \to \mathbf{R}$ that helps us to predict the behavior of Y
- Q: Can we get it?

DS Reconstruction II

- At first sight we would need to know X to do any reconstruction
- So the first question could be:

  Can we reconstruct the internal state of the system from such a TS?
- But getting X is hopeless; we may at most answer another question:

  Can we get a somewhat equivalent representation of the internal state X from the TS?
- Tool: **reconstruction maps over time delays**
- For a fixed k define

$$R_k(x) = (f(x), f(T(x)), \ldots, f(T^{k-1}(x))$$

- Then $R_k(x_i) = (y_i, \ldots, y_{i+k-1})$

Takens Theorem

- Assume $\mathcal{X}$ is bounded and set $\mathcal{T} \times \mathcal{F}$ be the Cartesian product of the spaces of C^1 mappings T and C^1 readouts f; then

  There is an open dense subset $U \subset \mathcal{T} \times \mathcal{F}$ such that if $(T, f) \in U$ and $k > 2\ dim(\mathcal{X})$, the reconstruction map R_k is a C^1 embedding of $\mathcal{T}$ in $\mathbf{R}^k$ with a C^1 inverse
- The density of U implies that such an embedding exists "very near" any (T, f)
- Moreover, **the embedding preserves the structural properties of** T: the image $R_k(A)$ of an attractor A is an attractor (embedded) in $\mathbf{R}^k$ and the observed orbit has the "same properties" of the underlying one
- More importantly, we have a path to predict the next state of the time series

$$\begin{tikzcd}
\text{Current state} \arrow[r, "Det.rule", "T"'] \arrow[d, shift right, "\mathrm{Rec}_k"'] & \text{Next state} \arrow[d, "\mathrm{Rec}_k"'] \\
(y_i, \dots, y_{i+k-1}) \arrow[u, shift right, "\mathrm{Rec}_k^{-1}"'] \arrow[r, "Prediction", "\mathrm{Rec}_k \circ T \circ \mathrm{Rec}_k^{-1}"'] & (y_{i+1}, \dots, y_{i+k})
\end{tikzcd}$$

S. Laur, Time Series of Deterministic Dynamic Systems, 2004

From Takens Theorem to Predictions

- We have the following diagram
- Setting $G = R_k^{-1} \circ T \circ R_k$, we have

$$(y_{i+1}, \dots, y_{i+k-1}, y_{i+k}) = G(y_i, \dots, y_{i+k-1}),$$

  i.e., there is a function $g = G_1$ s.t. $y_{t+1} = g(y_t, \dots, y_{t-k+1})$ for all t

Time Series Prediction

- To exploit the preceding we need
  - A way to estimate an appropriate k
  - A way to estimate the function g
- The **correlation dimension** $cdim(A)$ of the attractor can be used to estimate an adequate k_0: since $cdim(A) = cdim(R_k(A))$, we may look for a k_0 after which $cdim(R_k(A))$ stabilizes
  - Often this is easier said than done!!
- In any case, we get back to non–linear regression problems and ...
- We may look to say, MLPs (o SVR) models to approximate the non–linear AR model $y_{i+k} = g(y_i, \dots, y_{i+k-1})$

Back to TS Prediction by ML

- Predicting temporal information coming from either a time series or a dynamical system ends up in building proper models over time delays
  - Quite often we can/have to add exogenous information that can improve our predictions
- We thus fall in a natural way into possibly non–linear regression models that we may try to build using ML models

  - The temporal structure should be preserved for instance when choosing train/test splits
  - But this structure is often no longer taken into account after that

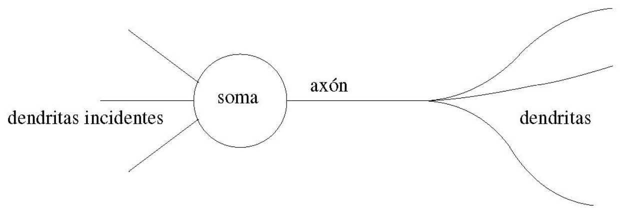
- ML models are usually built parameterizing the desired model as $f(x, w)$, defining a loss $L(w|S)$ over a sample S and minimizing L iteratively
- Handling temporal information comes thus back when actually training models
  - Because of sample patterns appearing sequentially
  - Or just in the iterative way model building proceeds
- We turn now to these issues

3 Machine Learning Basics

3.1 Rosenblatt's Perceptrons

Basic Neural Models

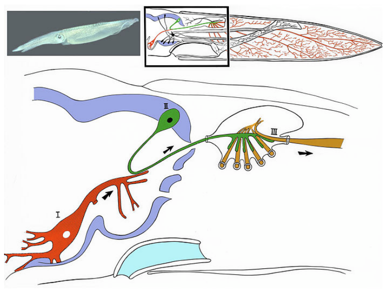
- Basic model: Ramón y Cajal's neuron (1900)



- Basic behavior: the neuron either fires or stays at rest depending basically on its inputs
- The brain has about 10^{11} neurons with each one having about 7,000 connections, often recurrent

Hodgkin–Huxley

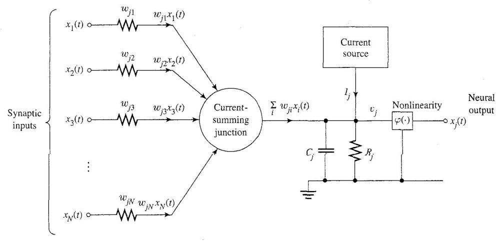
- They developed (circa 1935) the first model to describe the generation and propagation of electrical **action potentials** in neurons



From Wikipedia's Squid Giant Synapse

Electronic Neuron

- Electronic version: McCulloch–Pitts (1940)



  – Since $I = \frac{V}{R} = wV$, with w the conductance, the McC–P neuron output is

$$H\left(\sum_{j=1}^{N} w_j V_j + I\right) = V$$

  with the Heaviside function H ensuring a 0–1 output

- The **perceptron** is a mathematical formulation of the McC–P neuron where **the conductances are to be learned**

Rosenblatt's Perceptron

- Given a sample $S = \{(x^p, y^p)\}$ with $y^p = \pm 1$, Rosenblatt's Perceptrons (PCPs) are linear machines $w \cdot x$ such that

$$\begin{aligned} w \cdot x^p > 0 \quad & \text{if} \quad y^p = 1 \\ w \cdot x^p < 0 \quad & \text{if} \quad y^p = -1 \end{aligned}$$

- This can be seen as a **homogeneous** classification problem

- In more compact form we want for all p

$$y^p \; w \cdot x^p > 0$$

- If such a **separating** w exists, it can be computed in many ways
- Rosenblatt's goal was to **learn** w, i.e., to arrive at a separating w by repeatedly examining the (x^p, y^p) and adjusting w if necessary

Rosenblatt's Delta Rule

- The **Delta Rule** algorithm

```
Start with $w = 0$
While stopping condition not met:
   Get a new pattern $(x^p, y^p) = \left(x^{p(t)}, y^{p(t)}\right)$
   if $y^p w \cdot x^p \leq 0$ then:
      $w = w + y^p x^p$
```

- Writing w for the new vector and w' for the previous one, notice that

$$y^p \; w \cdot x^p = y^p \; w' \cdot x^p + \|x^p\|^2 > y^p \; w' \cdot x^p$$

Novikov's Theorem

- It states that if the sample S is linearly separable, the Delta rule yields a separating w in a finite number of steps
- More precisely, if for some separating hyperplane w with $\|w\| = 1$ we have $\min_p y^p w \cdot x^p = \gamma > 0$, then

  a separating w' can be found after at most $T = R^2/\gamma^2$ wrong classifications, where $R = \max_p \|x^p\|$

- Notice that γ is the margin of the w hyperplane. Thus, if

$$\gamma^* = \min_{\|w\|=1} m(w, S) = \min_{\|w\|=1} \min_p y^p w \cdot x^p$$

  is the **maximum margin**, we will need at most $R^2/(\gamma^*)^2$ iterations to train a Perceptron

- The maximum margin hyperplane gives the best (smallest) Novikov bound
- And problems with smaller margins require more iterations

Proof Sketch

- We consider only time–labelled wrong patterns, i.e., x^t is the t–th pattern such that $y^t w^{t-1} \cdot x^t \leq 0$

- We then have $\|w^t\|^2 \leq \|x^t\|^2 + \|w^{t-1}\|^2$, which leads to

$$\|w^t\|^2 \leq \|x^t\|^2 + \ldots + \|x^1\|^2 + \|w^0\|^2 \leq tR^2$$

  if $w^0 = 0$ and $\|x^p\| \leq R$ for all p

- Similarly, if w' is a 1–norm separating hyperplane i.e., $\min_p y^p w' \cdot x^p = \gamma > 0$ for all p, we have

$$w^t \cdot w' \geq \gamma + w^{t-1} \cdot w' \geq \ldots \geq t\gamma + w^0 \cdot w'$$

- Putting both together we have for all t for which x^t is erroneous

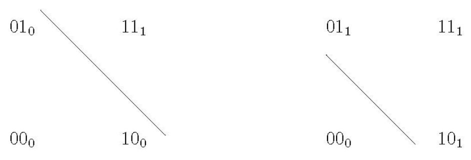
$$t\gamma \leq w^t \cdot w' \leq \|w^t\| \leq \sqrt{t}R$$

  which implies $t \leq R^2/\gamma^2$

- Thus, the Delta Rule can stop after at most R^2/γ^2 iterations

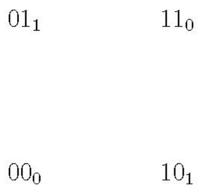
Problems with Perceptrons

- Observation 1: A PCP can learn (obviously) only linearly separable problems; for instance, it can learn the AND and OR predicates



(a) AND *(b)* OR

- But a PCP cannot learn the XOR predicate



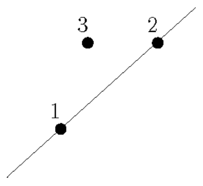
Dichotomies

- Observation 2: for PCPs to be useful, linearly separable problems must be frequent
- Q: how frequent are they?

- A dichotomy for $S = \left\{x^1, x^2, \ldots, x^N\right\}$ is any separation of S in two classes
- It corresponds to any asignment of the ± 1 values to the x^p
- The total number of dichotomies is 2^N
- Thus, the total number of 2-class problems over S is 2^N

Cover's Theorem

- The points in S are in general position if there are not $d+1$ patterns in a $d-1$–dimensional hyperplane



- **Theorem** If S is in general position, the number $L(N,d)$ of linearly separable dichotomies is

$$L(N,d) = \left\{ \begin{array}{cc} 2^N & \text{si } N \leq d+1 \\ 2\sum_{i=0}^{d} \binom{N-1}{i} & \text{si } N \geq d+1 \end{array} \right\}$$

Thus ...

- Notice that for d fixed,
$$\frac{L(N,d)}{2^N} \to 0$$
as $N \to \infty$
- In practice we can expect $N \gg d$ and the fraction of separable dichotomies will be very small
- Thus, linear PCPs will not be useful
- But a one hidden layer PCP can solve XOR and, in fact, separate any convex region
- And a two hidden layers PCP can separate any polyhedral region and hence solve any classification problem
- The same is essentially true for regression problems
- But: **no algorithm is known to learn these 2–hidden layer PCPs**

Why Not Try to Minimize Some Loss?

- Rosenblatt's Perceptrons are natural **online** machines:
  - Patterns appear sequentially and each may change the current weights
- We can also define a perceptron loss over a given sample as

$$L(w|S) = -\sum_{\{p: y^p w \cdot x^p < 0\}} y^p w \cdot x^p$$

- The local error would thus be $-y^p w \cdot x^p$ when $y^p w \cdot x^p < 0$ and 0 otherwise
- Since $\nabla_w(-y^p \ w \cdot x^p) = -y^p \ x^p$, the Delta Rule can be seen as a kind of local gradient descent on the loss $L(w|S)$
- However $L(w|S)$ is not differentiable and it is not clear what to make of $\nabla_w L(w|S)$ ...

What Can We Do?

- First option: work with linear models but somehow ensure that $d \gg N$
  - Have to overcome that in practice we usually have $N \gg d$
  - Transform the original d–dimensional features x in new D–dimensional ones such that $D \gg d$
  - This is the goal of kernel–based Support Vector Machines
- Second option: build trainable PCPs replacing the Heaviside function
  - Work with differentiable PCP transfer function
  - Transform PCP training into a differentiable optimization problem
  - This is what we will do next
- Rosenblatt's PCPs were the first Machine Learning model
- And modern ML started around 1985 along the PCP's ideas

3.2 Machine Learning Modeling Basics

What Is Machine Learning (ML)?

- Lofty definition: make machines learn!!!
  - Have to make "machines" and "learn" more precise
- The machines of ML: mathematical input–output processes that lend themselves to some form of (numerical) parameterization
- The learning process: adjust the machine's parameters until a goal is reached
- New thing: "goal"?

  - At first sight, get something done
  - Ultimately, to minimize some error measure

- Summing things up: a ML process tries to find a concrete mathematical/algorithmic **input–output parameterized transformation** that **minimizes an error measure** by iteratively **adjusting the transformation's parameters**

Where Lies ML?

- In the middle of a possibly long process chain
- Before ML starts we must
  - Go from **raw to organized** data: accesing, gathering, cleaning, formatting, ...
  - Go from **organized to** (potentially) **informative** data: extracting basic and derived features
- After ML finishes we must perform
  - Outcome **evaluation**: how good/actionable it is
  - Outcome **exploitation**: collect, organize, act
  - **Individual model maintenance**: monitor performance, tune hyper–parameters
  - **Modeling life cycle maintenance**: discard old models, introduce new ones and **communicate** our work/results

Supervised/Unsupervised Models

- ML model types: **supervised, unsupervised**
- Supervised models:
  - Targets y^p are known and the model tries to predict or estimate them
  - These known targets guide, or **supervise**, model building
  - Main emphasis here
- Unsupervised models:
  - There are no predetermined or supervising outputs
  - But nevertheless the model is supposed to learn relations or find structure in the data
  - Often as a first step towards a supervised model
- Semisupervised models lie in between: some patterns have targets, some don't

Regression and Classification

- Problems (usually) to be solved by ML models: regression, classification
- Patterns come in pairs (x, y)

  - x: inputs, predictors, features, independent variables
  - y: target, response, dependent variable; numerical in regression, class labels in classification

- **Regression**: the desired output y is **regressed** into the inputs x to derive a model $\widehat{y} = f(x)$
  - We want $y \simeq \widehat{y}$ so having $y - \widehat{y}$ "small" is the natural goal
- **Classification**: inputs are derived from several classes $C_1, \ldots, C_K$, to which labels ℓ_k are assigned
  - The model now assigns a label $\ell(x)$ to an input x
  - If x is derived from C_k we want to have $\ell(x) = \ell_k$
  - Here having $\ell(x) - \ell_k$ "small" may not make sense

The Boston Housing Problem

- This is a first "toy" problem
- We want to estimate the median of house values over an area from some information about it which we believe relevant
- Features x: several real estate–related variables of Boston areas
  - CRIM: per capita crime rate by town
  - RM: average number of rooms per dwelling
  - NOX: nitric oxides concentration (parts per 10 million)
  - AGE: proportion of owner-occupied units built prior to 1940
  - LSTAT: % lower status of the population
  - ...
- Target y: MEDV, median value of owner-occupied homes in $1,000's

3.3 Regression Basics

Model Parameterization

- Usually individual models are selected through (ideally optimal) **parameter sets**
  - The parameters (weights) $W \in R^M$ select a concrete f in a model family $\mathcal{F}$
- **Parametric** models have a fixed functional form $f(x) = f(x; W)$
  - Simplest example: **linear regression**, where x has dimension d, $M = d + 1$ and $W = (w_0, w)$

$$f(x; w_0, w) = w_0 + \sum_{j=1}^{d} w_j x_j = w_0 + w \cdot x$$

- **Semi–parametric** models: also use weights but without a predefined functional form

  - MLPs but also RF or GBR

- **Non parametric** models do not use weights nor follow any broad functional form; Nearest Neighbor models

Issues in Model Building

- There are some initial questions when working with models from a given family $\mathcal{F}$:
  - Which assumptions do they make?
  - How do they work on the input features?
  - How we do build them?
- In turn, these questions lead to another two:
  - How do we select the best model from the given family for the problem at hand?
  - How do we control the model building procedure?
- All of them address fundamental issues that require a **moderately deep** understanding of what is going on under the model's hood
- This understanding is usually framed in **mathematical language**

How to Build Regression Models

- In general we have a sample $S = \{x^p, y^p\}$, $1 \leq p \leq N$, with x^p the **features** and y^p the **targets**
- We want to build a model $\widehat{y} = f(x)$ so that $\widehat{y}^p = f(x^p) \simeq y^p$;
  - I.e., we want to **regress** y to the $f(x)$
- The concrete f is chosen within a certain family $\mathcal{F}$
  - Examples here: linear regression, multilayer perceptrons (MLPs), SVMs
  - And also: Random Forests (RF), Gradient Boosting (GB), nearest neighbor (NN)
- Natural option to ensure $f(x^p) \simeq y^p$: choose f to minimize the sample **Mean Square Error (MSE)**

$$\widehat{e}(f) = \widehat{e}_S(f) = \frac{1}{2N}\sum_{p=1}^{N}(y^p - f(x^p))^2$$

- Thus, the model we select is $\widehat{f} = \widehat{f}_S = \arg\min_{f\in\mathcal{F}} \widehat{e}_S(f)$

Model Estimation as Error Minimization

- For a parametric or semiparametric $f(x; W)$ we can write $\widehat{e}_S(f) = \widehat{e}_S(f(\cdot; W)) = \widehat{e}_S(W)$
- The problem to solve becomes

$$\widehat{W}^* = \widehat{W}^*_S = \arg\min_W \widehat{e}_S(W), \text{ i.e., } \widehat{e}_S(\widehat{W}^*) \leq \widehat{e}_S(W)\ \forall W$$

- In linear regression the sample error is

$$\widehat{e}(w_0, w) = \frac{1}{2N} \sum_p (y^p - w_0 - w \cdot x^p)^2$$

and the population error is

$$e(w_0, w) = \frac{1}{2} E[(y - w_0 - w \cdot x)^2 p(x, y) dx dy]$$

- Linear regression reduces to **minimize the quadratic form** $\widehat{e}_S(W)$

Regression Assumptions

- **Key assumption**: x and y are related as $y = \phi(x) + n$ where
  - $\phi(x)$ is the **true** underlying function
  - n is **additive noise** with 0 mean and finite variance σ_N^2
- Our sample is just a particular instance of a deeper **sample generation process**
- Thus x, n are produced by **random variables** X, N
  - And so is y, given by $Y = \phi(X) + N$
- Moreover, X and N are **independent distributions** with densities $q(x)$, $\nu(n)$
- Thus, X and Y (or X and N) have a joint density

$$p(x, y) = p(x, \phi(x) + n) = q(x)\ \nu(n) = q(x)\ \nu(y - \phi(x))$$

MSE Decomposition

- We can decompose the MSE error of any model f as

$$\begin{aligned}
2\text{mse}(f) = E_{x,y}[(y - f(x))^2] &= \int (n + \phi(x) - f(x))^2 q(x) \nu(n) dx dn \\
&= \int \left(n^2 + 2n(\phi(x) - f(x)) + (\phi(x) - f(x))^2\right) q(x) \nu(n) dx dn \\
&= \int n^2 \nu(n) dn + \int (\phi(x) - f(x))^2 q(x) dx + \\
&\quad 2 \left(\int n \nu(n) dn\right) \left(\int (\phi(x) - f(x)) q(x) dx\right) \\
&= \sigma_N^2 + E_x[(\phi(x) - f(x))^2]
\end{aligned}$$

- Thus **for any model we have mse**$(f) \geq \sigma_N^2$ **always**
- And we should focus on achieving on $f \simeq \phi$ (which we don't know!!)

The Best Regression Model

- It is easy to see that the best f is simply $f(x) = E_y[y|x]$, for

$$E_y[y|x] = E_n[\phi(x) + n] = \int (\phi(x) + n)\, \nu(n) dn = \phi(x)$$

- Have we finished? In theory yes; in practice, not at all!!!
  - We do not know ν and, thus, cannot compute the required integral
  - If we would have several M values y^j for any x, we could try $\widehat{\phi}(x) = \frac{1}{M} \sum_1^M y^j$
  - But this doesn't happen either
- Now we have two options:
  - Try to stretch the $E[y|x]$ approach, for instance, by k–**NN Regression**
  - Forget about it and get back to get models f such that $f \simeq \phi$

k–NN Regression

- We can try to use the y^p values of several x^p close to a new x
- We fix a number k of neighbors $x^{p_1}, \ldots, x^{p_k}$ and estimate $\hat{y} = \hat{y}(x)$ as

$$\hat{y}(x) = \frac{1}{k} \sum_{j=1}^{k} y^{p_j}$$

- $\hat{y}(x) = \hat{Y}_k^{NN}(x)$ is the k–**Nearest Neighbor (NN)** regressor which can be refined to weighted versions, such as

$$\hat{y}(x) = \frac{1}{C_k(x)} \sum_{j=1}^{k} \frac{1}{\|x^{p_j} - x\|^2} y^{p_j}$$

  with $C_k(x) = \sum_{j=1}^{k} \frac{1}{\|x^{p_j} - x\|^2}$ a normalizing constant

- But we need that close predictors give predictions that are also close, **and** that there are enough of them close by
  - This is very unlikely

The Curse of Dimensionality

- Even for low dimensions and large samples, **the sample space is essentially empty**
- Assume we have 1,000 d–dimensional x patterns whose features have values between 1 and 10
  - In dimension $d = 1$ there are 100 patterns per unit value
  - But when $d = 3$ we have just 1 pattern per volume unit
  - And if $d = 6$ we have just 1 pattern per 1,000 units of volume
  - And in dimension 10 (not a big one nowadays) we have just ... !!!

- Thus, for most problems, **there never will be enough close points**
- As a consequence, to get k observations we may go too far away from x and the average will not be meaningful
- Therefore, unless we deal with violently non–linear problems, a simple linear model may be better than k–NN regression for moderate dimensions

Linear Models

- Assuming $x \in R^d$, the basic linear model is

$$f(x) = w_0 + \sum_1^d w_i x_i = w_0 + w \cdot x$$

- w_0 complicates notation; to drop it we center x and y so that $E[x_i] = E[y] = 0$; then $w_0 = 0$
- Then we are left with the simpler **homogeneous** model $f(x) = w \cdot x$
- In practice we will always **normalize** x, for instance to have 0 mean and 1 standard deviation (std) on each feature
  - But not y if we may help it (or it is easy to reverse this)
- But: how do we find w?

1–dimensional Linear Regression (LR)

- Assume that features X and target Y are **centered**, i.e., have 0 means
- For 1-dimensional patterns x the LR model then becomes $f(x) = w\ x$
- And the error is then the function

$$\begin{aligned}\widehat{e}(w) &= \frac{1}{2N}\sum_{p=1}^N (w\ x^p - y^p)^2 = \frac{1}{2N}\sum_p \left(w^2\ (x^p)^2 - 2x^p\ y^p\ w + (y^p)^2\right) \\ &= w^2 \left(\frac{1}{2N}\sum_p (x^p)^2\right) - w\left(\frac{1}{N}\sum_p x^p\ y^p\right) + \frac{1}{2N}\sum_p (y^p)^2\end{aligned}$$

- Thus, $\widehat{e}(w) = aw^2 + bw + c$ with $a > 0$ and it has obviously a minimum w^*
- To find it we just solve $\widehat{e}'(w) = 0$

Solving $\widehat{e}'(w) = 0$

- To compute $\widehat{e}'(w)$ we have

$$\widehat{e}'(w) = w\left(\frac{1}{N}\sum_p (x^p)^2\right) - \frac{1}{N}\sum_p x^p\ y^p$$

- The optimal w^* solves $\widehat{e}'(w) = 0$ and is given by

$$w^* = \frac{\frac{1}{N}\sum_p x^p y^p}{\frac{1}{N}\sum_p (x^p)^2} \quad = \quad \frac{\frac{1}{N}X^tY}{\frac{1}{N}X^tX} = \frac{\mathrm{covar}(x,y)}{\mathrm{var}(x)}$$

  where X and Y denote the N dimensional vectors $(x^1, \ldots, x^N)^t$, $(y^1, \ldots, y^N)^t$

General Linear Regression

- Assume again that X and Y are centered
- The LR model becomes now $f(x) = \sum_1^d w_i x_i = w \cdot x$
- If Y is the $N \times 1$ **target** vector and we organize the sample S in a $N \times d$ **data matrix** X, the sample mse is given by

$$\begin{aligned}\widehat{e}(w) &= \frac{1}{2N}\sum_p (w \cdot x^p - y^p)^2 = \frac{1}{2N}(Xw - Y)^t(Xw - Y) \\ &= \frac{1}{2N}(w^tX^tXw - 2w^tX^tY + Y^tY)\end{aligned}$$

- Now we have to solve $\nabla\widehat{e}(w) = 0$, i.e., $\frac{\partial \widehat{e}}{\partial w_i}(w) = 0$
- It is easy to see that

$$\nabla\widehat{e}(w) = \frac{1}{N}X^tXw - \frac{1}{N}X^tY = \widehat{R}w - \widehat{b}$$

Solving the Linear Equations

- The optimal $\widehat{w}^*$ must verify $\nabla\widehat{e}(\widehat{w}) = \widehat{R}\,\widehat{w} - \widehat{b} = 0$, where

$$\widehat{R} = \frac{1}{N}X^tX, \;\; \widehat{b} = \frac{1}{N}X^tY$$

- Over the original, non–centered data matrix we have

$$\widehat{R} = \frac{1}{N}(X - \overline{X})^t(X - \overline{X});$$

  i.e., $\widehat{R}$ is the **sample covariance matrix**
- If $\widehat{R}$ is invertible, we just solve the linear system $\widehat{R}\,\widehat{w} - \widehat{b} = 0$
- And obtain the sample–dependent optimal $\widehat{w}^*$ as

$$\widehat{w}^* = \widehat{R}^{-1}\widehat{b} = \left(X^tX\right)^{-1} X^tY = \mathrm{covar}(X)^{-1}\,\mathrm{covar}(X,Y)$$

- $\widehat{R}$ is oftent inverted through the **Singular Value Decomposition** (SVD) of the data matrix X

SVD

- The **Singular Value Decomposition** (SVD) of a $N \times d$ matrix X is $X = UDV^t$ where
  - U, V are orthonormal matrices with dimensions $N \times d$ and $d \times d$
  - D is a $d \times d$ diagonal matrix with diagonal elements $\delta_1 \geq \delta_2 \geq \dots$
  - We can use D to check the invertibility of $\widehat{R}$
- Then $XX^t = UDV^tVDU^t = UD^2U^t$
- The columns of U and V are called the left and right singular vectors; the elements δ_i are the singular values
- The columns of U and V are unique up to a sign change (so different packages may give seemingly different U and V)

Gradient Descent MSE Minimization

- Computing the covariance matrix has a $O(N \times d^2)$ cost and invert it has a $O(d^3)$ cost
  - For big data problems it may not possible to solve analytically the normal equation $\nabla\widehat{e}(w) = 0$
- The simplest numerical alternative is **gradient descent**:
  - Starting from some random w^0 we iteratively compute
  $$w^{k+1} = w^k - \rho_k \nabla\widehat{e}(w^k) = w^k - \frac{\rho}{n_B}\left(\widehat{X}_B^t\widehat{X}_B w^k - \widehat{X}_B^t Y\right)$$
  over a **mini-batch** B with n_B samples
  - Component wise: $w_i^{k+1} = w_i^k - \rho_k \frac{\partial\widehat{e}}{\partial w_i}(w^k)$
  - ρ_k is the **learning rate**
- If $w^k \to w^*$, then $\nabla\widehat{e}(w^*) = 0$
  - Since our problems have obviously minima, this should be enough

Measuring Model Fit

- First option: **Root Square Error** $RSE = \sqrt{\frac{1}{N}\sum(y^p - \widehat{y}^p)^2}$
- OK, but how good is this? We must always have a **base model** to benchmark our results
- Simplest "model": a constant w_0, which yields the mean $\overline{y} = \frac{1}{N}\sum_1^N y^p$, with square error
$$\frac{1}{N}\sum(y^p - \overline{y})^2 = \mathrm{Var}(y)$$
- We can compare our model against this base model by computing
$$\frac{\sum(y^p - \widehat{y}^p)^2}{\sum(y^p - \overline{y})^2} = \frac{RSE^2}{\mathrm{Var}(y)}$$
- The widely used R^2 coefficient is simply $R^2 = 1 - \frac{RSS}{TSS}$

Regularization

- Our regression solution $\widehat{w}^* = \left(X^t X\right)^{-1} X^t Y$ won't work if $X^t X$ is not invertible
  - For instance, when some features are correlated
- We could fix this working instead with $X^t X + \alpha I$ for some $\alpha > 0$
  - It is then always positive definite and we can thus invert it
- To make this practical, note that $\widehat{w}^* = \left(X^t X + \alpha I\right)^{-1} X^t Y$ minimizes

$$e_R(w) = \frac{1}{2N} \sum_p (y^p - w \cdot x_p^p)^2 + \frac{\alpha}{2} \|w\|^2,$$

- This is the **Ridge Regression** problem
  - Our first example of **regularization**, a key technique in Machine Learning
  - **All ML models must be regularized in some way**
- Important issue: how to find the right choice for α?

Takeaways on Linear Regression

1. We introduced **supervised** models
2. We have reviewed the essentials of the **linear regression model** (always the first thing to try)
3. We have considered model estimation as a problem on **error minimization**
4. We have seen how to build linear models **analytically and numerically**
5. We have seen how to **measure model fit**
6. We have introduced **regularization**

3.4 Bias, Variance and Cross Validation

Sample Bias and Variance

- With several **independent** samples $S_1, \ldots, S_M$, it is natural to use as our best final model the average of their associated $\widehat{f}_{S_m}(x)$ models, i.e.,

$$\frac{1}{M} \sum_1^M \widehat{f}_{S_m}(x) \simeq E_S[\widehat{f}_S(x)] = \widehat{f}_N(x)$$

  - The expectation $E_S[\widehat{f}_S(x)]$ is taken over all possible samples S of size N
- $\hat{f}_N(x) = E_S[\widehat{f}_S(x)]$ is our ideal **best model**

- The **variance** of the $\widehat{f}_S(x)$ estimates is then

$$V_N(x) = E_S\left[(\widehat{f}_S(x) - \widehat{f}_N(x))^2\right]$$

Bias Versus Variance

- Ideally we would like to have a model such that

$$\widehat{f}_N(x) - \phi(x) \simeq 0,$$

  i.e., a model with small **bias**

  - This should be achievable if we have rich, highly flexible models
  - Or, if not, with essentially no regularization

- But we would also like to have a model such that

$$V_N(x) \simeq 0,$$

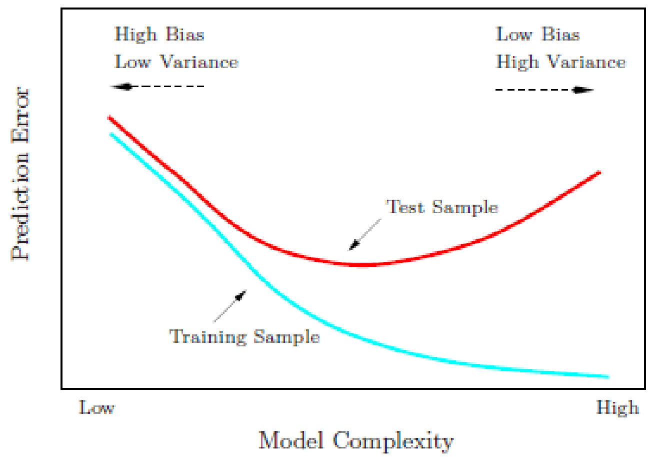
  i.e., a model with small **variance** $V_N(x)$

  - This should be achievable if models are simple with few parameters
  - Or with more severe regularization if not

- But obviously both goals are contradictory to a large extent

The Bias–Variance Tradeoff

- There is thus a **tradeoff** between bias (low for complex models) and variance (low for simple models)



Taken from *Hastie et al., p. 38*

Two Examples

- In k–NN regression the parameter that controls the tradeoff is just k
  - If $k = N$, the sample size, the N–NN estimator is just the mean: $Y_N^{NN} = \overline{y}$, with very small variance but large bias (it's an obviously bad model!!)
  - If $k = 1$, the 1–NN estimator will have smaller bias but a large variance: changing the sample is very likely to change the sample point nearest to x
- In Ridge regression the parameter that controls the tradeoff is the regularization penalty α
- If $\alpha \gg 1$, any non zero w implies a large regularization penalty
  - It is thus likely that $w \simeq 0$ and the Ridge model reduces again to the mean $\overline{y}$, with large bias and small variance
- But if $\alpha \simeq 0$, w can wander on the entire $\mathbf{R}^d$
  - The bias will be then smaller, but the weights w_S and $w_{S'}$ from different samples are likely to be very different, resulting in larger variances

Evaluating Expected Performance

- It is obvious that before we start applying a model, we should have a reasonably accurate idea of its performance in practice
- I.e., we want to estimate the model's **generalization performance**
- Estimating the generalization performance **only over the sample S used for training results in misleading error values**
- The preceding suggests to have M independent subsamples S_m and then
  - To compute $\widehat{f}_M(x) = \frac{1}{M} \sum_m \widehat{f}_{S_m}(x) \simeq \widehat{f}_N(x)$
  - To get the error estimate $\widehat{e} = \frac{1}{N} \sum_p (y^p - \widehat{f}_M(x^p))^2$ over a new, **unseen** sample $S' = \{(x^p, y^p)\}$
- But since usually we only have a single S, we "simulate" thus by **Cross Validation** (CV) to get our first realistic generalization error estimates

Cross Validation

- In Cross Validation (CV) we
  - Randomly split the sample S in M subsets $S_1, \ldots, S_M$
  - Work with M **folds**: pairs (S_m, S_m^c), with

$$S_m^c = S - S_m = \cup_{i \neq m} S_i$$

  - Build M different models **using the S_m^c as training subsets**

  - Compute their errors e_m on the folds' **validation subsets** S_m
  - Use these errors' average as a first estimate of the true model performance

- CV can and **must be used** in any model building procedure
- Most data science packages have tools to simplify this
- We will also use CV to find **optimal model hyper–parameters** such as α in Ridge Regression

Grid Hyper–parameter Selection

- Consider for Ridge regression a hyperparameter range $0 < \epsilon \leq \alpha \leq A]$
  - $\alpha = \epsilon$: essentially no penalty, small bias and high variance
  - $\alpha = \Lambda$: large penalty, small variance but high bias
- Select an $L + 1$ point **grid** $\{\alpha_0 = \epsilon, \alpha_1, \ldots, \alpha_L = A\}$
  - The α_j can be equi–spaced, log equi–spaced, random, ...
- At each α_ℓ
  - Train M models on the S_m^c using the hyperparameter α_ℓ
  - Average their test errors e_m on the S_m to get the error $e(\alpha_\ell)$ at α_ℓ
- And choose the final (hopefully) optimal hyperparameter α^* as

$$\alpha^* = \arg\min_{0 \leq \ell \leq L} e(\alpha_\ell)$$

- α^* gives the model with the **best expected generalization among all possible** α **choices**

Takeaways on Bias, Variance and CV

1. We have stressed that **any model estimation is sample–dependent** and that this has to be controlled
2. We have introduced the **bias** and **variance** as the two key components of any model error
3. We have discussed **bias–variance trade–off**
4. We have introduced **Cross Validation** here as a tool to estimate a **model's generalization performance**
5. We have also introduced **Cross Validation** as a tool to estimate a **model's hyper–parameters**

3.5 Logistic Regression

Linear Regression for Classification?

- Building a regression model with targets given by some coding of class labels usually doesn't make sense
- However, for a binary 0–1 response, it can be shown that the $w_0 + w \cdot x$ obtained using linear regression is in fact an estimate of $P(1|x)$
  - We may thus fix a threshold δ_0 and decide 0 if $w_0 + w\dot{x} < \delta_0$ and 1 otherwise
  - However, we may end up with probability estimates less than 0 o bigger than 1!!!
- Better idea: try to transform the linear output $w_0 + w \cdot x \in (-\infty, \infty)$ into a probability $P(1, x) \in (0, 1)$
- The sigmoid function $\frac{1}{1+e^{-z}}$, a smoothing of the Heaviside function, does just that

Logistic Regression (LR)

- We assume
$$P(1|x) = P(1|x; w_0, w) = \frac{1}{1 + e^{-(w_0 + w \cdot x)}}$$
  - Then $0 \leq P(1|x) \leq 1$ for any x and we have
$$P(0|x) = 1 - P(1|x) = \frac{e^{-(w_0 + w \cdot x)}}{1 + e^{-(w_0 + w \cdot x)}} = \frac{1}{1 + e^{w_0 + w \cdot x}}$$
  - Notice that if $w_0 + w \cdot x = 0$, $P(1|x) = P(0|x) = 0.5$
- The ratio $\frac{P(1|x)}{P(0|x)} = e^{w_0 + w \cdot x}$ is called the **odds** of x and its log the **log odds** or **logit**
- Thus, the basic assumption in LR is that the **logit is a linear function** $w_0 + w \cdot x$ of x
- We have the model $f(x; w)$; we need a **loss** function $L(w_0, w)$ to minimize for which we use the sample's **likelihood**

Estimating w_0^*, w^*

- Assume a single sample x, y and two possible model coefficients w_0, w and w_0', w'
- Denoting by $p = P(y|x; w_0, w)$ and $p' = P(y|x; w_0', w')$, it is clear that we should prefer w_0, w if $p > p'$ and w_0', w' if not
  - In other words, we prefer the coefficients that give a **higher posterior probability** to the sample (x, y)
- For an independent sample $S = \{(x^p, y^p)\}$, its joint probability under a posterior model $p = P(y|x; w_0, w)$ is
$$P(Y|X; w_0, w) = \prod_{p=1}^{N} P(y^p|x^p; w_0, w)$$

- And, again, given two possible model coefficients w_0, w and w'_0, w', we should prefer w_0, w iff
$$P(Y|X; w_0, w) > P(Y|X; w'_0, w')$$

Sample's Likelihood

- Therefore, we can estimate the optimal w_0^*, w^* as
$$w_0^*, w^* = \arg\max_{w_0,w} P(Y|X; w_0, w)$$
- By the independence assumption we have
$$\begin{aligned} P(Y|X; w_0, w) &= \left\{\prod_{y^p=1} P(1|x^p)\right\} \left\{\prod_{y^p=0} P(0|x^p)\right\} \\ &= \prod_{p=1}^{N} P(1|x^p)^{y^p} \;\; P(0|x^p)^{1-y^p} \end{aligned}$$
with the last equality follows from
  - If $y^p = 1$, $P(1|x^p) = P(1|x^p)^{y^p} = P(1|x^p)^{y^p} \;\; P(0|x^p)^{1-y^p}$, and
  - If $y^p = 0$, $P(0|x^p) = P(0|x^p)^{1-y^p} = P(1|x^p)^{y^p} \;\; P(0|x^p)^{1-y^p}$

Max Log–Likelihood Estimation

- The log–likelihood of w_0, w given S is then
$$\begin{aligned} \ell(w_0, w; S) &= \log P(Y|X; w_0, w) \\ &= \sum_{p=1}^{N} \{y^p \; \log P(1|x^p) + (1 - y^p) \; \log P(0|x^p)\} \\ &= \sum_p y^p \; \log \frac{P(1|x^p)}{P(0|x^p)} + \sum_p \log P(0|x^p) \\ &= \sum_p y^p \; (w_0 + w \cdot x^p) - \sum_p \log(1 + e^{w_0 + w \cdot x^p}) \end{aligned}$$
- We can thus estimate the optimal $\widehat{w}_0^*, \widehat{w}^*$ as
$$\widehat{w}_0^*, \widehat{w}^* = \arg\min_{w_0,w} \; -\ell(w_0, w; S)$$
- Extra bonus: $-\ell$ is a convex differentiable function of (w_0, w) and, thus, it is enough to solve $\nabla\ell(w_0, w) = 0$

Newton–Raphson Solution

- However, $\nabla\ell(W) = \nabla\ell(w_0, w) = 0$ doesn't admit a closed form solution but only an iterative, numerical one
- We solve it the **Newton–Raphson** iterative method (equivalent here to Newton's method for minimization)

- Starting from a random $W^0 = (w_0^0, w^0)$, Newton's iterations are

$$W^{k+1} = W^k + (\mathcal{H}_\ell(W^k))^{-1}\nabla\ell(W^k)$$

- $\mathcal{H}_\ell(W^k)$ denotes the Hessian of ℓ at W^k (which may or may not be invertible)
  - Everything is fine if the W^k are close enough to the optimum W^* but far away things may get tricky
- Just as before, we can add a regularization term $\frac{\alpha}{2}\|w\|^2$
- The iterations in Logistic Regression are again typical of many of the model building methods used in Machine Learning

Learning in ML

- The general approach to **learning** is usually the following:
  - A **model** $f(x; W)$ is chosen
  - Given a sample $S = \{(x^1, y^1), \ldots, (x^N, y^N)\}$ and a loss function $\ell(y, \widehat{y})$, we define a **sample dependent loss function**

$$L(W) = L(W|S) = \sum \ell\left(y^p, \widehat{y}^p = f(x^p; W)\right)$$

- $L(W)$ is often minimized from some W^0 by **iterations**

$$W^{k+1} = W^k - \rho_k G(W^k, S)$$

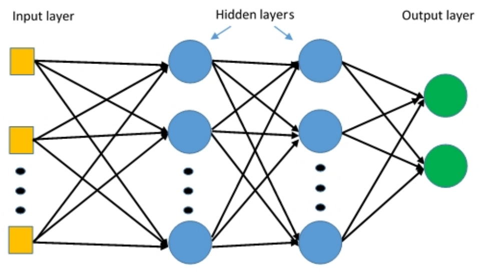
  with ρ_k a **learning rate** and G some vectorial function
  - When $G(W) = \nabla L(W)$ we have **gradient descent**
  - When $G(W) = \mathcal{H}(W)^{-1}\nabla L(W)$ we obtain **Newton's method**
- In **batch learning** the entire sample S is used at each iteration
- **On–line** or **minibatch learning**: we use either a single patterns (x^p, y^p) or small subsample

4 Multilayer Perceptrons

4.1 Classical MLPs

MLP Architecture

- General layout:
  - An input layer (input)
  - One or several hidden layers
  - One output layer
- Feedforward connections only



- Example: TensorFlow Playground

MLP Connections

- No feedback or lateral conections
- Fully connected layers
- Linear unit connections and (usually) non linear activations inside each unit
  - Combined effect of sucesive layers: potentially highly non–linear transformation
- General processing: layered and feedforward
- In practice (1990s), one hidden layer and only sometimes two
- Later (around 2010): Deep Networks with “many” (from 3 to 10) layers
- Now: almost anything!

Unit Activation and Output

- The **activations** a_i^h of a unit in layer h receives the **outputs** from processing in the previous layer

$$a_i^h = \sum_{j=1}^{n_{h-1}} w_{ij}^h o_j^{h-1} + b_i^h,$$

- In matrix/vector form:

$$a^h = W^h o^{h-1} + b^h$$

- **Output** o_i^h of a unit: non linear processing of its activation $o_i^h = \varphi(a_i^h)$
- In vector form:

$$o^h = \varphi(a^h),$$

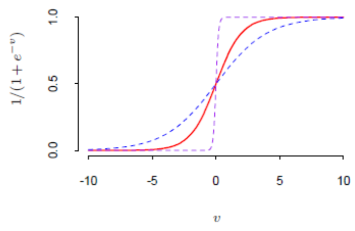
where φ is applied over each unit

Activation Functions

- Choices for f:
  - Heaviside (in the very first Rosenblatt's Perceptrons): $\varphi(a) = 0$ if $a \leq 0$, $\varphi(a) = 1$ if $a > 0$
  - Identity/linear: $\varphi(a) = a$
  - Sigmoid:
$$\varphi(a) = \sigma(a) = \frac{1}{1 + e^{-a}}$$
  - Hyperbolic tangent:
$$\varphi(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$
  - Rectified Linear Units (ReLUs): $\varphi(a) = r(a) = \max(0, a)$

Sigmoid and Hyperbolic Tangent

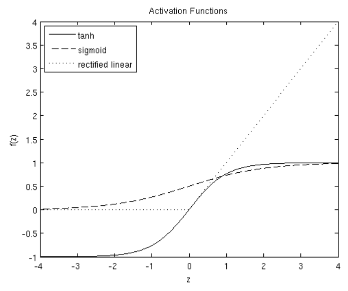
- Sigmoid and tanh: smooth version of Heaviside step function



- Classical choices:
  - Hyperbolic tangent for hidden units
  - Linear outputs for modelling (and sometimes) classification problems
  - Sigmoid outputs for classification problems (as in logistic regression)

ReLUs

- ReLU transfer function: $r(x) = \max(0, x)$



From Stanford's UFLDL Tutorial

- We have $r'(x)$ either 0 or 1 (hoping $x = 0$ never happens!!)
  - Many gradient elements will go to 0
  - Many units (neurons) will be "dead", with a 0/constant activation no matter the input

The Simplest MLP I

- The Single Hidden Layer (SHL) MLP
  - D inputs (determined by the problem at hand)
  - One hidden layer with H units (number to be chosen) and tanh activation
  - One linear or sigmoid output (according to the problem at hand)
- Input–hidden processing: denoting inputs by x and the hidden unit ouput as o,

$$o^h = \tanh\left(b_h^H + \sum_{j=1}^{D} W_{hj}^H x_j\right)$$

  - In matrix/vector form: $o = \tanh\left(W^H x + b^H\right)$

The Simplest MLP II

- Hidden–output processing: since targets are 1–dimensional, we have for the outputs $\widehat{y}$

$$\widehat{y} = \sum_{h=0}^{H} w_h^O o_h + b^O,$$

  - In vector form: $\widehat{y} = w^O \cdot o + b^O$
  - That is, a **linear model** on the last hidden layer outputs
- Global process:

$$\widehat{y} = f(x; w^O, W^H, b^O, b^H) = b^O + \sum_h w_h^O \tanh\left(b_h^H + \sum_j W_{hj}^H x_j\right)$$

  - Or in matrix/vector form

$$\widehat{y} = f(x; w^O, W^H, b^O, b^H) = b^O + w^O \cdot \tanh\left(b^H + W^H x\right)$$

MLPs and Universal Approximation

- Recall that the regression goal is to get $f \simeq \phi$
- We say that $\mathcal{F} = \{f(x; \mathcal{W})\}$ is a **Universal Approximation Family** over a domain $\mathcal{R}$ if

For any $\epsilon > 0$ and any reasonable ϕ, we can find an $f(x; \mathcal{W}_{\phi,\epsilon})$ s.t.

$$\int (\phi(x) - f(x; \mathcal{W}_{\phi,\epsilon}))^2 p(x) dx \leq \epsilon$$

- Notice that Universal Approximation is just what we need in regression
- In fact a **Single Hidden Layer MLP with enough hidden units is an effective universal approximator**
- But we have to be able to build them

MLP Error Function

- MSE is the standard error function for regression MLPs

$$\begin{aligned} e(\mathcal{W}) &= \frac{1}{2} E_{x,y}\left[(y - f(x;\mathcal{W}))^2\right] = E_{x,y}\left[e^\ell(x,y;\mathcal{W})\right] \\ &= \int e^\ell(x,y;\mathcal{W}) p(x,y) dx dy \end{aligned}$$

with $e^\ell(x,y;\mathcal{W})$ denotes the **local error**

$$e^\ell(x,y;\mathcal{W}) = \frac{1}{2}(y - \widehat{y})^2 = \frac{1}{2}(y - f(x;\mathcal{W}))^2$$

MSE Gradient

- The general idea would be to obtain $\mathcal{W}^*$ as a solution of $\nabla e(\mathcal{W}) = 0$, where we have

$$\begin{aligned} \nabla e(\mathcal{W}) &= E_{x,y}\left[\nabla_{\mathcal{W}} e^\ell(x,y;\mathcal{W})\right] \\ &= E_{x,y}\left[\nabla_{\mathcal{W}} f(x;\mathcal{W})(f(x;\mathcal{W}) - y)\right] \end{aligned}$$

for we have

$$\begin{aligned} \nabla_{\mathcal{W}} e^\ell(x,y;\mathcal{W}) &= -(y - f(x;\mathcal{W})\ \nabla_{\mathcal{W}} f(x;\mathcal{W}) \\ &= \nabla_{\mathcal{W}} f(x;\mathcal{W})(f(x;\mathcal{W}) - y) \end{aligned}$$

- We have therefore two tasks:
  - Compute ∇e
  - Exploit it to build MLPs
- We will exploit $\nabla e(\mathcal{W})$ through **optimization methods** after we compute it

SHL Forward Pass I

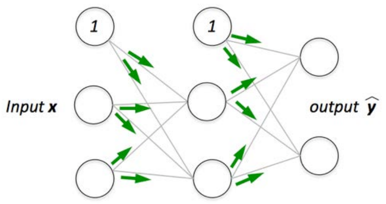
- We apply the preceding to a single hidden layer (SHL) MLP with
  - A single output unit and input–to–hidden weight matrix $W^H = (W^H_{hj})$ and bias b^H vector and

  - A hidden–to–output weight vector $w^O = (w_1^O, \dots, w_H^O)$ and scalar bias b^O

- Recall that the forward pass can be computed as follows
  - $a = W^H x + b^H$, $o = \varphi(a)$,
  - Or unit–wise: $a_h = \sum W^H_{hi} x_i + b^H_i$, $o_h = \varphi(a_h)$
  - $y = w^O \cdot o + b^O = \sum_h w^O_h o_h + b^O$
- Straightforward to program

SHL Forward Pass II

- Graphically we have the following scheme:



From Sebastian Raschka's A Visual Explanation of the Back Propagation Algorithm for Neural Networks, KDnuggets

- We turn next to the gradient computation
  - Do we need to work on that? Current Deep NN backends compute the error gradient "automatically"
  - But, as Andrej Karpathy says, Yes, you should understand backprop

Computing the Gradient

- If w_{ij} is the weight connecting unit j to unit i, we have

$$\frac{\partial e^\ell}{\partial w_{ij}} = \frac{\partial e^\ell}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial e^\ell}{\partial a_i} o_j = \delta_i o_h$$

- In the output layer $e^\ell = \frac{1}{2}(y - \widehat{y})^2$ and $a^O = \widehat{y}$, and thus,

$$\delta^O = \frac{\partial e^\ell}{\partial a^O} = \frac{\partial e^\ell}{\partial \widehat{y}} = \widehat{y} - y$$

- Therefore, in the output layer we have

$$\frac{\partial e^\ell}{\partial w^O_h} = (\widehat{y} - y)\frac{\partial a^O}{\partial w^O_h} = (\widehat{y} - y) o^H_h$$

SHL Gradient Backprop I

- In the hidden layer we **backpropagate** the error $\delta^O = \widehat{y} - y$:

$$\delta_h^H = \frac{\partial e^\ell}{\partial a_h^H} = \frac{\partial e^\ell}{\partial a^O} \frac{\partial a^O}{\partial a_h^H} = \delta^O \frac{\partial a^O}{\partial a_h^H} = (\widehat{y} - y) \frac{\partial a^O}{\partial a_h^H}$$

- And, therefore, we have

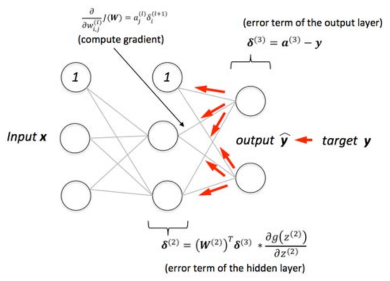
$$\frac{\partial a^O}{\partial a_h^H} = \frac{\partial a^O}{\partial o_h^H} \frac{\partial o_h^H}{\partial a_h^H} = \frac{\partial a^O}{\partial o_h^H} \varphi'(a_h^H) = w_h^O \varphi'(a_h^H)$$

- Moreover, $o_j^D = x_j$ and, therefore,

$$\frac{\partial e^\ell}{\partial w_{hj}^H} = \frac{\partial e^\ell}{\partial a_h^H} \frac{\partial a_h^H}{\partial w_{hj}^H} = \delta_h^H x_j = (\widehat{y} - y) w_h^O \varphi'(a_h^H) x_j$$

SHL Gradient Backprop II

- Graphically we have the following scheme:


$\frac{\partial}{\partial w_{i,j}^{(l)}} J(W) = a_j^{(l)} \delta_i^{(l+1)}$
(compute gradient)
(error term of the output layer)
$\delta^{(3)} = a^{(3)} - y$
1
1
Input x
output $\widehat{y}$
target y
$\delta^{(2)} = (W^{(2)})^T \delta^{(3)} * \frac{\partial g(z^{(2)})}{\partial z^{(2)}}$
(error term of the hidden layer)

From Sebastian Raschka's A Visual Explanation of the Back Propagation Algorithm for Neural Networks, KDnuggets

Takeaways on Clasical MLPs

1. They have a layered structure with outputs computed in a **forward pass** using differentiable activations
2. Usual activations: ReLUs, sigmoid, tanh, linear
3. MLPs are **universal approximators**: this is indispensable for regression but has to be handled with care
4. MSE is the usual regression cost; we will use cross entropy in classification
5. The error function gradients are computed by **backpropagation** of generalized errors
6. Backprop is basically a very simple procedure than can be **largely automated**
7. Once an MLP is defined (feedforward and backward passes), MLP training reduces to a (usually difficult and costly) **optimization problem**

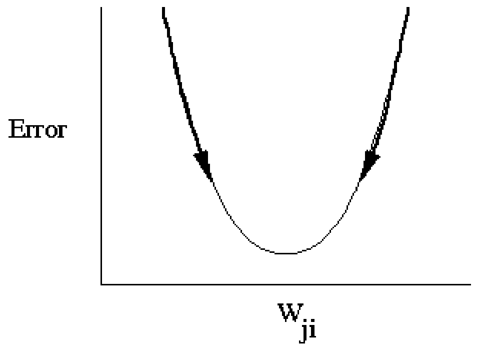
4.2 Unconstrained Smooth Optimization

Back to Optimization

- To build a ML model $\equiv$ to **minimize a loss function**
- General optimization theory is a key tool in Machine Learning (ML)
- There are two optimization set ups in ML
  - **Unconstrained** optimization, slightly simpler and the one used for MLPs
  - **Constrained** optimization, wider and more complex
- In ML we have also to consider the optimization of differentiable and also non differentiable error functions
- MLP optimization: **unconstrained and differentiable**
- And also **batch**, i.e., over the entire sample, **mini–batch** over subsamples or **on line**, pattern by pattern

Gradient Descent

- We "stretch" the weight set $\mathcal{W}$ into a vector w
- We recall that $-\nabla e(w)$ is the maximum descent direction
- First idea: to build a (hopefully convergent) sequence w^k iterating (small) steps along $-\nabla e(w^k)$



Gradient Descent II

- In more detail, we start from a random w^0 and compute

$$w^{k+1} = w^k - \rho_k \nabla_w e(w^k)$$

- ρ_k is the **learning rate** (LR)

- With a small ρ_k we ensure $e(w^{k+1}) < e(w^k)$ (although with possibly a very small descent)
- We can get a better iteration $w^{k+1} = w^k - \rho_k^* \nabla_w e(w^k)$ using a ρ_k^* given by

$$\rho_k^* = \arg\min_{\rho} e(w^k - \rho \nabla_w e(w^k));$$

  this is known as **line minimization**
- These GD methods are called **first order methods** in part because they only use ∇e

One Dimensional Newton's Method

- Assume a quadratic function $q(w) = aw^2 + bw + c$, with $a > 0$, and a minimum at w^*
- We can reach w^* from any w with a step Δw such that

$$0 = q'(w + \Delta w) = 2a(w + \Delta w) + b$$

- We have thus $\Delta w = \frac{-b-2aw}{2a}$, that is,

$$w^* = w - \frac{2aw + b}{2a} = w - \frac{1}{q''(w)} q'(w)$$

- This leads to **Newton's method**: minimize a general f iteratively using steps

$$w^{k+1} = w^k - \rho_k \frac{1}{f''(w^k)} f'(w^k)$$

  with ρ_k a suitable learning rate

One Dimensional Newton's Method II

- Notice how the $\frac{1}{f''(w_k)}$ acts as a **self adjusting learning rate**
- When the parabola is very sharp (i.e., $a \gg 1$), we will be close to the minimum and the descent can overstep it
- But then $f''(w_k)$ will be big and, hence, $\frac{1}{f''(w_k)}$ **moderates** the gradient step
- Conversely, when the parabola is very wide (i.e., $a \simeq f''(w_k) \ll 1$), the minimum will be far away and the gradient step small descent will
- But then $f''(w_k)$ will be rather small and, hence, $\frac{1}{f''(w_k)}$ **accelerates** the gradient step
- We can extend this to several dimensions

Multidimensional Newton's Method

- For a d dimensional w, the Taylor expansion of e at an optimum w^* is

$$e(w) \approx e(w^*) + \frac{1}{2}(w - w^*)^t \cdot \mathcal{H}(w^*) \cdot (w - w^*)$$

  - $\mathcal{H}(w^*)$ is the **Hessian** of e at w^* and $\nabla e(w^*) = 0$

- It follows that $\nabla e(w) \approx \mathcal{H}(w^*) \cdot (w - w^*)$ and, therefore,

$$w^* \approx w - \mathcal{H}(w^*)^{-1} \nabla_w e(w)$$

- This suggest to derive the w^k by

$$w^{k+1} = w^k - \rho_k \mathcal{H}(w^k)^{-1} \nabla_w e(w^k)$$

  which is known as **Newton's Method** (NM)

Variants of Newton's Method

- Theoretically NM converges very fast near w^*, but
  - Far from w^* convergence is not guaranteed
  - Moreover $\mathcal{H}(w^k)$ may not be invertible
  - Besides, computing $\mathcal{H}(w^k)$ is cumbersome and costly
- The **Gauss–Newton (GN)** approximation

$$\mathcal{H}(w) \simeq E[\nabla e(w) \nabla e(w)^\tau],$$

  holds for any quadratic cost and simplifies the third problem
- The **Levenberg–Marquardt (LM)** method deals with the other two problems combining
  - Gradient descent "away" from w^*
  - Gauss–Newton "near" w^*

  although "away" and "near" have to be properly addressed

One–dimensional GN Approximation

- Assume $e(w) = \frac{1}{2} \int (f(x; w) - y)^2 p(x, y) dx dy$; then

$$\begin{aligned} e'(w) &= \int (f(x;w) - y) \frac{\partial f}{\partial w}(x, y) p(x, y) dx dy; \\ e''(w) &= \int \left(\frac{\partial f}{\partial w} \right)^2 p(x, y) dx dy \\ &\quad + \int (f(x;w) - y) \frac{\partial^2 f}{\partial w^2}(x, y) p(x, y) dx dy \end{aligned}$$

- Near a minimum w^* we may expect $f(x; w) \simeq y$ and, therefore,

$$e''(w) \simeq \int \left(\frac{\partial f}{\partial w} \right)^2 p(x, y) dx dy > 0$$

- Thus, for square errors, we can use first derivates to approximate $e''(w)$

General GN Approximation I

- In the general case we have

$$\nabla e(w) = E\left[\nabla f(x;w)\left(f(x;w) - y\right)\right]$$

- And, therefore,

$$\begin{aligned}\nabla^2 e(w) \quad = \quad & E\left[\nabla^2 f(x;w)\left(f(x;w) - y\right)\right] + \\ & E\left[\nabla f(x;w) \nabla f(x;w)^\tau\right]\end{aligned}$$

- The second term is easy to compute once we have ∇f
- If $w \approx w^*$, $f(x;w) \approx y$; therefore $f(x;w) - y \approx 0$,
  - We can ignore the first, more complex, term

General GN Approximation II

- We arrive at $\nabla^2 e(w) \simeq E\left[\nabla f(x;w) \nabla f(x;w)^\tau\right]$ or, equivalently,

$$\mathcal{H}_{(i,j)(p,q)}(w) = \left(\frac{\partial^2 e}{\partial w_{ij} \partial w_{pq}}(w)\right) \simeq \left(E\left[\frac{\partial f}{\partial w_{pq}} \frac{\partial f}{\partial w_{ij}}\right]\right)_{(i,j)(p,q)}$$

- $\mathcal{J} = E\left[\nabla f(x;w) \nabla f(x;w)^\tau\right]$ is **Fisher's information matrix**
  - $\mathcal{J}$ is **semidefinite positive**
  - And thus not necessarily invertible
- Often only its diagonal is considered and we have

$$\mathcal{H}_{(i,j)(i,j)}(w) = \left(\frac{\partial f}{\partial w_{ij}}\right)^2$$

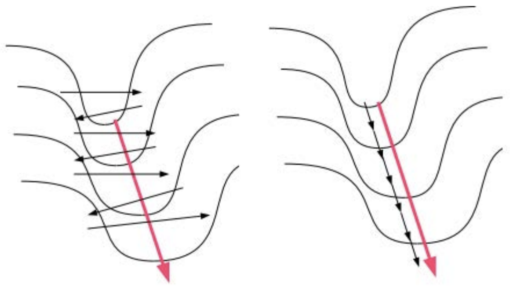
Advanced Optimization

- There are many more proposals in unconstrained optimization
- The **Conjugate Gradient** (CG) and **Quasi–Newton** (QN) methods are important in classical MLP training
- The basic idea in CG is to replace gradient descent directions $g_k = -\nabla e(w_k)$ with new conjugate directions that try to keep somehow the previous "good directions"
- The basic idea in QN is to iterate as in NM but with simple approximations $\mathcal{A}_k$ to $\mathcal{H}^{-1}(w^k)$ that converge to $\mathcal{H}^{-1}(w^*)$

- When training "small" NNs the **Limited-memory Broyden–Fletcher–Goldfarb–Shanno** (L-BFGS), a QN variant, is often used

Accelerating Gradient Descent

- A possibility on the error surface of a NN is to have many long, narrow ravines
  - Gradients bounce in the narrow section, but may be very small along the wider one



Momentum

- **Momentum** pushes them forward in the wider section (but we have to cope with a new parameter)
- Momentum tries to maintain descent's inertia with a term $\Delta^k = w^k - w^{k-1}$, i.e.,

$$w^{k+1} = w^k - \rho_k \nabla_w e(w^k) + \mu_k \Delta^k$$

  - More or less, what a **heavy ball** would do
- The goal is to keep w^k advancing in "plateaux", i.e., small gradient zones
- Momentum can be seen as a crude approximation of a CG step
- Nice explanation at Why Momentum Really Works

Nesterov's Accelerated Gradient

- Let's rewrite momentum in two steps
  1. Define $\Delta^{k+1} = -\rho_k \nabla_w e(w^k) + \mu_k \Delta^k$ and
  2. Apply
$$w^{k+1} = w^k + \Delta^{k+1} = w^k - \rho_k \nabla_w e(w^k) + \mu_k \Delta^k$$
- **Nesterov's Accelerated Gradient** is a variant of this

$$\begin{aligned} \widetilde{\Delta}^{k+1} &= -\rho_k \nabla_w e(w^k + \mu_k \widetilde{\Delta}^k) + \mu_k \widetilde{\Delta}^k; \\ w^{k+1} &= w^k + \widetilde{\Delta}^{k+1} \end{aligned}$$

- In convex optimization it improves GD and is often highly effective in Deep Network training

Adam

- Adam is currently the most widely used gradient-descent method for deep NN training
- At each step t Adam uses a new random mini–batch to
  - Update exponentially smoothed averages m_t of the gradient g_t and v_t of the the squared gradient $g_t^2 = g_t \odot g_t$ as
  $$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \ v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2;$$
  - Compute bias corrections $\widehat{m}_t$, $\widehat{v}_t$ as
  $$\widehat{m}_t = \frac{1}{1-\beta_1^t} m_t, \ \widehat{v}_t = \frac{1}{1-\beta_2^t} v_t;$$
  - Update weights as $W_t = W_{t-1} - \alpha \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon}$
- One can show $E[m_t] \simeq (1-\beta_1^t) E[g_t]$ and $E[v_t] \simeq (1-\beta_2^t) E[g_t^2]$
- Default values $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$, and $\epsilon = 10^{-8}$ usually work fine

Understanding Adam

- What is going on in Adam?
- At each step we work with estimates of the average local gradient:
$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \simeq E[\nabla_W f];$$
- Then $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \simeq E[(\nabla_W f)^2] \simeq \sigma(g_t)$
- Thus, we can see Adam as a kind of "normalized" gradient descent
- Or since we can expect $E[(\nabla_W f)^2] \simeq E[\nabla^2_{W^2} f]$, we can see Adam as a "cousin" of diagonal Gauss–Newton's steps

When to Stop Training

- Typically the $e(w_k)$ error diminishes towards an asymptotic minimum
  - If many units are used, we arrive to 0, which usually implies overfitting
- First solution: to use a separate **validation subset** V and stop training when the error in V, i.e., the **validation error** starts growing
  - But: How to choose V? What do we do for small samples?
- Second solution (better): **get a good regularization** (and forget about overfitting)

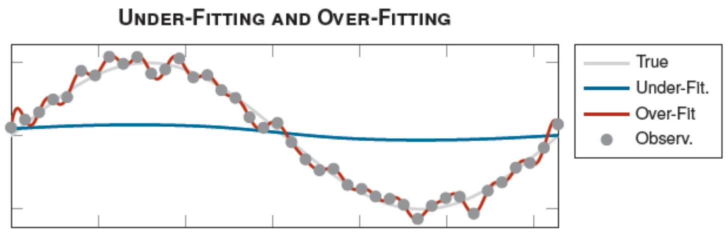
  - A low CV error is also a low validation error
  - Now training stops because of reasons such as computational cost, but not because of overfitting risk

Takeaways on MLP Optimization

- **Gradient descent** is the simplest method but possibly also the slowest
- **Momentum** can be used to speed it up
- **Newton's method** is much faster but may be very costly and difficult to apply in full form
- All the previous methods require the (usually tricky) selection of a **learning rate**
- Second order methods such as Conjugate Gradient and **Quasi–Newton** avoid learning rates and are more efficient but costlier
- **Limited Broyden–Fletcher–Goldfarb–Shanno** (L-BFGS) is currently the option of choice for "small" MLPs

Overfitting in MLPs

- Since MLPs are a UAF, they can also approximate the noise in the sample
  - Given $S = \{(x^p, y^p)\}$ if we allow enough hidden units in a SHL MLP we can arrive to a w^* s.t. $y^p = f(x^p; w^*)$
  - We get thus a sample error $\hat{e}(w^*) = 0$ but possibly with a very high generalization error
- I.e., MLPs may have very small bias but possibly large variance



(*Ph.D. Thesis of Carlos Alaíz*)

Regularization vs Overfitting

- Why is there overfitting?
  - Because we may end up having too many weights with respect to sample size
  - Because we allow these weights to explore the entire weight space

- We can avoid this wandering if we limit w's growth, for which we add a **regularization** term $g(\|w\|)$ to $e(w)$ with $g(r)$ increasing
- Working with $e_R(w) = e(w) + g(\|w\|)$ we have to **balance** the minimization of $e(w)$ and that of $g(\|w\|)$
- This balanced learning results in better generalization

L_2 Regularization

- The simplest regularization procedure adds a quadratic penalty to the square error e

$$e_R(w) = e(w) + \frac{\lambda}{2}\|w\|^2,$$

  with λ the **weight decay** factor

  - Also known as Tikhonov's regularization (or Ridge Regression for linear models)
- The desired effect is to constrain the evolution of w:
  - In fact, the regularized loss is essentially the **Lagrangian of the constrained problem**

$$\min_w e(w) \text{ subject to } \|w\|^2 \leq \rho, \quad \rho > 0$$

- The gradient becomes $\nabla e_R(w) = \nabla e(w) + \lambda w$
- And the Hessian is $\mathcal{H}_R(w) = \mathcal{H}(w) + \lambda I$

Regularized Algorithms

- The preceding methods apply straightforwardly to e_R
- Gradient descent becomes

$$w^{k+1} = w^k - \rho_k(\nabla_w e(w^k) + \lambda w^k)$$

- Newton steps are now

$$w^{k+1} = w^k - \rho_k \left(\mathcal{H}(w^k) + \lambda I\right)^{-1} \left(\nabla_w e(w^k) + \lambda w^k\right)$$

- And the Gauss–Newton approximation to $\mathcal{H}_R(w)$ is

$$\mathcal{H}_R(w) = \mathcal{H}(w) + \lambda I \sim E[\nabla f(w) \nabla f(w)^\tau] + \lambda I$$

  that is **definite positive** and, hence, **invertible**

How to choose λ

- Again, the correct choice of λ is crucial
- A small $\lambda \ll 1$ results in a small regularization effect and overfitting risk appears

- A large $\lambda \gg 1$ causes learning to forget about $e(w)$ and the model will be essentially constant and will underfit
- Usually λ is chosen by using CV by exploring a discrete set of values λ_j,
- The same can essentially done for any other hyper–parameter: number of hidden layers? number of hidden units? learning rate?? minibatch size???

MLP Ensembles

- Recall that $e(w)$ does not have a single minimum
- Moreover, the final MLP depends on the random initial w^0
- And mini–batch training adds extra randomness to the final model
- This suggests
  - To start from K independent initial weights and get K optimal weight sets w_k^*
  - To output the average $f_e(x) = \frac{1}{K}\sum_1^K f(x; w_k^*)$
- We expect outputs of the form $\widehat{y}_k^p = y^p + \epsilon_k^p$ with the ϵ_k^p independent
- Hence $\frac{1}{K}\sum_k \epsilon_k^p \simeq 0$ and $\frac{1}{K}\sum_k \widehat{y}_k^p \simeq y^p$

Takeaways on MLP Regularization

- MLPs have a **high risk of overfitting**
- Thus, they must be **regularized** to avoid overfitting
  - The regularization hyperparameter is chosen through **cross validation**
- MLP training has two **random components**: the initial point and minibatch selection
  - Thus, each training will converge to a **different optimum**
- MLP **ensembles** can take advantage of this
  - They imply extra costs but ensembles are **embarrassingly parallelizable**

4.3 Computational Costs of MLPs

MLPs and Big Data?

- Many Vs in Big Data: **Volume, Velocity**, Variety, Veracity, Value, ...
- Velocity: information flows in data streams that require fast processing and feed back
  - MLPs are rather fast
  - Less than linear models but much more so than SVMs
- Volume is probably the greatest attractive of Big Data

  - Huge samples and/or very large pattern dimension
  - Large impact in model training

Volume in MLPs

- Parallelism is the first answer to Big Volume
- On a standalone machine it may be
  - Passive: let the SO distribute work among several CPU cores, or use GPUs with low level parallelized libraries such as Linpack or BLAS for linear algebra
  - Active: explicitly exploit a problem's parallelism programming an algorithm in, say, OpenMP
- It is easy to passively parallelize the training of several MLPs
  - But memory costs multiply
- Training an isolated MLP is not CPU parallelizable:
  - There is a sequential layer dependence in Backprop
  - Each training iteration depends on the previous one
  - But matrix-vector multiplications can exploit GPU parallelism

Handling Huge Sample Training

- Two consequences of a large N are
  - Sample doesn't fit in memory and we have to split it somehow
  - Training gets "lost", for in the global gradient $\nabla e = E[\nabla e^\ell]$ we average many local gradients that may cancel each other out
- The fist problem has been always present in fields such as analog signal filtering
- Solution for both: adaptive or **on line**, i.e., pattern by pattern, weight updates
  - It also allows a relatively simple setting for a theoretical analysis
  - But not used today; instead, medium–to–large NNs are trained using **mini–batches**

MLP Complexity

- MLPs are fast to apply but costly to train
- How can we estimate this cost?
- MLP traning cost is determined by
  - Its **architecture**, that determines the number of weights to fit and that is also dependent on the input dimension D
  - The full or mini–batch **sample size** N, that determines the cost of the averages to be computed

  - The **training method**, with more or less iterations that, in turn, are more or less costly

Forward Pass Complexity

- The number of weights in a single hidden layer (SHL) MLP with D inputs, one output and one hidden layer with H units is

$$(D+1) \times H + (H+1) \simeq HD$$

- Each extra hidden layer pair with (H_1, H_2) units adds $(H_1+1) \times H_2 \simeq H_1 H_2$ weights
- For a general MLP the cost in floating point operations of a forward pass is $\simeq N \times \left(\sum_h H_h \times H_{h-1}\right)$
  - Very fast on GPUs as it theoretically becomes $O(N \times \sum_h H_h)$

The Cost of Computing ∇e^ℓ

- Computing a local gradient ∇e^ℓ in a SHL MLP with square error and one output essentially requires to compute
  - H components for the hidden to output connections, with a $O(1)$ cost each, for $\frac{\partial e^\ell}{\partial w_h^O} = (\widehat{y} - y)o_h$
  - DH components for the input to hidden connections with essentially an $O(1)$ cost each, for $\frac{\partial e^\ell}{\partial w_{h,j}^H} = \left(\sum_{i=1}^L \delta_i w_{ih}^O\right) \sigma'(a_h) x_j$
- In these we are omitting the partials with respect the bias, but they are much less
- Thus, the overall cost of computing ∇e^ℓ is essentially $O(DH)$
- More layers with H_h units add a cost $O(H_{h-1} \times H_h)$
  - $O(\sum_h H_h)$ on GPUs

The Cost of Computing ∇e

- For a mini–batch of size N_b, the cost of the mini–batch gradient ∇e of a SHL MLP is $O(N_b \times D \times H)$
- And extra layers add a cost $O(N_b \times H_{h-1} \times H_h)$
- This is of the same order of magnitude than the cost of the forward pass
- This also dominates the $O(H_{h-1} \times H_h)$ cost of updating the (H_{h-1}, H_h) weights in gradient descent
- And all these costs have to be multiplied by the number of training **epochs**

Training Complexity

- The important term in the overall training cost is nEps $\times$ cost of ∇e, with nEps the number of epochs
  - One epoch = one pass on the entire sample
- Thus, the **globally dominant term** in a SHL MLP is

$$\text{nEps} \times \text{cost of } \nabla e = O(\text{nEps} \times N \times D \times H)$$

  with N here the entire sample size
- And an extra cost

$$O(\text{nEps} \times N \times H_{h-1} \times H_h)$$

  for extra hidden layers
- Thus, training many layered, large MLPs can be **very costly**
  - But **linear in sample size** N

Takeaways on MLP's Cost

- MLP complexity is determined by its **architecture** $\{H_h\}$**, training procedure and sample size** N
- The forward and backward MLP passes have basically the **same complexity**
- Their cost per **pattern and layer** is $H_{h-1} \times H_h$
- For gradient descent these costs are **multiplied by the number nEps of epochs**
- First order methods essentially **do not add extra complexity**
- Second order methods add extra per iteration costs but should **require less iterations**
- Single MLP training is **not CPU parallelizable**
- But **GPUs** can greatly improve MLP processing costs

4.4 Convergence of MLP Training

Handling Huge Sample Training

- Two consequences of a large N are
  - Sample doesn't fit in memory and we have to split it somehow
  - Global gradient is no longer feasible and we have to exploit its average $\nabla e = E[\nabla e^\ell]$ over local gradients
- The fist problem has been always present in fields such as analog signal filtering
- Solution: adaptive or **on line**, i.e., pattern by pattern, weight updates
  - Not used today: Currently medium–to–large NNs are trained using **mini–batches**
  - But allows a relatively simple setting for a theoretical analysis

Adaptive Filtering

- The signal processing solution to not having storage: **adaptive lineal filters** (ADALINE)

$$\begin{aligned} w^{k+1} &= w^k - \rho_k(\hat{y}_k - y_k)X^k \\ &= w^k - \rho_k \nabla e^\ell(X^k, y_k; w^k) \end{aligned}$$

- This is just local gradient descent in linear regression
- This makes sense also for MLPs
- On Line Learning $\equiv$ local gradient descent

On Line Learning of MLPs

- Starting from w^0, at time $\tau = 1, 2, \ldots$
  - We get a random pattern $x^p = x^{p(\tau)}$ and its target target y^p
  - we compute $\hat{y}^\tau = F(x^p, w^\tau)$ and $\nabla e^\ell(x^p, y^p; w^\tau)$
  - We update $w^{\tau+1} = w^\tau - \rho_\tau \nabla e^\ell(x^p, y^p; w^\tau)$
- The Single Hidden Layer MLP complexity is now $O(\text{nIters} \times DH)$ and nIters $> N$ is to be expected:
  - A batch epoch has the same cost of N on line iterations
- But now we do not have to store or even to know the entire sample
- And N on line iterations usually are more effective than a batch epoch

Variants in On Line Learning

- It is straightforward to include quadratic regularization

$$w^{\tau+1} = w^\tau - \rho_\tau \left(\nabla e^\ell(x^p, y^p; w^\tau) + \lambda w^\tau \right)$$

- As well as momentum

$$w^{\tau+1} = w^\tau - \rho_\tau \nabla e^\ell(x^p, y^p; w^\tau) + \mu_\tau \Delta^\tau$$

- Moreover the MLP is constantly updated
- We can work with **minibatches** with a fixed number of patterns instead of isolated ones
  - The same tricks and stopping criteria of batch learning apply and also CG is often possible over minibatches
- But: is there convergence?

Learning as a Random Variable Evolution

- In on line learning
  - The initial weight w^0 is random
  - The pattern $x^{p(t)}$ received at time t is also random
- Thus, the sequence w^t is a **realization of a random variable** W^t
- We have deal with random variable convergence theory ...
- Rather heavy going, with many, more or less related, definitions

Convergence of Random Variables

- The basic limit concepts have to be adapted when dealing with random variables
- Set up: probability space $(\Omega, \mathcal{A}, P)$
- Pointwise limits are trickier so one starts with mean convergence
- $X_n \to X$ **in the** r**–th mean**, $r \geq 1$, (usually $r = 2$ or $r = 1$) if

$$E_P[|X_n - X|^r] = \int |X_n(\omega) - X(\omega)|^r dP(\omega) \to 0$$

- For a general probability space $(\Omega, \mathcal{A}, P)$ and suitable f, $E_P[f] = \int f(\omega)dP(\omega)$ can be properly defined
- Situation here: ω is given as a feature vector x and dP has a density $p(x)$; the previous definition becomes

$$E_P[|X_n - X|^r] = \int |X_n(x) - X(x)|^r p(x)dx \to 0$$

Pointwise Convergence of Random Variables

- We say $X_n \to X$ **almost surely** (AS) if

$$P(\{\omega : \lim X_n(\omega) = X(\omega)\}) = 1$$

  - It is the closest to the standard pointwise convergence
  - Has many synonimes: convergence **almost everywhere, strongly** or (slightly confusing) **with probability 1**
- An equivalent definition is that for all $\epsilon > 0$

$$\lim_n P(\{\omega : \sup_{m \geq n} |X_m(\omega) - X(\omega)| \geq \epsilon\}) = 0$$

- We say $X_n \to X$ **in probability** if for all $\epsilon > 0$

$$\lim P(\{\omega : |X_n(\omega) - X(\omega)| \geq \epsilon\}) = 0$$

- Convergence AS is thus stronger than convergence in probability
- In fact, convergence AS surely implies convergence in probability

Convergence in Distribution

- We say $X_n \to X$ **in distribution** if $\lim F_{X_n}(x) = F_X(x)$ at every x where F is continuous
- Synonimes: convergence **in law** or **weakly**
- The continuity requirement aims to avoid degenerate cases
  - If $X_n = \chi_{[0,1/n]}$, they converge in distribution (and pointwise) to $X \equiv 0$, for which $F_X(0) = 1$ while $F_{X_n}(0) = 0$
  - But 0 is not a point of continuity of F_X
- Convergence in distribution does not imply convergence of the density functions
- But convergence of the density functions implies convergence in distribution
- **Lévy's Theorem**: the sequence X_n converges in distribution to X iff the sequence of corresponding characteristic functions ϕ_n converges pointwise to the characteristic function ϕ of X

Relationships

- Recall that convergence almost surely implies convergence in probability
- Convergence in probability implies convergence in distribution
- Convergence in the mean for $r \geq 1$ implies convergence in probability (by Markov's inequality)
- If $r > s \geq 1$, convergence in the r–th mean implies convergence in the s–th mean
- Convergence in probability (and, hence, in the mean) implies convergence a.s. for some subsequence X_{n_k}
- If the X_n are bounded by an integrable Z and converge a.s. to X, they also converge in the r–th mean
- But the other implications are false

Limit Theorems

- **Weak Law of Large Numbers:** if the X_n are i.i.d., then the sample mean $\overline{X}_N = \frac{1}{N}\sum_1^N X_n$ converges in probability (weakly) to $\mu = E[X_n]$
- **Strong Law of Large Numbers:** if the X_n are i.i.d., then the sample mean $\overline{X}_N = \frac{1}{N}\sum_1^N X_n$ converges a.s. (strongly) to $\mu = E[X_n]$

- Besides weaker, the weak LLN is easier to prove
- **Central Limit Theorem:** if the X_n are i.i.d. with mean μ and variance σ^2, then the random variable $\sqrt{N}(\overline{X}_N - \mu)$ converges in distribution to a Normal $N(0, \sigma^2)$
- The LLN tells us that the sample mean converges; the CLT tells us how the sample mean error (approximately) behaves

Convergence of On Line Learning

- The end result for MLP on line learning is that the random variable sequence W^t converges in mean square to an w^* s.t. $0 = \nabla e(w^*)$
- That is, w^* is (very likely) a local minimum of e
- Theoretical foundation in the **Strong Law of Large Numbers:**

  If the X^n are i.i.d., then the sample mean $\overline{X}_N = \frac{1}{N}\sum_1^N X^n$ converges a.s. (strongly) to $\mu = E[X^n]$

Stochastic Optimization (SO)

- In a deeper look, assume the following:
  - We want to solve the equation $g(w) = 0$ that has a single root w^*
  - Where we don't know $g(w)$ ...
  - But can get an **oracle**, i.e., a random estimate $\hat{g}(X^n, w)$ with X^n i.i.d. such that $E[\hat{g}(X, w)] = g(w)$.
- Then, starting at a random variable W^0 and for some sequence ρ_k, we define random variables

$$W^{k+1} = W^k - \rho_k \hat{g}(X^k, W^k)$$

  and hope for the best

Robbins–Monro SO

- Robbins and Monro showed that $W^k \to w^*$ in square mean (and, hence, in probability) if (among others)
  1. g is non decreasing and $g'(w^*) > 0$,
  2. $\sum \rho_k = \infty, \sum \rho_k^2 < \infty$
- For a D dimensional extension we assume the Jacobian $J_g(w^*)$ to be positive definite
- We need a dictionary to translate this to on line MLP training
- But first let's try to understand it on a simple situation

Why it Works?

- Assume we want to solve the equation $g(w) = w - \mu$ where we do not know μ, but can get random estimates $\hat{g}(X_n, w) = w - X_n$ with X_n i.i.d. s.t. $E[X_n] = \mu$
- It is obvious that $E[\hat{g}(X, w)] = g(w)$
- Then, starting at $W_0 = 0$ and for $\rho_k = \frac{1}{k+1}$, let's define

$$W_{k+1} = W_k - \frac{1}{k+1}(W_k - X_{k+1});$$

- Then, unwinding the recurrence we arrive at

$$W_{k+1} = \frac{1}{k+1} \sum_{1}^{k+1} X_j,$$

  which, by the LLN converges to the root $\mu = E[X]$ of $w - \mu = 0$
- Thus Robbins–Monro essentially reduces here to the LLN

Stochastic Optimization and MLPs

- We begin with $e(w) = E[e^{\ell}(\cdot, w)]$, which is smooth with (we hope) a minimum w^*
- As g we take $g(w) = \nabla e(w) = E_X[\nabla e^{\ell}(x, w)]$ (unknown) and $\hat{g}(x, w) = \nabla e^{\ell}(x, w)$ (known)
- Then we have $E_X[\hat{g}(x, w)] = g(w)$
- Its Jacobian $J_g(w) = \mathcal{H}_e(w)$ is definite positive at and near w^*
- Starting at a random W^0 we take the X^k as i.i.d. and build the r.v. sequence for appropriate ρ_k

$$W^{k+1} = W^k - \rho_k \nabla e^{\ell}(X^k, W^k)$$

- Robbins–Monro then ensures that the W^k converge in mean square to an w^* s.t. $0 = g(w^*) = \nabla e(w^*)$, which by local convexity must then be a minimum of e

Takeaways on Online Learning

1. Training standalone MLPs is difficult to parallelize (or to be done under a MapReduce scheme)
2. On the other hand, simple parallel training of several MLPs is very easy
3. Very large samples require on line or (usually) mini–batch training
4. Theoretical results on Stochastic Gradient Descent imply convergence (in a random variable sense)
5. But training may be slow and unstable
6. And choosing the right learning rate may be tricky

4.5 Deep Networks

NN's Second Spring

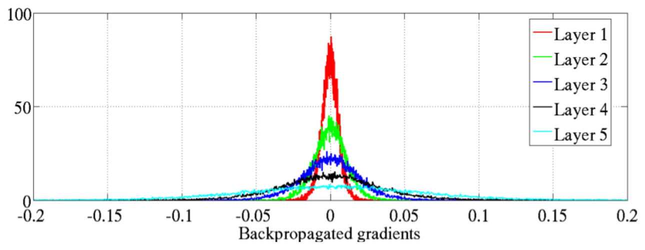
- There was a very intense academic interest in the (by now) standard MLPs in the 1990's
  - Several NN conferences and journals appear
- MLP working and training became well understood
  - Although losing much of neuronal plausibility
- MLPs found relevant applications in many fields
  - They were incorporated into data science tools and products
  - Although hyperparameter selection was (is) costly and had (has) to be done very carefully

NN's Golden Autumn?

- This went on strongly until the late 90's when
  - New relevant contributions decreased
  - New competitors appeared: Boosting, SVMs, Random Forests, Gradient Boosting Regression, ...
- A nagging issue were deeper MLPs
  - One hidden layer MLPs were enough for most applications
  - But nobody knew how to train MLPs with three or more hidden layer

Vanishing Gradients

- One main obstacle was **vanishing gradients**:
  - Consider the weight distribution in a 5 layer MLP



From Glorot & Bengio, AISTATS 2010

  - Gradients in the last (5–th) layer are nonzero but vanish as we go back towards the first layer

  - Training ceases to have any effect and learning stalls at an early, bad minimum

Towards Deep Networks

- Deep Nets: (initially) standard MLPs with 3 or more layers, either fully connected or **convolutional**
- Training impossible even in early 2000:
  - Poor results over limited HW
  - Addressable problems better solved by single layer nets
- First breakthrough around 2007: deep MLP **unsupervised pretraining** using stacked RBMs (Hinton) or autoencoders (Bengio)
- Easier fine-tunning afterwards by standard backprop

Renewed Interest

- Interest in NNs was rekindled and around 2010 the floodgates opened:
  - Large nets with huge number of weights
  - New convolutional layers, regularizations, initializations or activations
  - New techniques appear ... that are not that different from the old ones
- **New mood**: what was impossible before is now much easier and leads to better results
- Major breakthroughs were achieved in significant problems in computer vision and speech recognition

The Boom

- Things go from a mild NN stagnation around 2000 to big explosion in the 2010s
- Relatively large number of contributions and widely attented workshops in mayor conferences (ICML, NIPS)
- Strong groups in leading companies (Google, Baidu, Facebook, Microsoft)
- Great scientific (and mediatic) success: Deep learning. LeCun, Bengio & Hinton (Nature, May 2015)
- New field arising: **Representation Learning**
- New (possible and perhaps more plausible) connections with computational neuroscience (at least for image and audio recognition?)

What Is New In DNNs?

- New and fancy network structures:
  - Convolutional layers (with non–differentiable components)

  - More flexible feedforward connections

- **Automated symbolic backprop derivation**
- Network size: huge number of weights
- Very large sample size (sometimes)
- Many different new cost functions
- New (non differentiable) activations: **ReLUs**
- New regularization: **dropout**, dropconnect
- Recognition that a **good weight initialization is critical**

Changes In DNN Training

- Some things have to change:
  - Batch training becomes unfeasible for huge samples/networks
  - Strict online learning may become impractical as single patterns may get lost in huge samples and networks
- Minibatch training balances these extremes:
  - Choose a minibatch size M (a new DNN parameter?) and at each iteration randomly select M sample patterns
  - Perform SGD or some variant over the minibatch
  - Or even a second order method such as CG

Changes In DNN Training II

- But others do not
- Backprop is still the backbone of gradient computation
  - But it is no longer programmed but derived automatically by **symbolic differentiation**
  - Easily extended to convolutional layer weights
  - Imaginatively extended to non–differentiable elements: just pretend that they are so!
- Minibatch–based Stochastic Gradient Descent (SGD) still is the primary optimization approach
- And several hyper–parameters may still have to be chosen, with no clear cut procedures

Training Time and Technology

- Training time is a key issue as it usually shoots up:

  - Samples may be very large
  - Networks may be huge even for moderate samples
  - More hyperparameters may have to be optimized

- Heavy duty computing needed:
  - Multicore machines: very handy for hyperparameter selection; less so for single network tranining
  - Same true for cloud computing environments
  - GPUs: excellent for single network training
- Best: machines/computing centers with many GPUs

Ad Hoc Programming Tools

- Do–it–yourself programming no longer possible
- Increasingly better tools are being available with very fast evolution
- Initially PyLearn+Theano
- Next Caffe: C++ base with Python interface
- Now Keras: Python platform capable of running on top of Theano and Google's TensorFlow
- Plus open releases by large companies
  - Google‘s TensorFlow (plus TensorBoard)
  - Facebook's Torch, on top of the Lua language
  - Twitter's Autograd for Torch (improving its automatic differentiation capabilities)

New and Fancy New Deep Architectures

- The layers of a feedforward net are nodes in a linear chain graph
  - Backprop is straightforward on such a graph
- But it is also very easy in nets with layers in a DAG
  - They connect different input nodes to different outputs at varying depths and with different layer processing
  - The backprop path is also straightforward
  - And the backprop components at each layer node can be "collated" to the full network gradient
- We just "program" the DAG net defining layer nodes and connecting them in a DAG
  - Then a compiling step yields the forward pass and the backward gradient

- End result: fairly fancy networks
  - Perhaps useful; certainly very costly to train

And Much Better Technologies

- Advanced hardware is a must:
  - GPUs, multicore machines, cloud
- High–level programming:
  - Python as data preprocessing/pipelining + DNN model definition + experimental setup setting + results visualization
  - Python based high–level layers to symbolic GPU backends: Keras (coding in Python), TensorBoard (point and click?)
  - Git as the code and ideas exchange tool
- Also, and very important, new ideas on **initialization, regularization, architectures and learning methods**

Initialization

- If layers with M_i units used, the standard **Glorot–Bengio** (`xavier`) procedure is

$$W_i \sim U\left[-\frac{\sqrt{6}}{\sqrt{M_i + M_{i+1}}}, \frac{\sqrt{6}}{\sqrt{M_i + M_{i+1}}}\right]$$

- It ensures $Var\left(\frac{\partial e}{\partial w_i}\right) \simeq Var\left(\frac{\partial e}{\partial w'_i}\right)$ across successive layers when tanh activations are used
- Gradient vanishing is thus avoided
- **He**'s initialization recommended for ReLU activations
- Pretraining no longer indispensable (at least for large training data sets)

Dropout Regularization

- The extremely large weight numbers of Deep Neural Networks (DNNs) make regularization mandatory
- First choice: standard Tikhonov regularization (i.e., **weight decay**) for regression DNNs with linear output units
- **Dropout** in other fully connected layers, replacing standard output processing $o_i^\ell = f(a_i^\ell) = f(w_i^\ell o^{\ell-1} + b_i^\ell)$ by

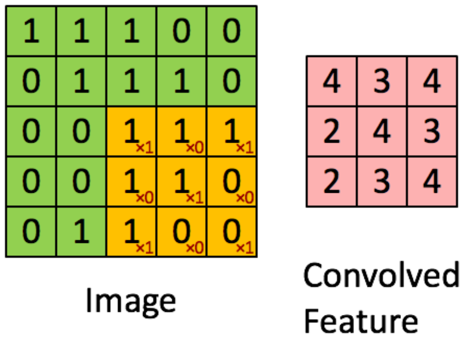
$$o_i^\ell = f(a_i^\ell) = f\left(w_i^\ell (o^{\ell-1} \odot r^\ell) + b_i^\ell\right),$$

  with each r_j^ℓ being 1 with probability p

- It somehows sub-samples a larger network at each layer
- Output errors are backpropagated but the final optimal weights w^* are **downscaled** as $w_f^* = pw^*$
- It adds **randomness** to the final DNN model (and some **independence** for two different models)
- Output errors are backpropagated but the final optimal weights w^* are downscaled as $w_f^* = pw^*$

Convolutional Layers

- Starting assumption: patterns organized in features having a one–, two– or multi–dimensional structure
- Basic processing: to apply a $K \times K$ convolutional filter w over an image patch x_j as $y_j = f(w * x_j + b)$



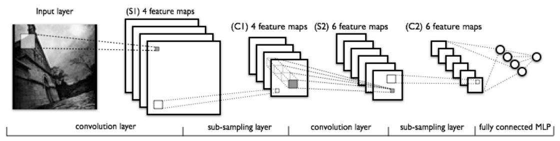
From Stanford's UFLDL Tutorial

Convolutional Layers II

- An $M \times N$ input "image" x is transformed into an $(M - K + 1) \times (N - K + 1)$ output $x' = C(x)$
- This is done over Q input **feature maps** $x_1, \ldots, x_Q$ and creates R output feature maps $x'_1, \ldots, x'_R$
- Then a **pooling** transformation $P(x')$ over $K' \times K'$ patches of each x'_j
  - Possible pooling transforms: averages, max
- We have to learn $Q \times R$ pairs of $K \times K$ filters (w_ℓ, b_ℓ)
  - And decide on K, K' and the pooling transformation
- The forward pass has a cost of $O(Q \times M \times N \times K^2 \times R)$ per pattern, which can be quite costly

Deep Convolutional NNs

- Important goals may be achieved: invariance preservation, structural feature extraction, balancing layer sizes
- Deep Convolutional NNs combine the previous steps
  - An initial number of convolutional layers, followed by
  - A number of fully connected inner product layers and, finally
  - A readout layer that yields the NN's response



A typical architecture for image processing. From Convolutional Neural Networks (LeNet) tutorial

- Possibly with connections and weights in the millions

New Optimization Techniques

- Second order methods across iterations are only possible over small minibatches
- New ideas have been progressively introduced
  - Either refinements of previous approaches more or less sidelined: **Rprop, momentum a la Nesterov**
  - Or often borrowed from other optimization contexts: **Adagrad, Adadelta, Adam**
  - Or simply (overlooked) common sense: minibatch training
- Two main goals:
  - To shorten computation time (obviously)
  - To simplify hyperparameter handling and selection (even more so!)

Takeaways in Deep Networks

1. **Right initialization** crucial
2. **ReLUs** as new activation function
3. Alternative **dropout** regularization for fully connected layer regularization
4. **Convolutional layers** to be used on structured inputs (but processing much costlier)
5. **New optimization** ideas (Adagrad, Adadelta, Adam) to simplify handling of learning rates

6. **Heavy duty computing** environments, particularly for hyperparameterization
7. Need to use tools able to derive **symbolic backpropagation**, even for fancy DAG–like networks possible
8. To read more: M. Nielsen's Neural Networks and Deep Learning online book