MACHINE LEARNING AND PATTERN RECOGNITION

Spring 2004, Lecture 1: Introduction and Basic Concepts

Yann LeCun
The Courant Institute,
New York University
http://yann.lecun.com

Before we get started...

- Course web site: http://www.cs.nyu.edu/ yann
- Evaluation: Assignements (mostly small programming projects) [50%] + larger final project [50%].
- Course mailing list: http://www.cs.nyu.edu/mailman/listinfo/g22_3033_014_sp04
- Text Books: mainly “Element of Statistical Learning” by Hastie, Tibshirani and Friedman, but a number of other books can be used reference material: “Neural Networks for Pattern Recognition” by Bishop, and “Pattern Classification” by Duda, Hart, and Stork....
- ... but we will mostly use resarch papers and tutorials.
- Prerequisites: linear algebra, probability theory. You might want to brush up on multivariate calculus (partial derivatives ...), optimization (least square method...), and the method of Lagrange multipliers for constrained optimization.
- Programming projects: can be done in any language, but I **STRONGLY** recommend to use Lush (http://lush.sf.net).

What is Learning?

- Learning is improving performance through experience
- Pretty much all animals with a central nervous system are capable of learning (even the simplest ones).
- What does it mean for a computer to learn? Why would we want them to learn? How do we get them to learn?
- We want computers to learn when it is too difficult or too expensive to program them directly to perform a task.
- Get the computer to program itself by showing examples of inputs and outputs.
- In reality: write a "parameterized" program, and let the learning algorithm find the set of parameters that best approximates the desired function or behavior.

Different Types of Learning

- Supervised Learning: given training examples of inputs and corresponding outputs, produce the “correct” outputs for new inputs. Example: character recognition.
- Reinforcement Learning (similar to animal learning): an agent takes inputs from the environment, and takes actions that affect the environment. Occasionally, the agent gets a scalar reward or punishment. The goal is to learn to produce action sequences that maximize the expected reward (e.g. driving a robot without bumping into obstacles). I won’t talk much about that in this course.
- Unsupervised Learning: given only inputs as training, find structure in the world: discover clusters, manifolds, characterize the areas of the space to which the observed inputs belong (e.g.: clustering, probability density estimation, novelty detection, compression, embedding).

Related Fields

- Statistical Estimation: statistical estimation attempts to solve the same problem as machine learning. Most learning techniques are statistical in nature.
- Pattern Recognition: pattern recognition is when the output of the learning machine is a set of discrete categories.
- Neural Networks: neural nets are now one many techniques for statistical machine learning.
- Data Mining: data mining is a large application area for machine learning.
- Adaptive Optimal Control: non-linear adaptive control techniques are very similar to machine learning methods.
- Machine Learning methods are an essential ingredient in many fields: bio-informatics, natural language processing, web search and text classification, speech and handwriting recognition, fraud detection, financial time-series prediction, industrial process control, database marketing....

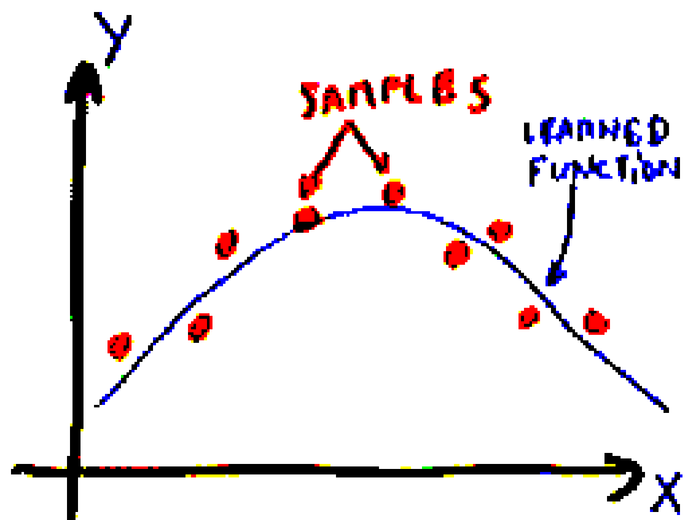
Applications

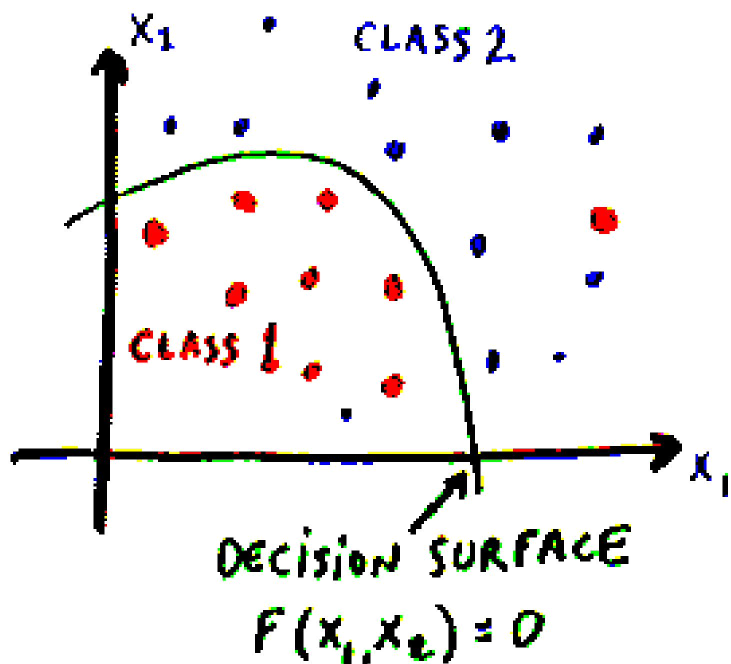
- handwriting recognition, OCR: reading checks and zipcodes, handwriting recognition for tablet PCs.
- speech recognition, speaker recognition/verification
- security: face detection and recognition, event detection in videos.
- text classification: indexing, web search.
- computer vision: object detection and recognition.
- diagnosis: medical diagnosis (e.g. pap smears processing)
- adaptive control: locomotion control for legged robots, navigation for mobile robots, minimizing pollutant emissions for chemical plants, predicting consumption for utilites...
- fraud detection: e.g. detection of "unusual" usage patterns for credit cards or calling cards.
- database marketing: predicting who is more likely to respond to an ad campaign.
- (...and the antidote) spam filtering.
- games (e.g. backgammon).
- Financial prediction (many people on Wall Street use machine learning).

Demos / Concrete Examples

- Handwritten Digit Recognition: supervised learning for classification
- Handwritten Word Recognition: weakly supervised learning for classification with many classes
- Face detection: supervised learning for detection (faces against everything else in the world).
- Object Recognition: supervised learning for detection and recognition with highly complex variabilities
- Robot Navigation: supervised learning and reinforcement learning for control.

Two Kinds of Supervised Learning

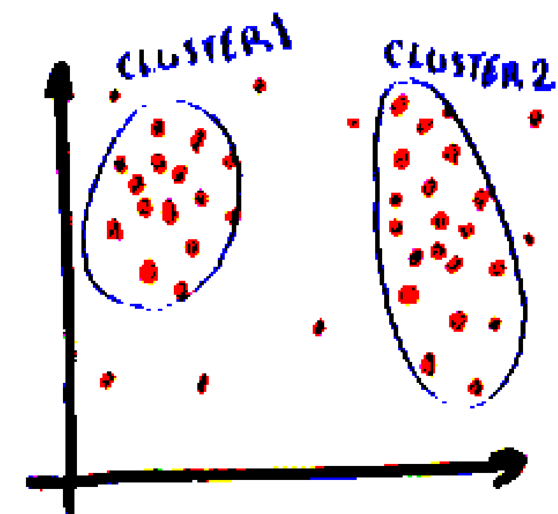


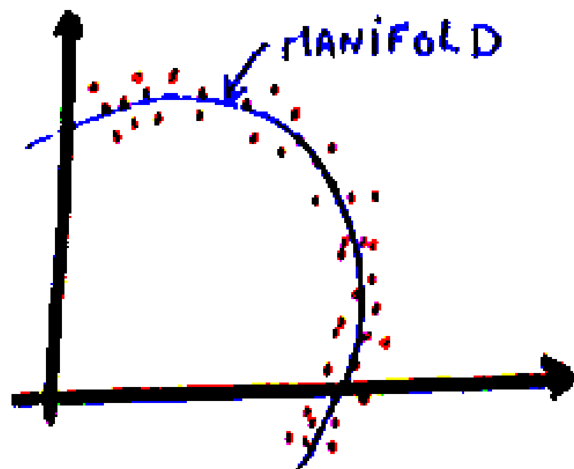


- Regression: also known as “curve fitting” or “function approximation”. Learn a continuous input-output mapping from a limited number of examples (possibly noisy).
- Classification: outputs are discrete variables (category labels). Learn a decision boundary that separates one class the the other. Generally, a “confidence” is also desired (how sure are we that the input belongs to the chosen category).

Unsupervised Learning

Unsupervised learning comes down to this: if the input looks like the training samples, output a small number, if it doesn't, output a large number.



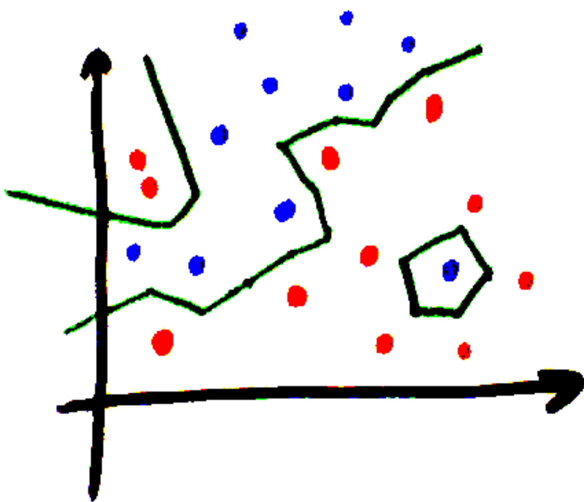


- This is a horrendously ill-posed problem in high dimension. To do it right, we must guess/discover the hidden structure of the inputs. Methods differ by their assumptions about the nature of the data.
- A Special Case: Density Estimation. Find a function f such $f(X)$ approximates the probability density of X, $p(X)$, as well as possible.
- Clustering: discover "clumps" of points
- Embedding: discover low-dimensional manifold or surface near which the data lives.
- Compression/Quantization: discover a function that for each input computes a compact "code" from which the input can be reconstructed.

Learning is NOT Memorization

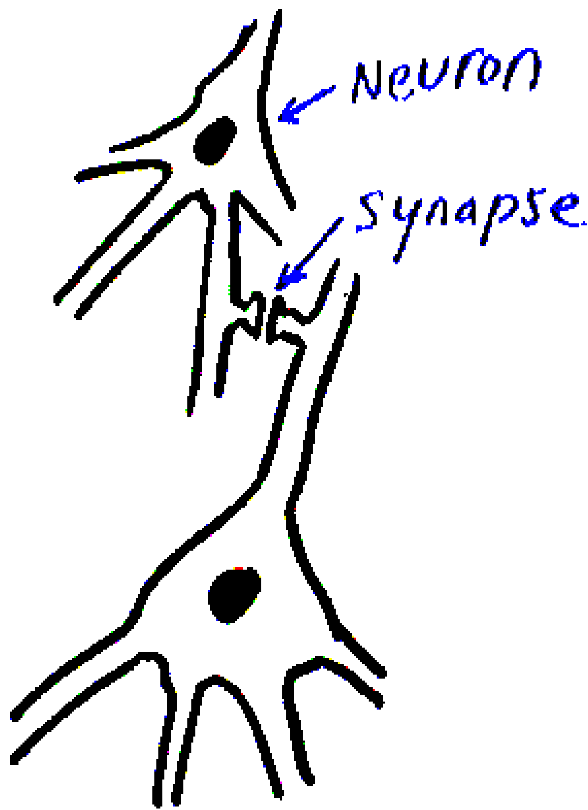
- rote learning is easy: just memorize all the training examples and their corresponding outputs.
- when a new input comes in, compare it to all the memorized samples, and produce the output associated with the matching sample.
- PROBLEM: in general, new inputs are different from training samples.
- The ability to produce correct outputs or behavior on previously unseen inputs is called GENERALIZATION.
- rote learning is memorization without generalization.
- The big question of Learning Theory (and practice): how to get good generalization with a limited number of examples.

A Simple Trick: Nearest Neighbor Matching



- Instead of insisting that the input be exactly identical to one of the training samples, let's compute the "distances" between the input and all the memorized samples (aka the prototypes).
- 1-Nearest Neighbor Rule: pick the class of the nearest prototype.
- K-Nearest Neighbor Rule: pick the class that has the majority among the K nearest prototypes.
- PROBLEM: What is the right distance measure?
- PROBLEM: This is horrendously expensive if the number of prototypes is large.
- PROBLEM: do we have any guarantee that we get the best possible performance as the number of training samples increases?

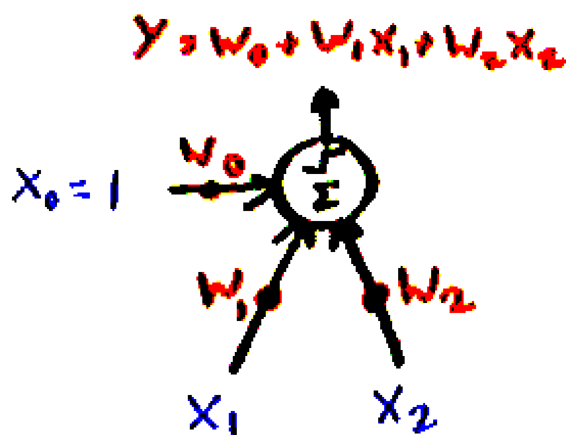
How Biology Does It

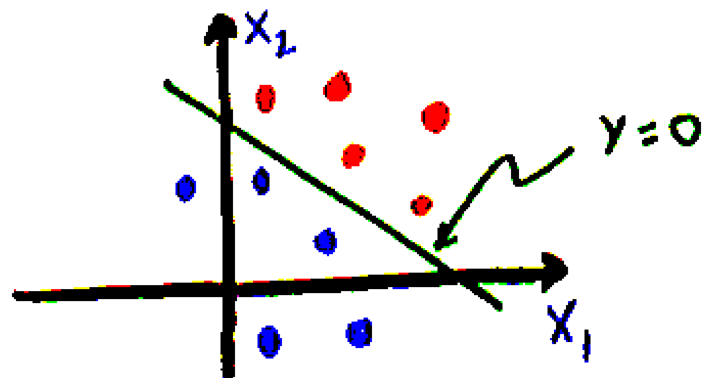


- The first attempts at machine learning in the 50's, and the development of artificial neural networks in the 80's and 90's were inspired by biology.
- Nervous Systems are networks of neurons interconnected through synapses
- Learning and memory are changes in the "efficacy" of the synapses
- HUGE SIMPLIFICATION: a neuron computes a weighted sum of its inputs (where the weights are the synaptic efficacies) and fires when that sum exceeds a threshold.
- Hebbian learning (from Hebb, 1947): synaptic weights change as a function of the pre- and post-synaptic activities.
- orders of magnitude: each neuron has 10^3 to 10^5 synapses. Brain sizes (number of neurons): house fly: 10^5; mouse: 5.10^6, human: 10^{10}.

The Linear Classifier

Historically, the Linear Classifier was designed as a highly simplified model of the neuron (McCulloch and Pitts 1943, Rosenblatt 1957):





$$y = f(\sum_{i=0}^{i=N} w_i x_i)$$

With f is the threshold function: $f(z) = 1$ iff $z > 0$, $f(z) = -1$ otherwise. x_0 is assumed to be constant equal to 1, and w_0 is interpreted as a bias.
In vector form: $W = (w_0, w_1 .... w_n), X = (1, x_1 ... x_n)$:

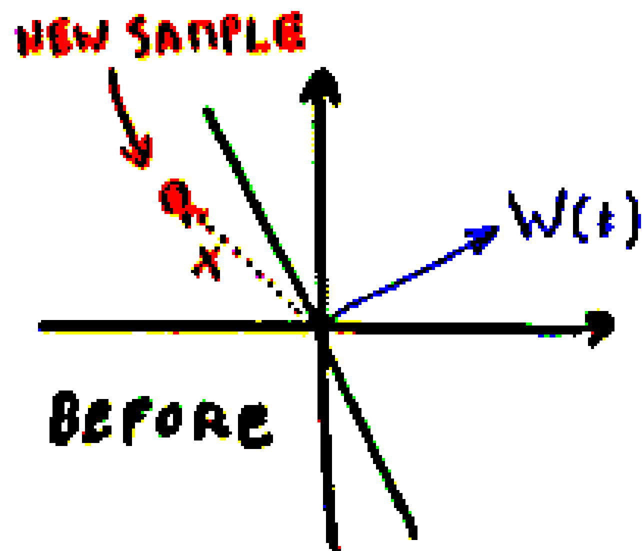
$$y = f(W'X)$$

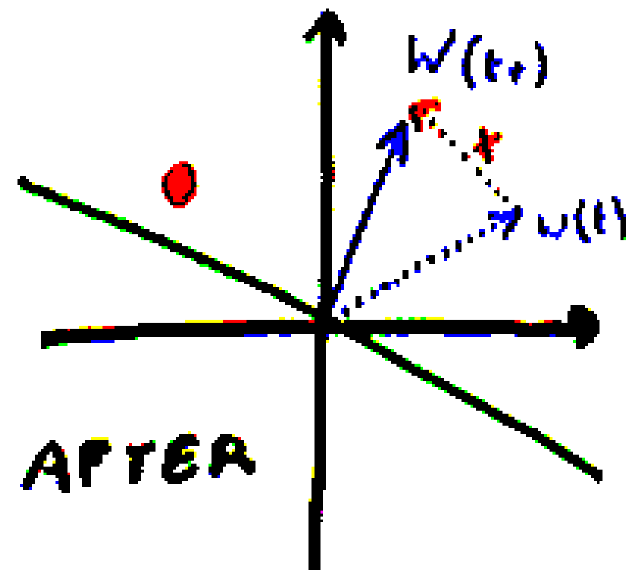
The hyperplane $W'X = 0$ partitions the space in two categories. W is orthogonal to the hyperplane.

Vector Inputs

With vector-based classifiers such as the linear classifier, we must represent objects in the world as vectors.
Each component is a measurement or a feature of the the object to be classified.
For example, the grayscale values of all the pixels in an image can be seen as a (very high-dimensional) vector.

A Simple Idea for Learning: Error Correction





We have a **training set** consisting of P input-output pairs: $(X^1, d^1), (X^2, d^2), ...(X^P, d^P)$.
A very simple algorithm:
- show each sample in sequence repetitively
- if the output is correct: do nothing
- if the output is -1 and the desired output +1: increase the weights whose inputs are positive, decrease the weights whose inputs are negative.
- if the output is +1 and the desired output -1: decrease the weights whose inputs are positive, increase the weights whose inputs are negative.

More formally, for sample p:

$$w_i(t+1) = w_i(t) + (d_i^p - f(W'X^p))x_i^p$$

This simple algorithm is called the Perceptron learning procedure (Rosenblatt 1957).

The Perceptron Learning Procedure

Theorem: If the classes are linearly separable (i.e. separable by a hyperplane), then the Perceptron procedure will converge to a solution in a finite number of steps.
Proof: Let's denote by W^* a normalized vector in the direction of a solution. Suppose all X are within a ball of radius R. Without loss of generality, we replace all X^p whose d^p is -1 by $-X^p$, and set all d^p to 1. Let us now define the margin $M = min_p W^* X^p$. Each time there is an error, $W.W^*$ increases by at least $X.W^* \geq M$. This means $W_{final}.W^* \geq NM$ where N is the total number of weight updates (total number of errors). But, the change in square magnitude of W is bounded by the square magnitude of the current sample X^p, which is itself bounded by R^2. Therefore, $|W_{final}|^2 \leq NR^2$. combining the two inequalities $W_{final}.W^* \geq NM$ and $|W_{final}| \leq \sqrt{N}R$, we have

$$W_{final}.W^*/|W_{final}| \geq \sqrt(N)M/R$$

. Since the left hand side is upper bounded by 1, we deduce

$$N \leq R^2/M^2$$

Good News, Bad News

The perceptron learning procedure can learn a linear decision surface, **if such a surface exists** that separates the two classes. If no perfect solution exists, the perceptron procedure will keep wobbling around.

What class of problems is **Linearly Separable**, and learnable by a Perceptron?

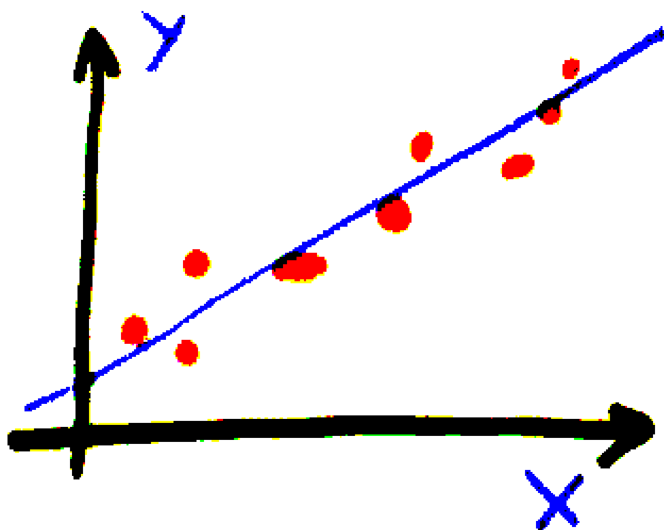
There are many interesting applications where the data can be represented in a way that makes the classes (nearly) linearly separable: e.g. text classification using "bag of words" representations (e.g. for spam filtering).

Unfortunately, the really interesting applications are generally not linearly separable.

This is why most people abandonned the field between the late 60's and the early 80's.

We will come back to the linear separability problem later.

Regression, Mean Squared Error



Regression or function approximation is finding a function that approximates a set of samples as well as possible.

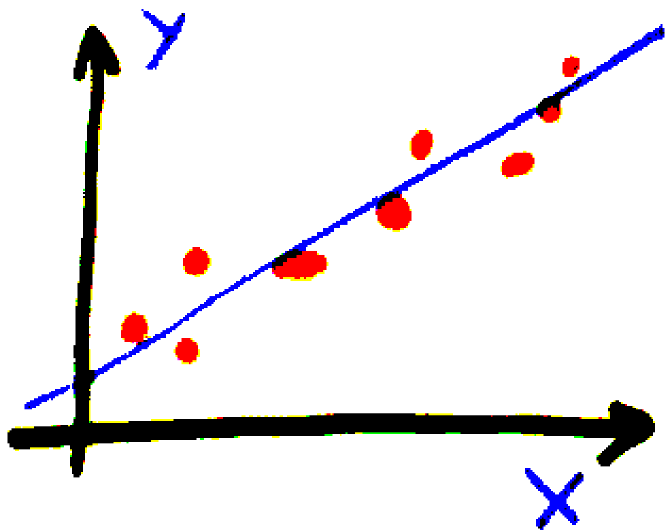
Classic example: linear regression. We are given a set of pairs $(X^1, y^1), (X^2, y^2)....(X^P, y^P)$, and we must find the parameters of a linear function that best approximates the samples in the least square sense, i.e. that minimizes the energy function $E(W)$:

$$W^* = argmin_W E(W) = argmin_W \frac{1}{2P} \sum_{i=1}^{P} (y^i - W' X^i)^2$$

The solution is characterized by:

$$\frac{\partial E(W)}{\partial W} = 0 \Leftrightarrow \frac{1}{P} \sum_{i=1}^{P} (y^i - W' X^i) X^{i'} = 0$$

Regression, Solution



$$\sum_{i=1}^{P} y^i X^i - [\sum_{i=1}^{P} X^i X^{i'}] W = 0$$

This is a linear system that can be solved with a number of traditional numerical methods (although it may be ill-conditioned or singular).

If the **covariance matrix** $[\sum_{i=1}^{P} X^i X^{i'}]$ is non singular, the solution is:

$$W* = [\sum_{i=1}^{P} X^i X^{i'}]^{-1} \sum_{i=1}^{P} y^i X^i$$

Regression, Iterative Solution

Gradient descent minimization:

$$w_k(t+1) = w_k(t) - \eta \frac{\partial \sum_{i=1}^{P} (d^i - W(t)'X^i)^2}{\partial w_k(t)}$$

Batch gradient descent:

$$w_k(t+1) = w_k(t) - \eta \sum_{i=1}^{P} (d^i - W(t)'X^i) x_k^i$$

Converges for small values of η (more on this later).

Regression, Online/Stochastic Gradient

Online gradient descent, aka Stochastic Gradient:

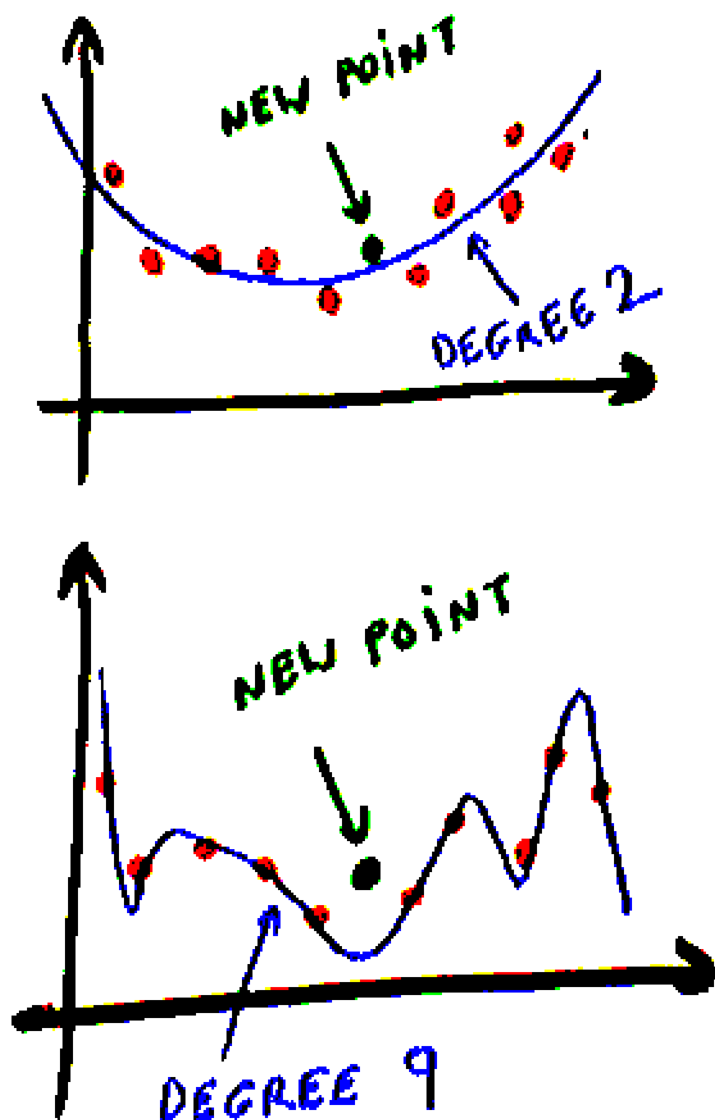
$$w_k(t+1) = w_k(t) - \eta(t)(d^i - W(t)'X^i)x_k^i$$

No sum! The average gradient is replaced by its instantaneous value. The convergence analysis of this is very tricky. One condition for convergence is that $\eta(t)$ is decreased according to a schedule such that
$\sum_t \eta(t)^2$ converges while $\sum_t \eta(t)$ diverges.
One possible such sequence is $\eta(t) = \eta_0/t$.
In many practical situation stochastic gradient is **enormously faster** than batch gradient.
We can also use second-order methods, but we will keep that for later.

MSE for Classification

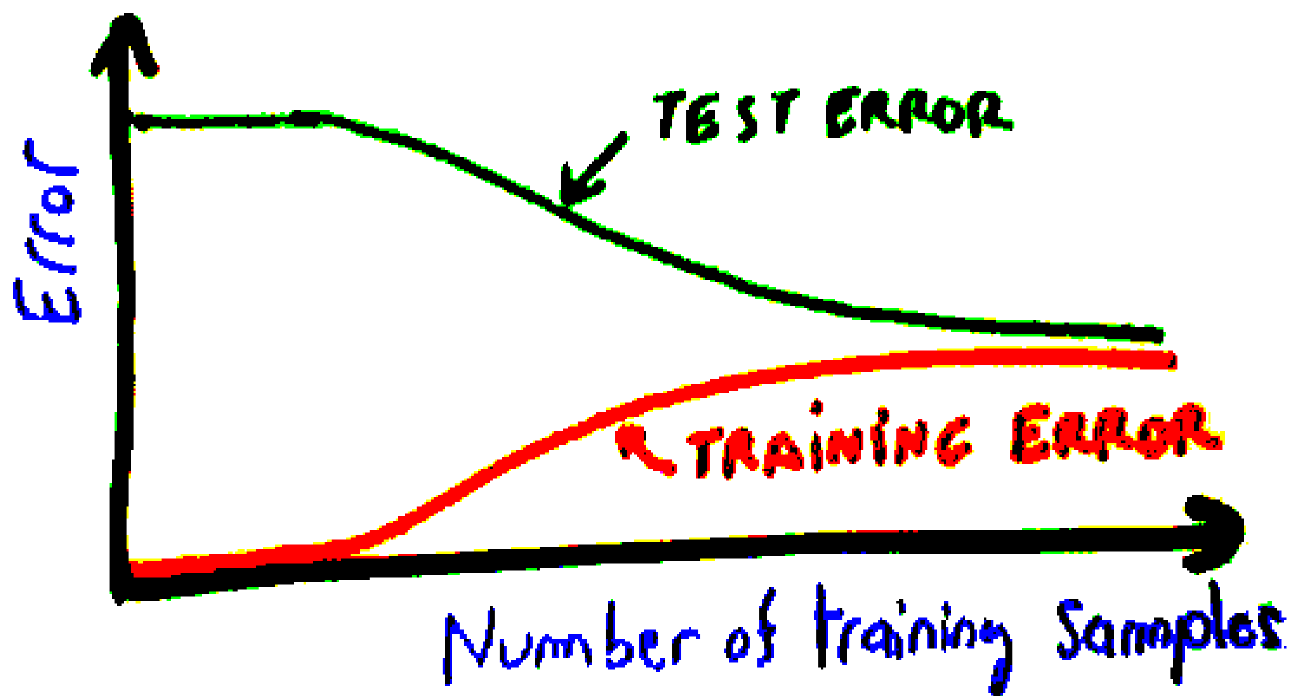
- We can use the Mean Squared Error criterion with a linear regressor to perform classification (although this is clearly suboptimal).
- Simply perform linear regression with binary targets: +1 for class 1, -1 for class 2.
- This is called the Adaline algorithm (Widrow-Hoff 1960).

A Richer Class of Functions



What if we know that our data is not linear? We can use a richer **family of functions**, e.g. polynomials, sum of trigonometric functions....
PROBLEM: if the family of functions is too rich, we run the risk of **overfitting** the data. If the family is too restrictive we run the risk of not being able to approximate the training data very well.
QUESTIONS: How can we choose the richness of the family of functions? Can we predict the performance on new data as a function of the training error and the richness of the family of functions?
Simply minimizing the training error may not give us a solution that will do well on new data.

Training Error, Test Error

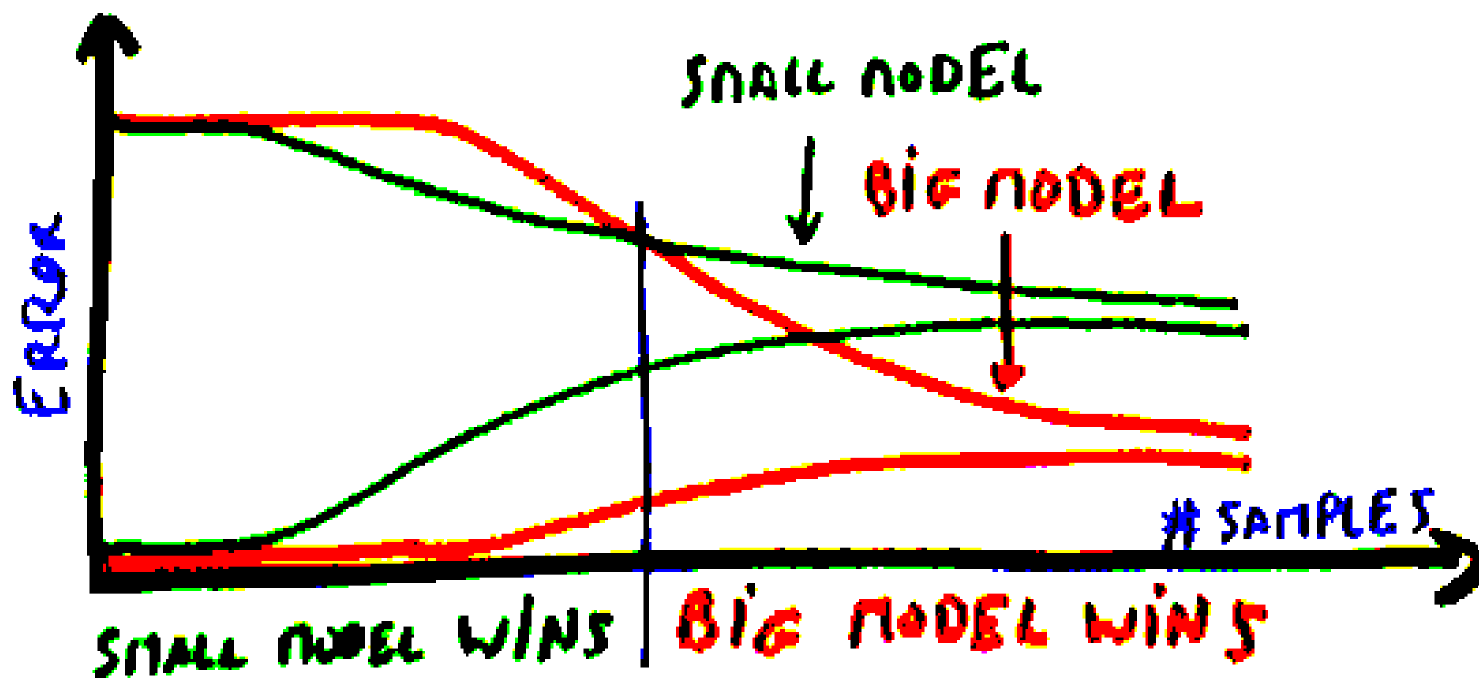


What we are **really** interested in is good performance on unseen data. In practice, we often partition the dataset into two subsets: a training set and a test set. We train the machine on the training set, and measure its performance on the test set.

The error on the training set (the average of the energy function) is often called the **emprical risk**. The average energy on an infinite test set drawn from the same distribution as the training set is often called the **expected risk**.

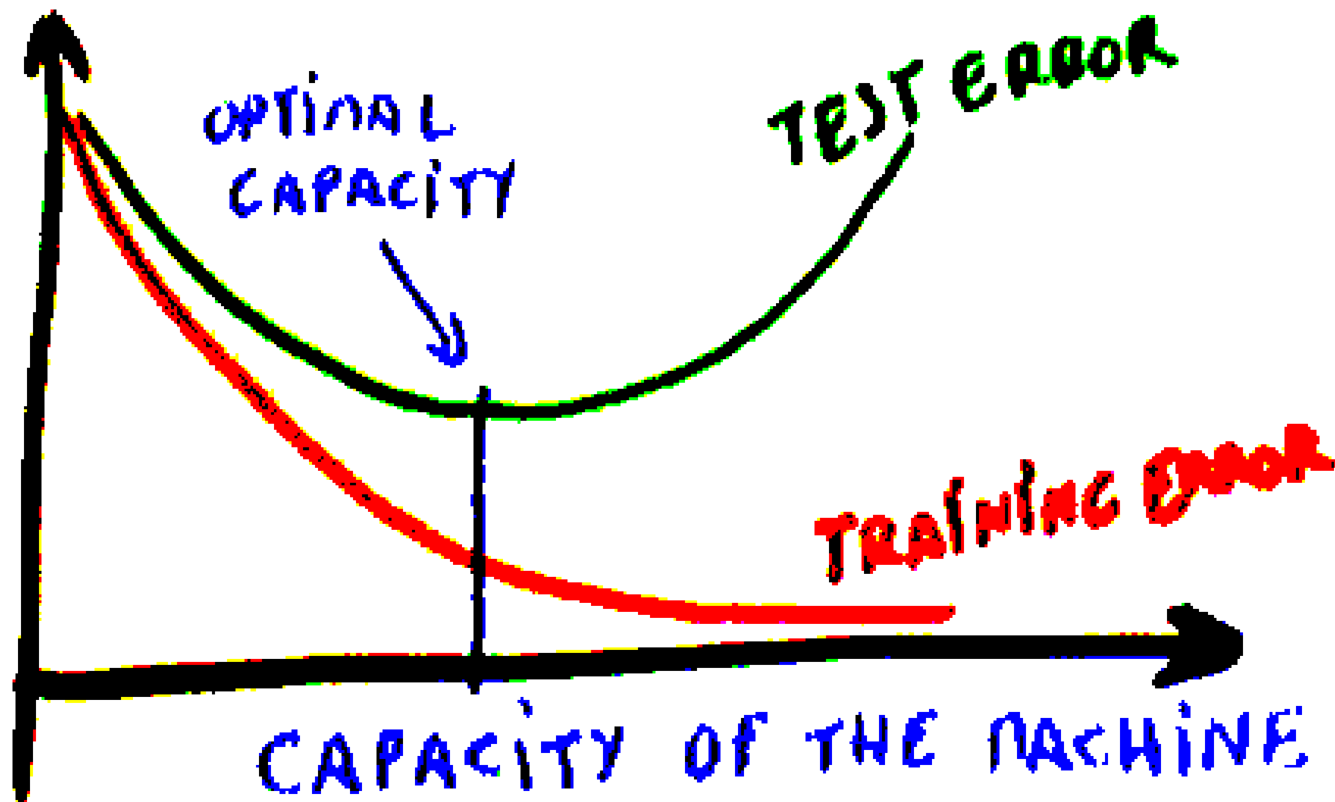
The number of training samples at which the training error leaves zero is called the "capacity" of the learning machine.

Learning Curves



Simple models: may not do well on the training data, but the difference between training and test error quickly drops.
Rich models: will learn the training data, but the difference between training and test error can be large.
How much a model deviates from the desired mapping on average is called the **bias** of the family of functions. How much the output of a model varies when different drawings of the training set are used is called the **variance**. There is a dilemma between bias and variance.

Optimal Over-Parameterization



The curve of training error and test error for a given training set size, as a function of the capacity of the machine (the richness of the class of function) has a minimum. The is the optimal size for the machine.

A Penalty Term

What we need to minimize is an energy function of the form:

$$L(W) = \sum_{i=1}^{P} E(W, X^i, d^i) + H(W)$$

where E is the conventional energy function (e.g. squared error) and $H(W)$ is a **regularization term** that penalizes solutions taken from "rich" families of function more than those taken from "leaner" families of functions.

- How we pick this penalty term is entirely up to us! No theory will tell us how to build the penalty function.
- By picking the family of function and the penalty term, we choose an **inductive bias**, i.e. we privilege certain solutions over others.
- Sometimes we do not need to add an explicit penalty term because it is built implicitely into the optimization algorithm (more on this later).

Induction Principles

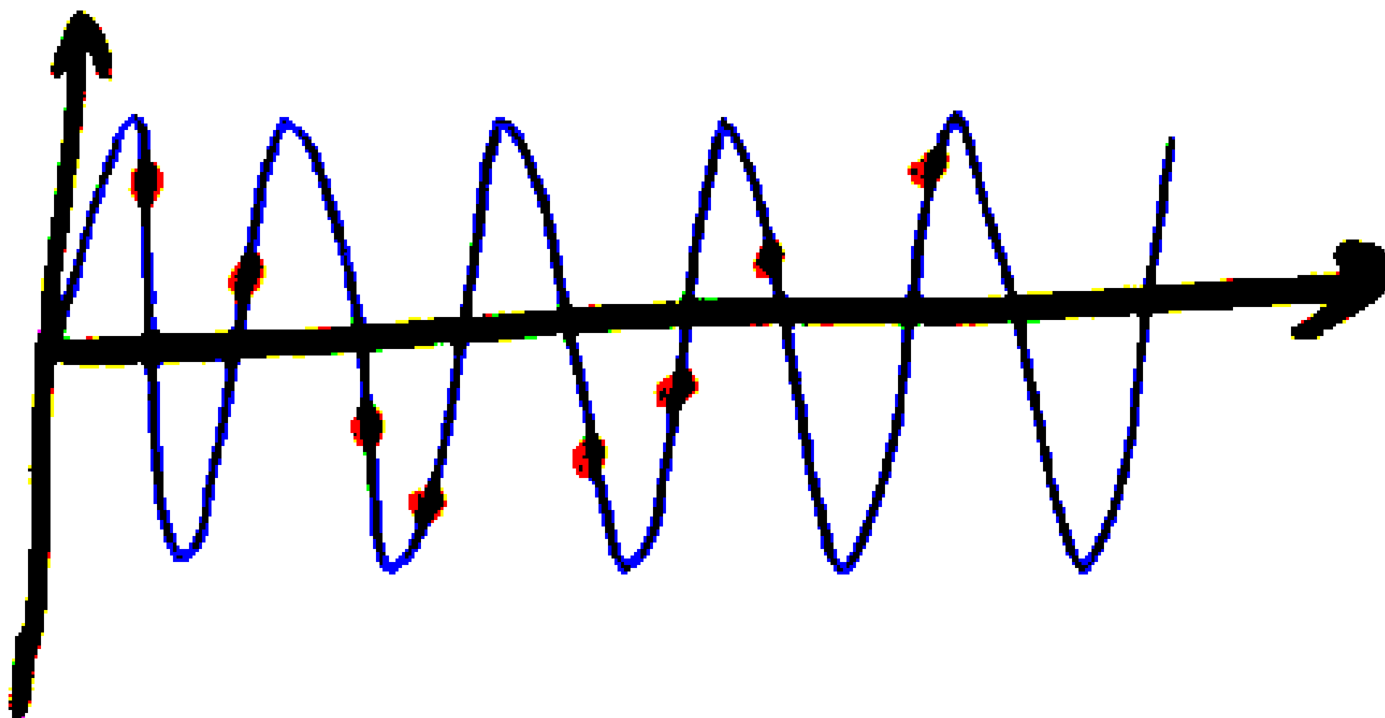
Assuming our samples are drawn from a distribution $P(X, Y)$, what we really want to minimize with respect to our paramter W is the expected risk:

$$E_{expected} = \int E(W, X, Y)P(X, Y)dXdY.$$

but we do not have access to $P(X, Y)$, we only have access to a few training samples drawn from it.
The method we will employ to replace the expected risk by another quantity that we can minimize is called the **induction principle**.
The simplest induction principle is called **Empirical Risk Minimization** and simply consists in minimizing the training error.
The alternative, which is to include a penalty term to penalize members of "rich" families of functions is called **Structural Risk Minimization**.

Examples of Penalty Terms

What about a regularization term that simply counts the number of free parameters in the machine?
It works in some cases, but not in others. For example, the function $a.\sin(wx + b)$ has only three parameters but can exactly fit as many points as we want. This is an example of a very high-capacity function with just a few parameters:



Examples of Penalty Terms

Ridge Regression: penalizes large values of the parameters.

$$L(W) = \frac{1}{2P}\sum_{i=1}^{P}(d^i - W'X^i)^2 + \lambda|W|^2$$

Direct solution:

$$W* = [\frac{1}{P}\sum_{i=1}^{P} X^i X^{i'} + \lambda I]^{-1} \sum_{i=1}^{P} y^i X^i$$

Lasso: penalize all parameter values with a linear term (this tends to shrink small, useless parameters to 0):

$$L(W) = \frac{1}{2P}\sum_{i=1}^{P}(d^i - W'X^i)^2 + \lambda|W|$$

Minimum Description Length

A popular way of deriving penalty terms is the Minimum Description Length Principle.

$$L(W) = \sum_{i=1}^{P} E(W, X^i, d^i) + H(W)$$

The idea is to view the objective function as the number of bits necessary to transmit the training data. The penalty term counts the number of bits to code the hypothesis (e.g. the value of the parameter vector), and the error term counts the number of bits to code the residual error (i.e. the difference between the predicted output and the real output. Using efficient coding, the length of the code for a symbol is equal to the log of the probability of that symbol.

MDL: Learning as Compression

MDL comes from the idea that "compact" internal representations of a set of data are preferable to non compact ones This principle is know as Occam's Razor: do not multiply hypotheses beyond the strict necessary.

Example:

complete this sequence: 01010101010101010.......

now complete that one : 01100010110010001......

The second sequence looks "random", we cannot find a compact "theory" for it.

QUESTION: How do we measure randomness?

Sometimes, a simple rule exists but is very hard to find.

Example: 9265358979323846264338328.....

Can you guess?

Measuring Randomness?

The Kolmogoroff/Chaitin/Solomonoff theory of complexity gives us a theoretical framework:
The KCS complexity of a string of bits S relative to computer C is the length of the shortest program that will output S when run on C.
Good news: the complexity given by two different universal computers differ at most by a constant (the size of the program that will make one computer emulate the other).
Bad News 1: that constant can be very, very, very large. So in practice, there is no absolute measure randomness for finite strings.
Bad New 2: the KCS complexity of a string is non-computable in general (you can't enumerate all the programs, because some won't halt).
Although this is a very rich and cool theoretical concept, we can't really use it in practice.