MACHINE LEARNING AND PATTERN RECOGNITION

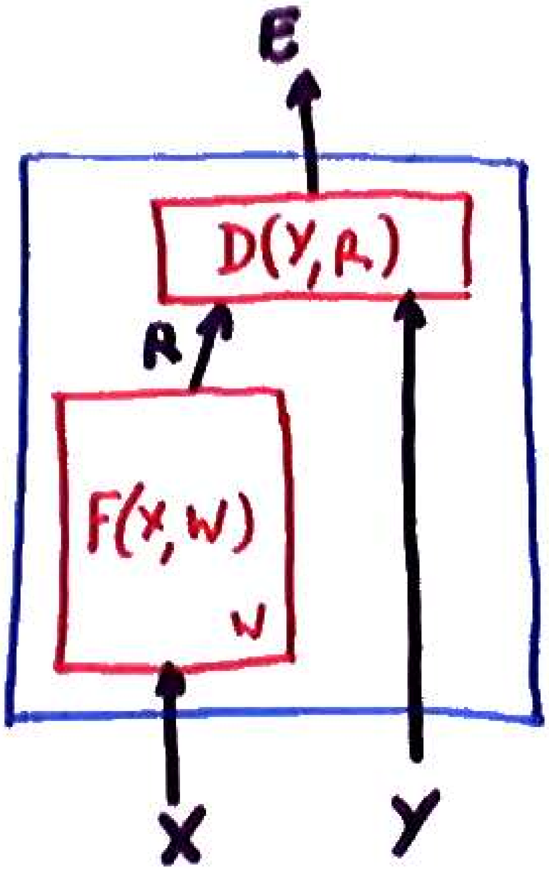
Spring 2004, Lecture 4: Intro to Gradient-Based Learning I: Beyond Linear Classifiers

Yann LeCun

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

Energy-Based Feed-Forward Supervised Learning

$$L(W) = \sum_i [E(Y^i, X^i, W) + \frac{1}{\beta} \log \int \exp(-\beta E(Y, X^i, W)) dY] + H(W)$$



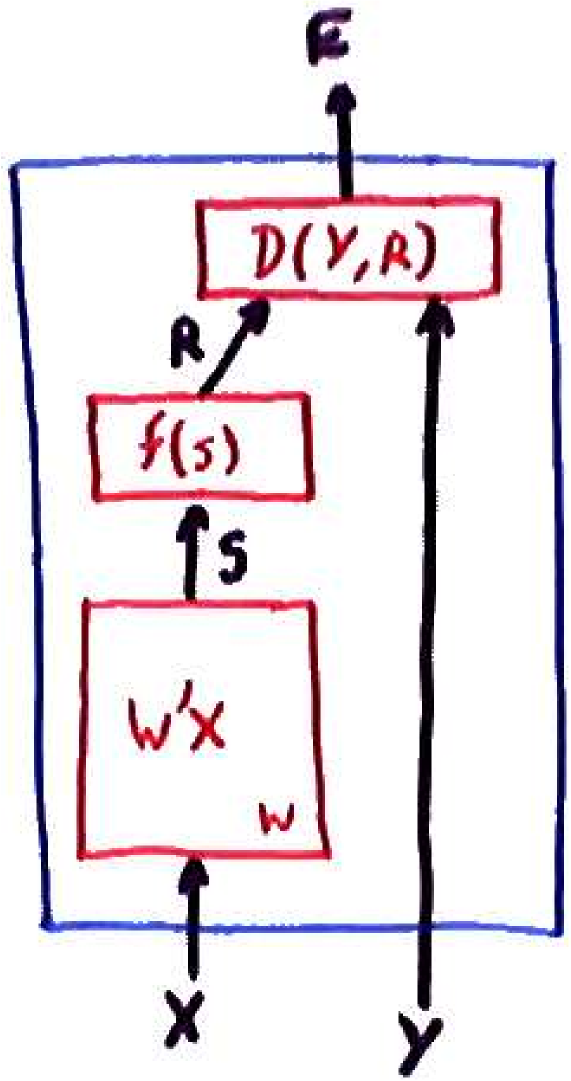
- Learning comes down to finding the W that minimizes an objective function averaged over a training set.
- A *feed-forward supervised* system parameterizes E as follows:

  $$E(Y, X, W) = D(Y, F(X, W))$$

  where $F(X, W)$ is a suitably chosen *discriminant function* parameterized by W, and D is an approrpiately chosen dissimilarity measure.
- A popular example would be

  $$E(Y, X, W) = ||Y - F(X, W)||^2$$

Linear Machines



- The learning algorithms we have seen so far (perceptron, linear regression, logistic regression) are of that form, with the assumption that $F(X, W)$ only depends on the dot product of W and X.
- In other words, The E function of linear classifiers can be written as:

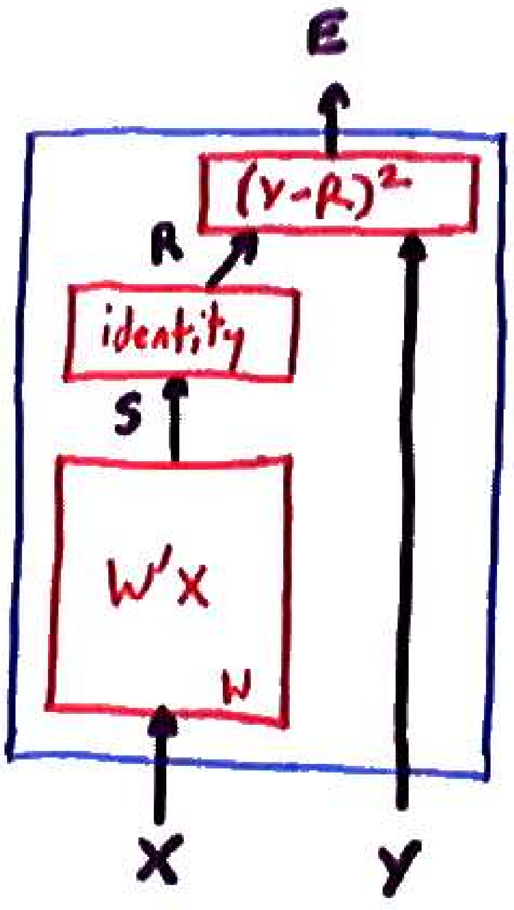
$$E(Y, X, W) = D(Y, f(W'X))$$

where f is a monotonically increasing function.

- in the following, we assume $Y = -1$ for class 1, and $Y = +1$ for class 2.

Linear Regression

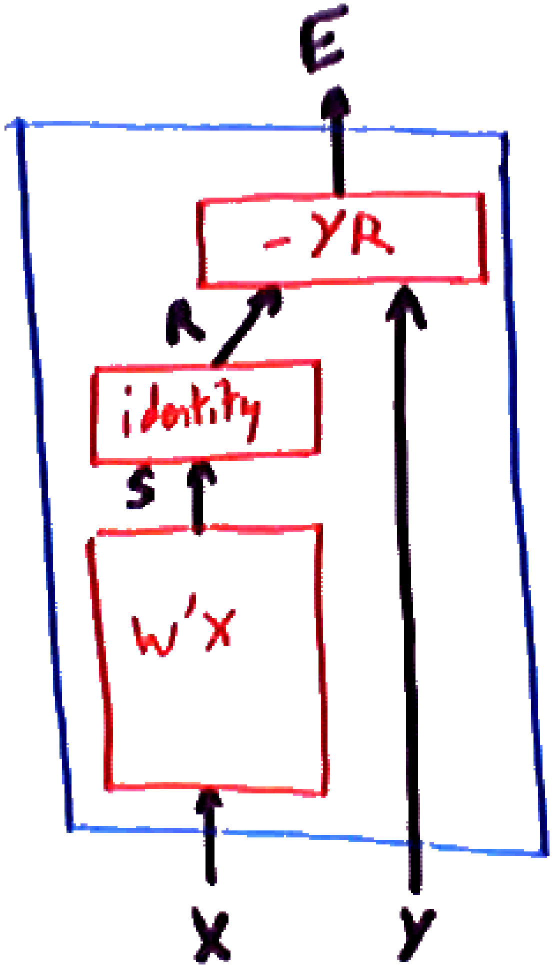
$$L(W) = \sum_i [E(Y^i, X^i, W) + \log \int \exp(-E(Y, X^i, W))dY]$$



- $R = W'X$
- $E(Y, X, W) = D(Y, R) = \frac{1}{2}||Y - R||^2$
- $L(W) = \sum_i D(Y^i, W'X^i) - \text{Constant}$
- $\frac{\partial L}{\partial W} = \sum_i \frac{\partial D(Y^i, R)}{\partial R} \frac{\partial R}{\partial W}$
- $\frac{\partial L}{\partial W} = \sum_i \frac{\partial D(Y^i, R)}{\partial R} \frac{\partial (W'X^i)}{\partial W} = \sum_i (R - Y^i) X^i$
- descent: $W \leftarrow W + \eta(Y^i - R)X^i$

Perceptron

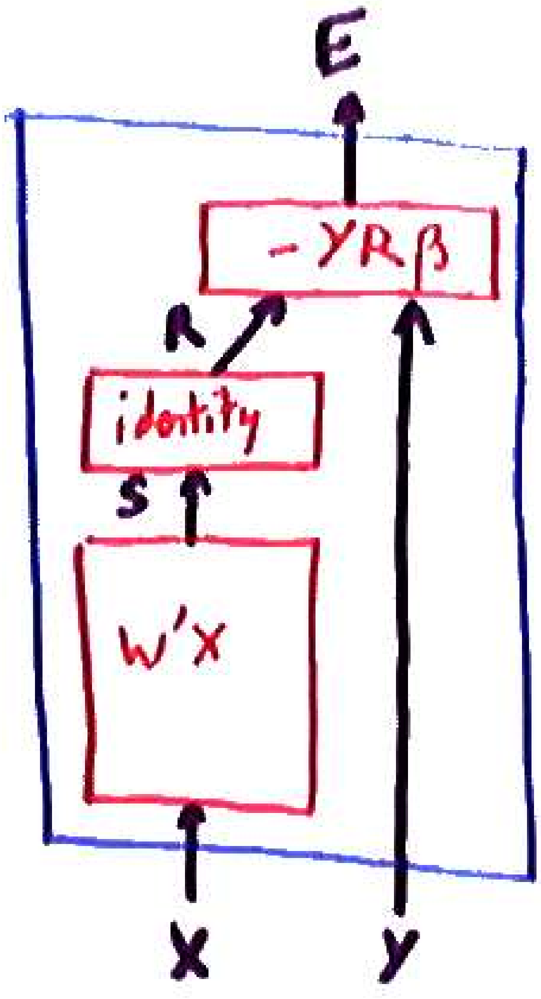
$$L(W) = \sum_i E(Y^i, X^i, W) - \min_Y E(Y, X^i, W)$$



- $R = W'X$
- $E(Y, X, W) = D(Y, R) = -YR$
- $Y \in \{-1, +1\}$, hence $\min_Y -YR = -\text{sign}(R)R$ where $\text{sign}(R) = 1$ iff $R > 0$, and -1 otherwise.
- $L(W) = \sum_i -(Y^i - \text{sign}(R))R$
- $\frac{\partial L}{\partial W} = \sum_i \frac{\partial -(Y^i - \text{sign}(R))R}{\partial R} \frac{\partial R}{\partial W}$
- $\frac{\partial L}{\partial W} = \sum_i -(Y^i - \text{sign}(W'X^i))X^i$
- descent: $W \leftarrow W + \eta(Y^i - \text{sign}(W'X^i))X^i$

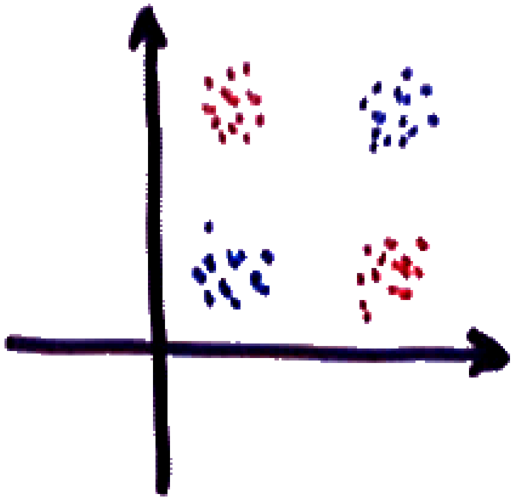
Logistic Regression

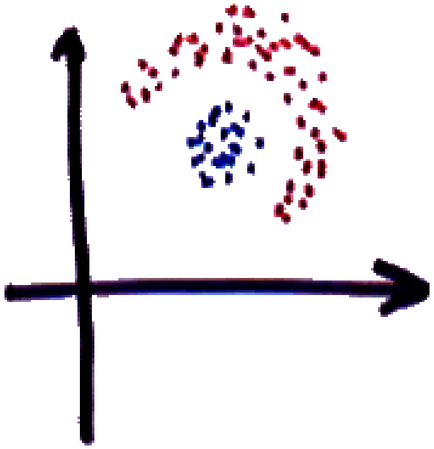
$$L(W) = \sum_i [E(Y^i, X^i, W) + \log(\exp(E(-1, X^i, W)) + \exp(E(+1, X^i, W)))]$$



- $R = \frac{1}{2} W'X$
- $E(Y, X, W) = D(Y, R) = -\frac{1}{2} Y R = -\frac{1}{2} Y W'X$
- $L(W) = \sum_i \log(1 + \exp(-Y^i W'X^i))$
- $\frac{\partial L}{\partial W} = \sum_i \frac{\partial D(Y^i, R)}{\partial R} \frac{\partial S}{\partial W}$
- $\frac{\partial L}{\partial W} = \sum_i -\left(\frac{Y^i+1}{2} - \frac{1}{1+\exp(-W'X^i)}\right) X^i$
- descent: $W \leftarrow W + \eta \left(\frac{Y^i+1}{2} - \frac{1}{1+\exp(-W'X^i)}\right) X^i$

Limitations of Linear Machines

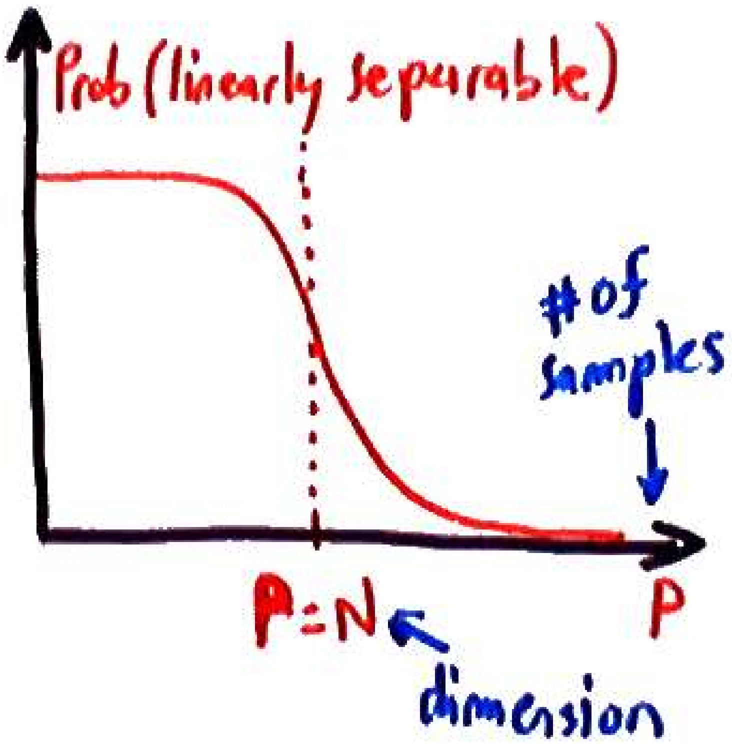




The *Linearly separable* dichotomies are the partitions that are realizable by a linear classifier (the boundary between the classes is a hyperplane).

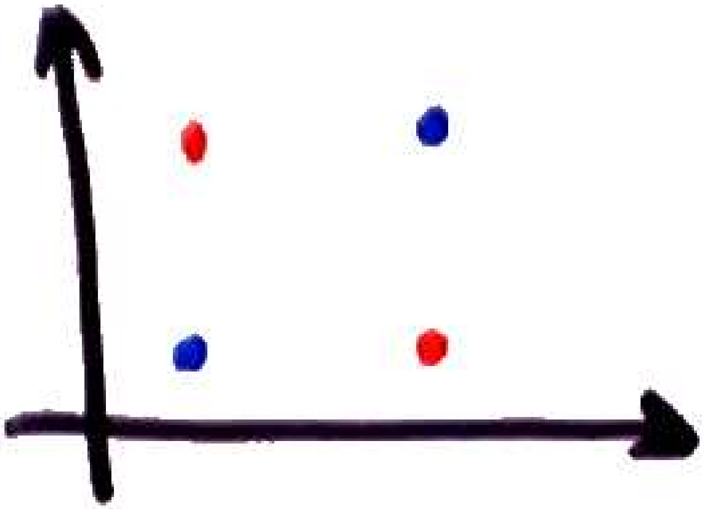
Number of Linearly Separable Dichotomies

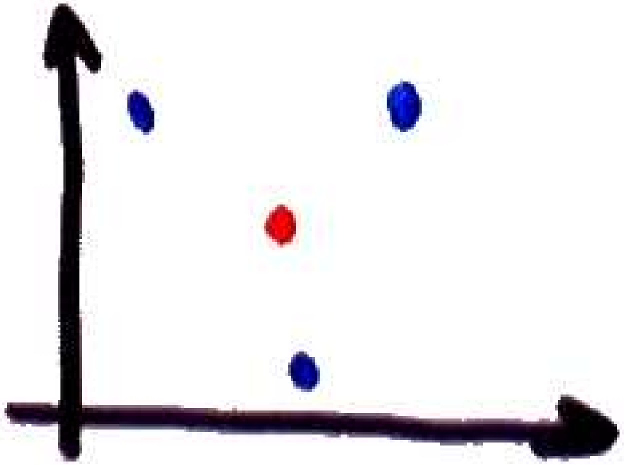
The probability that P samples of dimension N are linearly separable goes to zero very quickly as P grows larger than N (Cover's theorem, 1966).



- Problem: there are 2^P possible dichotomies of P points.
- Only about N are linearly separable.
- If P is larger than N, the probability that a random dichotomy is linearly separable is very, very small.

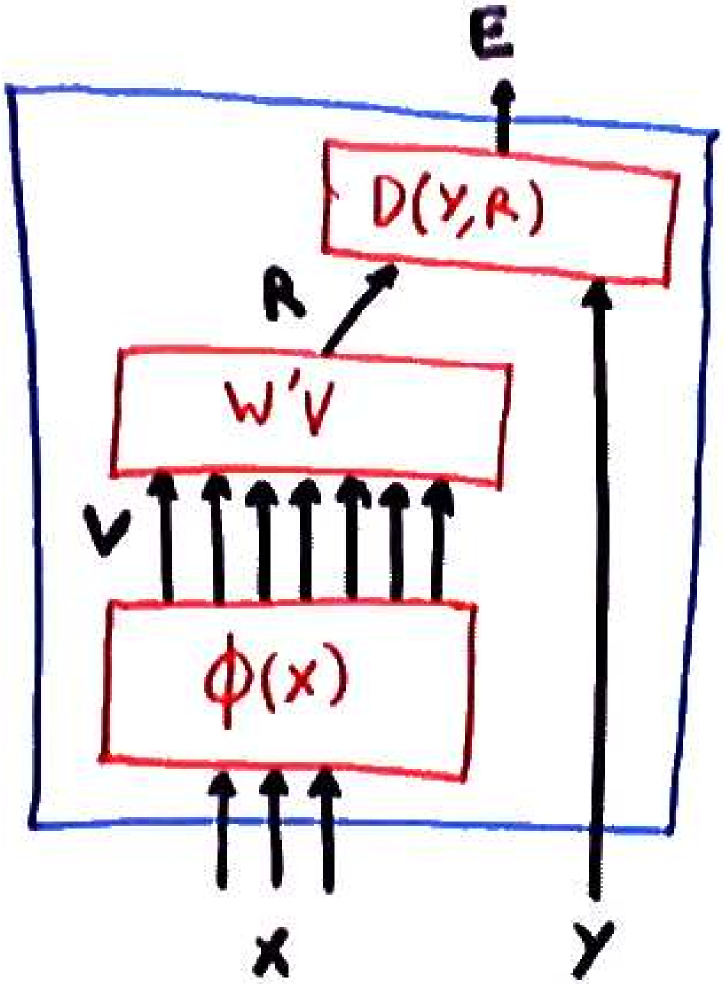
Example of Non-Linearly Separable Dichotomies





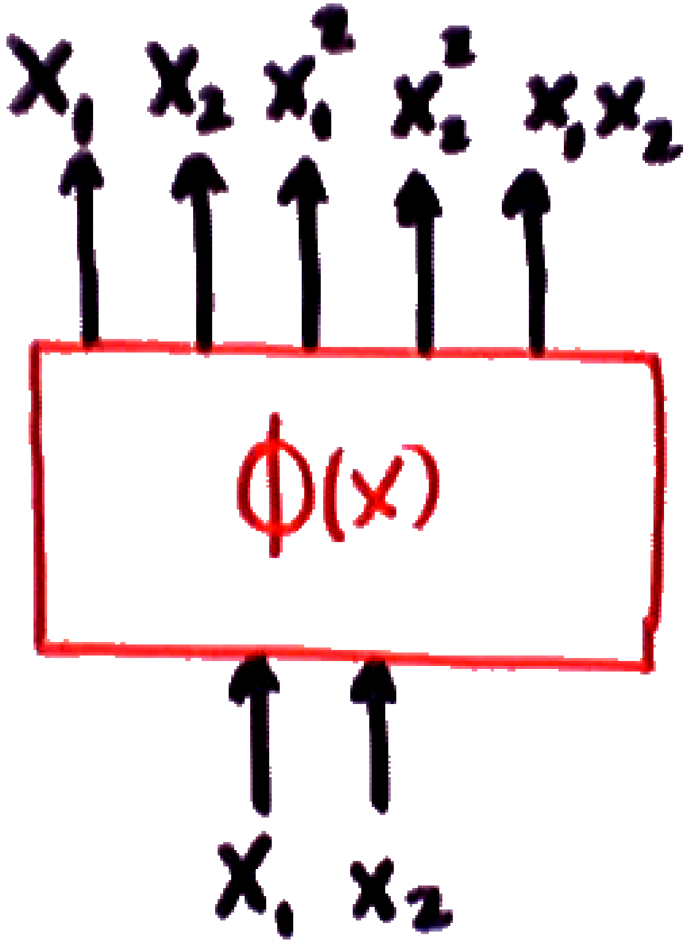
- Some seemingly simple dichotomies are not linearly separable
- Question: How do we make a given problem linearly separable?

Making N Larger: Preprocessing



- Answer 1: we make N larger by augmenting the input variables with new "features".
- we map/project X from its original N-dimensional space into a higher dimensional space where things are more likely to be linearly separable, using a vector function $\Phi(X)$.
- $E(Y, X, W) = D(Y, R)$
- $R = f(W'V)$
- $V = \Phi(X)$

Adding Cross-Product Terms



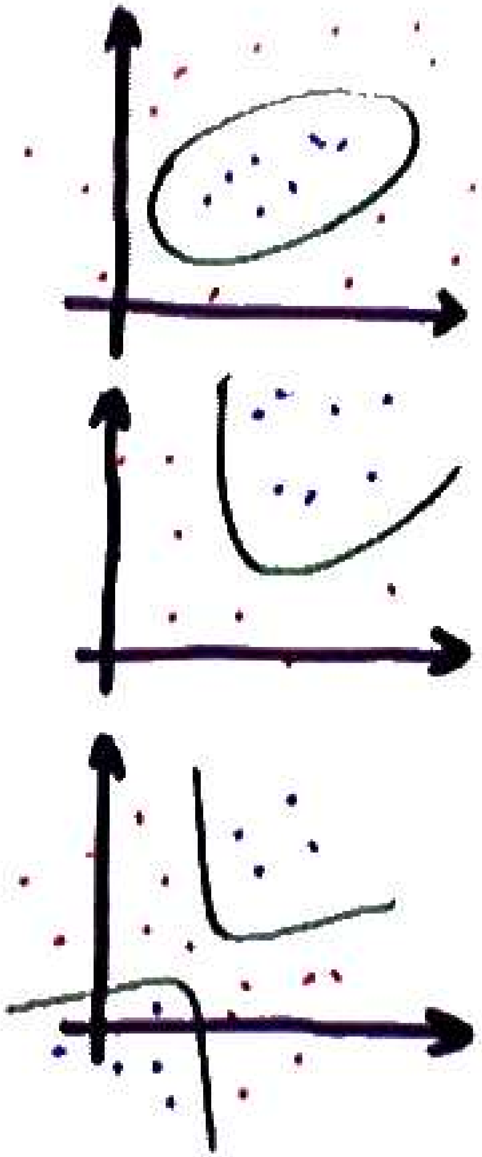
- Polynomial Expansion.
- If our original input variables are $(1, x_1, x_2)$, we construct a new *feature vector* with the following components:

$$\Phi(1, x_1, x_2) = (1, x_1, x_2, x_1^2, x_2^2, x_1x_2)$$

  i.e. we add all the cross-products of the original variables.
- we map/project X from its original N-dimensional space into a higher dimensional space with $N(N+1)/2$ dimensions.

Polynomial Mapping

o



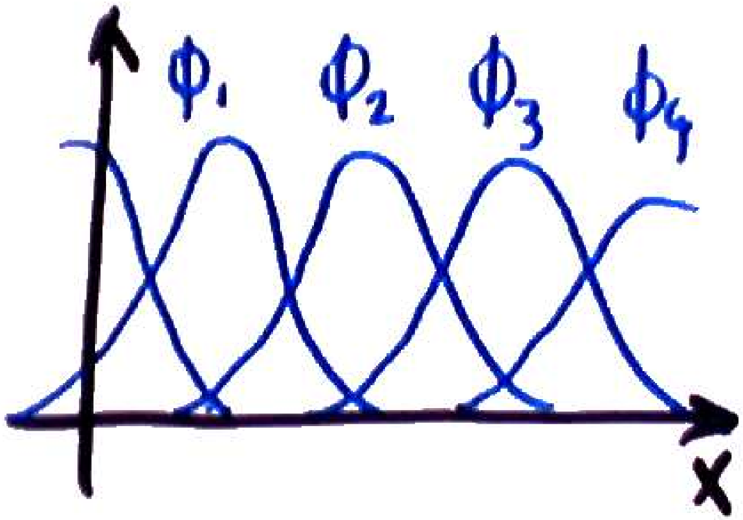
- Many new functions are now separable with the new architecture.
- With cross-product features, the family of class boundaries in the original space is the conic sections (ellipse, parabola, hyperbola).
- to each possible boundary in the original space corresponds a linear boundary in the transformed space.
- Because this is essentially a linear classifier with a preprocessing, we can use standard linear learning algorithms (perceptron, linear regression, logistic regression...).

Problems with Polynomial Mapping

- We can generalize this idea to higher degree polynomials, adding cross-product terms with 3, 4 or more variables.
- Unfortunately, the number of terms is the number of combinations d choose N, which grows like N^d, where d is the degree, and N the number of original variables.
- In particular, the number of free parameters that must be learned is also of order N^d.
- This is impractical for large N and for $d > 2$.
- Example: handwritten digit recognition (16x16 pixel images). Number of variables: 256. Degree 2: 32,896 variables. Degree 3: 2,796,160. Degre 4: 247,460,160.....

Next Idea: Tile the Space

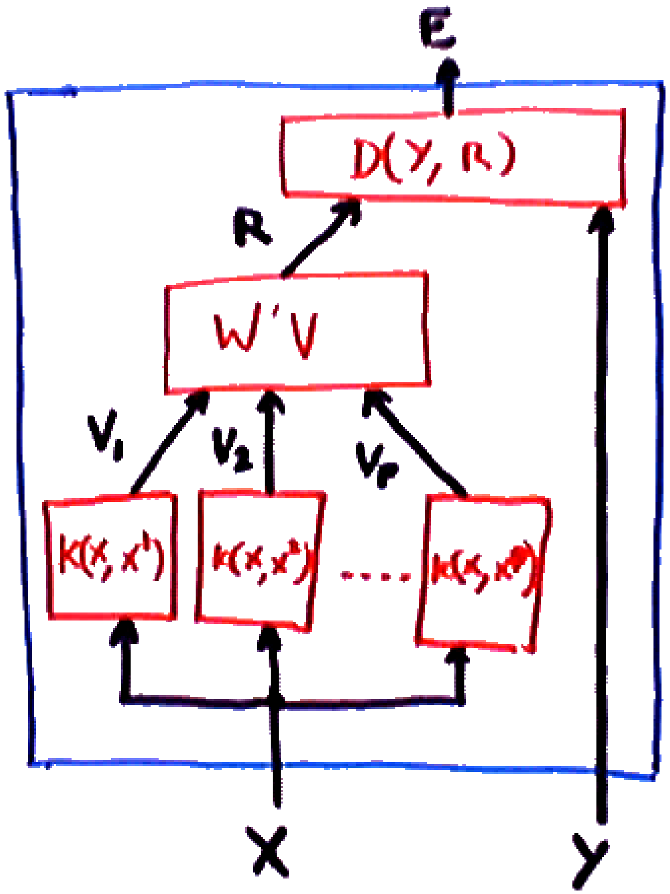
place a number of equally-spaced "bumps" that cover the entire input space.



- For classification, the bumps can be Gaussians
- For regression, the basis functions can be wavelets, sine/cosine, splines (pieces of polynomials)....
- problem: this does not work with more than a few dimensions.
- The number of bumps necessary to cover an N dimensional space grows exponentially with N.

Sample-Centered Basis Functions (Kernels)

Place the center of a basis function around each training sample. That way, we only spend resources on regions of the space where we actually have training samples.

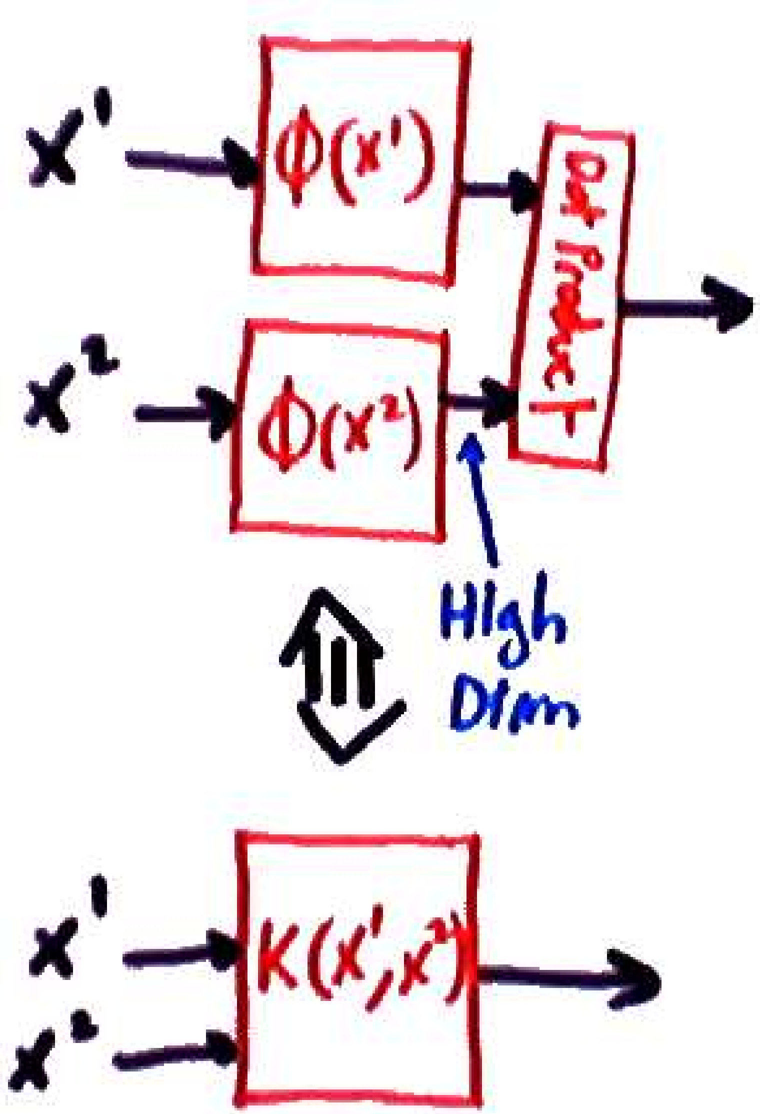


- Discriminant function:

$$f(X, W) = \sum_{k=1}^{k=P} W_k K(X, X^k)$$

- $K(X, X')$ often takes the form of a *radial basis function*: $K(X, X') = \exp(b||X - X'||^2)$ or a polynomial $K(X, X') = (X.X' + 1)^m$
- This is a very common architecture, which can be used with a number of energy functions.
- In particular, this is the architecture of the so-called Support Vector Machine (SVM), but the energy function of the SVM is a bit special. We will study it later in the course.

The Kernel Trick



- If the kernel function $K(X, X')$ verifies the *Mercer conditions*, then there exist a mapping Φ, such that $\Phi(X).\Phi(X') = K(X, X')$.
- The Mercer conditions are that K must be symmetric, and must be positive definite (i.e $K(X, X)$ must be positive for all X).
- In other words, if we want to map our X into a high-dimensional space (so as to make them linearly separable), and all we have to do in that space is compute dot products, we can take a shortcut and simply compute $K(X^1, X^2)$ without going through the high-dimensional space.
- This is called the "kernel trick". It is used in many so-called Kernel-based methods, including Support Vector Machines.

Examples of Kernels

- Quadratic kernel: $\Phi(X) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1 x2, x_1^2, x_2^2)$ then

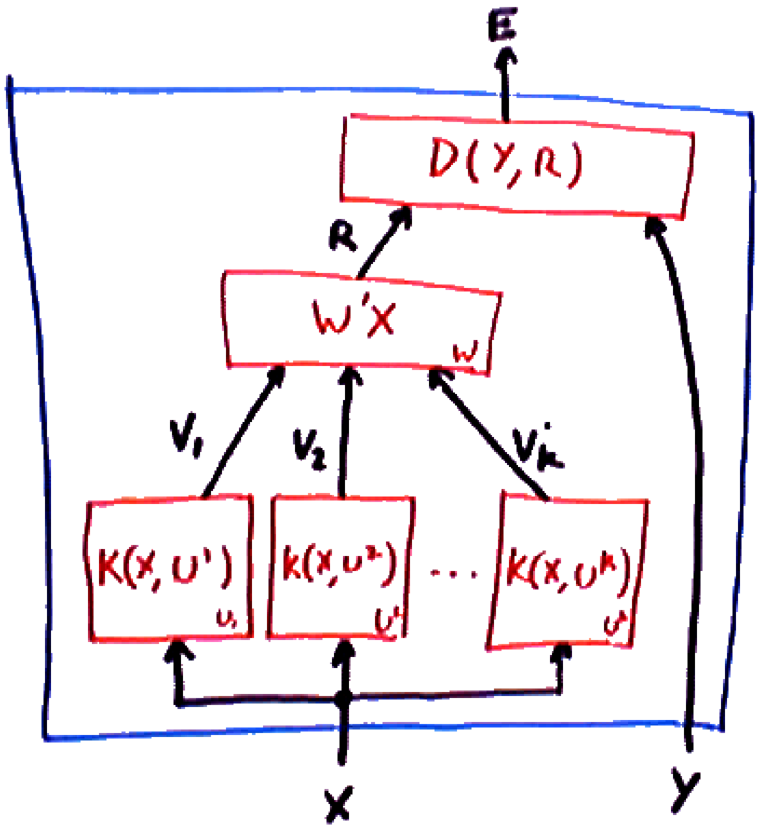
$$K(X, X') = \Phi(X).\Phi(X') = (X.X' + 1)^2$$

- Polynomial kernel: this generalizes to any degree d. The kernel that corresponds to $\Phi(X)$ bieng a polynomial of degree d is $K(X, X') = \Phi(X).\Phi(X') = (X.X' + 1)^d$.
- Gaussian Kernel:

$$K(X, X') = \exp(-b||X - X'||^2)$$

  This kernel, sometimes called the Gaussian Radial Basis Function, is very commonly used.

Sparse Basis Functions

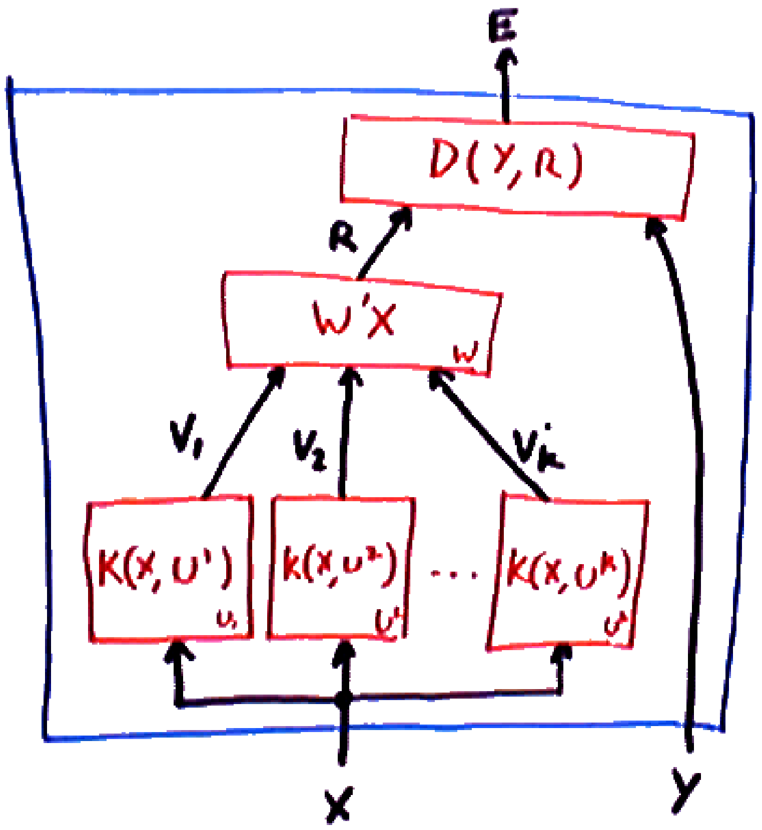


- Place the center of a basis function around areas containing training samples.
- Idea 1: use an unsupervised clustering algorithm (such as K-means or mixture of Gaussians) to place the centers of the basis functions in areas of high sample density.
- Idea 2: adjust the basis function centers through gradient descent in the objective function.

The discriminant function F is:

$$F(X, W, U^1, \ldots, U^K) = \sum_{k=1}^{k=K} W_k K(X, U^k)$$

Supervised Adjustment of the RBF Centers



- To adjust the U's we must compute the partial derivatives of L with respect to the U's.
- by posing and $V_k = K(X, U^k)$, and $R = \sum_{k=1}^{k=K} W_k V_k$ we can write:

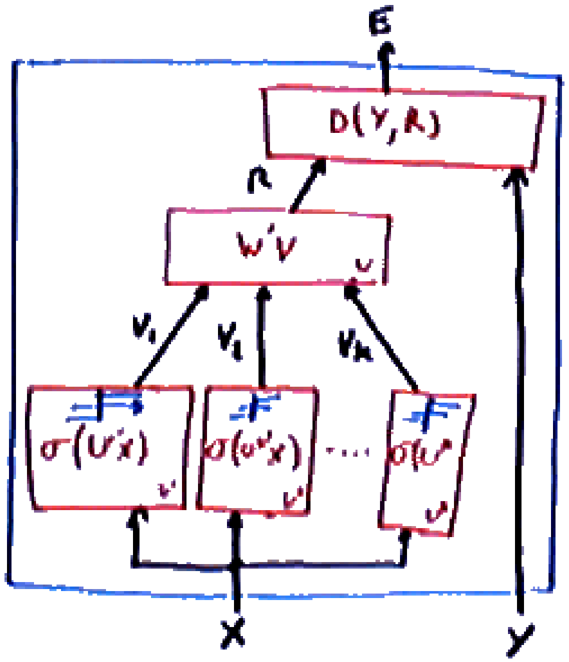
$$\frac{\partial L(W)}{\partial U^j} = \frac{\partial L(W)}{\partial R} \frac{\partial R}{\partial V_j} \frac{\partial V_j}{\partial U_j}$$

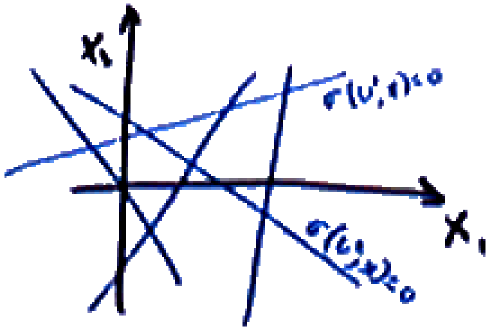
- Which comes down to:

$$\frac{\partial L(W)}{\partial U^j} = \frac{\partial L(W)}{\partial R} W_j \frac{\partial K(X, U_j)}{\partial U_j}$$

Now, there is a *very general method* for dealing with those multiple applications of chain rule. We will see that next time.

Other Idea: Random Directions

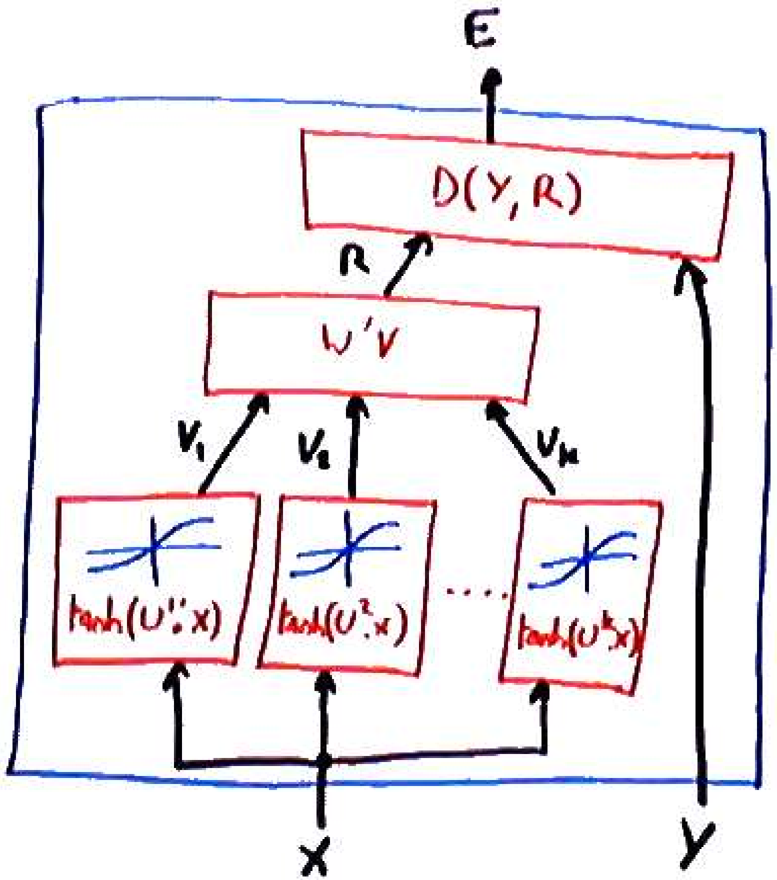




- Partition the space in lots of little domains by randomly placing lits of hyperplanes.
- Use many variables of the type $q(W^k X)$, where q is the threshold function (or some other squashing function) and W_k is a randomly picked vector.
- This is the original Perceptron.
- Without the non-linearity, the whole system would be linear (product of linear operations), and therefore would be no more powerful than a linear classifier.
- problem: a bit of a wishful thinking, but it works occasionally.

Neural Net with a Single Hidden Layer

A particularly interesting type of basis function is the sigmoid unit: $V_k = \tanh(U'^k X)$



- a network using these basis functions, whose output is $R = \sum_{k=1}^{k=K} W_k V_k$ is called a *single hidden-layer neural network*.
- Similarly to the RBF network, we can compute the gradient of the obejctive function with respect to the U^k:

$$\frac{\partial L(W)}{\partial U^j} = \frac{\partial L(W)}{\partial R} W_j \frac{\partial tanh(U_j' X)}{\partial U_j}$$

$$= \frac{\partial L(W)}{\partial R} W_j tanh'(U_j' X) X'$$

Any well-behaved function can be approximated as close as we wish by such networks (but K might be very large).