Outline

Outline

Math notations

Math notations

"Computer Science is no more a science about computers than astronomy is about telescopes." -- Edsger Dijkstra

Set

- In mathematics, a *set* is a collection of distinct objects, which are called *elements*. An element A belonging to a set B is denoted as $A \in B$, read as "A in B." No two elements can be the same in a set.
- We use curly brackets to enclose all members of a set, e.g., $\{4, 2, 1, 3\}$.
- A set can be finite, infinite or even empty, i.e., $\emptyset$ [1].
- Special set notations:
  - $\mathbb{Z}$ for all integers, $\mathbb{Z}^+$ for all positive integers.
  - $\mathbb{R}$ for all real numbers. And $\mathbb{Z}^+$ for?
  - $[X..Y]$ all integers between X and Y.
  - *closed interval*: $[X, Y]$ means all real numbers a such that $X \leq a \leq Y$.
  - *open interval*: $[X, Y]$ means all real numbers a such that $X \leq a \leq Y$.
- We use the notation $\{x_i\}_{i=N}^{M}$ or $\{x_i\}_{i\in[N..M]}$ as a shorthand for the set $\{x_N, x_{N+1}, \ldots, x_M\}$.
- *Set-builder notation*: e.g., $A = \{2 \cdot x | x \in \mathbb{Z}, x^2 > 7\} = \{6, 8, 9, \ldots\}$

Set (cont.)

- Operations on set:
  - Union: $A \cup B = \{x | x \in A\text{~or~}x \in B\}$, e.g., $\{1, 2, 3\} \cup \{4, 5, 6\} = \{1, 2, 3, 4, 5, 6\}$

- Intersection: $A \cap B = \{x | x \in A \text{~and~} x \in B\}$, e.g., $\{1,2,3\} \cup \{5,6,3\} = \{3\}$
- Difference [2]: $A \color{red}{\backslash\text{slash}} B = \{x | x \in A \text{~and~} x \notin B\}$, e.g., $\{1,2,3\} \color{red}{\backslash\text{slash}} \{5,6,3\} = \{1,2,5,6\}$
- subset and superset: $A \subseteq B$ iff $\forall x \in A$, $x \in B$ holds. We say that A is a *subset* of B and B a *superset* of A.

  iff is read as "if and only if" while $\forall$ is read as "for all."
- True subset or superset: $A \subset B$ iff $A \subseteq B$ and $B \color{red}{\backslash\text{slash}} A = \emptyset$. We say that A is a *true subset* of B and B a *true superset* of A.
- Homework: Prove that $A \subseteq B$ iff $A \cup B = B$.

* Venn Diagram
* Cartesian Product: $A \times B = \{(a,b) | a \in A \text{~and~} b \in B\}$.

Sequence, Tuple and Vector

* A *sequence* is an ordered collection of objects in which repetitions are allowed. Note that in set, there is no order nor repetition. A sequence is also called an *ordered list*.
* An n-*tuple* is a sequence of n elements, where n is a non-negative integer. In other words, a tuple is a finite sequence. It is also used interchangeably with the term *vector* in the context of this class.
* We usually use sharp bracket or square bracket. E.g., $< 1, 23, 3 >$ $[x_i]_{i=N}^{M}$, or $[x_i]_{i \in [N..M]}$.
* For the sake of space, we usually use bold font to denote a vector and usually the vector name is related to name for each of its elements, e.g., $\mathbf{X} = [x_1, \dots, x_N]$.
* The number of element in the vector is called its *dimension*, denoted as $\dim(\mathbf{X})$ or $|\mathbf{X}|$.
* It is also common to use a rightarrow over to denote a vector, e.g., $\overrightarrow{X}$.

Functions

* A function is a mapping from a non-empty set (called *domain*, could be a Cartesian product) of numbers to a non-emptyset (called *range*) of numbers. Remember, everything is a number in the computer.

- The map is one-to-one or many-to-one. But cannot be one-to-many.
- A function is denoted in the following form usually: $f: A \times B \mapsto C$ where f is the function name, A and B are the arguments or parameters, and C the output or return. The symbol $\mapsto$ is read as "maps to."
- If the input is one variable, this function is called *univariate*. If it has more than one, *multivariate*, including *bivariate*.
- For example, the sine function is $\sin: \mathbb{R} \mapsto [0, 1]$.
- Function composition: $(g \circ f)(x) = g(f(x))$.

Functions (cont.)

- Inverse function: $f^{-1}: Y \mapsto X$ if $f: X \mapsto Y$ and, both f and f^{-1} are one-to-one mappings.
- The ratio of change rate between the output of a multivariate function f and one input x is called the *derivative*. In discrete domains, it is denoted as $\frac{\partial f}{\partial x}\bigg|_{f=f_n, x=x_n} \equiv \frac{f_n - f_{n-1}}{x_n - x_{n-1}}$ where n is the index for both inputs and outputs. The big vertical bar is read as "evaluated at". Note that there is no *analytical* expression of derivative in discrete domains.
- For the sake of space, people could apply a function on a vector, e.g., $f(\mathbf{X}, \mathbf{Y}) = \mathbf{Z}$ where $\mathbf{X} = [x_1, \dots, x_N]$, $\mathbf{Y} = [y_1, \dots, y_N]$, $\mathbf{Z} = [z_1, \dots, z_N]$, and $\forall i \in [1..N]$, we have $f(x_i, y_i) = z_i$.

Linear Algebra

- Dimension: The dimension of a matrix is $N \times M$ if it has N rows (horizontal) and M columns (vertical).
- Matrix multiplication (not to be confused with *element-wise multiplication*). Why is matrix multiplication defined in this way?
- Dot product of vectors: $\sum_{i=1}^{N} x_i \cdot y_i = \mathbf{X} \cdot \mathbf{Y}$, short as $\mathbf{XY}$.
- Euclidean norm (L^2 norm): $\|\boldsymbol{X}\| := \sqrt{x_1^2 + \cdots + x_n^2}$ where $\mathbf{X} = [x_1, \dots, x_n]$.
- Transpose: $\mathbf{X} = \begin{bmatrix} x_1 \\ x2 \end{bmatrix}$, $\mathbf{X}^T = [x_1, x_2]$.
- Trace of a matrix: $\mathbf{Tr}(A) = [a_{1,1}, \dots, a_{N,N}]$ for a square matrix A of dimension $N \times N$.

Linear Algebra II

- Linear systems as matrixes. For example
  $$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \times \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$
  is equivalent to
  $$x_1 = a_{1,2}y_1 + a_{1,2}y_2$$
  $$x_2 = a_{2,1}y_1 + a_{2,2}y_2$$
  which can also be written as $\mathbf{X} = \mathbf{AY}$
  where
  $A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}$.
- Identity matrix (or einheit matrix): $I_n = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$

Linear Algebra III

- Inverse of a matrix:
  $\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}_n$.
  A non-inversible matrix is called a singular matrix.
- Determinant of a matrix: A matrix A is inversible iff $\mathbf{det}(A) \neq 0$ (and many other equivalence).
- Eigenvalue and Eigenvector: A eigenvalue λ for a square matrix $\mathbf{A}$ is a scalar such that
  $\mathbf{Av} = \lambda\mathbf{v}$ where λ is
  another vector, called the *eigenvector*. A matrix can have many eigenvalues, each of which is paired with one eigenvector. A inversible matrix has them.

Introduction to Machine Learning

Mathematical formulation of machine learning

- The task of statistical ML is to build a numerical predictive *model/estimator*, which is a function
  $f : \boldsymbol{X} \mapsto \boldsymbol{y}$.
- Three kinds of machine learning:
- Function approximation/fitting:
  - 1. Supervised learning: fit the function f given pairs of

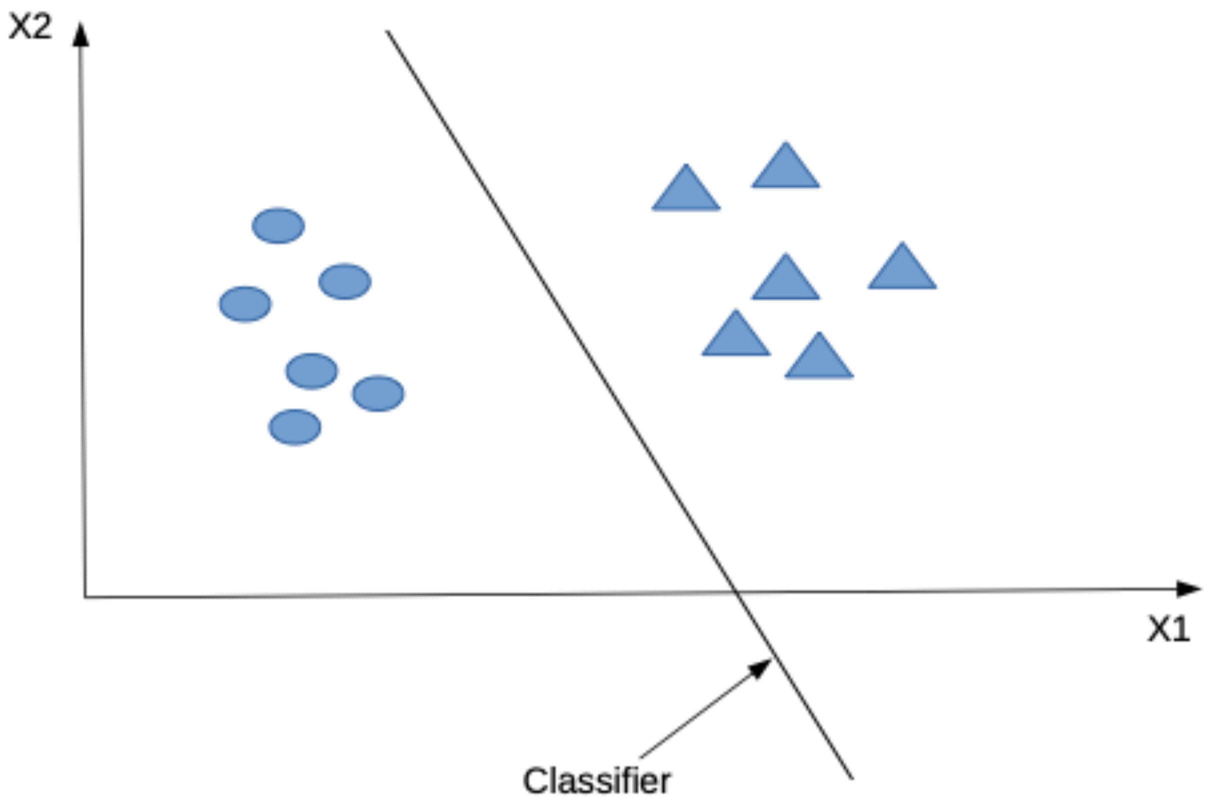
$\boldsymbol{X}$, called a training input (or feature vector if not raw) and $\boldsymbol{y}$, called the target/label.
    - 2. Reinforcement learning: fit the function f given pairs of $\boldsymbol{X}$, and $\boldsymbol{y}$, which is now called a value/cost function, defined by the interaction between the agent and the environment.
- 3. Unsupervised learning: learn to find the function f given only $\boldsymbol{X}$. No ground truth. Not function fitting.

- Deep learning: When the function f is highly complicated, that people usually use a deep neural network to represent. So there can be Deep X learning.

Supervised Learning

- In *supervised learning*, a pair (2-tuple) of an input/feature vector and a target form a (training) *sample*. A finite set of samples $\{(\mathbf{X_1}, y_1), \dots, (\mathbf{X_N}, y_N)\}$ form a *training set*, where each $\mathbf{X}_i \in \mathbb{R}^n$ ($i \in [1..N]$) is a feature vector while each y_i is a target.
- If the set $\boldsymbol{y}$ is discrete, e.g., $\boldsymbol{y} = \{+1, -1\}$, we call f a *classifier*. Otherwise, a *regressor*, e.g., $f : \mathbb{R}^n \mapsto \mathbb{R}$.
- Without losing generality, in this class a target is a real *scalar* while a feature vector is an n-dimensional real vector, i.e., $\boldsymbol{X} \subseteq \mathbb{R}^n$ and $\boldsymbol{y} \subseteq \mathbb{R}$.
- Given a training set, an ML algorithm will find such a function f, usually through solving a numerical optimization problem, to minimize the cost function, e.g., RMSE.
- Given a new label-less sample $\mathbf{X}_{new}$, the prediction is $f(\mathbf{X}_{new})$.
- The f is also called a *hypothesis*. And we can have many such hypotheses, forming the *hypothesis space*.

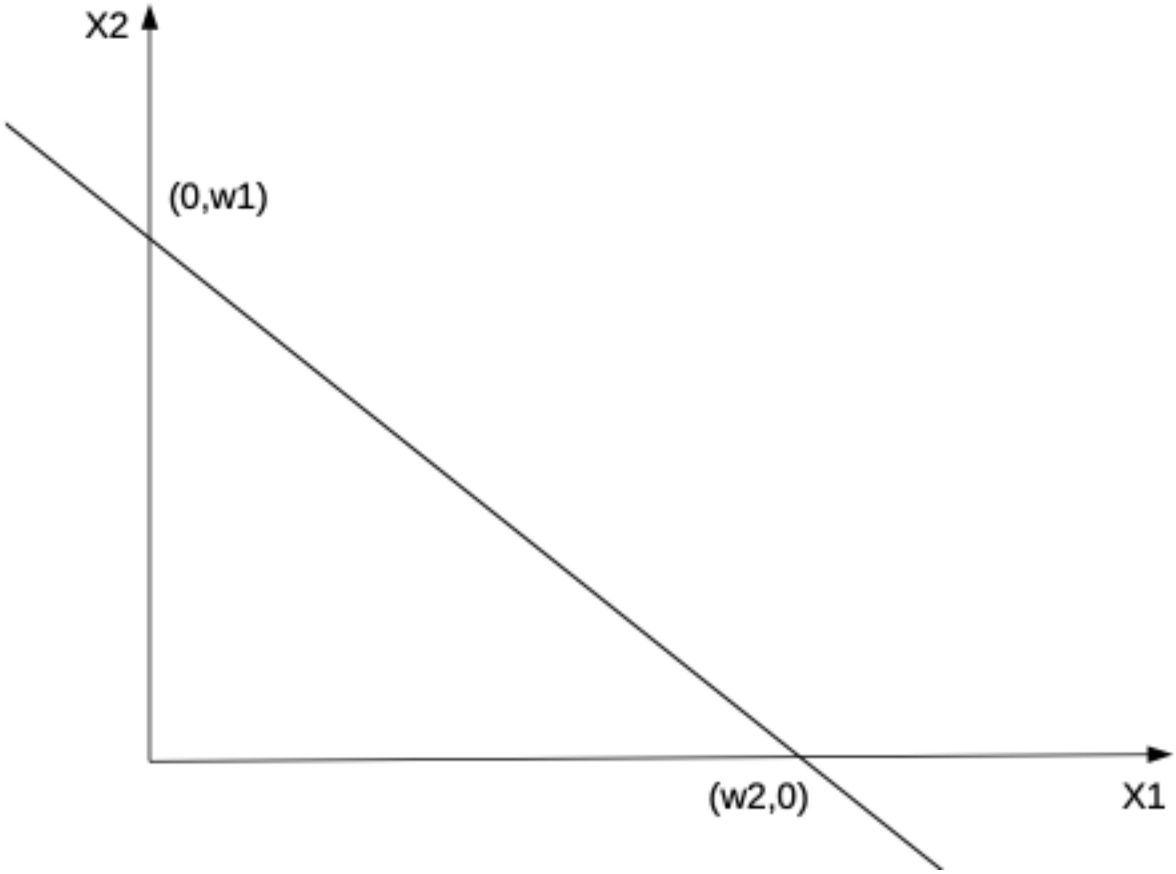
Features



- A sample has a feature vector to numerically represent it.
- For example, if you want to build a classifier to distinguish apples and banana, what features do you plan to use?
- Roundness, color, etc.
- Features can be non-human-readable, e.g., Google's Word2Vec.
- Deep learning is an approach to find features, or in the buzz word, the *abstract* of data.

Linear Classifiers

The hyperplane



- Now, let's begin our journey on supervised learning.
- Suppose we have a line going thru points $(0, w_1)$ and $(w_2, 0)$ (which are the *intercepts*) in a 2-D vector space spanned by two orthogonal bases x_1 and x_2.
- The equation of this line is $x_1 w_1 + x_2 w_2 - w_1 w_2 = 0$.

The hyperplane (cond.)

- Let $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$ and $\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ -w_1 w_2 \end{pmatrix}$. (By default, all vectors are column vectors.)\
  Then the equation is rewritten into vector form:
  $\mathbf{x}^T \cdot \mathbf{w} = 0.$\
  For space sake,
  $\mathbf{x}^T \mathbf{w} = \mathbf{x}^T \cdot \mathbf{w}.$

- Expand to n-dimension. $\mathbf{X} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ 1 \end{pmatrix}$ and

  $\mathbf{W} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ -w_1 w_2 \end{pmatrix}$.

  Then $\mathbf{X}^T \cdot \mathbf{W} = 0$, denoted as the *hyperplane* in $\mathbb{R}^n$.
- In our class, the $[x_1, x_2, \ldots, x_n]$ is a feature vector.
- The last term of $\mathbf{W}$ is often called a *bias* or a *threshold*.

Binary Linear Classifier

- A binary linear classier is a function $f(X) = \mathbf{W}\mathbf{X}$, such that $\begin{cases} \mathbf{W}^T\mathbf{X} > 0 & \forall X \in C_1 \\ \mathbf{W}^T\mathbf{X} < 0 & \forall X \in C_2 \end{cases}$ where C_1 and C_2 are the two classes. Note that the $\mathbf{X}$ has been augmented with 1 as mentioned before.
- Finding such an $\mathbf{W}$ is the *learning*.
- Using the function f to make decision is called *test*. Given a new sample whose augmented feature vector is $\mathbf{X}$, if $\mathbf{W}^T\mathbf{X} > 0$, then we classify the sample to class C_1. Otherwise, class C_2.
- Example. Let $\mathbf{W}^T = (2, 4, -8)$, what's the class for new sample $\mathbf{X} = (1, 1, 1)$ (1 is augmented)?
- $\mathbf{W}^T\mathbf{X} = -2 < 0$. Hence the sample of feature value $(1, 1)$ belongs to class C_1.

Solving inequalities: the simplest way to find the $\mathbf{W}$

- Let's look at a case where the feature vector is 1-D.
- Let the training set be $\{(4, C_1), (5, C_1), (1, C_2), (2, C_2)\}$. Their augmented feature vectors are: $X_1 = (4, 1)^T$, $X_2 = (5, 1)^T$, $X_3 = (1, 1)^T$, $X_4 = (2, 1)^T$.
- Let $\mathbf{W}^T = (w_1, w_2)$. In the training process, we can establish 4 inequalities: $\begin{cases} 4w_1 + w_2 & > 0 \\ 5w_1 + w_2 & > 0 \\ w_1 + w_2 & < 0 \\ 2w_1 + w_2 & < 0 \end{cases}$
- We can find many w_1 and w_2 to satisfy the inequalities. But,

how to pick the best?
- But let's talk about one more algorithm before defining the cost function.

Normalized feature vector

- I am lazy. I hate to write two cases.
- A correctly classified sample $(\mathbf{X_i}, y_i)$ shall satisfy the inequality $\mathbf{W}_i^T\mathbf{X}y_i > 0$. (The y_i flips the direction of the inequality.)
- *normalize* the feature vector: $\mathbf{X}_i y_i$ for $y_i \in \{+1, -1\}$.
- Example: Four samples, where\
  $\mathbf{x}'_1 = (0,0)^T$, $\mathbf{x}'_2 = (0,1)^T$,
  $\mathbf{x}'_3 = (1,0)^T$, $\mathbf{x}'_4 = (1,1)^T$
  $y_1 = 1, y_2 = 1, y_3 = -1, y_4 = -1$

  First, let's augment and normalize them:
  $\mathbf{x}_1 = (0,0,1)^T$, $\mathbf{x}_2 = (0,1,1)^T$,
  $\mathbf{x}_3 = (-1,0,-1)^T$, $\mathbf{x}_4 = (-1,-1,-1)^T$
- Please note that the term "normalized" could have different meanings in different context of ML.

least-squared and Fisher's criteria

Gradient

- The partial derivative of a multivariate function is a vector called the gradient, representing the derivatives of a function on different directions.
- For example, let $f(\mathbf{x}) = x_1^2 + 4x_1 + 2x_1x_2 + 2x_2^2 + 2x_2 + 14$. f maps a vector $\mathbf{x} = (x_1, x_2)^T$ to a scalar.
- Then we have $\nabla f = \frac{\partial f}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 2x_1 + 2x_2 - 4 \\ 4x_2 + 2x_1 + 2 \end{pmatrix}$
- The gradient is a special case of *Jacobian matrix*. (see also: *Hessian matrix* for second-order partial derivatives.)
- A *critical point* or a *stationary point* is reached where the derivative is zero on any direction.
  - local extremum
    - local maximum
    - local minimum

- saddle point
- if a function is convex, a local minimum/maxinum is the *global minimum/maximum*.

Find the linear classifier using an optimization way I

- Two steps here:
  1. Define a cost function to be minimized (The learning is the about minimizing the cost function)
  2. Choose an algorithm to minimize (e.g., gradient, least squared error etc.)
- One intuitive criterion can be the sum of error square: $J(\mathbf{W}) = \sum_{i=1}^{N}(\mathbf{W}^T\mathbf{x}_i - y_i)^2 = \sum_{i=1}^{N}(\mathbf{x}_i^T\mathbf{W} - y_i)^2$

Find the linear classifier using an optimization way II

- Minimizing $J(\mathbf{W})$ means (Convexity next time.) $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = 2\sum_{i=1}^{N}\mathbf{x}_i(\mathbf{x}_i^T\mathbf{W} - y_i) = (0, \ldots, 0)^T$
- Hence, $\sum_{i=1}^{N}\mathbf{x}_i\mathbf{x}_i^T\mathbf{W} = \sum_{i=1}^{N}\mathbf{x}_i y_i$
- The sum of a column vector multiplied with a row vector produces a matrix. $\sum_{i=1}^{N}\mathbf{x}_i\mathbf{x}_i^T = \begin{pmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_N \\ | & | & & | \end{pmatrix}\begin{pmatrix} — & \mathbf{x}_1^T & — \\ — & \mathbf{x}_2^T & — \\ & \vdots & \\ — & \mathbf{x}_N^T & — \end{pmatrix} = \mathbb{X}^T\mathbb{X}$

Find the linear classifier using an optimization way II

- $\sum_{i=1}^{N}\mathbf{x}_i y_i = \begin{pmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_N \\ | & | & & | \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} = \mathbb{X}^T\mathbf{y}$
- $\mathbb{X}^T\mathbb{X}\mathbf{W} = \mathbb{X}^T\mathbf{y}$
- $(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbb{X}\mathbf{W} = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbf{y}$
- $\mathbf{W} = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbf{y}$

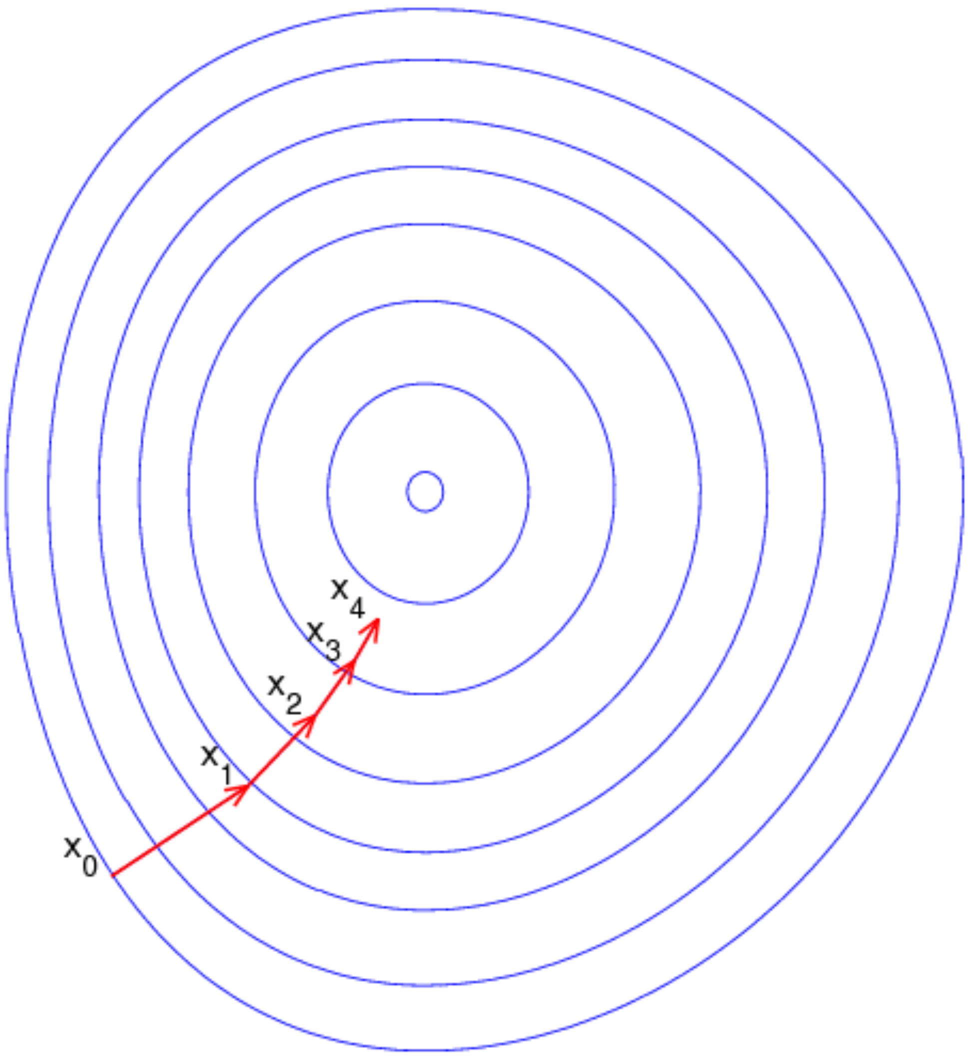
Gradient descent approach Since we define the target function as $J(\mathbf{W})$, finding $J(\mathbf{W}) = 0$ or minimizing $J(\mathbf{W})$ is intuitively the same as reducing $J(\mathbf{W})$ along the gradient. The algorithm below is a general approach to minimize any multivariate function: changing the input variable proportionally to the gradient.

.5

Input: an initial $\mathbf{w}$, stop criterion θ, a learning rate function $\rho(\cdot)$, iteration step $k=0$

$\mathbf{w}_{k+1} := \mathbf{w}_k - \rho(k)\nabla J(\mathbf{w})$
$k := k+1$

.36



In many cases, the $\rho(k)$'s amplitude (why amplitude but not the value?) decreases as k increases, e.g., $\rho(k) = \frac{1}{k}$, in order to shrink the adjustment. Also in some cases, the stop condition is $\rho(k)\nabla J(\mathbf{w}) > \theta$. The limit on k can also be included in stop condition -- do not run forever.

Fisher's linear discriminant

- What really is $\mathbf{w}^T x$? $\mathbf{w}$ is perpendicular to the hyper panel [3]
- $\mathbf{w}^T\mathbf{x}$ is the *projection* of the point $\mathbf{x}$ on the decision panel.

- Intuition in a simple example: for any two points
  $\mathbf{x}_1 \in C_1$ and $\mathbf{x}_2 \in C_2$, we want
  $\mathbf{w}^T\mathbf{x}_1$ to be as different from
  $\mathbf{w}^T\mathbf{x}_1$ as possible, i.e.,
  $\max(\mathbf{w}^T\mathbf{x}_1 - \mathbf{w}^T\mathbf{x}_2)^2$.
  [Fig. 4.6, Bishop book]
- For binary classification, intuitively, we want the projections of the same class to be close to each other (i.e., the smaller $\tilde{s}_1$ and $\tilde{s}_2$ the better) while the projects of different classes to be apart from each other (i.e., the larger $(\tilde{m}_1 - \tilde{m}_2)^2$ is better).
- That means $\max J(\mathbf{w}) = \frac{(\tilde{m}_1-\tilde{m}_2)^2}{\tilde{s}_1^2+\tilde{s}_2^2}$ where
  $\tilde{m}_i = \frac{1}{|C_i|} \sum\limits_{\mathbf{x}\in C_i} \mathbf{w}^T\mathbf{x}$
  and
  $\tilde{\mathbf{s}}_i^2 = \sum\limits_{\mathbf{x}\in C_i} (\mathbf{w}^T\mathbf{x} - \tilde{m}_i)^2$
  are the mean and the variance of the projection of all samples belonging to Class i on the decision panel, respectively.

Fisher's (cond.)

- between-class scatter:
  $(\tilde{m}_1 - \tilde{m}_2)^2 = (\mathbf{w}^T(\mathbf{m_1} - \mathbf{m_2}))^2 = \mathbf{w}^T(\mathbf{m_1} - \mathbf{m_2})(\mathbf{m_1} - \mathbf{m_2})^T\mathbf{w}$
  where
  $\mathbf{m}_i = \frac{1}{|C_i|} \sum\limits_{\mathbf{x}\in C_i} \mathbf{x}$
- within-class scatter:
  $\tilde{\mathbf{s}}_i^2 = \sum\limits_{\mathbf{x}\in C_i} (\mathbf{w}^T\mathbf{x} - \tilde{m}_i)^2 = \sum\limits_{\mathbf{x}\in C_i} (\mathbf{w}^T\mathbf{x} - \mathbf{w}^T\mathbf{m}_i)^2 = \mathbf{w}^T[\sum\limits_{\mathbf{x}\in C_i} (\mathbf{x} - \mathbf{m}_i)(\mathbf{x} - \mathbf{m}_i^T)]\mathbf{w}$
- Denote
  $\mathbf{S_w} = \tilde{\mathbf{s}}_1^2 + \tilde{\mathbf{s}}_2^2$ and
  $\mathbf{S}_B = (\mathbf{m_1} - \mathbf{m_2})(\mathbf{m_1} - \mathbf{m_2})^T$.
  We have
  $J(\mathbf{w}) = \frac{\mathbf{w}^T\mathbf{S}_B\mathbf{w}}{\mathbf{w}^T\mathbf{S}_W\mathbf{w}}$.
  This expression is known as *Rayleigh quotient*.
- To maximize $J(\mathbf{w})$, the $\mathbf{w}$ must satisfy
  $\mathbf{S}_B\mathbf{w} = \lambda\mathbf{S}_w\mathbf{w}$.
- Hence
  $\mathbf{w} = \mathbf{S}_w^{-1}(\mathbf{m}_1 - \mathbf{m}_2)$.
  (Derivate saved.)

Perceptron algorithm

The perceptron algorithm

- Recall earlier that a sample $(\mathbf{X}_i, y_i)$ is correctly classified if $\mathbf{W}^T\mathbf{X}_i y_i > 0$.
- Let's define a new cost function to be minimized:
  $J(\mathbf{W}) = \sum\limits_{x_i \in \mathcal{M}} -\mathbf{W}^T\mathbf{X}_i$
  where $\mathcal{M}$ is the set of all samples misclassified ($\mathbf{W}^T\mathbf{X}_i y_i < 0$).
- Then,
  $\nabla J(\mathbf{W}) = \sum\limits_{\mathbf{x}_i \in \mathcal{M}} -\mathbf{X}_i$
  (because $\mathbf{W}$ is the coefficients.)
- Batch perceptron algorithm: In each batch, computer $\nabla J(\mathbf{W})$ for all samples misclassified using the same old $\mathbf{W}$ and then update.

Single-sample perceptron algorithm

- Another common type of perceptron algorithm is called single-sample perceptron algorithm.
- Update $\mathbf{W}$ whenever a sample is misclassified.
  1. Initially, $\mathbf{W}$ has arbitrary values. $k = 1$.
  2. In the k-th iteration, we pick sample $\mathbf{X}_j$ such that $j = k \mod n$ to update the $\mathbf{W}$ by:
     $\mathbf{W}_{k+1} = \begin{cases} \mathbf{W}_k - \rho\mathbf{X}_j & , \text{if } \mathbf{W}_j^T\mathbf{X}_\mathbf{j} y_j \le 0, \text{~(wrong prediction)} \\ \mathbf{W}_k & , \text{if } \mathbf{W}_j^T\mathbf{X}_\mathbf{j} y_j > 0 \text{~(correct classification)} \end{cases}$ where ρ is a constant called *learning rate*.
  3. The algorithm terminates when all samples are classified correctly.
- Note that $\mathbf{X}_k$ is not necessarily the k-th training sample due to the loop.
- Note the difference between $\mathbf{W}_1$ and w_1.

The perceptron algorithm (cond.) Begin our iteration. Let $\mathbf{w}_1 = (0, 0, 0)^T$ and $\rho = 1$.

1. $\mathbf{W}_1^T \cdot \mathbf{x}_1 = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0 \le 0$. Need to update $\mathbf{W}$:
   $\mathbf{W}_2 = \mathbf{W}_1 + \rho \cdot \mathbf{x}_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$
2. $\mathbf{W}_2^T \cdot \mathbf{x}_2 = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} = 1 > 0$. No updated need. But since it does not

classify all samples correctly, we need to keep going. Just let $\mathbf{w}_3 = \mathbf{w}_2$.

3. $\mathbf{W}_3^T \cdot \mathbf{x}_3 = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} -1 \\ 1 \\ -1 \end{pmatrix} = -1 \leq 0$. Need to update $\mathbf{W}$:

$$\mathbf{W}_4 = \mathbf{W}_3 + \rho \cdot \mathbf{x}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}$$

The perceptron algorithm (cond.)

- In the end, we have $\mathbf{W}_{14} = \begin{pmatrix} -2 \\ 0 \\ 1 \end{pmatrix}$, let's verify how well it works
- $\begin{cases} \mathbf{w}_{14} \cdot \mathbf{x}_1 & = 1 > 0 \\ \mathbf{w}_{14} \cdot \mathbf{x}_2 & = 1 > 0 \\ \mathbf{w}_{14} \cdot \mathbf{x}_3 & = 1 > 0 \\ \mathbf{w}_{14} \cdot \mathbf{x}_4 & = 1 > 0 \end{cases}$
- Mission accomplished!
- However, the perceptron algorithm will not *converge* unless the data is linearly separable.
- Solution: Map the data to a space that are linear separable.

SVM

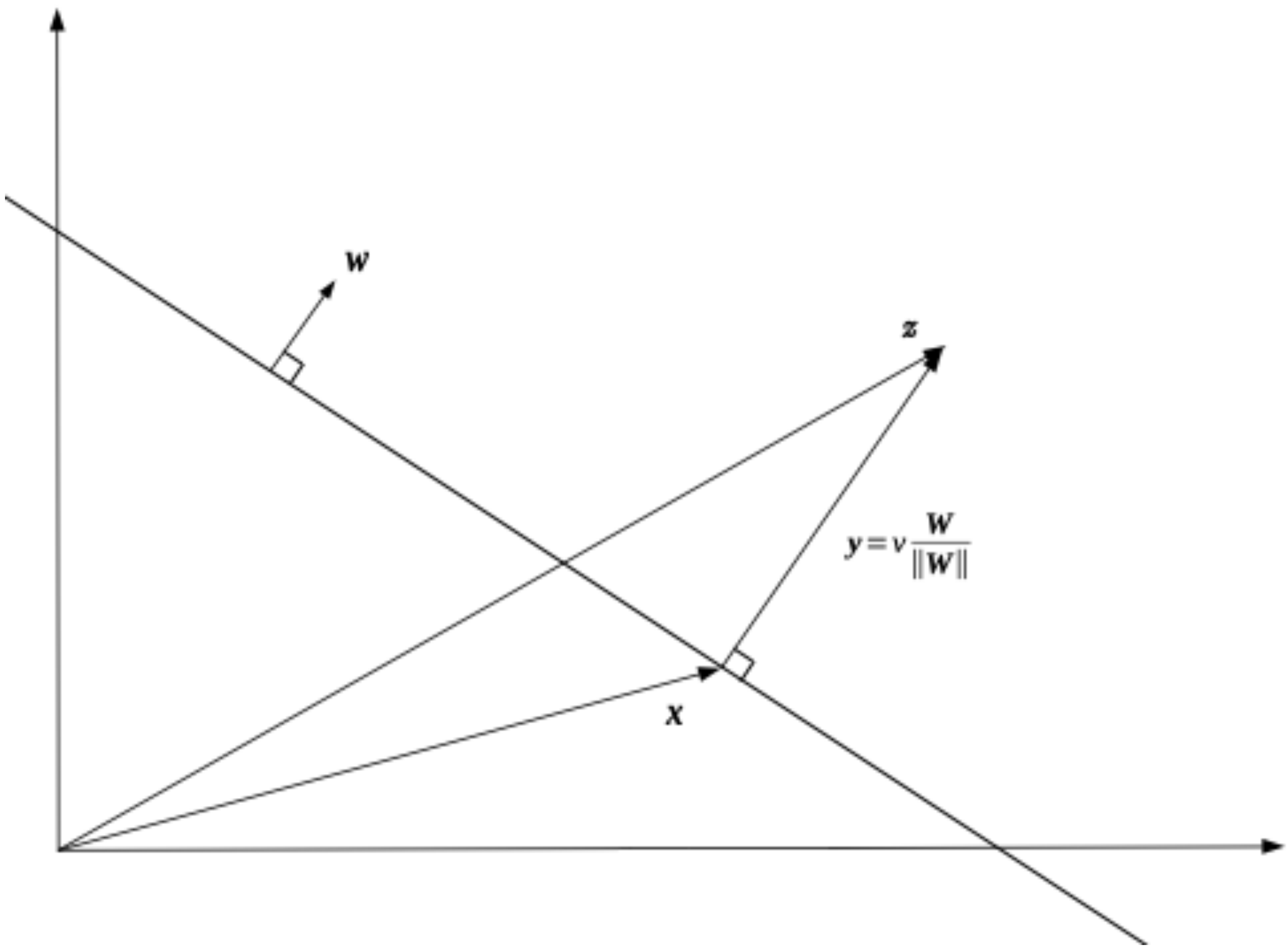
The distance from any point to the hyperplane

- Earlier our discussion used the augmented definition of linear binary classifier: the feature vector $\mathbf{x} = (x_1, \ldots, x_n, 1)^T$ and the weight vector $\mathbf{w} = (w_1, \ldots, w_n, w_b)^T$. The hyperplane is an equation $\mathbf{w}^T\mathbf{x} = 0$. If $\mathbf{w}^T\mathbf{x} > 0$, then the sample belongs to one class. If $\mathbf{w}^T\mathbf{x} < 0$, the other class.
- Let's go back to the un-augmented version. Let $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ and $\mathbf{w} = [w_1, w_2, \ldots, w_n]^T$. If $\mathbf{w}^T\mathbf{x} + w_b > 0$ then $\mathbf{x} \in C_1$. If $\mathbf{w}^T\mathbf{x} + w_b < 0$ then $\mathbf{x} \in C_2$. The equation $\mathbf{w}^T\mathbf{x} + w_b = 0$ is the hyperplane, where $\mathbf{w}$ only determines the direction of the hyperplane. To build a classifier is to search for the values for $w_1, \ldots, w_n$ and w_b, the bias/threshold.
- For convenience, we denote $g(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$.
- We have proved that $\mathbf{w}$, augmented or not, is perpendicular

to the hyperlane.

The distance from any point to the hyperplane

.3



.7

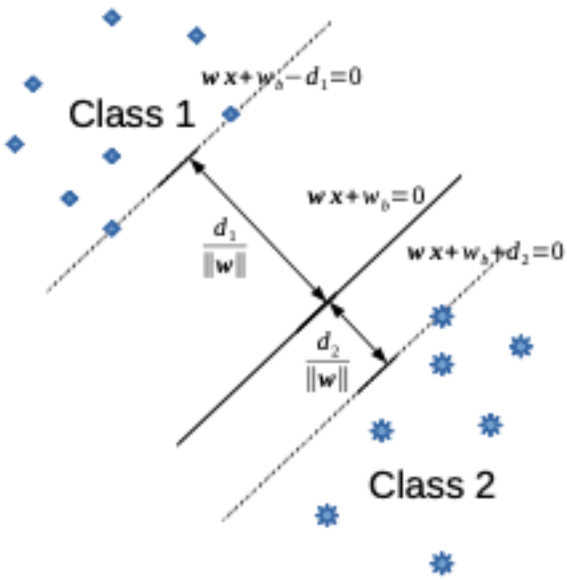
- Let the closest point on the hyperplane $\mathbf{w}^T\mathbf{x} = 0$ to $\mathbf{z}$ be $\mathbf{x}$. Define $\mathbf{z} = \mathbf{x} + \mathbf{y}$.
- Because both $\mathbf{y}$ and $\mathbf{w}$ are perpendicular to the hyperplane, we have $\mathbf{y} = v\frac{\mathbf{w}}{||\mathbf{w}||}$, where v is the Euclidean distance from $\mathbf{z}$ to $\mathbf{x}$ and $\frac{\mathbf{w}}{||\mathbf{w}||}$ is the unit vector pointing at the direction of $\mathbf{w}$.
- Therefore, $\mathbf{z} = \mathbf{x} + v\frac{\mathbf{w}}{||\mathbf{w}||}$.

$$\begin{aligned}\mathbf{w}^T\mathbf{z}+w_b &= \mathbf{w}(\mathbf{x}+\tfrac{\mathbf{w}}{||\mathbf{w}||})+w_b\end{aligned}$$

- Therefore we have $$\begin{aligned} &= \mathbf{wx}+\mathbf{wv}+w_b \\ &= v\frac{\mathbf{ww}}{||\mathbf{w}||} = v\frac{||\mathbf{w}||^2}{||\mathbf{w}||} = v||\mathbf{w}||.\end{aligned}$$
- Hence, $v = \frac{\mathbf{w}^T\mathbf{z}+w_b}{||\mathbf{w}||}$
- HW2: Prove that the distance from the origin to the hyperplane is $\frac{-w_b}{||\mathbf{w}||}$.

Hard margin linear SVM

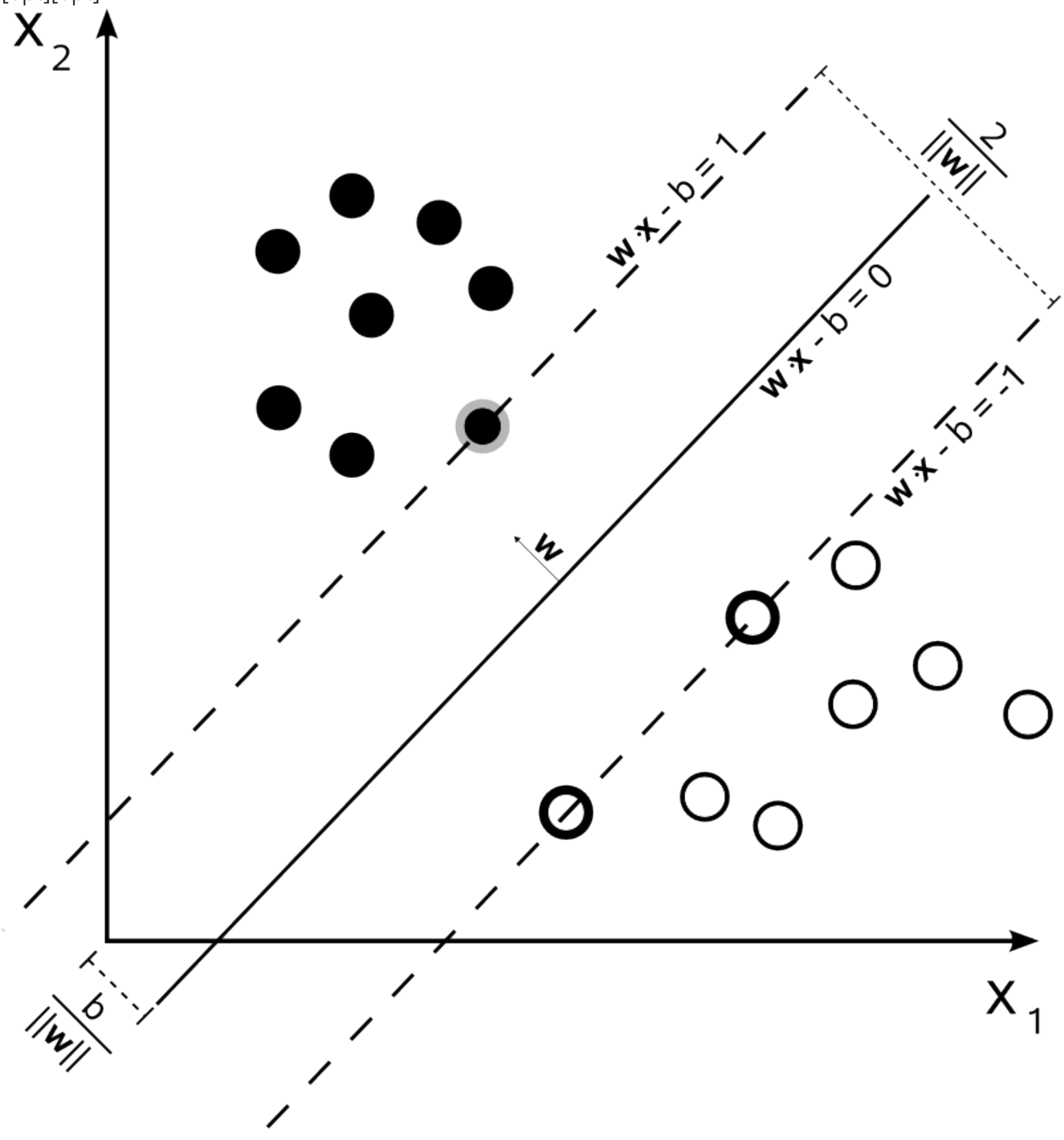
[0pt][0pt]



- Assume that the minimum distance from any point in Class C_1 and C_2 to the hyperplane are $d_1/||\mathbf{w}||$ and $d_2/||\mathbf{w}||$, respectively, where $d_1, d_2 > 0$.
- Then we have $\mathbf{w}^T\mathbf{x}+w_b-d_1 \geq 0, \forall x \in C_1$, and $\mathbf{w}^T\mathbf{x}+w_b+d_2 \geq 0, \forall x \in C_2$.
- In order to build a more discriminant classifier, we want to maximize the distance between the two classes to the decision plane, known as the *margin*, i.e. $\max\left(\frac{d_1}{||\mathbf{w}||}+\frac{d_2}{||\mathbf{w}||}\right)$.
- Hence, an SVM classifier is also called a *Maximum Margin Classifier*.
- Assuming the two classes are linearly separable, put things together: $$\begin{cases}\max & \frac{d_1}{||\mathbf{w}||}+\frac{d_2}{||\mathbf{w}||} \\ s.t. & \mathbf{w}^T\mathbf{x}+w_b-d_1 \geq 0, \forall x \in C_1 \\ & \mathbf{w}^T\mathbf{x}+w_b+d_2 \geq 0, \forall x \in C_2\end{cases}$$

Hard margin linear SVM (cond.)

[0pt][0pt]



- We prefer $d_1 = d_2$, to prefer no class than the other.
- Since d_1 and d_2 are constants, we can let them be 1. Also, denote $y_k \in \{+1, -1\}$ as the label for the k-th training sample $\mathbf{x}_k$, we can get a different form:

$$r^T \mathbf{x}_k + w_b) \geq 1, \forall \mathbf{x}_k \in C_1 \cup C_2.$$

- Maximizing $\frac{2}{||\mathbf{w}||}$ is equivalent to minimizing $\frac{||\mathbf{w}||}{2}$.
- Finally, we transform it into a quadratic programming problems: $\begin{cases} \min & \frac{1}{2}||\mathbf{w}||^2 = \frac{1}{2}\mathbf{w}^T\mathbf{w} \\ s.t. & y_k(\mathbf{w}^T\mathbf{x}_k + w_b) \geq 1, \forall \mathbf{x}_k. \end{cases}$

The Karush-Kuhn-Tucker conditions

- Given a nonlinear optimization problem $\begin{cases} \min & f(\mathbf{x}) \\ s.t. & h_k(\mathbf{x}) \geq 0, \forall k \in [1..K], \end{cases}$ where $\mathbf{x} = [x_1, \dots, x_n]$, and $h_k(\cdot)$ is linear, it's Lagrange multiplier (or Lagrangian) is: $L(\mathbf{x}, \lambda) = f(\mathbf{x}) - \sum_{k=1}^{K} \lambda_k h_k(\mathbf{x})$
- The necessary condition that the problem above has a solution is KKT condition: $\begin{cases} \frac{\partial L}{\partial \mathbf{x}} = \mathbf{0}, & \\ \lambda_k \geq 0, & \forall k \in [1..K] \\ \lambda_k h_k(\mathbf{x}) = 0, & \forall k \in [1..K] \end{cases}$

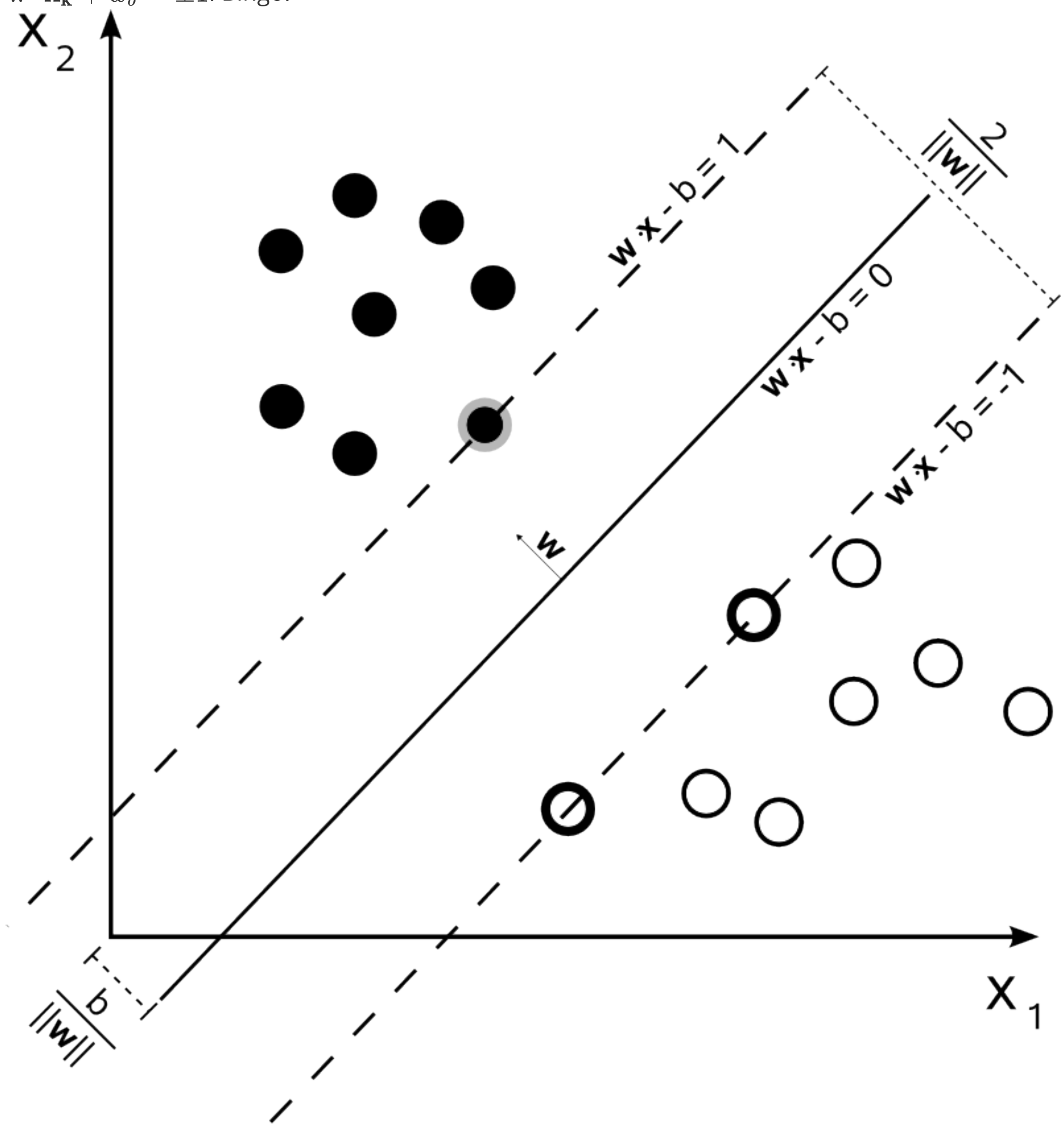
Properties of hard margin linear SVM

- The KKT condition of Eq. ([[eq:svm_problem]](#eq:svm_problem){reference-type="ref" reference="eq:svm_problem"}) is $\begin{cases} \frac{\partial L}{\partial w} = \mathbf{0}, & \\ \frac{\partial L}{\partial w_b} = 0, & \\ \lambda_k \geq 0, & \forall k \in [1..K] \\ \lambda_k[y_k(\mathbf{w}^T\mathbf{x_k} + w_b) - 1] = 0, & \forall k \in [1..K] \end{cases}$
- Let's solve it. $\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{k=1}^{K} \lambda_k y_k \mathbf{x_k} \Rightarrow \mathbf{w} = \sum_{k=1}^{K} \lambda_k y_k \mathbf{x_k}$ $\frac{\partial L}{\partial w_b} = \sum_{k=1}^{K} \lambda_k y_k = 0$
- λ_k is either positive or 0. Thus the solutions for Eqs. ([[eq:partial_on_weight_vector]](#eq:partial_on_weight_vector){reference-type="ref" reference="eq:partial_on_weight_vector"}) and ([[eq:partial_on_bias]](#eq:partial_on_bias){reference-type="ref" reference="eq:partial_on_bias"}) is only associated with samples that $\lambda_k \neq 0$. Let $N_s = \{\mathbf{x}_k | \lambda_k \neq 0, k \in [1..K]\}$.

Properties of hard margin linear SVM (cont.)

- Therefore, Eq. ([[eq:partial_on_weight_vector]](#eq:partial_on_weight_vector){reference-type="ref" reference="eq:partial_on_weight_vector"}) can be rewritten into $\mathbf{w} = \sum_{\mathbf{x}_k \in N_s} \lambda_k y_k \mathbf{x_k}$ The samples $\mathbf{x}_k \in N_s$ collectively determine the $\mathbf{w}$, and thus called *support vectors*, supporting the solution.

- The support vectors also have an interesting "visual" properties. Solving the last two equations in Eq. ([[eq:svm_kkt]](#eq:svm_kkt){reference-type="ref" reference="eq:svm_kkt"}) $\forall \mathbf{x}_k \in N_s$: $\lambda_k \neq 0$ and $\lambda_k [y_k(\mathbf{w}^T \mathbf{x_k} + w_b) - 1] = 0$, we have $y_k(\mathbf{w}^T \mathbf{x_k} + w_b) = 1$.
- Given that $y_k \in \{+1, -1\}$, we have $\mathbf{w}^T \mathbf{x_k} + w_b = \pm 1$. Bingo!

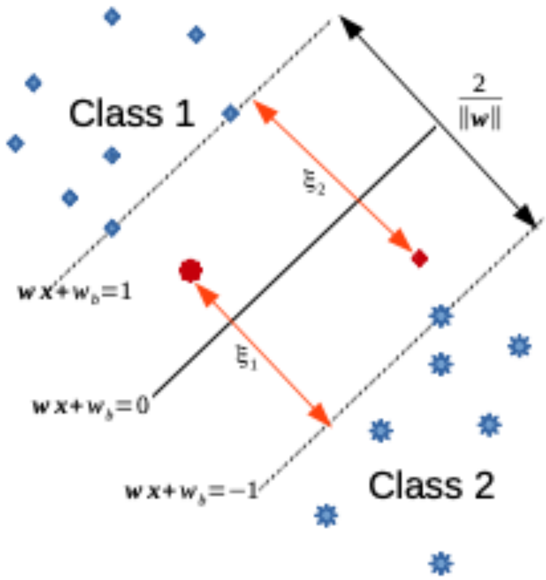


Solving hard margin linear SVM

- Remember that KKT condition is a necessary condition, not sufficient condition.

- Eq. ([[eq:svm_problem]](#eq:svm_problem){reference-type="ref" reference="eq:svm_problem"}) is a quadratic programming problem. There are many documents on the Internet about solving hard margin linear SVM as a quadratic programming problem. Here is one in MATLAB http://www.robots.ox.ac.uk/~az/lectures/ml/matlab2.pdf. For Python, use the `cvxopt` toolbox. I have some hints at here: http://forrestbao.blogspot.com/2015/05/guide-to-cvxopts-quadprog-for-row-major.html If you still cannot figure out, read my recent paper at PLoS Computational Biology (http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004838) and its source code (https://bitbucket.org/forrestbao/mflux).

Soft margin linear SVM

.4 

.6

- What if the samples are not linearly separable?
- Let $\xi_k = 0$ for all samples on or inside the correct margin boundary.
- Let $\xi_k = |y_k - (\mathbf{w}^T\mathbf{x}_k + w_b)|$, i.e., the prediction error, for all samples that are misclassified (red in the left figure), where the operator $|\cdot|$ stands for absolute value.
- In this case, we want to maximize the margin but minimize the number of misclassified samples.

```
<!-- -->
```

- Therefore, we have a new optimization problem: $\begin{cases} \min & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{k=1}^{K}\xi_k \\ s.t. & y_k(\mathbf{w}^T\mathbf{x}_k + w_b) \geq 1 - \xi_k, \forall \mathbf{x}_k \\ & \xi_k \geq 0. \end{cases}$

  where C is a constant.
- Such SVM is called *soft-margin*.

Soft margin linear SVM

- The constant C provides a balance between maximizing the margin and minimizing the quality, instead of quantity, of misclassification.
- Given a training set, how to find the optimal C? Grid search using cross-validation.

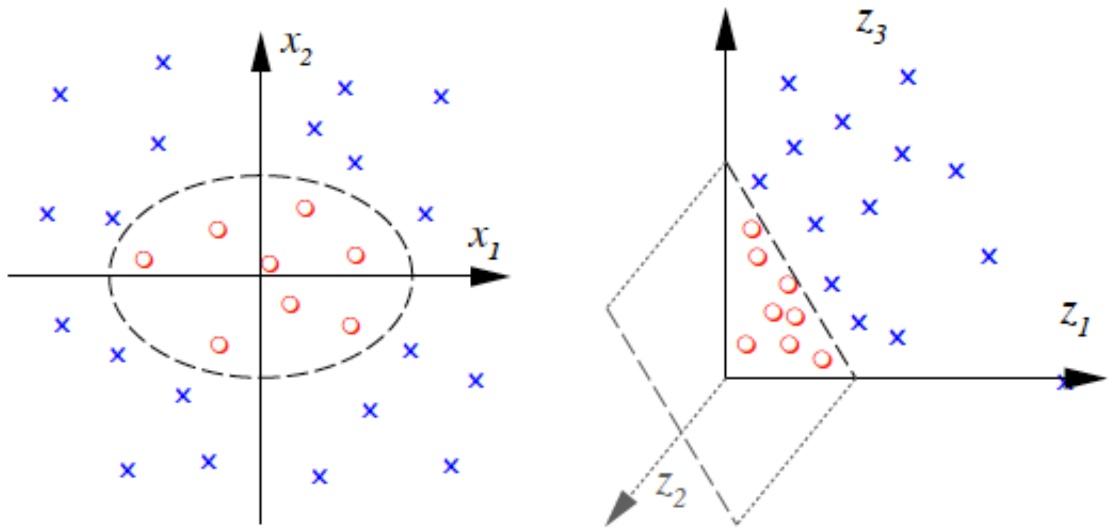
Generalized Linear Classifier

- What if a problem is not linearly separable? One wise solution is to convert it into a linearly separable one.
- Let $f_1(\cdot)$, $f_2(\cdot)$, ..., $f_P(\cdot)$ be P nonlinear functions where $f_p : \mathbb{R}^n \mapsto \mathbb{R}, \forall p \in [1..P]$.
- Then we can define a mapping from a feature vector $\mathbf{x} \in \mathbb{R}^n$ ($\mathbb{R}^n$ is called the *input space*) to a vector in another space $\mathbf{z} = [f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_P(\mathbf{x})]^T \in \mathbb{R}^P$, which is called the *featrue space*.
- The problem then becomes finding the value P and the functions $f_p(\cdot)$ such that the two classes are linearly separable.
- Once the space transform is done, we wanna find a weight vector $\mathbf{w} \in \mathbb{R}^P$ such that $\begin{cases} \mathbf{w}^T\mathbf{z} + w_b > 0 & \text{if } \mathbf{z} \in C_1 \\ \mathbf{w}^T\mathbf{z} + w_b < 0 & \text{if } \mathbf{z} \in C_2. \end{cases}$
- Essentially, we are building a new hyperplane $g(\mathbf{x}) = 0$ such that $g(\mathbf{x}) = w_b + \sum_{p=1}^{P} w_p f_p(\mathbf{x})$.
  Instead of computing the weighted sum of elements of feature vector, we compute that of elements of the transformed vector.

Generalized Linear Classifier (cont.)

- For example,
  $g(\mathbf{x}) = w_b + w_1x_1 + w_2x_2 + w_{12}x_1x_2 + w_{11}x_1^2 + w_{22}x_2^2$
- Here is another example,

$$\Phi : R^2 \rightarrow R^3$$
$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt(2) x_1 x_2, x_2^2)$$



- This approach is often called *kernel tricks*.

Artificial Neural Networks

Why artificial neural networks (ANNs) work

- Supervised or Reinforcement learning is about function fitting.
- We don't care about the analytical form of the function hidden in the pairs of input and output data.
- As long as we can mimic/fit it accurately enough, it's good.
- An ANN is a magical structure that can mimic any function [Fig. 5.3, Bishop book], if the ANN is "complex" enough.

One neuron/preceptron

- This computes a weighted sum: $\mathbf{w}^T\mathbf{x}$
- This is a linear classifier: $f(\mathbf{w}^T\mathbf{x})$ where $f(\cdot)$ can be, e.g., a step function. (If using a continuous function for $f(\cdot)$, we can get a linear regressor.)
- Why one of the algorithms we have seen is called "preceptron" algorithm?
- Because $f(\mathbf{w}^T\mathbf{x})$ is exactly one neuron/preceptron in an ANN.
- Frank Rosenblatt published his preceptron algorithm in 1962 titled "Principles of Neurodynamics: Perceptrons and the Theory of Brain

Mechanisms."
- Linearly separable cases only! It cannot even do XOR.
- Therefore, Marvin Minsky jumped to the conclusion that ANNs were useless. [Perceptrons, Marvin Minsky and Seymour Papert, MIT Press, 1969]
- However, Minsky is an AAAI fellow but not a prophet.

Adding a layer of neurons?

- United, they (neurons) stand.
- First, we built a 2nd layer of neurons resulted from transforming the input vector $\mathbf{x}$: $f(\mathbf{w}_1^T\mathbf{x})$, $f(\mathbf{w}_2^T\mathbf{x})$, $\dots$, $f(\mathbf{w}_n^T\mathbf{x})$.
- For simplicity sake, we call the function $f(\cdot)$ an activation function, which could be nonlinear, e.g., step.
- Then we add them together: $\phi(\mathbf{x}) = g\left(\sum_{i=1}^{N} f(\mathbf{w}_i^T\mathbf{x})\right)$. If $g(\cdot)$ is also nonlinear, can this new function $\phi(\mathbf{x})$ handle (at least some) cases that $f(\mathbf{w}^T\mathbf{x})$ cannot handle because not linearly separable?
- Let's generalize:
  1. Even the $f(\cdot)$ can vary: $f_1(\mathbf{w}_1^T\mathbf{x})$, $f_2(\mathbf{w}_2^T\mathbf{x})$, $\dots$, $f_n(\mathbf{w}_n^T\mathbf{x})$
  2. Weight when adding the 2nd layer of neurons together: $\phi(\mathbf{x}) = g\left(\sum_{i=1}^{n} u_i f_i(\mathbf{w}_i^T\mathbf{x})\right)$

Just applying nonlinear weighted sum again and again

- The output from different neurons in the 2nd layer is a vector: $\mathbf{o} = [o_1, o_2, \cdots, o_n] = \left[f_1(\mathbf{w}_1^T\mathbf{x}), f_2(\mathbf{w}_2^T\mathbf{x}), \cdots, f_n(\mathbf{w}_n^T\mathbf{x})\right]$
- Let $\mathbf{u} = [u_1, u_2, \cdots, u_n]$
- Rewrite: $\phi(\mathbf{x}) = g(\mathbf{u}^T\mathbf{o})$.
- Deja Vu?
- It's nonlinear weighted sum ($g(\cdot)$ is non-linear) from inputs, again!
- How does it look like? A directed graph.
- Using this principle, we can populate many layers (could of different neurons) between the input and output to mimic a highly complex function.

- The function can be multivariate at the output too: more than one neurons in the output layer.
- By training an ANN, we find the weights between any two consecutive layers.

Something terminology

- activation: the output of a neuron, which is applying an activation function onto the a weighted sum of its inputs, denoted as $f(\mathbf{w}^T\mathbf{x})$.
- Input/Hidden/Output layer
- Forward or forward propagation
- connection/synapse

Gradient descent on multilayer perceptrons

- New challenge: How to compute the gradient for neurons not directly connected to final output? Just find its "fair share" to cost function.
- As an example, let cost function be the difference between prediction ϕ and label y. $\nabla(J(\mathbf{w}_i)) = \underbrace{\frac{\partial(\phi-y)}{\partial \mathbf{w}_i} \textcolor{red}{\backslash\text{pause}} = \frac{\partial\phi}{\partial \mathbf{w}_i}}_{y \text{ has nothing to do with } \mathbf{w}_i} = \textcolor{red}{\backslash\text{pause}} \underbrace{\frac{\partial\phi}{\partial o_i}}_{\substack{\text{layers 2/hidden}\\ \text{to 3/output}}} \textcolor{red}{\backslash\text{pause}} \cdot \underbrace{\frac{\partial o_i}{\partial \mathbf{w}_i}}_{\substack{\text{layers 1/input}\\ \text{to 2/hidden}}}$
- -.5em Because of composition in each layer ($\phi = g(\mathbf{u}^T\mathbf{o})$ and each $\underset{i\in[1..p]}{o_i} = f(\mathbf{w}_i^T\mathbf{x})$), by expanding Eq. ([[eq:two_stage_derivative]](#eq:two_stage_derivative){reference-type="ref" reference="eq:two_stage_derivative"}), we have:

$$\frac{\partial\phi}{\partial \mathbf{w}_i} = \frac{\partial g(\mathbf{u}^T\mathbf{o})}{\partial(\mathbf{u}^T\mathbf{o})}\frac{\partial(\mathbf{u}^T\mathbf{o})}{\partial o_i} \textcolor{red}{\backslash\text{pause}} \frac{\partial f(\mathbf{w}_i^T\mathbf{x})}{\partial \mathbf{w}_i^T\mathbf{x}}\frac{\partial \mathbf{w}_i^T\mathbf{x}}{\partial \mathbf{w}_i} = \textcolor{red}{\backslash\text{pause}} \overbrace{\underbrace{g'(\mathbf{u}^T\mathbf{o})\cdot u_i}_{\substack{\text{error propagated}\\ \text{from 2nd layer}}} \textcolor{red}{\backslash\text{pause}} \cdot f'(\mathbf{w}_1^T\mathbf{x})}^{\text{all scalars}} \cdot \underbrace{\mathbf{x}}_{\substack{\text{perceptron}\\ \text{algorithm!}}}$$

Backpropagation

- Eq. ([[eq:two_stage_derivative]](#eq:two_stage_derivative){reference-type="ref" reference="eq:two_stage_derivative"}) tells us that in order to compute the gradient in the current layer, we must have the product of gradients from all forward (output-bound) layers in hand.
- Weights of connections are updated from the output to the input, against the direction of feedforward.
- It resembles that the cost function is propagated from the output layer to the input layer, layer by layer.

Gradient vanishing problem

- Will the gradient get larger or smaller as backpropagation moves on?
- The derivative of the activation function (e.g., sigmoid, hyporbalic tangent) usually yields of a value in $[-1, 1]$.
- When you multiple a number with another number in $[-1, 1]$, it becomes smaller.
- Hence, the gradient becomes smaller and smaller (vanishes) as we backpropagate toward the input layer.
- It takes really long to update weights near the input layer.
- Solution: LTSM, residual networks, etc.

Deep Learning

Deep Learning

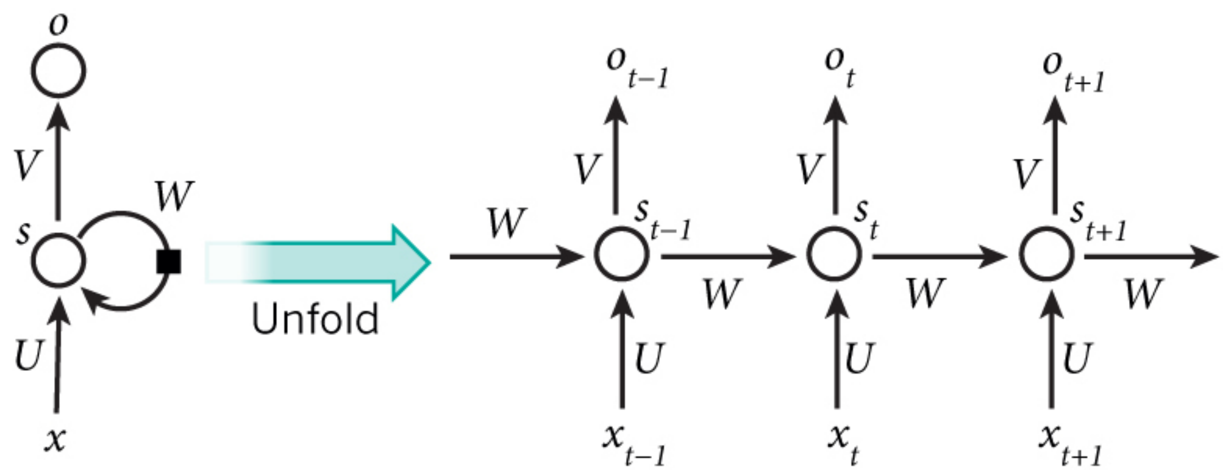
- Feature extraction:
  - Conventional ML: manually craft a set of features.
  - Problem: Sometimes features are too difficult to be manually designed, e.g., from I/O system log.
  - A (no-brainer) solution: let computers find it for us, even by brutal force.
- Not all weights matter:
  - There are more tasks that need function fitting beyond conventional classification and regression.
  - Ex. producing a sequence (e.g., a sentence)
  - Sometimes we use the network to get something else useful, such as word embedding.
  - Maybe weights of only a small set of layers are what we need from training.
- Equally important to network architecture, the training scheme also matters (not just simple pairs of feature/input vectors and labels).

CNN

- Convolutional layer: imagine convolution as matching two shapes/sequences/strings
- Pooling layer
- ReLU layer
- Fully connected layer (basically this is the regular ANN)
- Avoid overfitting: dropout, stochastic pooling, etc.
- implementation: text-cnn
- Visualization of the output at layers: http://cs231n.github.io/convolutional-networks/
- Do we use backpropagation to update weights in every layer?
- Some layers are unsupervised!

Vanilla RNN

- An RNN is just an IIR filter (are you also a EE major?): $y[n] = \sum_{l=1}^{N} a_l y[n-l] + \sum_{k=0}^{M} b_k x[n-k]$ where $x[i]$ (or $y[i]$) is the i-th element (a scalar) in the input (or output) sequence.
- RNN allows the output of a neuron to be used as the input of itself (typically) or others. Typically, $\mathbf{s}_{t+1} = U\mathbf{x}_t + W\mathbf{s}_t$ where $\mathbf{s}_{t+1}$ and $\mathbf{s}_t$ are the output of the neuron at steps $s+1$ and s respectively.
- Unrolling/unfolding an RNN unit:



Neural language model in Elman network

- Recall that a language model predicts the Probability of a sequence of (words, characters, etc.)
- Because of the properties of conditional probability, we want the probability of next word, given a short history of the sequence: $P(w_{t+1}|w_i, i \in [t-k..t])$
- Elman network/simple RNN. Three layers:
    - Input layer is the concatenation $\mathbf{x}(t)$ of two parts: the current **sequence** (not just one element!!!) $\mathbf{w}(t) = [w_{t-k}, \ldots, w_t]$, plus output from the hidden layer in previous step $\mathbf{s}(t-1)$.
    - hidden/context layer: $\mathbf{s}(t) = f(\mathbb{X}\mathbf{x}(t))$ where $\mathbb{X}$ is the matrix of weights from the input layer to hidden layer.
    - Output layer: multiple neurons, one of which of the highest activation corresponds to the best prediction. Each neuron

corresponds to one element in the sequence, e.g., word/character/etc.

- The new language model: $P(w_{t+1}|\mathbf{w}(t), \mathbf{s}(t-1))$, predicting the next output word given a short history $\mathbf{w}(t)$ up to current step t and the hidden layer up to previous step $t-1$.

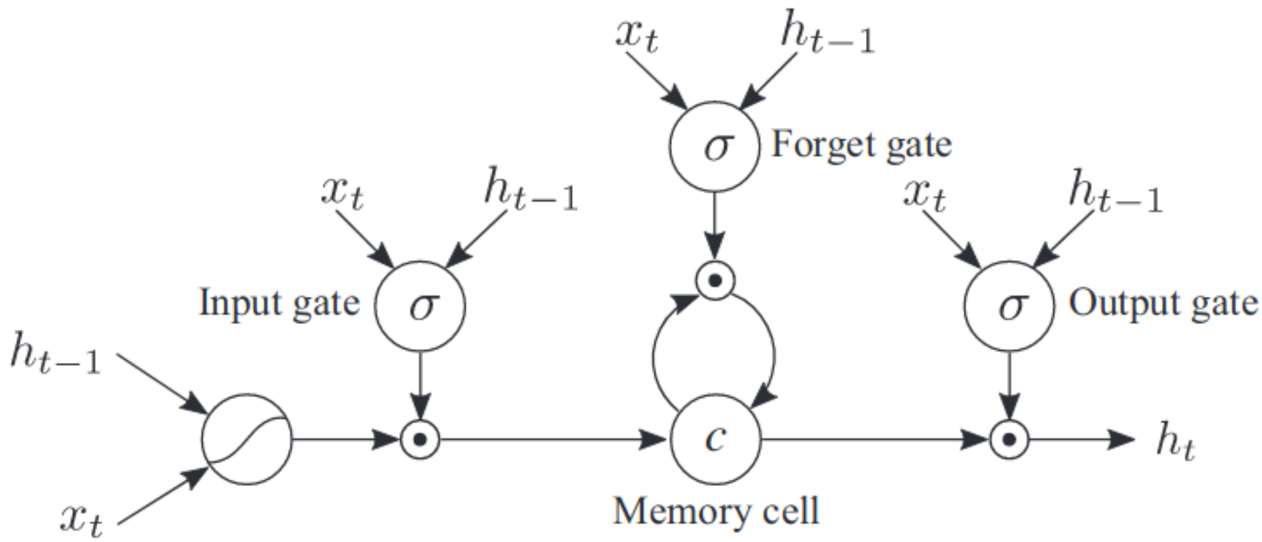
Neural language models

- Feedforward: "A neural Probabilistic Language Model", Beigio et al., JMLR, 3:1137--1155, 2003
- "Recurrent neural network based language model", Mikolov et al., Interspeech 2010
- Multiplicative RNN: "Generating Text with Recurrent Neural Networks", Sutskever, ICML 2011

LTSM and GRU

- Motivations:
  - An simply deep RNN can be unrolled into many many layers. Gradient vanishing is significant.
  - We also want weights to be attenuated/gated based on the states.
- LSTM and GRU
  - Instead of layers, we have cells.
  - LSTM: forget gate, input gate, and output gate. The 3 gates are computed from current input $\mathbf{x}(t)$ and output from previous cell $\mathbf{s}(t-1)$. Then we "make a choice" between using previous state $\mathbf{c}(t-1)$ and current input and use the choice and output gate to make the final output.
  - GRU: simpler, just reset gate and update gate.
  - http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano/
  - http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM



Source:
Listen, Attend, and Walk: Neural Mapping of Navigational Instructions to Action Sequences, Mei et al., AAAI-16

Seq-to-seq learning

- Let's go one more level up.
- Instead of predicting the next element in an input sequence, can we produce the entire output sequence from the input sequence?
- There could be no overlap between the two sequences, e.g., from a Chinese sentence to a German sentence.
- Two RNNs: encoder and decoder
- "Learning Phrase Representations using RNN Encoder--Decoder for Statistical Machine Translation", Cho et al., EMNLP 2014

```
decision panel,
$\mathbf{w}^T \mathbf{x}_1 = \mathbf{w}^T \mathbf{x}_2 = 0$.
Therefore $\mathbf{w}^T (\mathbf{x}_1 - \mathbf{x}_2) = 0$. Dot
product of 0 means that two vectors are orthogonal.
```

---

1. Outside the US, people use $\varnothing$ ↩
2. In other parts of the world: $A - B$ ↩
3. Given any two points $\mathbf{x_1}$ and $\mathbf{x_2}$ on the ↩