Introduction to Machine Learning

Perceptrons

Varun Chandola

February 21, 2017

Outline

Contents

Taking the next step

Hypothesis Space, $\mathcal{H}$

- Conjunctive

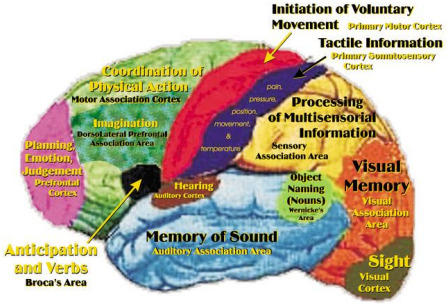


Figure 1: *Src: http://brainjackimage.blogspot.com/*

- Disjunctive
  - Disjunctions of k attributes
- Linear hyperplanes
- $\mathbf{c}_* \in \mathcal{H}$

Input Space, x

- $\mathbf{x} \in \{0,1\}^d$
- $\mathbf{x} \in \mathbb{R}^d$

Input Space, y

- $y \in \{0,1\}$
- $y \in \{-1,+1\}$

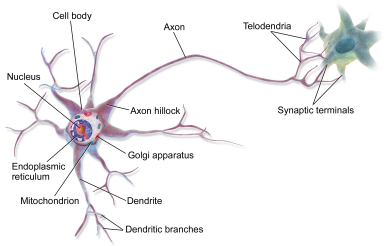


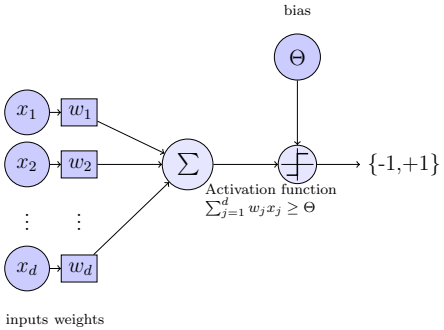
Figure 2: *Src: Wikipedia*

1 Perceptrons

- Human brain has 10^{11} neurons
- Each connected to 10^4 neighbors

So far we have focused on learning concepts defined over d binary attributes. Now we generalize this to the case when the attributes are numeric, i.e., they can take infinite values. Thus the concept space becomes infinite. How does one search for the target concept in this space and how is their performance analyzed. We begin with the simplest, but very effective, algorithm for learning in such a setting, called **perceptrons**. Perceptron algorithm was developed by Frank Rosenblatt [4] at the Calspan company in Cheetowaga, NY. In fact, it is not only the first machine learning algorithm, but also the first to be implemented in hardware, by Rosenblatt. The motivation for perceptrons comes directly from the model by McCulloch-Pitts [2] for the artificial neurons. The neuron model states that multiple inputs enter the main neuron (*soma*) through multiple *dendrites* and the output is sent out

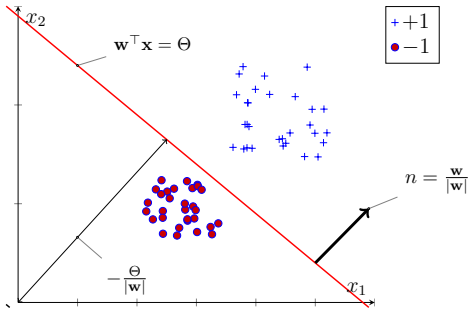
through a single *axon*.



Simply put, a perceptron computes a weighted sum of the input attributes and adds a bias term. The sum is compared to a threshold to classify the input as $+1$ or -1.

The bias term Θ signifies that the weighted sum of the actual attributes should be greater than Θ to become greater than 0.

1.1 Geometric Interpretation



1. The hyperplane, characterized by $\mathbf{w}$ is also known as the decision boundary (or surface).
2. The decision boundary divides the input space into two *half-spaces*.

3. Changing $\mathbf{w}$ *rotates* the hyperplane
4. Changing Θ *translates* the hyperplane
5. Hyperplane passes through the origin if $\Theta = 0$

1.2 Perceptron Training

- Add another attribute $x_{d+1} = 1$.
- w_{d+1} is $-\Theta$
- Desired hyperplane goes through origin in $(d+1)$ space
- **Assumption**: $\exists \mathbf{w} \in \mathbb{R}^{d+1}$ such that $\mathbf{w}$ can *strictly* classify all examples correctly.
- *Hypothesis space*: Set of all hyperplanes defined in the $(d+1)$-dimensional space passing through origin
  - The target hypothesis is also called **decision surface** or **decision boundary**.

```
w ← (0, 0, ..., 0)_{d+1}
for i=1, 2, ... do
   if w^T x^(i) > 0 then
      c(x^(i)) = +1
   else
      c(x^(i)) = −1
   end if
   if c(x^(i)) ≠ c_*(x^(i)) then
      w ← w + c_*(x^(i)) x^(i)
   end if
end for
```

- Every mistake *tweaks* the hyperplane
  - Rotation in $(d+1)$ space
  - Accomodate the offending point
- Stopping Criterion:

  - Exhaust all training examples, or
  - No further updates

To demonstrate the working of perceptron training algorithm, let us consider a simple 2D example in which all data points are located on a unit circle.

To get an intuitive sense of why this training procedure should work, we should note the following: Every mistake makes $\mathbf{w}^\top \mathbf{x}$ become more positive (if $c_*(x) = 1$) or more negative (if $c_*(x) = -1$).

Let $c_*(x) = 1$. The new $\mathbf{w}' = \mathbf{w} + \mathbf{x}$. Hence, by substitution, $\mathbf{w}'^\top \mathbf{x} = (\mathbf{w} + \mathbf{x})^\top \mathbf{x} = \mathbf{w}^\top \mathbf{x} + \mathbf{x}^\top \mathbf{x}$. The latter quantity is always positive, thereby making $\mathbf{w}^\top \mathbf{x}$ more positive.

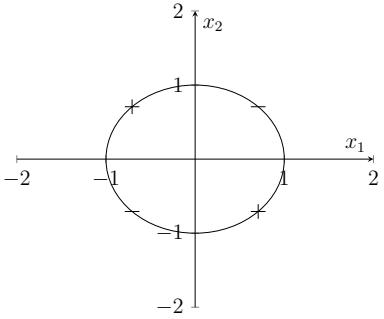
Thus whenever a mistake is made, the surface is "moved" to accomodate that mistake by increasing or decreasing $\mathbf{w}^\top \mathbf{x}$.

Sometimes a "learning rate" (η) is used to update the weights.

2 Perceptron Convergence

In the previous example we observe that the training algorithm is observed to converge to a separating hyperplane when points were distributed on a unit hypersphere and were linearly separable. Can we generalize the convergence guarantee for the perceptron training.

Obviously, the assumption made earlier has to hold, i.e., the training examples must be linearly separable.



1. Linearly separable examples

2. No errors
3. $|\mathbf{x}| = 1$
4. A positive δ gap exists that "contains" the target concept (hyperplane)
   - $(\exists\delta)(\exists\mathbf{v})$ such that $(\forall\mathbf{x})\mathbf{v}^\top\mathbf{x} > c_*(\mathbf{x})\delta$.

The last assumptions "relaxes" the requirement that w converges to the target hyperplane exactly.

Theorem 1. *For a set of unit length and linearly separable examples, the perceptron learning algorithm will converge after a finite number of mistakes (at most $\frac{1}{\delta^2}$).*

Proof discussed in Minsky's book [3]. Note that while this theorem guarantees convergence for the algorithm, its dependence on δ makes it inefficient for *hard* classes. In his book, *Perceptrons* [3] Minsky criticized perceptrons for the same reason. If, for a certain class of problem, δ is very small, the number of mistakes will be very high.

Even for classes over boolean variables, there exist cases where the gap δ is exponentially small, $\delta \approx 2^{-kd}$. The convergence theorem states that there could be $\approx 2^{kd}$ mistakes before the algorithm converges. Note that for d binary attributes, one can have $2^{\Theta}(d^2)$ possible linear threshold hyperplanes. Using the halving algorithm, one can learn the hyperplane with $O(\log_2 2^{\Theta}(d^2)) = O(d^2)$ mistakes, instead of $O(\frac{1}{\delta^2})$ mistakes made by the perceptron learning algorithm. But the latter is implementable and works better in practice.

Taking the next step

Hypothesis Space, $\mathcal{H}$

- Conjunctive
- Disjunctive
  - Disjunctions of k attributes
- Linear hyperplanes
- $\mathbf{c}_* \in \mathcal{H}$

- $\mathbf{c}_* \notin \mathcal{H}$

Input Space, x

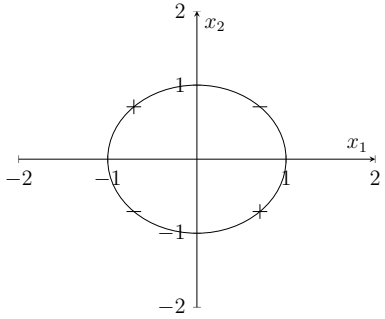
- $\mathbf{x} \in \{0,1\}^d$
- $\mathbf{x} \in \mathbb{R}^d$

Input Space, y

- $y \in \{0,1\}$
- $y \in \{-1,+1\}$
- $y \in \mathbb{R}$

3 Perceptron Learning in Non-separable Case

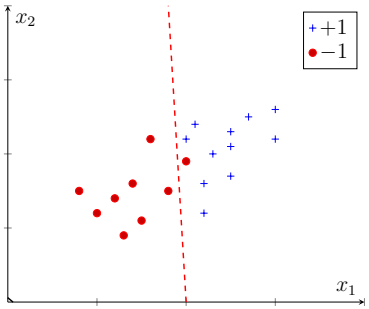
- Expand $\mathcal{H}$?
- Lower expectations
  - *Principle of good enough*



Expanding $\mathcal{H}$ so that it includes the target concept is tempting, but it also makes the search more challenging.

Moreover, one should always keep in mind that $\mathcal{C}$ is mostly unknown, so it is not even clear how much $\mathcal{H}$ should be expanded to.

Another way to address the XOR problem is to use different input attributes for the examples. For example, in this case, using the polar attributes, $\langle r, \theta \rangle$ can easily result in a simple threshold to discriminate between the positive and negative examples.



The example above clearly appears to be non-separable. Geometrically, one can always prove if two sets of points are separable by a straight line or not. If the convex hulls for each set intersect, the points are not linearly separable, otherwise they are.

To learn a linear decision boundary, one needs to "tolerate" mistakes. The question then becomes, which would be the best possible hyperplane, allowing for mistakes on the training data. Note that at this point we have moved from *online learning* to *batch learning*, where a batch of training examples is used to learn the best possible linear decision boundary.

In terms of the hypothesis search, instead of finding the target concept in the hypothesis space, we relax the assumption that the target concept belongs to the hypothesis space. Instead, we focus on finding the *most probable* hypothesis.

4 Gradient Descent and Delta Rule

- Which hyperplane to choose?
- Gives **best performance** on training data
  - Pose as an optimization problem

  - Objective function?
  - Optimization procedure?

4.1 Objective Function for Perceptron Learning

- An unthresholded perceptron (a linear unit)
- Training Examples: $\langle \mathbf{x}_i, y_i \rangle$
- Weight: $\mathbf{w}$

$$E(\mathbf{w}) = \frac{1}{2} \sum_i (y_i - \mathbf{w}^\top \mathbf{x}_i)^2$$

Note that we are denoting the weight vector as a vector in the coordinate space (denoted by w). The output y_i is a binary output $(0, 1)$.

4.2 Machine Learning as Optimization

At this point, we move away from the situation where a perfect solution exists, and the learning task it to reach the perfect solution. Instead, we focus on finding the *best possible* solution which optimizes certain criterion.

- Learning is optimization
- Faster optimization methods for faster learning
- Let $w \in \mathbb{R}^d$ and $S \subset \mathbb{R}^d$ and $f_0(w), f_1(w), \ldots, f_m(w)$ be real-valued functions.
- Standard optimization formulation is:

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & f_0(w) \\ \text{subject to} \quad & f_i(w) \leq 0, \ i = 1, \ldots, m. \end{aligned}$$

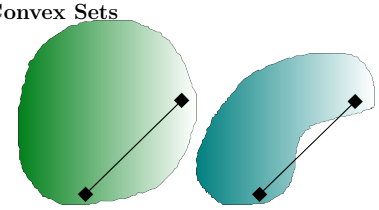
- Methods for **general optimization problems**

[1] Adapted from `http://ttic.uchicago.edu/~gregory/courses/ml2012/lectures/tutorial_optimization.pdf`. Also see, `http://www.stanford.edu/~boyd/cvxbook/` and `http://scipy-lectures.github.io/advanced/mathematical_optimization/`.

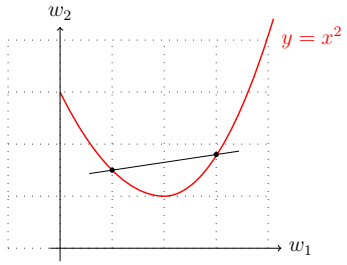
- Simulated annealing, genetic algorithms

- Exploiting *structure* in the optimization problem
  - **Convexity**, Lipschitz continuity, smoothness

Convex Sets



Convex Functions



Convexity is a property of certain functions which can be exploited by optimization algorithms. The idea of convexity can be understood by first considering *convex sets*. A convex set is a set of points in a coordinate space such that every point on the line segment joining any two points in the set are also within the set. Mathematically, this can be written as:

$$w_1, w_2 \in S \Rightarrow \lambda w_1 + (1-\lambda) w_2 \in S$$

where $\lambda \in [0, 1]$. A *convex function* is defined as follows:

- $f : \mathbb{R}^d \to \mathbb{R}$ is a convex function if the domain of f is a convex set and for all $\lambda \in [0, 1]$:

$$f(\lambda w_1 + (1-\lambda) w_2) \leq \lambda f(w_1) + (1-\lambda) f(w_2)$$

Some examples of convex functions are:

- Affine functions: $w^\top x + b$
- $||w||_p$ for $p \geq 1$
- Logistic loss: $\log(1 + e^{-yw^\top x})$

4.3 Convex Optimization

- Optimality Criterion

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & f_0(w) \\ \text{subject to} \quad & f_i(w) \leq 0,\ i = 1, \ldots, m. \end{aligned}$$

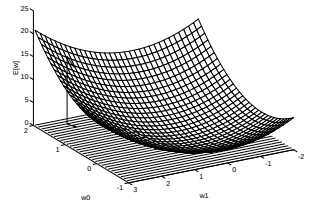
  where all $f_i(w)$ are **convex functions**.

- w_0 is feasible if $w_0 \in Dom\ f_0$ and all constraints are satisfied
- A feasible w^* is optimal if $f_0(w^*) \leq f_0(w)$ for all w satisfying the constraints

4.4 Gradient Descent

- Denotes the direction of steepest ascent

$$\nabla E(\mathbf{w}) = \begin{bmatrix} \frac{\partial E}{\partial w_0} \\ \frac{\partial E}{\partial w_1} \\ \vdots \\ \frac{\partial E}{\partial w_d} \end{bmatrix}$$



A small step in the weight space from $\mathbf{w}$ to $\mathbf{w} + \delta\mathbf{w}$ changes the objective (or error) function. This change is maximum if $\delta\mathbf{w}$ is along the direction of the gradient at $\mathbf{w}$ and is given by:

$$\delta E \simeq \delta\mathbf{w}^\top \nabla E(\mathbf{w})$$

Since $E(\mathbf{w})$ is a smooth continuous function of $\mathbf{w}$, the extreme values of E will occur at the location in the input space ($\mathbf{w}$) where the gradient of the error function vanishes, such that:

$$\nabla E(\mathbf{w}) = 0$$

The vanishing points can be further analyzed to identify them as saddle, minima, or maxima points.

One can also derive the local approximations done by first order and second order methods using the Taylor expansion of $E(\mathbf{w})$ around some point $\mathbf{w}'$ in the weight space.

$$E(\mathbf{w}') \simeq E(\mathbf{w}) + (\mathbf{w}' - \mathbf{w})^\top \nabla + \frac{1}{2}(\mathbf{w}' - \mathbf{w})^\top \mathbf{H}(\mathbf{w}' - \mathbf{w})$$

For first order optimization methods, we ignore the second order derivative (denoted by $\mathbf{H}$ or the *Hessian*). It is easy to see that for $\mathbf{w}$ to be the local

minimum, $E(\mathbf{w}) - E(\mathbf{w}') \leq 0$, $\forall \mathbf{w}'$ in the vicinity of $\mathbf{w}$. Since we can choose any arbitrary $\mathbf{w}'$, it means that every component of the gradient ∇ must be zero.

- Set derivative to 0
- Second derivative for minima or maxima

1. Start from any point in variable space
2. Move along the direction of the steepest descent (or ascent)
   - By how much?
   - A learning rate (η)
   - What is the direction of steepest descent?
     - Gradient of E at $\mathbf{w}$

Gradient descent is a first-order optimization method for convex optimization problems. It is analogous to "hill-climbing" where the gradient indicates the direction of steepest ascent in the local sense.

Training Rule for Gradient Descent

$$\mathbf{w} = \mathbf{w} - \eta \nabla E(\mathbf{w})$$

For each weight component:

$$w_j = w_j - \eta \frac{\partial E}{\partial w_j}$$

The key operation in the above update step is the calculation of each partial derivative. This can be computed for perceptron error function as follows:

$$\begin{aligned}
\frac{\partial E}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 \\
&= \frac{1}{2} \sum_i \frac{\partial}{\partial w_j} (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 \\
&= \frac{1}{2} \sum_i 2(y_i - \mathbf{w}^\top \mathbf{x}_i) \frac{\partial}{\partial w_j} (y_i - \mathbf{w}^\top \mathbf{x}_i) \\
&= \sum_i (y_i - \mathbf{w}^\top \mathbf{x}_i)(-x_{ij})
\end{aligned}$$

where x_{ij} denotes the j^{th} attribute value for the i^{th} training example. The final weight update rule becomes:

$$w_j = w_j + \eta \sum_i (y_i - \mathbf{w}^\top \mathbf{x}_i) x_{ij}$$

- Error surface contains only one global minimum
- Algorithm *will* converge
  - Examples need not be linearly separable
- η should be *small enough*
- Impact of too large η?
- Too small η?

If the learning rate is set very large, the algorithm runs the risk of overshooting the target minima. For very small values, the algorithm will converge very slowly. Often, η is set to a moderately high value and reduced after each iteration.

4.5 Issues with Gradient Descent

- Slow convergence
- Stuck in local minima

One should note that the second issue will not arise in the case of Perceptron training as the error surface has only one global minima. But for general setting, including multi-layer perceptrons, this is a typical issue.

More efficient algorithms exist for batch optimization, including *Conjugate Gradient Descent* and other *quasi*-Newton methods. Another approach is to consider training examples in an online or incremental fashion, resulting in an online algorithm called **Stochastic Gradient Descent** [1], which will be discussed next.

4.6 Stochastic Gradient Descent

- **Update weights after every training example.**
- For sufficiently small η, closely approximates Gradient Descent.

Gradient Descent	**Stochastic Gradient Descent**
Weights updated after summing error over all examples	Weights updated after examining each example
More computations per weight update step	Significantly lesser computations
Risk of local minima	Avoids local minima

References

References

[1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, Dec. 1989.

[2] W. Mcculloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[3] M. L. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

[4] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.