Notes on the Perceptron algorithm and kernel methods

COMS 4252, Fall 2023

Columbia University

1 The Perceptron Algorithm

The Perceptron algorithm is about 30 years older than the Winnow algorithm; Winnow (due to Nick Littlestone) is from the late 1980s, while the Perceptron algorithm was given by Frank Rosenblatt in the late 1950s. Like the Winnow algorithm, the Perceptron algorithm maintains a linear threshold function as its hypothesis; the chief difference is that Perceptron makes *additive* updates to its hypothesis weight vector (rather than the multiplicative updates that Winnow makes).

The version of the Perceptron algorithm that we will consider can be used to learn a target LTF of the form $c(x) = \text{sign}(v \cdot x)$ with a finite mistake bound, provided that the examples used for learning satisfy a certain separation/"margin" condition described below. An LTF of this sort (for which the threshold θ is zero) is sometimes called an *origin-centered LTF* (because the separating hyperplane, which is the set of points x satisfying $v \cdot x = 0$, passes through the origin 0^n in $\mathbb{R}^n$). It is convenient to think of the target LTF as outputting values in ± 1 rather than $\{0, 1\}$; the function $\text{sign}(t)$ takes value 1 if $t \geq 0$ and takes value -1 otherwise. Thus the target concept outputs $+1$ or -1 according to whether or not $v \cdot x \geq 0$.

1.1 The Perceptron algorithm

The Perceptron algorithm is very simple. It works as follows:

- The initial hypothesis is the LTF $h(x) = \text{sign}(w \cdot x)$ where the weight vector w is initially the all-0 vector. (The algorithm works by updating the weight vector w; its hypothesis is always $h(x) = \text{sign}(w \cdot x)$.)
- Given an example $x \in \mathbb{R}^n$ and its label $c(x) = \text{sign}(v \cdot x)$,
  - if $h(x) = c(x)$ then (as usual) no update is performed;
  - if $h(x) \neq c(x)$, then the weight vector w is updated by setting w_{new} to $w + c(x)x$. Note that $c(x) \in \{-1, 1\}$, so this update either adds the vector x to w (if $h(x) = -1$ and $c(x) = 1$) or subtracts x from w (if $h(x) = 1$ and $c(x) = -1$).

A moment's thought shows that the update rule of the Perceptron algorithm is a reasonable one. Suppose an example x causes the Perceptron algorithm to make a false positive mistake,

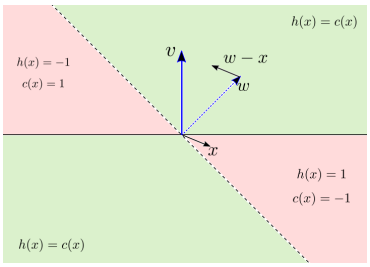
i.e. $h(x) = 1$ (so $w \cdot x > 0$) but $c(x) = -1$ (so $v \cdot x < 0$). The new weight vector after the update will be $w - x$, which will have a smaller inner product with the just-received example x than the previous weight vector w, since

$$(w - x) \cdot x = w \cdot x - x \cdot x < w \cdot x,$$

(assuming x is not the identically-0 vector), so intuitively the weight vector is "being adjusted in the right direction," at least as far as the just-received example x is concerned. (A similar argument goes through for false negative mistakes.)

To help with geometric intuition, below is a pictorial representation of what happens with a false positive mistake: the horizontal solid black line is the separating hyperplane consisting of all points whose inner product with the target vector v is zero, and points above that line have positive inner product with v (hence are points that the target concept c would label as positive). The solid blue arrow depicts the target vector v (which is orthogonal to the separating hyperplane). The dashed black line is the hyperplane consisting of all points whose inner product with the hypothesis vector w is zero; points above that line have positive inner product with w (hence are points that the hypothesis would label as positive). The dashed blue arrow depicts the hypothesis vector w.

The example x that is shown has $h(x) = 1$ (because it is above the dashed line) but $c(x) = -1$ (because it is below the solid line). Hence the Perceptron algorithm updates the hypothesis vector from w to $w - x$. As the figure shows, this causes the hypothesis vector to "get closer" to the target vector v.



1.2 The Perceptron Convergence Theorem

The Perceptron Convergence Theorem, first proved by Novikoff in the early 1960s, establishes a mistake bound for the Perceptron algorithm under suitable conditions on the sequence of examples it is given. (We write $\|z\|$ below to denote the Euclidean length $\sqrt{z_1^2 + \cdots + z_n^2}$ of a vector in $\mathbb{R}^n$.)

Theorem 1.1. (Perceptron Convergence Theorem) *Suppose the Perceptron algorithm is run on a sequence of examples $x^1, x^2, \cdots \in \mathbb{R}^n$ where the target concept c is $c(x) = \text{sign}(v \cdot x)$. Suppose that $\|v\| = 1$, that each example has $\|x^i\| = 1$, and that $\min_{x^i} |v \cdot x^i| = \delta > 0$, where the min is taken over all examples that are given to the Perceptron algorithm in the course of its execution. Then the Perceptron algorithm makes at most $1/\delta^2$ mistakes on this sequence of examples.*

Before proving the theorem, we make a few comments. First, it is notable that the mistake bound depends neither on the dimension n nor on the number of examples given to Perceptron, only on the minimum "separation" (or "margin") parameter δ. Second the δ parameter can be thought of as the minimum Euclidean distance that any example lies away from the separating hyperplane.

Finally, it is often possible to reduce other LTF learning scenarios (in which the target LTF has a nonzero threshold θ, or the examples are not guaranteed to all have Euclidean norm exactly 1) to the setting of the Perceptron Convergence Theorem (satisfying the assumptions $\|v\| = \|x^i\| = 1$ and $\theta = 0$) by simple geometric transformations.

The proof of the theorem follows from a few simple claims. These claims analyze two key quantities: $w \cdot v$ (equivalently, the (signed) length of the projection of the hypothesis vector w onto the direction of the target vector v) and $w \cdot w = \|w\|^2$ (equivalently, the squared length of the hypothesis vector).

Claim 1.2. *After Perceptron has made M mistakes, we have $w \cdot v \geq \delta M$.*

Proof. Initially $w = (0, \ldots, 0)$ so $w \cdot v = 0$. We show that each mistake causes $w \cdot v$ to increase by at least δ; this clearly proves the claim. If w is the old hypothesis vector (before a mistake is made on example x), then $w + \text{sign}(v \cdot x)x$ is the new hypothesis vector. We have that

$$(w + \text{sign}(v \cdot x)x) \cdot v = w \cdot v + \text{sign}(v \cdot x)v \cdot x = w \cdot v + |v \cdot x| \geq w \cdot v + \delta$$

which gives the claim. □

Claim 1.3. *After Perceptron has made M mistakes, we have $\|w\|^2 \leq M$.*

Proof. Initially $w = 0$ so $\|w\|^2 = 0$. We show that each mistake causes $\|w\|^2$ to go up by at most 1:

$$\begin{aligned} \|w + \text{sign}(v \cdot x)x\|^2 &= w \cdot w + 2\text{sign}(v \cdot x)w \cdot x + \text{sign}(v \cdot x)^2 x \cdot x \\ &= \|w\|^2 + 1 + 2\text{sign}(v \cdot x)w \cdot x \leq \|w\|^2 + 1, \end{aligned}$$

where the inequality holds because we made a mistake and this means the sign of $v \cdot x$ differs from the sign of $w \cdot x$. □

With these two claims in hand the proof of the theorem is very simple. Recall that for any vector w and unit vector v, we have that $w \cdot v = \|w\| \cos(\alpha)$, where α is the angle between w and v, and hence $w \cdot v \leq \|w\|$. So if the Perceptron algorithm has made M mistakes at some point in its execution, then by the above claims it must be the case that

$$\delta M \overset{\text{first claim}}{\leq} w \cdot v \leq \|w\| \overset{\text{second claim}}{\leq} \sqrt{M},$$

so $M \leq 1/\delta^2$. □

One nice thing about the Perceptron algorithm is that it (or rather a dual form of it) plays nicely with *kernel functions*, which we describe below. This makes it possible to sometimes run Perceptron in a computationally efficient way even over very high (or even infinite!) sdimensional spaces, without incurring a running time penalty (to evaluate or update the hypothesis) that scales with the dimension of the relevant space.

2 A Rough and Casual Introduction to Kernels and Kernel Perceptron

There is a great deal of sophisticated mathematical theory dealing with kernels; to keep things simple we will ignore almost all of it, and confine ourselves below to simple and concrete finite-dimensional settings.

2.1 Kernel Functions

Let X,X' be two instance spaces each of which is equipped with an inner product. You are encouraged to think of $X = \mathbb{R}^n$ and $X' = \mathbb{R}^N$, where $N \gg n$ (we'll give an explicit example below).

A *feature expansion* is a mapping $\Phi : X \to X'$. If $X = \mathbb{R}^n$ and $X' = \mathbb{R}^N$, then this means that $\Phi(\cdot)$ takes as input an n-dimensional vector (think of this as a data point described by n numerical features), and its output is a description of the data point that uses N numerical features.

Example: Suppose that X is $\{0,1\}^n$, that X' is $\{0,1\}^{2^n}$ (so $N = 2^n$), and that $\Phi : \{0,1\}^n \to \{0,1\}^{2^n}$ is the feature expansion which has one feature for every possible monotone conjunction over the input variables $x_1, \ldots, x_n$. For example, if $n = 3$ then $\Phi(x_1, x_2, x_3)$ equals

$$(1, x_1, x_2, x_3, x_1 \wedge x_2, x_1 \wedge x_3, x_2 \wedge x_3, x_1 \wedge x_2 \wedge x_3).$$

It is helpful to think of $\Phi(x)$ as an "expanded" version of x which "explicitly writes down" information that is only implicitly represented in the vector x. (This particular feature expansion writes down the value of all monotone conjunctions of the "base features" $x_1, \ldots, x_n$.)

Now the key definition: a *kernel for feature expansion* Φ is the function $K(\cdot, \cdot) : X \times X \to \mathbb{R}$ that computes inner products in X' according to Φ. In other words, a kernel K for Φ is the function for which

$$K(\underbrace{a, b}_{\text{in } X}) = \underbrace{\Phi(a)}_{\text{in } X'} \cdot \underbrace{\Phi(b)}_{\text{in } X'}.$$

Why is this useful? If you had all the time in the world, it wouldn't be useful: given any two examples $a, b \in X$, you could write down $\Phi(a)$, write down $\Phi(b)$, and compute the inner product $\Phi(a) \cdot \Phi(b)$ yourself, without ever thinking about the kernel function K. But if the dimension of X' (the "N" parameter from earlier) is very high, then even just writing down $\Phi(a)$ or $\Phi(b)$ (each of which is a vector of length N) could take an enormous amount of time — this is certainly the case for the example we saw earlier, where N is 2^n! So if we are cognizant of time constraints (which we very much are), then this explicit approach to computing $\Phi(a) \cdot \Phi(b)$ might not be workable.

Kernels offer a potential shortcut. **For some feature expansions Φ, it turns out that it is possible to evaluate the kernel K corresponding to Φ much faster than writing out $\Phi(a)$ and $\Phi(b)$ explicitly.** Let's return to the example from earlier to see how this can happen:

Above Example Revisited: We have $X = \{0,1\}^n$ and $X' = \{0,1\}^{2^n}$, where $\Phi(\cdot)$ is the "all-monotone-conjunctions feature expansion" as described above. We'll now see that given $a, b \in \{0,1\}^n$, the "monotone conjunction kernel" $K(a,b)$ for the monotone conjunction feature expansion Φ can be computed in time poly(n), even though the dimension N of the feature space X' is 2^n.

Let $\texttt{Ones}(a,b) \subseteq [n]$ denote the set of all bit positions $i \in \{1,\ldots,n\}$ such that $a_i = b_i = 1$. We have the following simple claim:

Claim 2.1. $K(a,b) = 2^{|\texttt{Ones}(a,b)|}$.

Proof. Every subset $S \subseteq \texttt{Ones}(a,b)$ corresponds to a different monotone conjunction such that that coordinate is 1 in both $\Phi(a)$ and $\Phi(b)$. Moreover, it is easy to see that these are the only such conjunctions that evaluate to 1 in both $\Phi(a)$ and in $\Phi(b)$ (because if S contains some $i \in [n] \setminus \texttt{Ones}(a,b)$, then either a_i or b_i is 0, so the corresponding conjunction in either $\Phi(a)$ or in $\Phi(b)$ will evaluate to 0 and not 1).
So, imagining that we wrote out the entire inner product as a sum of 2^n values (each of which is $0 \cdot 0$, $0 \cdot 1$, $1 \cdot 0$, or $1 \cdot 1$), we see that the number of $1 \cdot 1$ summands is precisely $2^{|\texttt{Ones}(a,b)|}$, so $\Phi(a) \cdot \Phi(b) = K(a,b) = 2^{|\texttt{Ones}(a,b)|}$. □

Given the claim, clearly it is trivial to compute $K(a,b) = 2^{|\texttt{Ones}(a,b)|}$ in $O(n)$ time, which is indeed faster than we could write out the 2^n dimensional vector $\Phi(a)$.

So, we see that indeed there are interesting very high dimensional feature expansions for which the associated kernels can be computed efficiently.

Why is this useful? The answer is because it means that any learning algorithm which "works only by computing inner products between examples" can in fact be run over the (high-dimensional) expanded feature space (using $\Phi(x) \in X'$ in place of each example $x \in X$), without paying the computational price of being in high dimension. As we will see below, the Perceptron algorithm (after a change of perspective) turns out to be such an algorithm.

2.2 Dual Formulation of Perceptron

As described in Section 1, the Perceptron algorithm works by maintaining a hypothesis vector $w \in \mathbb{R}^n$. However, it turns out that it is also possible to give an equivalent formulation of the Perceptron algorithm in which a collection of examples (the ones on which it has made a mistake) are used in place of the hypothesis vector. This formulation, sometimes called "dual Perceptron" has the advantage that to run the algorithm it is only necessary to take inner products between examples in $X = \mathbb{R}^n$. (So, from our discussion above, this means it is possible to run a "kernelized form" of Perceptron in a computationally efficient way, when we have a feature expansion with an associated kernel function that can be evaluated computationally efficiently, and thereby run the algorithm over the high-dimensional expanded feature space X' in a computationally efficient way; more on this in the next subsection.)

The key insight of the dual Perceptron formulation is the following: Suppose you are at some intermediate step in running Perceptron, and up to this point the mistakes that the Perceptron algorithm has made have been on examples $x^{i_1}, \ldots, x^{i_k} \in \mathbb{R}^n$ (with associated labels $c(x^{i_1}), \ldots, c(x^{i_k}) \in \{-1, 1\}$). Then at this point in the execution of Perceptron, the hypothesis

vector w is precisely

$$w = \sum_{j=1}^{k} c(x^{i_j}) x^{i_j},$$

and consequently the value of $w \cdot x$ (the key thing that the Perceptron needs to compute to make its prediction) is precisely

$$w \cdot x = \left(\sum_{j=1}^{k} c(x^{i_j}) x^{i_j} \right) \cdot x = \sum_{j=1}^{k} c(x^{i_j}) x^{i_j} \cdot x. \tag{1}$$

This makes it clear that in order to run (the dual) Perceptron, it is only necessary to be able to compute inner products between examples. The dual Perceptron stores every example x^{i_j} on which a mistake is made and its associated label $c(x^{i_j})$. With these values in hand, Equation (2) makes it clear that it is possible to compute $w \cdot x$ by computing k inner products (between x and x^{i_1}, between x and x^{i_2}, ..., between x and x^{i_k}) and combining the results using the $c(x^{i_j})$'s.

2.3 Kernelized Dual Perceptron

If you're running Perceptron over the original feature space X, it probably doesn't make sense to use the dual formulation - after all, it requires that you keep a list of all the examples that mistakes have been made on so far, and as k gets larger and larger it takes more and more time to compute the hypothesis $\text{sign}(w \cdot x)$. But the dual perceptron makes it possible to run the Perceptron algorithm over the high-dimensional expanded feature space X' corresponding to the feature expansion Φ, without ever "getting your hands dirty" and having to actually write down an element of that high dimensional space! If you have a kernel corresponding to the feature expansion Φ, you can simply replace every inner product between two elements of X with an evaluation of the kernel, and the net effect will be exactly the same as if you had explicitly expanded out each example $x \in X$ to the corresponding expanded version $\Phi(x) \in X'$.

So to summarize, the "kernelized dual Perceptron" perfectly simulates what would happen if you ran the Perceptron algorithm over X' rather than X, using the high-dimensional feature expansion version $\Phi(x)$ of each example x. If the examples $x \in X$ are labeled according to $c(x) = \text{sign}(v \cdot \Phi(x))$ (note that v is an N-dimensional vector now, not an n-dimensional vector as in the original version), then the N-dimensional hypothesis vector w after k mistakes on examples $x^{i_1}, \ldots, x^{i_k}$ is now

$$w = \sum_{j=1}^{k} c(x^{i_j}) \Phi(x^{i_j}),$$

and the value of $w \cdot \Phi(x)$ (the key thing that needs to be evaluated so that the Perceptron algorithm over X' can make its prediction) is precisely

$$w \cdot \Phi(x) = \left(\sum_{j=1}^{k} c(x^{i_j}) x^{i_j} \right) \cdot x = \sum_{j=1}^{k} c(x^{i_j}) \Phi(x^{i_j}) \cdot \Phi(x) = \sum_{j=1}^{k} c(x^{i_j}) K(x^{i_j}, x), \tag{2}$$

which can be computed efficiently (for k not too large) if $K(\cdot, \cdot)$ can be computed efficiently. (Note that in running the kernelized dual Perceptron algorithm, the high-dimensional hypothesis vector w is never explicitly written down!)

So indeed, kernel functions can (sometimes) make it possible to run the Perceptron algorithm efficiently over very high (even exponentially high) dimensional feature spaces.