Kernels

Nuno Vasconcelos

ECE Department, UCSD
Classification

- a classification problem has two types of variables
  - $X$ - vector of observations (features) in the world
  - $Y$ - state (class) of the world

- e.g.
  - $x \in X \subseteq \mathbb{R}^2$ = (fever, blood pressure)
  - $y \in Y = \{\text{disease, no disease}\}$

- $X, Y$ related by a (unknown) function

\[
\begin{align*}
X & \quad \rightarrow \quad f(.) \\
\rightarrow y = f(x)
\end{align*}
\]

- goal: design a classifier $h: X \rightarrow Y$ such that $h(x) = f(x) \ \forall x$
Perceptron

- classifier implements the linear decision rule

\[ h(x) = \text{sgn}[g(x)] \text{ with } g(x) = w^T x + b \]

- learning is formulated as an optimization problem
  
  - define set of errors
    \[ E = \left\{ x_i \mid y_i (w^T x_i + b) < 0 \right\} \]
  
  - define the cost
    \[ J_p(w, b) = - \sum_{i \mid x_i \in E} y_i (w^T x_i + b) \]
  
  - and minimize
  
  - \( J_p \) cannot be negative since, in \( E \), all \( y_i (w^T x_i + b) \) are negative
  
  - at zero we know we have the best possible solution (\( E \) empty)
Perceptron learning

is simple **stochastic gradient descent** on this cost:

```
set k = 0, w_k = 0, b_k = 0
set R = max_i ||x_i||
do {
    for i = 1:n {
        if y_i(w^T x_i + b_k) < 0 then {
            w_{k+1} = w_k + \eta y_i x_i
            b_{k+1} = b_k + \eta y_i R^2
            k = k+1
        }
    }
} until y_i(w^T x_i + b_k) \geq 0, \forall i (no errors)
```
Perceptron learning

- The interesting part is that this is guaranteed to converge in finite time.

**Theorem:** Let $D = \{(x_1,y_1), \ldots, (x_n,y_n)\}$ and

$$R = \max_i \|x_i\|$$

If there is $(w^*,b^*)$ such that $\|w^*\| = 1$ and

$$y_i(w^T x_i + b^*) > \gamma, \quad \forall i$$

then the Perceptron will find an error free hyper-plane in at most

$$\left(\frac{2R}{\gamma}\right)^2$$

iterations.

- The main problem is that it only implements a linear discriminant.
Linear discriminant

▶ when is this a good decision function?
▶ clearly works if data is **linearly separable**
  - there is a plane which has
    - all 0’s on one side
    - all 1’s on the other
▶ we have also shown that it is **optimal** for
  - Gaussian classes
  - equal class probability and covariance
▶ but clearly **will not work even for**
  - only slightly more general Gaussian cases
▶ what are possible solutions to this problem?
Alternatives

1) more complex classifier
   - let’s try to avoid this

2) transform the space:
   - introduce a mapping
     $$\Phi: \mathcal{X} \rightarrow \mathcal{Z}$$
     such that $\dim(\mathcal{Z}) > \dim(\mathcal{X})$

learning a linear boundary in $\mathcal{Z}$ is equivalent to learning a non-linear boundary in $\mathcal{X}$

how do we do this?
   - we already mentioned two possibilities
Solution one

- using Bayes decision rule

\[ h(x) = \arg \max_i P_{X \mid Y}[x \mid i] P_Y[i] \]

- e.g. for \( P_{X \mid Y}(x \mid i) = G(x, 0, \sigma_i) \)

leads to this

- cannot be implemented with a linear discriminant

- but becomes feasible by mapping to 2D

\[ \Phi : \mathbb{R} \rightarrow \mathbb{R}^2 \]
\[ x \rightarrow (x, x^2) \]
In general

because the BDR is

- pick \( h(x) = 1 \) if
  \[
  \frac{P_{X|Y}[x \mid 1]P_Y[1]}{P_{X|Y}[x \mid -1]P_Y[-1]} > 1
  \]
- and \( h(x) = -1 \) otherwise

the mapping

\[
\Phi_{BDR} : \mathbb{R}^d \rightarrow \mathbb{R}^{d+1}, \quad \text{with} \quad \Phi_{BDR}(x) = \left( x, \frac{P_{X|Y}[x \mid 1]P_Y[1]}{P_{X|Y}[x \mid -1]P_Y[-1]} \right)
\]

always works, since the hyperplane

\[
w^T \Phi_{BDR}(x) + b = 0, \quad \text{with} \quad w = (0, \ldots, 0, 1)^T \quad \text{and} \quad b = -1
\]

optimally separates the classes
Solution two

- add Perceptron layers: MLP as non-linear feature transformation + linear discriminant

\[ y = \Phi(x) \]
MLP feature transformation

- learned by the backpropagation algorithm
- gradient descent on \( J(W) = \sum_{i=1}^{n} [t_i - z(x_i; W)]^2 \)
- for any pair \((i,j)\)
  \[
  \frac{\partial J}{\partial w_{ji}} = -\delta_j x_i
  \]
- with
  \[
  \delta_j = \sum_k \delta_k w_{kj}
  \]
  if \(j\) is hidden and
  \[
  \delta_j = -(t_j - z_j)s'(u_j)
  \]
  if \(j\) is output. The weight updates are
  \[
  w_{ji}^{(n+1)} = w_{ji}^{(n+1)} - \eta \frac{\partial J}{w_{ji}}
  \]
Problems

- while theoretically feasible, these solutions have various problems

- $\Phi_{BDR}$:
  - requires the knowledge of the densities $P_{X|Y(x|i)}$
  - density estimation is quite hard, specially when $d$ is large

- $\Phi_{MLP}$:
  - takes long time to search
  - how many hidden units, layers, what configuration?
  - exhaustive search is infeasible

- the problem is that we have been trying to do this “on the cheap”:
  - from $\mathbb{R}^d$ to $\mathbb{R}^k$ where $k > d$ but not by much!
Alternative

▷ let’s aim big and make $k = \dim [\Phi(X)]$ really large

▷ intuitively, for larger $k$:
  • it will be easier to separate the classes linearly, i.e.
  • the set of mappings that achieves linear separation grows

▷ here is a bold plan:
  • let’s pick $\Phi(x)$ randomly
  • as $k \to \infty$ the probability that this make the data linearly separable increases

▷ where do we stop?
  • well, we can go all the way to $k = \infty$
  • i.e. we map each point into a function

$$\Phi(X) = (\phi_1(X), \ldots, \phi_k(X)) \xrightarrow{k \to \infty} \phi(X; t)$$
Implementation

- at first, this looks like a bad idea

```plaintext
set k = 0, w_k = 0, b_k = 0
set R = \max_i \|x_i\|

do {
  for i = 1:n {
    if y_i(w^T \Phi(x_i) + b_k) < 0 then {
      - w_{k+1} = w_k + \eta y_i x_i
      - b_{k+1} = b_k + \eta y_i R^2
      - k = k+1
    }
  }
}

} until y_i(w^T \Phi(x_i) + b_k) \geq 0, \forall i (no errors)
```

- how do I:
  - compute \(w^T \Phi(X_i)\)
  - store \(\Phi(X_i)\) as \(k \to \infty\)?
The dot product implementation

- This turns out to be possible when the learning algorithm can be written in “dot-product” form.

**Definition:** A learning algorithm is in dot product form if, given a training set

\[ D = \{(x_1,y_1), \ldots, (x_n,y_n)\}, \]

it only depends on the points \( X_i \) through their dot products \( X_i^T X_j \).

- We will see that, luckily, this is a natural representation for many optimization procedures.

- The Perceptron learning algorithm can be written in this form quite easily (homework).
Perceptron learning

in dot-product form:

\[ \text{set } \alpha_i = 0, \ b = 0 \]

\[ \text{set } R = \max_i ||x_i|| \]

do {
  
  for i = 1:n {
    if \( y_i \left( \sum_{j=1}^{n} \alpha_j y_j x_j^T x_i + b \right) \leq 0 \) then {
      \[ \alpha_i = \alpha_i + 1 \]
      \[ b = b + y_i R^2 \]
    }
  }
}

} until no errors

Notes:

- in original form we update \( w \in \mathbb{R}^d \)
- in dot-product form we update \( \alpha \in \mathbb{R}^n \)
- note that \( n \) is training set size and \( n >> d \), in general
- at first look does not appear very productive
- the benefits only become visible when we introduce \( \Phi \).
Perceptron learning

- in range space:

  set \( \alpha_i = 0, \ b = 0 \)

  set \( R = \max_i ||\Phi(x_i)|| \)

  do {
      for \( i = 1:n \) {
          if \( y_i \left( \sum_{j=1}^{n} \alpha_j y_j \Phi(x_j)^T \Phi(x_i) + b \right) \leq 0 \) then {
              \( \alpha_i = \alpha_i + 1 \)
              \( b = b + y_i R^2 \)
          }
      }
  }

  } until no errors

- Notes:
  - only requires dot-products \( \Phi(x_j)^T \Phi(x_i) \)
  - no-longer need to store the \( \Phi(x_j) \)
  - only the \( n^2 \) dot-product matrix
  - when \( \text{dim}[\Phi(x_j)] \) is infinite this is significant
  - what about the computation of the dot products?
The “kernel trick”

“instead of defining \( \Phi(x) \), computing \( \Phi(x_i) \) for each \( i \) and \( \Phi(x_i)^T \Phi(x_j) \) for each pair \((i,j)\), simply define the function

\[
K(x, z) = \Phi(x)^T \Phi(z)
\]

and work with it directly.”

- \( K(x,z) \) is called a dot-product kernel
- in fact, since we only use the kernel, why define \( \Phi(x) \)?
- just define the kernel \( K(x,z) \) directly!
- in this way we never have to deal with the complexity of \( \Phi(x) \)...
- this is usually called the “kernel trick”
Questions

- I am confused!

- how do I know that if I pick a function $K(x,z)$, it is equivalent to $\Phi(x)^T \Phi(z)$?
  - in general, it is not. We will talk about this later.

- if it is, how do I know what $\Phi(x)$ is?
  - you may never know. E.g. the Gaussian kernel
    $$K(x, z) = e^{-\|x-z\|^2/\sigma}$$
    is very popular. It is not obvious what $\Phi(x)$ is...
  - on the positive side, we did not know how to choose $\Phi(x)$. Choosing instead $K(x,z)$ makes no difference.

- why is it that using $K(x,z)$ is easier/better?
  - complexity. let’s look at an example.
Polynomial kernels

still in $\mathbb{R}^d$, consider the square of the dot product between two vectors

$$
\left( x^T z \right)^2 = \left( \sum_{i=1}^{d} x_i z_i \right)^2 = \left( \sum_{i=1}^{d} x_i z_i \right) \left( \sum_{j=1}^{d} x_j z_j \right) = \\
= \sum_{i=1}^{d} \sum_{j=1}^{d} x_i x_j z_i z_j \\
= x_1 x_1 z_1 z_1 + x_1 x_2 z_1 z_2 + \ldots + x_1 x_d z_1 z_d + \\
+ x_2 x_1 z_2 z_1 + x_2 x_2 z_2 z_2 + \ldots + x_2 x_d z_2 z_d + \\
\vdots \\
+ x_d x_1 z_d z_1 + x_d x_2 z_d z_2 + \ldots + x_d x_d z_d z_d
$$
Polynomial kernels

- can be written as

\[
\left(x^T z\right)^2 = \left[xx_1x_1, x_1x_2, \ldots, x_1x_d, \ldots, x_dx_1, x_dx_2, \ldots, x_dx_d, \ldots\right] \\
\Phi(x)^T \\
\]

- hence, we have

\[
K(x, z) = \left(x^T z\right)^2 = \Phi(x)^T \Phi(z) \\
\]

with \( \Phi : \mathbb{R}^d \rightarrow \mathbb{R}^{d^2} \)

\[
\begin{pmatrix}
X_1 \\
\vdots \\
X_d
\end{pmatrix} \rightarrow \left(x_1x_1, x_1x_2, \ldots, x_1x_d, \ldots, x_dx_1, x_dx_2, \ldots, x_dx_d\right)^T
\]
Polynomial kernels

► the point is that
  • while $\Phi(x)^T \Phi(z)$ has complexity $O(d^2)$
  • direct computation of $K(x, z) = (x^T z)^2$ has complexity $O(d)$
► direct evaluation is more efficient by a factor of $d$
► as $d$ goes to infinity this makes the idea feasible
► BTW, you just met another kernel family
  • this implements polynomials of second order
  • in general, the family of polynomial kernels is defined as

\[
K(x, z) = \left(1 + x^T z\right)^k, \quad k \in \{1, 2, \ldots\}
\]

• I don’t even want to think about writing down $\Phi(x)$!
Kernel summary

1. $D$ not linearly separable in $\mathcal{X}$, apply feature transformation $\Phi: \mathcal{X} \rightarrow \mathcal{Z}$, such that $\text{dim}(\mathcal{Z}) \gg \text{dim}(\mathcal{X})$

2. computing $\Phi(x)$ too expensive:
   - write your learning algorithm in dot-product form
   - instead of $\Phi(x_i)$, we only need $\Phi(x_i)^T \Phi(x_j) \forall ij$

3. instead of computing $\Phi(x_i)^T \Phi(x_j) \forall ij$, define the “dot-product kernel”

   $$K(x, z) = \Phi(x)^T \Phi(z)$$

   and compute $K(x_i, x_j) \forall ij$ directly
   
   - note: the matrix
     $$K = \begin{bmatrix}
     \vdots \\
     \cdots K(x_i, z_j) \cdots \\
     \vdots 
     \end{bmatrix}$$

     is called the “kernel” or Gram matrix

4. forget about $\Phi(x)$ and use $K(x, z)$ from the start!
Dot-product kernels

- have various nice properties
- **Perceptron example:**
  - using the "kernelized" Perceptron in domain space $\mathcal{X}$
  - is equivalent to using original Perceptron in range space $\mathcal{Z} = \Phi(\mathcal{X})$
  - we have shown that Perceptron learning will converge in a finite number of iterations
  - the proof did not make any assumptions about the space
  - hence, this holds in range space and shows that
- **Perceptron can learn non-linear bounds in finite time!**
Kernelized Perceptron learning

just for completeness, we recall that this is:

\[
\begin{align*}
\text{set } &\alpha_i = 0, \ b = 0 \\
\text{set } &R = \max_i \|\Phi(x_i)\|
\end{align*}
\]

\[
\text{do } \{ \\
\text{for } i = 1:n \{ \\
\quad \text{if } \quad \{ \\
\quad - \quad \alpha_i = \alpha_i + 1 \\
\quad - \quad b = b + y_i R^2 \\
\quad \} \\
\} \\
\} \text{ until no errors}
\]
Kernelized Perceptron learning

**Theorem:** Let $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, $\Phi(x)$ a feature transformation and

$$R = \max_i \|\Phi(x_i)\|$$

If there is $(w^*, b^*)$ such that $\|w^*\| = 1$ and

$$y_i (w^T \Phi(x_i) + b^*) > \gamma, \quad \forall i$$

then the kernelized Perceptron will find an error free hyper-plane in at most

$$\left(\frac{2R}{\gamma}\right)^2$$

iterations

**note that the margin $\gamma$ is now in range space $\mathcal{Z}$**

**this implies that the choice of kernel must matter!**
Question

what is a good dot-product kernel?

• the above result suggests that a good kernel is one that maximizes the margin $\gamma$ in range space
• however, nobody knows how to do this

in practice:

• pick a kernel from a library of known kernels
• we have already met
  • the linear kernel $K(x,z) = x^Tz$
  • the Gaussian family
    $$K(x,z) = \exp\left(-\frac{\|x-z\|^2}{\sigma}\right)$$
  • the polynomial family
    $$K(x,z) = \left(1 + x^Tz\right)^k, \quad k \in \{1, 2, \cdots\}$$
Dot-product kernels

- this may not be a bad idea
  - we rip the benefits of moving to a high-dimensional space
  - without paying a price in complexity
  - the kernel simply adds a few parameters \((\sigma, k)\)
  - learning it would imply introducing many parameters (up to \(n^2\))
- recall that
  - the learning algorithm is still maximizing \(\gamma\) in \(\mathcal{Z}\)
  - it is just that the maximum may not be as large as if the kernel were chosen optimally
  - but the difference may not justify the risk of overfitting
  - anyway, this is an open research question
Question

» does the kernel have to be a dot-product kernel?

» not necessarily. For example, neural networks can be seen as implementing kernels that are not of this type however:

  • you loose the parallelism. what you know about the learning machine may no longer hold after you kernelize
  • most of what we are going to do in the next lectures no longer holds
  • dot-product kernels usually lead to convex learning problems. Usually you loose this guarantee for non dot-product

» i.e. you have to be careful
Question

- how do I know if a function is a dot-product kernel?
- we will start looking at this next class

homework:

- we are going to enter in a slightly more technical stage of the course
- e.g. we will use abstract dot-products $\langle x, y \rangle$ instead of $\sum x_i z_i$
- you may want to revise some core concepts
- for next class, try to revise the definitions of
  - vector space, dot product, bilinear form, norm