Lecture 3

Kernels, Beginning of neural networks

Reading

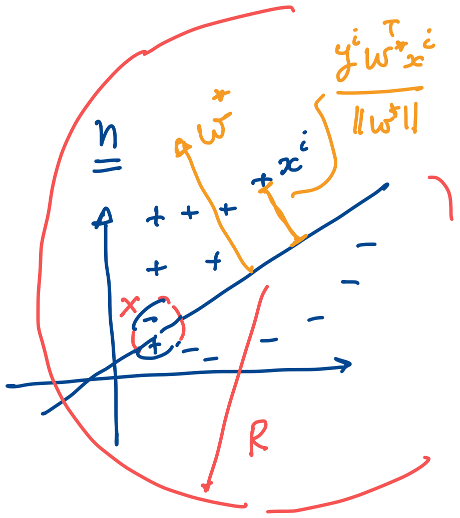
1. Bishop 6.1-6.3
2. Goodfellow 6.1-6.4
3. "Random features for large-scale kernel machines" by Rahimi and Recht (2008).

3.1 Digging deeper into the perceptron

3.1.1 Convergence rate

How many iterations does a perceptron need to fit on a given dataset? We will assume that the training data are bounded, i.e., $\|x^i\| \leq R$ for some R and for all $i \in \{1, \ldots, n\}$. Let us also assume that the training dataset is indeed linearly separable, i.e., a solution w^* exists for the perceptron weights with training error exactly zero. This means

$$y^i {w^*}^\top x^i > 0 \quad \forall i.$$



We will also assume that this classifier *separates the data well*. Note that the distance of each input x^i from the decision boundary (i.e., all x such that ${w^*}^\top x = 0$) is given by the component of x^i in the direction of w^* if the label is $y^* = +1$ and in the direction $-w^*$ if the label is negative. In other words,

$$\frac{y^i {w^*}^\top x^i}{\|w^*\|} = \rho^i \qquad \text{(Margin)}$$

gives the distance to the decision boundary. The quantity on the right hand side is called the *margin*, it is simply the distance of the sample i from the decision boundary. If w^* is the classifier with the largest average margin,

$$\rho = \min_{i \in \{1,\ldots,n\}} \rho^i$$

is a good measure of how hard a particular machine learning problem is.

You can now try to prove that after each update of the perceptron the inner product of the current weights with the try solution $\langle w_t, w^* \rangle$ increases at least linearly and that the squared norm $\|w_t\|^2$ increases at most linearly in the number of updates t. Together the two will give you a result that after t weight updates

$$t \leq \frac{R^2}{\rho^2} \tag{3.1}$$

all training data are classified correctly. Notice a few things about this expression.

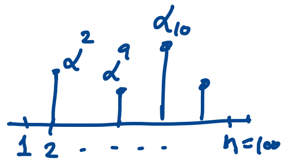
1. The quantity $\frac{R^2}{\rho^2}$ is dimension independent; that the number of steps reach a given accuracy is independent of the dimension of the data will be a property shared by optimization algorithms in general.

2. There are no constant factors, this is also the worst case number of updates; this is rare.

3. The number of updates scales with the hardness of the problem; if the margin ρ was small, we need lots of updates to drive the training error to zero.

3.1.2 Dual representation

Let us see how the parameters of the perceptron look after training on the entire dataset. At each iteration, the weights are updated in the direction (x^t, y^t) or they are not updated at all. Therefore, if α^i is the number of times the perceptron sampled the datum (x^i, y^i) during the course of its training and got it wrong, we can write the weights of the perceptron as a linear combination



$t \in \{1, \ldots \infty\}$

$t = 200$

$$w = \sum_{i=1}^{n} \alpha^i y^i x^i. \tag{3.2}$$

where $\alpha^i \in \{0, 1, \ldots, \}$. The perceptron therefore using the classifier

$$\begin{aligned} f(x, w) &= \text{sign}(\hat{y}) \\ \text{where} \quad \hat{y} &= \left(\sum_{i=1}^{n} \alpha^i y^i x^i \right)^\top x \\ &= \sum_{i=1}^{n} \alpha^i y^i {x^i}^\top x. \end{aligned} \tag{3.3}$$

⚠ As you see in (3.3), computing the prediction for a new input x involves, either remembering all the weights w at the end of training, or storing all the $\{\alpha^i\}_{i=1,\ldots,n}$ along with the training dataset. The latter is called the dual representation of a perceptron and the scalars $\{\alpha^i\}$ are called the dual parameters.

Remember this special form: the inner product of the new input x with all the other inputs x^i in the training dataset is combined linearly to get the prediction. The weights of this linear combination are the dual variables which is a measure of how many tries it took the perceptron to fit that sample during training.

3.2 Creating nonlinear classifiers from linear ones

Linear classifiers such as the perceptron, or the support vector machine (SVM) can be extended to nonlinear ones. The trick is essentially the same that we saw when we fit polynomials (polynomials are nonlinear) using the formula for linear regression. We are interested in mapping input data x to some different space, this is usually a higher-dimensional space called the *feature space*.

05/12/16,

$$x \mapsto \phi(x).$$

$x \in \mathbb{R}$

$\phi(x) = z = [1, \sqrt{2}x, x^2]$

प्रतिक

प्रतिक

चौधरी

The quantity $\phi(x)$ is called a feature vector.

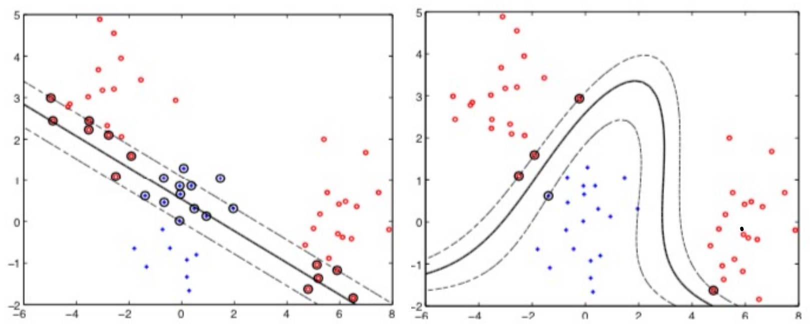
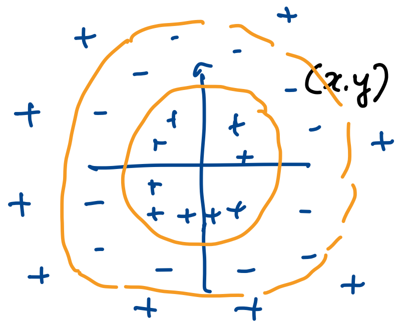


Figure 3.1



$z = [x^2+y^2]$

$z = [\mathbb{1}_{\{x^2+y^2 \leq 1\}}, \mathbb{1}_{\{x^2+y^2 \geq 3\}}]$

For example, in the polynomial regression case for scalar input data $x \in \mathbb{R}$ we used

$$\phi(x) := \left[1, \sqrt{2}x, x^2\right]^\top$$

to get a quadratic feature space. The role of $\sqrt{2}$ will become clear shortly. Certainly this trick of polynomial expansion also works for higher dimensional input

$x \in \mathbb{R}^2$

$$\phi(x) := \left[1, x_1, x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2\right]^\top.$$

Having fixed a feature vector $\phi(x)$, we can now fit a linear perceptron on the input data $\{\phi(x^i), y^i\}$. This involves updating the weights at each iteration as

$$w_{t+1} = \begin{cases} w_t + y^t\phi(x^t) & \text{if } \operatorname{sign}(w_t^\top \phi(x^t)) \neq y^t \\ w_t & \text{else.} \end{cases} \tag{3.4}$$

At the end of such training, the perceptron is

$$w = \sum_{i=1}^n \alpha^i y^i \phi(x^i)$$

and predictions are made by first mapping the new input to our feature space

$$f(x; w) = \operatorname{sign}\left(\sum_{i=1}^n \alpha^i y^i \phi(x^i)^\top \phi(x)\right). \tag{3.5}$$

Notice that we now have a linear combination of the *features* not the data directly.

? The concept of a feature space seems like a panacea. If we have complex data, we simply map it to some high-dimensional feature and fit a linear function to these features. However, the "curse of dimensionality" coined by Richard Bellman states that to fit a function in $\mathbb{R}^d$ the number of data needs to be exponential in d. It therefore stands to reason that we need a lot more data to fit a classifier in feature space than in the original input space. Why would we still be interested in the feature space then?

3.3 Kernels

Observe the expression of the classifier in (3.5). Each time we make predictions on the new input, we need to compute n inner products of the form

$$\phi(x^i)^\top \phi(x).$$

If the feature dimension is high, we need to enumerate the large number of feature dimensions if we are using the weights of the perceptron, or these inner products if we are using the dual variables. Observe however that even if the feature vector is large, we can compactly evaluate the inner product

⚠ Feature spaces can become large very quickly. What is the dimensionality of $\phi(x)$ for a tenth-order polynomial with a three-dimensional input data?

$$\begin{aligned}\phi(x) &= \left[1, \sqrt{2}x, x^2\right] \checkmark\\ \phi(x') &= \left[1, \sqrt{2}x', x'^2\right]\\ \phi(x)^\top \phi(x') &= 1 + 2xx' + (xx')^2 = (1 + xx')^2 .\end{aligned}$$

for input $x \in \mathbb{R}$. Kernels are a formalization of exactly this idea. A kernel

$$k : X \times X \to \mathbb{R}.$$

is a symmetric, positive semi-definite function of its two arguments for which it holds that

$$k(x, x') = \phi(x)^\top \phi(x)$$

for some feature ϕ. Few examples of kernels are

$$\begin{aligned}k(x, x') &= \left(x^\top x' + c\right)^2,\\ k(x, x') &= \underline{\underline{\exp\left(-\|x - x'\|^2/(2\sigma^2)\right)}} . = 1 + \frac{-\|x-x'\|^2}{2\sigma} + \frac{+\|x-x'\|^4}{2(2\sigma)^2} + \vdots\end{aligned}$$

3.3.1 Kernel perceptron

We can now give the kernel version of the perceptron algorithm. The idea is to simply replace any inner product in the algorithm that looks like $\phi(x)^\top \phi(x')$ by the kernel $k(x, x')$.

Kernel perceptron

Initialize dual variables $\alpha^i = 0$ for all $i \in \{1, \ldots, n\}$. Perform the following steps for iterations $t = 1, 2, \ldots$.

1. At the tth iteration, sample a data point from D_{train} uniformly randomly, call it (x^t, y^t).
2. If there is a mistake, i.e., if

$$\begin{aligned}0 &\geq y^t \left(\sum_{i=1}^n \alpha^i y^i \phi(x^i)^\top \phi(x^t)\right)\\ &= y^t \left(\sum_{i=1}^n \alpha^i y^i k(x^i, x^t)\right),\end{aligned}$$

then update

$$\alpha^t \leftarrow \alpha^t + 1.$$

Notice that we do not ever compute $\phi(x)$ so it does not matter what the dimensionality of the feature vector is. It can even be infinite, e.g., for the radial basis function kernel.

Note that the kernel perceptron computes the kernel over *all* data samples in the training set at each iteration. It is expensive and seems wasteful. The Gram matrix denoted by $G \in \mathbb{R}^{n \times n}$

$$G_{ij} = k(x^i, \underset{x}{\cancel{k}^j}) \tag{3.6}$$

helps address this problem by computing the kernel on all pairs in the training dataset. With this in hand, we can modify step 2 in the kernel perceptron using

$$y^t \left(\underbrace{\sum_{i=1}^n \alpha^i y^i k(x^i, x^t)}_{\in \mathbb{R}^n} \right) = \overset{y^t \beta^\top G e_t}{\cancel{y^t \alpha^\top \underset{\in \mathbb{R}^{n\times n}}{G}}} e_t. \qquad \beta = \begin{bmatrix} \alpha^1 y^1 \\ \vdots \\ \alpha^n y^n \end{bmatrix}$$

where $e_t = [0, \ldots, 0, 1, 0, \ldots]$ with a 1 on the tth element and $\alpha = [\alpha^1, \ldots, \alpha^n]$ denotes the vector of all the dual variables. This expression now only involves a matrix-vector multiplication, which is much easier than computing the kernel at each iteration. Gram matrices can become very big. If the number of samples is $n = 10^6$, not an unusual number today, the Gram matrix has 10^{12} elements. The big failing of kernel methods is that they require a large amount of memory at training time. Nystrom methods compute low-rank approximations of the Gram matrix which makes operations with kernels easier.

❓ Kernels look great, you can fit perceptrons in powerful feature spaces using essentially the same algorithm. How expensive is each iteration of the perceptron?

⚠ When ML algorithms are implemented in a system, there exist tradeoffs between the feature-space version and the Gram matrix version of linear classifiers. The former is preferable if the number of samples in the dataset is large, while the latter is used when the dimensionality of features is large.

3.3.2 Mercer's theorem

This theorem shows that given any kernel that satisfies some regularity properties can be rewritten as an inner product.

Theorem 1 (Mercer's Theorem). For any symmetric function $k : X \times X \to \mathbb{R}$ which is square integrable in $X \times X$ and satisfies

$$\int_{X \times X} k(x, x') \, f(x) \, f(x') \, \mathrm{d}x \, \mathrm{d}x' \geq 0 \tag{3.7}$$

for all square integrable functions $f \in L_2(X)$, there exist functions $\phi_i : X \to \mathbb{R}$ and numbers $\lambda_i \geq 0$ where

$$k(x, x') = \sum_i \lambda_i \phi_i(x) \, \phi_i(x')$$

for all $x, x' \in X$. The condition in (3.7) is called Mercer's condition. You will also have seen it written as *for any finite set of inputs* $\{x^1, \ldots, x^n\}$ *and any choice of real-valued coefficients* $c_1, \ldots, c_n$ *a valid kernel should satisfy*

$$\sum_{i,j} c_i c_j k(x^i, x^j) \geq 0.$$

There can be an infinite number of coefficients λ_i in the summation.

❓ Logistic regression with a loss function

$$\ell_{\text{logistic}}(w) = \log \left(1 + e^{-y w^\top x} \right)$$

is also a linear classifier. Write down how you will fit a logistic regression using stochastic gradient descent; this is similar to the perceptron algorithm. Write down the feature-space version of the algorithm and a kernelized logistic regression that uses the Gram matrix.

💡 A function $f : X \to \mathbb{R}$ is square integrable iff

$$\int_{x \in X} |f(x)|^2 \, \mathrm{d}x < \infty.$$

Remark 2 (Checking if a function is a valid kernel). Note that Mercer's condition states that the Gram matrix of any dataset is positive semi-definite:

$$u^\top G u \geq 0 \quad \text{for all } u \in \mathbb{R}^n.$$

This is easy to show.

$$\begin{aligned}
u^\top G u &= \sum_{ij} u_i u_j G_{ij} \\
&= \sum_{ij} u_i u_j \phi(x_i)^\top \phi(x_j) \\
&= \left(\sum_i u_i \phi(x_i)\right)^\top \left(\sum_j u_j \phi(x_j)\right) \\
&= \|\sum_i u_i \phi(x_i)\|^2 \\
&\geq 0.
\end{aligned}$$

The integral in Theorem 1 in Mercer's condition is really just the continuous analogue of the vector-matrix-vector multiplication above. So if you have a function that you would like to use as a kernel, checking its validity is easy by showing that the Gram matrix is positive semi-definite.

Kernels are powerful because they do not require you to think of the feature and parameter spaces. For instance, we may wish to design a machine learning algorithm for spam detection that takes in a variable length of feature vector depending on the particular input. If $x[i]$ is the i^{th} character of a string, a good feature vector to use is to consider the set of all length k subsequences. The number of components in this feature vector is exponential. However, as you can imagine, given two strings x, x'

```
this string is interesting
txws sbhtqg is iyubqtnhpqg
```

you can write a Python function to check their similarities with respect to some rules *you define*. Mercer's theorem is useful here because it says that so long as your function satisfies the basic properties of a kernel function, there exists some feature space which your function implicitly constructs.

3.4 Learning the feature vector

3.5 Deep feed-forward networks

Bibliography

Rahimi, A. and Recht, B. (2008). Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184.