Lecture 18: Kernels and Feature Extraction

CS4787/5777 — Principles of Large-Scale Machine Learning Systems

Deep neural networks are a good way to learn arbitrary functions...but they're not the only way. Another way to do this that is more amenable to theoretical analys and that we'll be using over the next few weeks is to use **kernels**. Recall that a kernel (also sometimes called a positive definite kernel) over a set $\mathcal{X}$ is a symmetric function $K(x, y) : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ that maps pairs of objects in $\mathcal{X}$ onto the real numbers. By *symmetric*, we mean that $K(x, y) = K(y, x)$ for all x and y in $\mathcal{X}$. Note that $\mathcal{X}$ does not have to be a vector space: it can be any set! To be a kernel, K must satisfy the condition that for any $x_1, x_2, \ldots, x_n \in \mathcal{X}$, and for any scalars $c_1, c_2, \ldots, c_n \in \mathbb{R}$

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K(x_i, x_j) \geq 0.$$

This is equivalent to saying that if we define the matrix $\mathbf{K} \in \mathbb{R}^{n \times n}$ by

$$\mathbf{K}_{i,j} = K(x_i, x_j),$$

then K will be positive semidefinite matrix.[1] I.e. for any vector $c \in \mathbb{R}^n$, $c^T \mathbf{K} c \geq 0$. This is also equivalent to saying that any eigenvalue of $\mathbf{K}$ must be nonnegative.

Usually $K(x, y)$ represents how similar x and y are, where more similar objects will have larger values of $K(x, y)$ and less similar objects will have smaller values. One very popular kernel is the **radial basis function** kernel or RBF kernel, sometimes also called the Gaussian kernel. The RBF kernel is over a Euclidean space $\mathcal{X} = \mathbb{R}^d$ and takes the form

$$K(x, y) = \exp\left(-\gamma \|x - y\|^2\right).$$

It's easy to see that $0 < K(x, y) \leq 1$ and $K(x, y) = 1$ if and only if $x = y$. These two properties informally mean that this kernel is expressing a similarity between x and y that ranges between 0 and 1, where 1 is the most similar (i.e. literally identical objects $x = y$).

Question: What other kernels have you seen or used in your work?

A kernel determines a feature map. Important property of kernels: for every kernel over a set $\mathcal{X}$, there exists a real Hilbert space[2] $\mathcal{H}$ and a feature map $\phi : \mathcal{X} \to \mathcal{H}$ such that for any $x, y \in \mathcal{X}$,

$$K(x, y) = \phi(x)^T \phi(y) = \langle \phi(x), \phi(y) \rangle.$$

[1]Occasionally you will see authors distinguish between positve definite kernels and positive semidefinite kernels, which correspond to positive definite and positive semidefinite matrices respectively in this condition. But for this class and for most practical machine learning we won't depend on this distinction.

[2]A Hilbert space is just a potentially infinite-dimensional generalization of Euclidean space that supports a dot product/inner product operation with the same properties as the Euclidean dot product.

That is, every kernel is just a dot product in a (possibly infinite-dimensional) transformed space. This fact also holds in reverse: every feature map determines a kernel.

Constructing kernels. We can construct kernels from other kernels. Given any kernels K_1 and K_2 over a set $\mathcal{X}$ with corresponding features maps ϕ_1 and ϕ_2 mapping onto sets of dimension D_1 and D_2 respectively, any positive semi-definite matrix A, any scalar $c \geq 0$, and any function $f : \mathcal{X} \to \mathbb{R}$, the following are kernels.

Kernel	**Corresponding Feature Map**	**Dimension of ϕ**
$K(x, y) = x^T A y$	$\phi(x) = \sqrt{A} \cdot x$	d (if $\mathcal{X} = \mathbb{R}^d$)
$K(x, y) = c \cdot K_1(x, y)$	$\phi(x) = \sqrt{c} \cdot \phi_1(x)$	D_1
$K(x, y) = K_1(x, y) + K_2(x, y)$	$\phi(x) = \begin{bmatrix} \phi_1(x) \\ \phi_2(x) \end{bmatrix}$	$D_1 + D_2$
$K(x, y) = K_1(x, y) \cdot K_2(x, y)$	$\phi(x) = \phi_1(x) \otimes \phi_2(x)$	$D_1 \cdot D_2$
$K(x, y) = f(x) \cdot K_1(x, y) \cdot f(y)$	$\phi(x) = \phi_1(x) \cdot f(x)$	D_1
$K(x, y) = \exp(K_1(x, y))$	—[3]	∞

These rules let us easily construct a wide range of highly interpretive kernels. But, as a trade-off, the dimension of the feature map ϕ **quickly becomes very large** as we apply the rules. It can even become infinite! This can make computing directly with the mapped features costly or even impossible.

This property motivates the technique called **the kernel trick** in which we can "kernelize" a linear model to efficiently learn a model over some transformed features. Concretely, suppose that I want to train a simple linear model with parameter w and empirical risk

$$f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w) = \frac{1}{n} \sum_{i=1}^{n} L(w^T \phi(x_i); y_i)$$

where L is some loss function, and x_i, y_i are our training examples, and ϕ is some feature map. By the chain rule, the gradient of a single example i with respect to w is

$$\nabla f_i(w) = L'(w^T \phi(x_i); y_i) \cdot \phi(x_i)$$

so our updates in SGD will all be of the form

$$w_{t+1} = w_t - \alpha_t \cdot L'(w^T \phi(x_i); y_i) \cdot \phi(x_i).$$

If we initialize at $w_0 = 0$, we can show inductively that the parameters w_t are always in the span[4] of the transformed features, i.e.

$$w_t \in \operatorname{span}\{\phi(x_1), \phi(x_2), \ldots, \phi(x_n)\}$$

or, equivalently, that there exists some scalars $u_{1,t}, u_{2,t}, \ldots, u_{n,t}$ such that

$$w_t = \sum_{i=1}^{n} u_{i,t} \cdot \phi(x_i).$$

[3]Omitted for space.

[4]Recall that the span of some vectors in a vector space is defined as the linear subspace consisting of vectors that can result from any linear combination of vectors in the set

$$\operatorname{span}\{z_1, z_2, \ldots, z_n\} = \{a_1 z_1 + a_2 z_2 + \cdots + a_n z_n | a_i \in \mathbb{R}\}.$$

We can write the SGD update in terms of these scalars $u_{i,t}$ explicitly as follows. At each iteration, we choose a random index i and then update

$$\begin{aligned} u_{i,t+1} &= u_{i,t} - \alpha_t \cdot L' \left(\left(\sum_{j=1}^n u_{j,t} \cdot \phi(x_j) \right)^T \phi(x_i); y_i \right) \\ &= u_{i,t} - \alpha_t \cdot L' \left(\sum_{j=1}^n u_{j,t} \cdot \phi(x_j)^T \phi(x_i); y_i \right) \\ &= u_{i,t} - \alpha_t \cdot L' \left(\sum_{j=1}^n u_{j,t} \cdot K(x_j, x_i); y_i \right). \end{aligned}$$

Four simple ways to compute this:

① Transform the features on the fly and compute SGD on w.

② Pre-compute and cache the transformed features, forming vectors $z_i = \phi(x_i)$, and compute SGD on w.

③ Run SGD on u and compute the kernel values $K(x_j, x_i)$ on the fly as they are needed.

④ Run SGD on u and pre-compute the kernel values for each pair of examples, forming the **Gram matrix $\mathbf{K}$** where $\mathbf{K}_{i,j} = K(x_i, x_j)$, store it in memory, and then use this during training by running

$$u_{i,t+1} = u_{i,t} - \alpha_t \cdot L' \left(\sum_{j=1}^n u_{j,t} \cdot \mathbf{K}_{j,i}; y_i \right) = u_{i,t} - \alpha_t \cdot L' \left(\vec{u}_t^T K e_i; y_i \right),$$

where $\vec{u}_t$ denotes the vector of $u_{1,t}, u_{2,t}, \ldots, u_{n,t}$, and e_i is the ith normal basis vector.

All of these methods will result in the same trained model at the end (assuming no numerical imprecision). **But how does their computational cost compare?**

Activity. Suppose that the original examples are $x_i \in \mathbb{R}^d$, the transformed features are $\phi(x_i) \in \mathbb{R}^D$, there are n total training examples, and we run T iterations of SGD. Also suppose that the cost of computing a single feature map $\phi(x_i)$ is $\Theta(d \cdot D)$ and the cost of computing the kernel function $K(x_i, y_i)$ is $\Theta(d)$. What are the computational cost and memory required for these four methods, up to a big-Θ analysis? Remember to include the cost of precomputation!

Takeaway: This analysis shows there is a trade-off between the four different methods for kernel learning. No one method is better than the others in all cases, and you should reason about how your learning task scales before deciding which method to use.

One problematic case: what if we want to use a kernel with an infinite-dimensional feature map? For example: the RBF kernel. Then $D = \infty$ in the analysis above, and methods (1) and (2) become impossible to run. On the other hand, methods (1) and (2) had the best dependence on n, and we would want to consider running one of these when the training set size becomes large. **How can we handle the case of large n and large/infinite D?**

Approximate feature maps. Also called "feature extraction" sometimes. Idea: approximate a possibly infinite-dimensional feature map ϕ of a kernel with a finite-dimensional feature map $\psi : \mathcal{X} \to \mathbb{R}^D$ such that for all $x, y \in \mathcal{X}$

$$K(x, y) = \phi(x)^T \phi(y) \approx \psi(x)^T \psi(y).$$

Then we can use the approximate feature map ψ to learn with strategy (1) or (2) above.

Question: What are we trading off when we do this?

How do we construct these approximate feature maps? There are many ways to do it. Here, I'm going to be talking about one way we can do this for the RBF kernel, a method called *Random Fourier features*.[5] The strategy is straightforward (although the math can get a little tricky).

- First, we write the kernel $K(x, y)$ in terms of an expected value.
- Second, we use our *subsampling principle* to approximate this expected value with a finite sample to within the desired level of accuracy .
- Third, we use this finite sample to construct an approximate feature map.

If we sample ω from a d-dimensional multivariate Gaussian distribution with mean 0 and covariance $2\gamma I$, and independently sample b to be uniform on $[0, 2\pi]$, then we can show that for any $x, y \in \mathbb{R}^d$,

$$K(x, y) = \exp\left(-\gamma \|x - y\|^2\right) = \mathbf{E}_{\omega,b}\left[2 \cdot \cos(\omega^T x + b) \cdot \cos(\omega^T y + b)\right].$$

Now we can apply our subsampling principle to approximate this expected value with a finite sum. Pick some number D and let $\omega_1, b_1, \omega_2, b_2, \ldots, \omega_D, b_D$ be independent random samples of ω and b. Then

$$\begin{aligned} K(x, y) &\approx \frac{1}{D} \sum_{i=1}^{D} 2 \cdot \cos(\omega_i^T x + b_i) \cdot \cos(\omega_i^T y + b_i) \\ &= \sum_{i=1}^{D} \left(\sqrt{\frac{2}{D}} \cdot \cos(\omega_i^T x + b_i) \right) \cdot \left(\sqrt{\frac{2}{D}} \cdot \cos(\omega_i^T y + b_i) \right). \end{aligned}$$

So if we define the feature map $\psi(x)$ such that its ith element is

$$(\psi(x))_i = \sqrt{\frac{2}{D}} \cdot \cos(\omega_i^T x + b_i),$$

then

$$K(x, y) \approx \sum_{i=1}^{D} (\psi(x))_i (\psi(y))_i = \psi(x)^T \psi(y)$$

and we have our approximate feature map! Note that we can also write ψ in terms of matrix multiply as

$$\psi(x) = \sqrt{\frac{2}{D}} \cdot \cos(\Omega x + \mathbf{b}),$$

where Ω and $\mathbf{b}$ denote the matrix and vector

$$\Omega = \begin{bmatrix} \omega_1^T \\ \vdots \\ \omega_D^T \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_D \end{bmatrix}$$

[5] Ali Rahimi and Ben Recht, "Random Features for Large-Scale Kernel Machines." Advances in neural information processing systems, 2008. This paper won the test of time award at NeurIPS 2017.

and the $\cos$ operates elementwise. **What thing that we've already seen does this look like?** It looks like a neural network with random weights and a cosine nonlinearity!

How close will this be to the exact RBF kernel? One way to get a sense of it is to use Hoeffding's inequality. Since the elements of this sum are all of the form $2\cos(\cdot)\cos(\cdot)$, they must have magnitude no greater than 2. As a result, $z_{\min} = -2$, $z_{\max} = 2$, and we get

$$\mathbf{P}\left(\left|K(x,y) - \psi(x)^T\psi(y)\right| \geq a\right) \leq 2\exp\left(-\frac{2Da^2}{(2-(-2))^2}\right) = 2\exp\left(-\frac{Da^2}{8}\right).$$

Takeaway: We can get arbitrarily good accuracy with arbitrarily high probability by increasing D.[6]

This is just one example of a broad class of techniques that can be used for feature extraction for kernels. This adds the following extra techniques to our list of ways to do kernel learning:

(5) Pre-compute an approximate feature map (e.g. sample Ω and $\mathbf{b}$ for random Fourier features on RBF) and use it to compute approximate features on the fly to compute SGD.

(6) Pre-compute an approximate feature map, then also pre-compute cache the transformed approximate features, forming vectors $z_i = \psi(x_i)$. Then run SGD.

Often, using an approximate feature map like this is the most efficient way of training a model. But...not always! When learning with a kernel, we need to **keep the properties of the problem in mind** to decide how to proceed with learning most efficiently.

For the curious, a proof of the fact that in one dimension

$$\exp\left(-\gamma(x-y)^2\right) = \mathbf{E}_{\omega,b}\left[2\cdot\cos(\omega x + b)\cdot\cos(\omega y + b)\right].$$

I don't expect you to know this proof. It's just here for completeness, in case you're curious, or would like to develop more intuition about this result. You can find a more general derivation in Rahimi and Recht's paper, but their proof depends on Fourier analysis, which I expect many people may be unfamiliar with. So I've written out a proof here that does not use Fourier analysis.

We start by deriving an integral. Define the function

$$g(z) = \int_{-\infty}^{\infty} e^{-\frac{\omega^2}{4\gamma}} \cdot \cos(z\omega)\, d\omega.$$

The derivative of this function is

$$g'(z) = -\int_{-\infty}^{\infty} e^{-\frac{\omega^2}{4\gamma}} \cdot \omega \cdot \sin(z\omega)\, d\omega.$$

Now integrating by parts, we have

$$\int \omega \cdot e^{-\frac{\omega^2}{4\gamma}}\, d\omega = -2\gamma e^{-\frac{\omega^2}{4\gamma}} + C \qquad \text{and} \qquad \frac{d}{d\omega}\sin(z\omega) = z\cos(z\omega).$$

for some constant C. So,

$$\begin{aligned} g'(z) &= -\left[-2\gamma e^{-\frac{\omega^2}{4\gamma}} \cdot \sin(z\omega)\right]_{-\infty}^{\infty} + \int_{-\infty}^{\infty} -2\gamma e^{-\frac{\omega^2}{4\gamma}} \cdot z\cos(z\omega)\, d\omega \\ &= (0-0) - 2\gamma z g(z). \end{aligned}$$

[6]In their paper, Rahimi and Recht show how they can make a similar statement that holds for all pairs x, y in a bounded region, as opposed to just for a specific pair.

It follows that

$$\frac{g'(z)}{g(z)} = -2\gamma z.$$

Integrating both sides gives us

$$\log(g(z)) = -\gamma z^2 + C \qquad \Rightarrow \qquad g(z) = \exp(-\gamma z^2) \cdot e^C$$

for some (possibly different) constant C. What is this constant C? Well, $g(0)$ is

$$g(0) = \int_{-\infty}^{\infty} e^{-\frac{\omega^2}{4\gamma}} \, d\omega = \sqrt{4\pi\gamma},$$

so it follows that

$$g(z) = \int_{-\infty}^{\infty} e^{-\frac{\omega^2}{4\gamma}} \cdot \cos(z\omega) \, d\omega = \sqrt{4\pi\gamma} \cdot \exp(-\gamma z^2).$$

From here, if ω is Gaussian-distributed with mean 0 and variance 2γ, by the definition of expected value

$$\mathbf{E}_\omega \left[\cos(z\omega)\right] = \frac{1}{4\pi\gamma} \int_{-\infty}^{\infty} e^{-\frac{\omega^2}{4\gamma}} \cdot \cos(z\omega) \, d\omega = \exp(-\gamma z^2).$$

Next, note that if b is uniformly distributed in $[0, 2\pi]$, it will hold for any x and y that

$$\begin{aligned}
\mathbf{E}_b \left[2\cos(x+b)\cos(y+b)\right] &= \mathbf{E}_b \left[\cos((x+b)-(y+b)) + \cos((x+b)+(y+b))\right] \\
&= \mathbf{E}_b \left[\cos(x-y) + \cos(x+y+2b)\right] \\
&= \cos(x-y) + \mathbf{E}_b \left[\cos(x+y+2b)\right] = \cos(x-y),
\end{aligned}$$

where this last expectation is 0 by symmetry of the cosine function over its period. It follows that, if we substitute $z = x - y$,

$$\mathbf{E}_{\omega,b} \left[2\cos(\omega x + b)\cos(\omega y + b)\right] = \mathbf{E}_\omega \left[\cos(\omega(x-y))\right] = \exp(-\gamma(x-y)^2) = K(x,y).$$

This shows that the result holds in one dimension. The proof that this works in higher dimensions is similar, and is left as an exercise to the reader (if you wish to attempt it — although I'm certainly not going to expect you to know it in this course).