CSCI-567: Machine Learning (Fall 2019)

Prof. Victor Adamchik

U of Southern California

Oct. 01, 2019

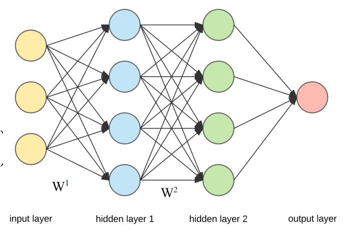
Outline

Outline

Math formulation

An L-layer neural net can be written as

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{h}_\mathsf{L}\left(\boldsymbol{W}^L \boldsymbol{h}_{\mathsf{L}-1}\left(\boldsymbol{W}^{L-1} \cdots \boldsymbol{h}_1\left(\boldsymbol{W}^1 \boldsymbol{x}\right)\right)\right)$$



To ease notation, for a given input $\boldsymbol{x}$, define recursively

$$\boldsymbol{o}^0 = \boldsymbol{x}, \qquad \boldsymbol{a}^\ell = \boldsymbol{W}^\ell \boldsymbol{o}^{\ell-1}, \qquad \boldsymbol{o}^\ell = \boldsymbol{h}_\ell(\boldsymbol{a}^\ell) \qquad (\ell = 1, \ldots, \mathsf{L})$$

where

- $\mathsf{D}_0 = \mathsf{D}, \mathsf{D}_1, \ldots, \mathsf{D}_\mathsf{L}$ are numbers of neurons at each layer
- $\boldsymbol{W}^\ell \in \mathbb{R}^{\mathsf{D}_\ell \times \mathsf{D}_{\ell-1}}$ is the weights for layer ℓ
- $\boldsymbol{a}^\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is input to layer ℓ
- $\boldsymbol{o}^\ell \in \mathbb{R}^{\mathsf{D}_\ell}$ is output to layer ℓ
- $\boldsymbol{h}_\ell : \mathbb{R}^{\mathsf{D}_\ell} \to \mathbb{R}^{\mathsf{D}_\ell}$ is activation functions at layer ℓ

Putting everything into SGD

The backpropagation algorithm (1986) computes the gradient of the cost function for a single training example.

Initialize $\boldsymbol{W}^1, \ldots, \boldsymbol{W}^\mathsf{L}$ (all $\mathbf{0}$ or randomly). Repeat:

1. randomly pick one data point $n \in [\mathsf{N}]$
2. **forward propagation**: for each layer $\ell = 1, \ldots, \mathsf{L}$
   - compute $\boldsymbol{a}^\ell = \boldsymbol{W}^\ell \boldsymbol{o}^{\ell-1}$ and $\boldsymbol{o}^\ell = \boldsymbol{h}_\ell(\boldsymbol{a}^\ell)$ $\qquad (\boldsymbol{o}^0 = \boldsymbol{x}_n)$
3. **backward propagation**: for each $\ell = L, \ldots, 1$
   - compute $$\frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}^\ell}$$
   - update weights $$\boldsymbol{W}^\ell \leftarrow \boldsymbol{W}^\ell - \lambda \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{W}^\ell} = \boldsymbol{W}^\ell - \lambda \frac{\partial \mathcal{E}_n}{\partial \boldsymbol{a}^\ell} (\boldsymbol{o}^{\ell-1})^{\mathrm{T}}$$

Overfitting

Overfitting is very likely since the models are too powerful.

Methods to overcome overfitting:

- data augmentation
- regularization
- dropout
- early stopping
- ...

Conclusions for neural nets

Deep neural networks

- are hugely popular, achieving *best performance* on many problems
- do need *a lot of data* to work well
- take *a lot of time* to train (need GPUs for massive parallel computing)
- take some work to select architecture and hyperparameters
- are still not well understood in theory

Outline

Outline

Motivation

Recall the question: *how to choose nonlinear basis $\phi : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{\mathsf{M}}$?*

$$\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})$$

- neural network is one approach: learn ϕ from data
- **kernel method** is another one: sidestep the issue of choosing ϕ by using *kernel functions*

Case study: regularized linear regression

Recall the regularized least square solution:

$$\boldsymbol{w}^* = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}$$
$$\boldsymbol{w}^* = \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{y}$$

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1^{\mathrm{T}} \\ \boldsymbol{x}_2^{\mathrm{T}} \\ \vdots \\ \boldsymbol{x}_{\mathsf{N}}^{\mathrm{T}} \end{pmatrix}, \boldsymbol{\Phi} = \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}} \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}} \\ \vdots \\ \boldsymbol{\phi}(\boldsymbol{x}_{\mathsf{N}})^{\mathrm{T}} \end{pmatrix}$$

Here $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ is $\mathsf{D} \times \mathsf{D}$ matrix, and $\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$ is $\mathsf{M} \times \mathsf{M}$ matrix,

Issue: *M could be huge or even infinity!*

We will rewrite the solution in a different form.

Another solution

Another minimizer is (we will prove it later on slide 16)

$$\boldsymbol{w}^* = \boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$
$$\boldsymbol{w}^* = \boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{K} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$
$$\boldsymbol{w}^* = \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} = \sum_{n=1}^{N}\alpha_n\boldsymbol{\phi}(\boldsymbol{x}_n)$$

where $\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} \in \mathbb{R}^{\mathsf{N}\times\mathsf{N}}$ is the Gram/Kernel matrix

and $\boldsymbol{\alpha}$ is a new vector.

Solution $\boldsymbol{w}^*$ is a linear combination of features!

Gram matrix

We call $\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$ **Gram matrix** or **kernel matrix** where the (i, j) entry is

$$\boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_j)$$

Therefore,

$$\begin{aligned}\boldsymbol{K} &= \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} \\ &= \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \\ \cdots & \cdots & \cdots & \cdots \\ \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_2) & \cdots & \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_N) \end{pmatrix} \in \mathbb{R}^{\mathrm{N}\times\mathrm{N}}\end{aligned}$$

Examples of kernel matrix

3 data points in $\mathbb{R}$

$$x_1 = -1, x_2 = 0, x_3 = 1$$

$\boldsymbol{\phi}$ is polynomial basis with degree 4:

$$\boldsymbol{\phi}(x) = \begin{pmatrix} 1 \\ x \\ x^2 \\ x^3 \end{pmatrix}$$

$$\boldsymbol{\phi}(x_1) = \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix} \quad \boldsymbol{\phi}(x_2) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \boldsymbol{\phi}(x_3) = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Calculation of the Gram matrix

$$\boldsymbol{\phi}(x_1) = \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix} \quad \boldsymbol{\phi}(x_2) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \boldsymbol{\phi}(x_3) = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Gram/Kernel matrix

$$\begin{aligned}\boldsymbol{K} &= \begin{pmatrix} \boldsymbol{\phi}(x_1)^{\mathrm{T}}\boldsymbol{\phi}(x_1) & \boldsymbol{\phi}(x_1)^{\mathrm{T}}\boldsymbol{\phi}(x_2) & \boldsymbol{\phi}(x_1)^{\mathrm{T}}\boldsymbol{\phi}(x_3) \\ \boldsymbol{\phi}(x_2)^{\mathrm{T}}\boldsymbol{\phi}(x_1) & \boldsymbol{\phi}(x_2)^{\mathrm{T}}\boldsymbol{\phi}(x_2) & \boldsymbol{\phi}(x_2)^{\mathrm{T}}\boldsymbol{\phi}(x_3) \\ \boldsymbol{\phi}(x_3)^{\mathrm{T}}\boldsymbol{\phi}(x_1) & \boldsymbol{\phi}(x_3)^{\mathrm{T}}\boldsymbol{\phi}(x_2) & \boldsymbol{\phi}(x_3)^{\mathrm{T}}\boldsymbol{\phi}(x_3) \end{pmatrix} \\ &= \begin{pmatrix} 4 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 4 \end{pmatrix}\end{aligned}$$

Another solution

Here we prove that two solutions

$$\begin{aligned}\boldsymbol{w}^* &= (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I})^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{y} \\ \boldsymbol{w}^* &= \boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}\end{aligned}$$

are the same.

$$\begin{aligned}&(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I}_1)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{y} \\ &= (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I}_1)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I}_2)(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I}_2)^{-1}\boldsymbol{y} \\ &= (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I}_1)^{-1}(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{\Phi}^{\mathrm{T}})(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I}_2)^{-1}\boldsymbol{y} \\ &= (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I}_1)^{-1}(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I}_1)\boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I}_2)^{-1}\boldsymbol{y} \\ &= \boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I}_2)^{-1}\boldsymbol{y}\end{aligned}$$

Then what is the difference?

First, computing $(\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} + \lambda \boldsymbol{I})^{-1} = (\boldsymbol{K} + \lambda \boldsymbol{I})^{-1} = \boldsymbol{\alpha}$ can be more efficient than computing $(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda \boldsymbol{I})^{-1}$ when $\mathsf{N} \leq \mathsf{M}$.

More importantly, computing $(\boldsymbol{K} + \lambda \boldsymbol{I})^{-1}$ *only requires computing inner products in the new feature space!*

Now we can conclude that the exact form of $\boldsymbol{\phi}(\cdot)$ is not essential; *all we need is computing inner products $\boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\phi}(\boldsymbol{x}')$.*

For some $\boldsymbol{\phi}$ it is indeed possible to compute $\boldsymbol{\phi}(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}')$ without computing/knowing $\boldsymbol{\phi}$. This is the *kernel trick*.

Why is this helpful?

The prediction of $\boldsymbol{w}^*$ on a new example $\boldsymbol{x}$ is

$$\boldsymbol{w}^{*\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}) = \left(\sum_{n=1}^{N} \alpha_n \boldsymbol{\phi}(\boldsymbol{x}_n)^{\mathrm{T}}\right)\boldsymbol{\phi}(\boldsymbol{x}) = \sum_{n=1}^{N} \alpha_n \left(\boldsymbol{\phi}(\boldsymbol{x}_n)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})\right)$$

Therefore we do not really need to know a nonlinear mapping ϕ, only inner products in the new feature space matter!

Kernel methods are exactly about computing inner products *without knowing $\boldsymbol{\phi}$*.

Example of the kernel trick

Consider the following polynomial basis $\boldsymbol{\phi} : \mathbb{R}^2 \to \mathbb{R}^3$:

$$\boldsymbol{\phi}(\boldsymbol{x}) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

What is the inner product between $\boldsymbol{\phi}(\boldsymbol{x})$ and $\boldsymbol{\phi}(\boldsymbol{x}')$?

$$\begin{aligned}\boldsymbol{\phi}(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}') &= {x_1}^2{x_1'}^2 + 2x_1x_2x_1'x_2' + {x_2}^2{x_2'}^2 \\ &= (x_1x_1' + x_2x_2')^2 = (\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}')^2 = k(\boldsymbol{x}, \boldsymbol{x}')\end{aligned}$$

Therefore, the inner product in the new space is simply a function of the inner product in the original space.

Count the number of multiplications.

Another example

$\boldsymbol{\phi} : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{2\mathsf{D}}$ is parameterized by θ:

$$\boldsymbol{\phi}_\theta(\boldsymbol{x}) = \begin{pmatrix} \cos(\theta x_1) \\ \sin(\theta x_1) \\ \vdots \\ \cos(\theta x_{\mathsf{D}}) \\ \sin(\theta x_{\mathsf{D}}) \end{pmatrix}$$

What is the inner product between $\boldsymbol{\phi}_\theta(\boldsymbol{x})$ and $\boldsymbol{\phi}_\theta(\boldsymbol{x}')$?

$$\begin{aligned}\boldsymbol{\phi}_\theta(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}_\theta(\boldsymbol{x}') &= \sum_{d=1}^{\mathsf{D}} \cos(\theta x_d)\cos(\theta x_d') + \sin(\theta x_d)\sin(\theta x_d') \\ &= \sum_{d=1}^{\mathsf{D}} \cos(\theta(x_d - x_d')) = k(\boldsymbol{x}, \boldsymbol{x}')\end{aligned}$$

Once again, *the inner product in the new space is a simple function of the features in the original space*.

More complicated example

Based on the previous example mapping $\boldsymbol{\phi}_\theta$, we define a new one $\boldsymbol{\phi}_L : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{2\mathsf{D}(L+1)}$ as follows:

$$\boldsymbol{\phi}_L(\boldsymbol{x}) = \begin{pmatrix} \boldsymbol{\phi}_0(\boldsymbol{x}) \\ \boldsymbol{\phi}_{\frac{2\pi}{L}}(\boldsymbol{x}) \\ \boldsymbol{\phi}_{2\frac{2\pi}{L}}(\boldsymbol{x}) \\ \vdots \\ \boldsymbol{\phi}_{L\frac{2\pi}{L}}(\boldsymbol{x}) \end{pmatrix}$$

What is the inner product between $\boldsymbol{\phi}_L(\boldsymbol{x})$ and $\boldsymbol{\phi}_L(\boldsymbol{x}')$?

$$\begin{aligned} \boldsymbol{\phi}_L(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}_L(\boldsymbol{x}') &= \sum_{\ell=0}^{L} \boldsymbol{\phi}_{\frac{2\pi\ell}{L}}(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}_{\frac{2\pi\ell}{L}}(\boldsymbol{x}') \\ &= \sum_{\ell=0}^{L}\sum_{d=1}^{\mathsf{D}} \cos\left(\frac{2\pi\ell}{L}(x_d - x'_d)\right) \end{aligned}$$

Infinite dimensional mapping

Let us set $L \to \infty$. This means that $\boldsymbol{\phi}_L(\boldsymbol{x})$ vector has infinite dimension. Clearly we cannot compute $\boldsymbol{\phi}_L(\boldsymbol{x})$, but we can still compute the inner product:

$$\begin{aligned} \boldsymbol{\phi}_\infty(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}_\infty(\boldsymbol{x}') &= \int_0^{2\pi}\sum_{d=1}^{\mathsf{D}} \cos(\theta(x_d - x'_d))\, d\theta \\ &= \sum_{d=1}^{\mathsf{D}} \frac{\sin(2\pi(x_d - x'_d))}{x_d - x'_d} \end{aligned}$$

Again, a simple function of the original features.

Note that using this mapping in linear regression, we are *learning a weight $\boldsymbol{w}^*$ with infinite dimension!*

Kernel functions

Definition: a function $k : \mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}} \to \mathbb{R}$ is called a *(positive semidefinite) kernel function* if there exists a function $\boldsymbol{\phi} : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{\mathsf{M}}$ so that for any $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^{\mathsf{D}}$,

$$k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}')$$

Kernel functions are used to quantify similarity between a pair of points $\boldsymbol{x}$ and $\boldsymbol{x}'$.

Examples we have seen

$$k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}'$$
$$k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}')^2$$
$$k(\boldsymbol{x}, \boldsymbol{x}') = \sum_{d=1}^{\mathsf{D}} \frac{\sin(2\pi(x_d - x'_d))}{x_d - x'_d}$$

Using kernel functions

Choosing a nonlinear basis $\boldsymbol{\phi}$ becomes choosing a kernel function.

As long as computing the kernel function is more efficient, we should apply the kernel trick.

Gram/kernel matrix becomes:

$$\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & k(\boldsymbol{x}_1, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ k(\boldsymbol{x}_2, \boldsymbol{x}_1) & k(\boldsymbol{x}_2, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_2, \boldsymbol{x}_N) \\ \vdots & \vdots & \vdots & \vdots \\ k(\boldsymbol{x}_N, \boldsymbol{x}_1) & k(\boldsymbol{x}_N, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix}$$

In fact, k is a kernel if and only if $\boldsymbol{K}$ is positive semidefinite for *any $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, ..., $\boldsymbol{x}_N$* (**Mercer theorem**).

Using kernel functions

The prediction on a new example $\boldsymbol{x}$ is

$$\boldsymbol{w}^{*\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}) = \left(\sum_{n=1}^{N}\alpha_n\boldsymbol{\phi}(\boldsymbol{x}_n)^{\mathrm{T}}\right)\boldsymbol{\phi}(\boldsymbol{x}) = \sum_{n=1}^{N}\alpha_n\left(\boldsymbol{\phi}(\boldsymbol{x}_n)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})\right)$$
$$= \sum_{n=1}^{N}\alpha_n k(\boldsymbol{x}_n, \boldsymbol{x})$$

More examples of kernel functions

Two most commonly used kernel functions in practice:

Polynomial kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}' + c)^d$$

for $c \geq 0$ and d is a positive integer.

Gaussian kernel or Radial basis function (RBF) kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = e^{-\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|_2^2}{2\sigma^2}}$$

for some $\sigma > 0$.

Think about *what the corresponding ϕ is* for each kernel.

Composing kernels

Creating more kernel functions using the following rules:

If $k_1(\cdot,\cdot)$ and $k_2(\cdot,\cdot)$ are kernels, the followings are kernels too

- **linear combination**: $\alpha k_1(\cdot,\cdot) + \beta k_2(\cdot,\cdot)$ if $\alpha, \beta \geq 0$
- **product**: $k_1(\cdot,\cdot)k_2(\cdot,\cdot)$
- **exponential**: $e^{k(\cdot,\cdot)}$
- $\cdots$

Verify using the definition of kernel!

Kernelizing ML algorithms

There are two **main aspects** of kernelized algorithms:

- the solution is expressed as a linear combination of training examples
- algorithm relies only on inner products between data points

Kernel trick is applicable to **many** ML algorithms:

- nearest neighbor classifier
- perceptron
- logistic regression
- $\cdots$

Kernelizing NNC

For NNC with **L2 distance**, $\|\boldsymbol{x}-\boldsymbol{x}'\|_2^2$ is not a valid kernel.

But we can covert the norm

$$d(\boldsymbol{x},\boldsymbol{x}') = \|\boldsymbol{x}-\boldsymbol{x}'\|_2^2 = \boldsymbol{x}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{x}'^{\mathrm{T}}\boldsymbol{x}' - 2\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}'$$

into the following kernel function

$$d^{\mathrm{KERNEL}}(\boldsymbol{x},\boldsymbol{x}') = k(\boldsymbol{x},\boldsymbol{x}) + k(\boldsymbol{x}',\boldsymbol{x}') - 2k(\boldsymbol{x},\boldsymbol{x}')$$

which by definition is the **L2 distance in a new feature space**

$$d^{\mathrm{KERNEL}}(\boldsymbol{x},\boldsymbol{x}') = \|\boldsymbol{\phi}(\boldsymbol{x}) - \boldsymbol{\phi}(\boldsymbol{x}')\|_2^2$$

Kernelizing NNC

Regular KNN algorithm has two shortcomings.

- all neighbors receive equal weight
- the number of neighbors must be chosen globally.

Kernel addresses these issues.

Instead of selected nearest neighbors, all neighbors are used, but with different weights.

Closer neighbors receive higher weight.

The weighting function is a kernel.

One of the most common way is the Gaussian kernel

$$k(\boldsymbol{x},\boldsymbol{x}') = e^{-\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|_2^2}{2\sigma^2}}$$

Kernelizing NNC

Kernel Binary Classification Algorithm.

Given

- training data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1, 2, \ldots, \mathrm{N}\}$, where $y_n \in \{-1, 1\}$
- kernel function $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$
- input $\boldsymbol{x}$ to classify

Return the class given by

$$\operatorname{sign}\left(\sum_{n=1}^{\mathrm{N}} k(\boldsymbol{x}, \boldsymbol{x_n}) y_n\right)$$

Kernelizing Perceptron

Perceptron Algorithm:

- Pick $(\boldsymbol{x}_n, y_n)$ randomly
- Compute $y^* = \operatorname{sign}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n)$
- If $y^* \neq y_n$ then
  - $\boldsymbol{w} = \boldsymbol{w} + y_n\boldsymbol{x}_n$
- Solution $\boldsymbol{w}$ is a linear combination of features.

- Pick $(\boldsymbol{x}_n, y_n)$ randomly
- Compute $y^* = \operatorname{sign}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n))$
- If $y^* \neq y_n$ then
  - $\boldsymbol{w} = \boldsymbol{w} + y_n\boldsymbol{\phi}(\boldsymbol{x}_n)$
  - $\alpha_n = \alpha_n + y_n$
- Solution $\boldsymbol{w} = \sum_n \alpha_n \boldsymbol{\phi}(\boldsymbol{x}_n)$ is a linear combination of features.

Kernelizing Perceptron

Kernelized Perceptron Algorithm:

- Pick $(\boldsymbol{x}_n, y_n)$ randomly
- Compute $y^* = \text{sign}(\boldsymbol{w}^\mathrm{T}\boldsymbol{\phi}(\boldsymbol{x}_n))$
- If $y^* \neq y_n$ then
  - $\boldsymbol{w} = \boldsymbol{w} + y_n\boldsymbol{\phi}(\boldsymbol{x}_n)$
  - $\alpha_n = \alpha_n + y_n$
- Solution $\boldsymbol{w} = \sum_n \alpha_n \boldsymbol{\phi}(\boldsymbol{x}_n)$ is a linear combination of features.

- Pick $(\boldsymbol{x}_n, y_n)$ randomly
- Compute $y^* = \text{sign}\left(\sum_i \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}_n)\right)$
- If $y^* \neq y_n$ then
  - $\alpha_n = \alpha_n + y_n$
- Prediction $\sum_n \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x})$.

Note, in the kernelized algorithm we do not need to compute weights $\boldsymbol{w}$.

Kernelizing Perceptron

XOR example

Kernel function: $k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\mathrm{T}\boldsymbol{x}')^2$

Four 2-dimensional training points:

$$\begin{pmatrix} x_1 & x_2 & y \\ 1 & 1 & 1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \\ 1 & -1 & -1 \end{pmatrix}$$

Gram matrix $\boldsymbol{K}$:

$$\begin{pmatrix} 4 & 0 & 4 & 0 \\ 0 & 4 & 0 & 4 \\ 4 & 0 & 4 & 0 \\ 0 & 4 & 0 & 4 \end{pmatrix}$$

$k(\boldsymbol{x}_1, \boldsymbol{x}_1) = \left((1,1)^\mathrm{T}(1,1)\right)^2 = 4$ $\quad k(\boldsymbol{x}_1, \boldsymbol{x}_2) = \left((1,1)^\mathrm{T}(-1,1)\right)^2 = 0$

$k(\boldsymbol{x}_1, \boldsymbol{x}_3) = \left((1,1)^\mathrm{T}(-1,-1)\right)^2 = 4$ $\quad k(\boldsymbol{x}_1, \boldsymbol{x}_4) = \left((1,1)^\mathrm{T}(1,-1)\right)^2 = 0$

Kernelizing Perceptron

XOR example

Initialization: $\boldsymbol{\alpha} = (0,0,0,0)$.

First round

$\boldsymbol{x}_1$: compute $y^* = \text{sign}\left(\sum_{i=1}^4 \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}_1)\right) = \text{sign}(0) = -1 \neq y_1$
Thus, $\alpha_1 = y_1 = 1$.

$\boldsymbol{x}_2$: compute $y^* = \text{sign}((1,0,0,0)^\mathrm{T}(0,4,0,4)) = \text{sign}(0) = -1 = y_2$

$\boldsymbol{x}_3$: compute $y^* = \text{sign}((1,0,0,0)^\mathrm{T}(4,0,4,0)) = \text{sign}(4) = 1 = y_3$

$\boldsymbol{x}_4$: compute $y^* = \text{sign}((1,0,0,0)^\mathrm{T}(0,4,0,4)) = \text{sign}(0) = -1 = y_4$

Kernelizing Perceptron

XOR example

$\boldsymbol{\alpha} = (1,0,0,0)$.

Second round.

$\boldsymbol{x}_1$: compute $y^* = \text{sign}((1,0,0,0)^\mathrm{T}(4,0,4,0)) = \text{sign}(4) = 1 = y_1$

$\boldsymbol{x}_2$: compute $y^* = \text{sign}((1,0,0,0)^\mathrm{T}(0,4,0,4)) = \text{sign}(0) = -1 = y_2$

$\boldsymbol{x}_3$: compute $y^* = \text{sign}((1,0,0,0)^\mathrm{T}(4,0,4,0)) = \text{sign}(4) = 1 = y_3$

$\boldsymbol{x}_4$: compute $y^* = \text{sign}((1,0,0,0)^\mathrm{T}(0,4,0,4)) = \text{sign}(0) = -1 = y_4$

Converged! The prediction on a new example $\boldsymbol{x}$ is

$$\sum_{i=1}^4 \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}) = k(\boldsymbol{x}_1, \boldsymbol{x}) = \left((1,1)^\mathrm{T}\boldsymbol{x}\right)^2 = (x_1 + x_2)^2$$

Outline