CSCI567 Machine Learning (Spring 2025)

Haipeng Luo

University of Southern California

Feb 14, 2025

Administration

HW2 will be released today. Due on Feb 27th.

Outline

Outline

Acknowledgements

Not much math, a lot of empirical intuitions

The materials borrow heavily from the following sources:

- Stanford Course CS231n: `http://cs231n.stanford.edu/`
- Dr. Ian Goodfellow's lectures on deep learning: `http://deeplearningbook.org`

Both website provides tons of useful resources: notes, demos, videos, etc.

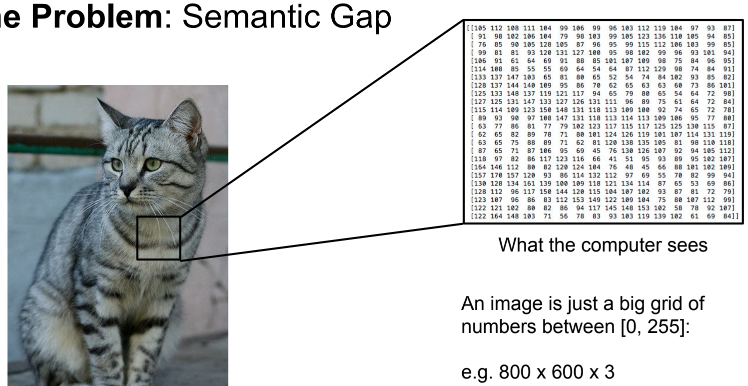
Also, demo from `https://poloclub.github.io/cnn-explainer/`

Image Classification: A core task in Computer Vision

(assume given set of discrete labels)
{dog, cat, truck, plane, ...}

→ cat

The Problem: Semantic Gap

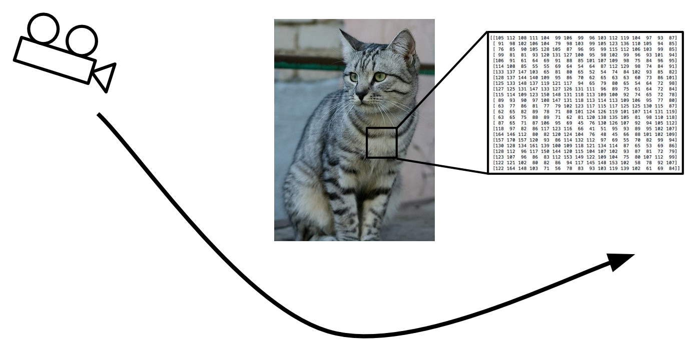


What the computer sees

An image is just a big grid of numbers between [0, 255]:

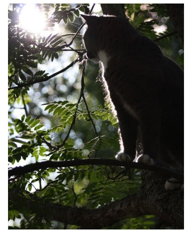
e.g. 800 x 600 x 3
(3 channels RGB)

Challenges: Viewpoint variation

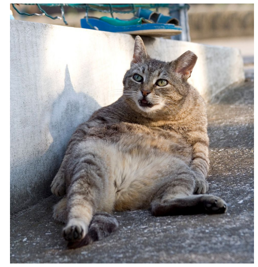


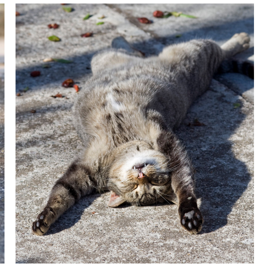
All pixels change when the camera moves!

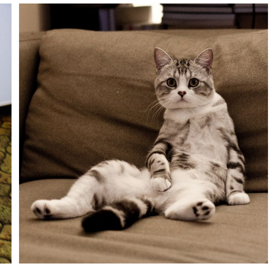
Challenges: Illumination



Challenges: Deformation







Challenges: Occlusion

Challenges: Background Clutter

Challenges: Intraclass variation

This image is CC0 1.0 public domain

Fei-Fei Li & Justin Johnson & Serena Yeung   Lecture 2 - 13   April 6, 2017

Fundamental problems in vision

The key challenge

How to train a model that can tolerate all those variations?

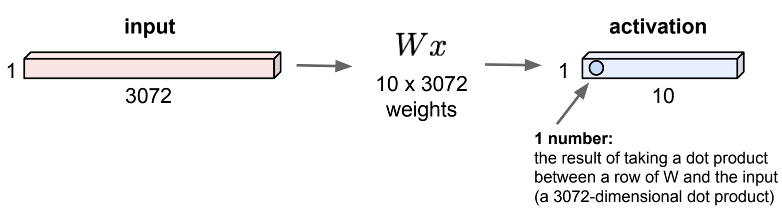
Main ideas

- need a lot of data that exhibits those variations
- need more specialized models to capture the invariance

Issues of standard NN for image inputs

Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



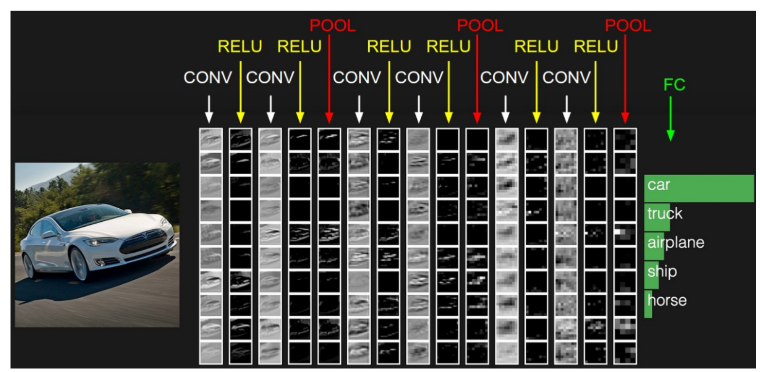
Fei-Fei Li & Justin Johnson & Serena Yeung   Lecture 5 - 27   April 18, 2017

Spatial structure is lost!

Solution: Convolutional Neural Net (ConvNet/CNN)

A special case of fully connected neural nets

- usually consist of **convolution layers**, ReLU layers, **pooling layers**, and regular fully connected layers
- key idea: *learning from low-level to high-level features*

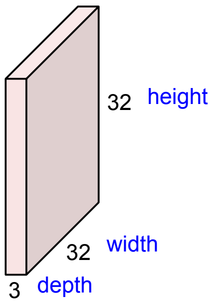


Convolution layer

Arrange neurons as a **3D volume** naturally

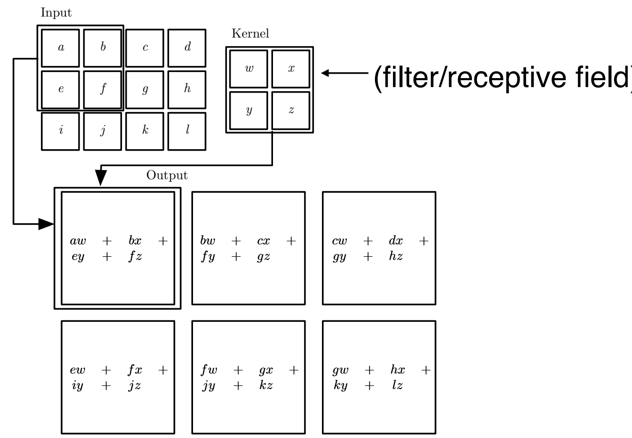
Convolution Layer

32x32x3 image -> preserve spatial structure



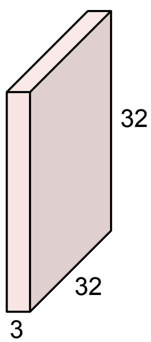
Convolution

2D Convolution

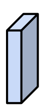


Convolution Layer

32x32x3 image



5x5x3 filter

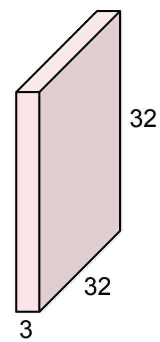


Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

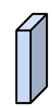
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

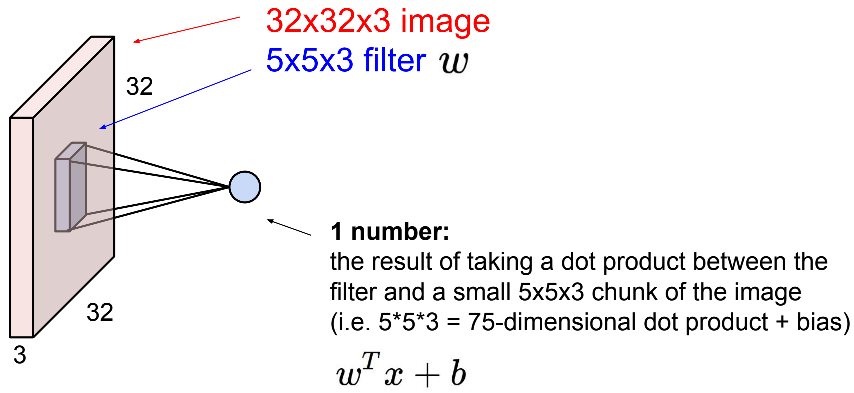


5x5x3 filter

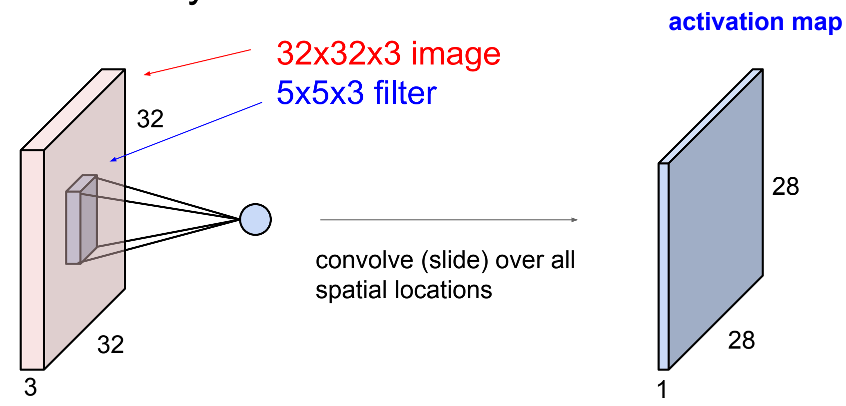


Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

Convolution Layer

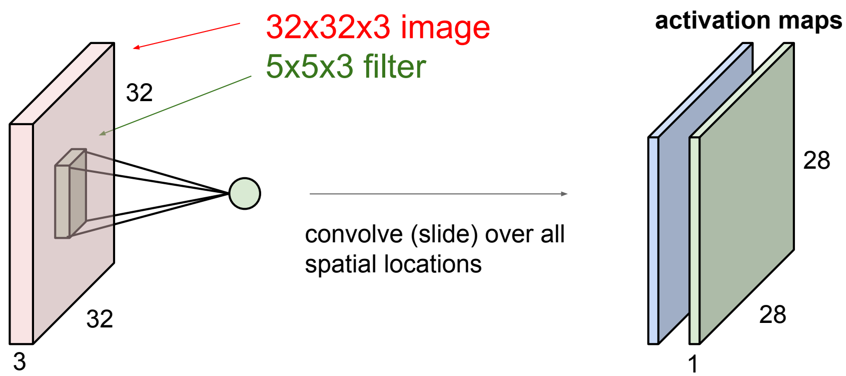


Convolution Layer

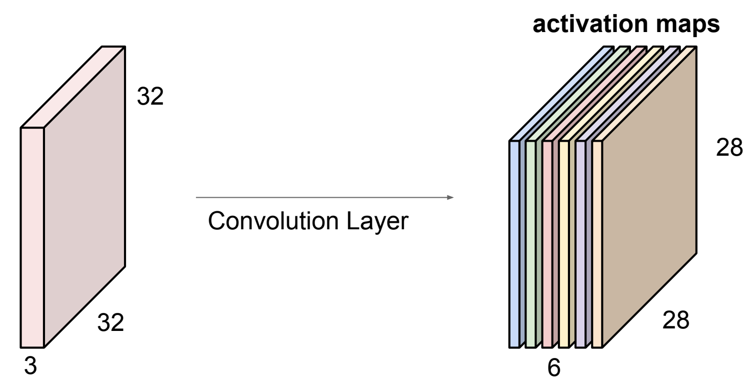


Convolution Layer

consider a second, green filter

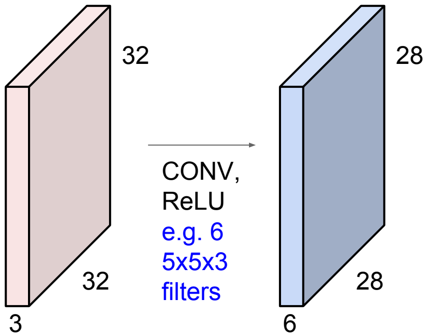


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

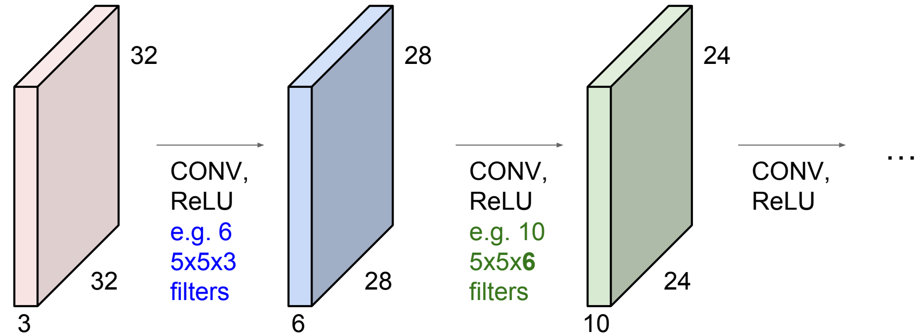


We stack these up to get a "new image" of size 28x28x6!

Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



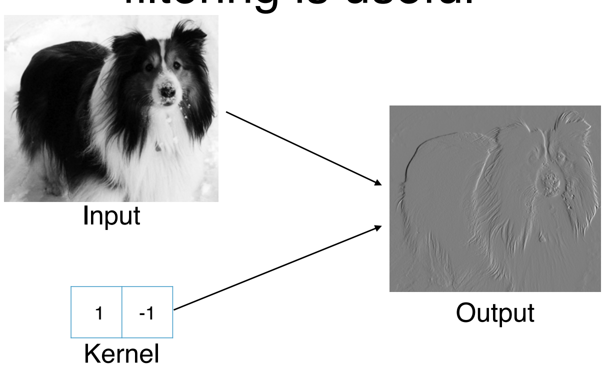
Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



Why convolution makes sense?

Main idea: **if a filter is useful at one location, it should be useful at other locations.**

A simple example why filtering is useful



Connection to fully connected NNs

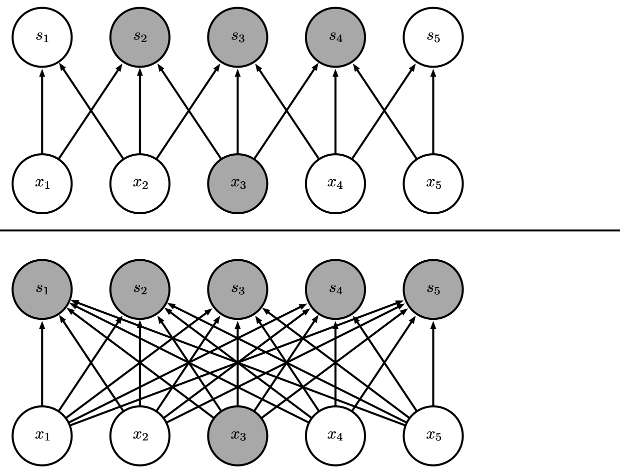
A convolution layer is a special case of a fully connected layer:

- filter = weights with **sparse connection**

Local Receptive Field Leads to Sparse Connectivity (affects less)

Sparse connections due to small convolution kernel

Dense connections



(Goodfellow 2016)

Sparse connectivity: being affected by less

Sparse connections due to small convolution kernel

Dense connections

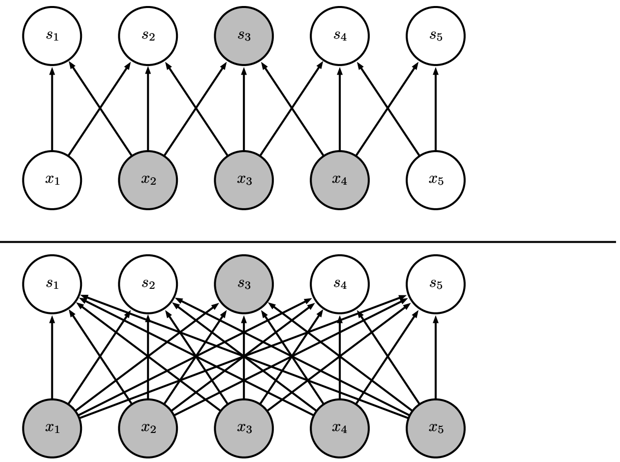


Figure 9.3

(Goodfellow 2016)

Connection to fully connected NNs

A convolution layer is a special case of a fully connected layer:

- filter = weights with **sparse connection**
- **parameters sharing**

Parameter Sharing

Convolution shares the same parameters across all spatial locations

Traditional matrix multiplication does not share any parameters

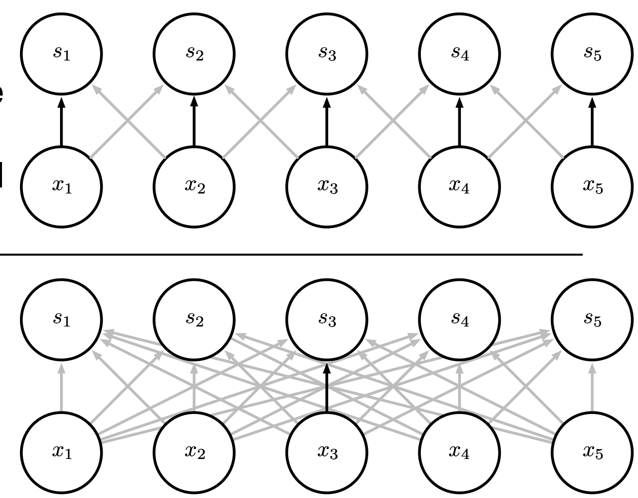


Figure 9.5

(Goodfellow 2016)

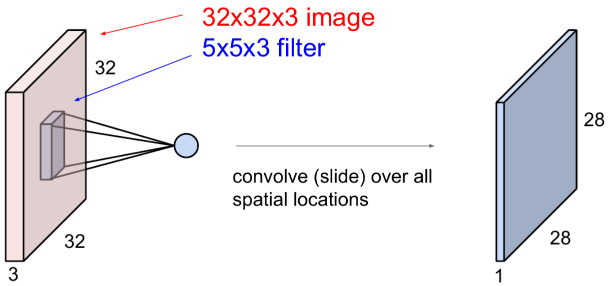
Connection to fully connected NNs

A convolution layer is a special case of a fully connected layer:

- filter = weights with **sparse connection**
- **parameters sharing**

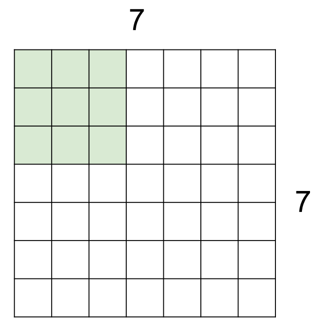
Much fewer parameters! Example (ignore bias terms):

- FC: $(32 \times 32 \times 3) \times (28 \times 28) \approx 2.4M$
- CNN: $5 \times 5 \times 3 = 75$



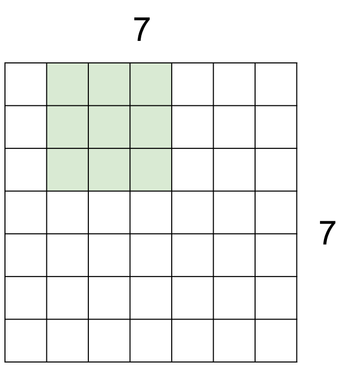
Spatial arrangement: stride and padding

A closer look at spatial dimensions:



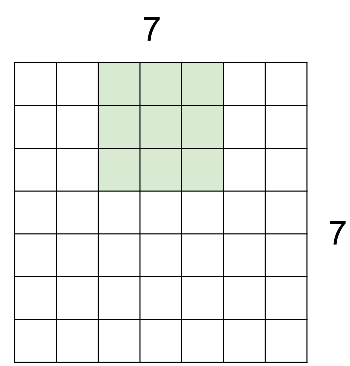
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



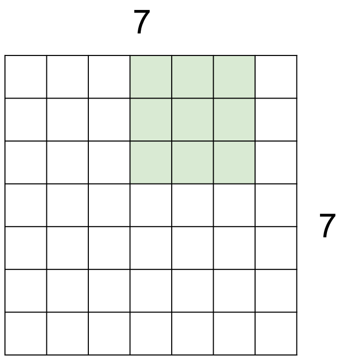
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



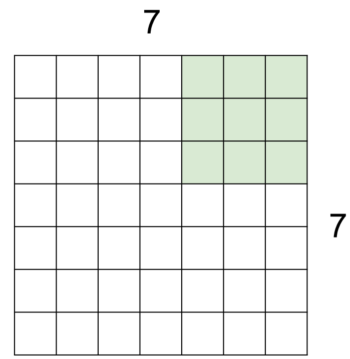
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

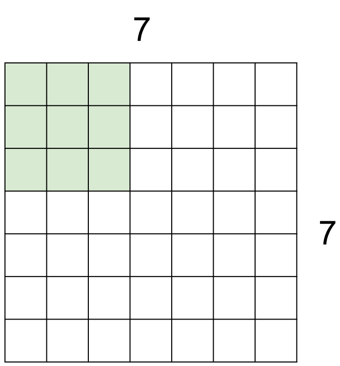
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

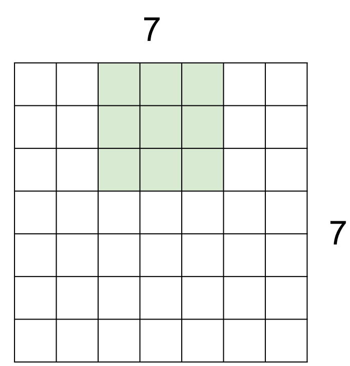
=> 5x5 output

A closer look at spatial dimensions:



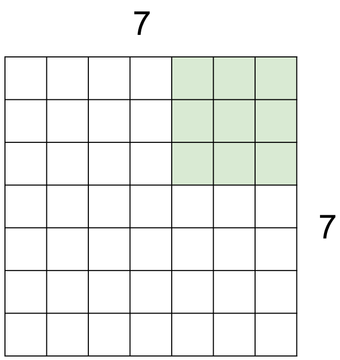
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



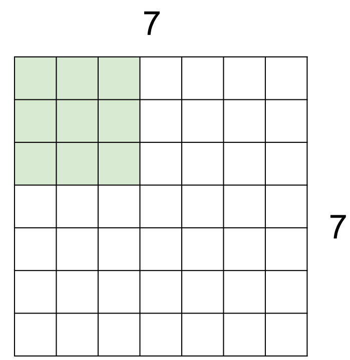
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



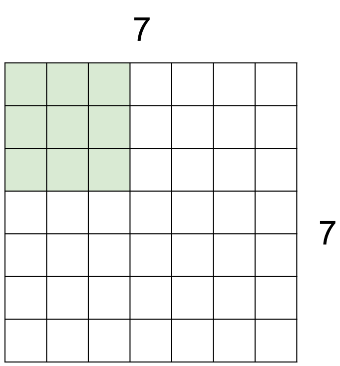
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



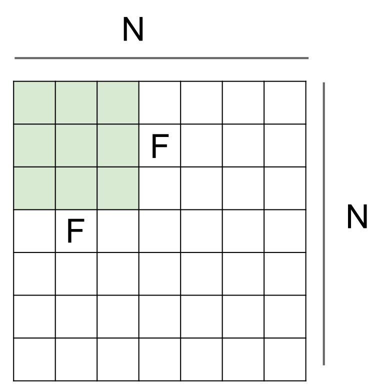
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

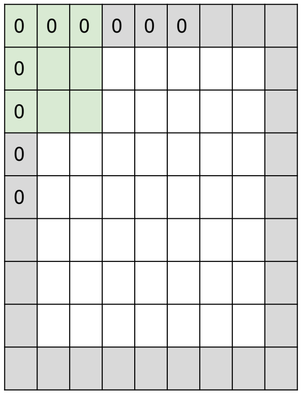
doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.



Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

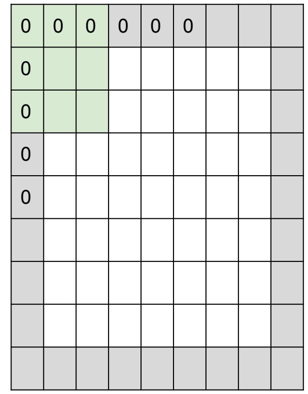
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

(recall:)
(N - F) / stride + 1

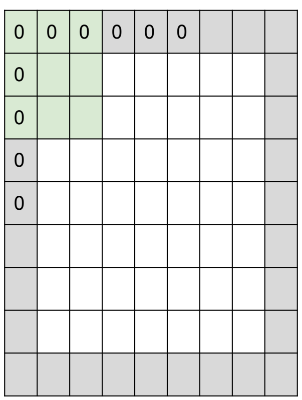
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

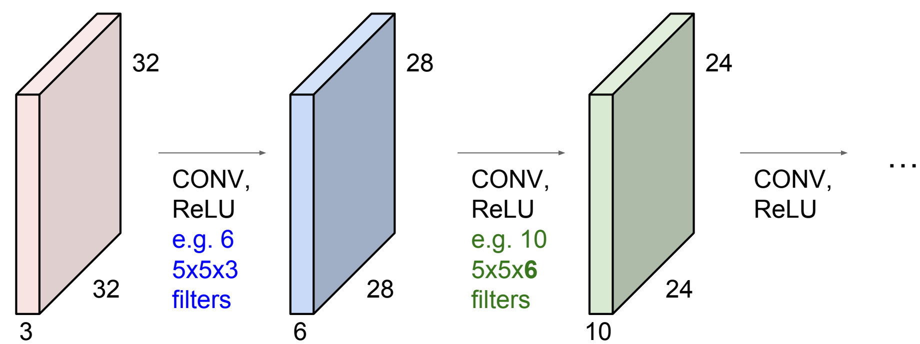
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Remember back to…
E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn't work well.



Summary for convolution layer

Input: a volume of size $W_1 \times H_1 \times D_1$

Hyperparameters:

- K filters of size $F \times F$
- stride S
- amount of zero padding P (for one side)

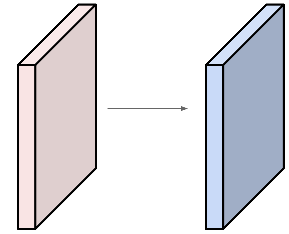
Output: a volume of size $W_2 \times H_2 \times D_2$ where

- $W_2 = (W_1 + 2P - F)/S + 1$
- $H_2 = (H_1 + 2P - F)/S + 1$
- $D_2 = K$

#parameters: $(F \times F \times D_1 + 1) \times K$ weights

Common setting: $F = 3, S = P = 1$

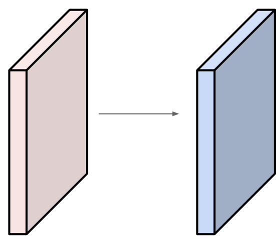
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size: ?

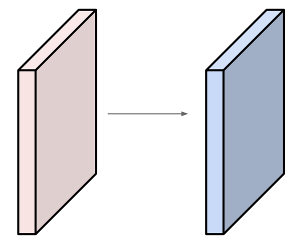
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

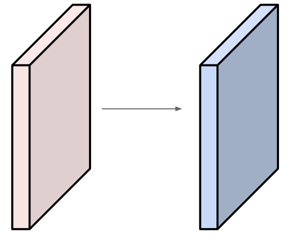
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

Examples time:



Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

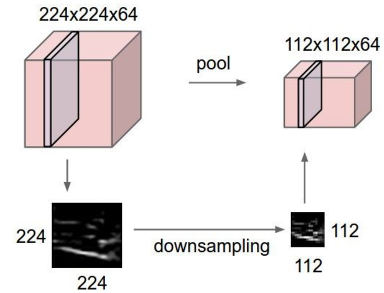
each filter has 5*5*3 + 1 = 76 params (+1 for bias)

=> 76*10 = **760**

Another element: pooling

Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:



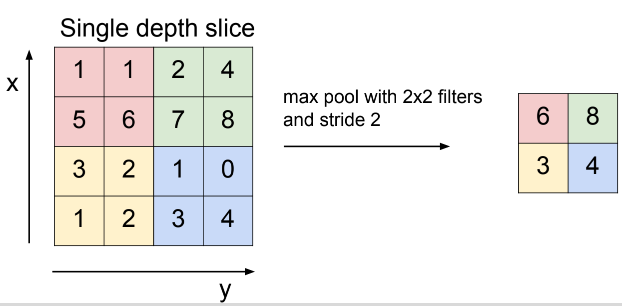
Pooling

Similar to a filter, except

- depth is always 1
- different operations: average, L2-norm, max
- no parameters to be learned

Max pooling with 2×2 filter and stride 2 is very common

MAX POOLING



Putting everything together

Typical architecture for CNNs:

Input $\rightarrow$ [[Conv $\rightarrow$ ReLU]*N $\rightarrow$ Pool?]*M $\rightarrow$ [FC $\rightarrow$ ReLU]*Q $\rightarrow$ FC

Common choices: $N \leq 5, Q \leq 2$, M is large

Well-known CNNs: LeNet, AlexNet, ZF Net, GoogLeNet, VGGNet, etc.

All achieve excellent performance on image classification tasks.

How to train a CNN?

How do we learn the filters/weights?

Essentially the same as FC NNs: apply **SGD/backpropagation**

Outline

Motivation

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

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

- neural network is one approach: learn $\boldsymbol{\phi}$ from data
- **kernel method** is another one: sidestep the issue of choosing $\boldsymbol{\phi}$ by using *kernel functions*

Case study: regularized linear regression

Kernel methods work for *many problems* and we take **regularized linear regression** as an example.

Recall the regularized least square solution:

$$\begin{aligned}\boldsymbol{w}^* &= \operatorname*{argmin}_{\boldsymbol{w}} F(\boldsymbol{w}) \\ &= \operatorname*{argmin}_{\boldsymbol{w}} \left(\|\boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \lambda\|\boldsymbol{w}\|_2^2\right) \\ &= \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{y}\end{aligned} \quad \Bigg| \quad \boldsymbol{\Phi} = \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}} \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}} \\ \vdots \\ \boldsymbol{\phi}(\boldsymbol{x}_{\mathsf{N}})^{\mathrm{T}} \end{pmatrix}, \quad \boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{\mathsf{N}} \end{pmatrix}$$

Issue: *operate in space $\mathbb{R}^{\mathsf{M}}$ and M could be huge or even infinity!*

A closer look at the least square solution

By setting the gradient of $F(\boldsymbol{w}) = \|\boldsymbol{\Phi w} - \boldsymbol{y}\|_2^2 + \lambda\|\boldsymbol{w}\|_2^2$ to be $\boldsymbol{0}$:

$$\boldsymbol{\Phi}^{\mathrm{T}}(\boldsymbol{\Phi w}^* - \boldsymbol{y}) + \lambda \boldsymbol{w}^* = \boldsymbol{0}$$

we know

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

Thus the least square solution is **a linear combination of features**!

Note this is true for perceptron and many other problems.

Of course, the above calculation does not show what $\boldsymbol{\alpha}$ is.

Why is this helpful?

Assuming we know $\boldsymbol{\alpha}$, the prediction of $\boldsymbol{w}^*$ on a new example $\boldsymbol{x}$ is

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

Therefore we do not really need to know $\boldsymbol{w}^*$. *Only inner products in the new feature space matter!*

Kernel methods are exactly about computing inner products *without knowing ϕ*.

But we need to figure out what $\boldsymbol{\alpha}$ is first!

How to find $\boldsymbol{\alpha}$?

Plugging in $\boldsymbol{w} = \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}$ into $F(\boldsymbol{w})$ gives

$$\begin{aligned} G(\boldsymbol{\alpha}) &= F(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}) \\ &= \|\boldsymbol{\Phi\Phi}^{\mathrm{T}}\boldsymbol{\alpha} - \boldsymbol{y}\|_2^2 + \lambda\|\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}\|_2^2 \\ &= \|\boldsymbol{K\alpha} - \boldsymbol{y}\|_2^2 + \lambda\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K\alpha} \qquad (\boldsymbol{K} = \boldsymbol{\Phi\Phi}^{\mathrm{T}} \in \mathbb{R}^{\mathsf{N}\times\mathsf{N}}) \end{aligned}$$

$\boldsymbol{K}$ is called **Gram matrix** or **kernel matrix** where the (i, j) entry is

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

Example 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

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

Gram matrix vs covariance matrix

	dimensions	**entry (i,j)**	**property**
$\mathbf{\Phi}\mathbf{\Phi}^{\mathrm{T}}$	N × N	$\phi(\boldsymbol{x}_i)^{\mathrm{T}}\phi(\boldsymbol{x}_j)$	both are **symmetric** and **positive semidefinite**
$\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi}$	M × M	$\sum_{n=1}^{N}\phi(\boldsymbol{x}_n)_i\phi(\boldsymbol{x}_n)_j$	

How to find $\boldsymbol{\alpha}$?

Minimize (the so-called *dual formulation*)

$$G(\boldsymbol{\alpha}) = \|\boldsymbol{K}\boldsymbol{\alpha} - \boldsymbol{y}\|_2^2 + \lambda\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\alpha}$$

Setting the derivative to $\mathbf{0}$ we have

$$\mathbf{0} = (\boldsymbol{K}^2 + \lambda\boldsymbol{K})\boldsymbol{\alpha} - \boldsymbol{K}\boldsymbol{y} = \boldsymbol{K}\left((\boldsymbol{K} + \lambda\boldsymbol{I})\boldsymbol{\alpha} - \boldsymbol{y}\right)$$

Thus $\boldsymbol{\alpha} = (\boldsymbol{K} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$ **is a minimizer** and we obtain

$$\boldsymbol{w}^* = \mathbf{\Phi}^{\mathrm{T}}\boldsymbol{\alpha} = \mathbf{\Phi}^{\mathrm{T}}(\boldsymbol{K} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$

Exercise: *are there other minimizers? and are there other $\boldsymbol{w}^*$'s?*

Comparing two solutions

Minimizing $F(\boldsymbol{w})$ gives $\boldsymbol{w}^* = (\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi} + \lambda\boldsymbol{I})^{-1}\mathbf{\Phi}^{\mathrm{T}}\boldsymbol{y}$

Minimizing $G(\boldsymbol{\alpha})$ gives $\boldsymbol{w}^* = \mathbf{\Phi}^{\mathrm{T}}(\mathbf{\Phi}\mathbf{\Phi}^{\mathrm{T}} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$

Note $\boldsymbol{I}$ has different dimensions in these two formulas.

Natural question: *are they the same or different?*

They have to be the same because $F(\boldsymbol{w})$ has a unique minimizer!

And they are:

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

Then what is the difference?

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

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

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

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

Example

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 $\boldsymbol{\phi}(\boldsymbol{x}')$?

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

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

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 $\boldsymbol{\phi}_\theta(\boldsymbol{x}')$?

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

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 $\boldsymbol{\phi}_\theta$, define $\boldsymbol{\phi}_L : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{2\mathsf{D}(L+1)}$ for some integer L:

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

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

When $L \to \infty$, even if we cannot compute $\boldsymbol{\phi}(\boldsymbol{x})$, a vector of *infinite dimension*, we can still compute 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 $k : \mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}$ is called a *kernel function* if there exists a function $\phi : \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}^{\mathsf{M}}$ so that for any $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^{\mathsf{D}}$,

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

Examples we have seen

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

$$k(\boldsymbol{x}, \boldsymbol{x}') = \sum_{d=1}^{\mathsf{D}} \frac{\sin(2\pi(x_d - x'_d))}{x_d - x'_d}$$

Using kernel functions

Choosing a nonlinear basis ϕ 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 N and any $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$* (**Mercer theorem**).

- useful for proving that a function is not a kernel

Examples that are not kernels

Function

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

is *not a kernel*, why?

If it is a kernel, the kernel matrix for two data points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$:

$$\boldsymbol{K} = \begin{pmatrix} 0 & \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2^2 \\ \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2^2 & 0 \end{pmatrix}$$

must be positive semidefinite, *but is it?*

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

- **conical 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!

Predicting with a kernel function

As long as $\boldsymbol{w}^* = \sum_{n=1}^N \alpha_n \boldsymbol{\phi}(\boldsymbol{x}_n)$, prediction on a new example $\boldsymbol{x}$ becomes

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

This is a non-parametric method!

Kernelizing other ML algorithms

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

- nearest neighbor classifier
- Perceptron (HW2)
- logistic regression
- SVM (next week)
- $\cdots$