CSCI-567: Machine Learning

Prof. Victor Adamchik

U of Southern California

July 13, 2020

Your model is only as good as your data.

Administration

Midterm Exam

- time: July 16 (Thursday) from noon to 2:50 pm
- location: remote
- you can start preparing after this lecture
- practice exam will be posted
- TA's Review: Wednesday during the lecture time

Midterm Exam

Instructions:

- This is a take-home open-book midterm exam.
- The exams is paper-and-pencil, which you scan it and submit electronically.
- Questions should be answered concisely.
- Write legibly, avoid cursive writings.

Outline

Outline

Acknowledgements

The materials I used to develop these notes include the following sources:

- Stanford Course Cs231n: `http://cs231n.stanford.edu/` (taught by Prof. Fei-Fei Li and her students).
- Dr. Ian Goodfellow's lectures on deep learning: `http://deeplearningbook.org`

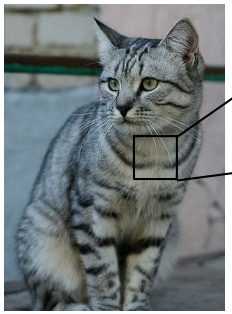
Both website provides tons of useful resources: notes, demos, videos, etc.

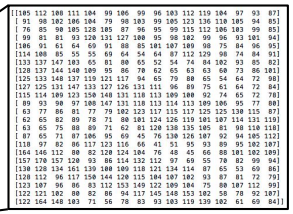
Image Classification: A core task in Computer Vision

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

cat

The Problem: Semantic Gap



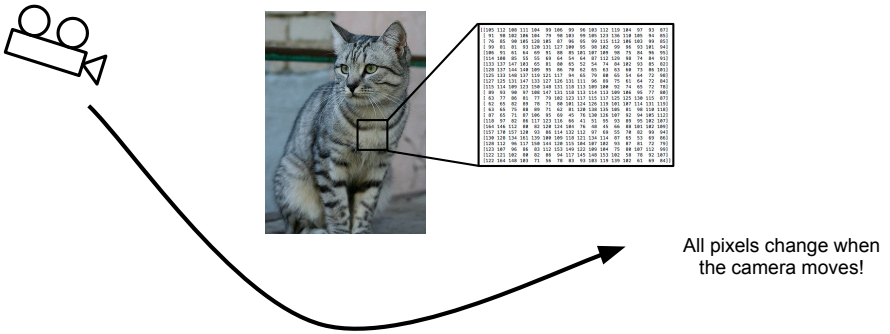


What the computer sees

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



Challenges: Illumination

Challenges: Deformation

Challenges: Occlusion

Challenges: Background Clutter

This image is CC0 1.0 public domain

This image is CC0 1.0 public domain

Challenges: Intraclass variation

This image is CC0 1.0 public domain

Fundamental problems in vision

An image classifier is not like sorting a list of numbers: there no obvious way to develop an algorithm for recognizing a cat, or other classes.

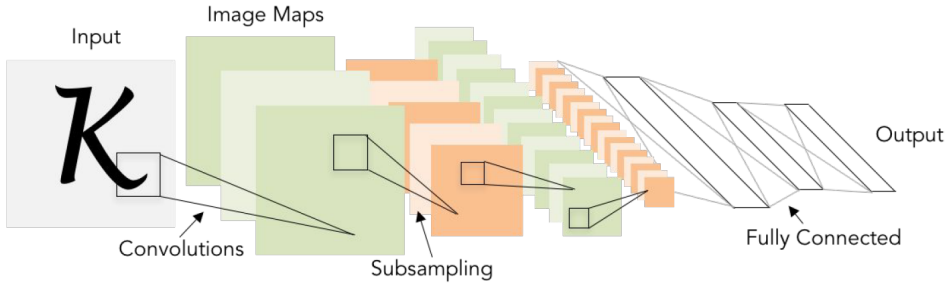
The key challenge
How to train a model that can tolerate all those variations?

Main ideas

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

A bit of history:

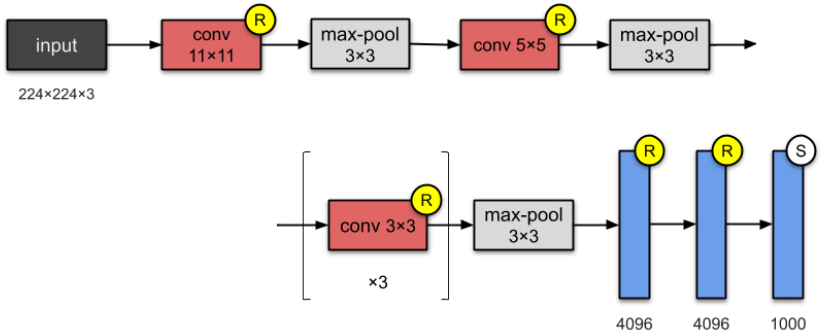
Gradient-based learning applied to document recognition [LeCun, Bottou, Bengio, Haffner 1998]



LeNet-5

A bit of history:

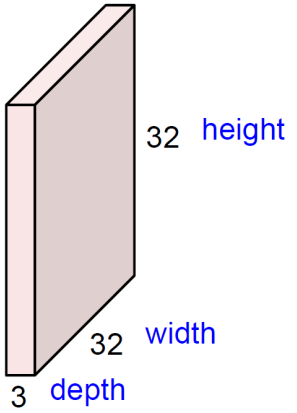
ImageNet Classification with Deep Convolutional Neural Networks
[Krizhevsky, Sutskever, Hinton, 2012]



Convolution layer

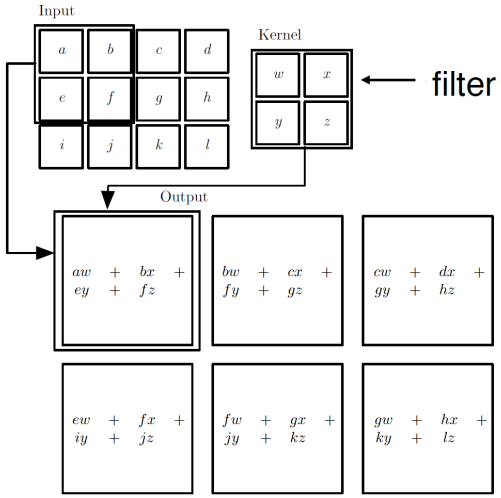
Arrange neurons as a **3D matrix/tensor** naturally.

$32 \times 32 \times 3$ image – this preserves spatial structure



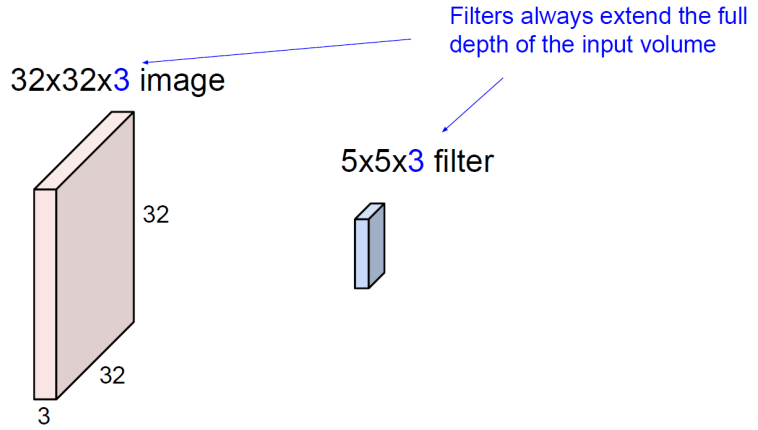
2D Convolution

This is a computation of a dot product.



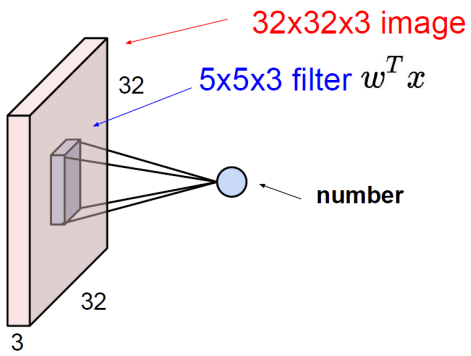
Convolution layer

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



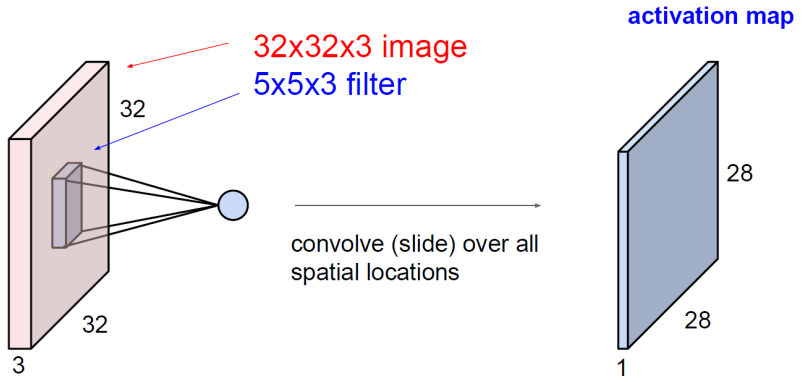
Convolution layer

The result of taking a dot product between the filter $w^T x$ and a small 5x5x3 chunk of the image.



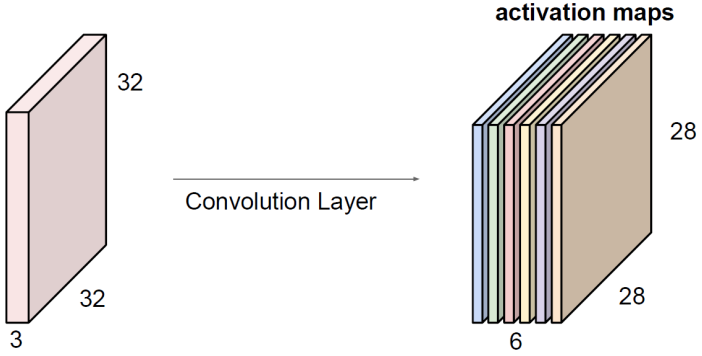
Convolution layer

The result of taking a dot product between the filter $w^T x$ and the image.



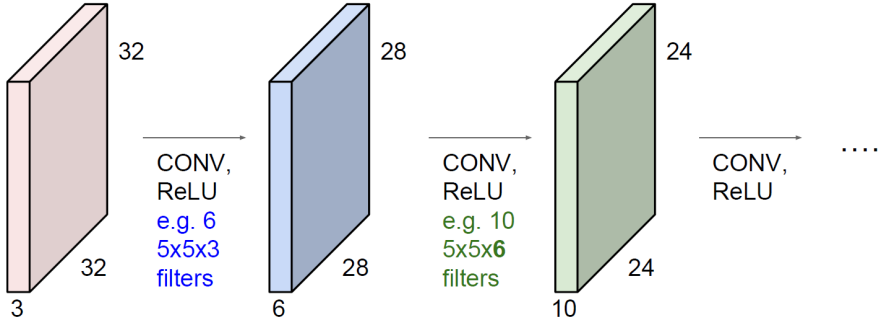
Convolution layer

If we had six $5 \times 5 \times 3$ filters, we'll get 6 separate activation maps. We stack them up to get a new image of size $28 \times 28 \times 6$.



Convolution layer

Convolution Network is a sequence of Convolutional Layers, interspersed with activation functions



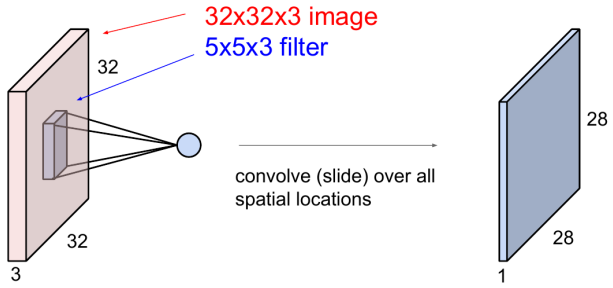
Connection to fully connected NNs

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

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

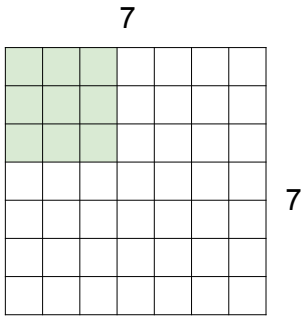
Much less parameters! Example:

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



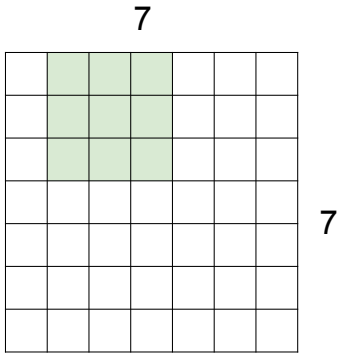
Spatial arrangement: stride and padding

A closer look at spatial dimensions:



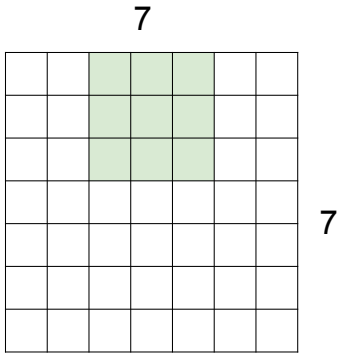
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



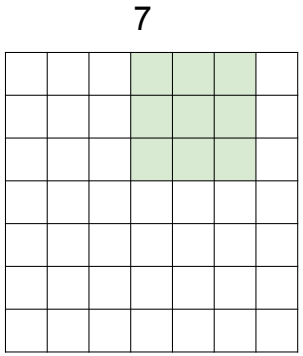
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



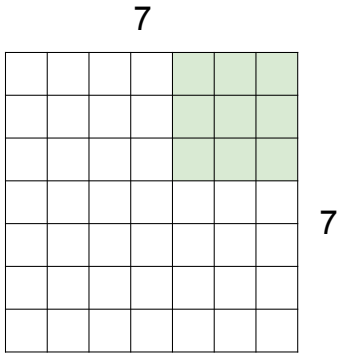
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

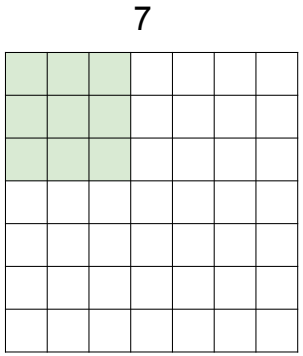
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

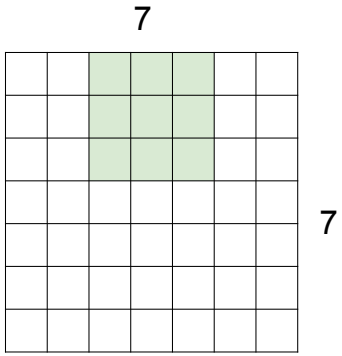
=> 5x5 output

A closer look at spatial dimensions:



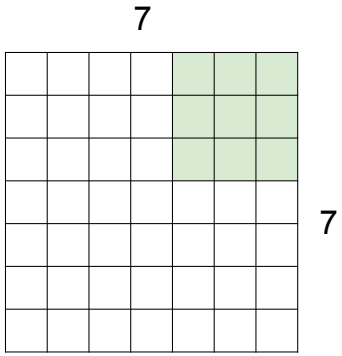
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



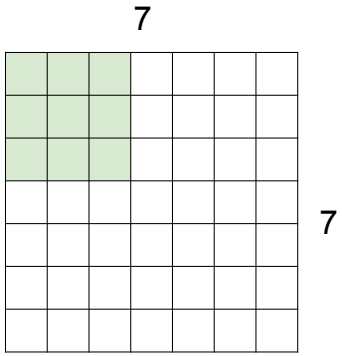
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



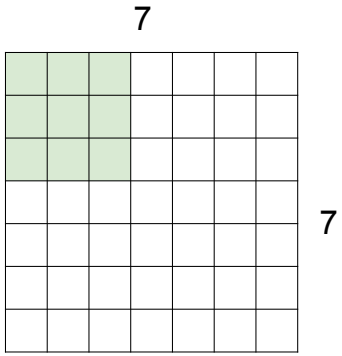
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



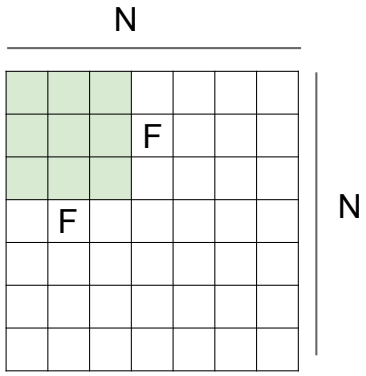
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



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

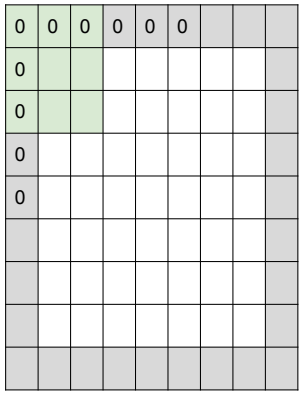
doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.



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 :\

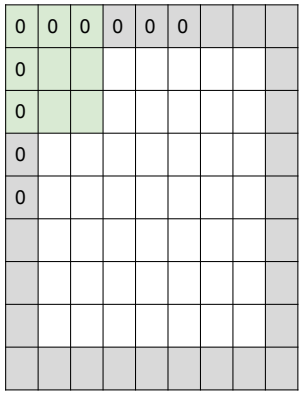
In practice: Common to zero pad the border



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

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

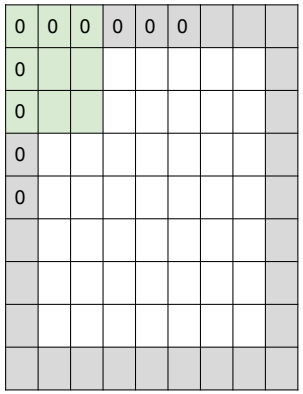
In practice: Common to zero pad the border



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

7x7 output!

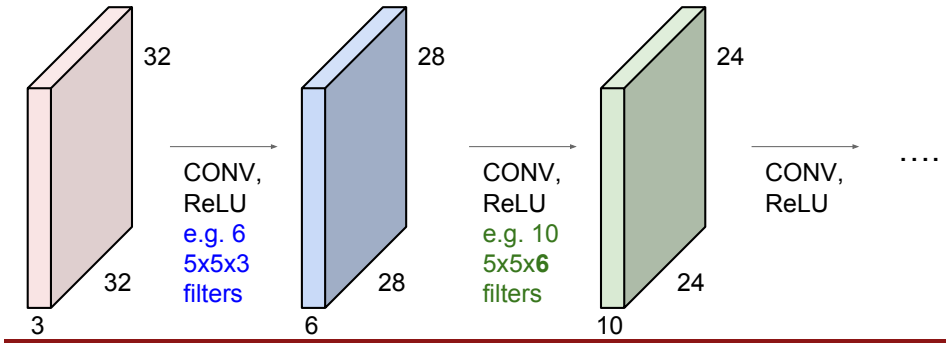
In practice: Common to zero pad the border



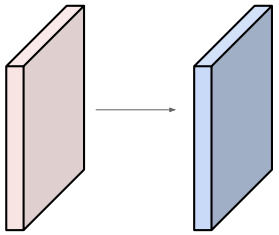
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

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.



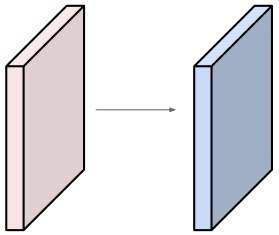
Examples time:



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

Output volume size: ?

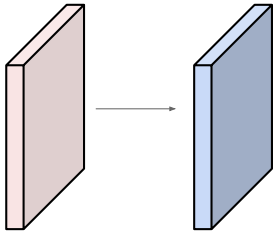
Examples time:



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

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

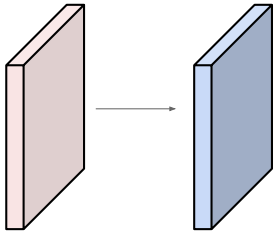
Examples time:



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**

Summary for convolution layer

Input: a tensor 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 tensor of size $W_2 \times H_2 \times D_2$ where

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

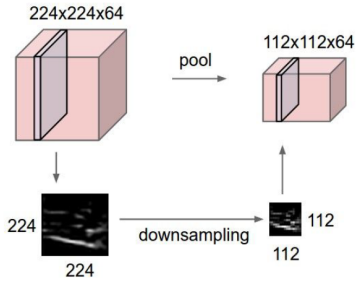
#parameters: $(F \times F \times D_1 + 1) \times K$ weights

Common setting: $F = 3, S = P = 1$ and K is a power of two.

Another element: pooling

Pooling layer:

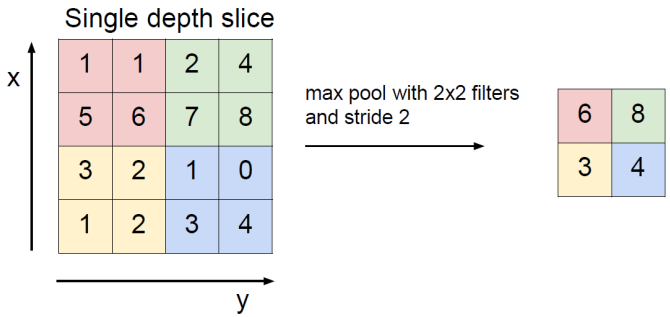
- makes the representations smaller and more manageable
- operates over each activation map independently:



Types of pooling: average, L2-norm, max

Max pooling

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



Putting everything together

Typical architecture for CNNs:

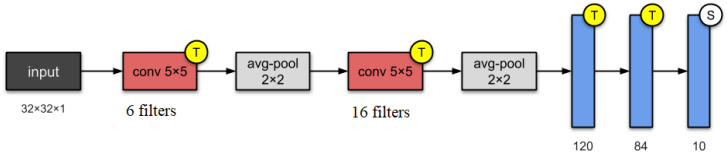
Input → [[Conv → ReLU]*N → Pool]*M → [FC → ReLU]*Q → Softmax

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

Well-known CNNs:

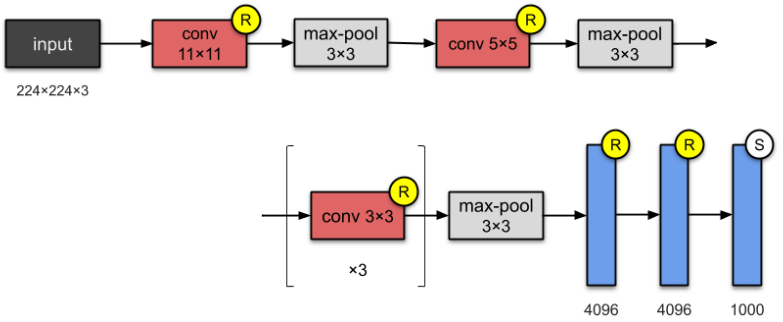
LeNet (1998), AlexNet(2012), ZF Net(2013), GoogLeNet(2014), VGGNet(2014), ResNet(2015), Soushen (2016), SENet(2017)

LeNet - 1996



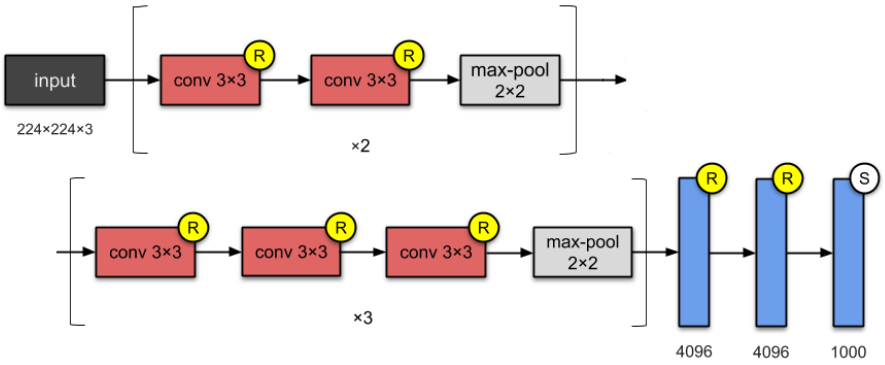
T stands for the tanh function.
S stands for the softmax function

AlexNet - 2012

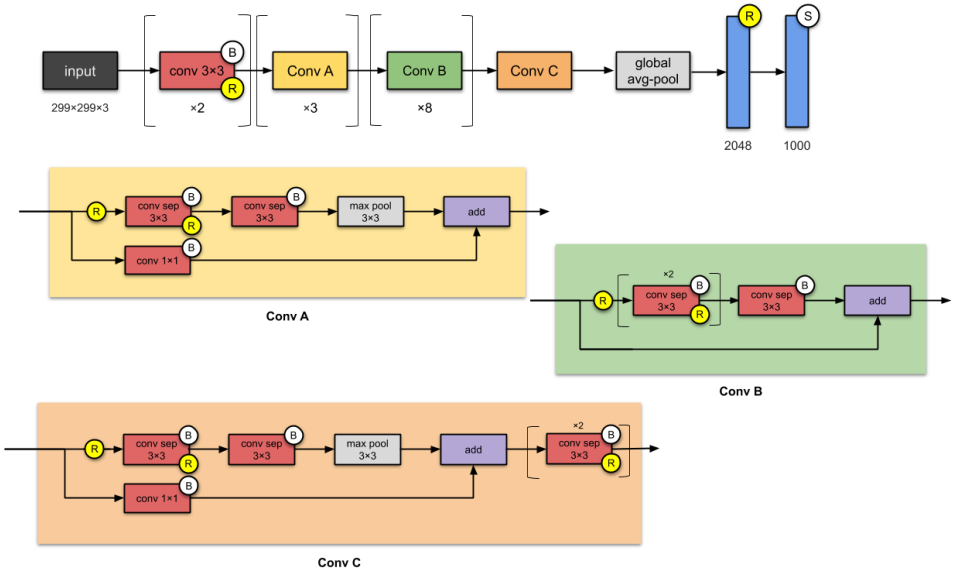


R stands for the ReLU function.

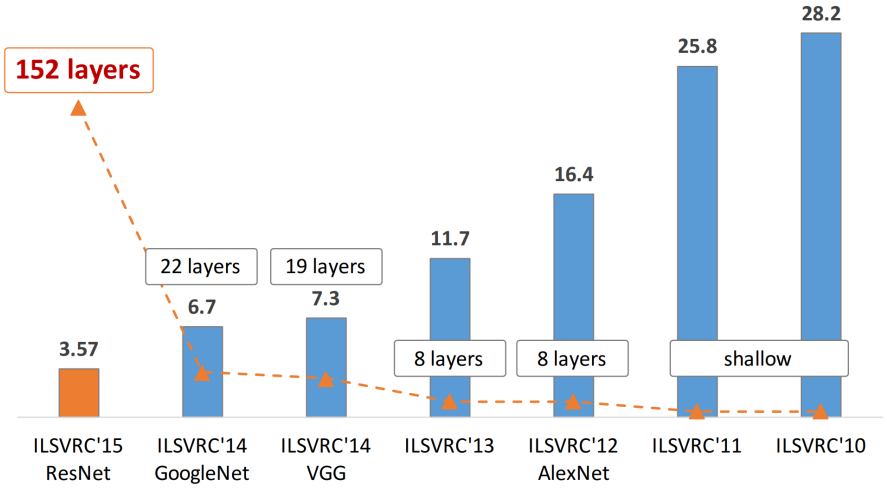
Visual Geometry Group - 2014



ResNet - 2015



Revolution of Depth



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition".

How to train a CNN?

How do we learn the filters/weights?

Run the image through 30+ layers of convolutions and train the filters with **SGD/backpropagation**

What computer architecture to use?

buy 10 or more GPUs (NVIDIA graphic card)

What software package to use?

State-of-the-art: TensorFlow plus Keras or Caffe on the top.

See demo at:
`http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html`

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

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

We will rewrite the solution in a different form.

Another solution

Another minimizer is

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

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,

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

Another solution

Here we prove that two solutions

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

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 N $\leq$ 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* ϕ.

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

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

$$= \begin{pmatrix} 4 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 4 \end{pmatrix}$$

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**: $e^{k(\cdot,\cdot)}$
- ...

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
- ...

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 $\mathcal{D} = \{(\boldsymbol{x}_n, y_n), n = 1, 2, \ldots, \mathsf{N}\}$, 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

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

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 Perceptron

Perceptron Algorithm:

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

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 $\boldsymbol{K}$:

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

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

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

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

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

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