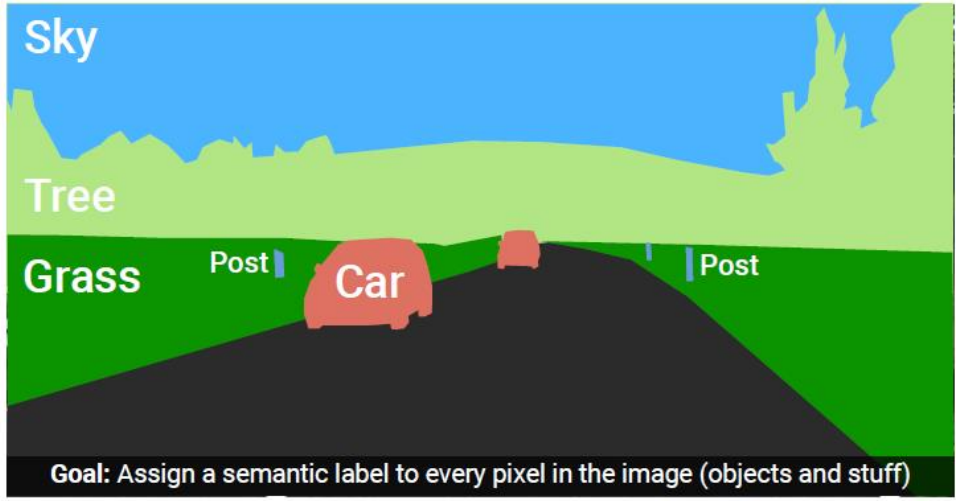
Image Classification: From Classic ML to Deep Learning

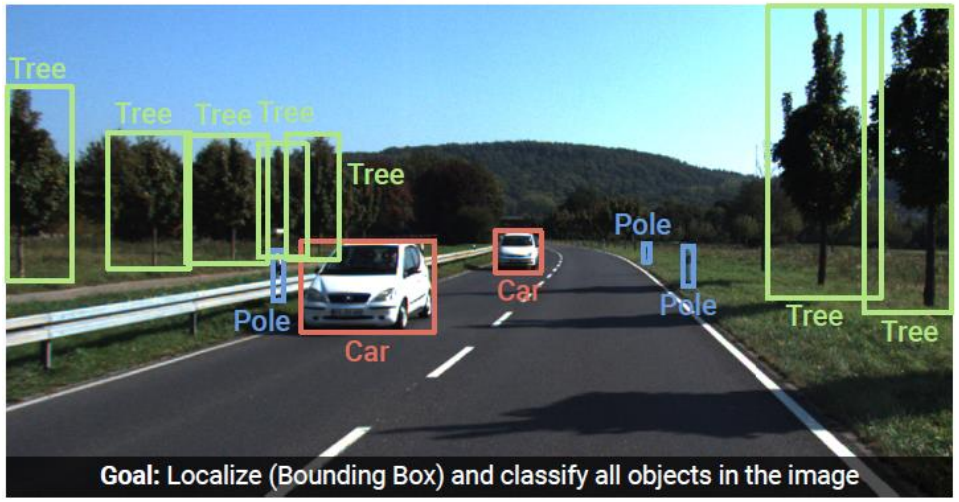
Computer Vision

Image Understanding (Recognition)

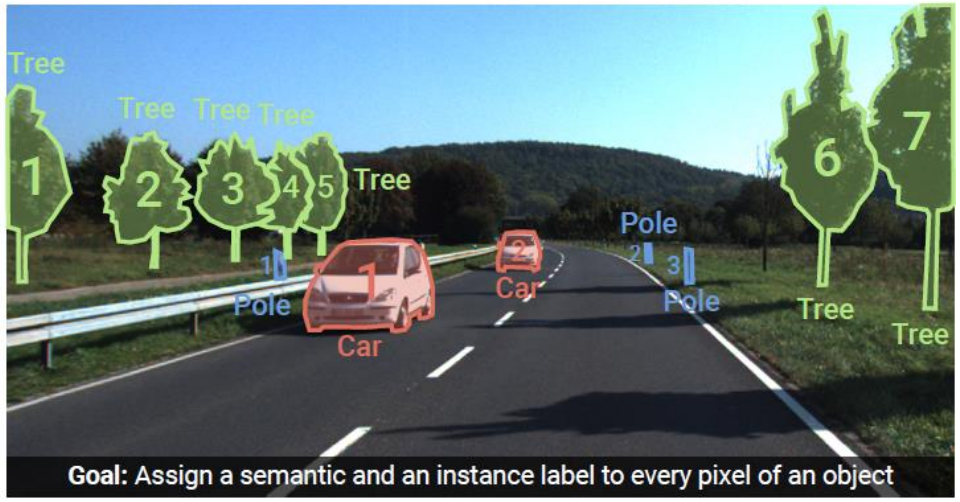
Image Classification



Semantic Segmentation

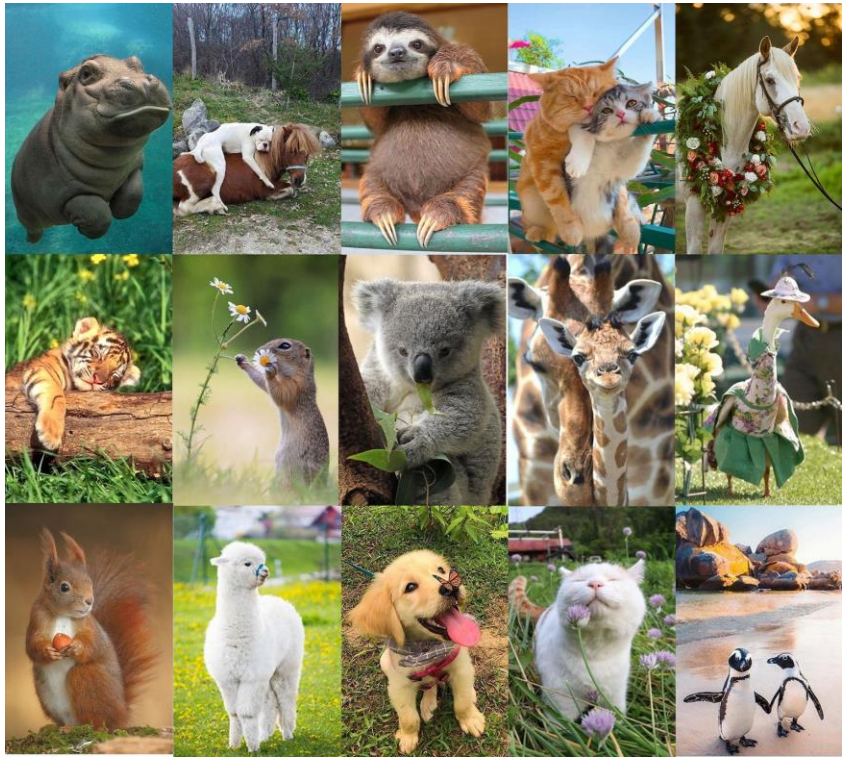


Object Detection



Instance Segmentation

Need for ML



prime example (and also foundation for detection and segmentation):

image classification (whole-image class recognition) according to generic object categories (e.g., cat)

plain keypoint-feature matching only really works for specific instances of a class

→ need to compare with generic objects (e.g., kind of “abstract cat”)

Let’s learn it from data …

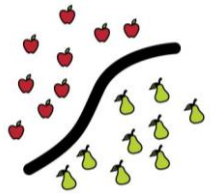
What if you haven’t seen this very cat before?

MACHINE LEARNING

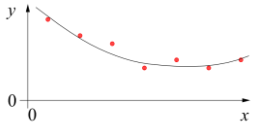
training target available (labeled or past data)

SUPERVISED

CLASSIFICATION



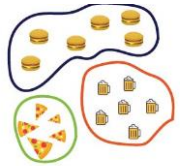
REGRESSION



data not labeled in any way

UNSUPERVISED

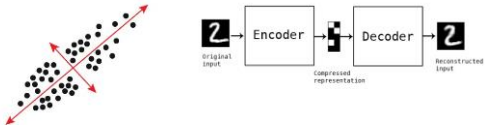
CLUSTERING



SELF-SUPERVISED

ASSOCIATION

DIMENSIONALITY REDUCTION



no supervision, but goal-based interaction with environment

REINFORCEMENT LEARNING

LEARN STATE OR ACTION VALUES

LEARN POLICY DIRECTLY

learning by teacher (high-dimensional curve fitting)

learning by observation (pattern recognition)

learning by trial-and-error (sequential decision making)

unsupervised and reinforcement learning can both be cast as supervised-learning setup

Supervised Learning

Target Quantity

- **known in training**: labeled samples or observations from past
- to be **predicted** for unknown cases (e.g., future values)

Features

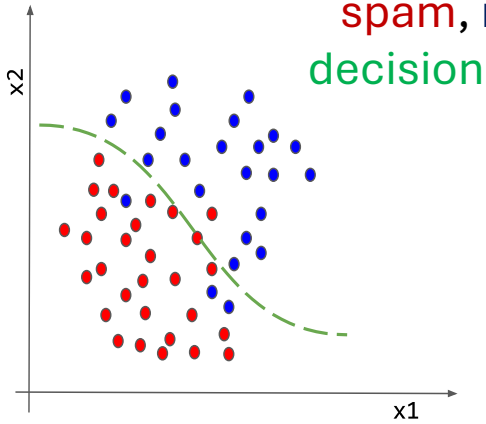
(prepared) input information that is
- correlated to target quantity
- known at prediction time

Example: Spam Filtering

classify emails as spam or no spam

*use accordingly **labeled emails as training set***

use information like occurrence of specific words or email length as **features**



features x1 and x2

spam, no spam

decision boundary

Optimization

General Recipe of Statistical Learning

ML algorithm by combining:

- **model** (e.g., linear function & Gaussian distribution)
- **objective function** (e.g., squared residuals)
- **optimization algorithm** (e.g., gradient descent)
- **regularization** (e.g., L2, dropout)

Supervised Learning in Mathematical Terms

map input to output: $y = f(\boldsymbol{x})$ (estimated: $\hat{f}(\boldsymbol{x})$)

random variables Y and $\boldsymbol{X} = (X_1, X_2, \cdots, X_p)$ ← usually high-dimensional

training (curve fitting / parameter estimation):

fit data set of $(y_i, \boldsymbol{x}_i)$ pairs → minimize deviations between y and $\hat{f}(\boldsymbol{x})$

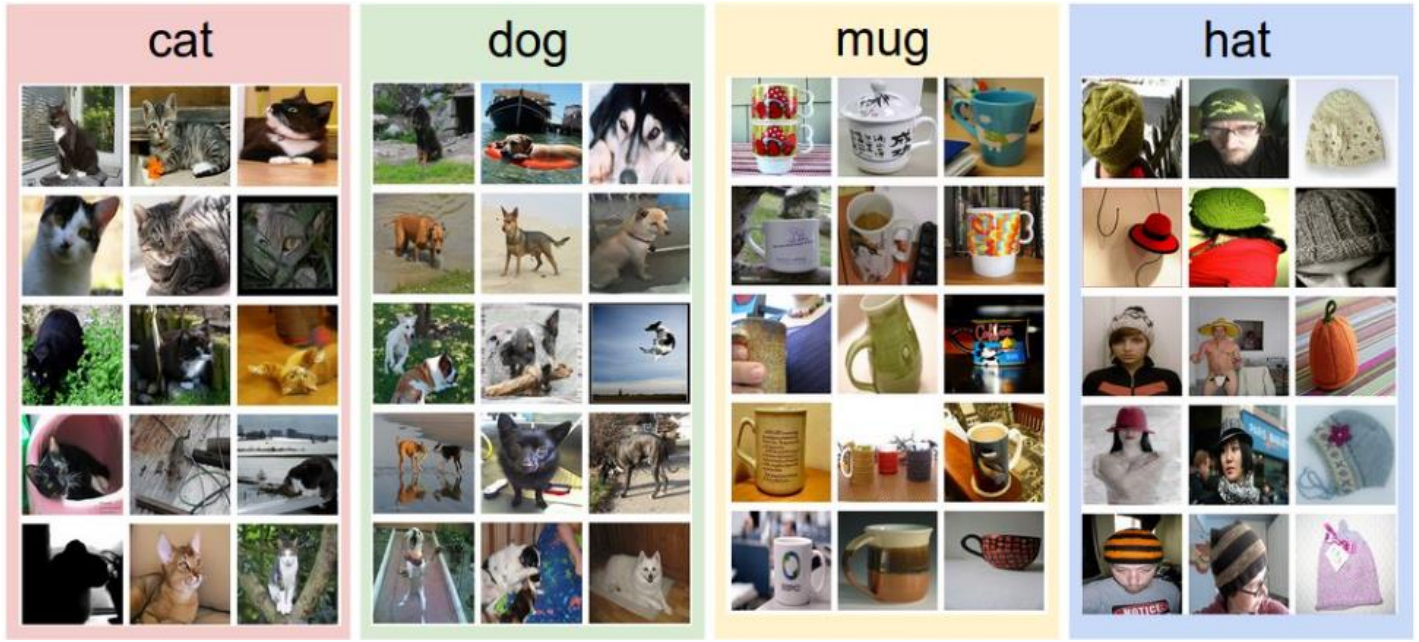
discriminative models: predict conditional density function $p(y|\boldsymbol{x})$

as opposed to generative models: predict $p(y, \boldsymbol{x})$ (or just $p(\boldsymbol{x})$) → $\boldsymbol{x}$ not given, more difficult

Example: Image Classification

training data set:

labels:



test data with learned classifier:

$f($  $) =$ “Cat”

$f($  $) =$ “Dog”

Linear Regression

fit:

$$y_i = \hat{b} + \sum_{j=1}^{p} \hat{w}_j \, x_{ij} + \varepsilon_i \qquad \hat{f}(\boldsymbol{x}_i)$$

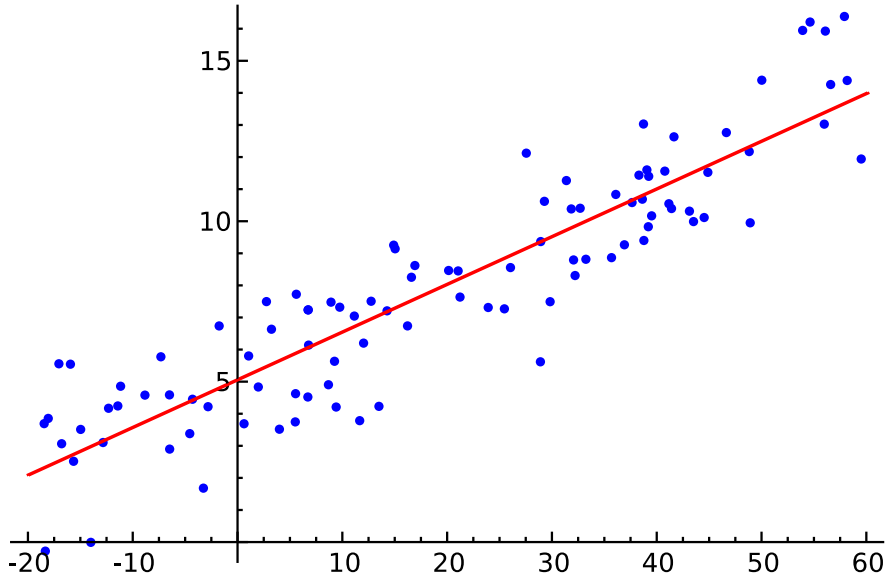
predict:

$$\hat{y}_i = E[Y|\boldsymbol{X} = \boldsymbol{x}_i] = \hat{f}(\boldsymbol{x}_i) = \hat{\alpha} + \sum_{j=1}^{p} \hat{\beta}_j \, x_{ij}$$

Gaussian

$$p(y|\boldsymbol{x}_i) = \mathcal{N}(y; \hat{y}_i, \hat{\sigma}^2)$$

Gaussian mean variance



parameter estimation:
e.g., least squares

parameters to be estimated: $\hat{\alpha}, \hat{\boldsymbol{\beta}}$

$\rightarrow \hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{f}(\boldsymbol{x}_i)\right)^2$

(approximating assumed true $\alpha, \boldsymbol{\beta}, \sigma$)

Brief Notation

data: vector with p different feature values for this example i

$$f(\boldsymbol{x}_i) = \boldsymbol{w} \cdot \boldsymbol{x}_i + b$$

vector with slope parameters (one for each of the p features), aka weights

intercept parameter, aka bias

for simplicity of notation, dropping the hats of estimated parameters from now on

Loss Function

loss function L: expressing deviation between prediction and target

$L\left(y_i, \hat{f}(\boldsymbol{x}_i); \widehat{\boldsymbol{\theta}}\right)$
with $\widehat{\boldsymbol{\theta}}$ corresponding to parameters of model $\hat{f}(\boldsymbol{x})$
e.g., $\hat{\alpha}, \widehat{\boldsymbol{\beta}}$ in linear regression

prime example: squared residuals for regression problems
$L\left(y_i, \hat{f}(\boldsymbol{x}_i); \widehat{\boldsymbol{\theta}}\right) = \left(y_i - \hat{f}\left(\boldsymbol{x}_i; \widehat{\boldsymbol{\theta}}\right)\right)^2$

Cost Function

averaging losses over (empirical) training data set:

$$J(\widehat{\boldsymbol{\theta}}) = \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, \hat{f}(\boldsymbol{x}_i); \widehat{\boldsymbol{\theta}}\right)$$

cost function to be minimized according to model parameters $\widehat{\boldsymbol{\theta}}$

→ objective function

Cost Minimization

minimize training costs $J(\widehat{\boldsymbol{\theta}})$ according to model parameters $\widehat{\boldsymbol{\theta}}$:

$$\nabla_{\widehat{\boldsymbol{\theta}}} J(\widehat{\boldsymbol{\theta}}) = 0$$

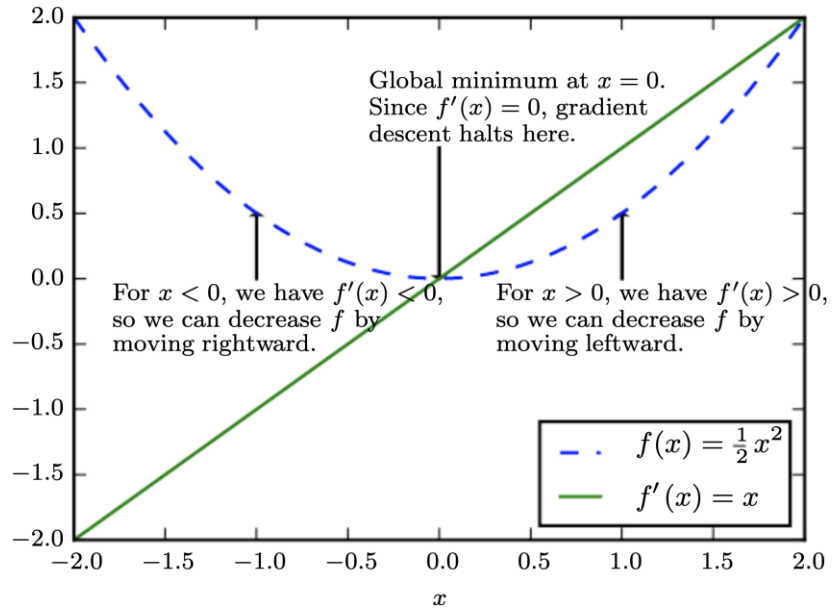
for mean squared error (aka least squares method):

$$\nabla_{\widehat{\boldsymbol{\theta}}} \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{f}(\boldsymbol{x}_i; \widehat{\boldsymbol{\theta}}) \right)^2 = 0$$

Gradient Descent

typically no closed-form solution to minimization of cost function: $\nabla_{\hat{\boldsymbol{\theta}}} J(\hat{\boldsymbol{\theta}}) = 0$

→ need for numerical methods



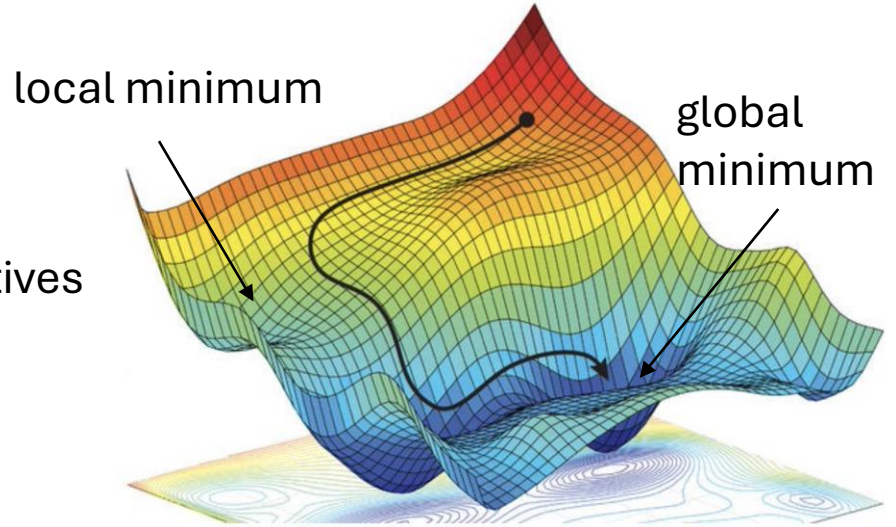
source

popular choice: gradient descent

iteratively moving in direction of steepest descent (negative gradient): $\hat{\boldsymbol{\theta}} \leftarrow \hat{\boldsymbol{\theta}} - \eta \nabla_{\hat{\boldsymbol{\theta}}} J(\hat{\boldsymbol{\theta}})$

step size (learning rate)

vector containing all partial derivatives



L^2 Regularization (Ridge Regression)

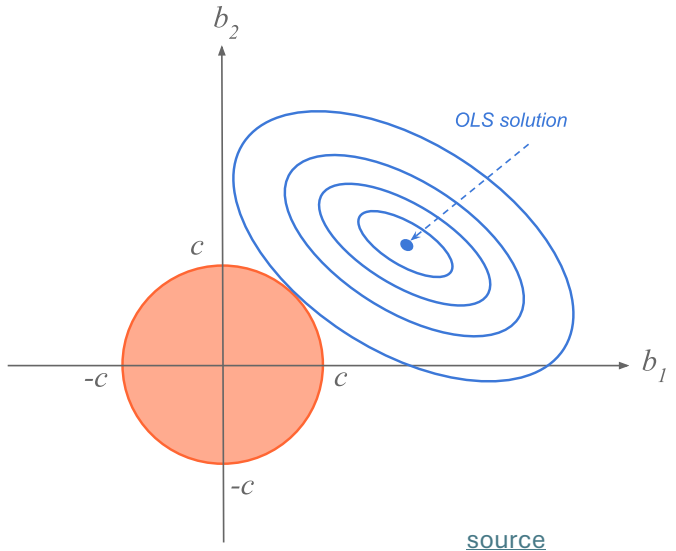
add **penalty term** on parameter L^2 norm to cost function

$$\tilde{J}(\hat{\boldsymbol{\theta}}) = J(\hat{\boldsymbol{\theta}}) + \lambda \cdot \hat{\boldsymbol{\theta}}^{\mathrm{T}} \hat{\boldsymbol{\theta}}$$

hyperparameter controlling strength of regularization

aka **weight decay** in neural networks

corresponds to imposing constraint on parameters to lie in L^2 region (size controlled by λ)



source

Classification: Logistic Regression

binary classification: $y = 1$ or $y = 0$

→ predict probabilities p_i for $y = 1$

- logit (log-odds) as link function
- Y following Bernoulli distribution

$$\text{logit}(E[Y|\boldsymbol{X} = \boldsymbol{x}_i]) = \ln\left(\frac{p_i}{1 - p_i}\right) = \boldsymbol{w} \cdot \boldsymbol{x}_i + b$$

Multi-Classification: Softmax

for classification in K categories:

- use “score” function with K outputs

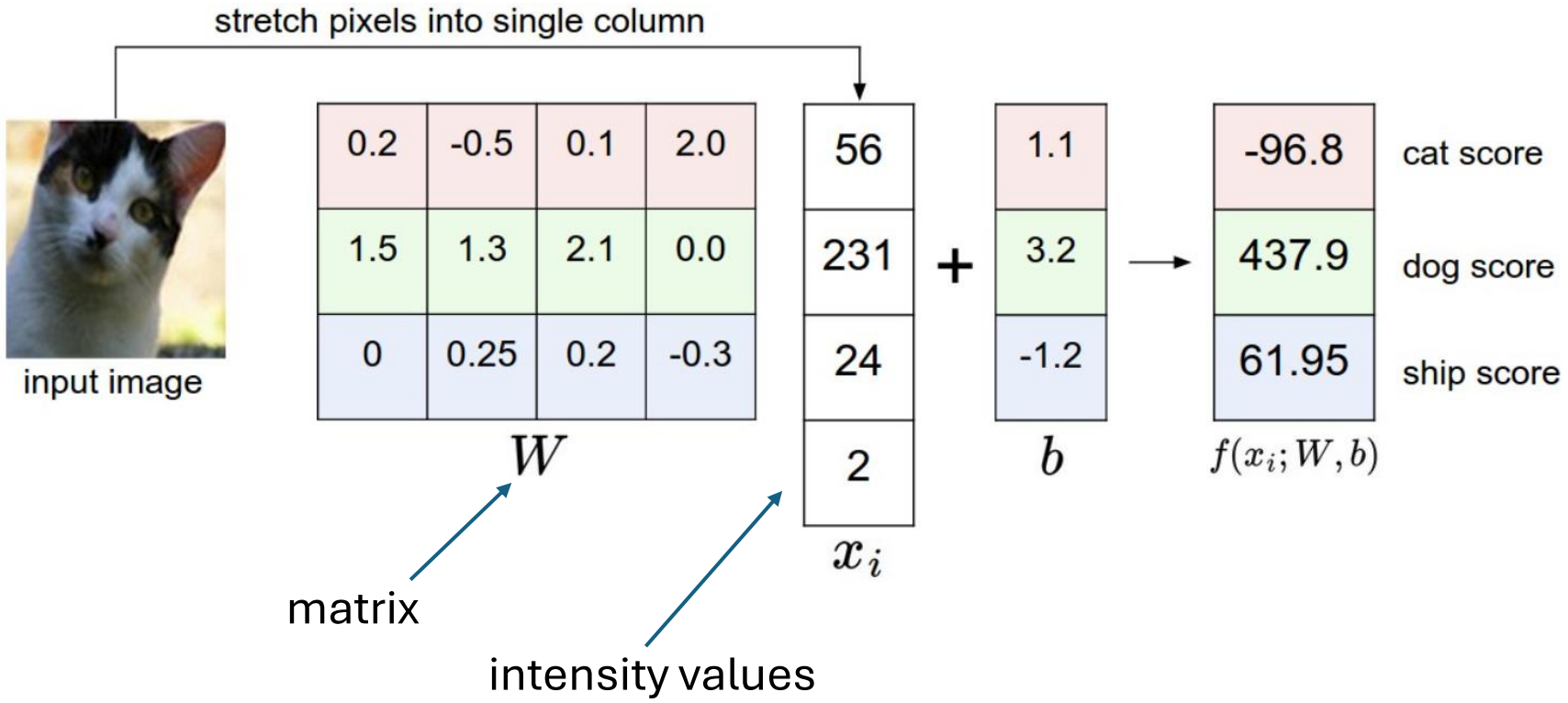
$$f_k(\boldsymbol{x}_i) = \boldsymbol{w}_k \cdot \boldsymbol{x}_i + b_k$$

- and convert into probabilities by means of softmax function

$$\frac{\exp\left(f_k(\boldsymbol{x}_i)\right)}{\sum_{l=1}^{K} \exp\left(f_l(\boldsymbol{x}_i)\right)}$$

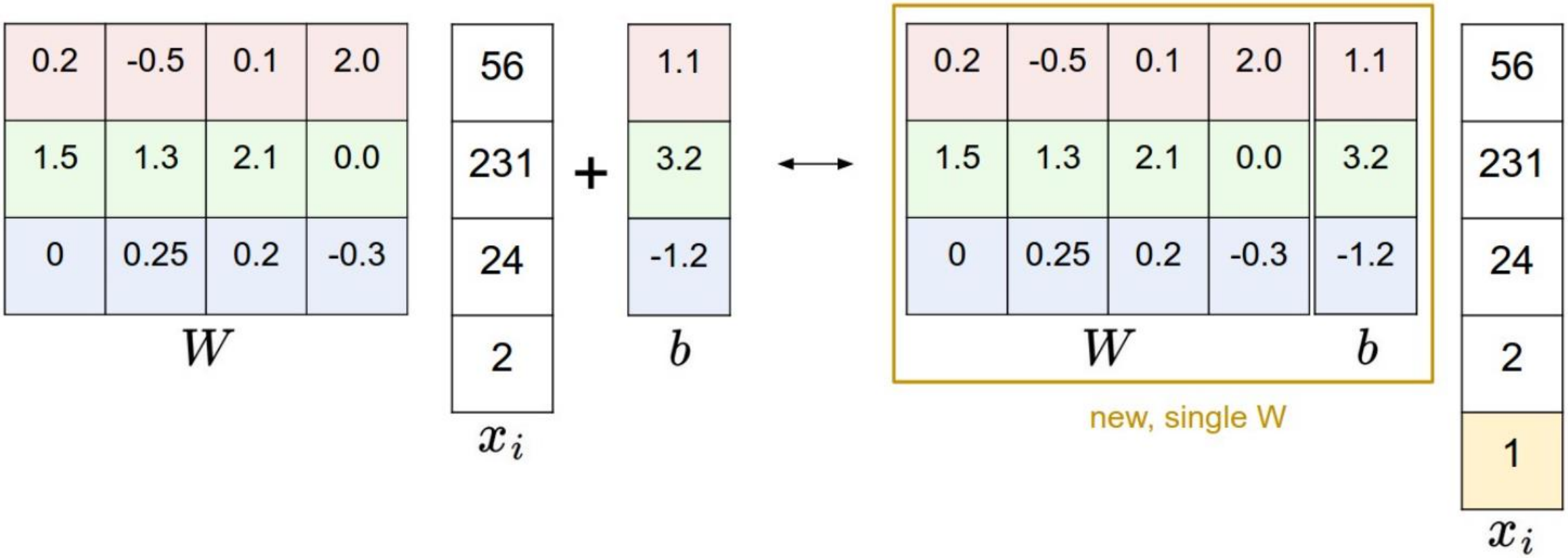
Image Classification with Linear Regression

simplified example: 4-pixel image, 3 classes

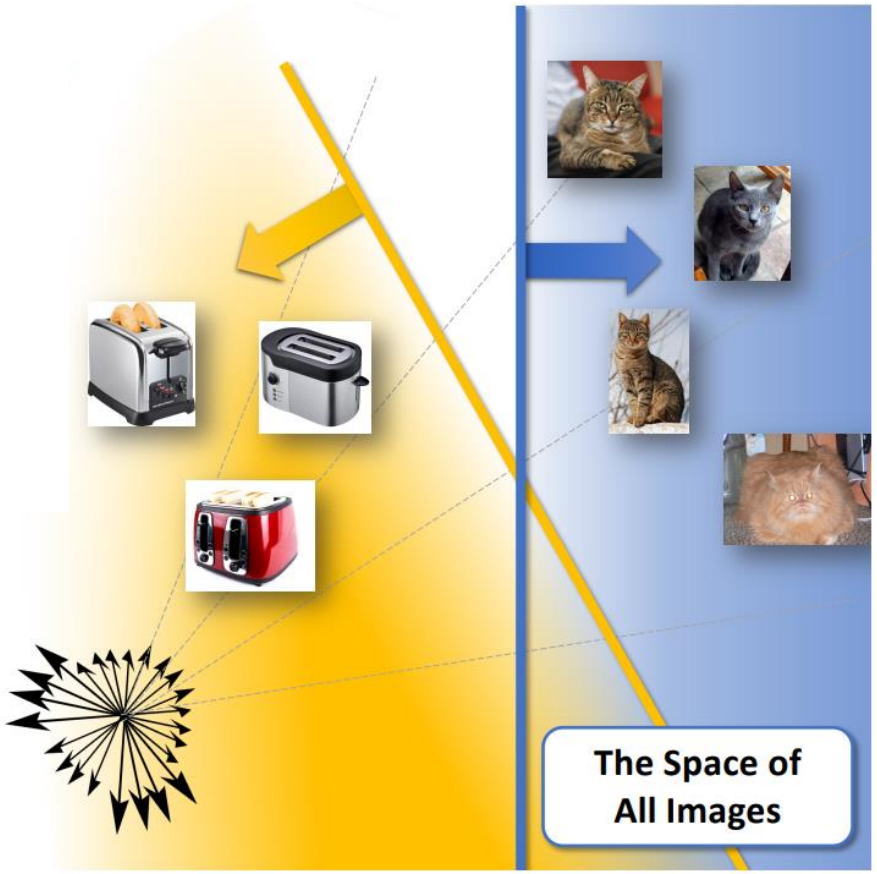


need for one common image size per model

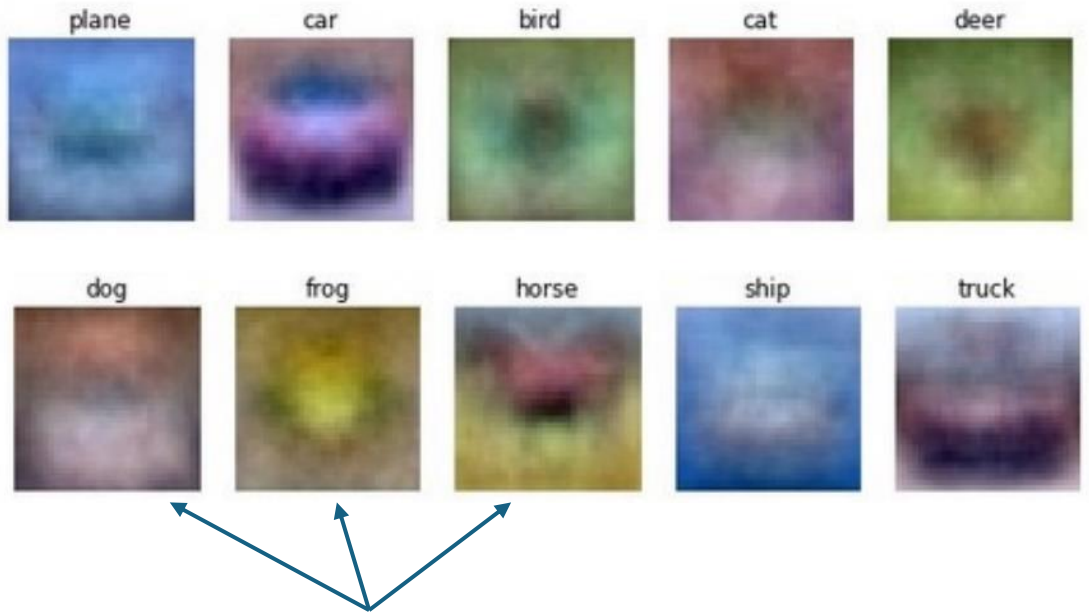
Simplified Matrix Multiplication: Bias Trick



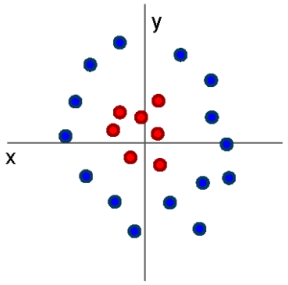
geometric interpretation:
separating hyperplanes



matching interpretation:
generic class templates



different rows in matrix W



Cannot separate red and blue points with linear classifier

raw-pixel images not linearly separable
→ linear model has not enough representational power

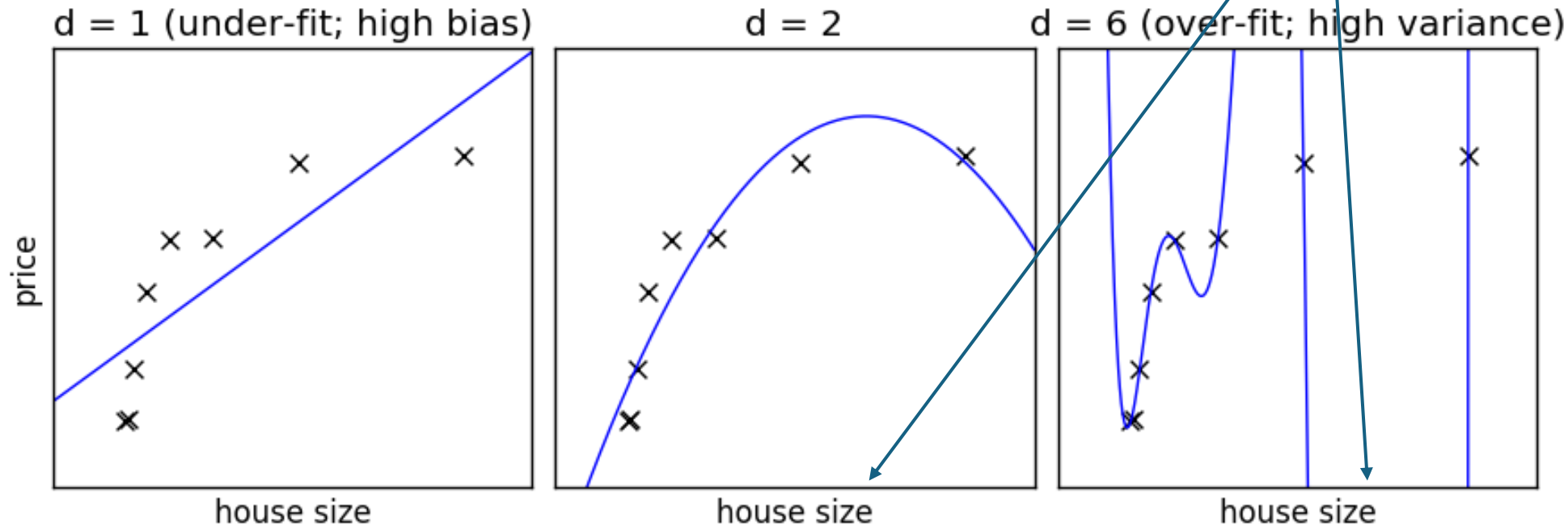
Need for Generalization

Under- and Over-Fitting

example: polynomial fit

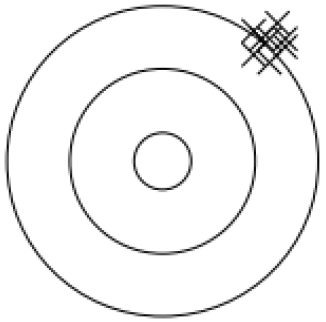
need to predict new samples (not in training data set): interpolation

degree of fitted polynomial

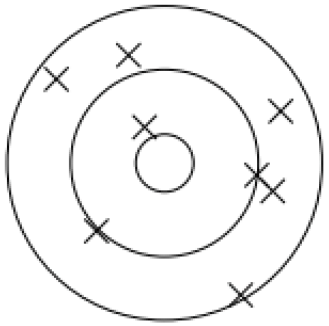


Bias, Variance, Irreducible Error

think of fitting ML algorithms as repeatable processes with different data sets



bias
due to too simplistic model
(same for all training data sets)
“underfitting”



variance
due to sensitivity to specifics (noise)
of different training data sets
“overfitting”

irreducible error (aka Bayes error):

inherent randomness (target generated from random variable)

→limiting accuracy of ideal model

Bias-Variance Tradeoff

models of higher complexity have lower bias but higher variance

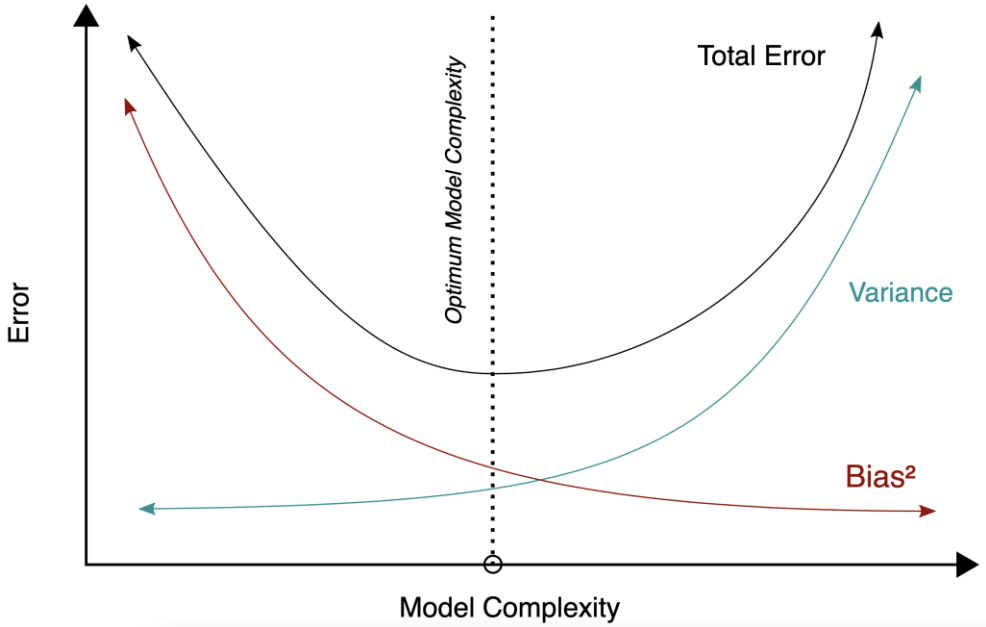
(given the same number of training examples)

generalization error follows U-shaped curve: overfitting once model complexity (number of parameters) passes certain threshold

overfitting: variance term dominating test error

→ increasing model complexity increases test error

but beware: training error keeps decreasing with more complex model …

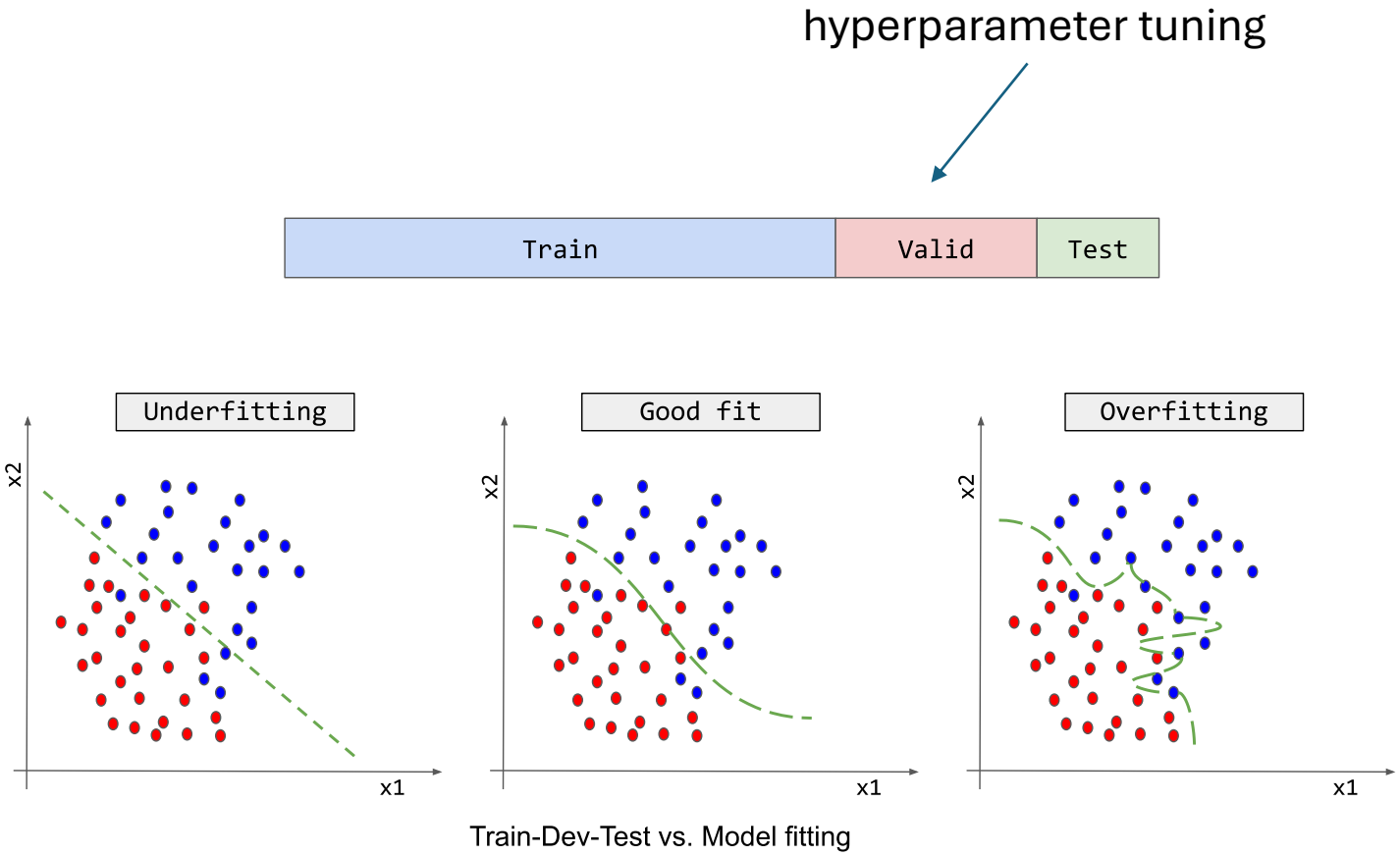


Hyperparameters

model complexity often controlled via hyperparameters (parameters not fitted but set in advance)

examples:

- degree of fitted polynomial d
- number of considered nearest neighbors k
- maximum depth of decision tree
- number of hidden nodes in neural network layer



Train-Dev-Test vs. Model fitting

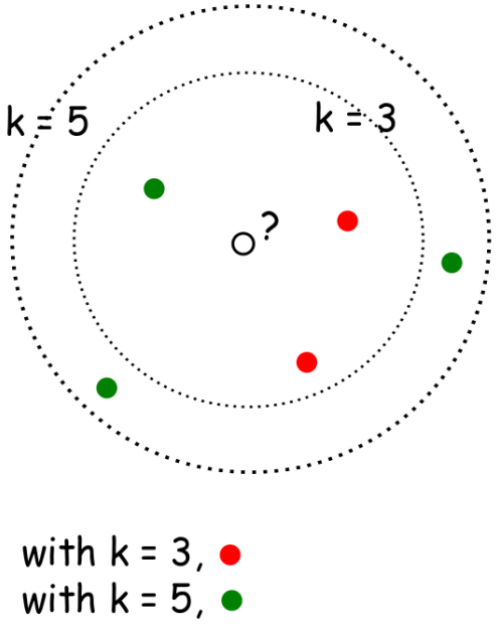
Another Example: k-Nearest Neighbors (kNN)

local method, instance-based learning
non-parametric
distance defined by metric on $\boldsymbol{x}$ (e.g., Euclidean)

regression:

$$\hat{f}(\boldsymbol{x}_0) = \frac{1}{k}\sum_{j=1}^{k} y_j$$

with j running over
k nearest neighbors of $\boldsymbol{x}_0$



low k: low bias but high variance
high k: low variance but high bias

$$bias = f(\boldsymbol{x}) - \frac{1}{k}\sum_{j=1}^{k} y_j$$

$$var = \frac{\sigma^2}{k}$$

Image Classification with kNN

choose an image distance, e.g., L1 distance:

$$d(\boldsymbol{I}_1, \boldsymbol{I}_2) = \sum_p |I_1(p) - I_2(p)|$$

test image

56	32	10	18
90	23	128	133
24	26	178	200
2	0	255	220

-

training image

10	20	24	17
8	10	89	100
12	16	178	170
4	32	233	112

=

pixel-wise absolute value differences

46	12	14	1
82	13	39	33
12	10	0	30
2	32	22	108

→ 456

Image Classification with kNN

training examples CIFAR-10 data set

10 classes

10 nearest neighbors to some test images

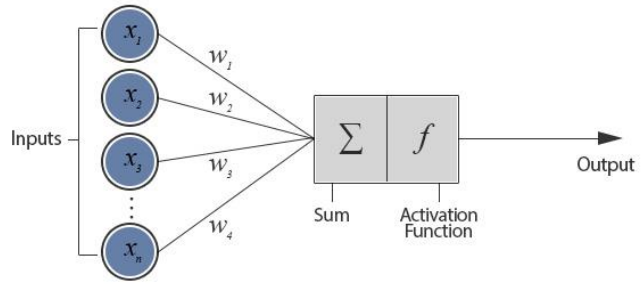
better than random guessing, but also not very impressive

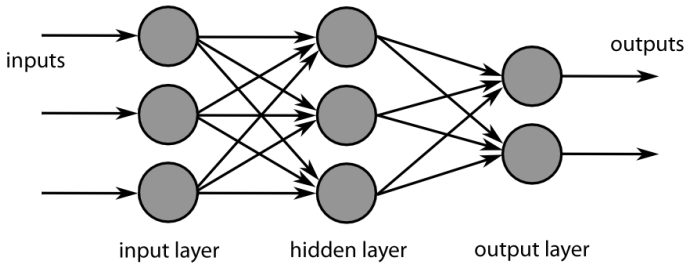
Algorithmic Families of Supervised Learning

parametric models

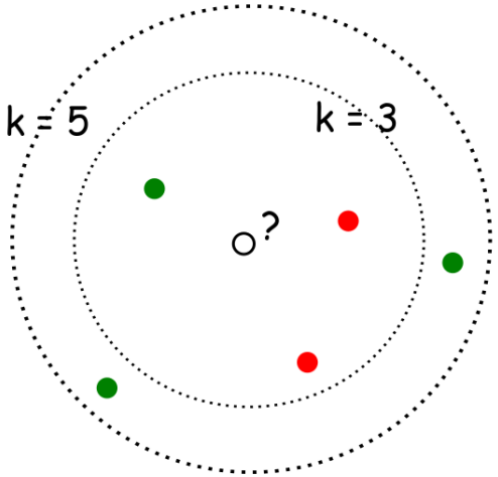
linear regression

neural networks: many linear models, non-linear by means of activation functions





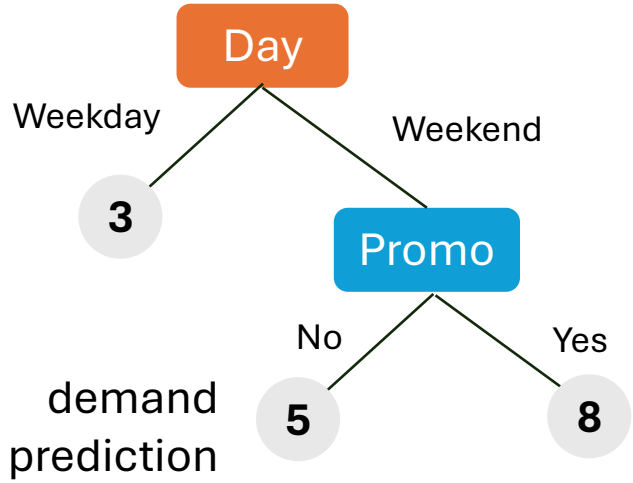
nearest neighbors (local methods, instance-based learning): non-parametric models



with k = 3,
with k = 5,

kernel/support-vector machines: linear model (maximum-margin hyperplane) with kernel trick

decision trees: rule learning



often used in ensemble methods

- bagging: random forests
- boosting: gradient boosting

Common Idea: Learning from Data

ML algorithm + data = explicit algorithm

→ much better generalizability than handcrafted algorithms

unfortunately, no general best ML algorithm:
need to pick right method for the task at hand

Generalization

core of ML: **empirical risk minimization** (training error) as proxy for minimizing unknown population risk (test error)

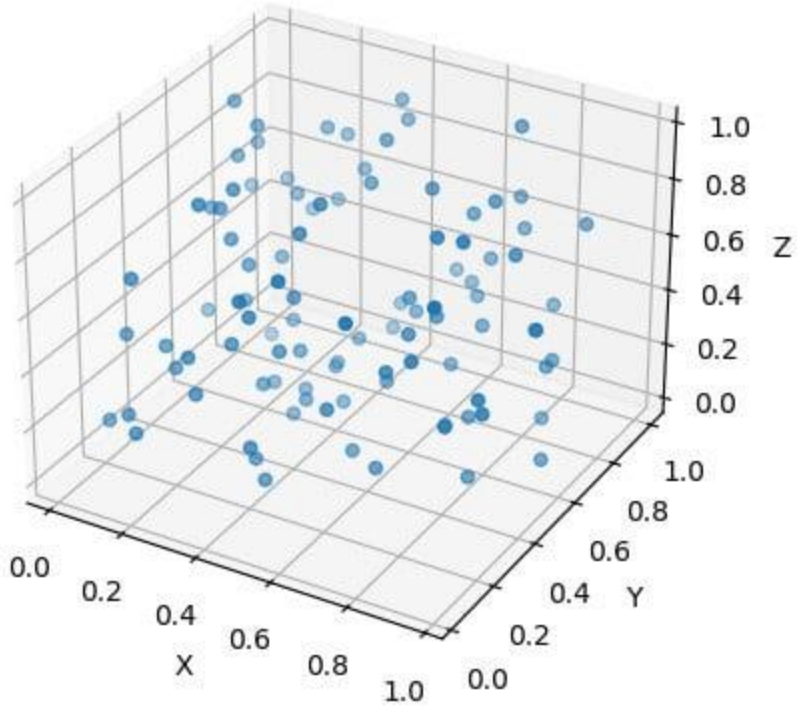
generalization gap: difference between test and training error

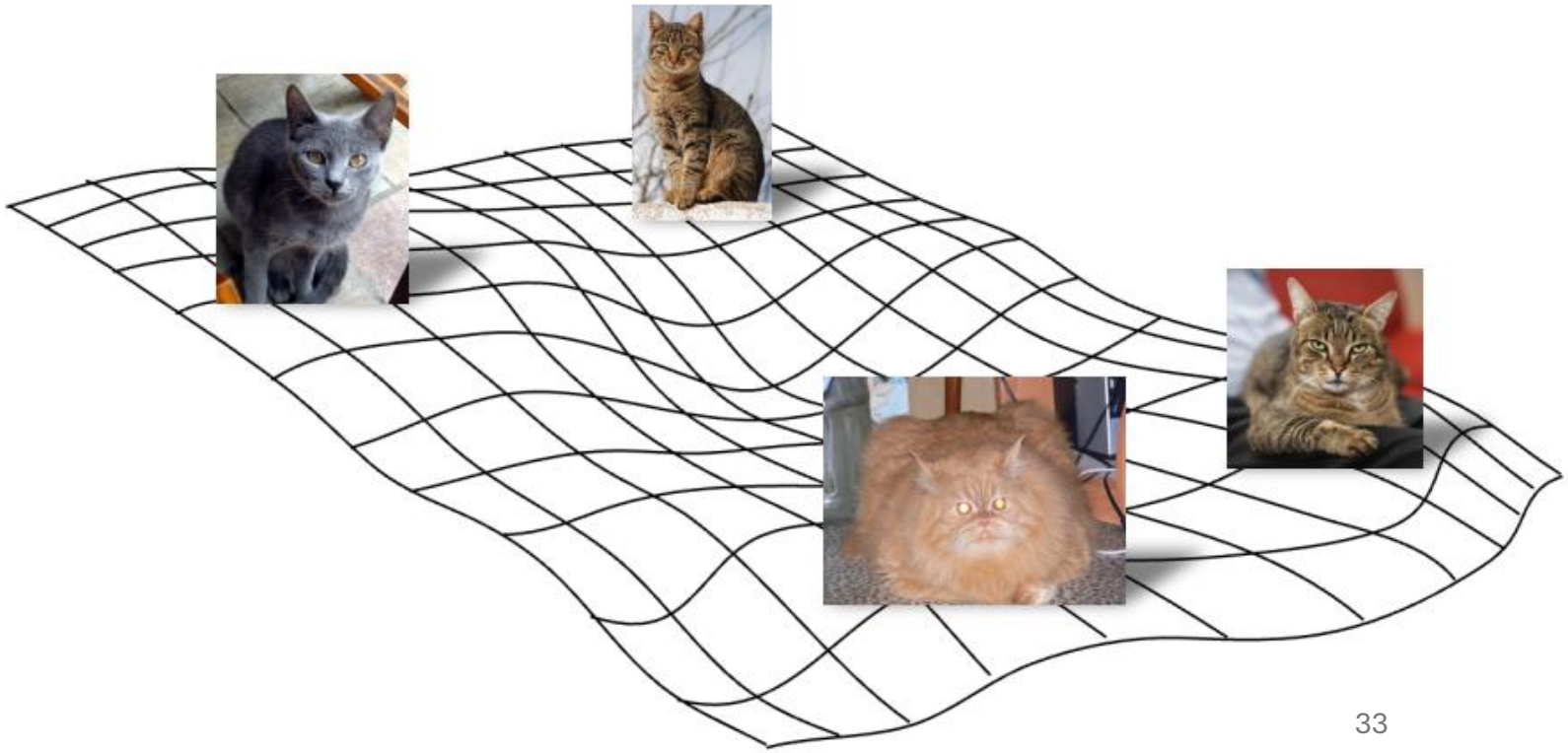
curse of dimensionality: typically, many features (dimensions)

→ lots of data needed to densely sample volume

Manifold Hypothesis

luckily, reality is friendly: most high-dimensional data sets reside on lower-dimensional manifolds → enabling effectiveness of ML





Which Features for Image Classification?

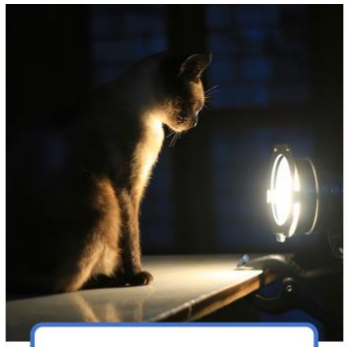
linear regression & kNN (same for tree-based methods):

learning directly from raw pixel intensities does not work great

→ try learning from pre-extracted features, such as HOG or SIFT

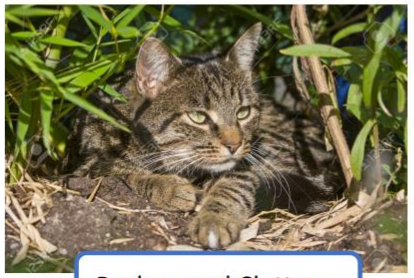
challenges:

Viewpoint Variation

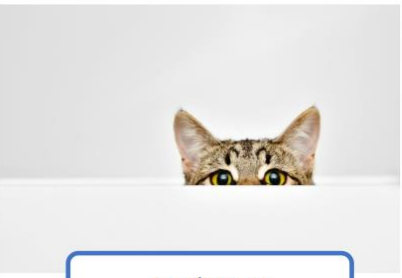


Lighting Variation

Deformation



Background Clutter

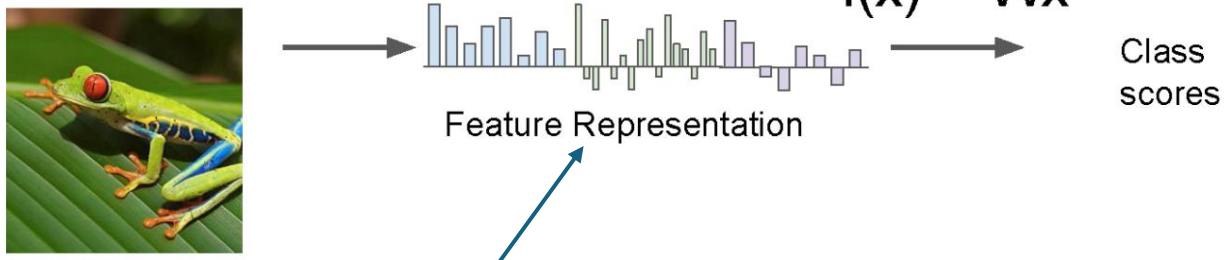


Occlusion

Global Features in Classic ML Method

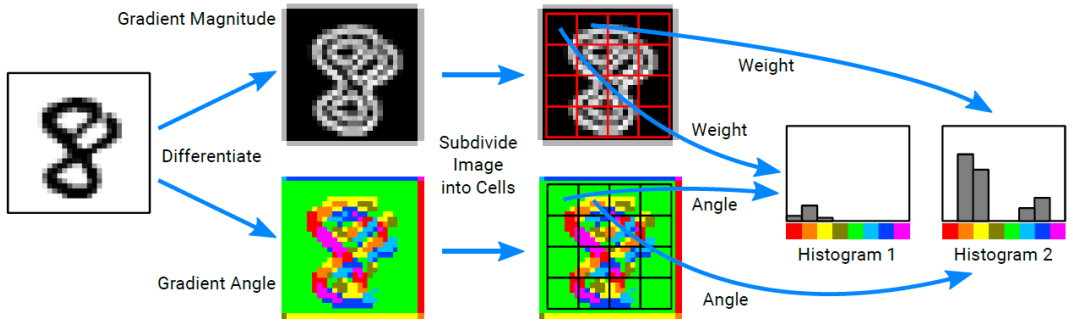
features covering entire image
(instead of only local patches)

or random forest, support-vector machine, ...

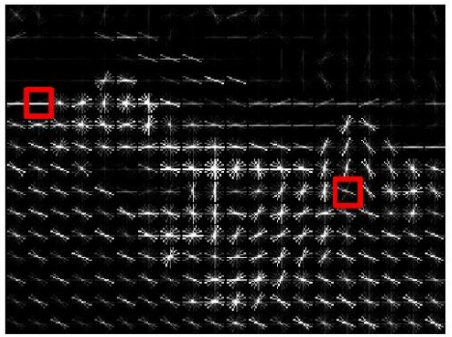


easier to separate than raw pixels

often used: HOG



Divide image into 8x8 pixel regions
Within each region quantize edge direction into 9 bins



Example: 320x240 image gets divided into 40x30 bins; in each bin there are 9 numbers so feature vector has 30*40*9 = 10,800 numbers

Lowe, "Object recognition from local scale-invariant features", ICCV 1999
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

important disadvantage: not translation invariant (due to ordering of patches)

Local Features in Classic ML Method

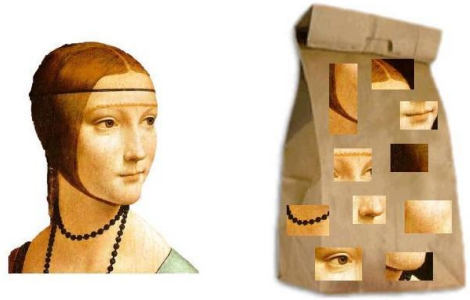
local features like SIFT do not cover the entire image

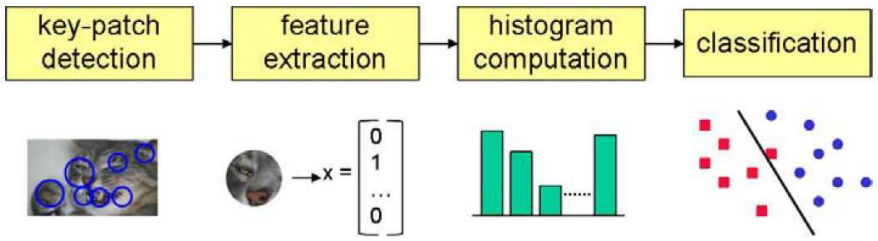
→ need for some processing to identify features in different images (to be able to compare different images with ML)

in HOG this is done by ordering of patches making up the complete image

one popular approach, called bag-of-words model (as it is orderless):

1. learn clustering of SIFT vectors (e.g., with K-means)
2. quantization of different clusters into visual “vocabulary” (see embeddings in language models)
3. create (sparse) histogram of visual “word” occurrences
4. train ML model (e.g., random forest) with histogram bins as features





important disadvantage: ignores spatial relationships among different patches

Need for Feature Learning

many hand-designed components in approach using pre-extracted features → poor generalization

Is there maybe some way after all to learn features end-to-end from raw pixel intensities?

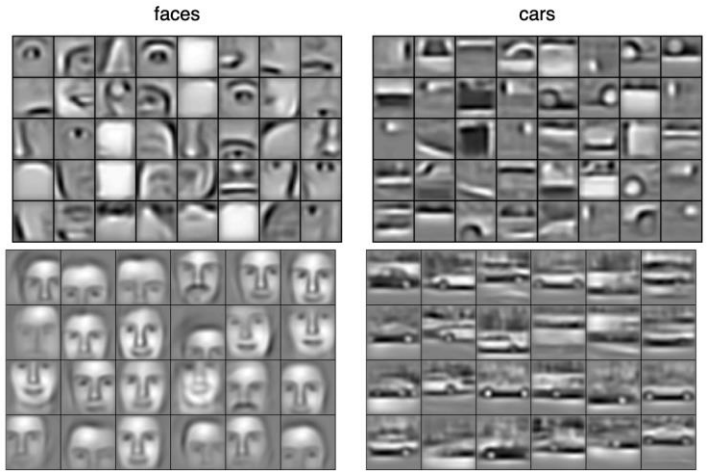
Deep Learning

Ladder of Generalization

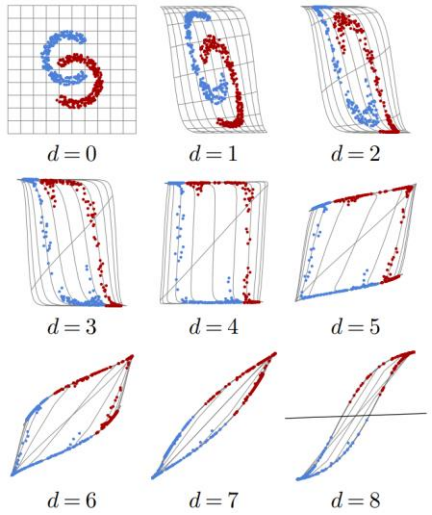
classic ML: feature engineering

deep learning: feature learning

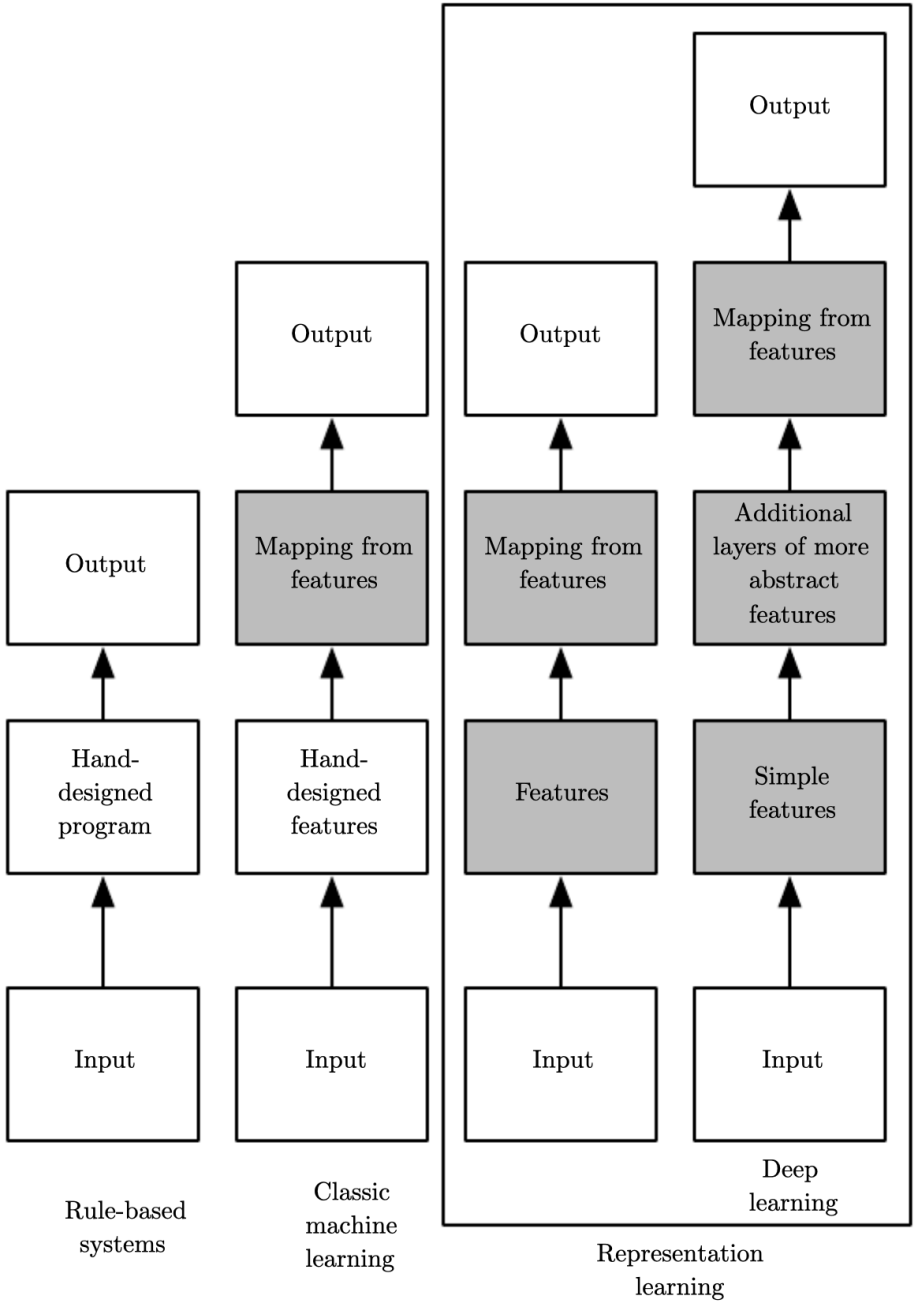
(hierarchy of concepts learned from raw data in deep graph with many layers)



source



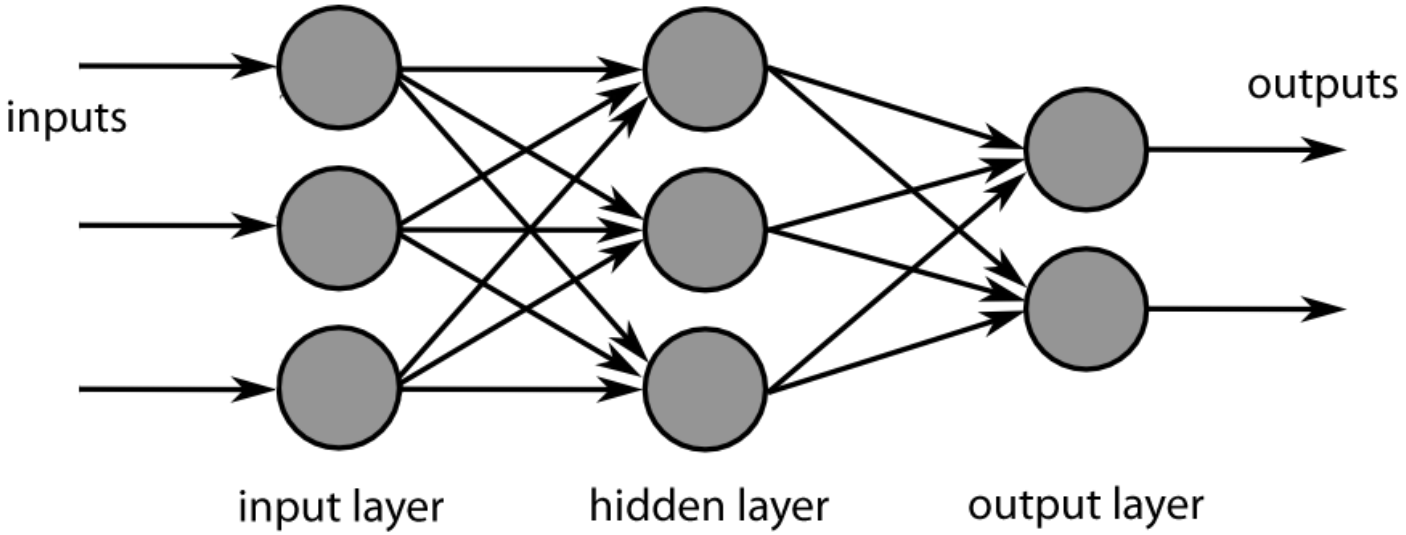
source



source

Neural Networks

idea: powerful algorithm by combining many linear building blocks



each node exchanges information only with directly connected nodes
→ parallel computation

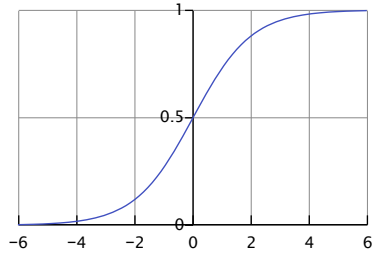
Artificial Neuron

linear model with parameters called weights $\boldsymbol{w}$ (including bias, not shown for simplicity)

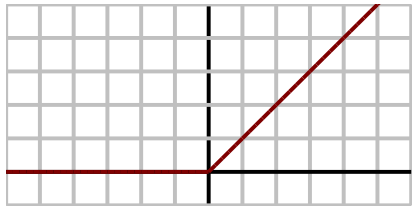
non-linear via (differentiable) activation function on top of linear model

examples:

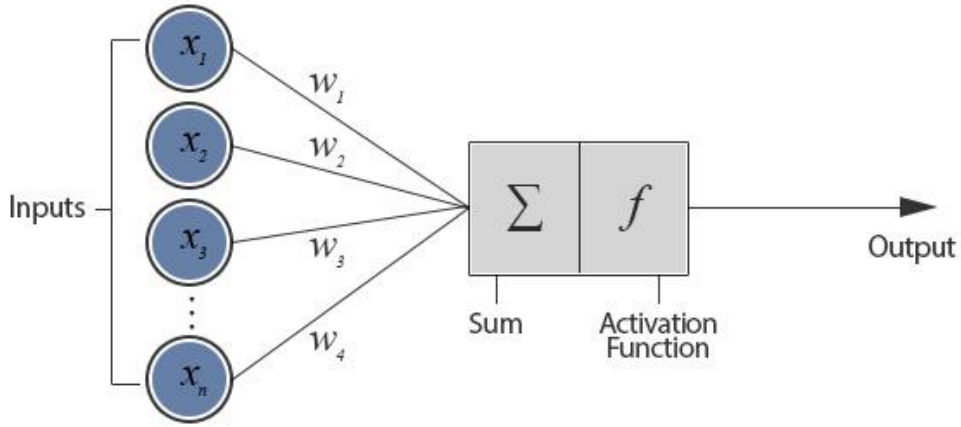
- sigmoid



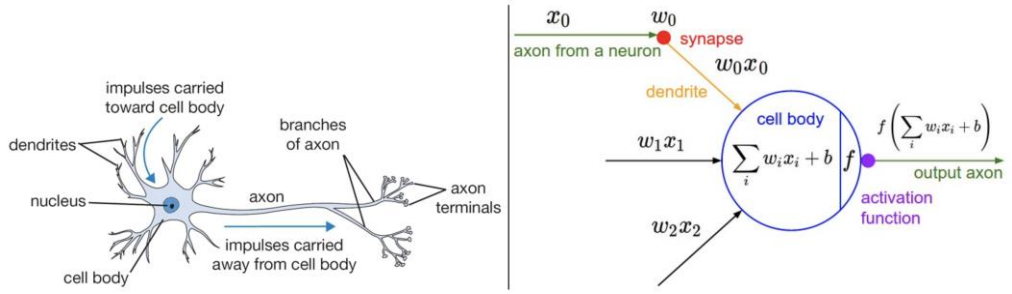
- ReLU (Rectified Linear Unit)



artificial neuron (node in neural network):



inspired from biological neurons:

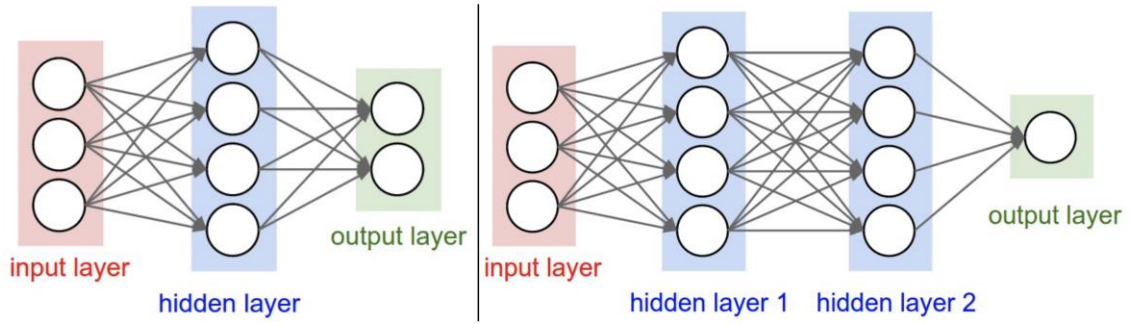


source

Multi-Layer Perceptron (MLP)

fully-connected feed-forward network with at least one hidden layer (universal function approximator)

binary classification



regression

toward deep learning: add hidden layers

more layers (depth) more efficient than just more nodes (width): less parameters needed for same function complexity

Classification Networks

cross-entropy loss:

$$L_i\left(y_i, \hat{f}(\boldsymbol{x}_i); \widehat{\boldsymbol{w}}\right) = -\sum_{k=1}^{K} y_{ik} \log \hat{f}_k(\boldsymbol{x}_i; \widehat{\boldsymbol{w}})$$

one output node for each class k

softmax on outputs $\boldsymbol{t}$ of final-layer nodes:

$$g_k(\boldsymbol{t}_i) = \frac{e^{t_{ik}}}{\sum_{l=1}^{K} e^{t_{il}}}$$

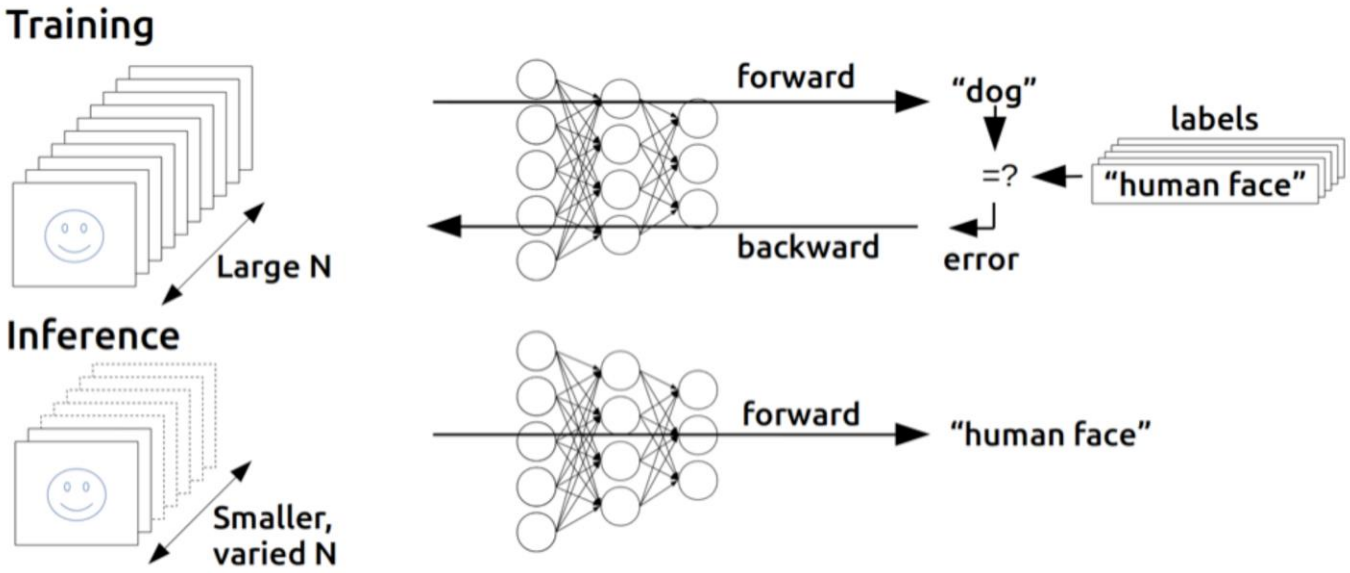
regression networks:

- squared error loss
- just one output node
- identity output function

Image Classification: Lots of Categories

Find Gradients for (Deep) Neural Networks

backpropagation of errors through network layers via chain rule of calculus (enables learning of deep neural networks)



source

forward pass:

- fix current weights
- compute predictions

backward pass:

- compute errors
- calculate gradients (backpropagation of errors)
- update weights accordingly (gradient descent)

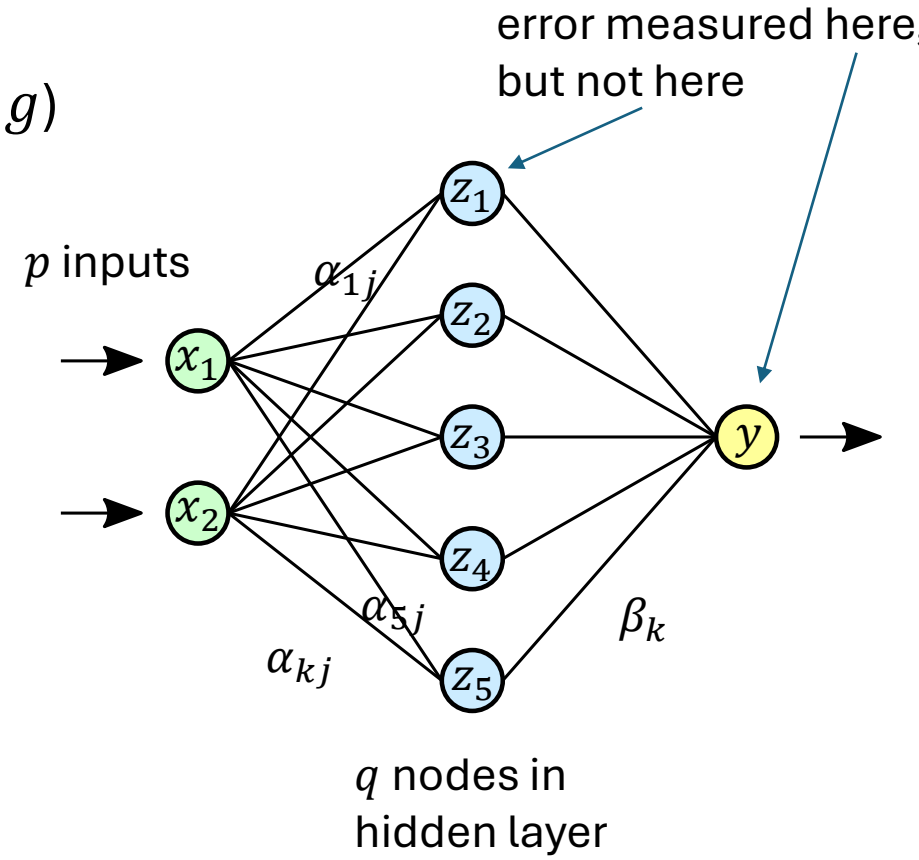
Example WOLOG

- regression (squared error loss, identity output function g)
- with one hidden layer ($\widehat{\boldsymbol{w}}$: $\widehat{\boldsymbol{\alpha}}$, $\widehat{\boldsymbol{\beta}}$)

$$\hat{y}_i = \hat{f}(\boldsymbol{x}_i; \widehat{\boldsymbol{w}}) = g(\boldsymbol{z}_i; \widehat{\boldsymbol{\beta}}) = \sum_{k=0}^{q} \hat{\beta}_k z_{ik}$$

$$z_{ik} = h(\boldsymbol{x}_i; \widehat{\boldsymbol{\alpha}}_k) = h\left(\sum_{j=0}^{p} \hat{\alpha}_{kj} x_{ij}\right)$$

activation function



cost function:

$$J(\widehat{\boldsymbol{w}}) = \sum_{i=1}^{n} L_i\left(y_i, \hat{f}(\boldsymbol{x}_i); \widehat{\boldsymbol{w}}\right) = \sum_{i=1}^{n} \left(y_i - \hat{f}\left(\boldsymbol{x}_i; \widehat{\boldsymbol{\alpha}}, \widehat{\boldsymbol{\beta}}\right)\right)^2$$

Example WOLOG

gradients:

$$\frac{\partial L_i}{\partial \hat{\beta}_k} = -2\left(y_i - \hat{f}(\boldsymbol{x}_i; \hat{\boldsymbol{w}})\right) z_{ik} = \delta_i z_{ik}$$

$$\frac{\partial L_i}{\partial \hat{\alpha}_{kj}} = -2\left(y_i - \hat{f}(\boldsymbol{x}_i; \hat{\boldsymbol{w}})\right) \hat{\beta}_k h_k' \left(\sum_{j=0}^{p} \hat{\alpha}_{kj} x_{ij}\right) x_{ij} = s_{ik} x_{ij}$$

backpropagation equations: $s_{ik} = h_k' \left(\sum_{j=0}^{p} \hat{\alpha}_{kj} x_{ij}\right) \hat{\beta}_k \delta_i$

use errors of later layers to calculate errors of earlier ones (avoiding redundant calculations of intermediate terms)

Using Gradients for Iterative Learning

use gradients found via backpropagation to iteratively update weights (gradient descent):

$$\hat{\beta}_k^{(r+1)} = \hat{\beta}_k^{(r)} - \eta_r \sum_{i=1}^{n} \frac{\partial L_i}{\partial \hat{\beta}_k^{(r)}} \qquad \hat{\alpha}_{kj}^{(r+1)} = \hat{\alpha}_{kj}^{(r)} - \eta_r \sum_{i=1}^{n} \frac{\partial L_i}{\partial \hat{\alpha}_{kj}^{(r)}}$$

- adaptive learning rate η_r: learning rate often adjusted per iteration
- weight initialization: choose small random weights as starting values to break symmetry (typically using some heuristics)

Stochastic Gradient Descent (SGD)

updates $\widehat{\boldsymbol{w}} \leftarrow \widehat{\boldsymbol{w}} - \eta \nabla_{\widehat{\boldsymbol{w}}} J(\widehat{\boldsymbol{w}})$ can be done with whole training data set (n observations), aka batch, or small random sample:

- $J(\widehat{\boldsymbol{w}}) = \frac{1}{n}\sum_{i=1}^{n} J_i(\widehat{\boldsymbol{w}})$ batch (or deterministic) gradient descent
- $J(\widehat{\boldsymbol{w}}) = J_i(\widehat{\boldsymbol{w}})$ stochastic gradient descent (single example)
- $J(\widehat{\boldsymbol{w}}) = \frac{1}{m}\sum_{i=1}^{m} J_i(\widehat{\boldsymbol{w}})$ mini-batch stochastic gradient descent (size m)

another hyperparameter: mini-batch size (tradeoff between runtime, memory, …)

SGD has implicit regularization effect, and its random fluctuations help to escape saddle points

The Art of Network Training

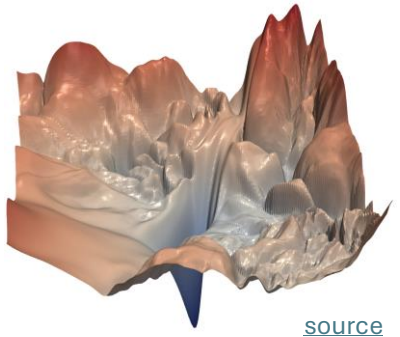
How to Train Deep Neural Networks Effectively?

optimization and regularization difficult:

- non-convex optimization problem (e.g., local vs global minima, saddle points), easily overfitting
- many hyperparameters to tune

but many methods to get it working in practice (despite partly patchy theoretical understanding)

typical loss surface:



source

optimization

- activation and loss functions
- weight initialization
- stochastic gradient descent
- adaptive learning rate
- batch normalization

explicit regularization

- weight decay
- dropout
- data augmentation
- weight sharing

implicit regularization

- early stopping
- batch normalization
- stochastic gradient descent

Gradient Descent with Momentum

algorithm:

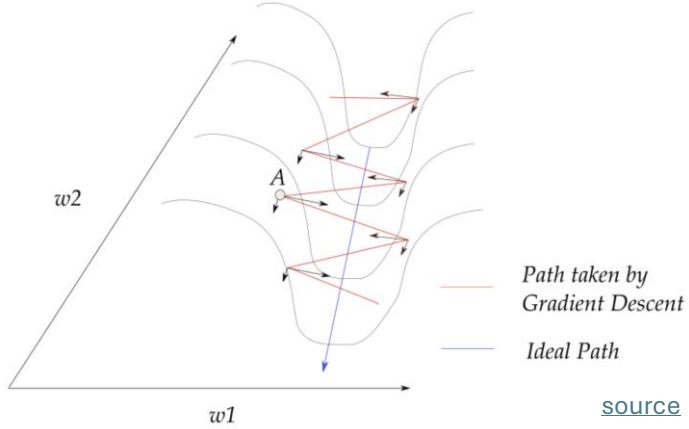
- estimate gradient $\boldsymbol{g}$
- compute velocity update: $\boldsymbol{v} \leftarrow \alpha\, \boldsymbol{v} - \eta\, \boldsymbol{g}$
- apply parameter update: $\widehat{\boldsymbol{\theta}} \leftarrow \widehat{\boldsymbol{\theta}} + \boldsymbol{v}$

hyperparameter ($0 < \alpha < 1$) specifying exponential decay (potentially adaptive)

velocity: parameter movement through search space, set to an exponentially decaying average of past negative gradients

→ no bouncing around of parameters

Adaptive Learning Rate



source

strategies for gradient descent learning rate: constant, decaying, with momentum (escape from local minima and saddle points)

better convergence by adapting learning rate for each weight: lower/higher learning rates for weights with large/small updates

popular methods ($g_{\hat{w}}$ denotes component of gradient for individual weight $\hat{w}$):

- Adagrad: $\hat{w} \leftarrow \hat{w} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_{\hat{w},\tau}^2}} g_{\hat{w}}$ with t, τ denoting current and past iterations

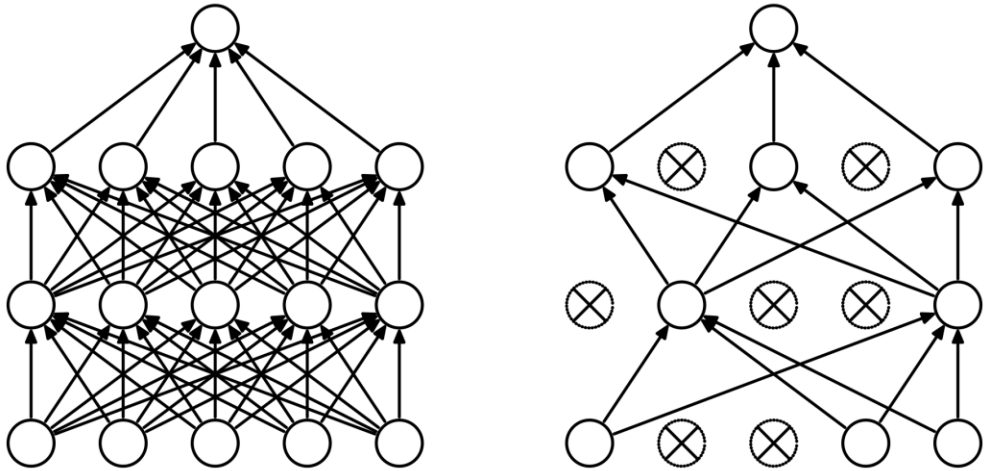
  (issue: sum in denominator grows with more iterations → can get stuck)
- RMSProp: $\hat{w} \leftarrow \hat{w} - \frac{\eta}{\sqrt{v(\hat{w})}} g_{\hat{w}}$ with $v(\hat{w}) \leftarrow \gamma\, v(\hat{w}) + (1-\gamma) g_{\hat{w}}^2$
- Adam (Adaptive Moment Optimization): combines RMSProp with momentum

Dropout in Neural Networks

goal: prevent overfitting of large neural networks

randomly drop non-output nodes (along with their connections) during training (not prediction)

→ adaptability: regularizing each hidden node to perform well regardless of which other hidden nodes are in the model



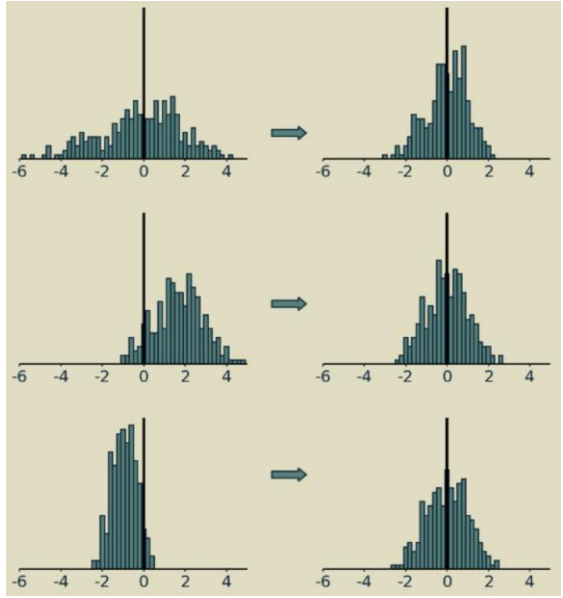
(a) Standard Neural Net (b) After applying dropout.

source

- for each mini-batch, randomly sample independent binary masks for the nodes
- destroying extracted features rather than input values

dropout for 2D structures (such as images) usually drops entire channels instead of individual nodes (because locality nullifies the effect of standard dropout)

Batch Normalization



source

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

source

adaptive reparameterization of inputs to network layer (before or after activation)

independently for each input/feature

(not to confuse with weight normalization)

to maintain expressive power (optional): introduce parameters γ, β (learned together with weights via backpropagation)

Benefits from Batch Normalization

- allows higher learning rates
- reduces importance of weight initialization
- alleviates vanishing/exploding gradients
- (implicit) regularization effect: introducing noise

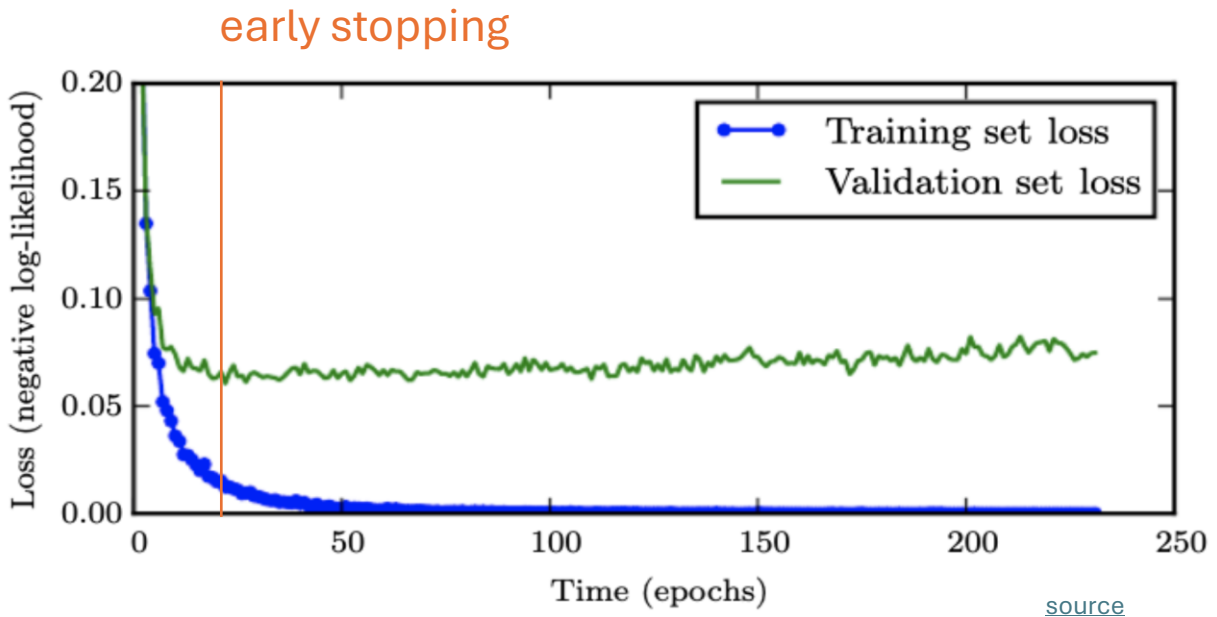
reason why batch normalization improves optimization still controversial

most plausible explanation: smoothening of loss landscape

Early Stopping

loss independently measured on validation set

halting training when overfitting begins to occur

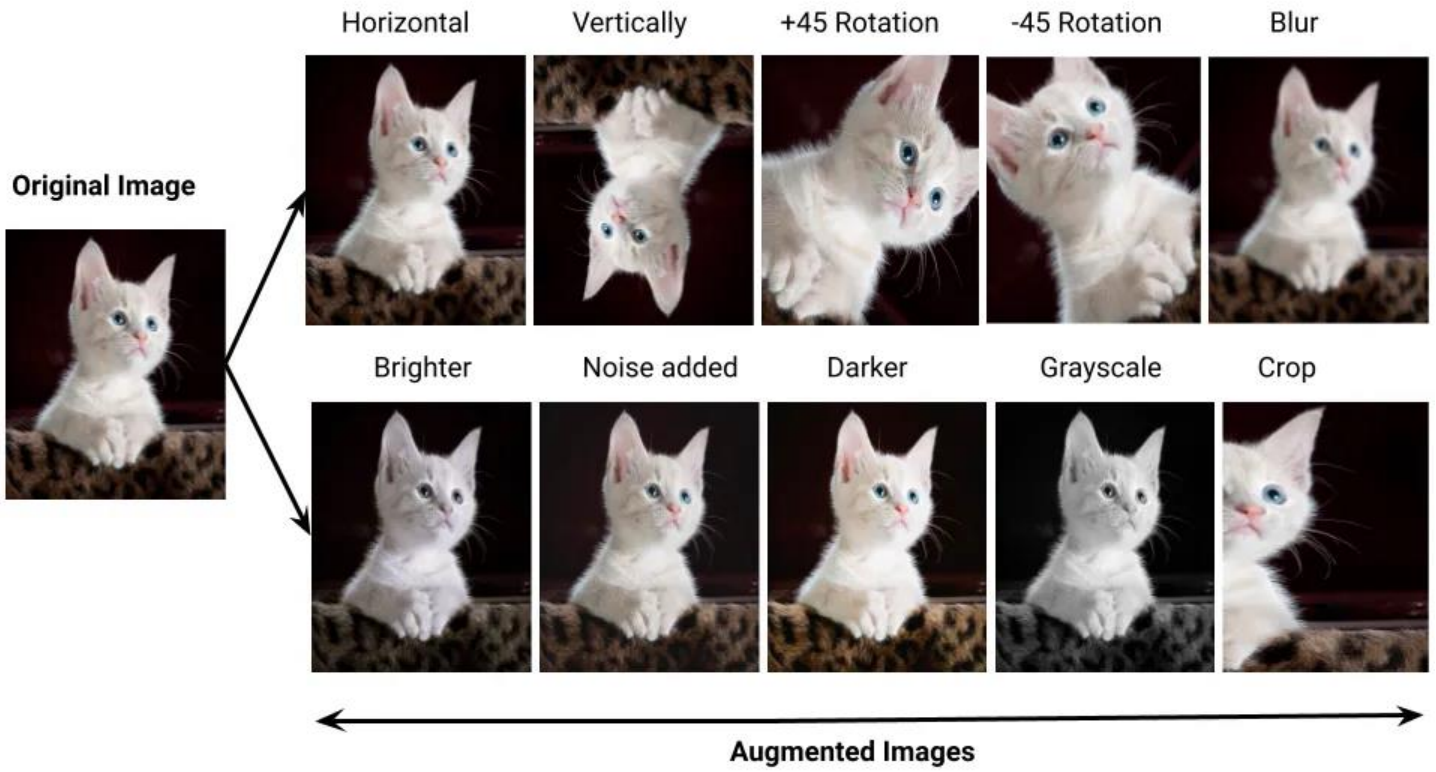


source

Data Augmentation

adding different variations of input data to training

idea: supporting the model in maintaining desired invariances



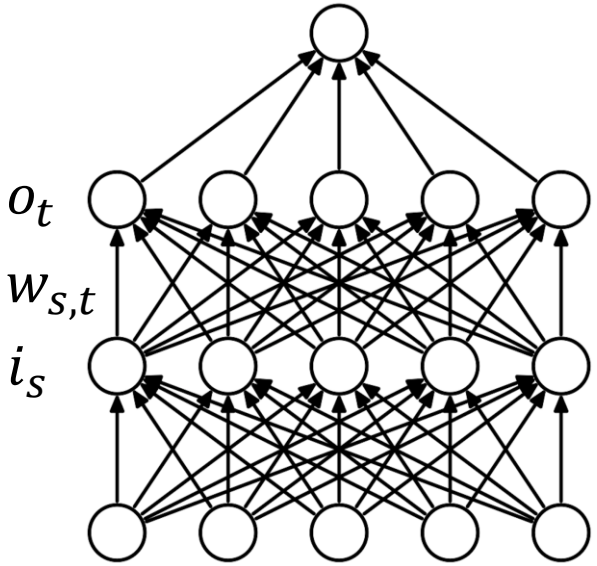
Convolutional Neural Networks (CNN or ConvNet)

Recap: Feed-Forward Neural Networks

computation in usual feed-forward network:

matrix multiplication of scalar inputs and weights

$$o_t = \sum_s w_{s,t}\, i_s$$



dropped dimension of different training observations (in mini-batch) in this view

Inductive Bias

plain feed-forward networks very inefficient for image classification:

make no use of spatial structure (locality of objects)

→ need to learn it from scratch

better way: introduce it right in the architecture of the ML method (called inductive bias)

→ convolution with spatial filters (**learned** rather than handcrafted)

Grid-Like Data

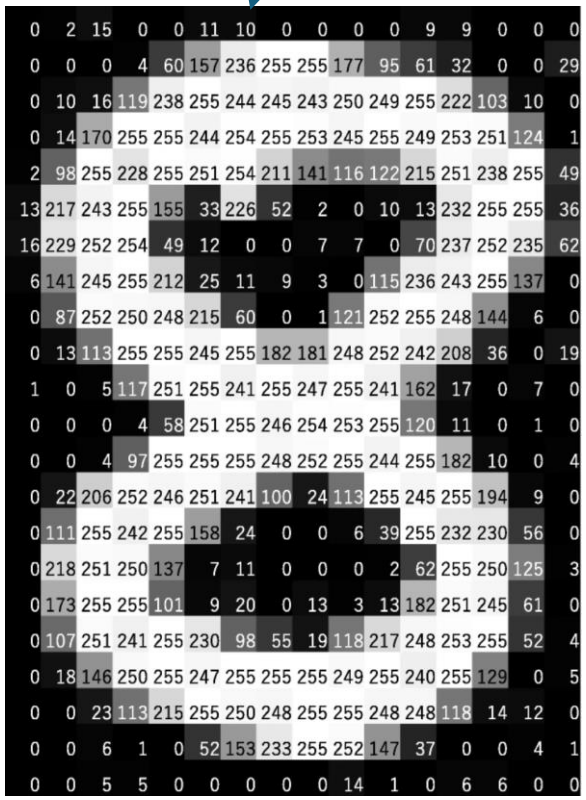
image data: 2-D grid of pixels (spatial structure)

convolutional networks:

neural networks using learned convolution kernels as weights (in place of general matrix multiplications)

→ highly regularized feed-forward networks

scalar values (like in usual feed-forward network)



source

Convolution Operation

actually, correlation (convolution would have – here):

$$o_{x,y} = \sum_{s,t} w_{s,t}\, i_{x+s,y+t}$$

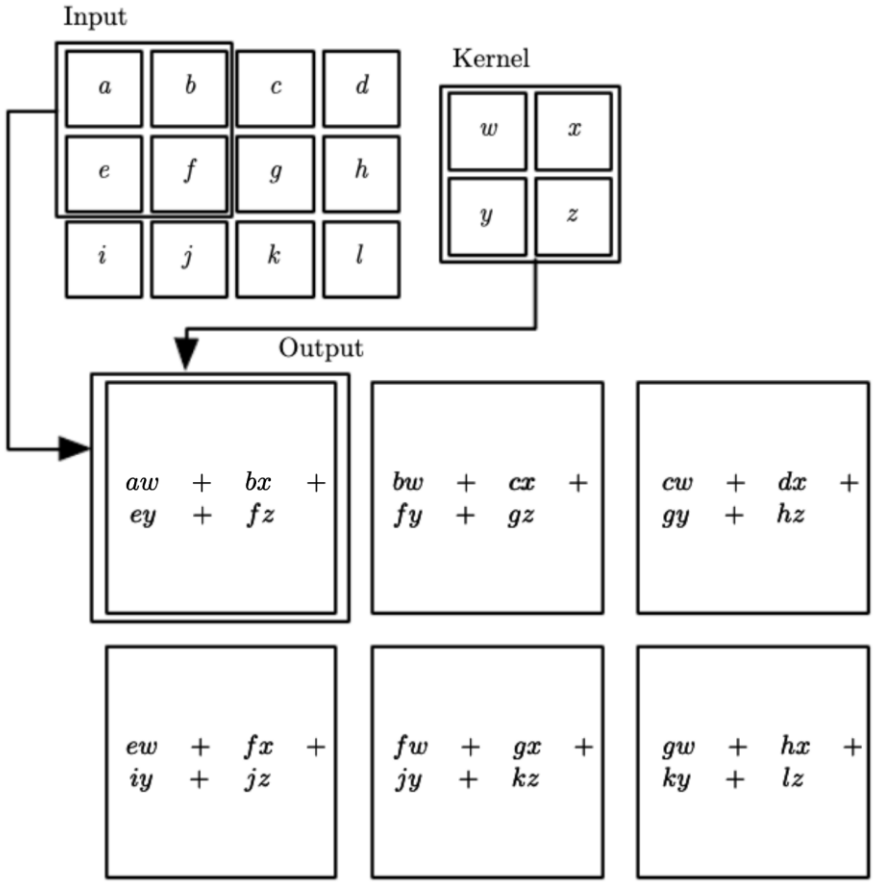
feature map
(transformed image)

kernel
(learned)

image
(input or feature map)

(again, dropping dimension of different training observations)

2D convolution

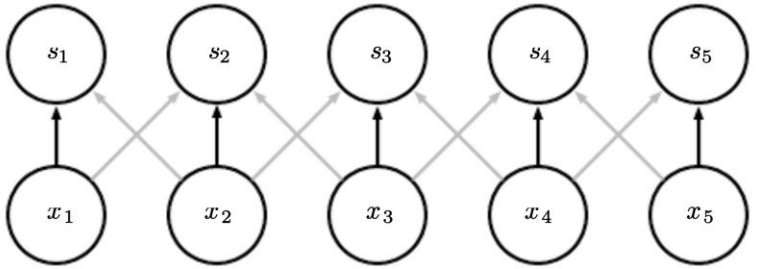


source

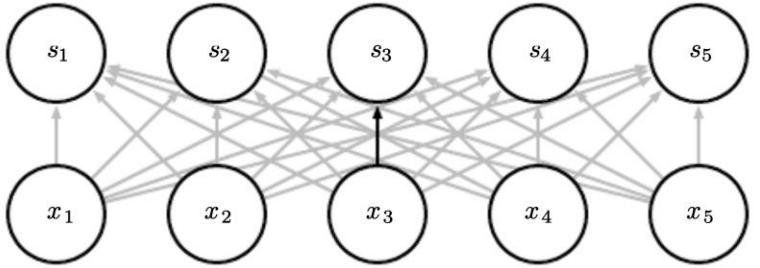
Regularization Effects

- sparse interactions: much less weights
- parameter sharing: use same weights for different connections

convolutional layer:



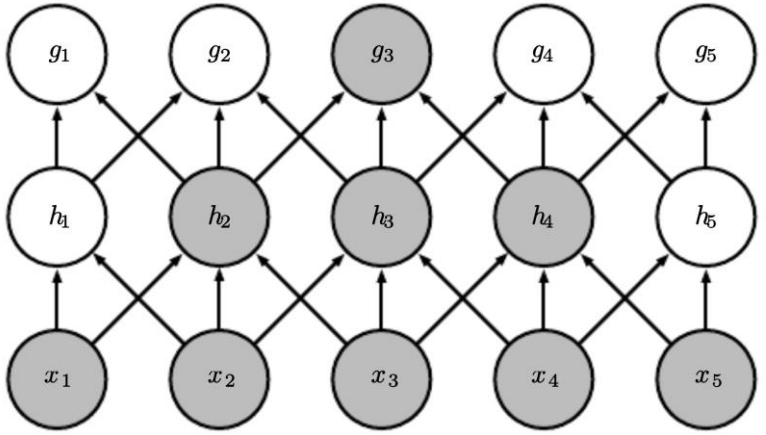
fully-connected layer:



effect of receptive field over several layers:

- consider only locally restricted number of input values from previous layer
- grows for earlier layers (indirect interactions)

→ hierarchical patterns from simple building blocks (many aspects of nature hierarchical)

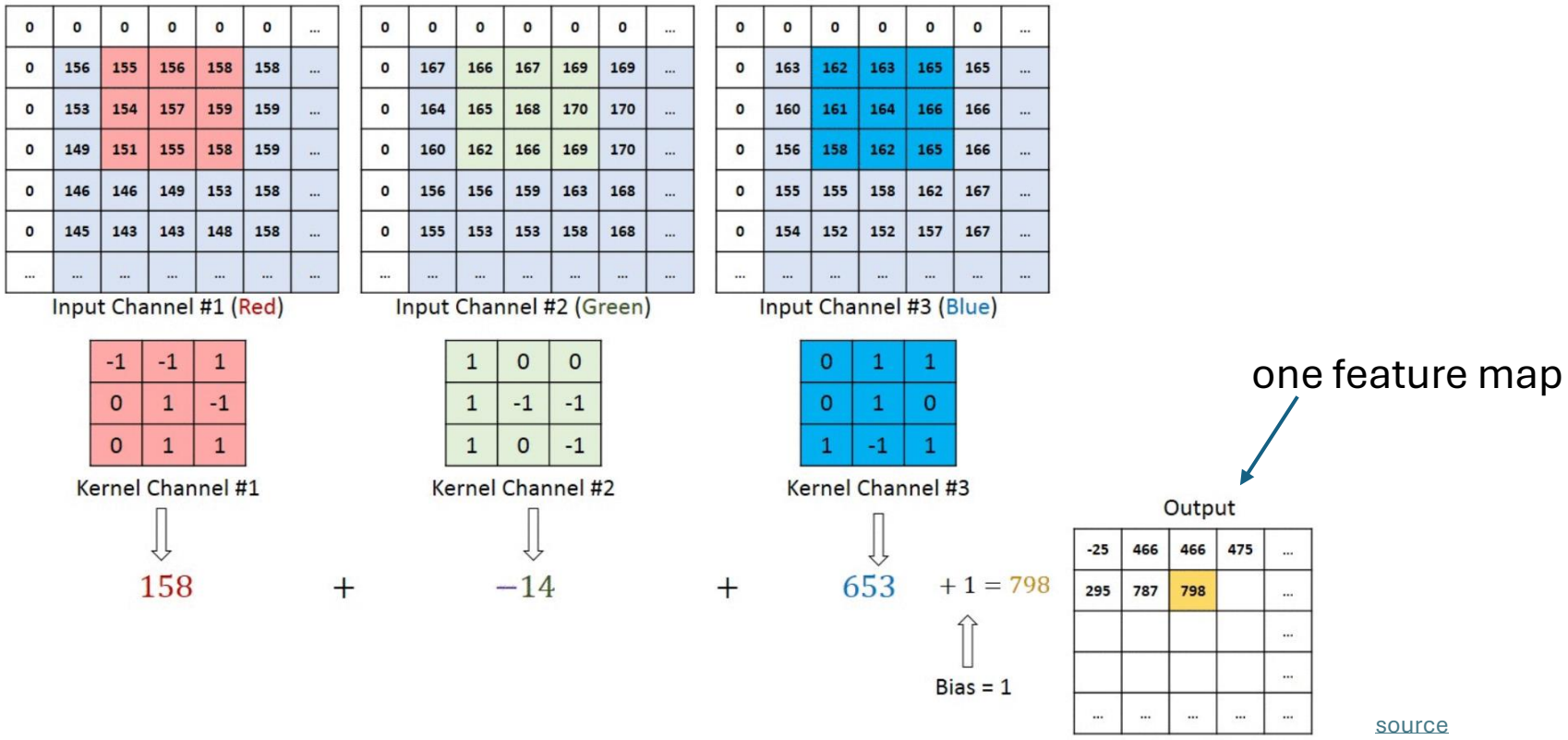


source

Channel Mixing

several input channels c (RGB) and several output channels f (different feature maps):

$$o_{f,x,y} = \sum_{c,s,t} w_{f,c,s,t}\ i_{c,x+s,y+t}$$



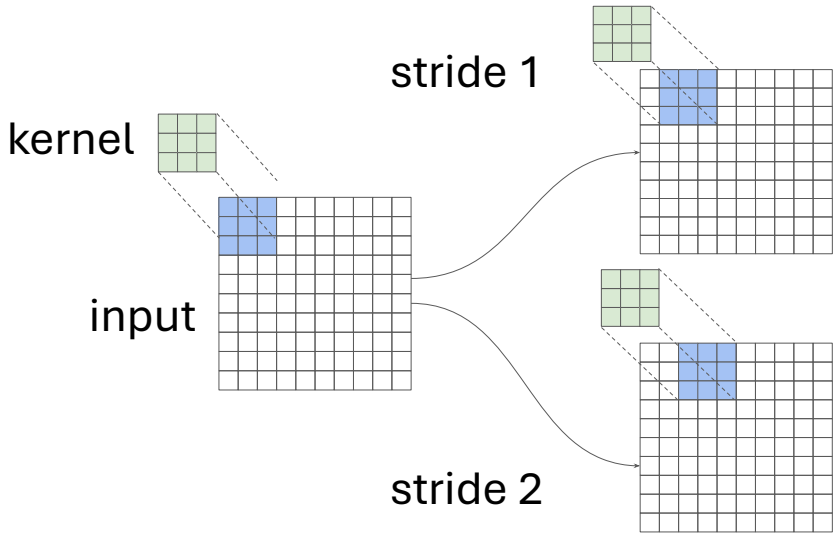
source

Striding and Padding

need to define how to stride over image

stride > 1 corresponds to down-sampling

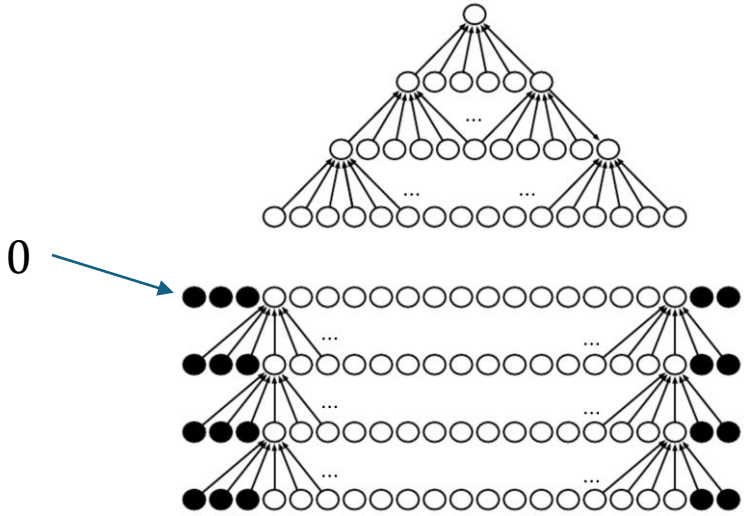
→ fewer nodes after convolutional layer



zero-padding of input to make it wider:

otherwise shrinking of represenation with each layer (depending on kernel size)

→ needed for large kernels and several layers



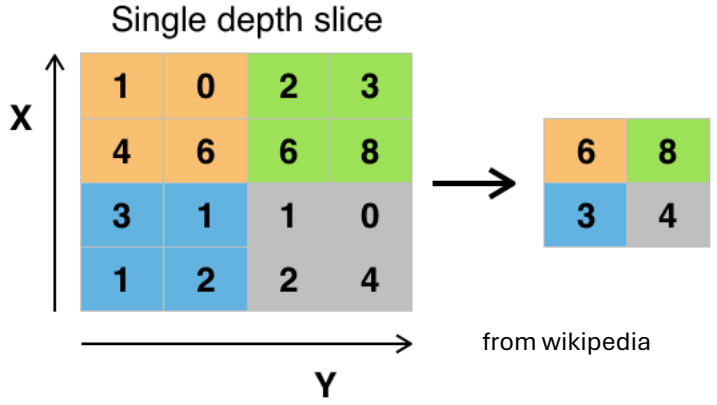
source

Another Ingredient: Pooling

replacing outputs of neighboring nodes with summary statistic (e.g., maximum or average value of nodes)

→ non-linear downsampling (regularization)

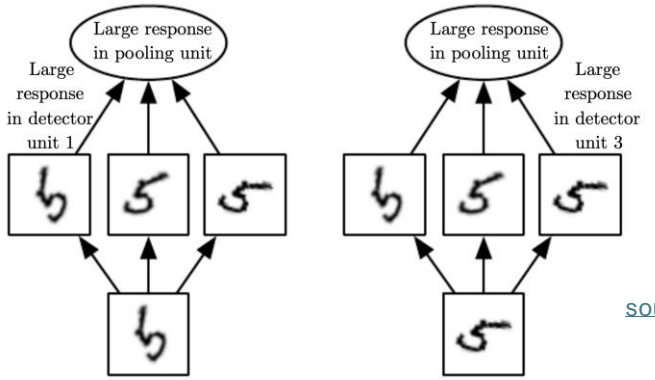
max pooling:



pooling is translation invariant: no interest in exact position of, e.g., maximum value

pooling over features learned by separate kernels (cross-channel pooling) can also learn other transformation invariances, like rotation or scale

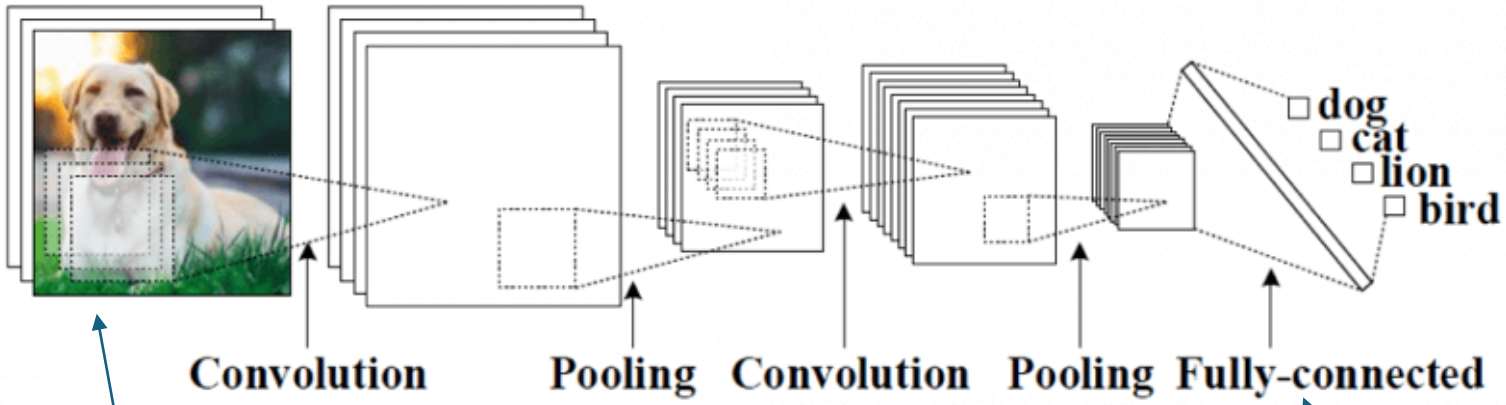
(convolutions can detect same translated motif across entire image, but not rotated or scaled versions of it)



source

Putting It All Together

convolutional neural networks in short:
local connections, shared weights, pooling, many layers

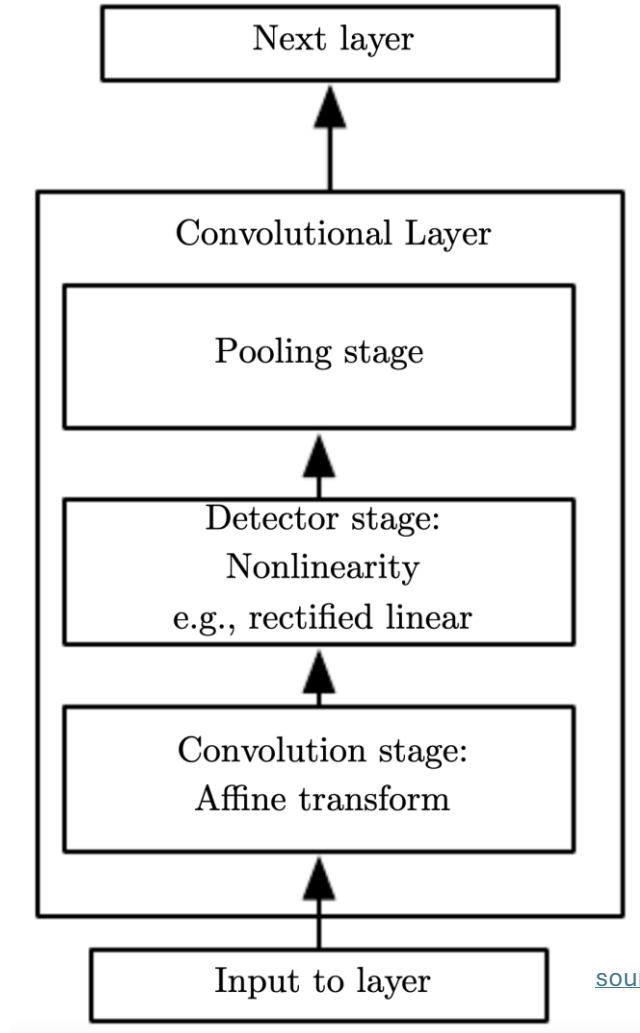


many images (training examples), potentially with several channels

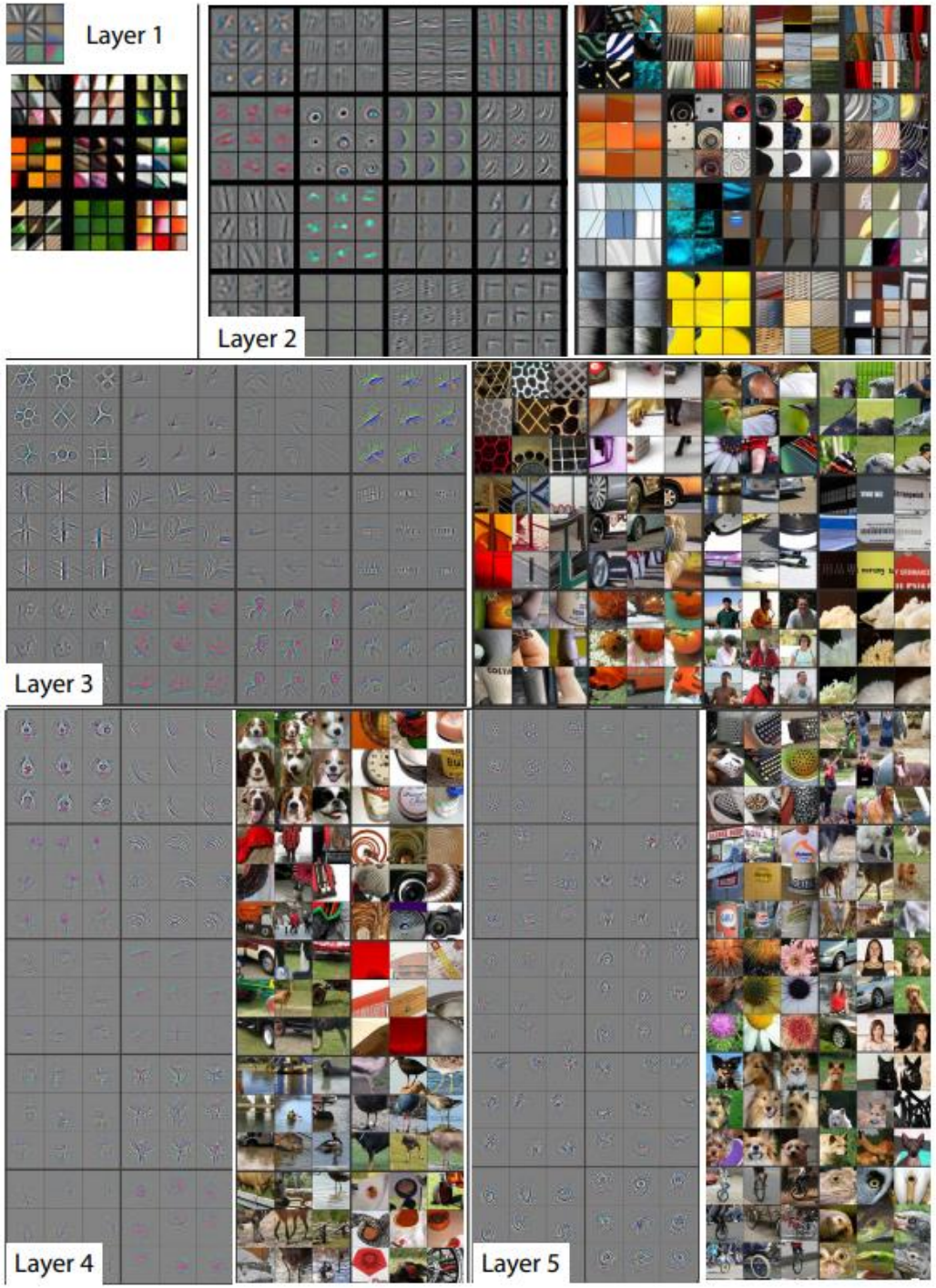
several kernels producing several feature maps

downsampling

flatten dimensions for final classification



source



Hierarchical Learning

shown here: top 9 activations of some random feature maps on test images, together with corresponding image patches
(projected down to pixel space using deconvolutional network)

some insights:

- strong groupings in feature maps, with exaggeration of discriminative image parts
- more global activations at higher layers
- greater invariance at higher layers

source

t-SNE Visualization

approach:
for every image, embed high-dimensional vector of last hidden layer's activations (right before final fully-connected classification layer) into two-dimensional vector (while preserving distances)

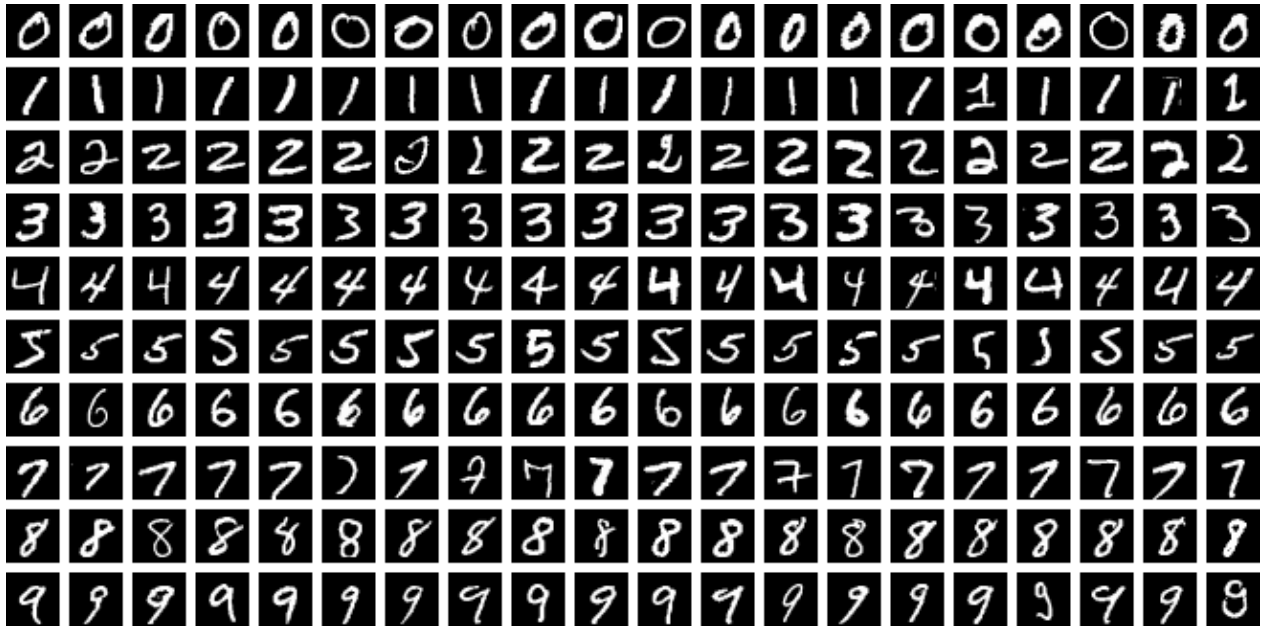
→ semantic similarities

source

Image Data Sets

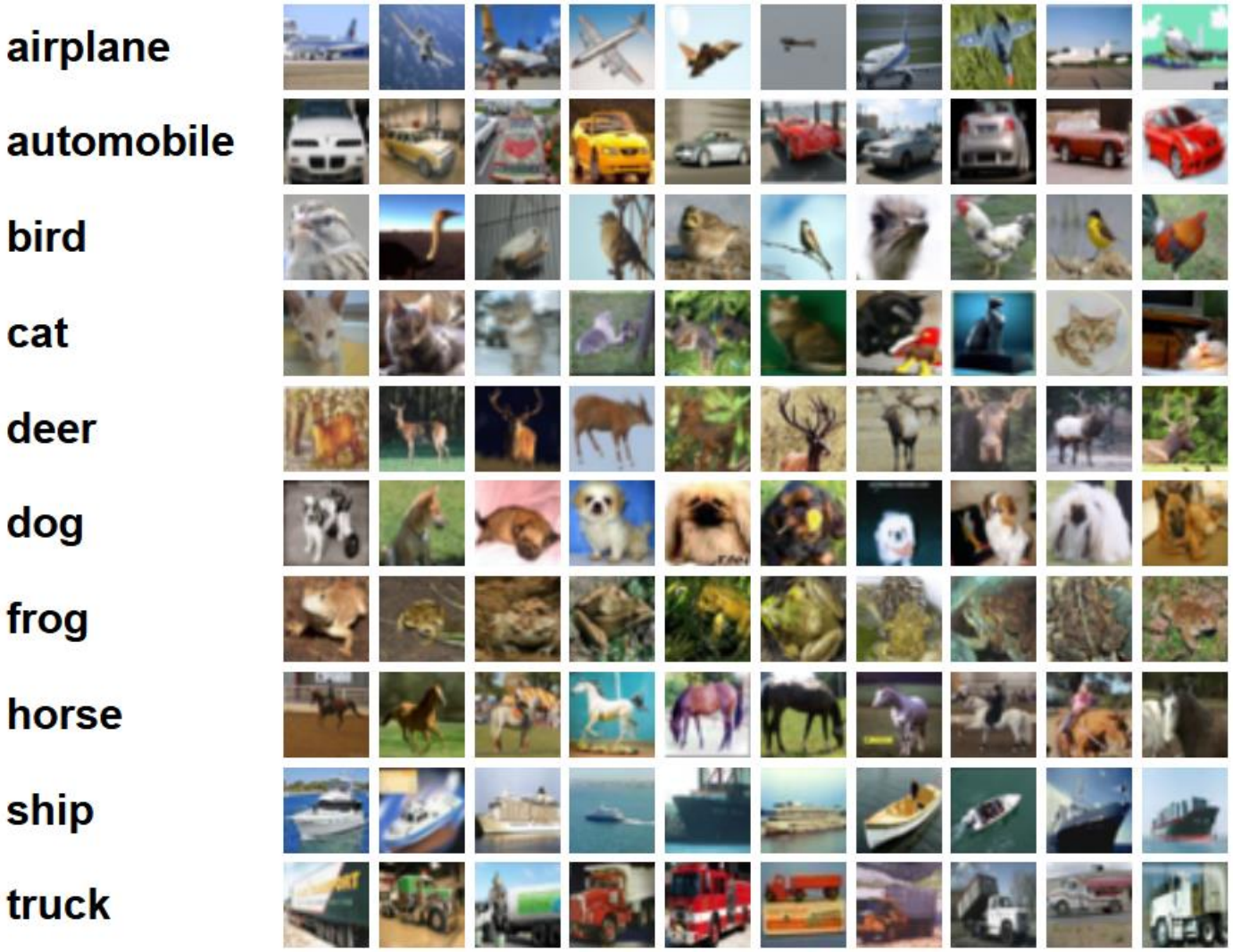
MNIST

- Modified National Institute of Standards and Technology
- handwritten digits (10 classes)
- black and white images
- 28 × 28 pixels
- 60k training and 10k test images



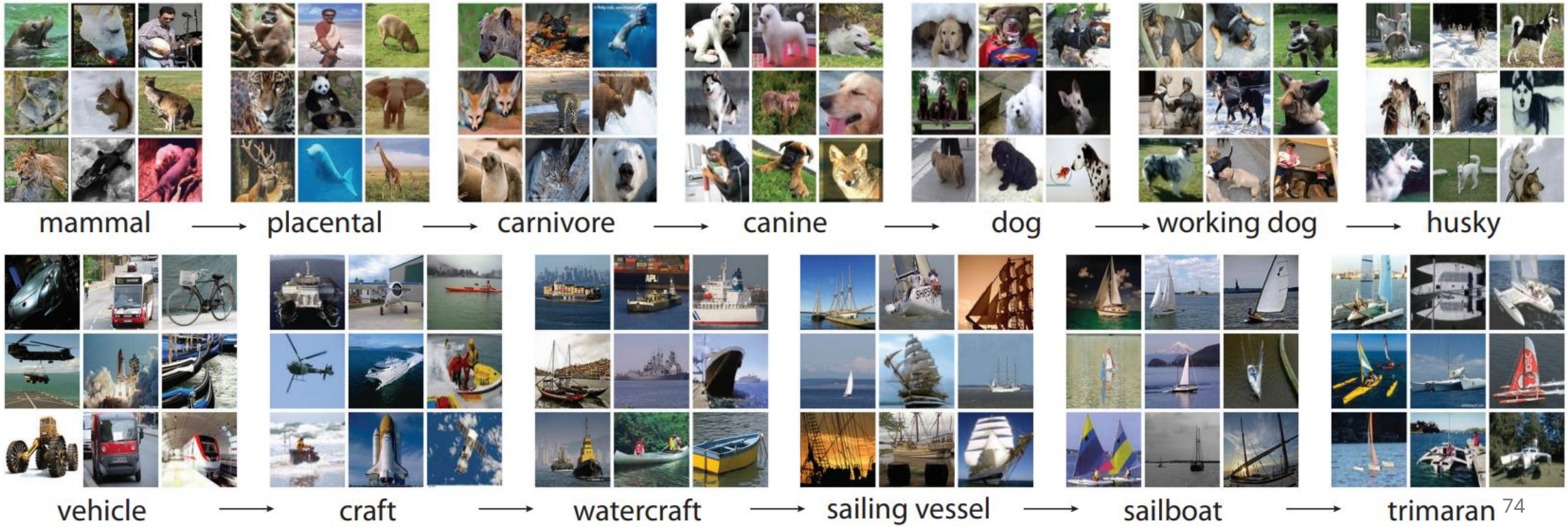
CIFAR-10

- Canadian Institute for Advanced Research
- 10 different labeled object classes
- color images
- 32 × 32 pixels
- 50k training and 10k test images

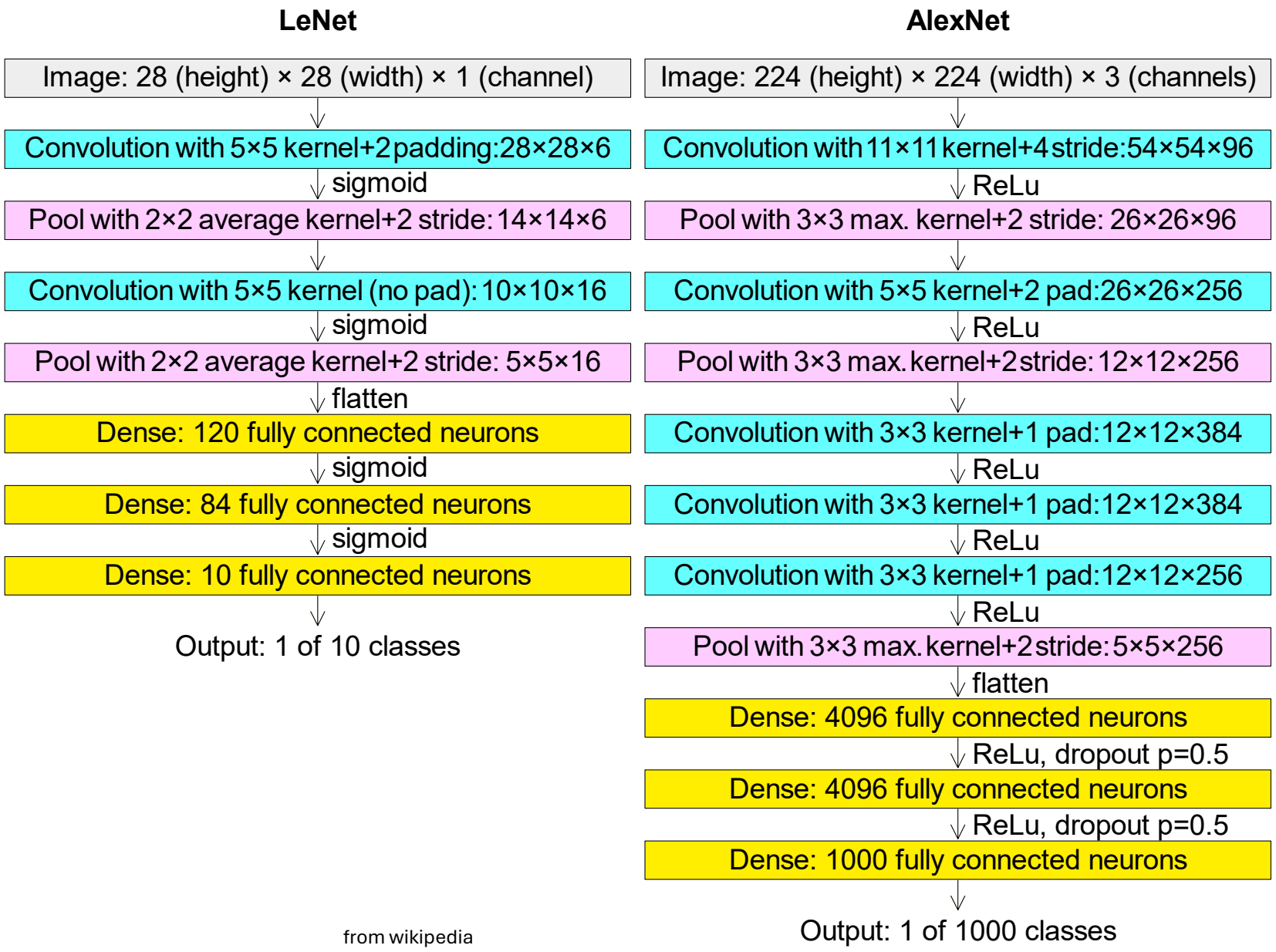


ImageNet

- more than 14 million color images with varying sizes
- more than 20k labeled categories



Going Deeper



from wikipedia

AlexNet finally started the deep learning hype.

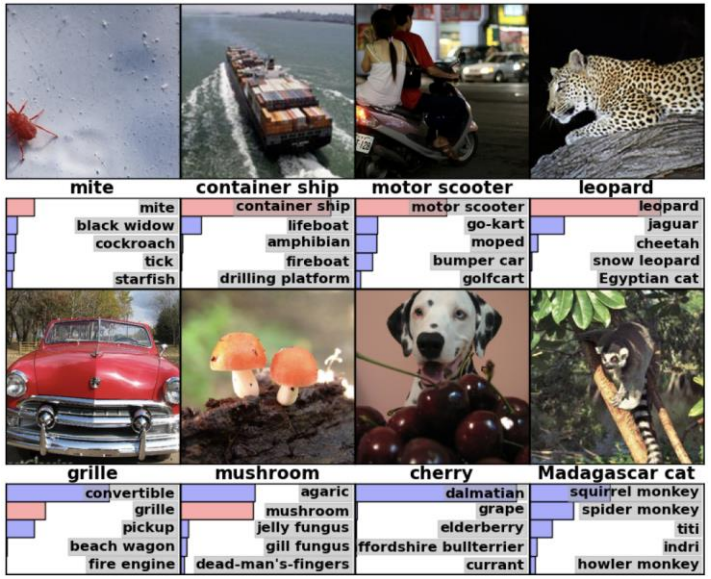
(winning the ImageNet challenge in 2012)

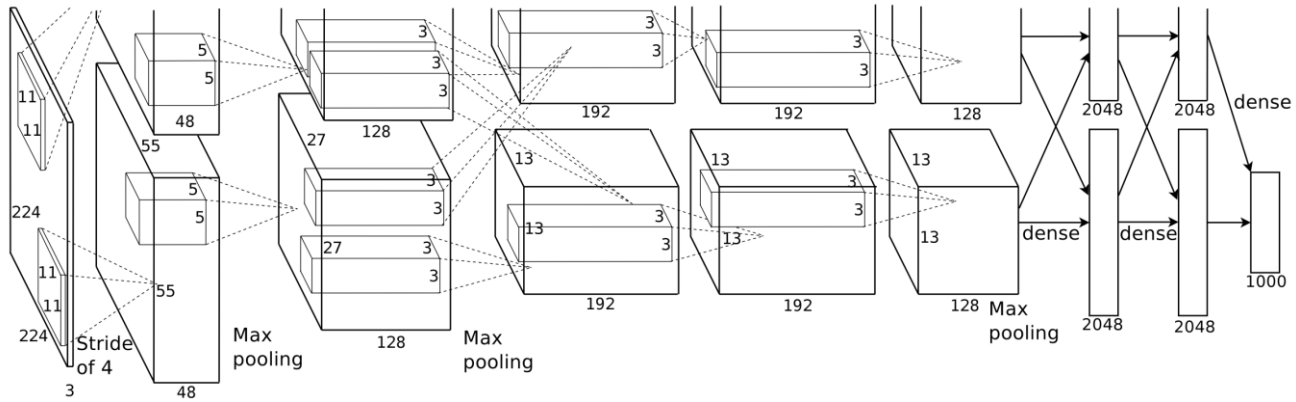
Rise of Deep Learning

a little bit oversimplified:

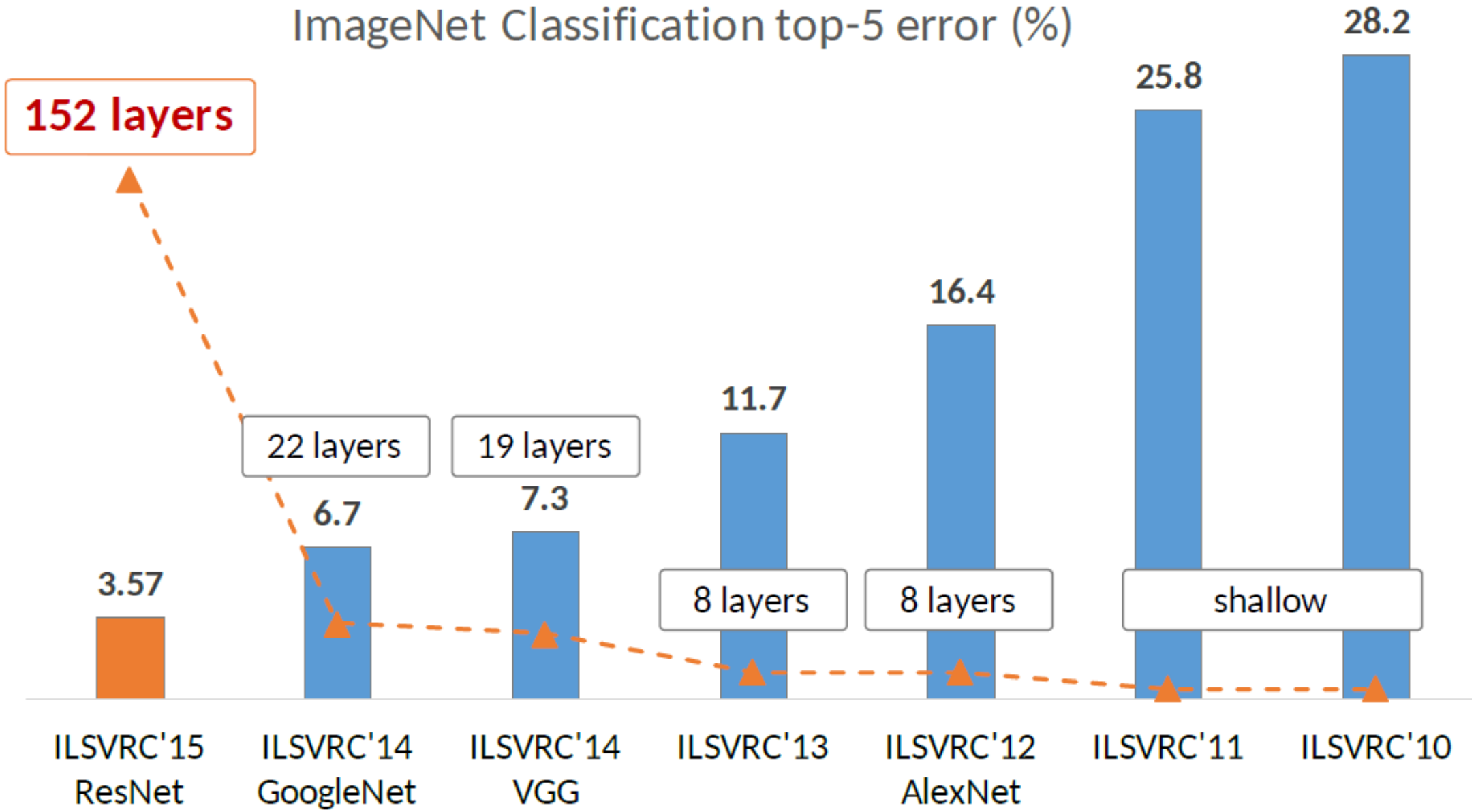
deep learning = lots of training data + parallel computation + smart algorithms

AlexNet: ImageNet (with data augmentation) + GPUs + ReLU, dropout, SGD

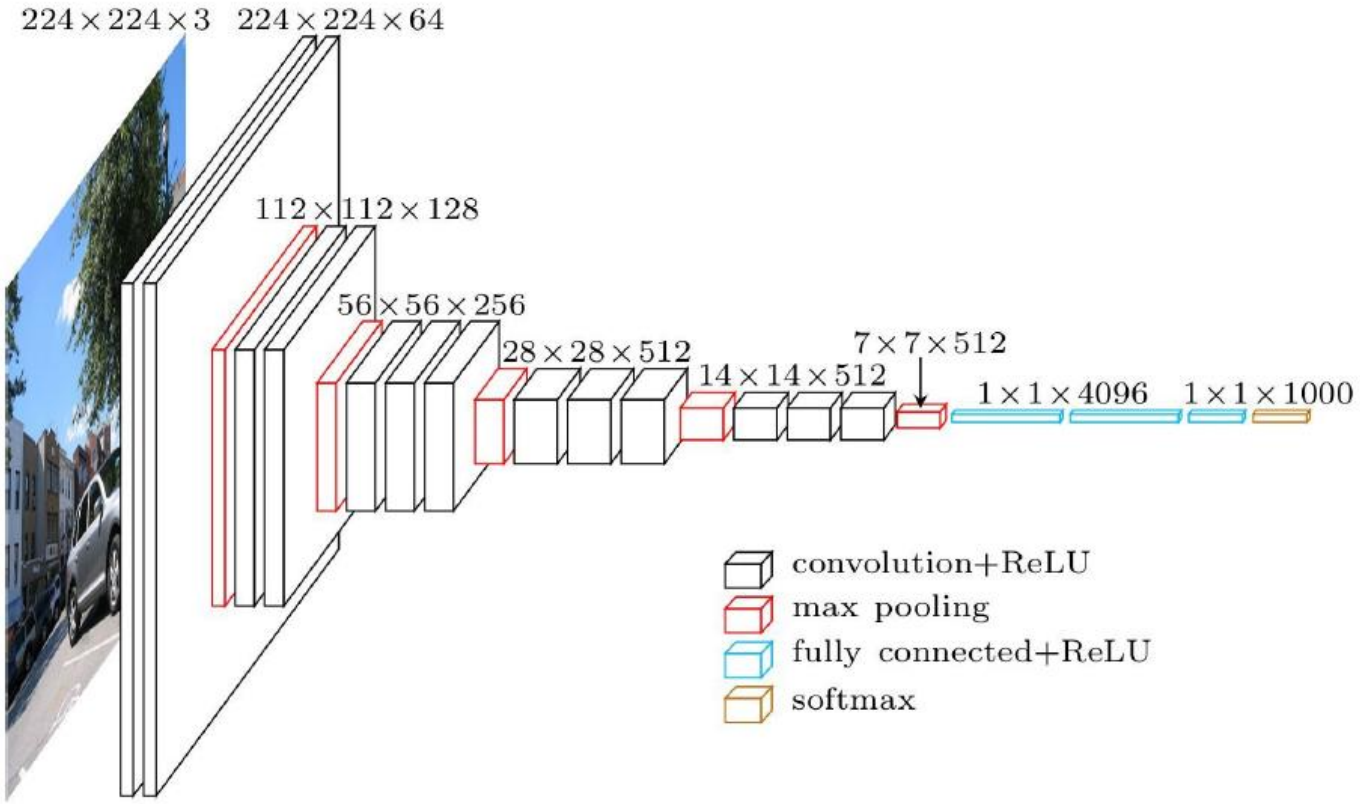




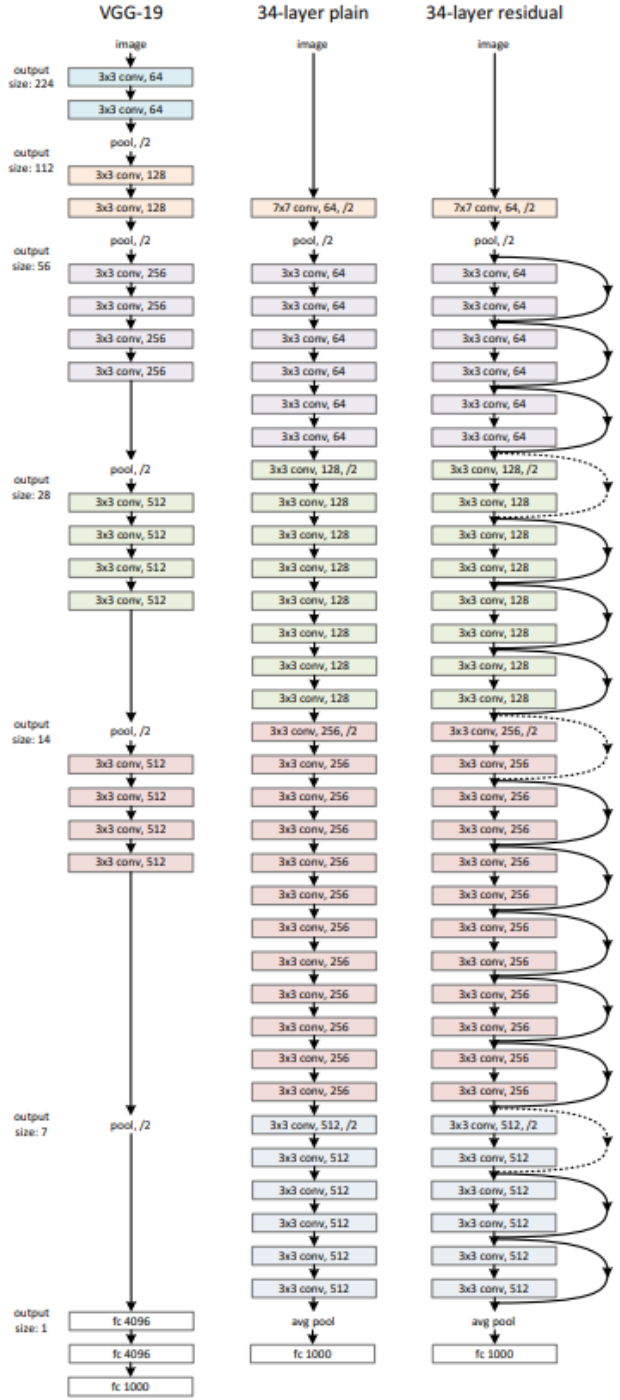
source


ImageNet Classification top-5 error (%)
152 layers
22 layers
19 layers
8 layers
8 layers
shallow
3.57
6.7
7.3
11.7
16.4
25.8
28.2
ILSVRC'15 ResNet
ILSVRC'14 GoogleNet
ILSVRC'14 VGG
ILSVRC'13
ILSVRC'12 AlexNet
ILSVRC'11
ILSVRC'10

VGG


224 × 224 × 3
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
1 × 1 × 4096
1 × 1 × 1000
convolution+ReLU
max pooling
fully connected+ReLU
softmax

ResNet


VGG-19
34-layer plain
34-layer residual
image
3x3 conv, 64
pool, /2
3x3 conv, 128
3x3 conv, 256
3x3 conv, 512
7x7 conv, 64, /2
3x3 conv, 128, /2
3x3 conv, 256, /2
3x3 conv, 512, /2
fc 4096
fc 1000
avg pool

Skip Connections

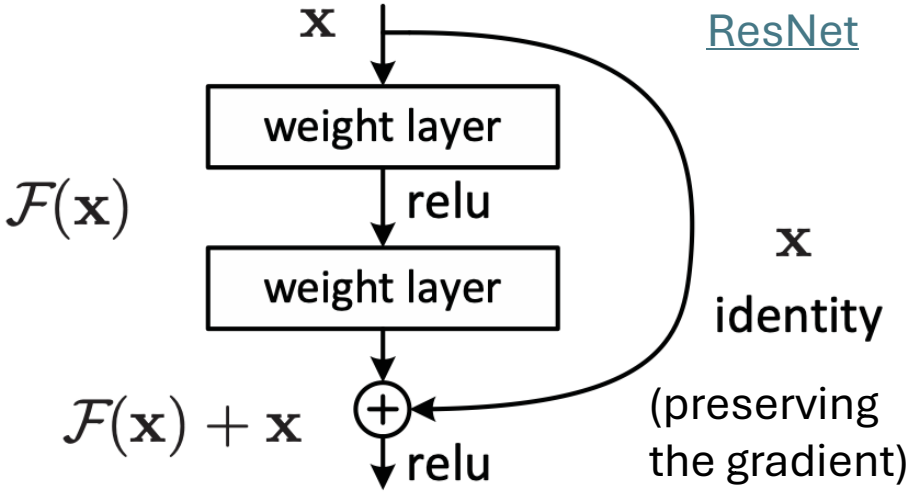
issue: degradation of training and test errors when adding more and more layers → not due to overfitting (but reason controversial)

solution: learning of residuals by means of skip connections (resulting in combination of different paths through computational graph)

→ produces loss functions that train easier

together with batch normalization (avoiding exploding gradients), skip connections enable extremely deep networks (>1000 layers) without degradation

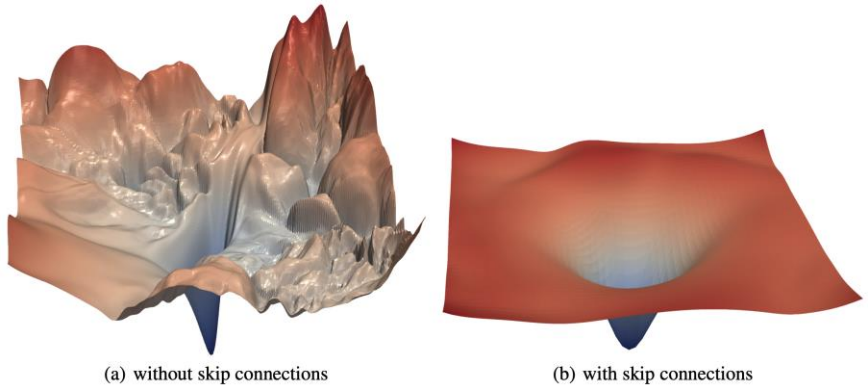
residual mapping (special kind of skip/shortcut connections):



ResNet

(preserving the gradient)

loss surface:



(a) without skip connections (b) with skip connections

source

Transfer Learning

Foundation Models

problem with ConvNets trained from scratch (in fact, with all ML methods): only suited for domain of training data

idea of transfer learning:

pretrain a big model (called foundation model) on a broad data set, and then use these learnings for subsequent trainings on specific (typically, narrow) data

two forms of transfer learning: feature extraction and finetuning

Feature Extraction

approach:

- get a pretrained ConvNet as foundation model
- remove final fully-connected layer (its outputs are the class scores from the pretraining task)
- pass training data (for the task at hand) through the network
- for each image, extract vector of last hidden layer‘s activations (e.g., 4096-dimensional for AlexNet)
- use the extracted values as features to train another model (to be adjusted to the task at hand)

(same idea as used before for SIFT features)

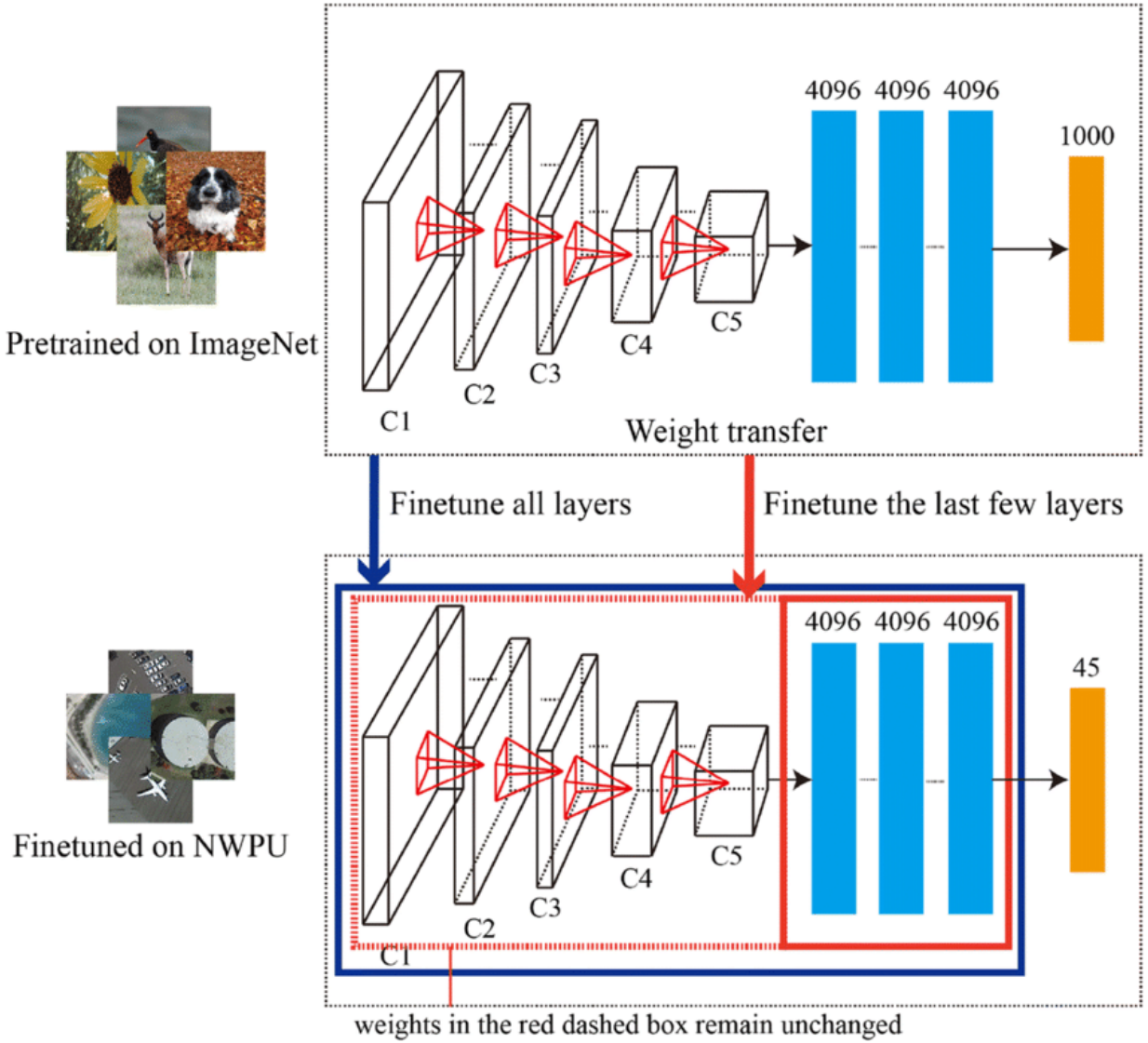
Finetuning

approach:

- get pretrained ConvNet as foundation model
- continue (starting with weights after pretraining) training with data for task at hand

typically, need to change architecture of final layer: different number of classes

options: finetune all weights or just some



source

Finetuning Scenarios

finetuning gives better adjustment to new task than feature extraction (important if new task differs a lot from pretraining task)

→ coming along with danger of overfitting

ConvNet features more generic in early layers (e.g., edges) and more specific to the data set of pretraining in later layers

→ for small finetuning data sets, typically keep weights of early layers fixed to avoid overfitting

for large finetuning data sets (less danger of overfitting), finetuning all layers can be beneficial

Brief Recap of Different Image Classification Techniques

- Hough transform: looking for predefined shapes such as lines or circles (feature-based recognition)
- feature description & matching: identify specific object instances (typically used for instance rather than class recognition)
- image features as input to classic ML method: generalization from data
- plain feed-forward network on raw image data: even more generalization without handcrafted components
- convolutional neural networks: use of spatial structure (inductive bias)
- pretraining & finetuning (or feature extraction): transfer learning

Transformer

Large Language Models (LLM)

recent hype around LLMs

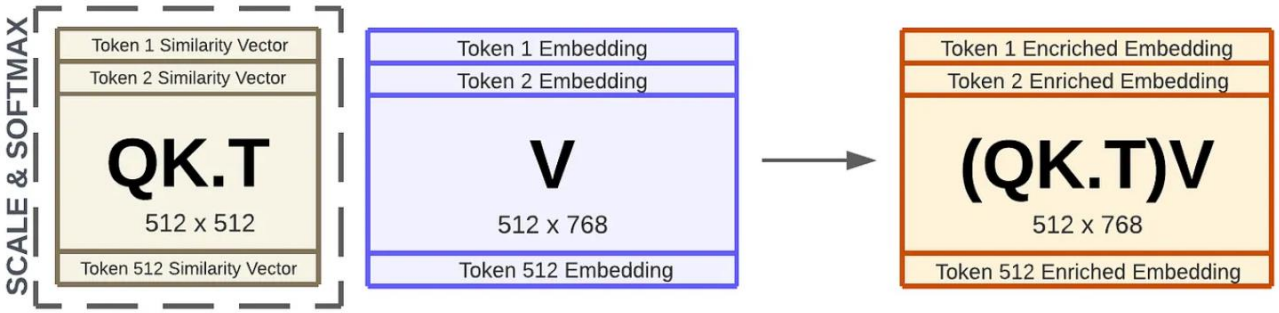
fully started with ChatGPT release end of 2022

technical backbone: transformer architecture

transformers also applicable to computer vision (alternative to convolutional neural networks)

Modern Language Models in a Nutshell

- self-supervised learning: e.g., next/masked-word prediction
- tokenization: split text into chunks (e.g., words)
- vector embeddings (learned via bag-of-words or end-to-end in transformer): semantic meaning of tokens (not context-aware though)
- add positional encoding to embeddings: use order of sequence (attention is permutation-invariant)
- self-attention (influence of other tokens) in transformer: context awareness



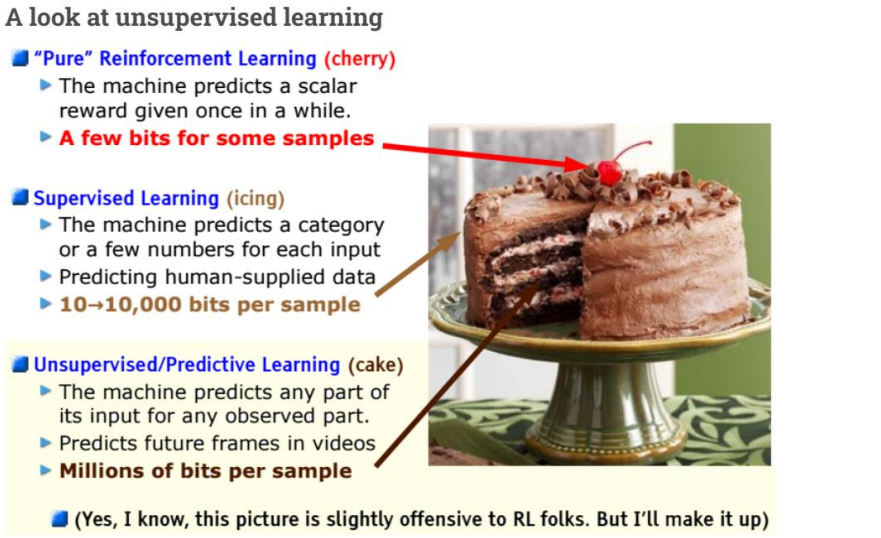
source

ML needs lots of training data

Self-Supervised Learning

unsupervised learning (learning by observation)

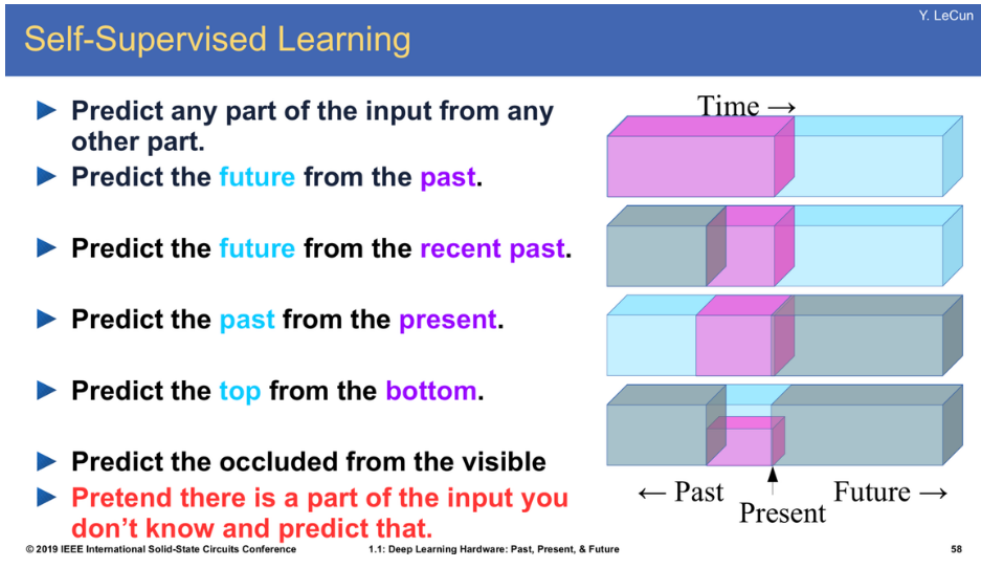
no target information → kind of “vague” pattern recognition (but plenty of data)



Original LeCun cake analogy slide presented at NIPS 2016, the highlighted area has now been updated. source

can be cast as **self-supervised learning**:

- input-output mapping like supervised learning
- but generating labels itself from input information



LeCun's self-supervised learning slide at ISSCC 2019

generative AI as unsupervised learning: generate variations of training data

Tokenization

tokenization: *breaking text in chunks*

- word tokens: different forms, spellings, etc → undefined and vast vocabulary
- character tokens: not enough semantic content (longer sequences)
- popular compromise: byte-pair encoding

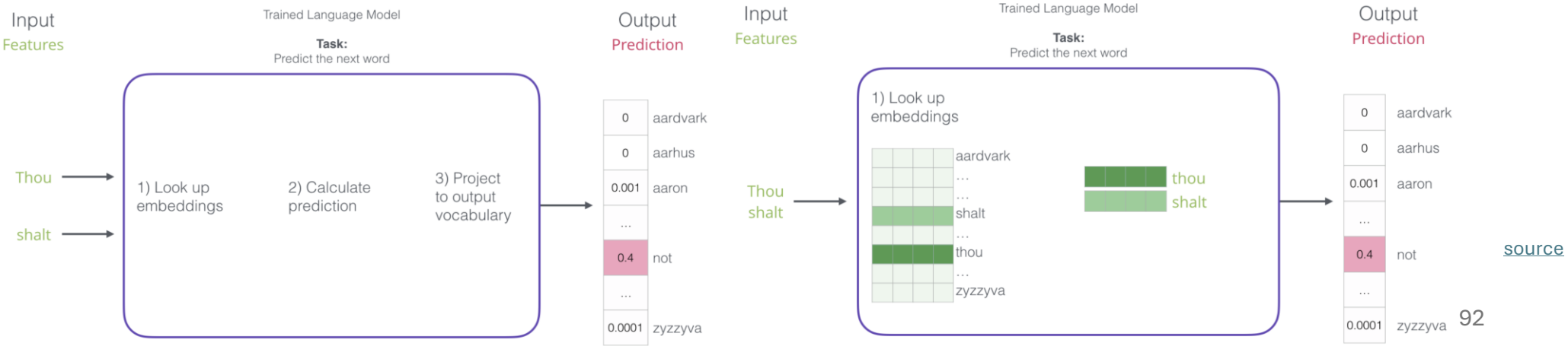
Word Embeddings as Part of Language Model

language models contain embedding matrix as part of learned parameters

- typically several hundred dimensions for word vectors (to be compared with vocabulary sizes of many thousands)
- trained on huge data sets

(can also be extracted and subsequently used as pretrained embeddings for other task)

next-word prediction:



source

In General: Contrastive Learning

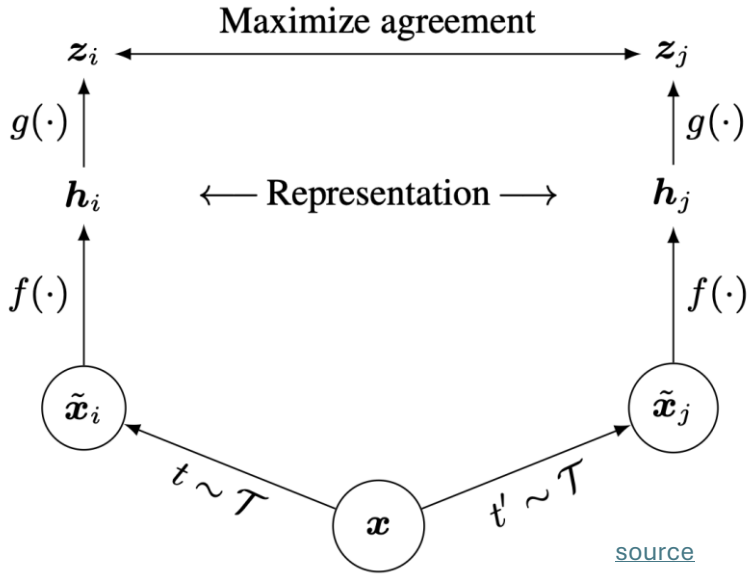
idea:

create embedding space in which similar samples are close to each other and dissimilar ones are far apart

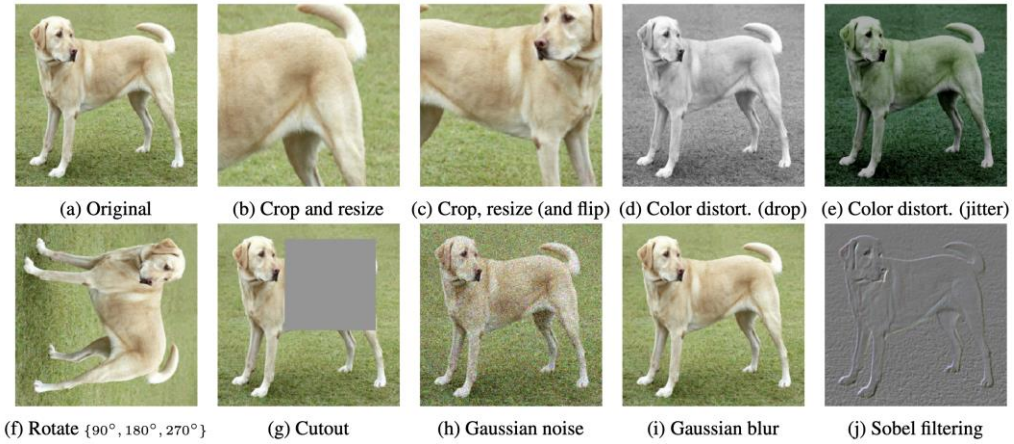
not only useful for language

→ learning of image representations

often learned in a self-supervised way



source



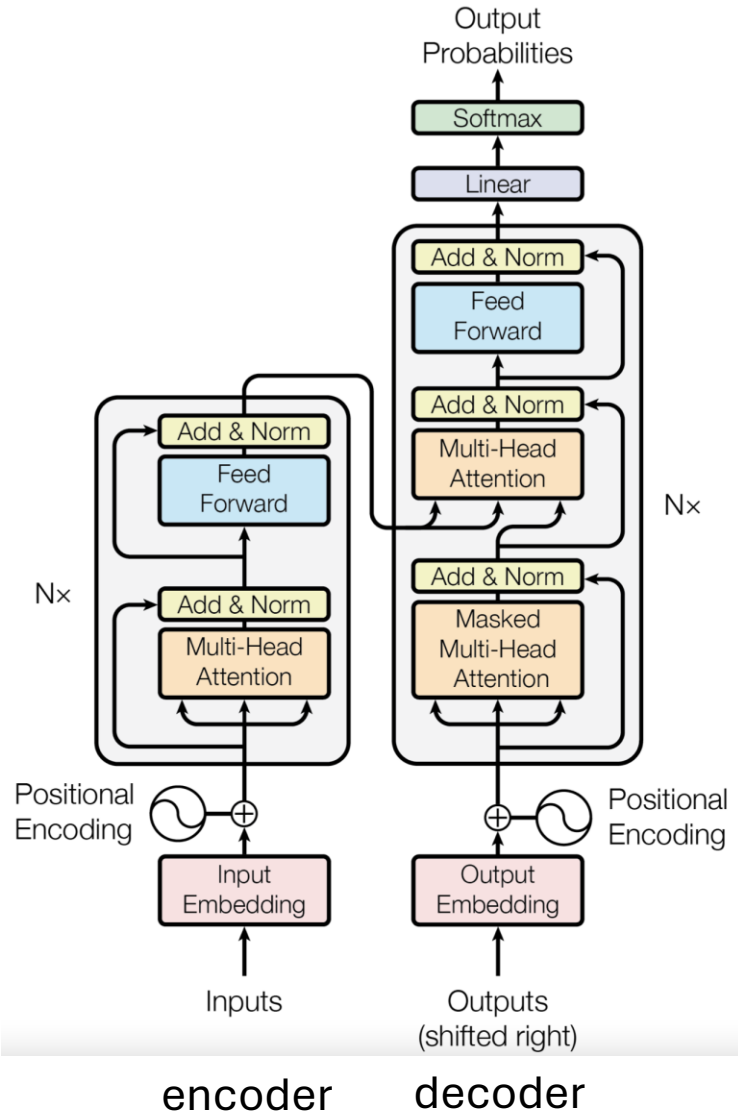
Context Awareness: Transformer

Attention Is All You Need:

completely replaced recurrent neural networks (RNN)

- much more parallelization → bigger models (more parameters)
- better long-range dependencies thanks to shorter path lengths in network (less sequential operations)

original transformer:
sequence-to-sequence model
(e.g., for machine translation)



Transformer Architectures for LLMs

encoder-decoder LLMs: sequence-to-sequence, e.g., machine translation

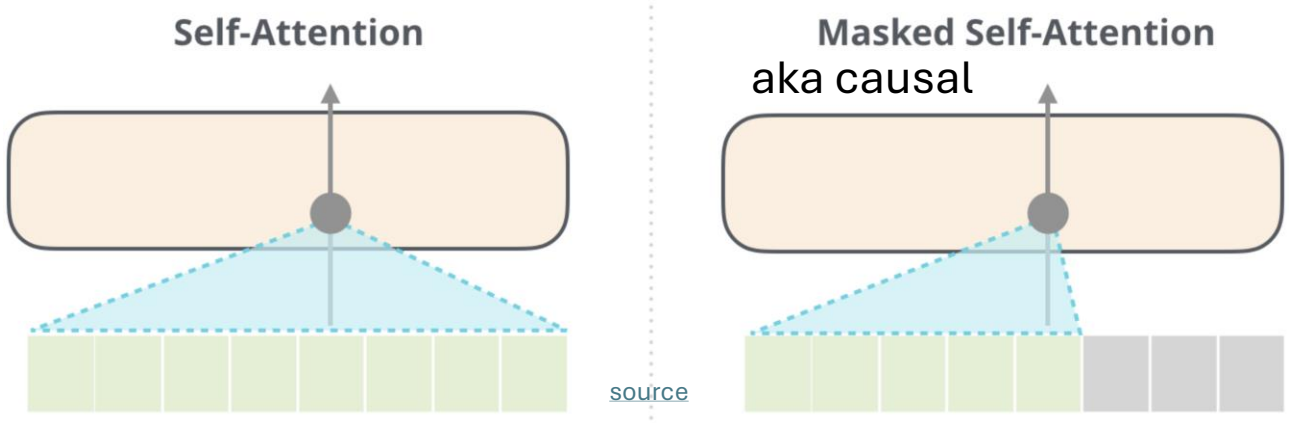
encoder-only LLMs:

- learning of contextual embeddings (for subsequent fine-tuning)
- training: prediction of masked words
- can't generate text, can't be prompted

decoder-only LLMs:

- text generation: e.g., chat bot (after instruction tuning)
- training: next-word prediction
- output one token at a time (auto-regressive: consuming its own output, potentially starting with prompts)

example:
BERT

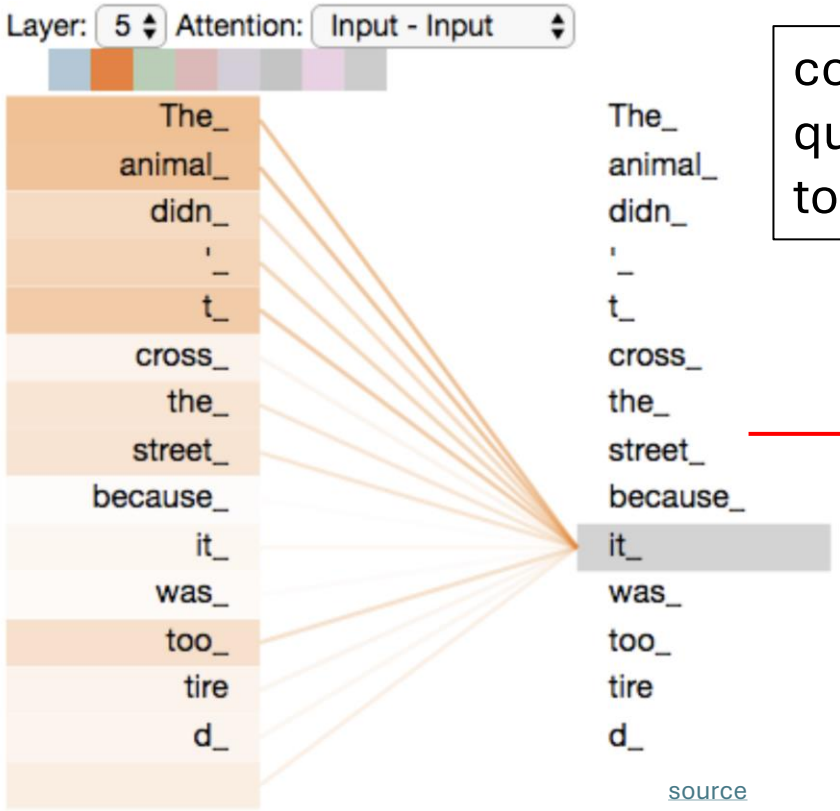


source

example:
GPT-3

Self-Attention Mechanism

evaluating other input tokens in terms of relevance for encoding of given token

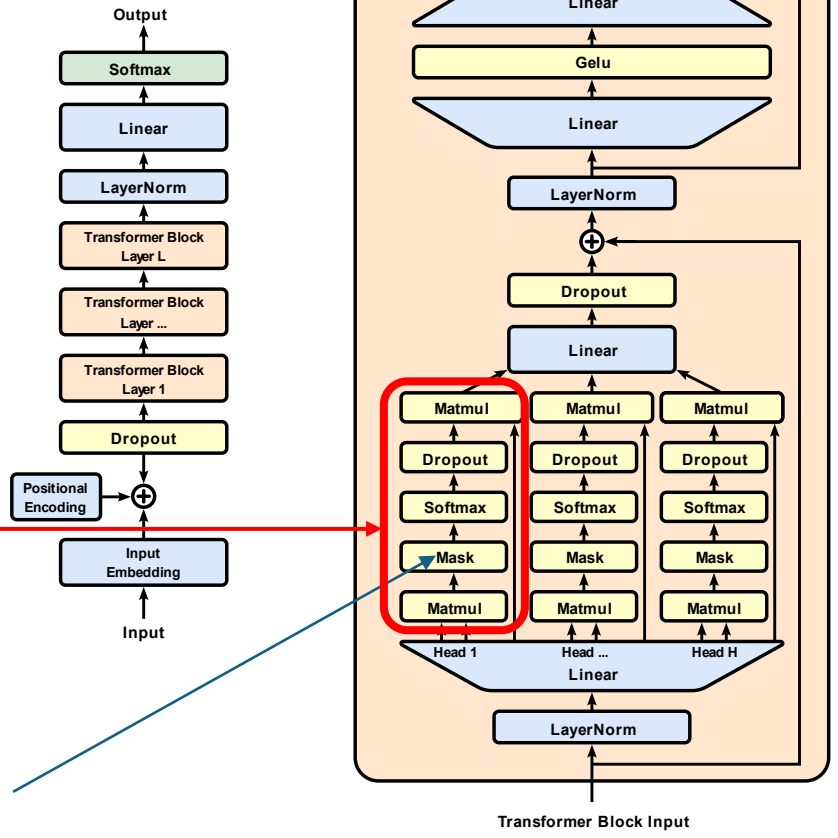


source

computational complexity quadratic in length of input (each token attends to each other token)

only allow to attend to earlier positions in sequence (masking future positions by setting them to $-\infty$)

GPT (decoder-only):



Scaled Dot-Product Attention

softmax not scale invariant: largest inputs dominate output for large inputs (more embedding dimensions d_k)

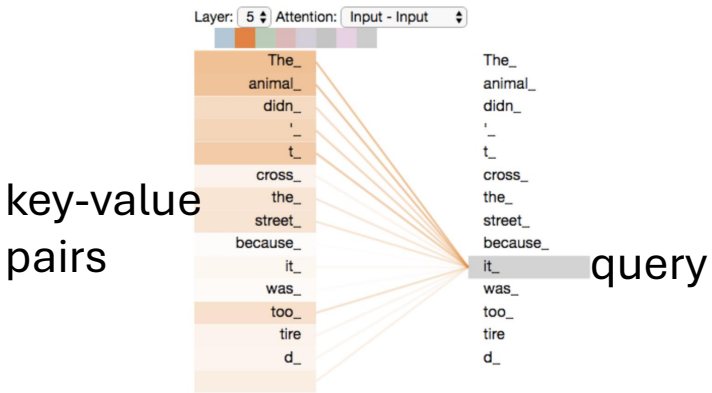
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention

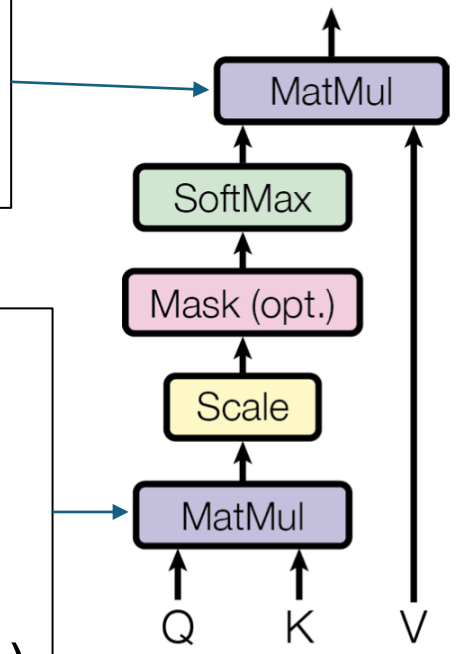
3 matrices created from input embeddings by multiplication with 3 different weight matrices

- query Q
- key K
- value V

filtering: multiplication of attention probabilities with corresponding key-word values

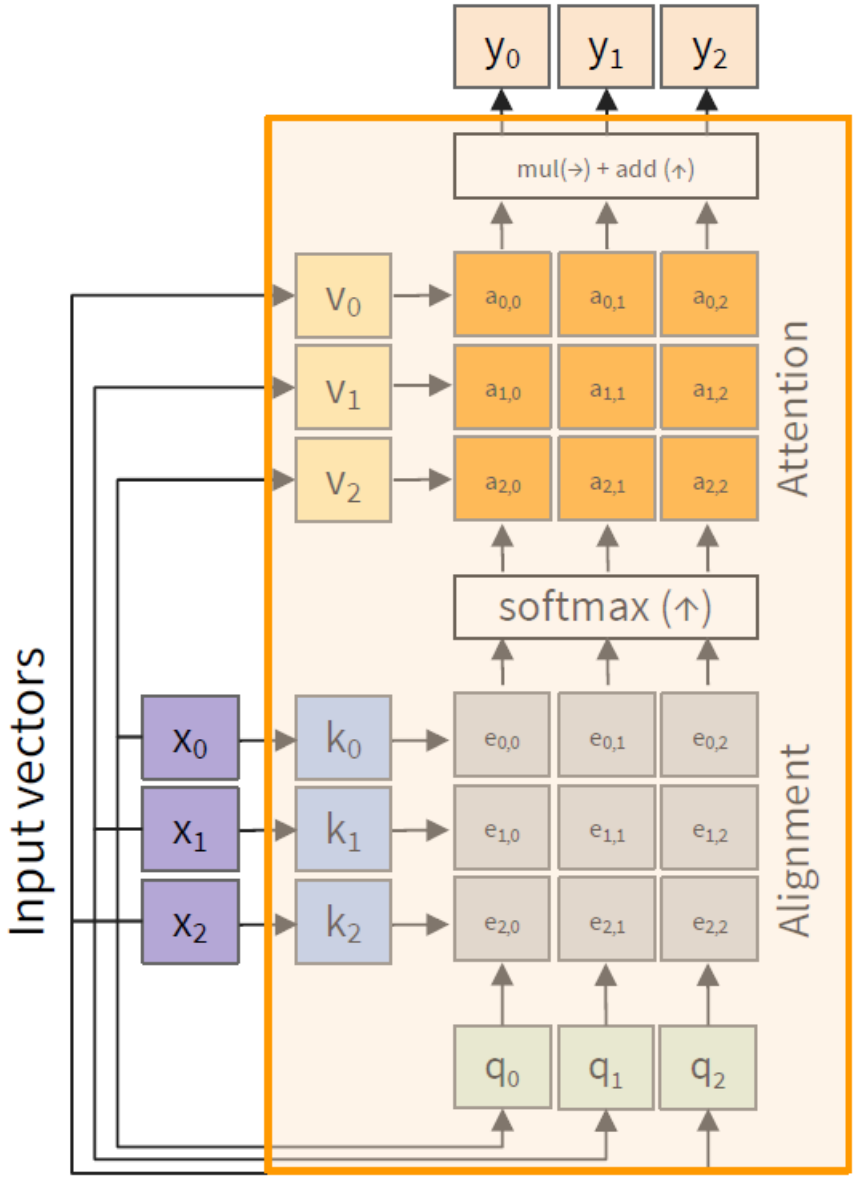


scoring each of the key words (context) with respect to current query word: multiplication of inputs (in contrast to inputs times weights in neural networks)



source

MatMul



Outputs:
context vectors: y (shape: D_v)

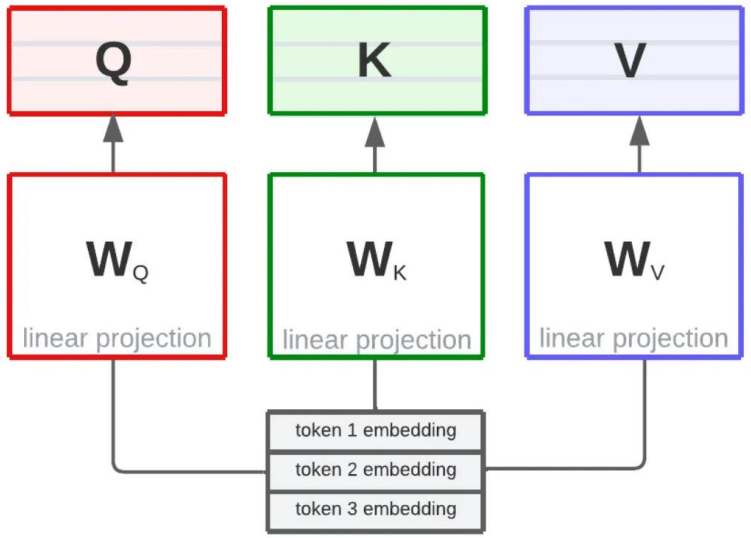
Operations:
Key vectors: $k = xW_k$
Value vectors: $v = xW_v$
Query vectors: $q = xW_q$
Alignment: $e_{i,j} = q_j \cdot k_i / \sqrt{D}$
Attention: $a = \text{softmax}(e)$
Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:
Input vectors: x (shape: N x D)

weighted average: reflecting to what degree a token is paying attention to the other tokens in the sequence

Example: Dimensions in BERT

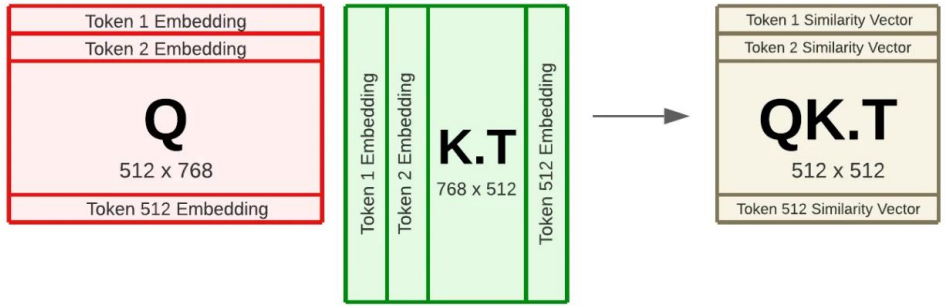
1



Projecting token embeddings into Query, Key, and Value vectors through matrix multiplication with their respective Weights (W).

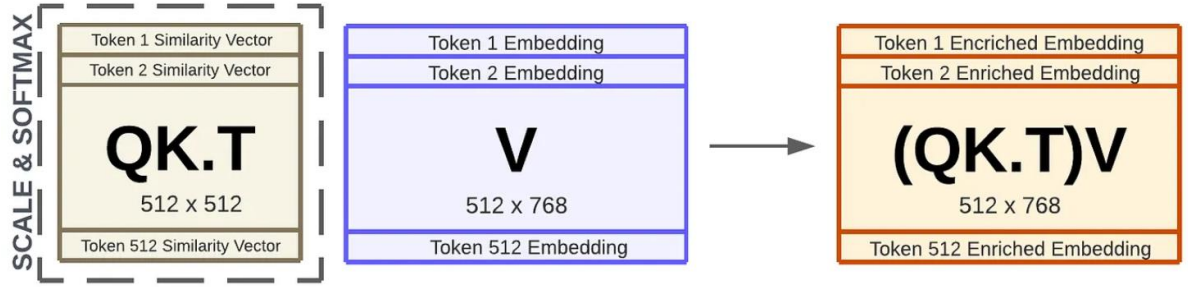
source

2



The Query (Q) matrix multiplied with the Key (K.T) resulting in the matrix of similarity of scores (QK.T).

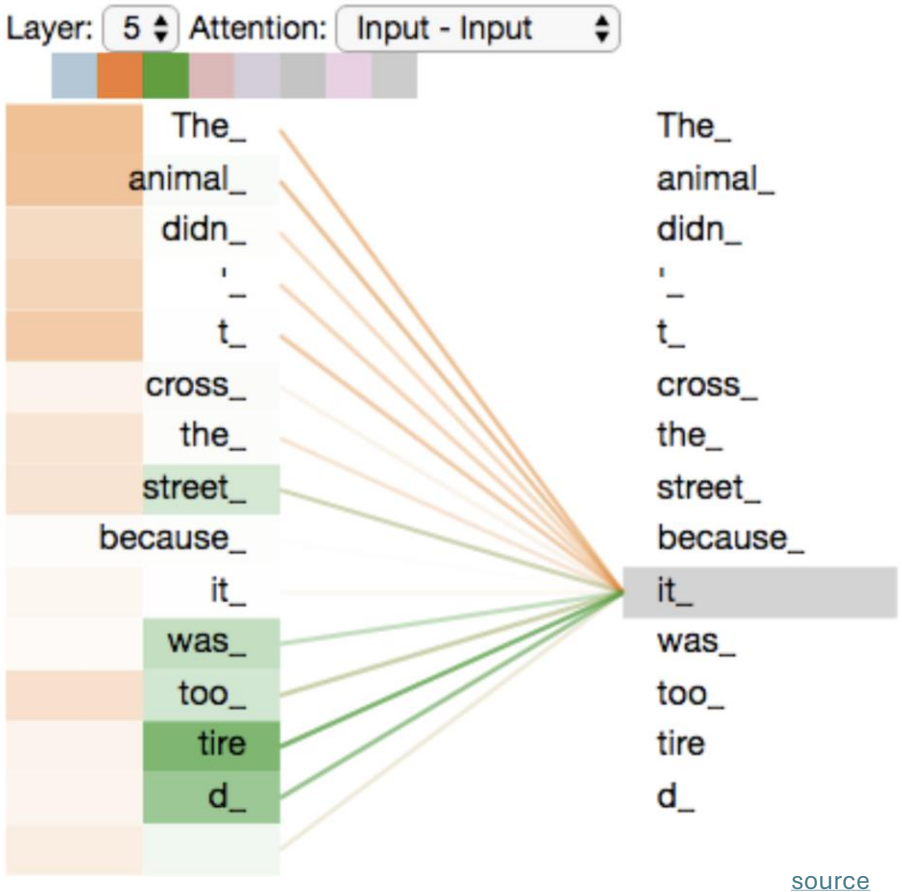
3



The scaled and soft similarity scores matrix multiplied with the values (V) resulting in enriched embeddings.

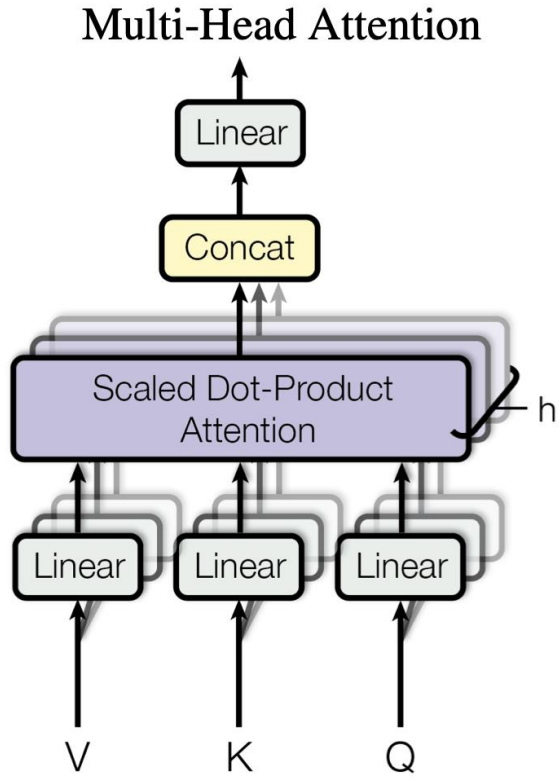
Multi-Head Attention

multiple heads: several attention layers running in parallel



source

different heads can pay attention to different aspects of input (multiple representation sub-spaces)



source

Some LLM Numbers

example Llama 3 405B:

- vocabulary size (tokens): 128K
- embedding/model dimensions: 16,384

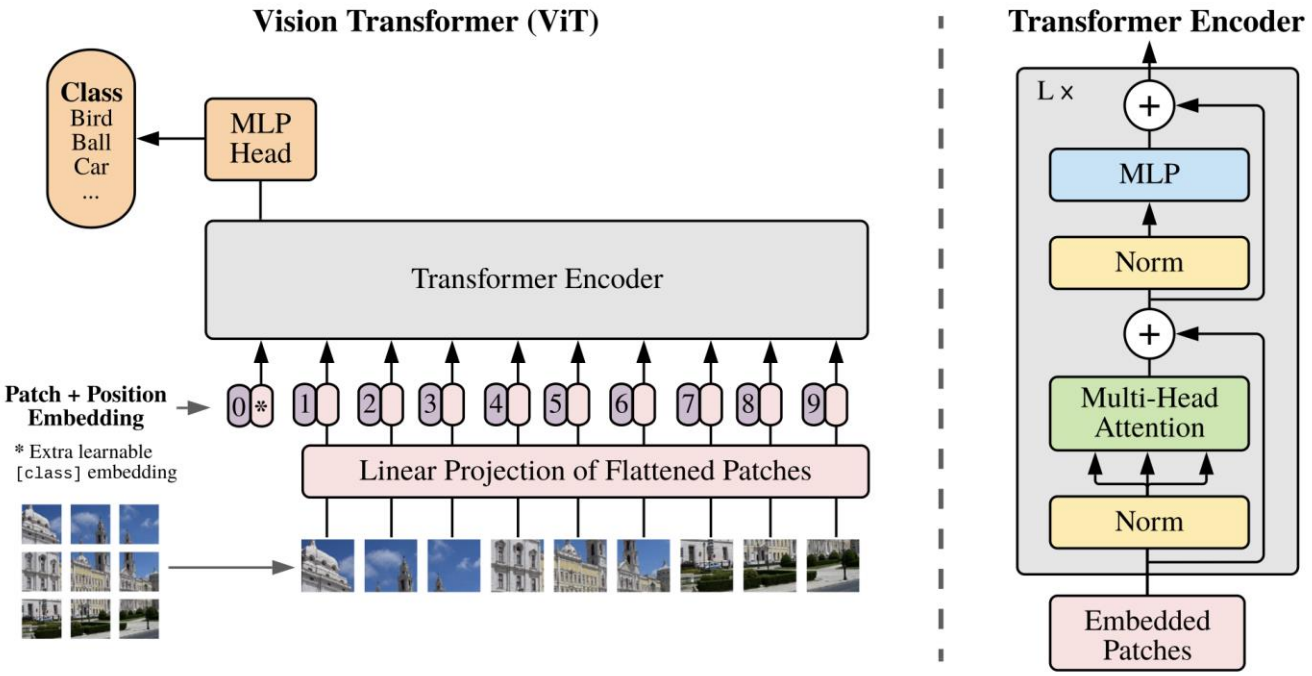
- parameters: 405B
- training tokens: 15.6T

factor less than 40
→ a lot of memorizing

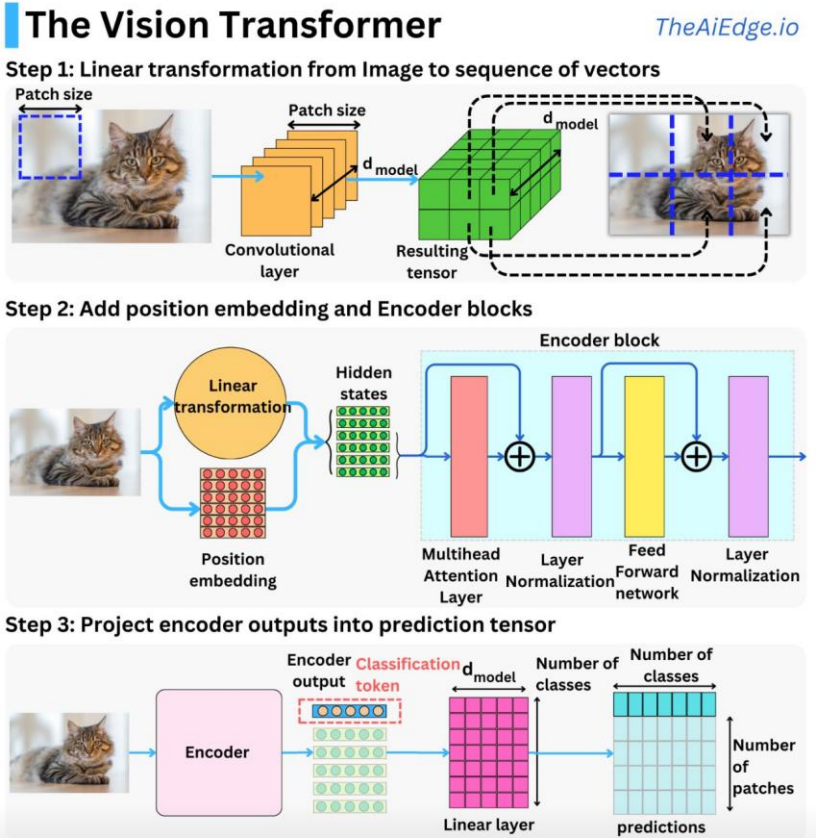
- context length/window (tokens): 128K

- training hardware: 16K GPUs (H100)

Image Classification with Vision Transformer



source



formulation as sequential problem: split image into patches (tokens) and flatten, add positional embeddings

processing by transformer encoder: pretrain with image labels, fine-tune on specific data set

Attention vs Convolution

fewer inductive biases in ViT than in CNN:

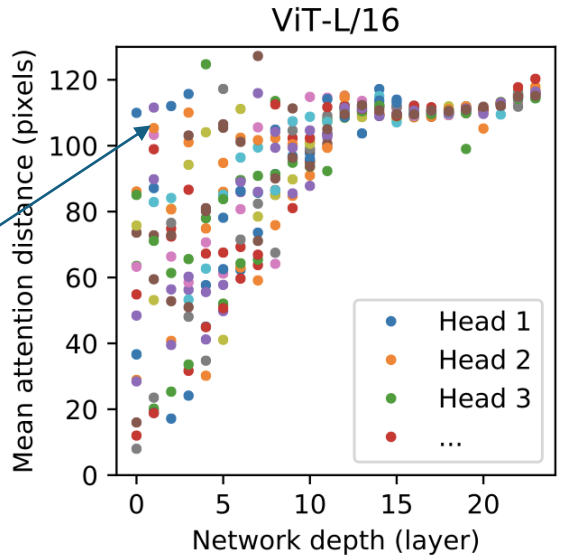
- no translation invariance
- no locally restricted receptive field

Since these are natural for vision tasks, ViTs (conceptionally) learn them from scratch. → ViTs need way more data.

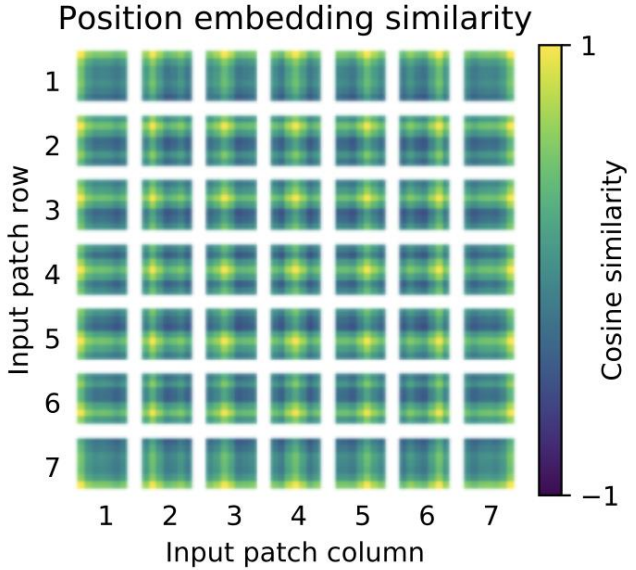
but can lead to beneficial effects (e.g., global attention in lower layers)

see MLP-Mixer results: given enough data, plain multi-layer perceptrons can learn crucial inductive biases

global attention in lower layers (unlike local receptive fields in CNNs)

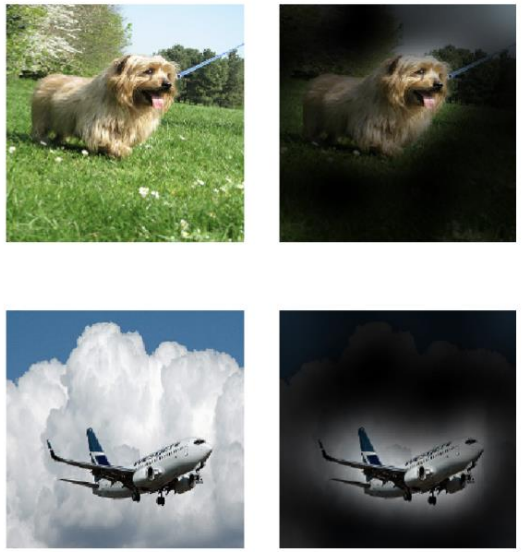


trainable position embedding:



added due to permuation invariance of attention

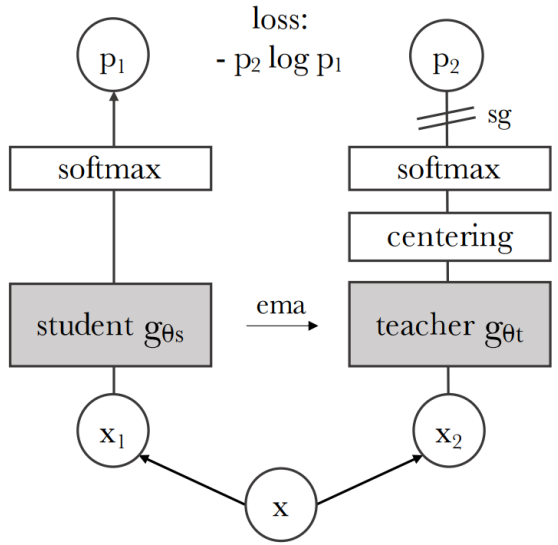
source



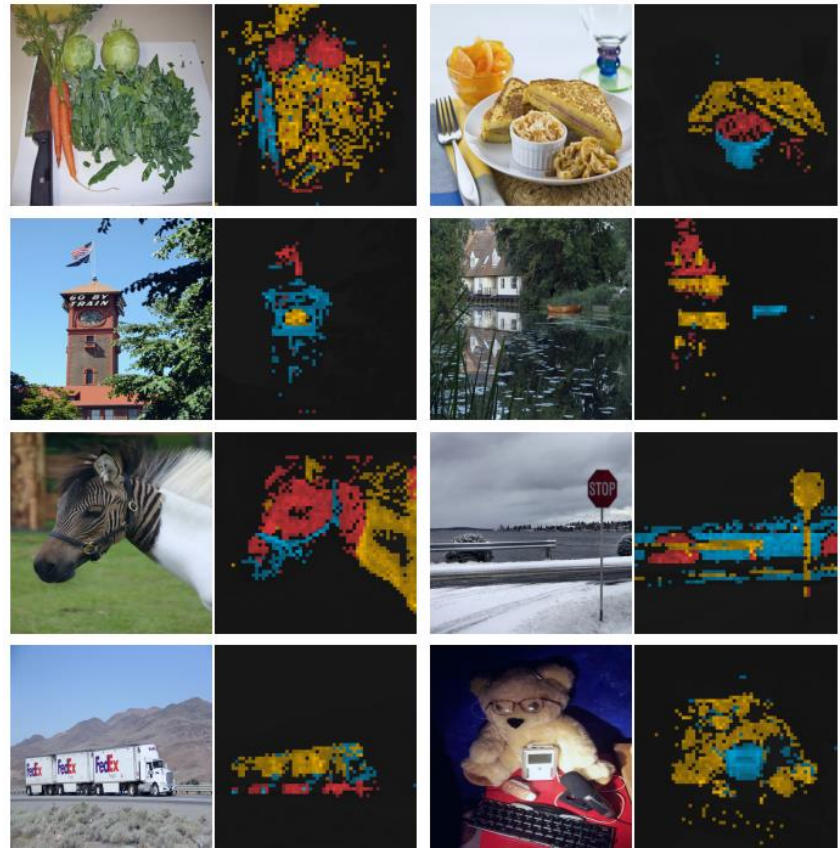
Self-Supervised Vision Transformers

DINO, DINOv2: knowledge **di**stillation with **no** labels (special form of contrastive learning)

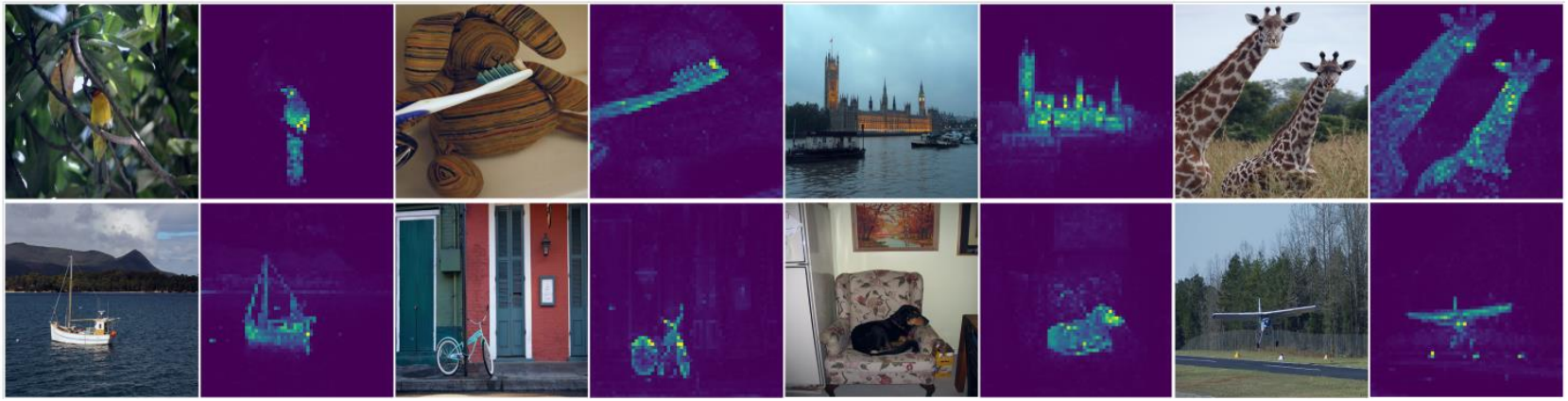
foundation model: extraction of self-supervised ViT features



different attention heads:



self-attention of last layer:

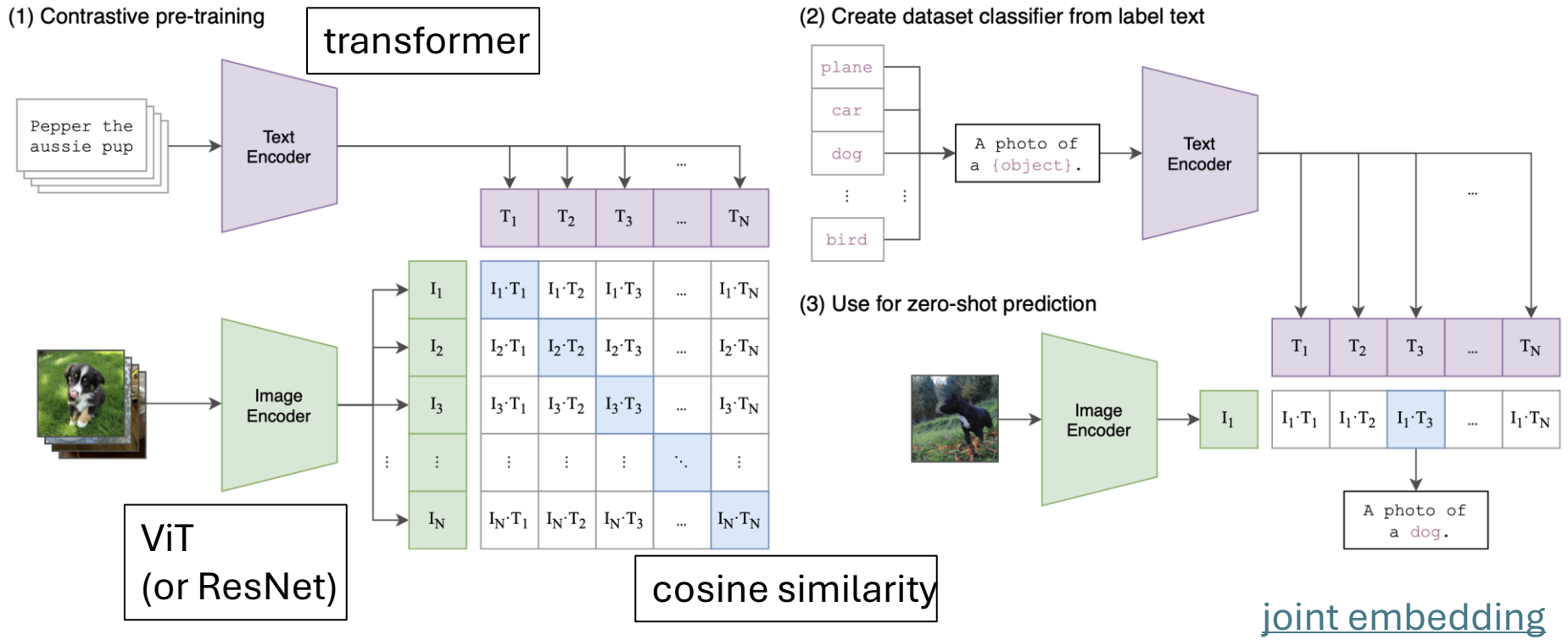


Combination of Vision and Text

example: CLIP (Contrastive Language-Image Pre-training)

learn image representations by predicting which caption goes with which image (pretraining)

→ zero-shot transfer (e.g., for image classification)



joint embedding