Deep Learning

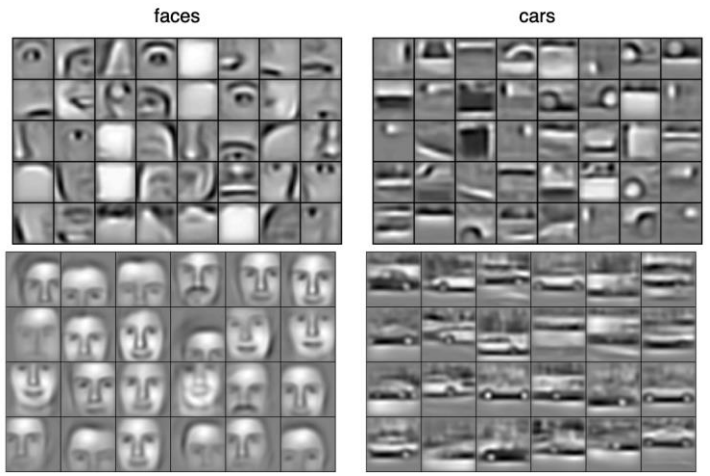
Deep Learning and Image Processing

Ladder of Generalization

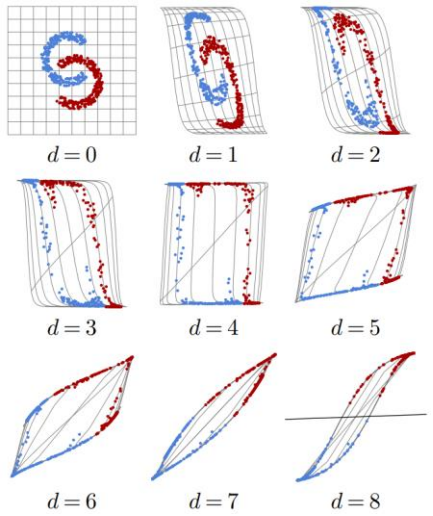
classic ML: feature engineering

deep learning: feature learning

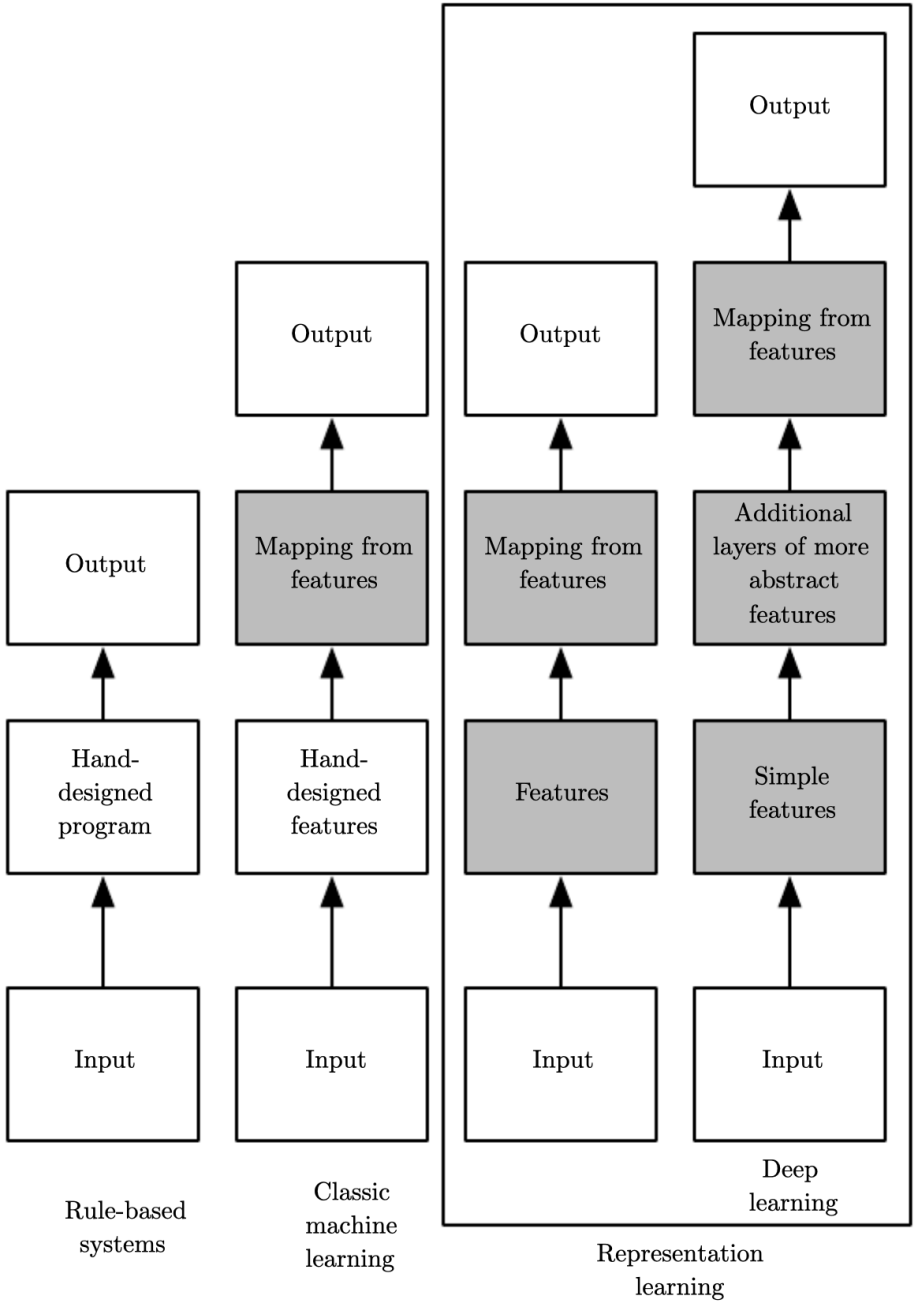
(hierarchy of concepts learned from raw data in deep graph with many layers)



source



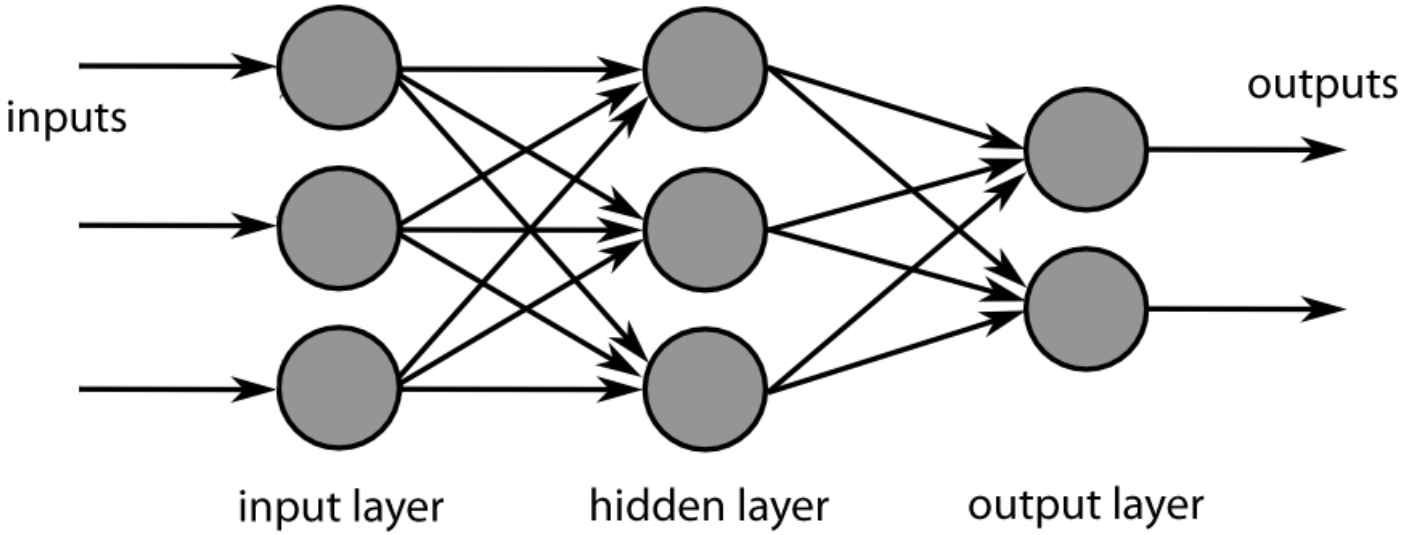
source



source

Neural Networks

idea: powerful algorithm by combining many linear building blocks



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

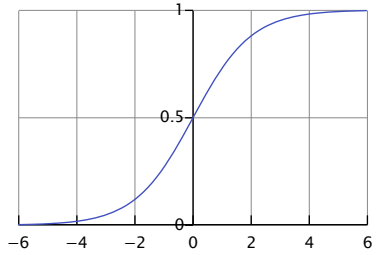
Artificial Neuron

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

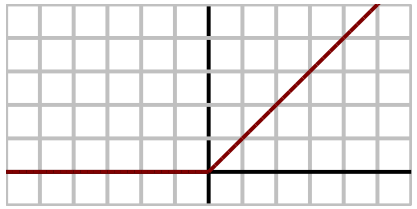
non-linear via (differentiable) activation function on top of linear model

examples:

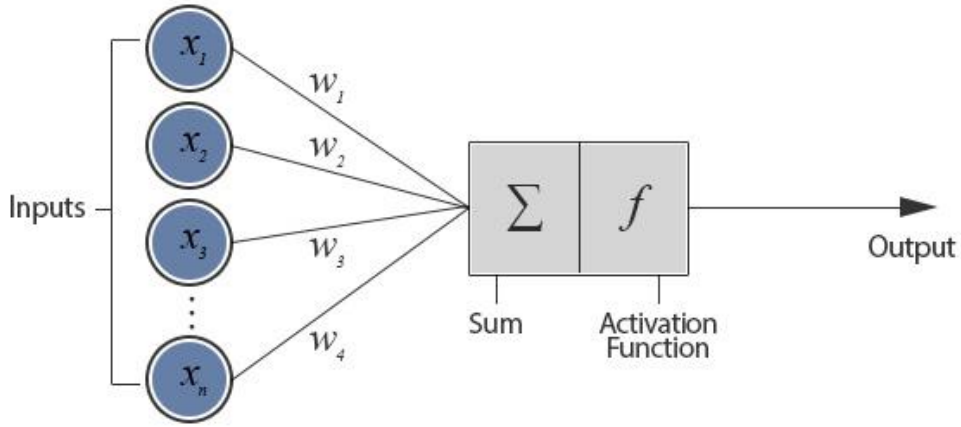
- sigmoid



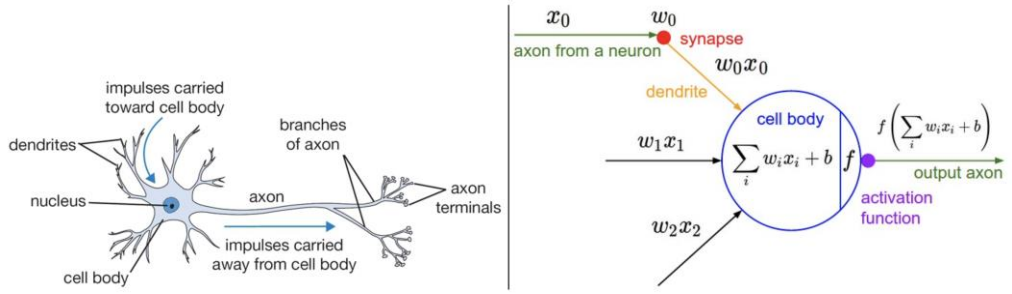
- ReLU (Rectified Linear Unit)



artificial neuron (node in neural network):



inspired from biological neurons:

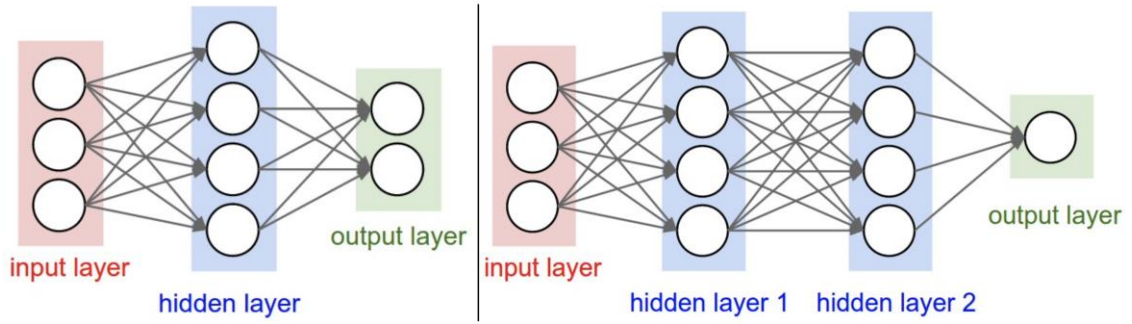


source

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:

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

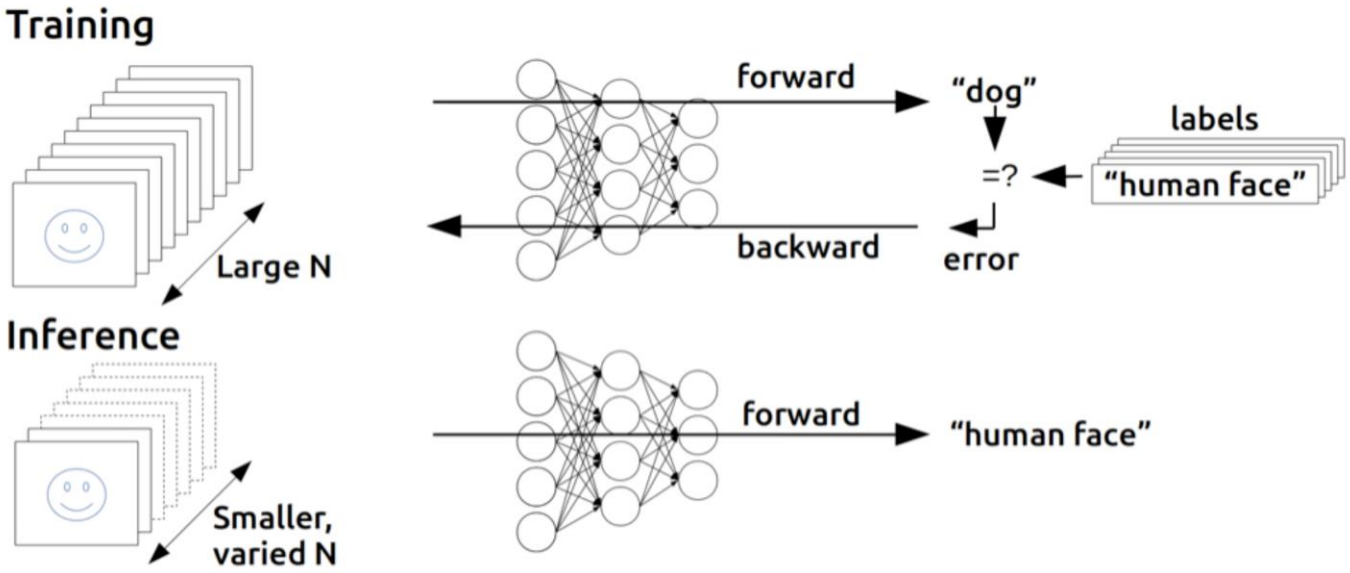
regression networks:

- 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)

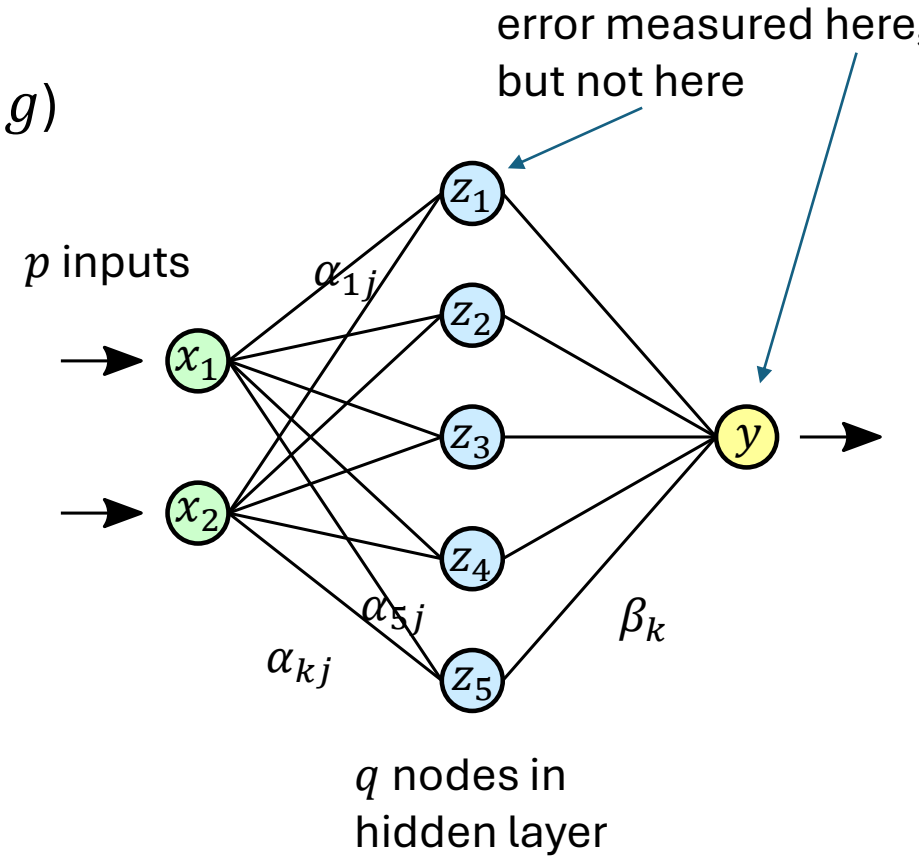
Example WOLOG

- 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

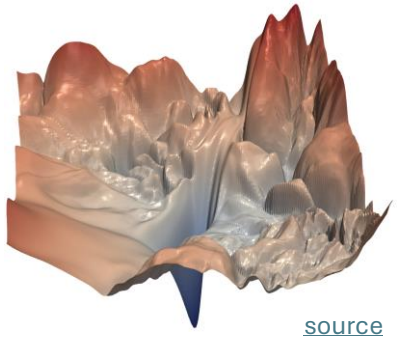
How to Train Deep Neural Networks Effectively?

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:

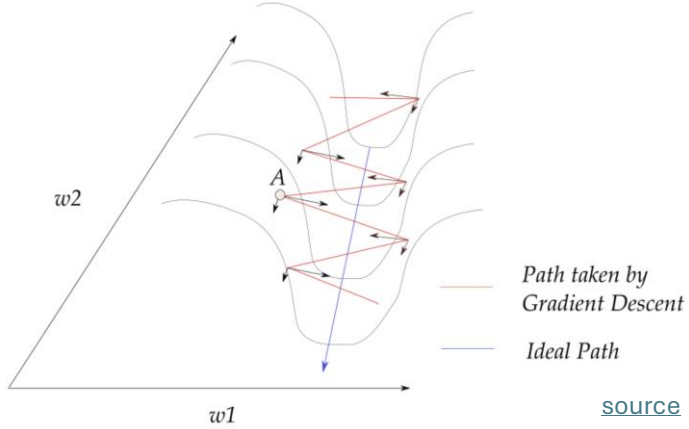
- estimate gradient $\boldsymbol{g}$
- compute velocity update: $\boldsymbol{v} \leftarrow \alpha\, \boldsymbol{v} - \eta\, \boldsymbol{g}$
- apply parameter update: $\widehat{\boldsymbol{\theta}} \leftarrow \widehat{\boldsymbol{\theta}} + \boldsymbol{v}$

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

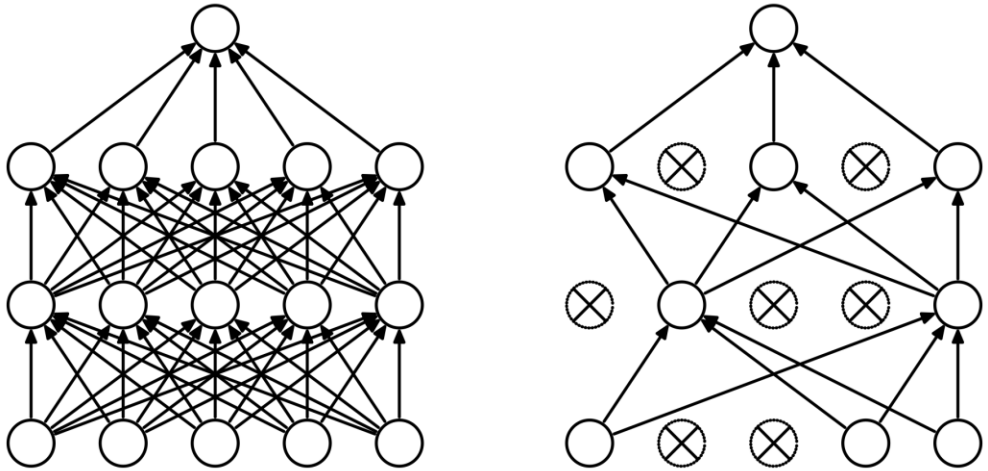
  (issue: sum in denominator grows with more iterations → can get stuck)
- RMSProp: $\hat{w} \leftarrow \hat{w} - \frac{\eta}{\sqrt{v(\hat{w})}} g_{\hat{w}}$ with $v(\hat{w}) \leftarrow \gamma\, v(\hat{w}) + (1-\gamma) g_{\hat{w}}^2$
- Adam (Adaptive Moment Optimization): combines RMSProp with momentum

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

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

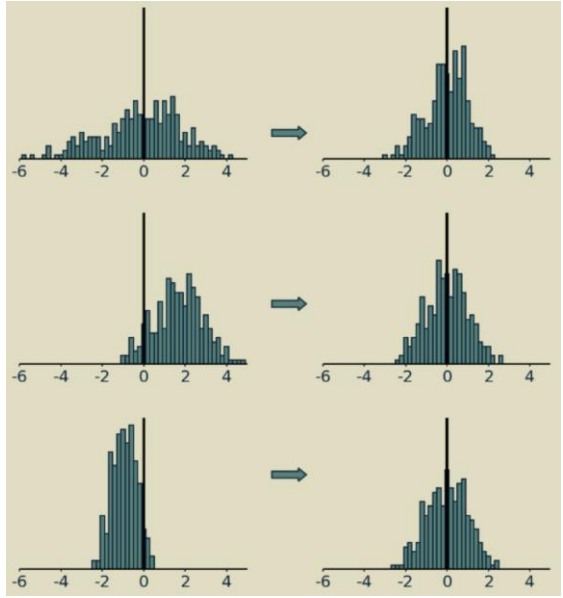
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

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

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

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

source



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

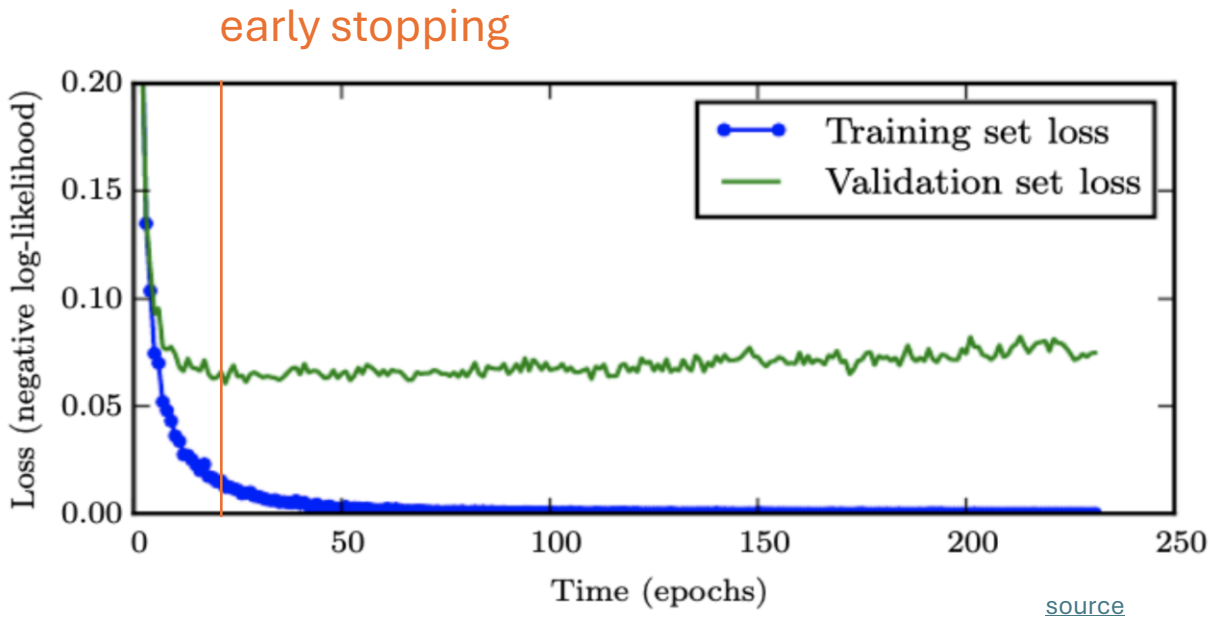
reason why batch normalization improves optimization still controversial

most plausible explanation: smoothening of loss landscape

Early Stopping

loss independently measured on validation set

halting training when overfitting begins to occur

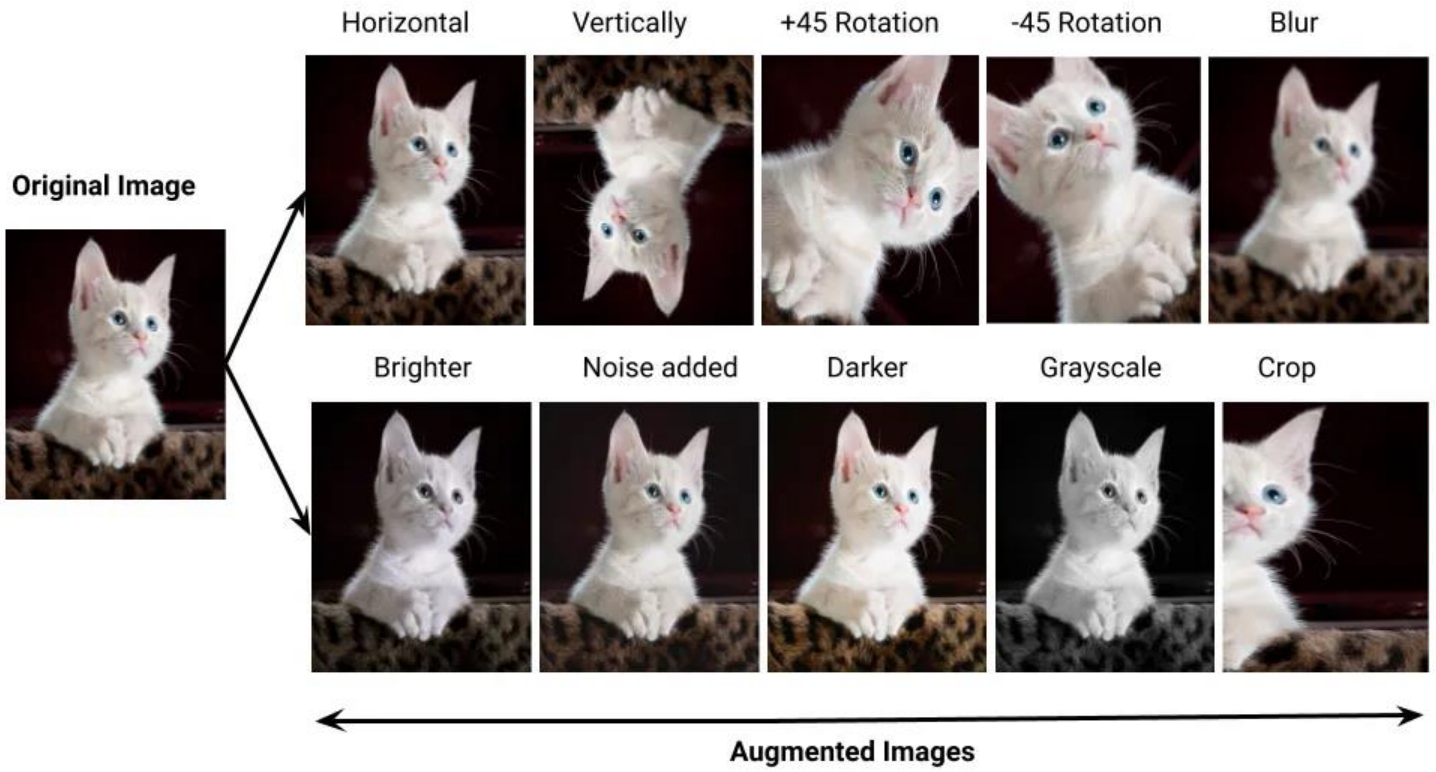


source

Data Augmentation

adding different variations of input data to training

idea: supporting the model in maintaining desired invariances



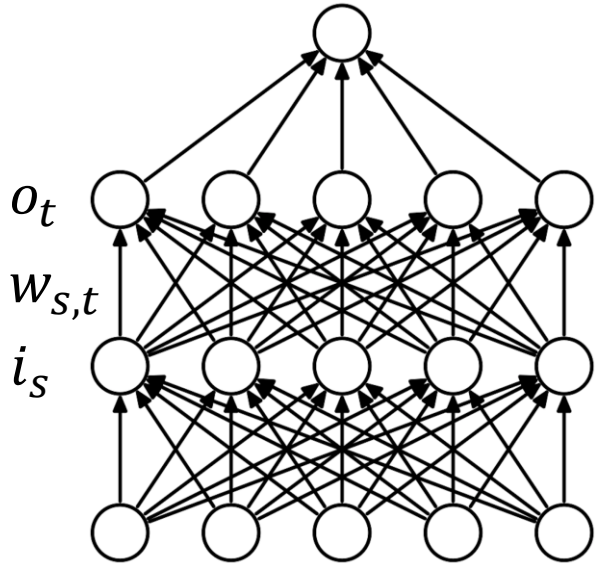
Convolutional Neural Networks (CNN or ConvNet)

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

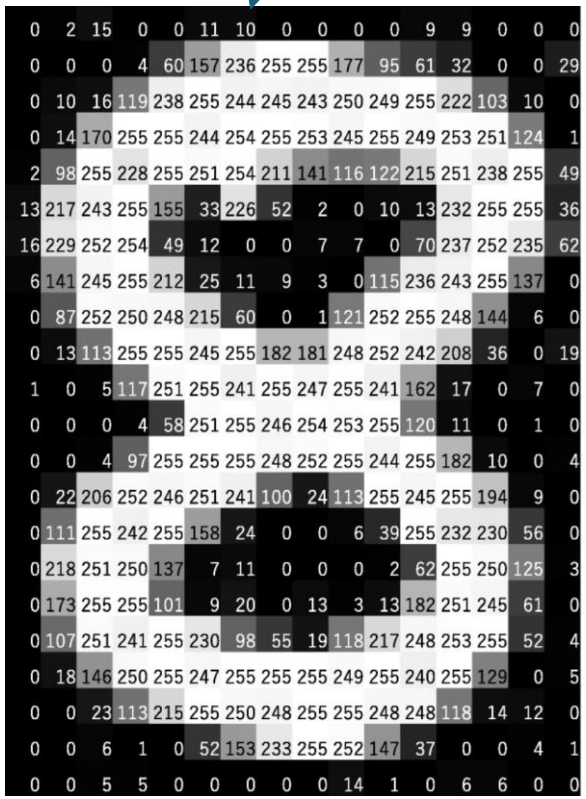
image data: 2-D grid of pixels (spatial structure)

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):

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

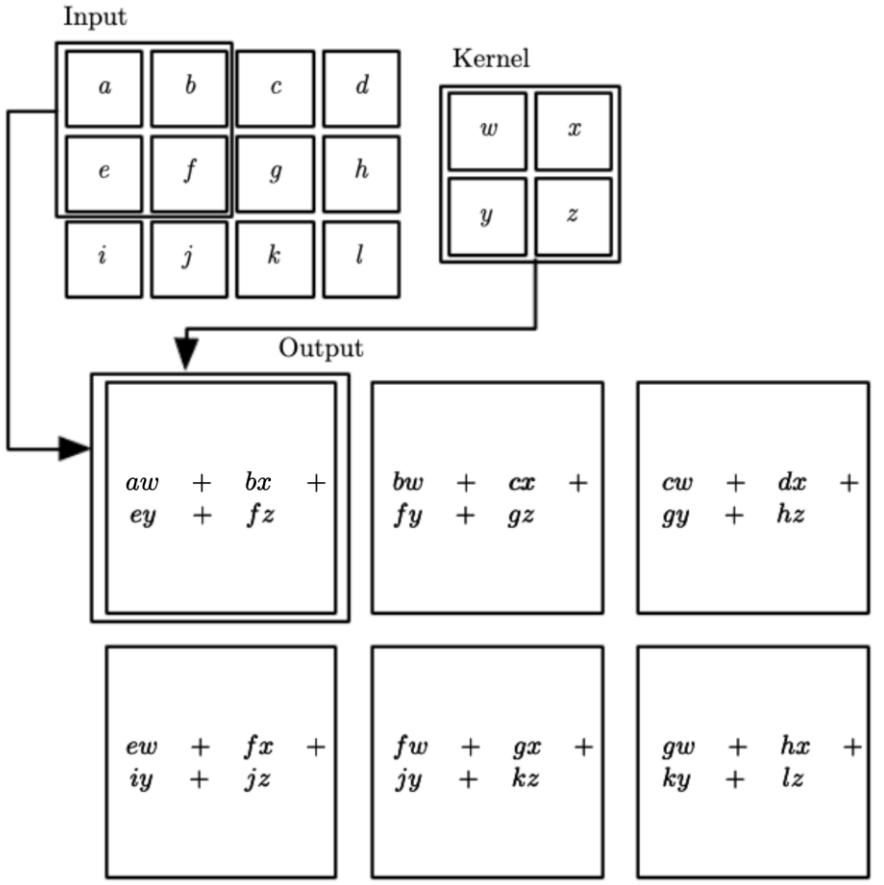
feature map
(transformed image)

kernel
(learned)

image
(input or feature map)

(again, dropping dimension of different training observations)

2D convolution

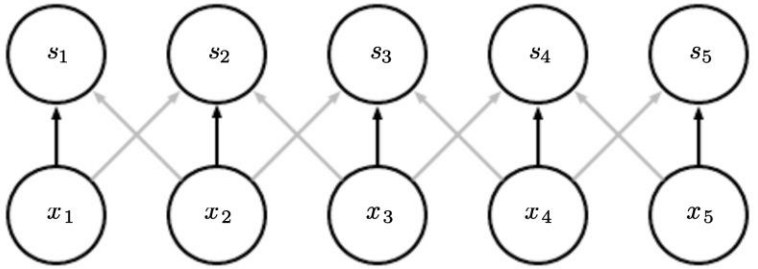


source

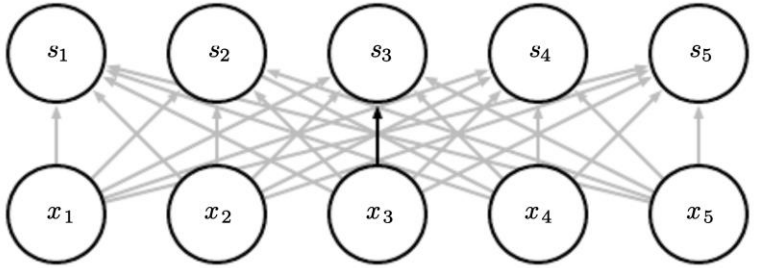
Regularization Effects

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

convolutional layer:



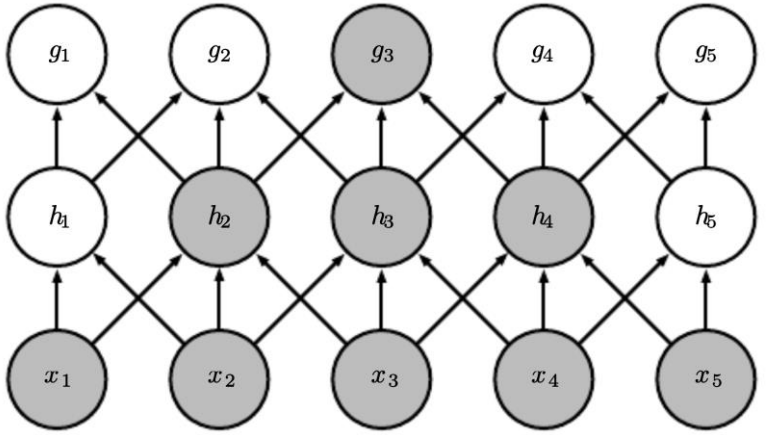
fully-connected layer:



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)

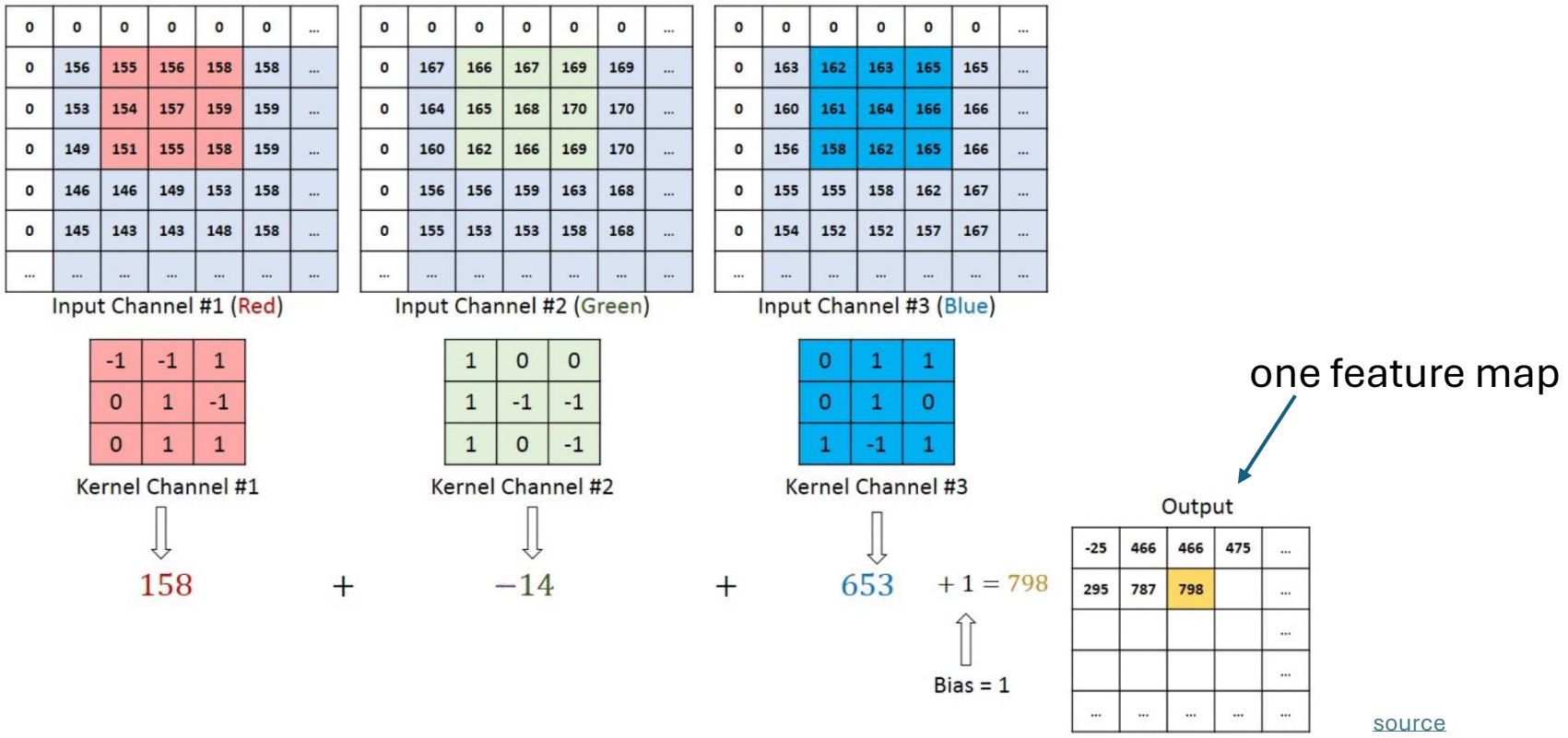


source

Channel Mixing

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

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



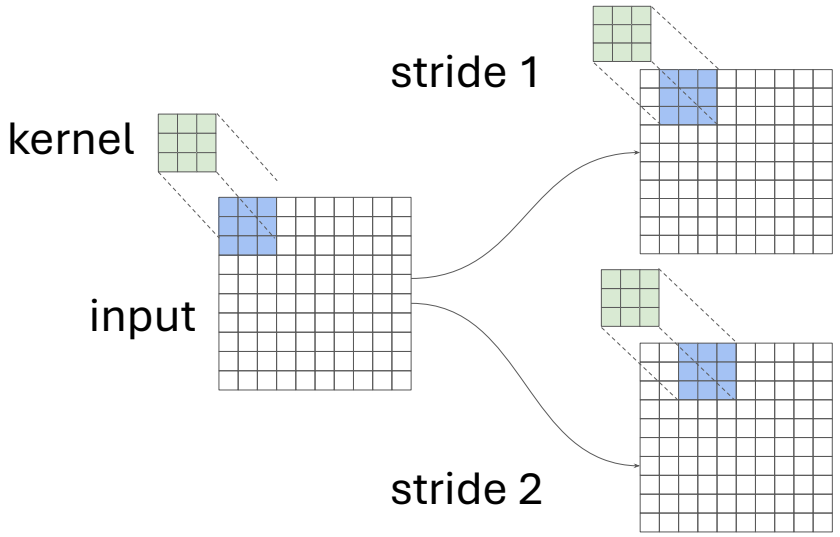
source

Striding and Padding

need to define how to stride over image

stride > 1 corresponds to down-sampling

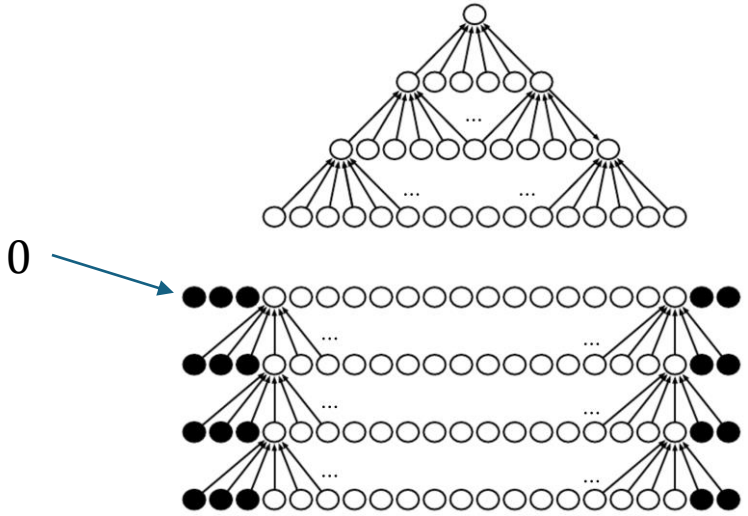
→ fewer nodes after convolutional layer



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



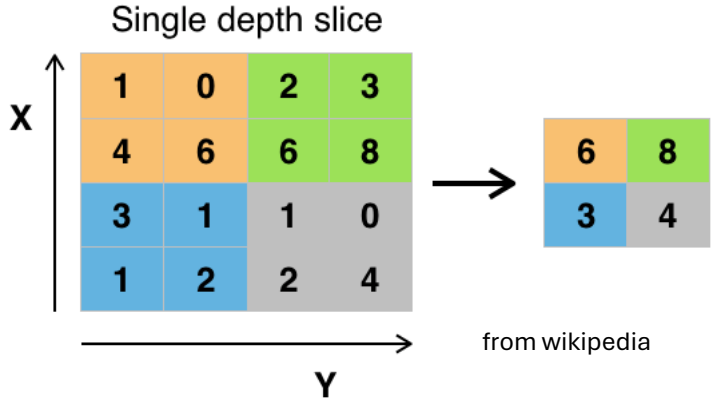
source

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:

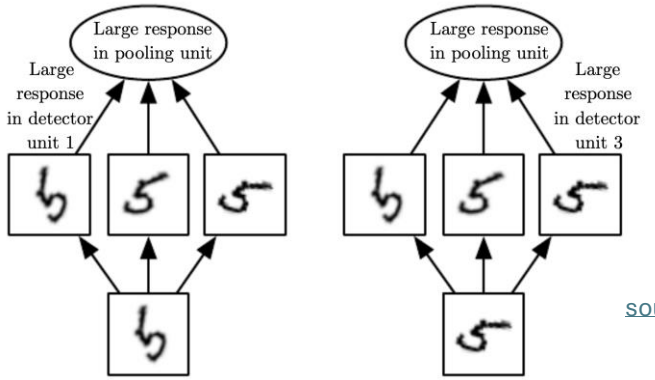


from wikipedia

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

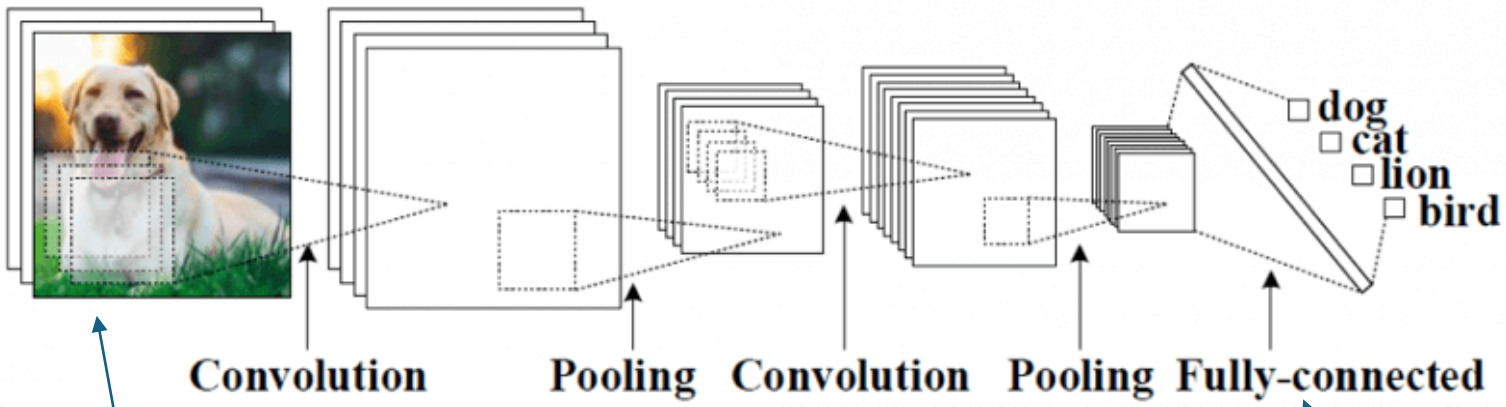
(convolutions can detect same translated motif across entire image, but not rotated or scaled versions of it)



source

Putting It All Together

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

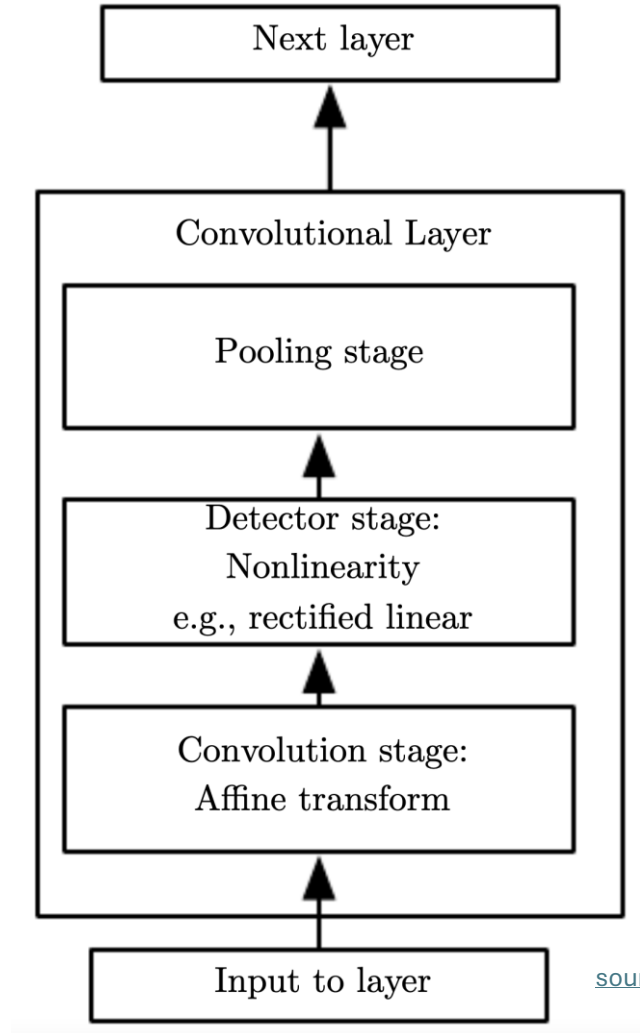


many images (training examples), potentially with several channels

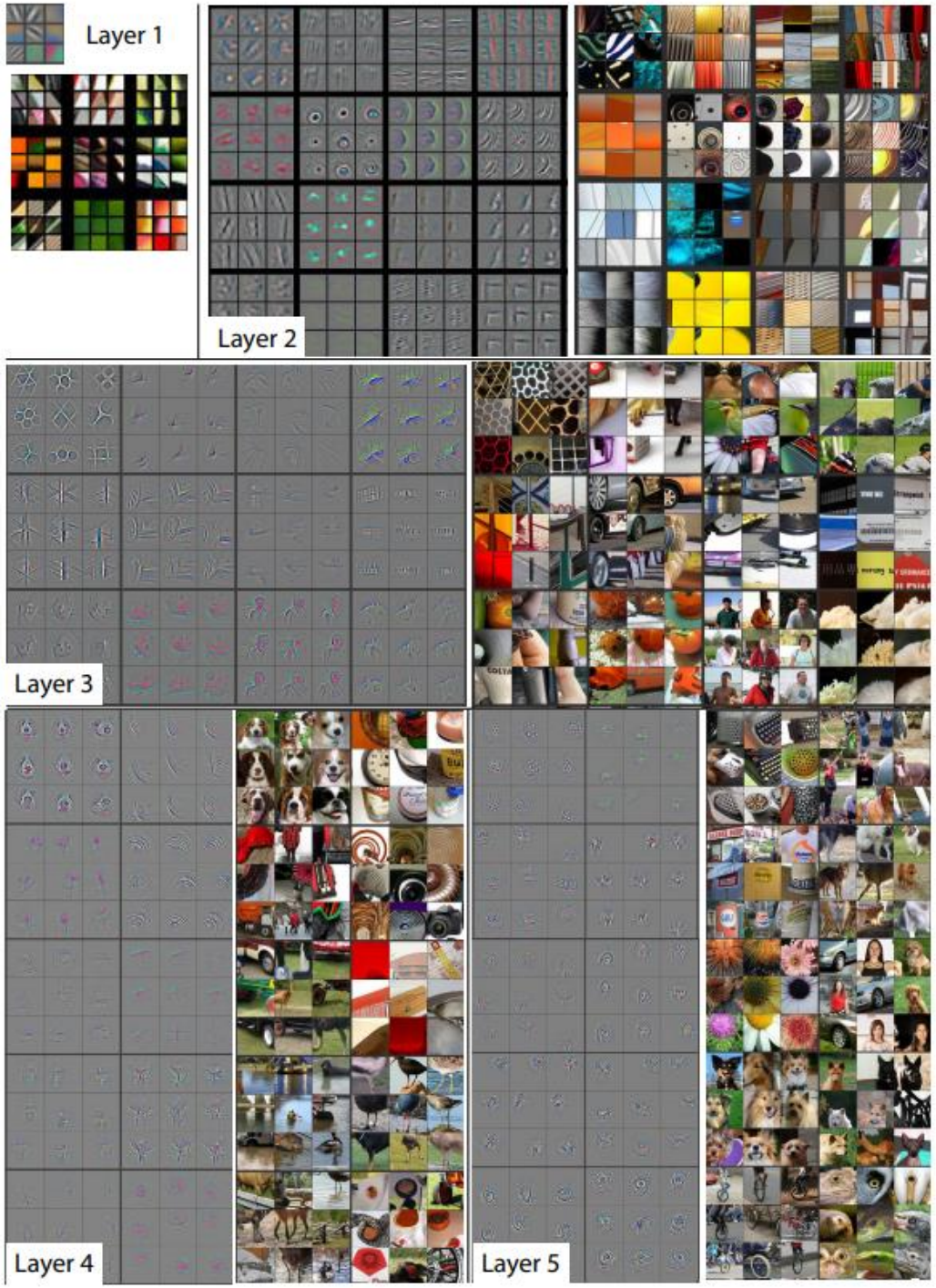
several kernels producing several feature maps

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

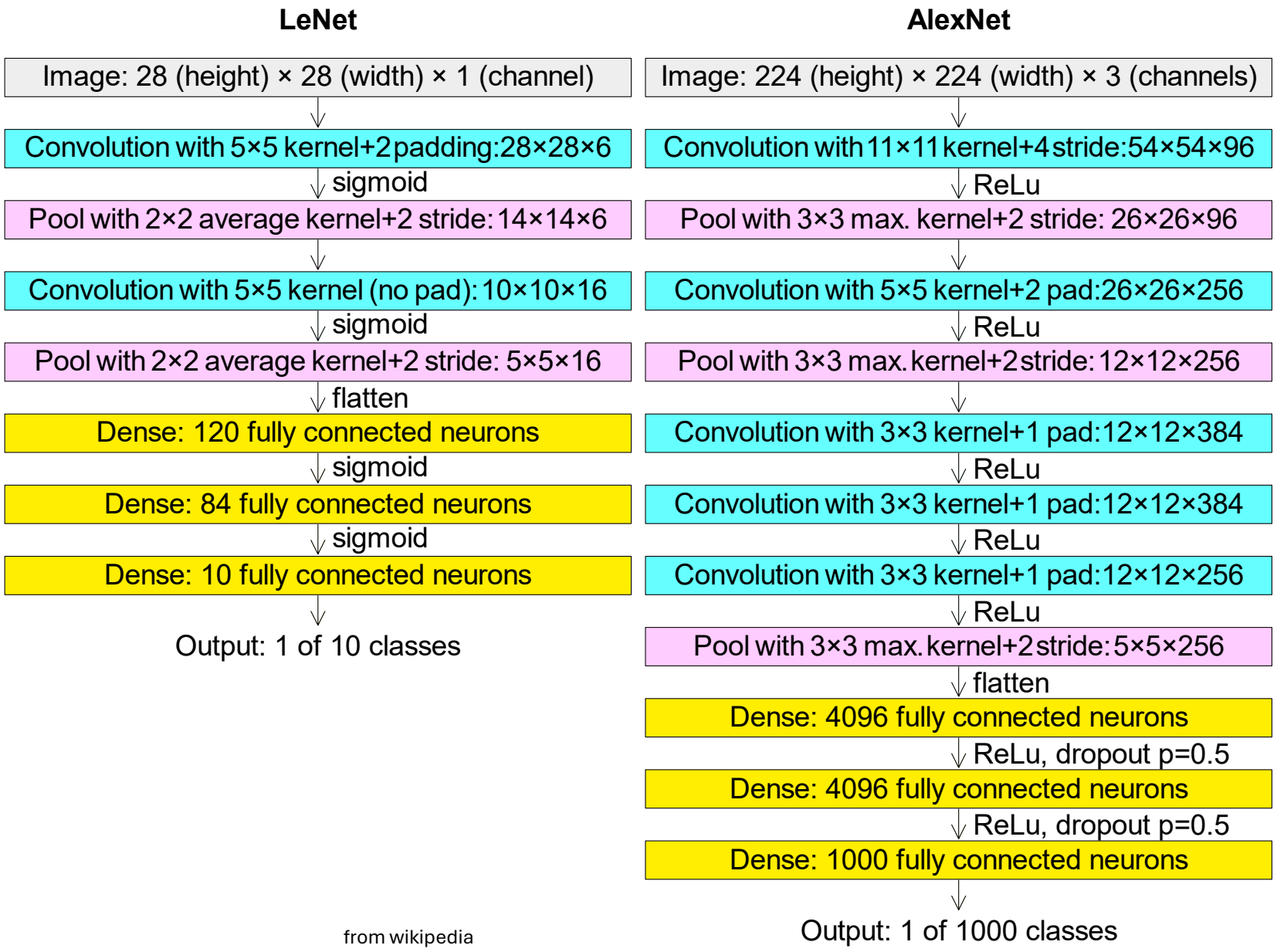
t-SNE Visualization

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

Going Deeper



from wikipedia

AlexNet finally started the deep learning hype.

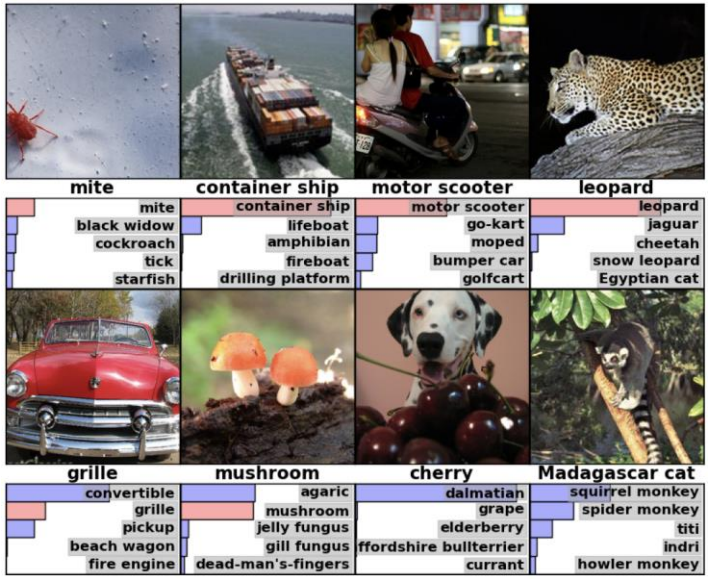
(winning the ImageNet challenge in 2012)

Rise of Deep Learning

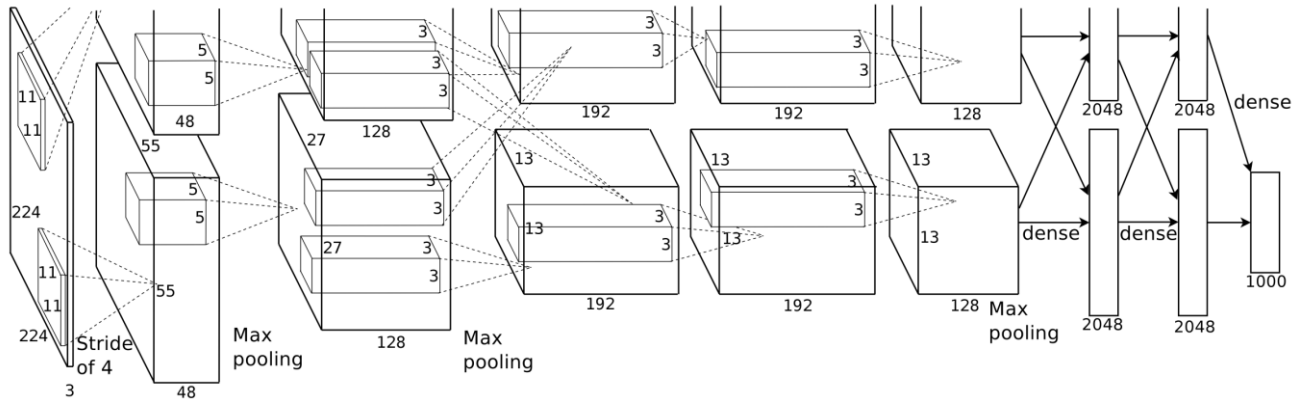
a little bit oversimplified:

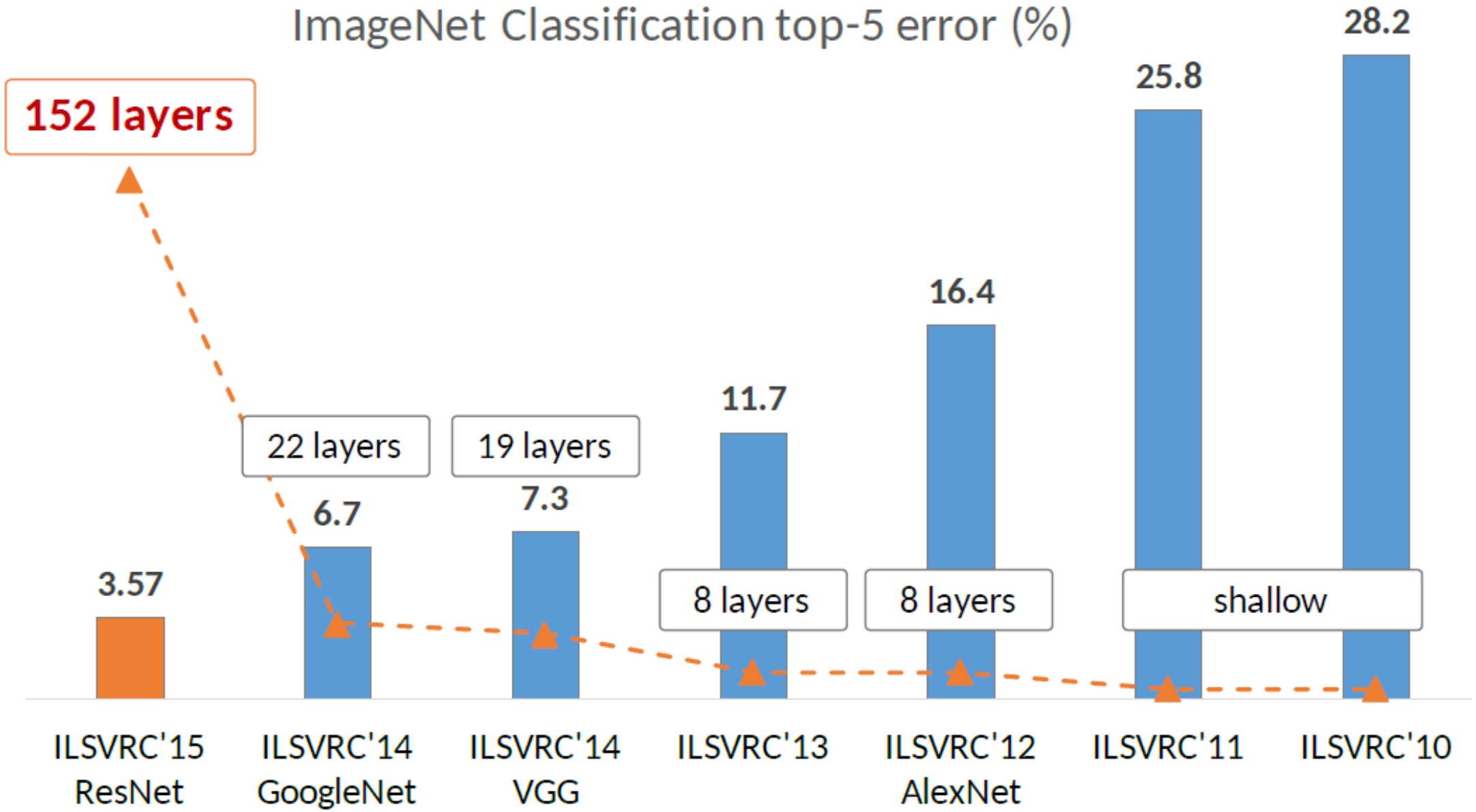
deep learning = lots of training data + parallel computation + smart algorithms

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

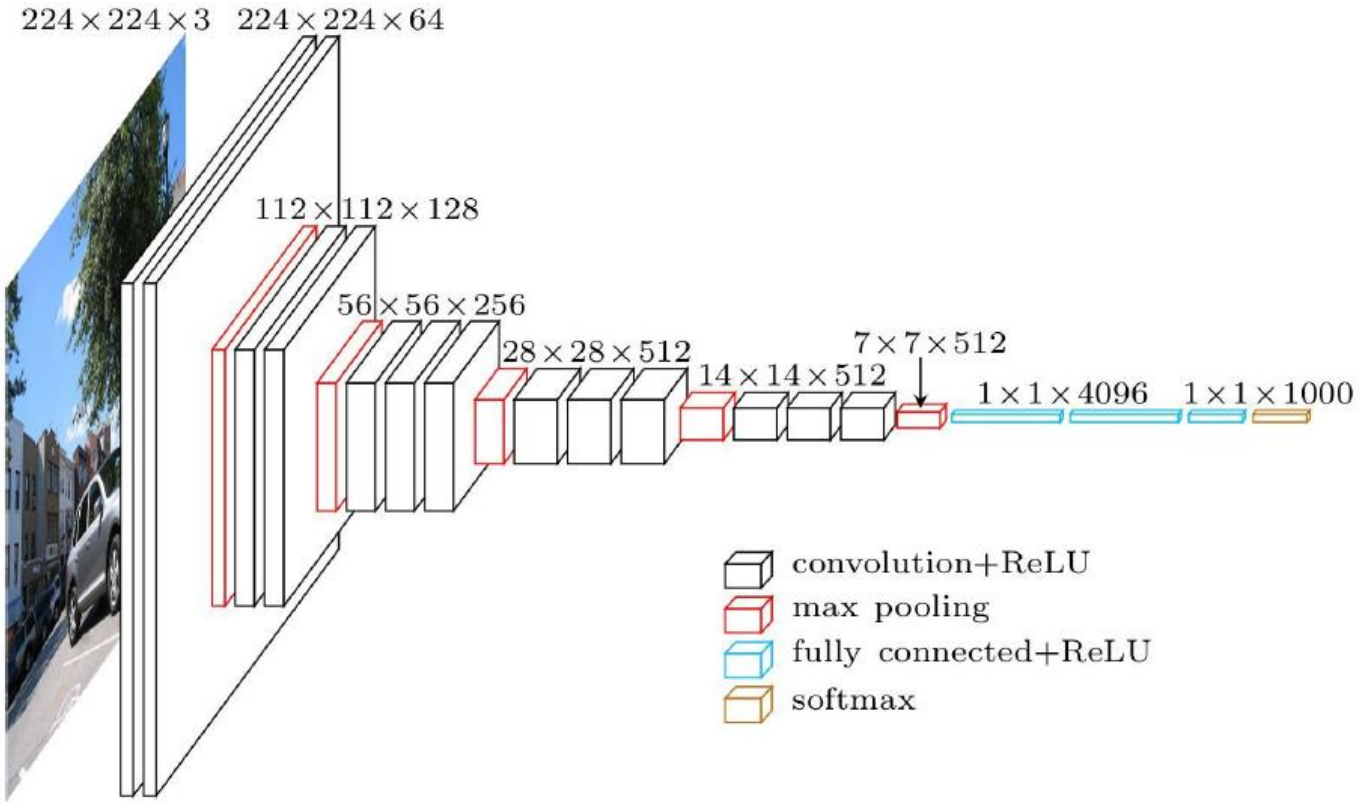


source

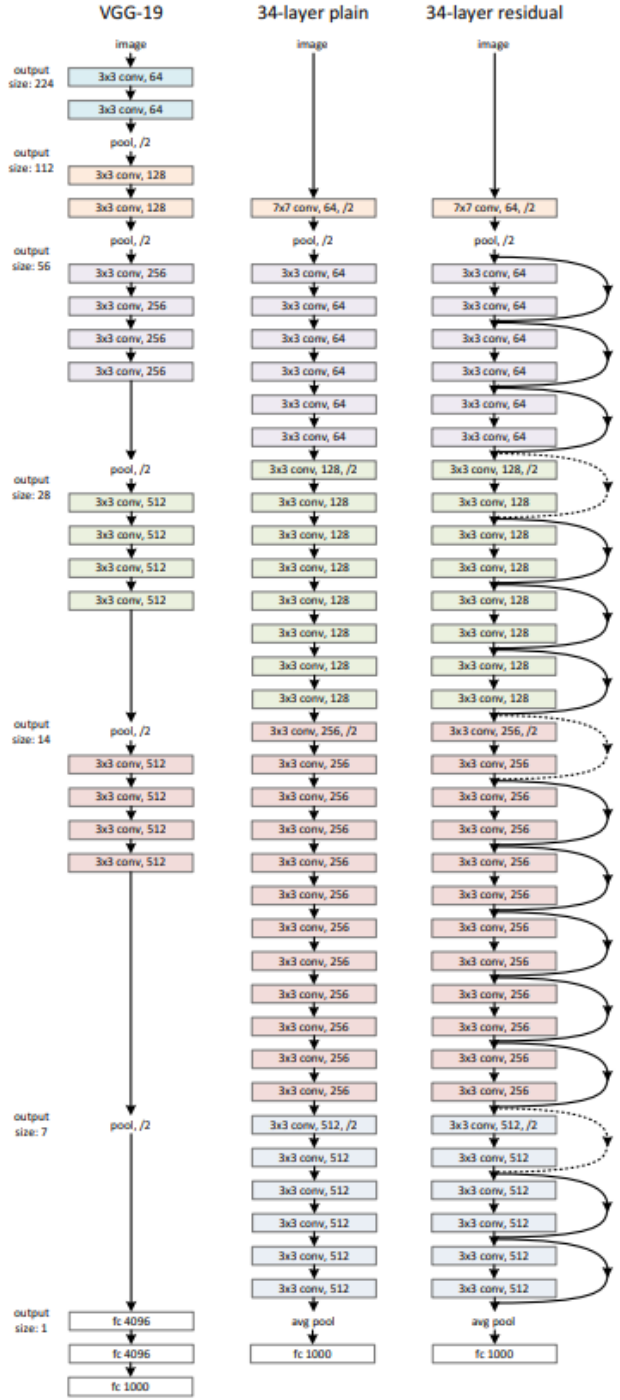



ImageNet Classification top-5 error (%)
152 layers
22 layers
19 layers
8 layers
8 layers
shallow
3.57
6.7
7.3
11.7
16.4
25.8
28.2
ILSVRC'15 ResNet
ILSVRC'14 GoogleNet
ILSVRC'14 VGG
ILSVRC'13
ILSVRC'12 AlexNet
ILSVRC'11
ILSVRC'10

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
output size: 224
3x3 conv, 64
pool, /2
output size: 112
3x3 conv, 128
output size: 56
3x3 conv, 256
output size: 28
3x3 conv, 512
output size: 14
output size: 7
fc 4096
fc 1000
7x7 conv, 64, /2
3x3 conv, 128, /2
3x3 conv, 256, /2
3x3 conv, 512, /2
avg pool

Skip Connections

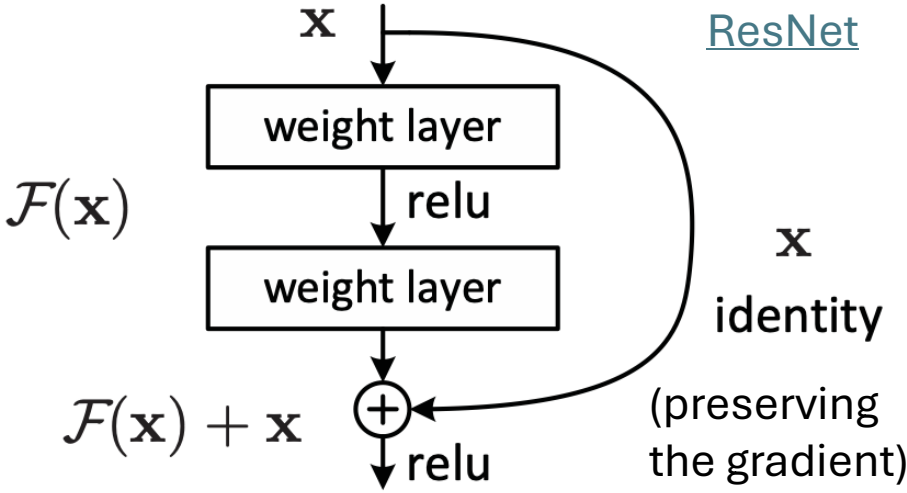
issue: degradation of training and test errors when adding more and more layers → not due to overfitting (but reason controversial)

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

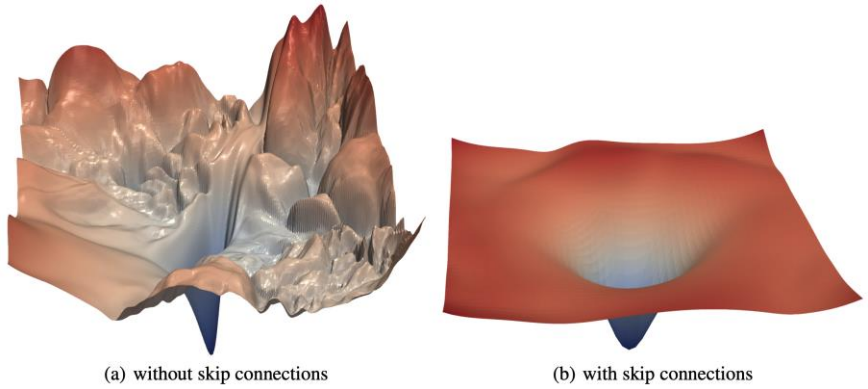
→ produces loss functions that train easier

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):



loss surface:



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)

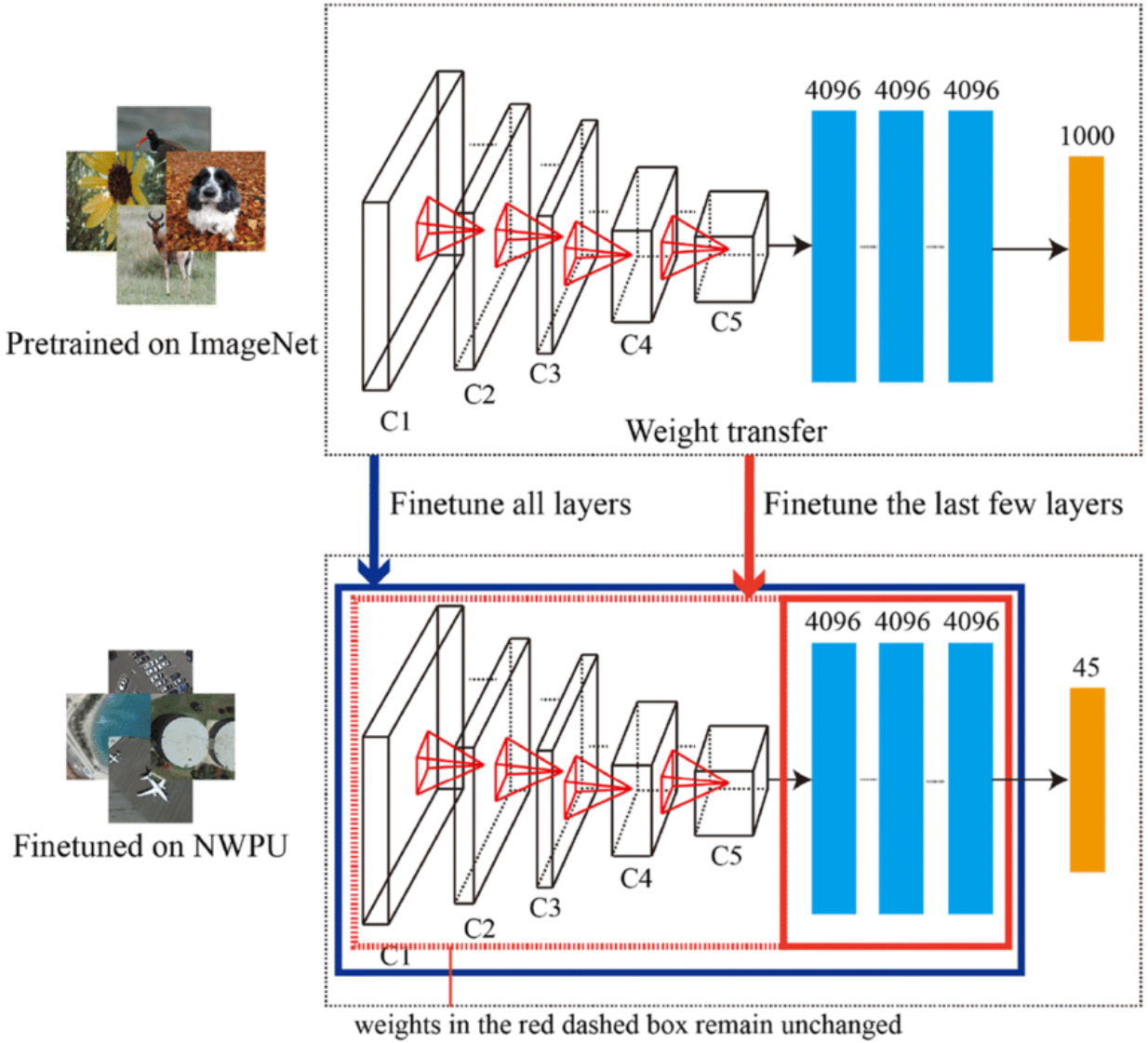
Finetuning

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

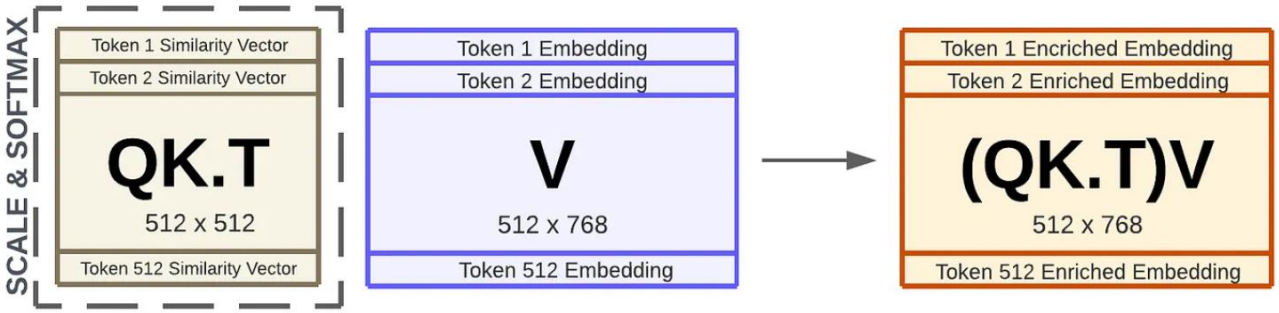
fully started with ChatGPT release end of 2022

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



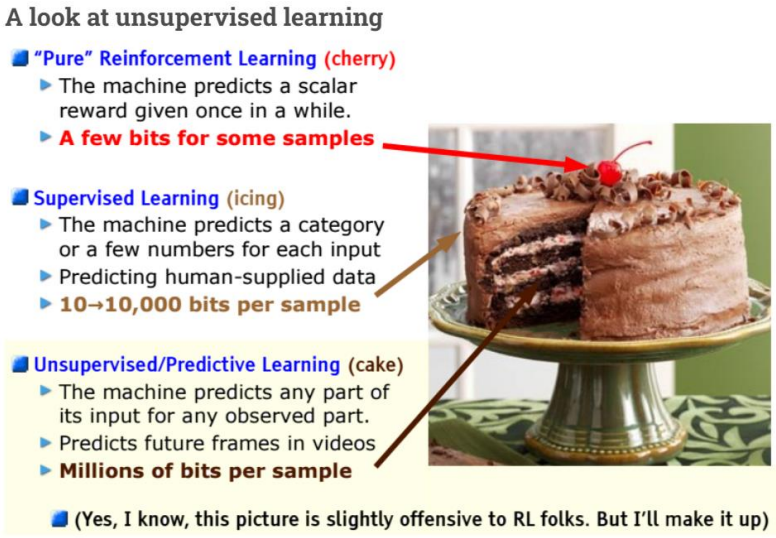
source

ML needs lots of training data

Self-Supervised Learning

unsupervised learning (learning by observation)

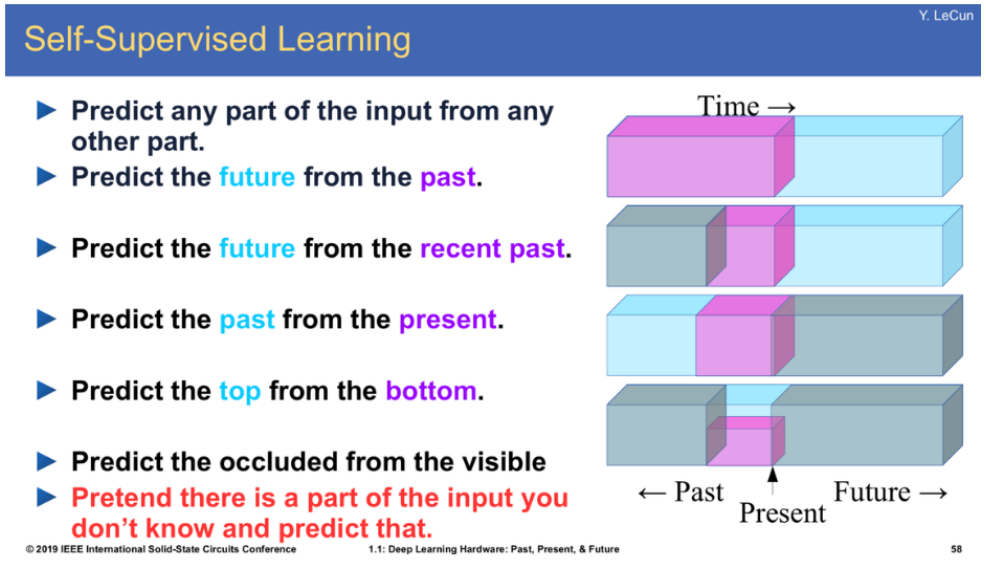
no target information → kind of “vague” pattern recognition (but plenty of data)



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

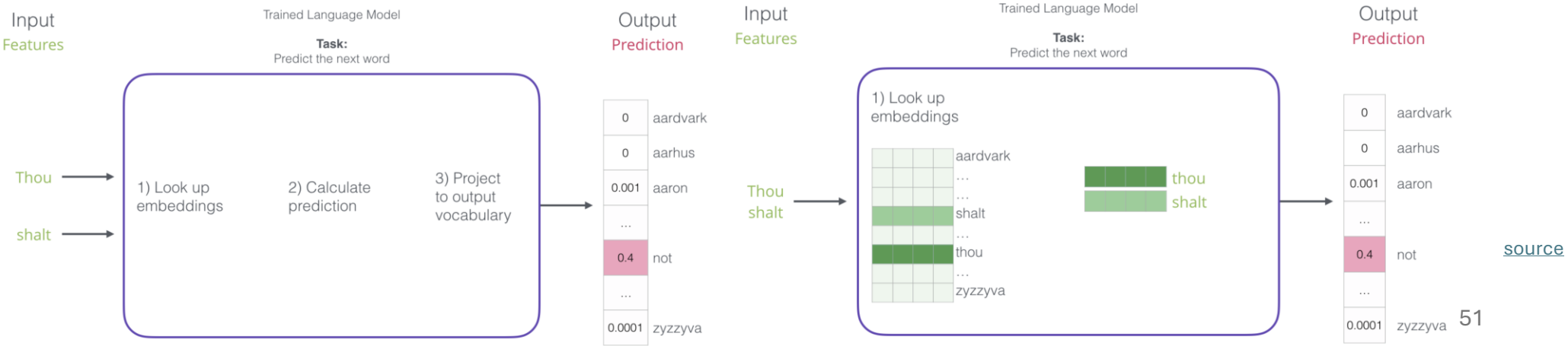
Word Embeddings as Part of Language Model

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:

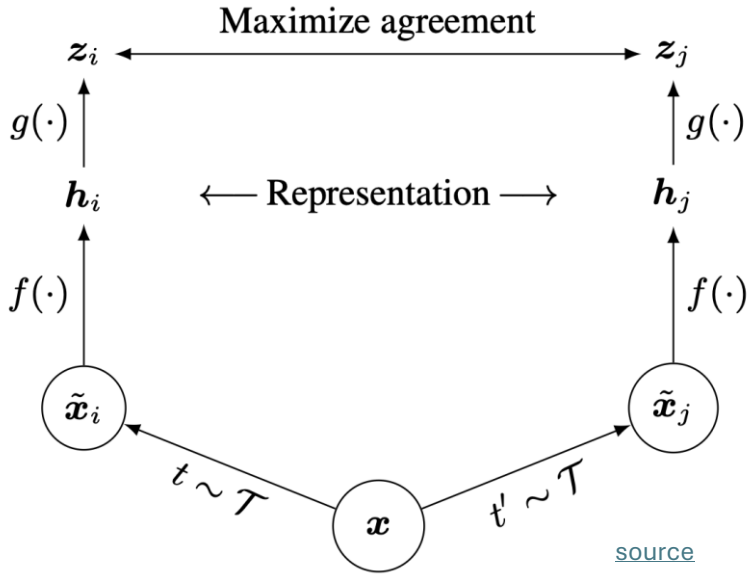


source

In General: Contrastive Learning

idea:

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

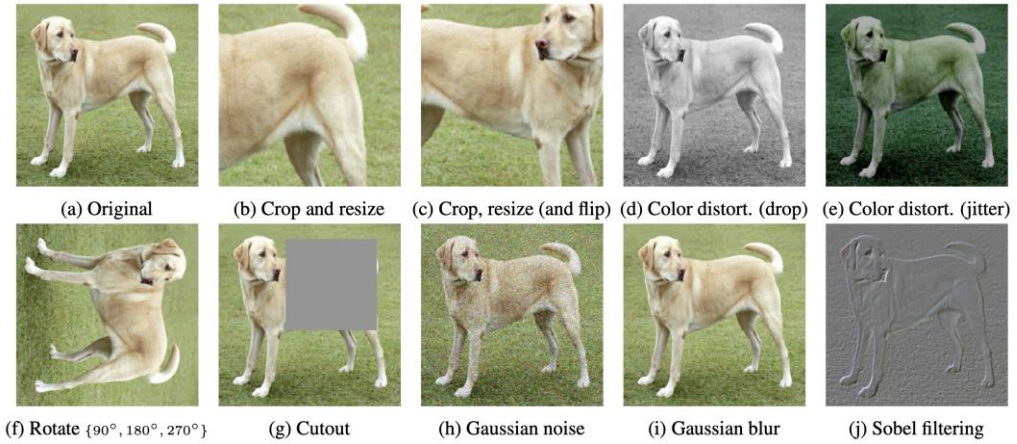


source

not only useful for language

→ learning of image representations

often learned in a self-supervised way



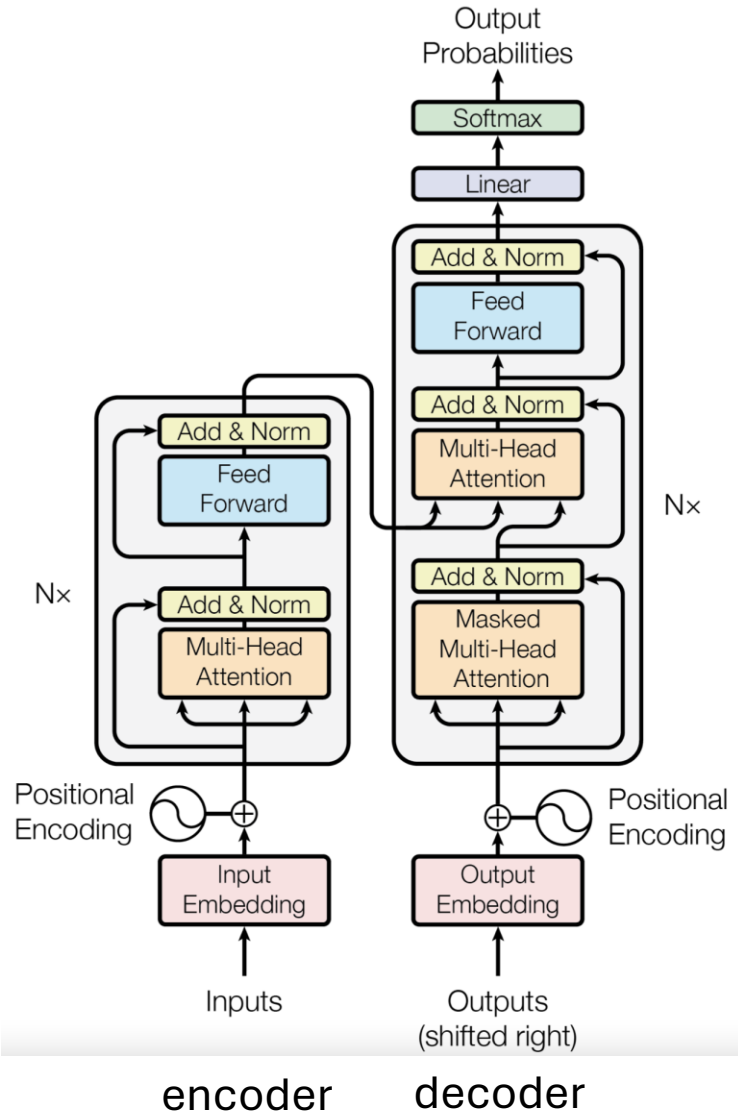
Context Awareness: Transformer

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)



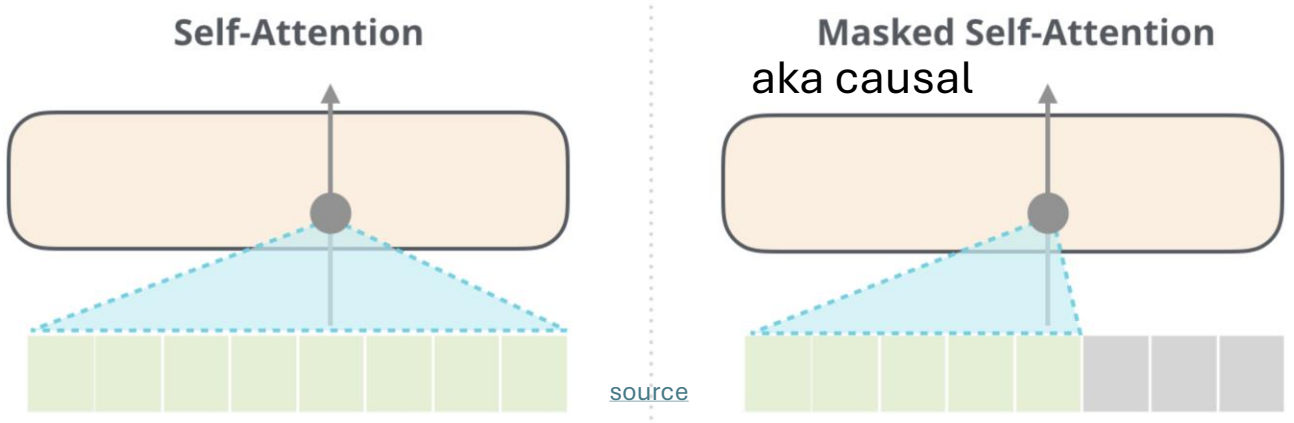
Transformer Architectures for LLMs

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

example:
BERT



source

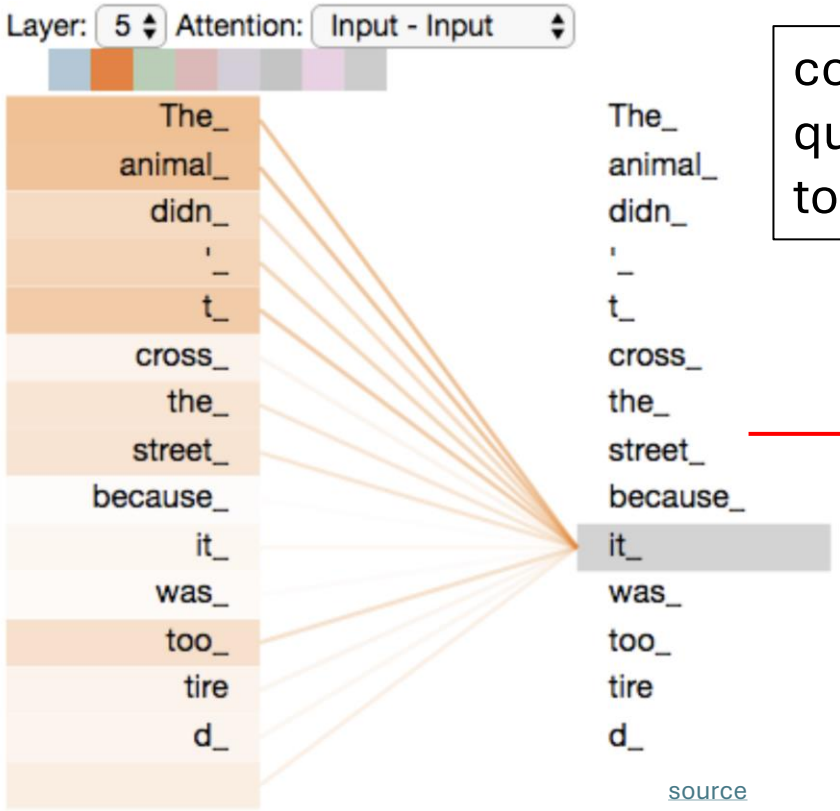
decoder-only LLMs:

- 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:
GPT-3

Self-Attention Mechanism

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

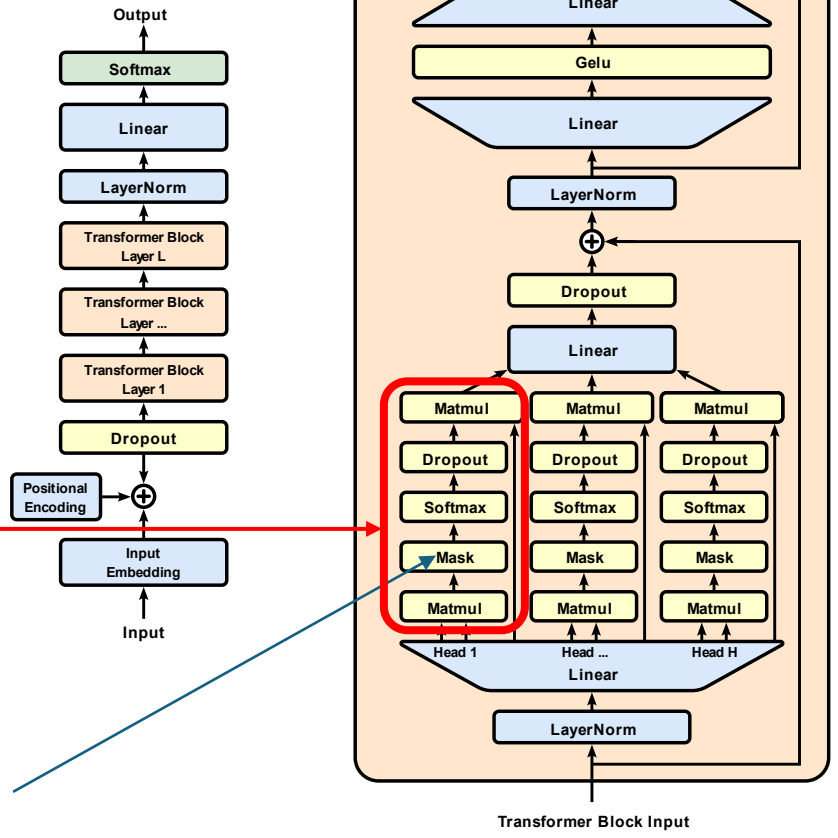


source

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)

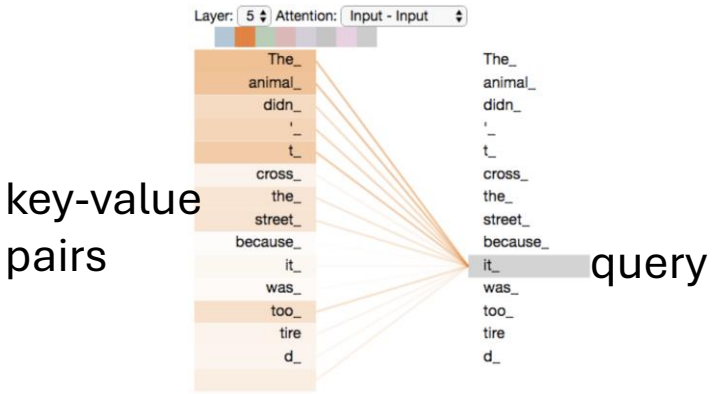
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention

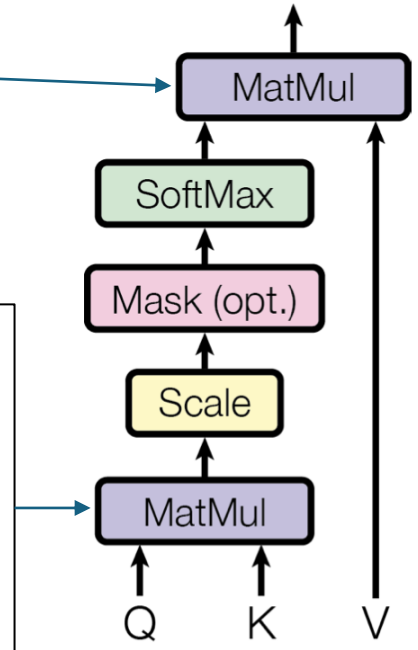
3 matrices created from input embeddings by multiplication with 3 different weight matrices

- query Q
- key K
- value V

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

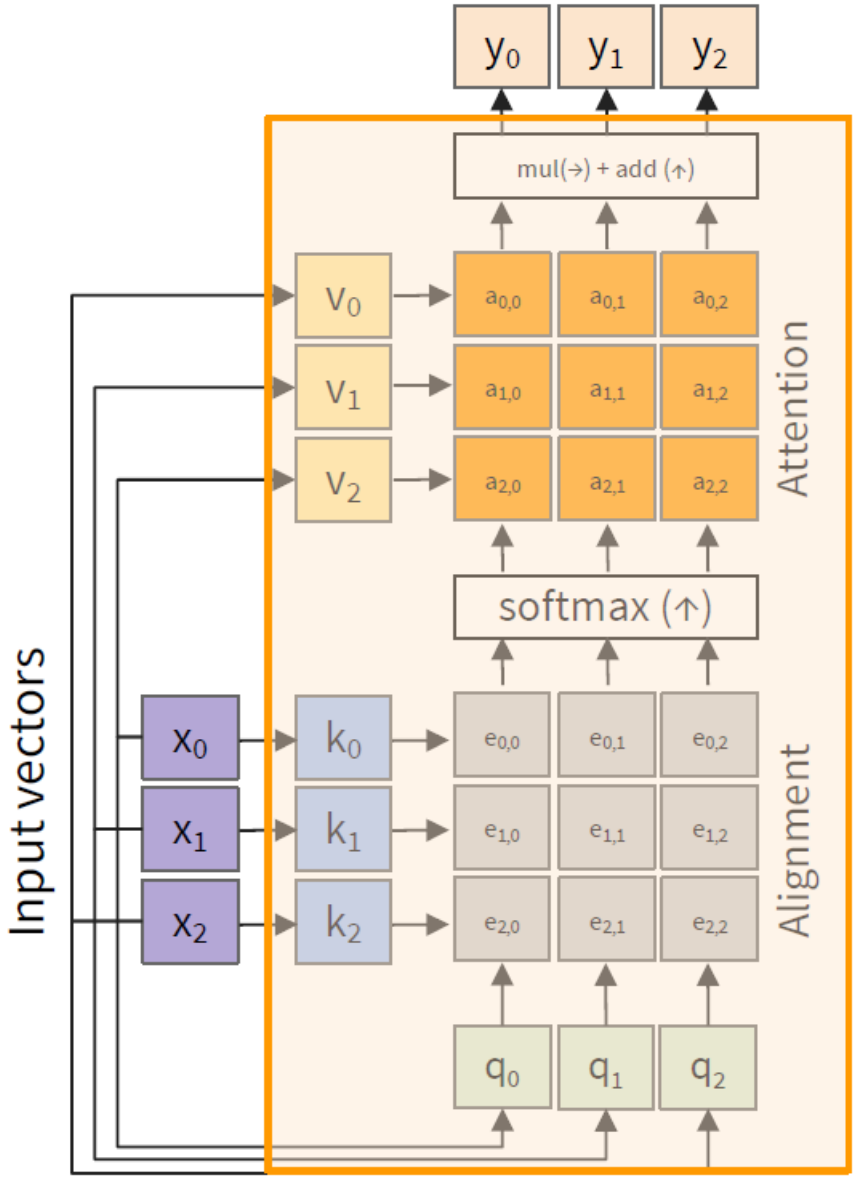


scoring each of the key words (context) with respect to current query word: multiplication of inputs (in contrast to inputs times weights in neural networks)



source

MatMul



Outputs:
context vectors: y (shape: D_v)

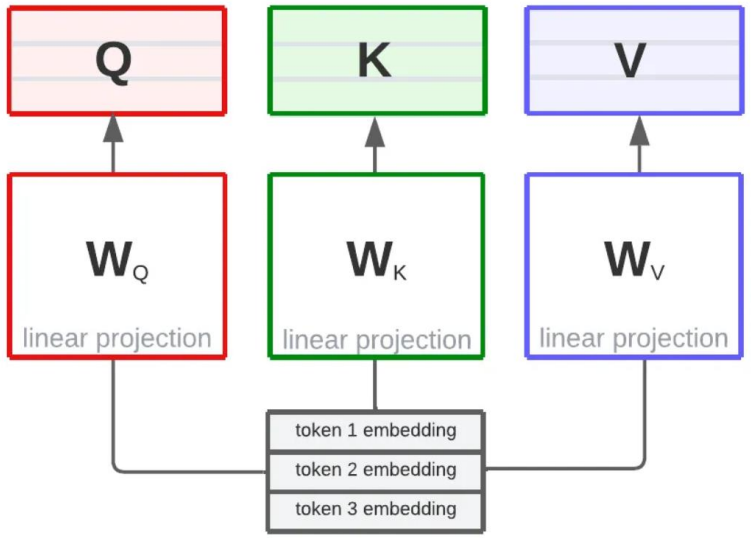
Operations:
Key vectors: $k = xW_k$
Value vectors: $v = xW_v$
Query vectors: $q = xW_q$
Alignment: $e_{i,j} = q_j \cdot k_i / \sqrt{D}$
Attention: $a = \text{softmax}(e)$
Output: $y_j = \sum_i a_{i,j} v_i$

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

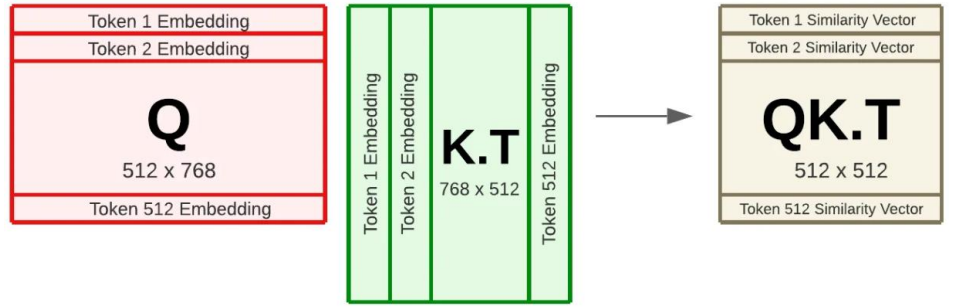
1



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

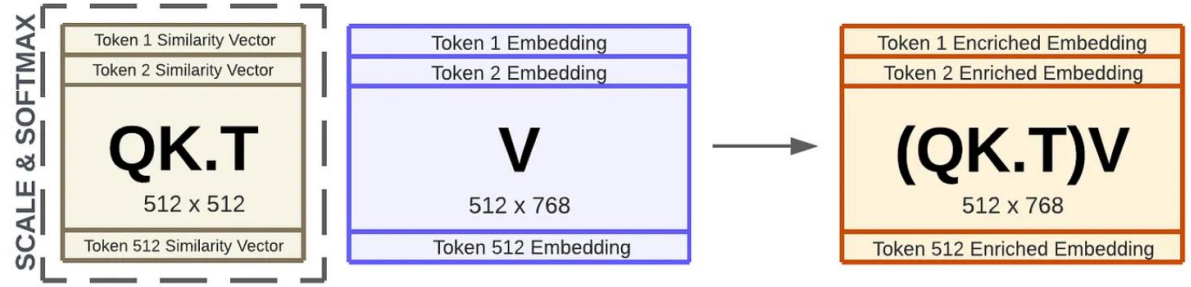
source

2



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

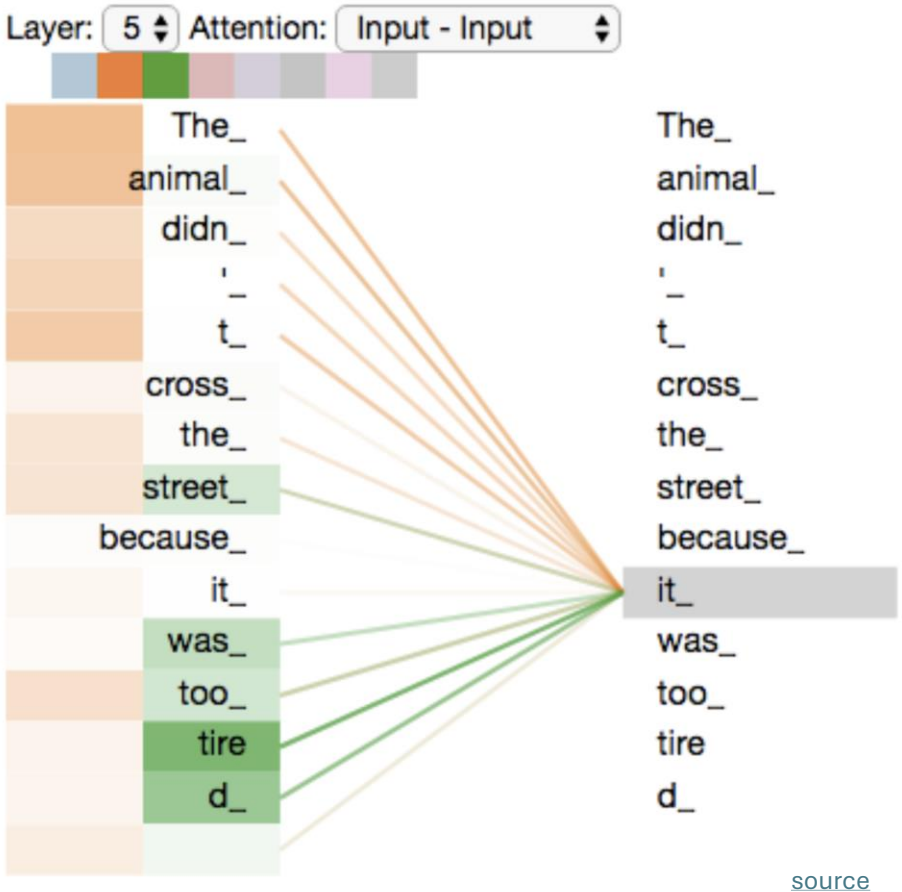
3



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

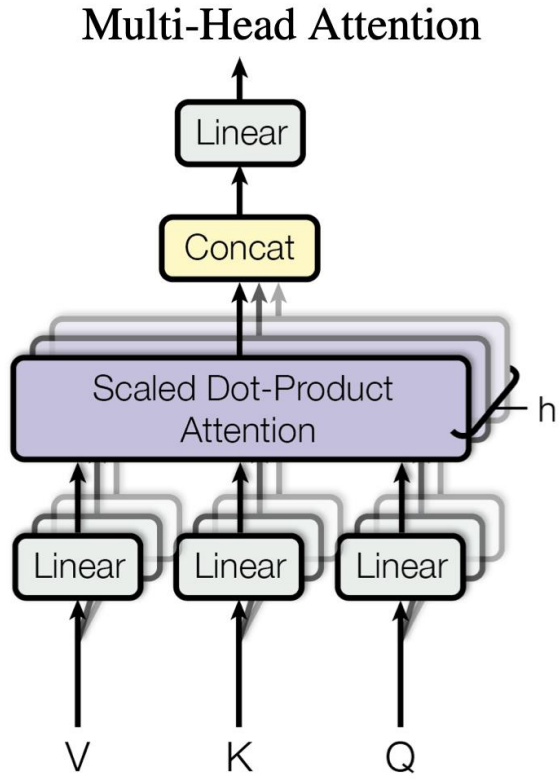
Multi-Head Attention

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

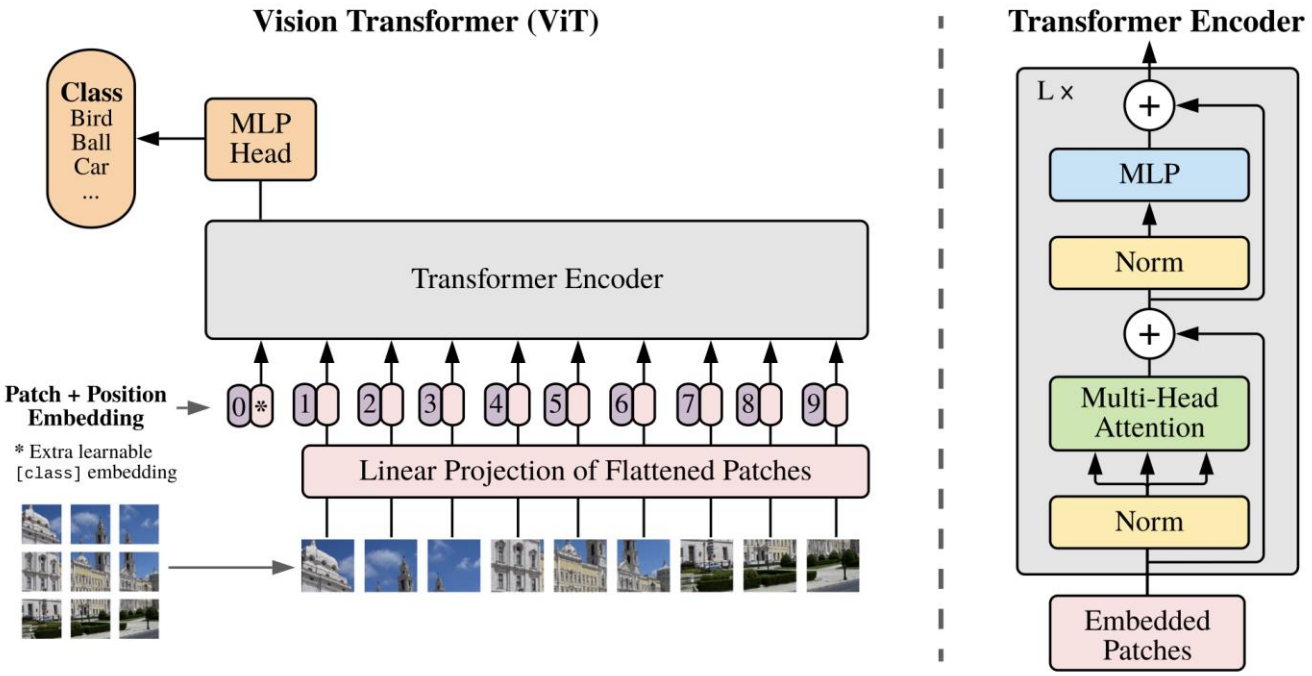
- parameters: 405B
- training tokens: 15.6T

factor less than 40
→ a lot of memorizing

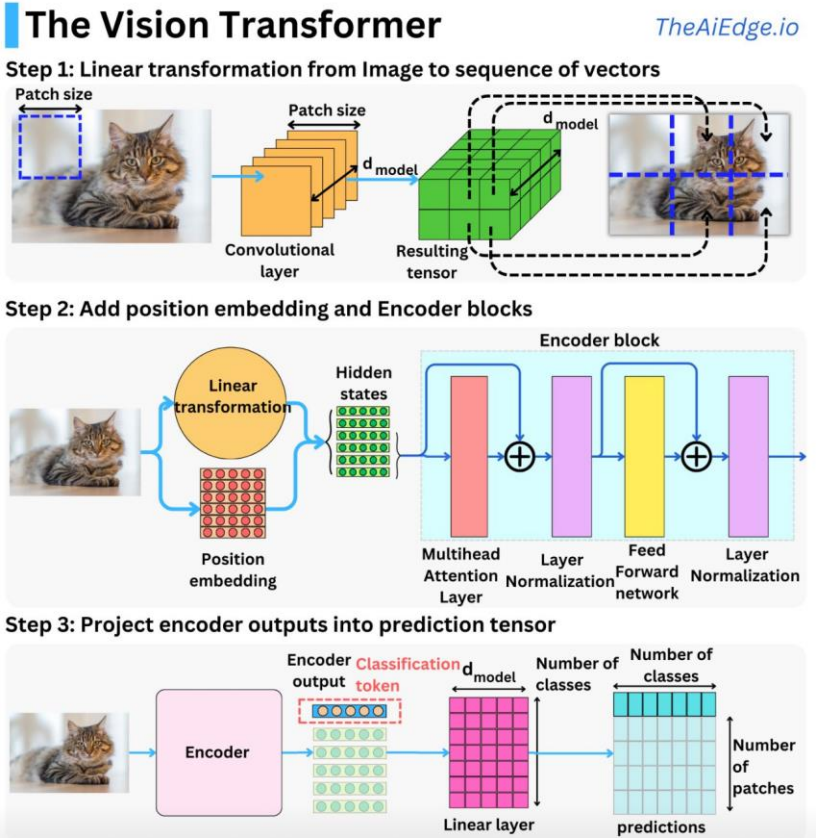
- context length/window (tokens): 128K

- 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:

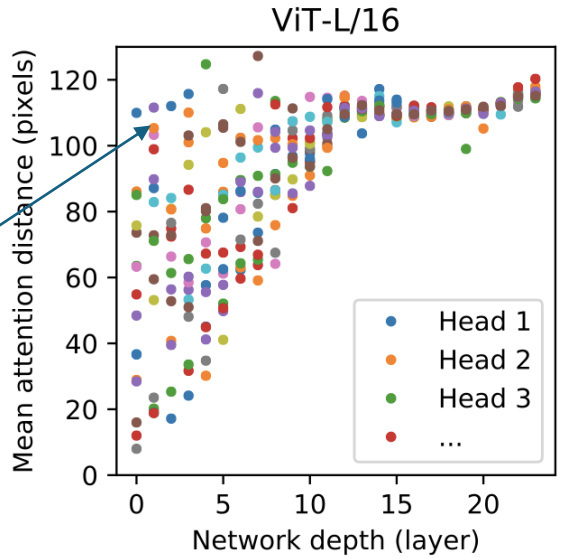
- no translation invariance
- no locally restricted receptive field

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

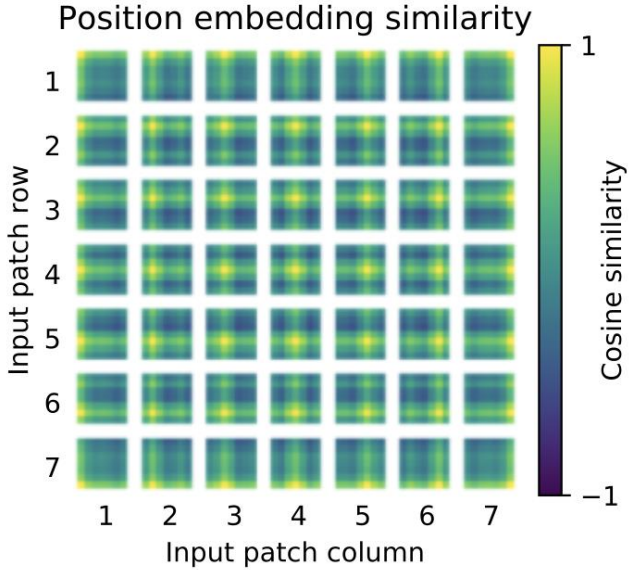
but can lead to beneficial effects (e.g., global attention in lower layers)

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:



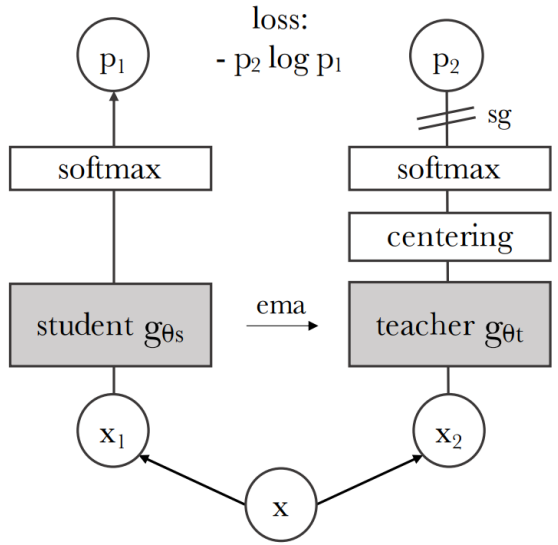
added due to permuation invariance of attention

source

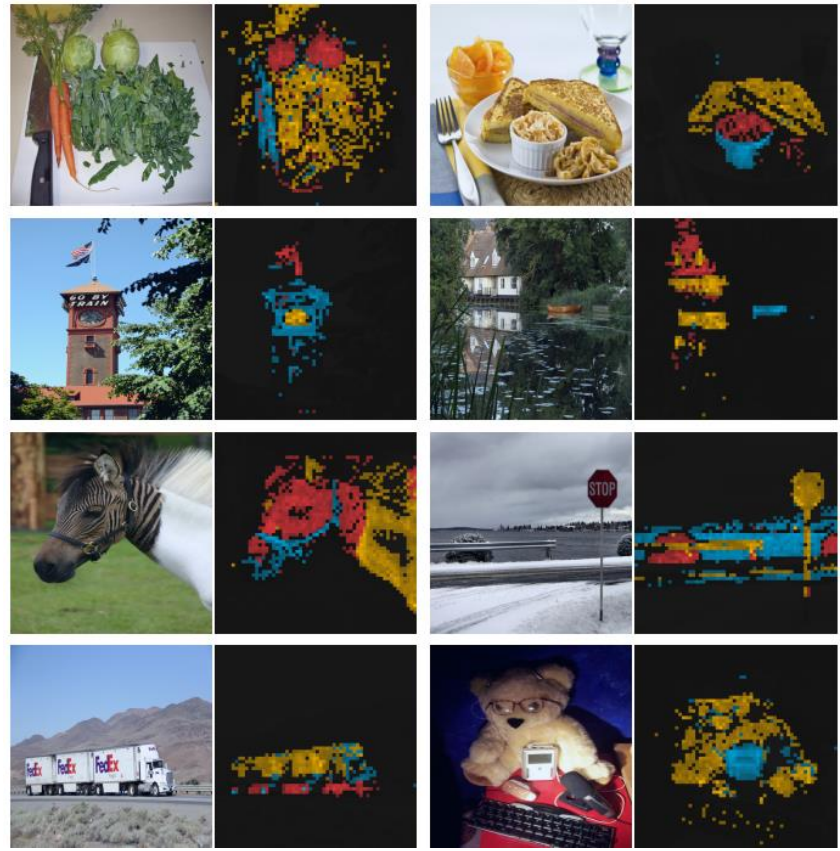
Self-Supervised Vision Transformers

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

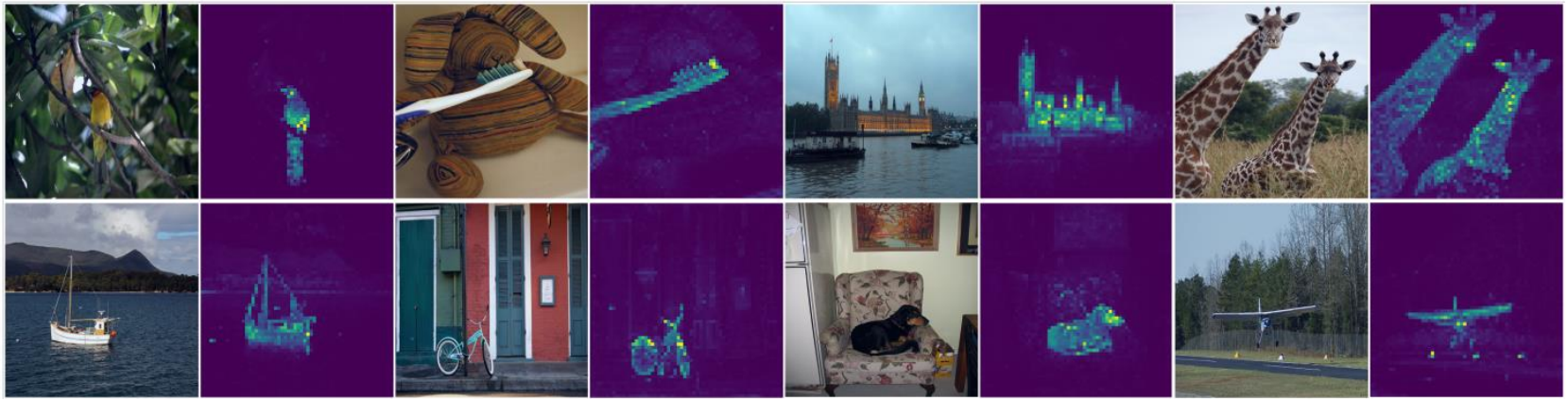
foundation model: extraction of self-supervised ViT features



different attention heads:



self-attention of last layer:

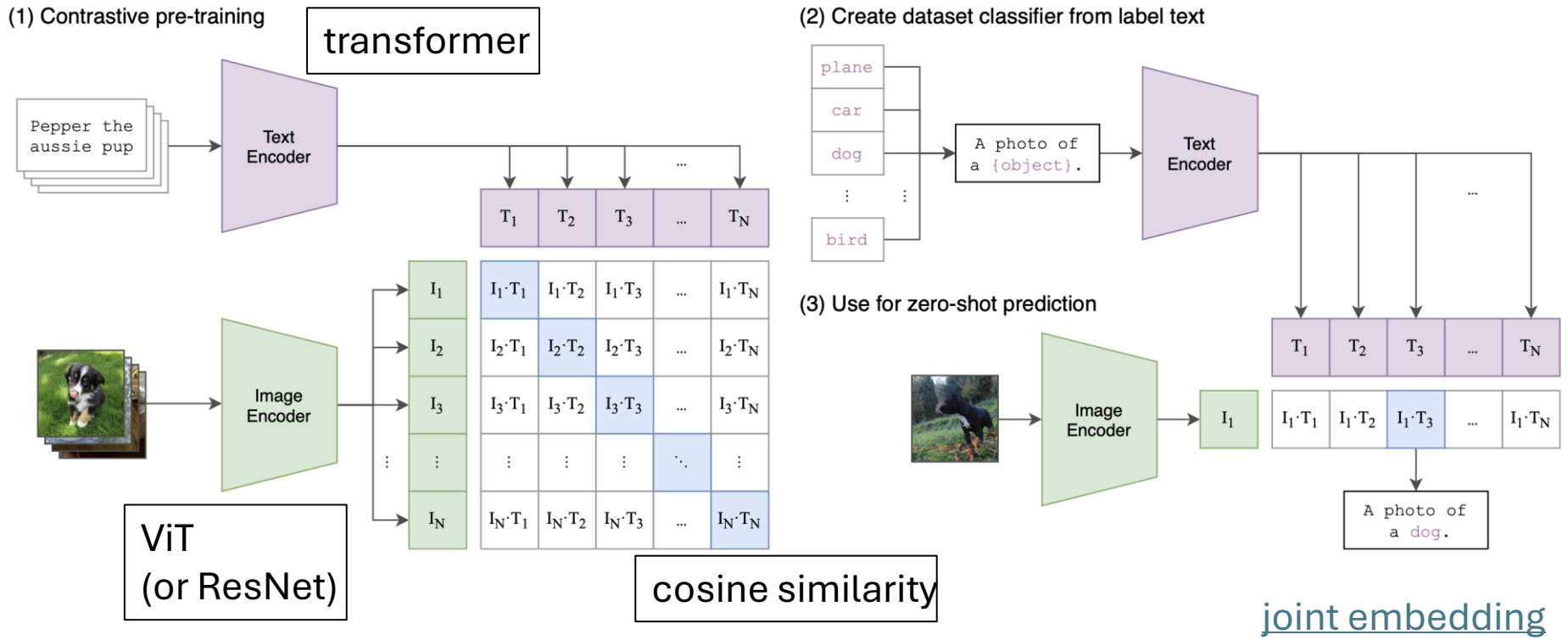


Combination of Vision and Text

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