Applied Machine Learning

Convolutional Neural Networks

Reihaneh Rabbany

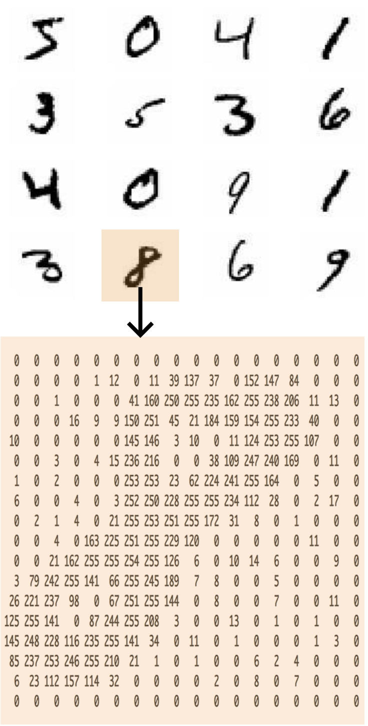
McGill School of Computer Science

Learning objectives

understand the convolution layer and the architecture of Conv-net

- its inductive bias
- its derivation from fully connected layer
- variations of convolution layer

MLP and image data



we can apply an MLP to image data

first vectorize the input $x \to \mathrm{vec}(x) \in \mathbb{R}^{784}$

feed it to the MLP (with L layers) and predict the labels

$$\mathrm{softmax} \circ W^{\{L\}} \circ \ldots \circ \mathrm{ReLU} \circ W^{\{1\}} \mathrm{vect}(x)$$

the model knows nothing about the image structure

we could shuffle all pixels and learn an MLP with similar performance

how to bias the model, so that it "knows" its input is image?

image is like 2D version of sequence data

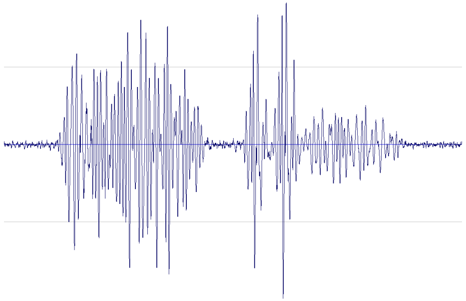
let's find the right model for sequence first!

image:https://medium.com/@rajatjain0807/machine-learning-6ecde3bfd2f4

Parameter-sharing

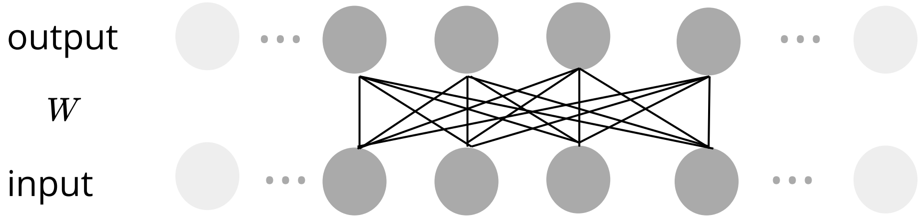
suppose we want to convert one sequence to another $\mathbb{R}^D \to \mathbb{R}^D$

example: *remove background noise from audio signal*



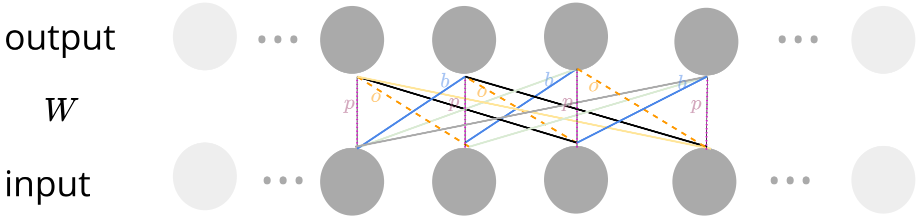
suppose we have a dataset of input-output pairs $\{(x^{(n)}, y^{(n)})\}_n$

consider only a single layer $y = g(Wx)$



we may assume, each output unit is the same function shifted along the sequence

when is this a good assumption?



elements of w of the same color are tied together (parameter-sharing)

Locality & sparse weight

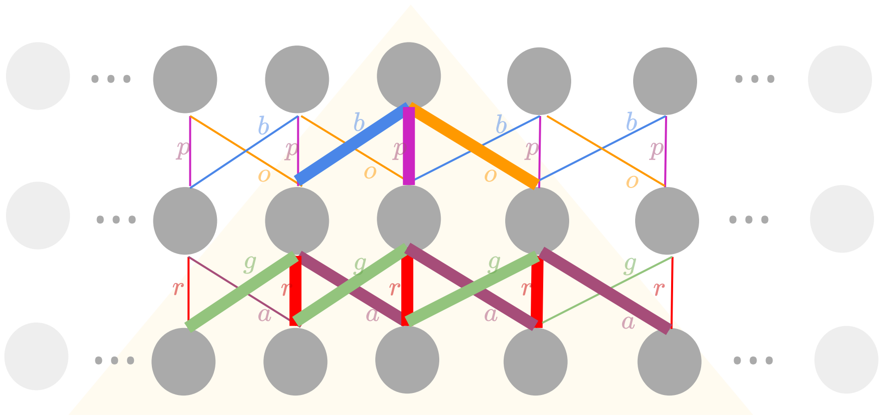
we may further assume each output is a **local** function of input

larger **receptive field** with multiple layers

one layer: the output units "see" 3 neighbouring inputs

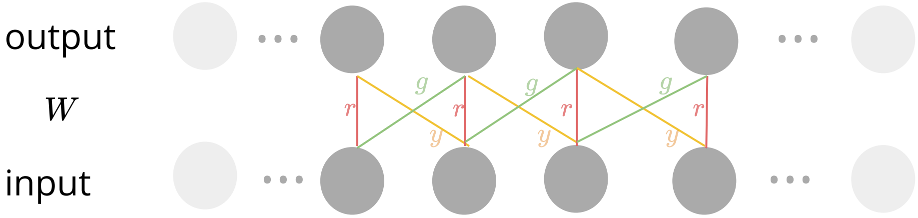
two layers: the output units "see" 5 neighbouring inputs

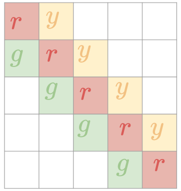
...



Cross-correlation (1D)

Let's look at the parameter matrix W

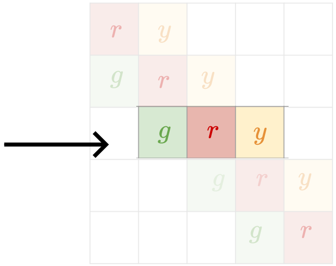




parameter-sharing in W
W is very sparse

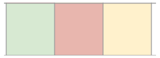
instead of the whole matrix we can keep the one set of nonzero values

$$w = [w_1, \ldots, w_K] = [W_{c,c-\lfloor \frac{K}{2} \rfloor}, \ldots, W_{c,c+\lfloor \frac{K}{2} \rfloor}]$$



we can write matrix multiplication as **cross-correlation** of w and x

$$y_c = g\left(\sum_{d=1}^{D} W_{c,d} x_d\right) = g\left(\sum_{k=1}^{K} w_k x_{c-\lfloor \frac{K}{2} \rfloor + k}\right)$$

feedforward layer: slide  on the input, calculate inner product and apply the nonlinearity

Convolution (1D)

Cross-correlation is similar to convolution

Cross-correlation $y_d = \sum_{k=-\infty}^{\infty} w_k x_{d+k}$ $w \star x$

w is called the filter or kernel

ignoring the activation (for simpler notation)

assuming w and x are zero for any index outside the input and filter bound

Convolution flips w or x (to be commutative)

$$y_d = \sum_{k=-\infty}^{\infty} w_k x_{d-k} = \sum_{k'=-\infty}^{\infty} w_{d-k'} x_{k'}$$

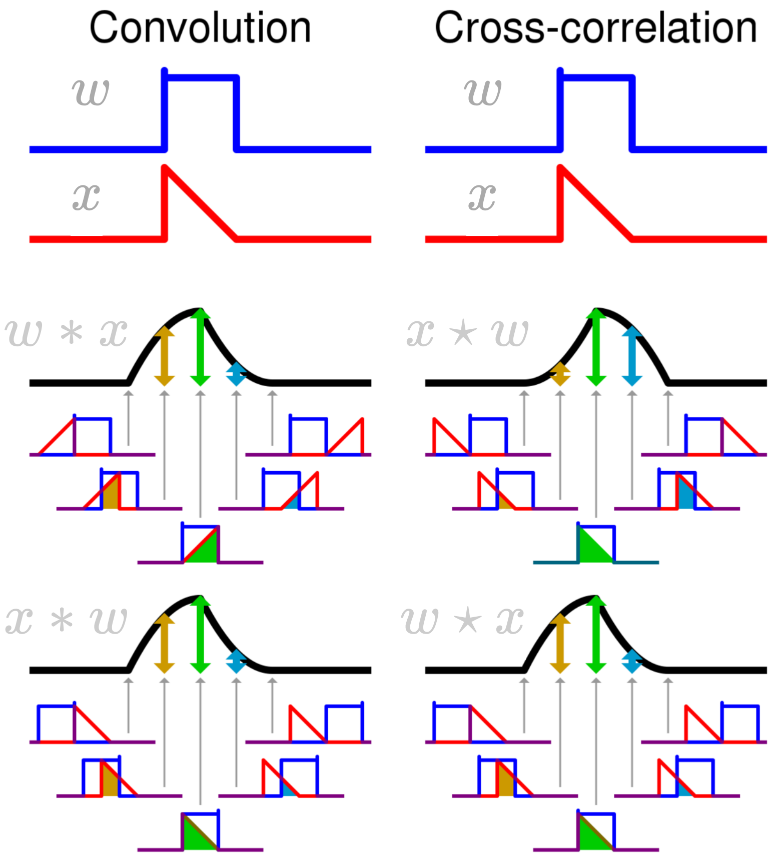
$w * x$ *change of variable* $k' = d - k$ $x * w$

since we **learn** w, flipping it makes no difference

in practice, we use cross correlation rather than convolution

convolution is equivariant wrt translation

-- *i.e.*, shifting **x**, shifts **w*x**



Convolution (1D)

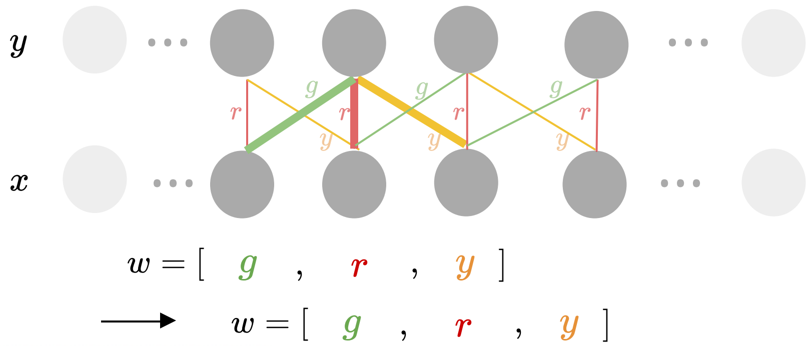
1D convolution layer **so far…**

$$y_d = \sum_{k=1}^{K} w_k x_{d+k-1}$$

-1 is because the indexing starts from 1
for d=1,k=1 we index the first element of x

```
def Conv1D(
    x, # D (length)
    w, # K (filter length)
    ):

    D, = x.shape
    K, = w.shape
    Dp = D - K + 1 #output length
    y = np.zeros(Dp)
    for dp in range(Dp):
        y[dp] = np.sum(x[dp:dp+K] * w)
    return y
```



Example:

Input: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | * Kernel: | 1 | 2 | = Output: | 2 | 5 | 8 | 11 | 14 | 17 |

Convolution (2D)

similar idea of parameter-sharing and locality extends to 2 dimension *(i.e. image data)*

$$y_{d_1,d_2} = \sum_{k_1=1}^{K_1} \sum_{k_2=1}^{K_2} x_{d_1+k_1-1,d_2+k_2-1} w_{k_1,k_2}$$

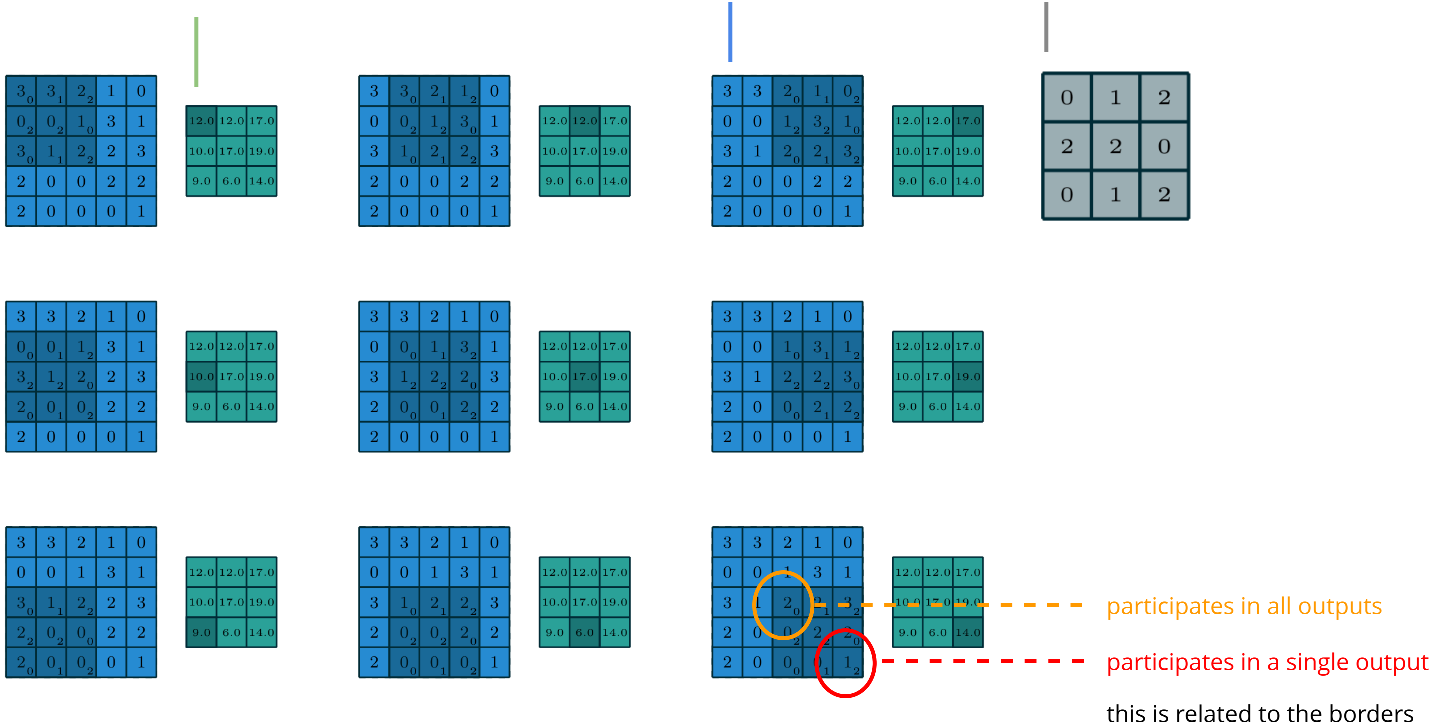


image credit: Vincent Dumoulin, Francesco Visin

Convolution (2D)

there are different ways of handling the borders

output length (for one dimension)

$$D + 2 \times \text{padding} - K + 1$$

zero-pad the input, and produce all non-zero outputs (full)
the output is larger than the input

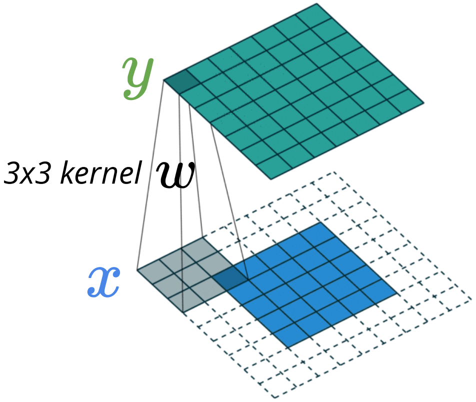
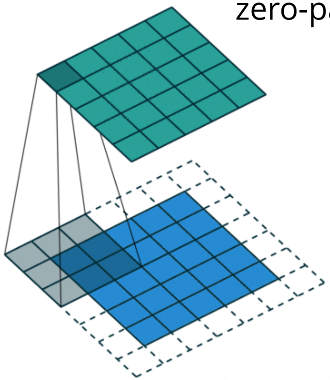
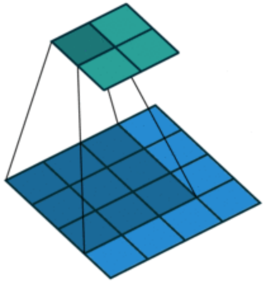


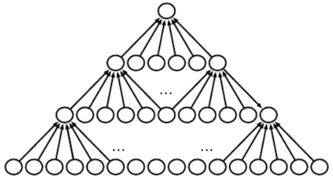
image credit: Vincent Dumoulin, Francesco Visin

zero-pad the input, to keep the output dims similar to input (same)



no padding at all (valid)
the output is smaller than the input
we can't stack many layers
sometimes we need to maintain the width





Pooling

sometimes we would like to reduce the size of output *e.g., from D x D to D/2 x D/2*

a combination of pooling and downsampling is used

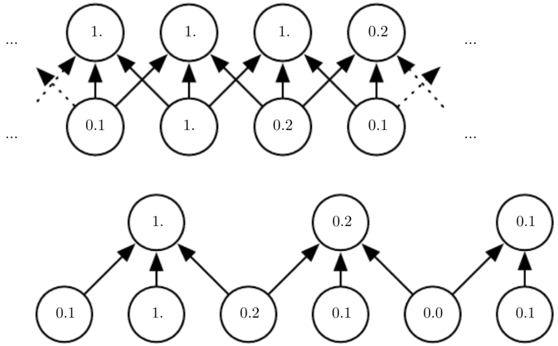
1. calculate the output $\tilde{y}_d = g\left(\sum_{k=1}^{K} x_{d+k-1} w_k\right)$

2. aggregate the output over different regions

$$y_d = \text{pool}\{\tilde{y}_d, \ldots, \tilde{y}_{d+p}\}$$

two common aggregation functions are ***max*** *and* ***mean***

3. often this is followed by subsampling using the same step size



the same idea extends to higher dimensions

12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

2×2 Max-Pool →

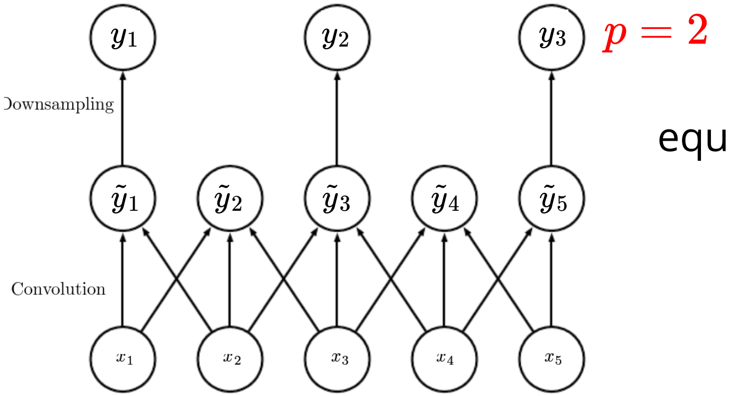
20	30
112	37

Strided convolution

alternatively we can directly subsample the output

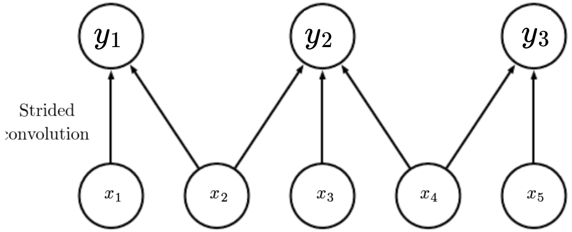
$$\tilde{y}_d = g\left(\sum_{k=1}^{K} x_{(d-1)+k} w_k\right)$$

$$y_d = \tilde{y}_{p(d-1)+1}$$



equivalent to →

$$\tilde{y}_d = g\left(\sum_{k=1}^{K} x_{p(d-1)+k} w_k\right)$$

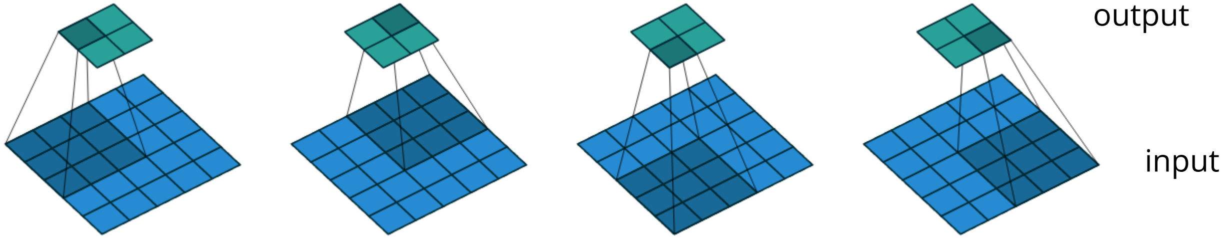


Strided convolution

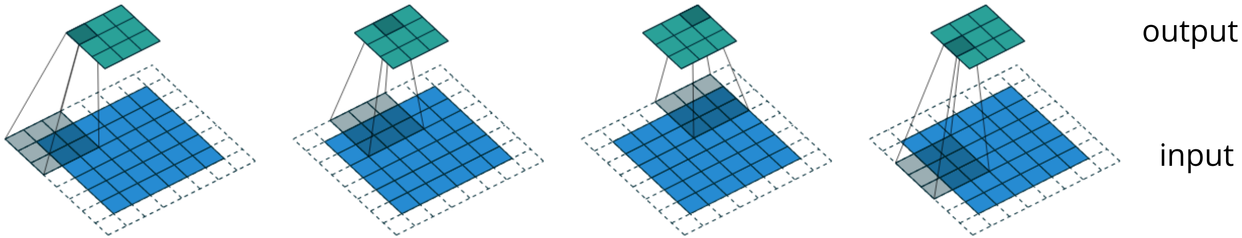
the same idea extends to higher dimensions

$$y_{d_1,d_2} = \sum_{k_1=1}^{K_1} \sum_{k_2=1}^{K_2} x_{p_1(d_1-1)+k_1, p_2(d_2-1)+k_2} w_{k_1,k_2}$$

different strides for different dimensions



with padding



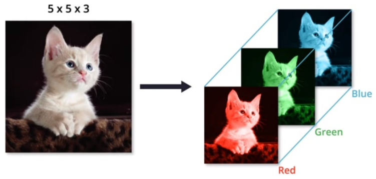
output length (for one dimension)

$$\left\lfloor \frac{D + 2\times\text{padding} - K}{\text{stride}} + 1 \right\rfloor$$

image: Dumoulin & Visin'16

Channels

so far we assumed a single input and output sequence or image

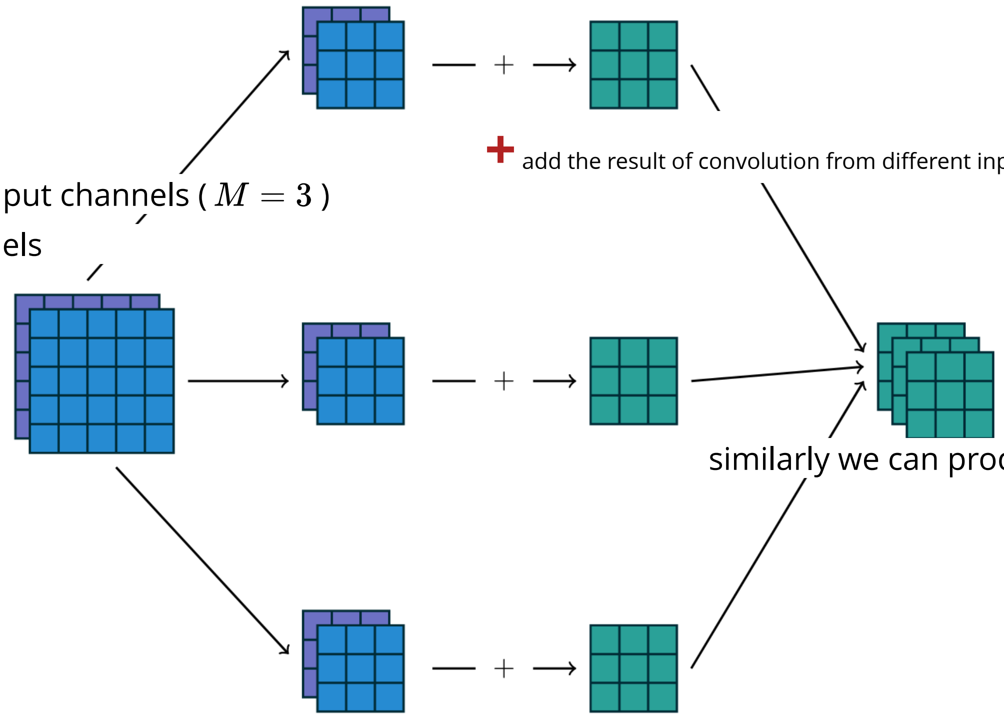


with RGB data, we have 3 input channels ($M = 3$)

this example: 2 input channels

$x \in \mathbb{R}^{M \times D_1 \times D_2}$

we have one $K_1 \times K_2$ filters per input-output channel combination $w \in \mathbb{R}^{M \times M' \times K_1 \times K_2}$



+ add the result of convolution from different input channels

similarly we can produce multiple output channels $M' = 3$

$y \in \mathbb{R}^{M' \times D_1' \times D_2'}$

image: Dumoulin & Visin'16

Channels

we can also add a *bias parameter (b),* one per each output channel

$$b \in \mathbb{R}^{M'}$$

$$y_{m',d_1,d_2} = g\left(\sum_{m=1}^{M} \sum_{k_1} \sum_{k_2} w_{m,m',k_1,k_2}\, x_{m,d_1+k_1-1,d_2+k_2-1} + b_{m'} \right)$$

$$y \in \mathbb{R}^{M' \times D_1' \times D_2'}$$

$$x \in \mathbb{R}^{M \times D_1 \times D_2}$$

$$w \in \mathbb{R}^{M \times M' \times K_1 \times K_2}$$

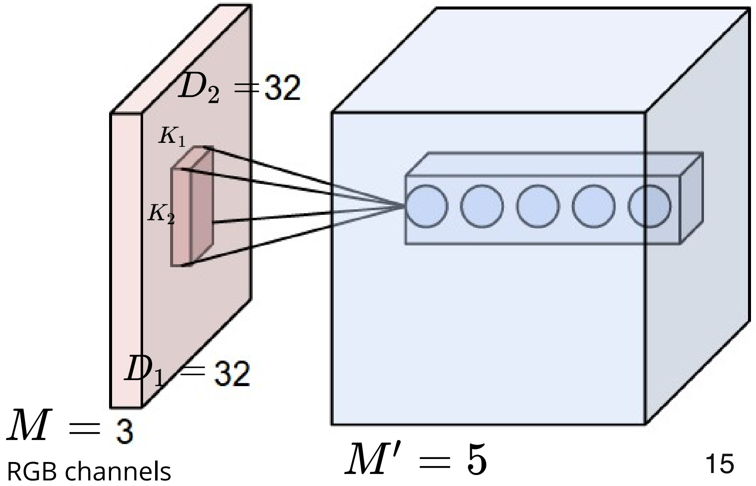


image: https://cs231n.github.io/convolutional-networks/

Example

https://cs231n.github.io/assets/conv-demo/

⋮

demo from: https://cs231n.github.io/convolutional-networks/

Convolutional Neural Network (CNN)

CNN or convnet is a neural network with convolutional layers

it could be applied to 1D sequence, 2D image or 3D volumetric data

example: conv-net architecture (LeNet, 1998) for digits recognition

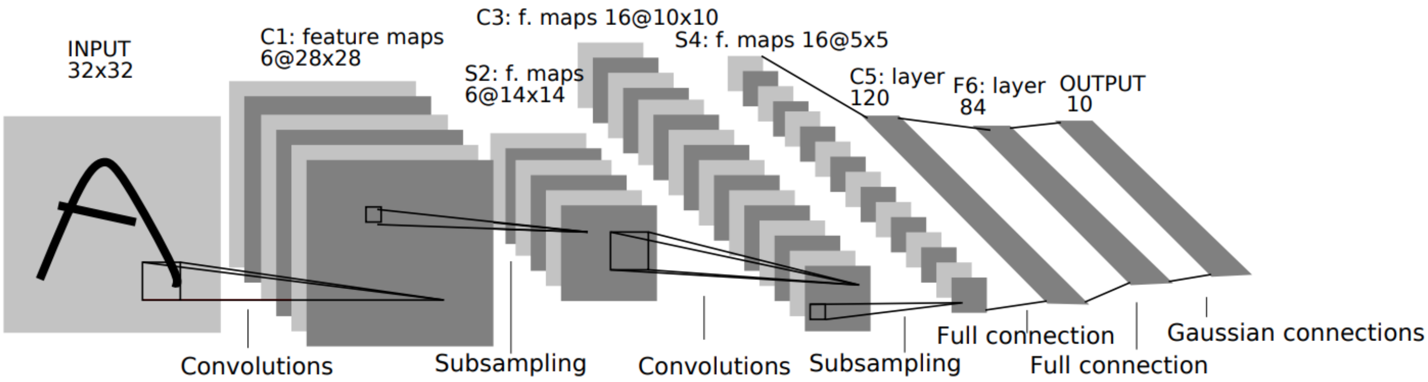
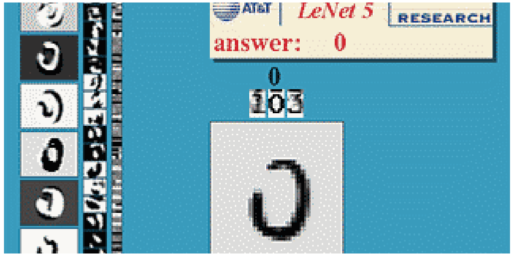


image from LeNet paper



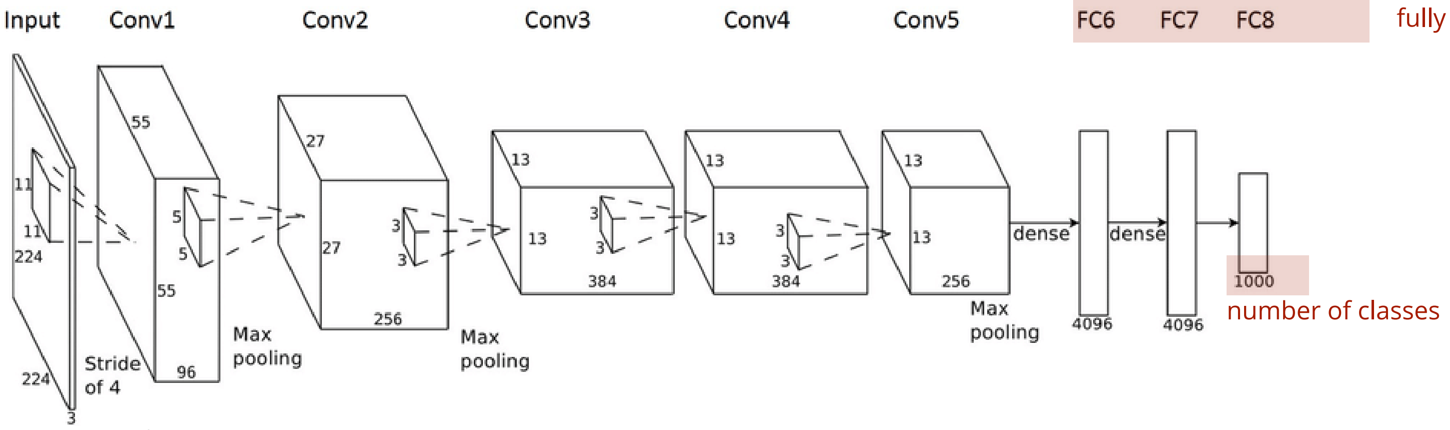
very accurate to be used in large scale in postal services (zip code recognition) and banks (cheques)

Convolutional Neural Network (CNN)

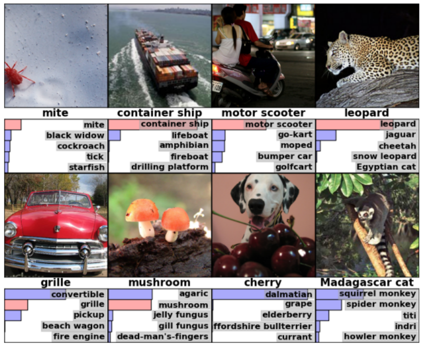
CNN or convnet is a neural network with convolutional layers

it could be applied to 1D sequence, 2D image or 3D volumetric data

example: conv-net architecture (derived from AlexNet) for image classification



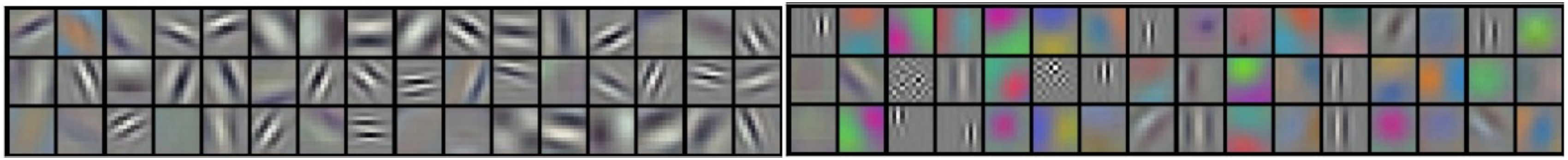
fully connected layers



read the paper here

visualization of the convolution kernel at the first layer 11x11x3x96

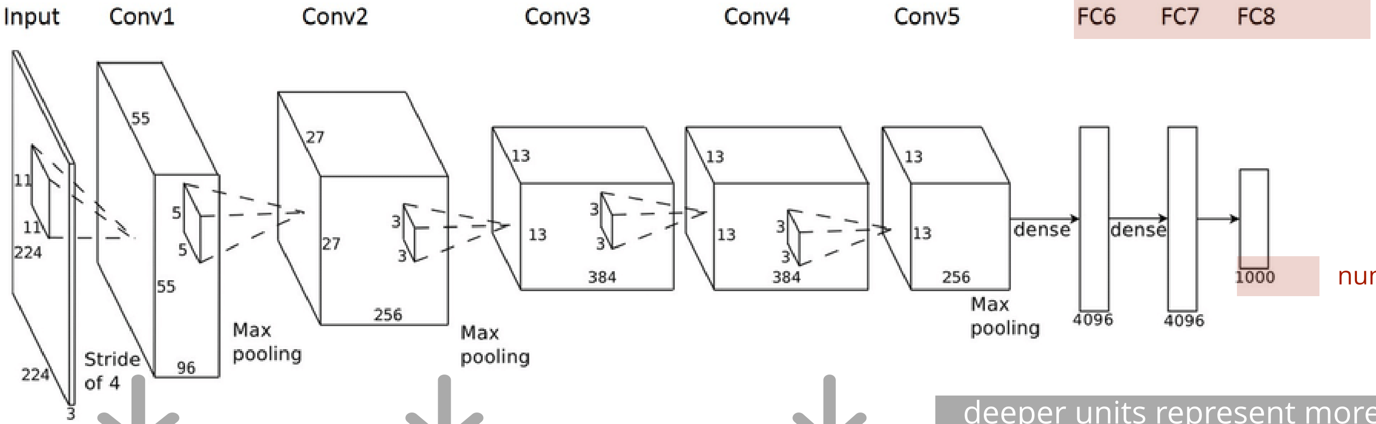
96 filters, each one is 11x11x3. each of these is responsible for one of 96 feature maps in the second layer



Convolutional Neural Network (CNN)

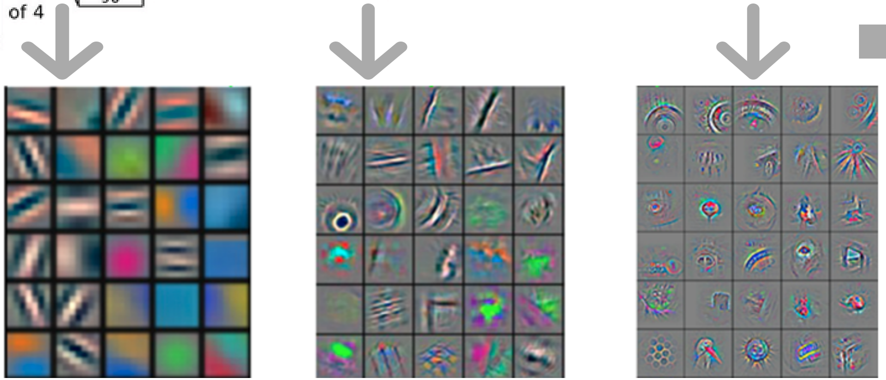
CNN or convnet is a neural network with convolutional layers (so it's a special type of MLP)
it could be applied to 1D sequence, 2D image or 3D volumetric data

example: conv-net architecture (derived from AlexNet) for image classification



fully connected layers

number of classes



deeper units represent more complex/abstract features

check this for more on feature visualization

optional

Application: image classification

Convnets have achieved super-human performance in image classification

ImageNet challenge: > 1M images, 1000 classes

GT: horse cart
1: horse cart
2: minibus
3: oxcart
4: stretcher
5: half track

GT: birdhouse
1: birdhouse
2: sliding door
3: window screen
4: mailbox
5: pot

GT: forklift
1: forklift
2: garbage truck
3: tow truck
4: trailer truck
5: go-kart

GT: coucal
1: coucal
2: indigo bunting
3: lorikeet
4: walking stick
5: custard apple

GT: komondor
1: komondor
2: patio
3: llama
4: mobile home
5: Old English sheepdog

GT: yellow lady's slipper
1: yellow lady's slipper
2: slug
3: hen-of-the-woods
4: stinkhorn
5: coral fungus

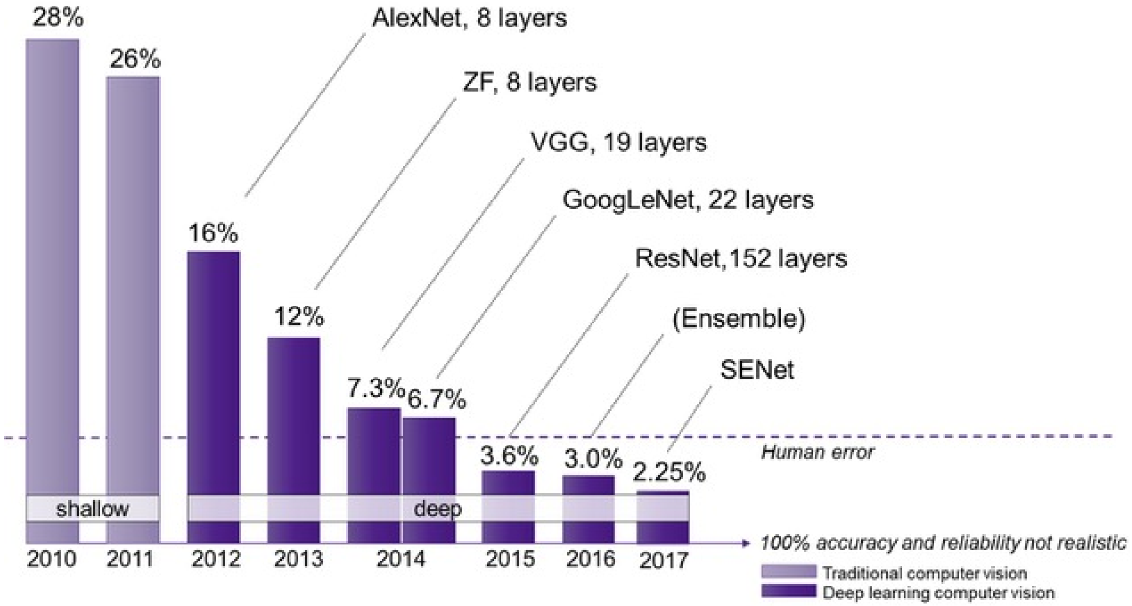


image credit: He et al'15, https://semiengineering.com/new-vision-technologies-for-real-world-applications/

Application: image classification

variety of increasingly deeper architectures have been proposed

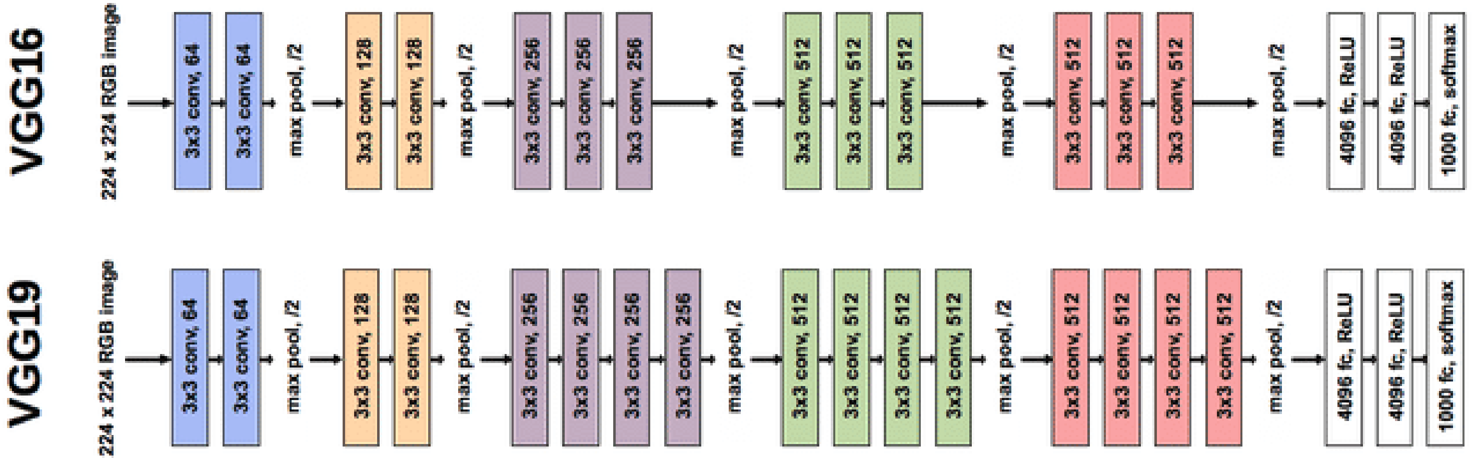


image credit: He et al'15, https://semiengineering.com/new-vision-technologies-for-real-world-applications/

optional

Application: image classification

variety of increasingly deeper architectures have been proposed

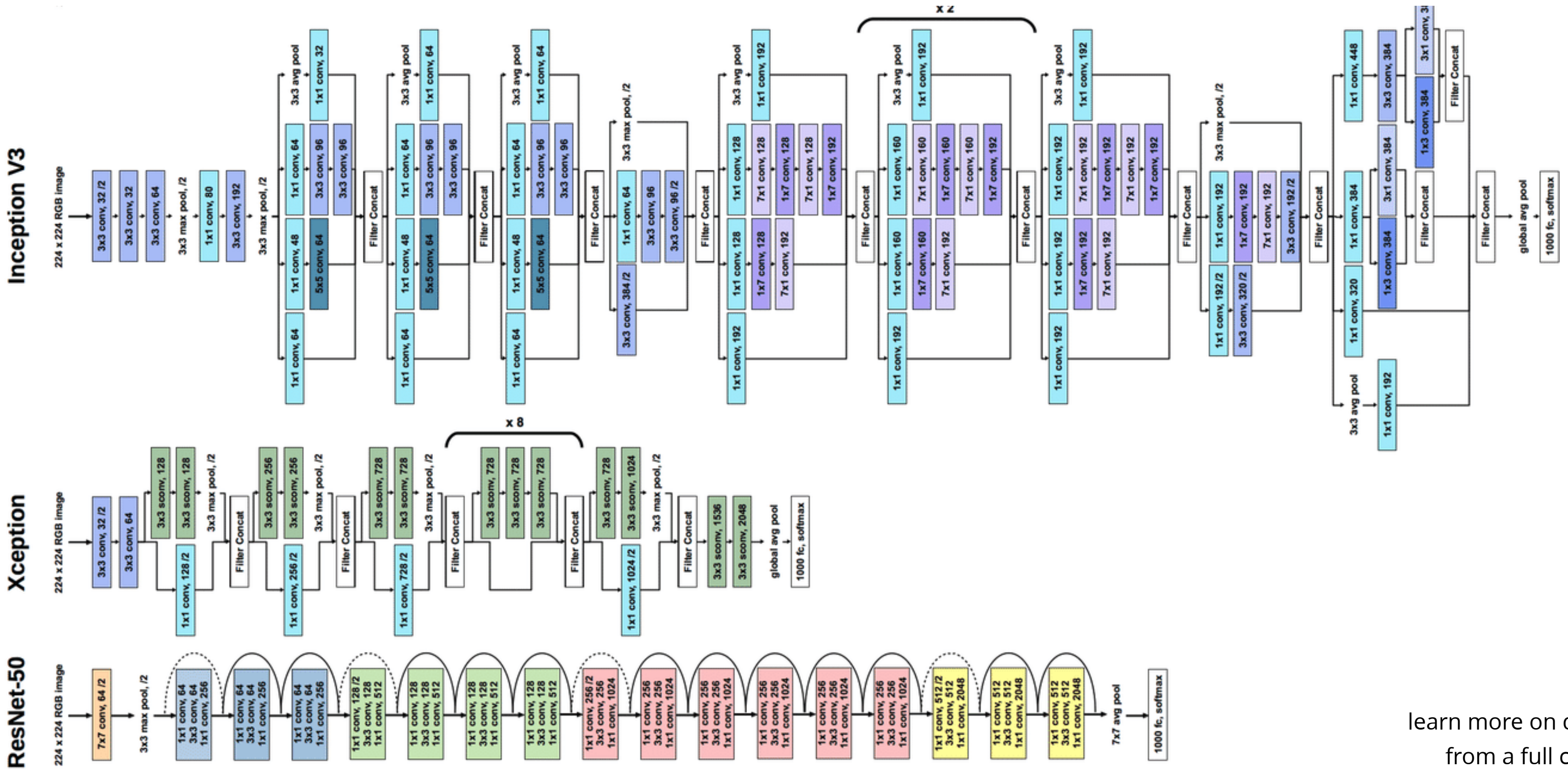
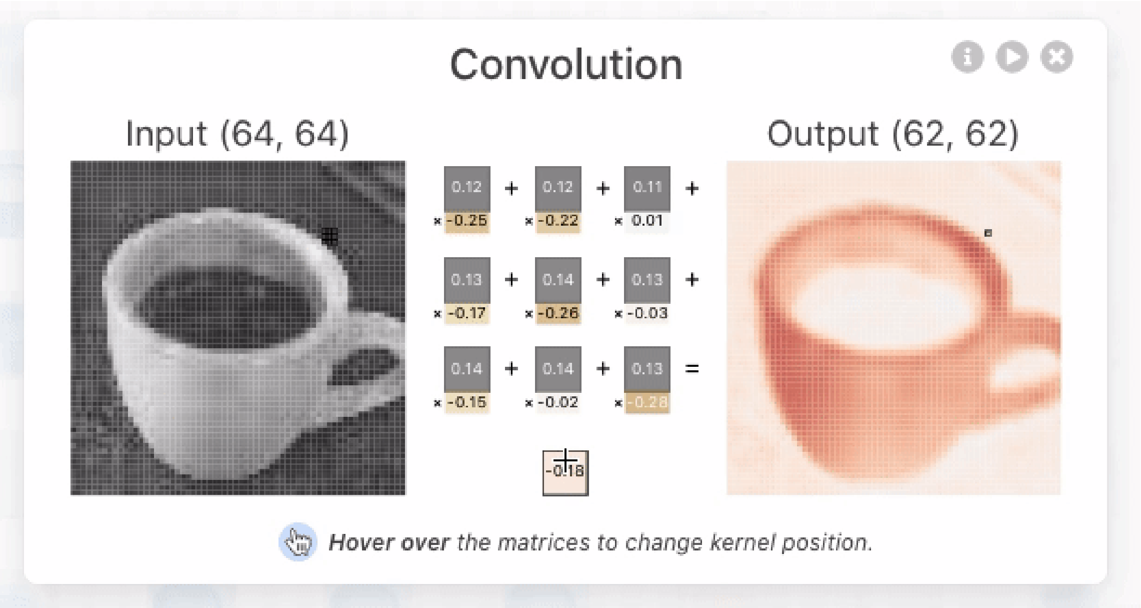


image credit: He et al'15, https://semiengineering.com/new-vision-technologies-for-real-world-applications/

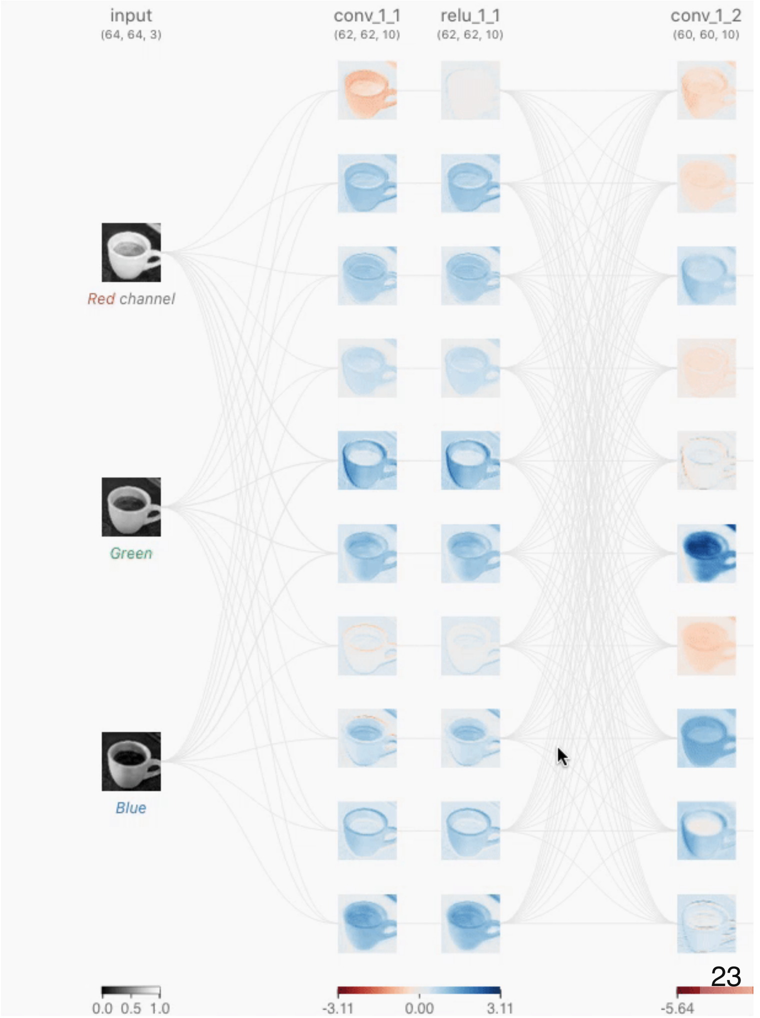
learn more on different CNN here

from a full course on CNN: http://cs231n.stanford.edu/

Visual Examples



see the interactive demo here



Training: backpropagation through convolution

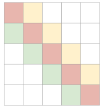
optional

consider the 1D convolution op. $y_d = \sum_k w_k x_{d+k-1}$

using backprop. we have $\frac{\partial J}{\partial y_d}$ so far and we need

1) $\frac{\partial J}{\partial w_k} = \sum_{d'} \frac{\partial J}{\partial y_{d'}} \frac{\partial y_{d'}}{\partial w_k}$ $\left(\frac{\partial y_{d'}}{\partial w_k} = x_{d'+k-1}\right)$ using this we can update the convolution kernel at the current layer

2) to backpropagate to previous layer $\frac{\partial J}{\partial x_d} = \sum_{d'} \frac{\partial J}{\partial y_{d'}} \frac{\partial y_{d'}}{\partial x_d}$ $\left(\frac{\partial y_{d'}}{\partial x_d} = w_{d-d'+1}\right)$

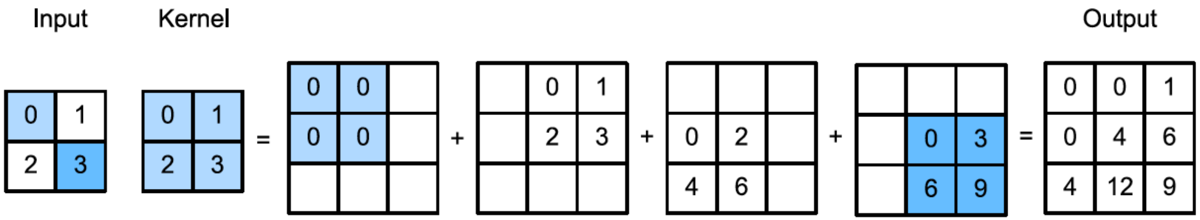


even when we have stride, and padding, this operation is similar to multiplication by transpose of the parameter-sharing matrix **(transposed convolution)**

Transposed convolution

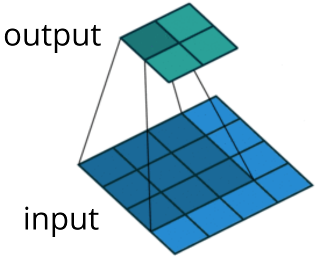
Transposed convolution produces a larger output from a smaller input

Example:



Transposed convolution can recover the shape of the original input

no padding of the original convolution corresponds to *full* padding of in transposed version



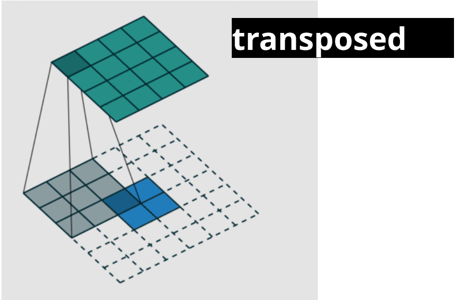


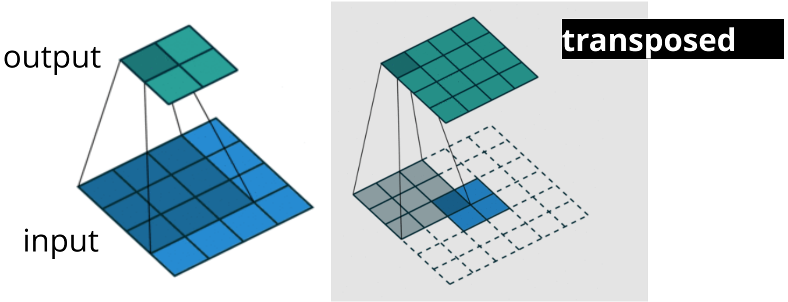
image: Dumoulin & Visin'16

optional

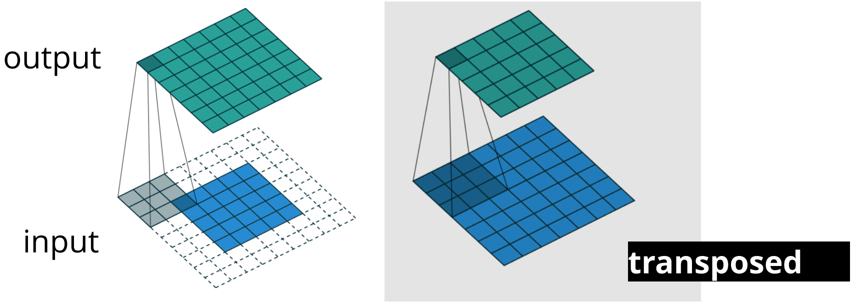
Transposed convolution

Transposed convolution recovers the shape of the original input

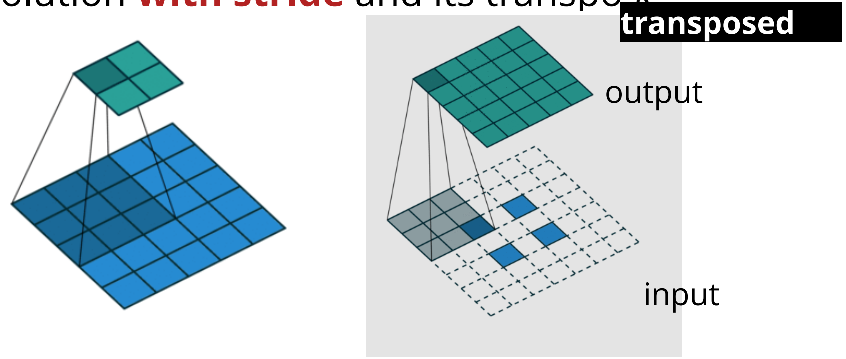
no padding of the original convolution corresponds to *full* padding of in transposed version



full padding of the original convolution corresponds to no padding of in transposed version



Convolution **with stride** and its transpose



this can be used for up-sampling (opposite of stride/pooling)

as expected the transpose of a transposed convolution is the original convolution

image: Dumoulin & Visin'16

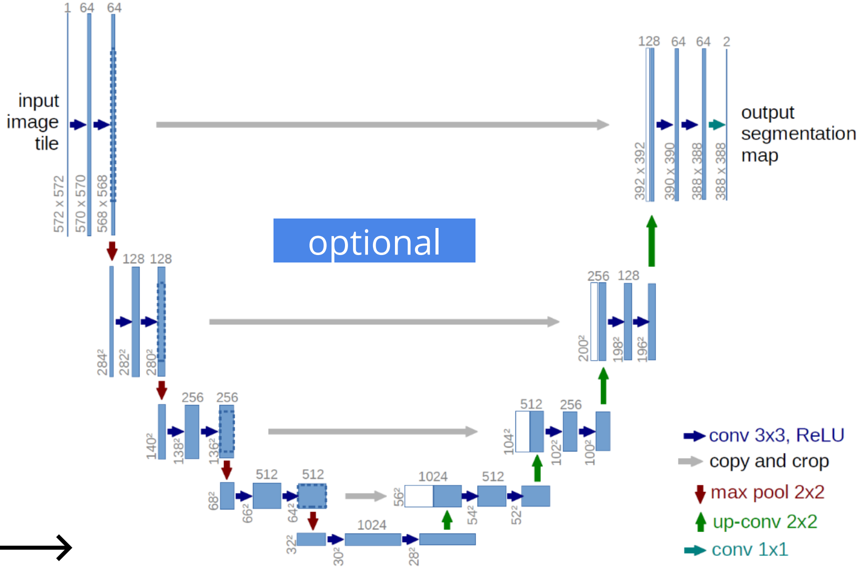
optional

Solving other discriminative vision tasks with CNNs

Structured Prediction: the output itself may have (image) structure (e.g., predicting text, audio, image)

example

in (semantic) segmentation, we classify each pixel
loss is the sum of cross-entropy loss across the whole image



variety of architectures... one that performs well is **U-Net** ⟶

transposed convolution (upconv), concatenation, and skip connection are common in architecture design
architecture search (i.e., combinatorial hyper-parameter search) is an expensive process and an active research area

image:https://sthalles.github.io/deep_segmentation_network/

Generating images by inverting CNNs

generating images which maximize the class label

$$p(x|y) \propto p(x)p(y|x)$$

CNN

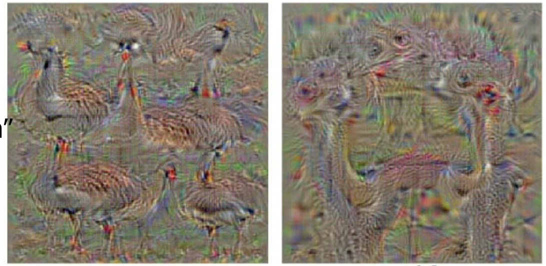
$$x_{t+1} = x_t + \epsilon_1 \frac{\partial \log p(x_t)}{\partial x_t} + \epsilon_2 \frac{\partial \log p(y=c|x_t)}{\partial x_t} + \mathcal{N}(0, \epsilon_3^2 \boldsymbol{I})$$

e.g. using Gaussian prior we have:

$$x_{t+1} = (1 - \epsilon_1)x_t + \frac{\partial \log p(y=c|x_t)}{\partial x_t}$$

gradients

Images that maximize the probability of ImageNet classes “goose” and “ostrich”



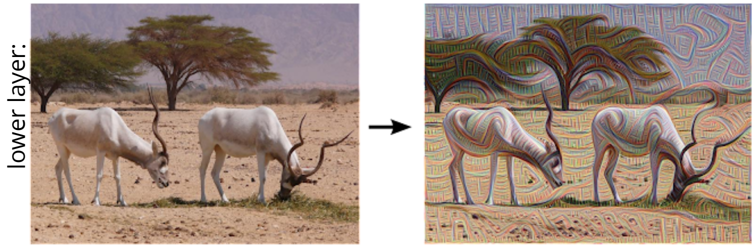
goose ostrich

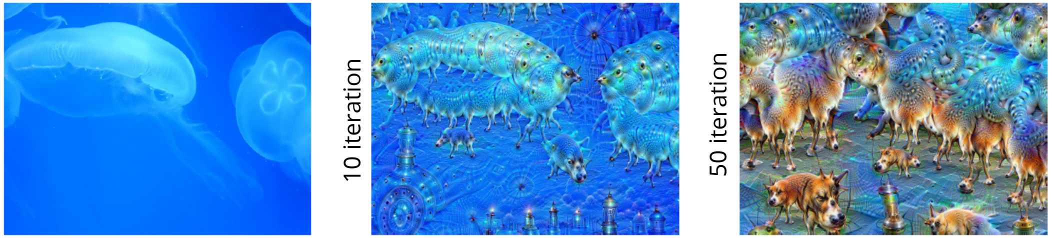
e.g. using *Total variation (TV)* prior gives more realistic images

Anemone Fish Banana Parachute Screw

Generating images by inverting CNNs

- Deep Dream
  - *generate versions of an input image that emphasize certain features by picking a layer and ask the network to enhance whatever it detected*





read more here

- Neural style transfer
  - *specify a reference "style image" xs and "content image" xc.*

Summary

convolution layer introduces an **inductive bias (**equivariance**)** to MLP

- translation of the same model is applied to produce different outputs (pixels)
- the layer is equivariant to **translation**
- achieved through **parameter-sharing**

conv-nets use combinations of

- convolution layers
- ReLU (or similar) activations
- pooling and/or stride for down-sampling
- skip-connection and/or batch-norm to help with optimization / regularization
- potentially fully connected layers in the end

training

- backpropagation (similar to MLP)
- SGD or its improved variations with adaptive learning rate
- monitor the validation error for early stopping