Convolutional Neural Networks

Introduction to Machine Learning (ICP363)

Paulo Mann · Instituto de Computação, UFRJ

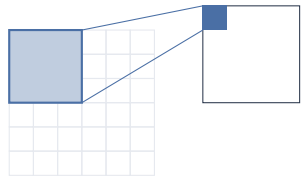
Prepared based on the teaching material of **Sebastian Raschka**[1] (STAT 453: Introduction to Deep Learning and Generative Models, Spring 2021)

[1] https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L13_intro-cnn__slides.pdf

About these slides

These slides were prepared based on the lecture material of **Sebastian Raschka**

STAT 453: Introduction to Deep Learning and Generative Models
University of Wisconsin–Madison, Spring 2021
Lecture 13: Introduction to Convolutional Neural Networks



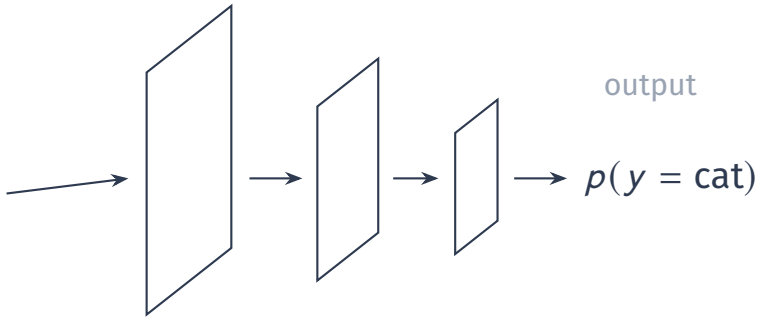
Order, concepts and examples follow the original lecture; the visual style was redesigned. All credit for the content goes to him.

Topics

1. **What CNNs can do**
2. Image classification
3. Convolutional neural network basics
4. Convolutional filters and weight sharing
5. CNNs and backpropagation
6. CNN architectures
7. What a CNN can see
8. CNNs in PyTorch

CNNs for image classification



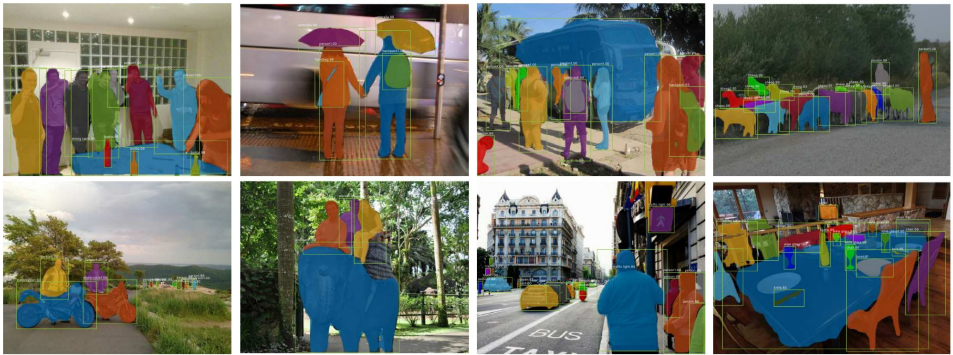


Object detection



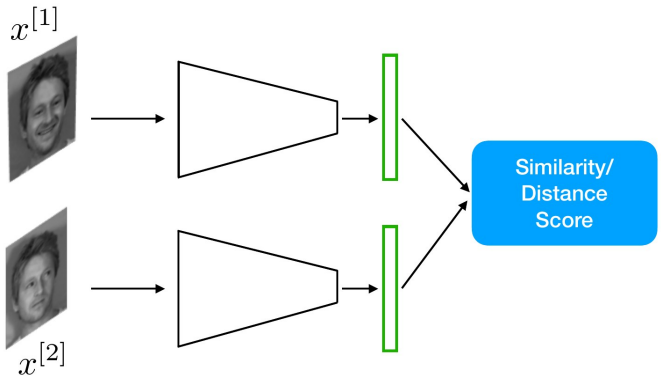
Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. CVPR, pp. 779–788.

Object segmentation



He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask R-CNN. ICCV, pp. 2961–2969.

Face recognition



Same network for both images: similar faces ⇒ small distance

Topics

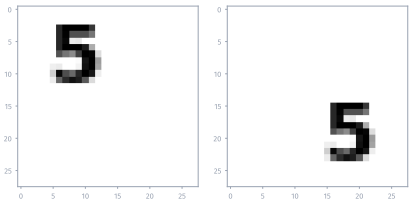
1. What CNNs can do
2. **Image classification**
3. Convolutional neural network basics
4. Convolutional filters and weight sharing
5. CNNs and backpropagation
6. CNN architectures
7. What a CNN can see
8. CNNs in PyTorch

Why image classification is hard

Different lighting, contrast, viewpoints, etc.



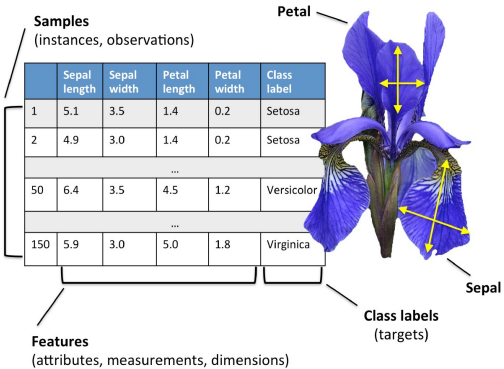
Or even simple translation



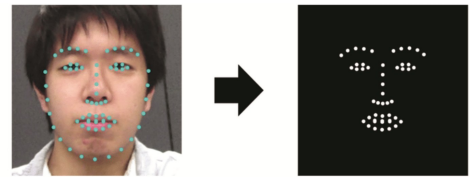
Hard for multi-layer perceptrons: the prediction is basically a weighted sum of pixel intensities, $\mathbf{W}^\top \mathbf{x} + \mathbf{b}$

Traditional approaches

a) Use hand-engineered features



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica



(a) Detected facial keypoints (b) Facial organ keypoints

Sasaki, K., Hashimoto, M., & Nagata, N. (2016). Person invariant classification of subtle facial expressions using coded movement direction of keypoints. Springer.

Traditional approaches

b) Preprocess images (centering, cropping, etc.)



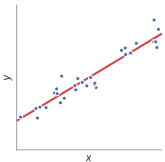


Topics

1. What CNNs can do
2. Image classification
3. **Convolutional neural network basics**
4. Convolutional filters and weight sharing
5. CNNs and backpropagation
6. CNN architectures
7. What a CNN can see
8. CNNs in PyTorch

What is an inductive bias?

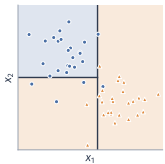
Inductive bias: the set of built-in assumptions or preferences that a learning algorithm uses to predict outputs for unseen data



Linear regression

straight-line relationship

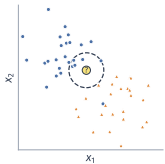
bias: linearity



Decision tree

rectangular blocks (step functions)

bias: hierarchical axis-aligned partitioning



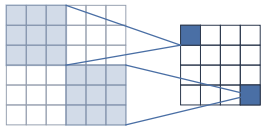
***k*-nearest neighbors**

nearby points share labels

bias: local smoothness / proximity

Inductive biases of neural networks

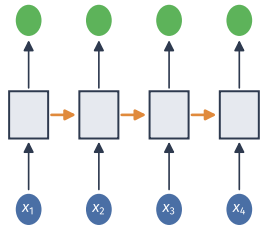
same weights **W** at every position



Convolutional network

nearby pixels matter together; the same pattern can appear anywhere

bias: locality + weight sharing

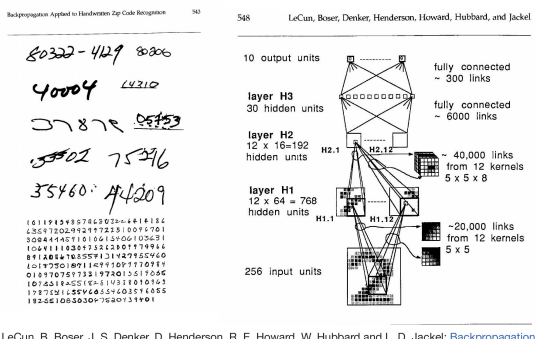


Recurrent network

inputs arrive in order; the same update at every time step

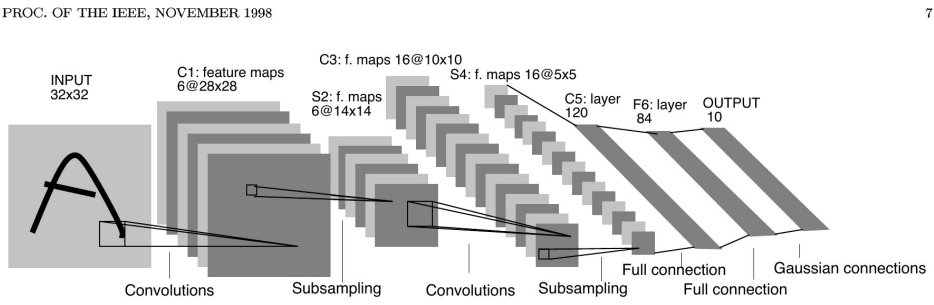
bias: sequentiality

Convolutional neural networks



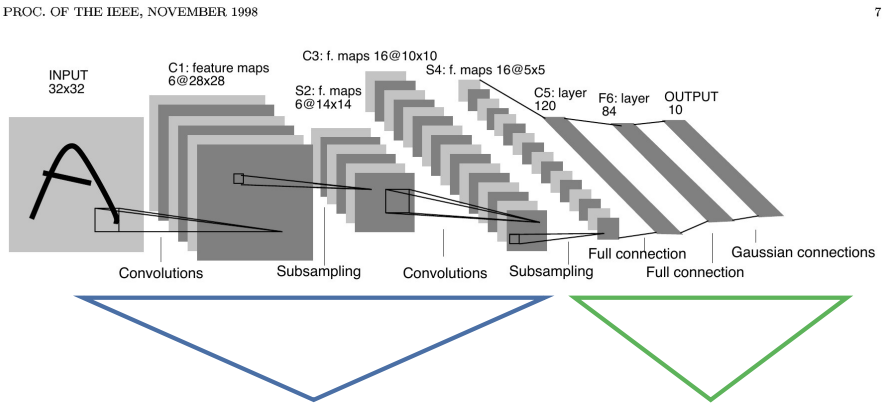
LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. Neural Computation, 1(4), 541–551.

Convolutional neural networks: LeNet-5

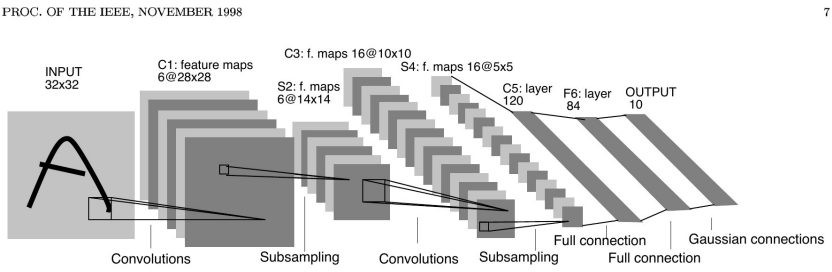


LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278–2324.

Hidden layers



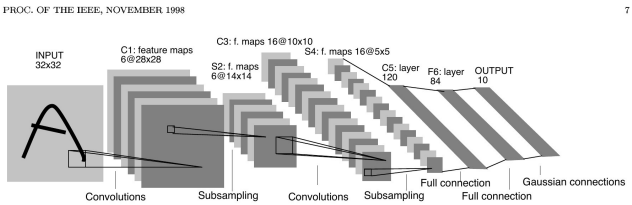
Hidden layers



Each “bunch” of feature maps is one hidden layer

Counting the fully connected layers, this network has 5 layers

Convolutional neural networks: reading LeNet-5



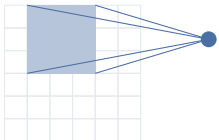
6@28×28: number of feature detectors @ size of the resulting maps

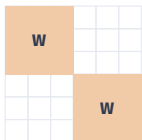
"Feature detectors" (weight matrices) reused across the image ("weight sharing"), also called **kernel** or **filter**

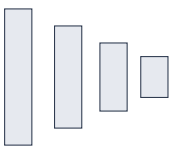
"Subsampling": nowadays called **pooling**

Last layers: a multi-layer perceptron (here with MSE loss; nowadays fully connected layers + softmax + cross-entropy)

Main concepts behind CNNs







Sparse connectivity: each element of the feature map is connected to a small patch of pixels only (an MLP connects every unit to the whole image)

Parameter sharing: the same weights are used for different patches of the image

Many layers: local patterns are combined into global patterns

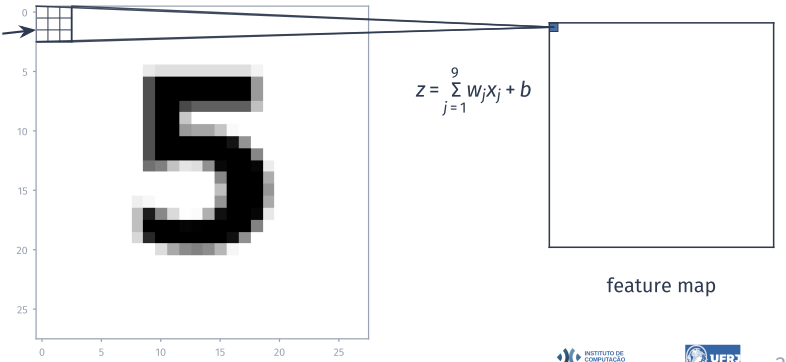
Topics

1. What CNNs can do
2. Image classification
3. Convolutional neural network basics
4. **Convolutional filters and weight sharing**
5. CNNs and backpropagation
6. CNN architectures
7. What a CNN can see
8. CNNs in PyTorch

Weight sharing

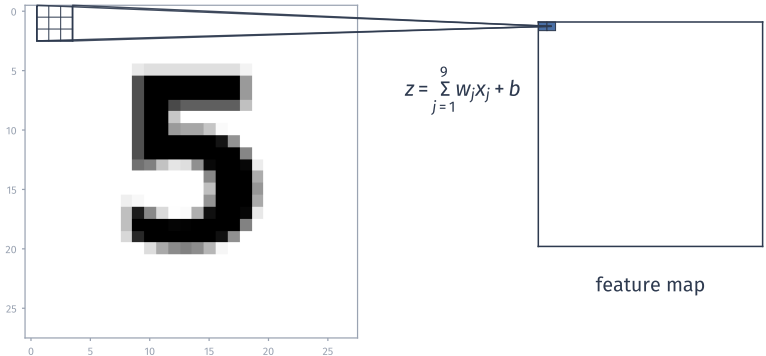
A “feature detector” (filter, kernel) slides over the input to generate a feature map

The pixels are referred to as the “receptive field”



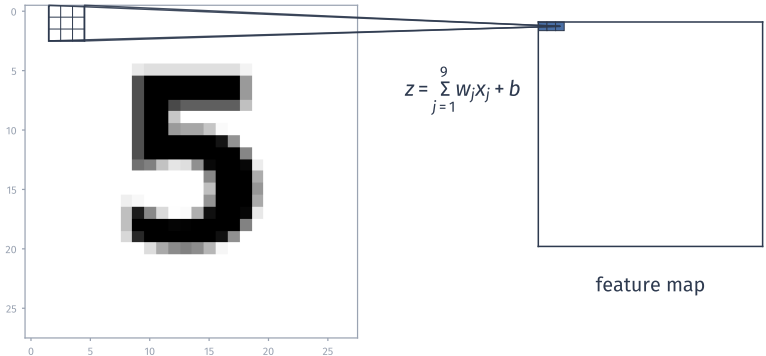
Weight sharing

A “feature detector” (filter, kernel) slides over the input to generate a feature map



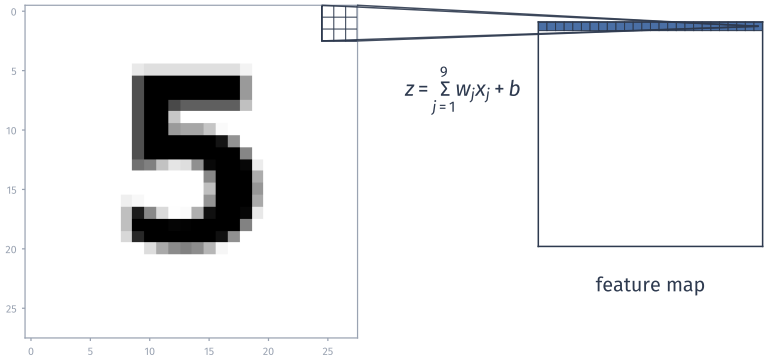
Weight sharing

A “feature detector” (filter, kernel) slides over the input to generate a feature map

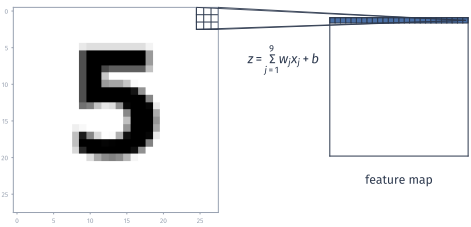


Weight sharing

A “feature detector” (filter, kernel) slides over the input to generate a feature map



Weight sharing

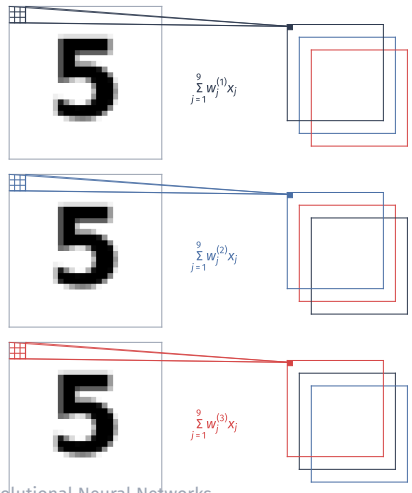


Rationale: a feature detector that works well in one region may also work well in another region

Plus, a nice reduction in parameters to fit:

$3 \times 3 + 1 = 10$ weights for the whole 26×26 map, instead of 784×676 for a fully connected layer

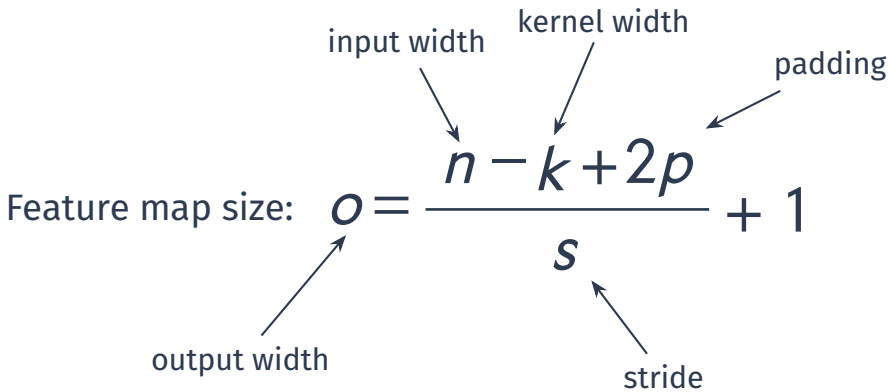
Multiple feature detectors



Multiple “feature detectors” (kernels) are used to create multiple feature maps

one kernel ⟹ one feature map
(one “channel” of the next layer)

Size before and after convolutions



Kernel dimensions and trainable parameters

```
a.shape           # one grayscale image
# (1, 28, 28)

import torch
conv = torch.nn.Conv2d(in_channels=1,
                       out_channels=8,
                       kernel_size=(5, 5),
                       stride=(1, 1))
conv.weight.size()
# torch.Size([8, 1, 5, 5])
conv.bias.size()
# torch.Size([8])
```

A grayscale 28×28 image with a 5×5 feature detector (kernel), 8 feature maps

What is the output size for this image?

Kernel dimensions and trainable parameters

```
a.shape            # one grayscale image
# (1, 28, 28)

import torch
conv = torch.nn.Conv2d(in_channels=1,
                       out_channels=8,
                       kernel_size=(5, 5),
                       stride=(1, 1))
conv.weight.size()
# torch.Size([8, 1, 5, 5])
conv.bias.size()
# torch.Size([8])
```

A grayscale 28×28 image with a 5×5 feature detector (kernel), 8 feature maps

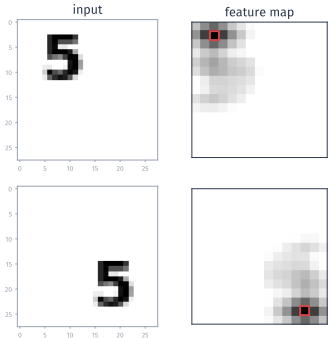
What is the output size for this image?

$$o = \frac{28 - 5 + 0}{1} + 1 = 24 \Longrightarrow$$
$$8 \times 24 \times 24$$

parameters: $8 \cdot 5 \cdot 5 + 8 = 208$

CNNs and translation / rotation / scale invariance

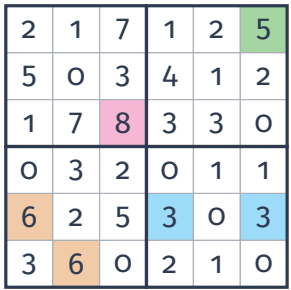
Note that CNNs are not really invariant to scale, rotation, translation, etc.



The activations are still dependent on the location

Same detector, same pattern: the strongest response (red) moves together with the digit

Pooling layers can help with local invariance



Max-pooling

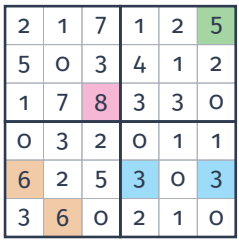
8	5
6	3

Mean-pooling

3.78	2.33
3	1.22

pooling 3×3, stride $= 3$

Pooling layers can help with local invariance



Max-pooling



Mean-pooling

3.78	2.33
3	1.22

pooling 3×3, stride $= 3$

Typical pooling layers do not have any learnable parameters

Downside: information is lost

May not matter for classification, but it does where relative position is important (e.g. face recognition). In practice, some image preprocessing is still recommended.

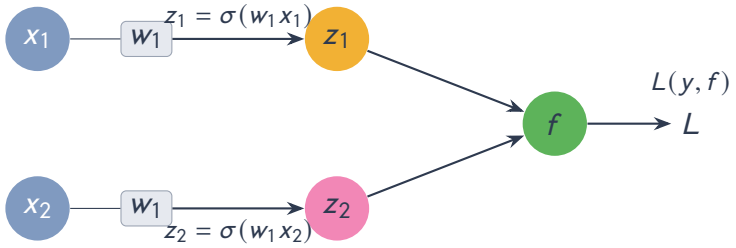
Topics

1. What CNNs can do
2. Image classification
3. Convolutional neural network basics
4. Convolutional filters and weight sharing
5. **CNNs and backpropagation**
6. CNN architectures
7. What a CNN can see
8. CNNs in PyTorch

Backpropagation in CNNs

Same overall concept as before: multivariable chain rule, but now with an additional weight sharing constraint

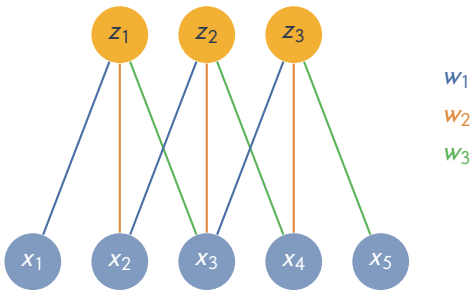
Recall: a graph with weight sharing



$$\frac{\partial L}{\partial w_1} = \underbrace{\frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}}_{\text{upper path}} + \underbrace{\frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_1}}_{\text{lower path}}$$

A weight used in several places receives **one term per place**, and the terms are **added**

A convolution is a sparse layer with shared weights



1D example: input $\mathbf{x} \in \mathbb{R}^5$, kernel $\mathbf{w} = (w_1, w_2, w_3)$

$$z_j = w_1 x_j + w_2 x_{j+1} + w_3 x_{j+2}$$

- each output sees only 3 inputs **(sparse)**
- edges of the same color share the **same weight**

It is an MLP layer whose weight matrix is mostly zeros and has tied entries.

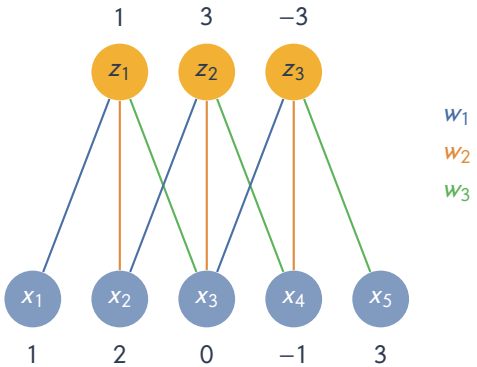
A convolution is a sparse layer with shared weights

The same layer written as a matrix–vector product, $\mathbf{z} = \mathbf{M}\mathbf{x}$

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} w_1 & w_2 & w_3 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}$$

15 entries, but only 3 free parameters: backprop works exactly as in an MLP, then the gradients of tied entries are added

Forward pass



$\mathbf{x} = (1, 2, 0, -1, 3)$,
$\mathbf{w} = (1, 0, -1)$

$z_1 = 1 \cdot 1 + 0 \cdot 2 + (-1) \cdot 0 = 1$

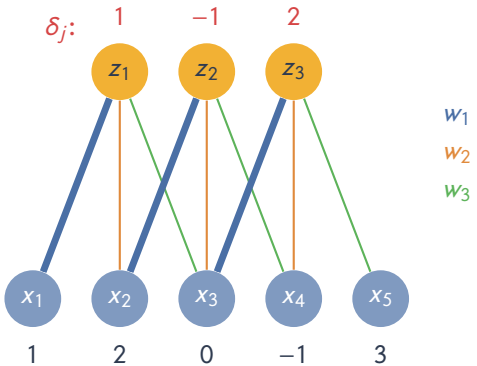
$z_2 = 1 \cdot 2 + 0 \cdot 0 + (-1) \cdot (-1) = 3$

$z_3 = 1 \cdot 0 + 0 \cdot (-1) + (-1) \cdot 3 = -3$

Backward pass: suppose the layers above give $\delta_j = \dfrac{\partial L}{\partial z_j}$, with

$\boldsymbol{\delta} = (1, -1, 2)$

Gradient with respect to the kernel



w_1 is used at **every** position j:

$$\frac{\partial L}{\partial w_1} = \sum_j \frac{\partial L}{\partial z_j}\frac{\partial z_j}{\partial w_1} = \sum_j \delta_j\, x_j$$

$$= 1 \cdot 1 + (-1) \cdot 2 + 2 \cdot 0 = -1$$

likewise:

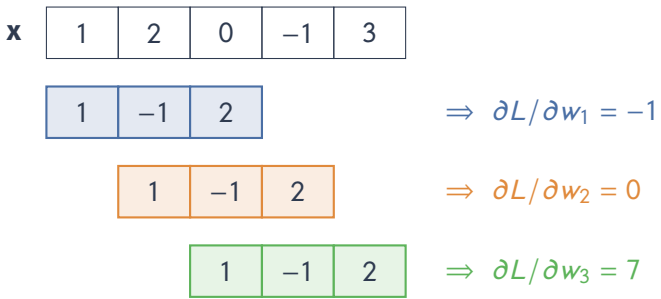
$$\partial L / \partial w_2 = \textstyle\sum_j \delta_j x_{j+1} = 2 + 0 - 2 = 0$$

$$\partial L / \partial w_3 = \textstyle\sum_j \delta_j x_{j+2} = 0 + 1 + 6 = 7$$

Gradient with respect to the kernel

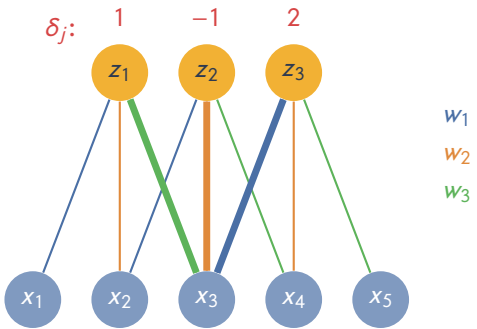
$$\frac{\partial L}{\partial w_u} = \sum_j \delta_j \, x_{j+u-1}$$

slide $\boldsymbol{\delta} = (1, -1, 2)$ over $\mathbf{x} = (1, 2, 0, -1, 3)$



The gradient of the kernel is itself a cross-correlation: input ⋆ upstream gradient

Gradient with respect to the input



To keep going **backwards**, we also need $\partial L / \partial x_i$:

x_3 influenced z_1, z_2 and z_3:

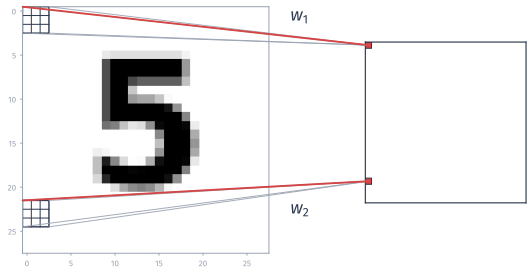
$$\frac{\partial L}{\partial x_3} = \delta_1 w_3 + \delta_2 w_2 + \delta_3 w_1$$

$$= 1 \cdot (-1) + (-1) \cdot 0 + 2 \cdot 1 = 1$$

all inputs: $\frac{\partial L}{\partial \mathbf{x}} = (1, -1, 1, 1, -2)$

(border inputs touch fewer outputs: fewer terms)

Weight update with weight sharing



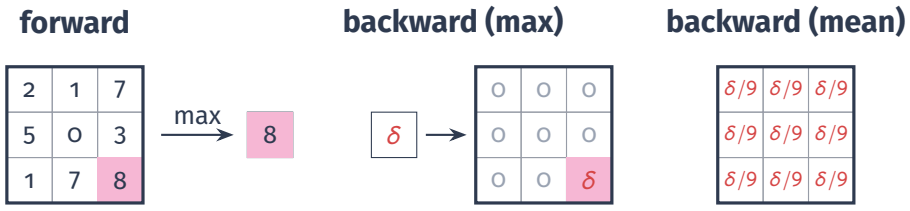
In 2D the same thing happens: the kernel weight w is used at every position of the image ($w_1 = w_2 = \dots$)

One gradient for the shared weight, the **sum** over all positions:

$$w := w - \alpha \sum_{\text{positions}} \frac{\partial L}{\partial w_{\text{position}}}$$

(PyTorch sums; dividing by the number of positions is an optional averaging)

Backpropagation through pooling



Max-pooling sends the gradient **only to the winner**; mean-pooling spreads it equally

Pooling has no parameters: it only routes gradients to the layer below

Checking with PyTorch autograd

```
import torch
import torch.nn.functional as F

x = torch.tensor([[[1., 2., 0., -1., 3.]]],
                 requires_grad=True)
w = torch.tensor([[[1., 0., -1.]]],
                 requires_grad=True)

z = F.conv1d(x, w)         # forward
print(z)  # [[[ 1.,  3., -3.]]]

delta = torch.tensor([[[1., -1., 2.]]])
z.backward(delta)          # backward
print(w.grad)  # [[[-1.,  0.,  7.]]]
print(x.grad)  # [[[ 1., -1.,  1.,  1., -2.]]]
```

Same numbers as by hand

In practice, `loss.backward()` does all of this automatically for every convolutional, pooling and fully connected layer

Topics

1. What CNNs can do
2. Image classification
3. Convolutional neural network basics
4. Convolutional filters and weight sharing
5. CNNs and backpropagation
6. **CNN architectures**
7. What a CNN can see
8. CNNs in PyTorch

Main breakthrough for CNNs: AlexNet & ImageNet

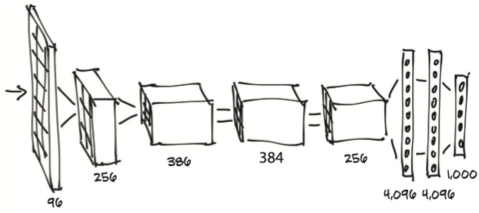


Figure: Stevens, Antiga & Viehmann (2020), Deep Learning with PyTorch, Manning.

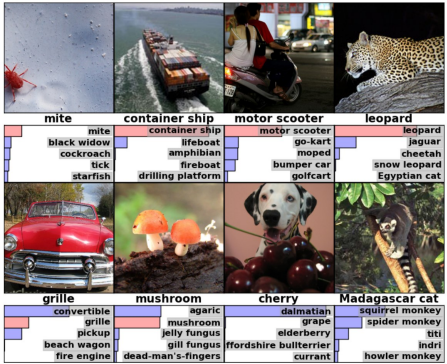
AlexNet achieved **15.4%** top-5 error in 2012

the 2nd best was not even close: 26.2%

nowadays: ≈ 3% error on ImageNet

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. NeurIPS, pp. 1097–1105.

Main breakthrough for CNNs: AlexNet & ImageNet



The actual network inputs were 224×224 images (random crops of downsampled 256×256 images)

224×224 is still a reasonable size today:
$224 \cdot 224 \cdot 3 = 150{,}528$ features

Common CNN architectures

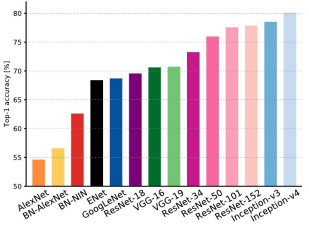


Figure 1: **Top1 *vs.* network.** Single-crop top-1 validation accuracies for top scoring single-model architectures. We introduce with this chart our choice of colour scheme, which will be used throughout this publication to distinguish effectively different architectures and their correspondent authors. Notice that networks of the same group share the same hue, for example ResNet are all variations of pink.

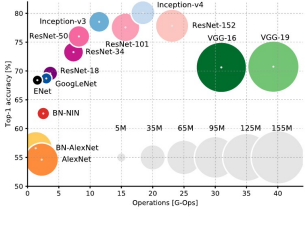
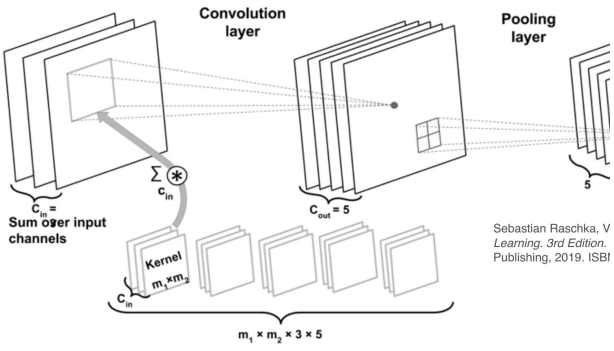


Figure 2: **Top1 *vs.* operations, size $\propto$ parameters.** Top-1 one-crop accuracy versus amount of operations required for a single forward pass. The size of the blobs is proportional to the number of network parameters; a legend is reported in the bottom right corner, spanning from 5×10^6 to 155×10^6 params. Both these figures share the same y-axis, and the grey dots highlight the centre of the blobs.

Canziani, A., Paszke, A., & Culurciello, E. (2016). An analysis of deep neural network models for practical applications. arXiv:1605.07678.

Convolutions with color channels



Sebastian Raschka, V
Learning. 3rd Edition.
Publishing, 2019. ISBI

Image: $\mathbf{X} \in \mathbb{R}^{n_1 \times n_2 \times c_{in}}$

Each kernel has size $m_1 \times m_2 \times c_{in}$ and **sums over the input channels**

c_{out} kernels $\Rightarrow$ c_{out} feature maps

NHWC format (above); CUDA & PyTorch use **NCHW**

Raschka, S., & Mirjalili, V. (2019). Python Machine Learning, 3rd ed. Packt.

Topics

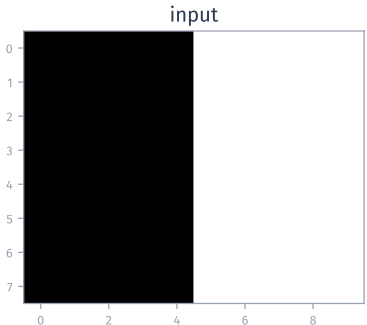
1. What CNNs can do
2. Image classification
3. Convolutional neural network basics
4. Convolutional filters and weight sharing
5. CNNs and backpropagation
6. CNN architectures
7. **What a CNN can see**
8. CNNs in PyTorch

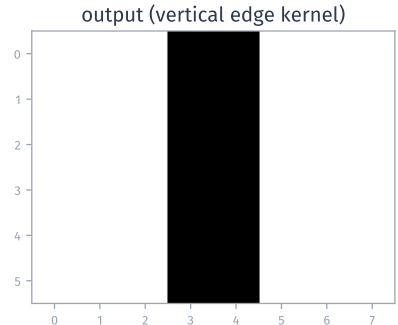
What a CNN can see

Simple example: vertical edge detector (from classical computer vision research)

kernel

$$\begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$



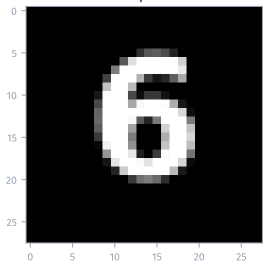


The output is non-zero only where the intensity changes from left to right

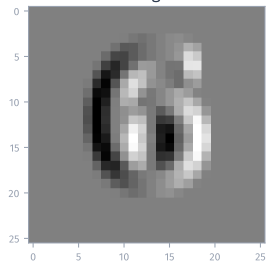
What a CNN can see

Vertical and horizontal edge detectors on a digit

input

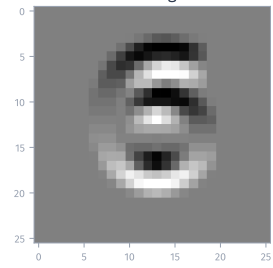


vertical edge kernel



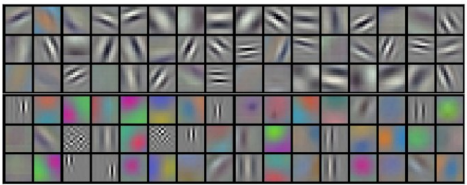
$$\begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$

horizontal edge kernel



$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

What a CNN can see



The 96 kernels of size $11 \times 11 \times 3$ learned by the first layer of AlexNet

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). NeurIPS.

A CNN can learn whatever it finds best by optimizing the objective

(e.g. minimizing a particular loss to achieve good classification accuracy)

Topics

1. What CNNs can do
2. Image classification
3. Convolutional neural network basics
4. Convolutional filters and weight sharing
5. CNNs and backpropagation
6. CNN architectures
7. What a CNN can see
8. **CNNs in PyTorch**

A small CNN in PyTorch

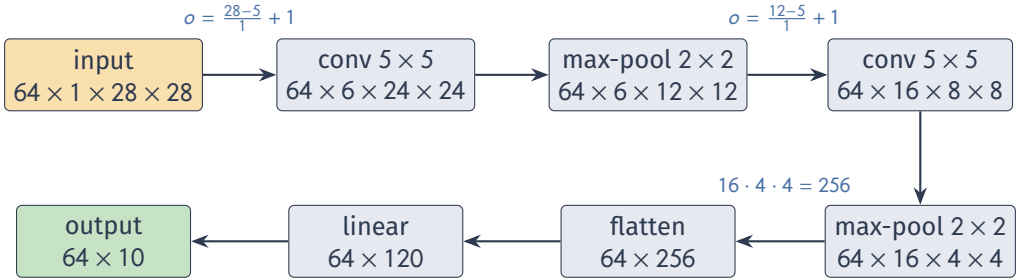
```
import torch.nn as nn

model = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5),  # 6 maps
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    nn.Conv2d(6, 16, kernel_size=5), # 16 maps
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    nn.Flatten(),
    nn.Linear(16 * 4 * 4, 120),
    nn.ReLU(),
    nn.Linear(120, 10),              # 10 classes
)
```

A LeNet-like network for 28×28 grayscale digits

The training loop is **the same** as for an MLP: batch, forward pass, loss, `zero_grad()`, `backward()`, `step()`

34,622 parameters in total

Following the shapes



Format NCHW: batch $\times$ channels $\times$ height $\times$ width (batch of 64 images)