

Neural Networks for Classification

Convolutional Networks for Image Classification

Joseph Le Roux

11/08/26



Outline

- Recap and Goal
- Convolutions
- Convolutions For Image Processing
- Convolution Network Architectures
- Conclusion



1

Recap and Goal

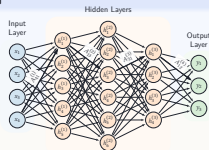
1 Build a Classifier for Images

From a set of images classified into **predefined** categories, we want to build: a system able to **classify new images**

Input



Output



dog

cat

hamburger

Two-step process

1. Compute a probability for **each** category
2. Return the **most probable** category

Data Mining perspective

1. Use a **Neural Network** (NN) to compute probabilities
2. **Learn NN parameters** from examples

Today

Exploit input image **structure** (rectangular grids of pixels) to build better classifiers

- Use **convolutions** to build feature extractors
- Features represent bigger and bigger **parts** of images
 - **deep** architectures: local pixels → neighbourhood → zone → global → prediction

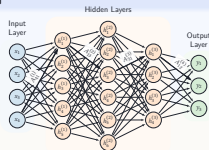
1 Build a Classifier for Images

From a set of images classified into **predefined** categories, we want to build: a system able to **classify new images**

Input



Output



dog 0.14

cat 0.85

hamburger 0.01

Two-step process

1. Compute a probability for **each** category
2. Return the **most probable** category

Data Mining perspective

1. Use a **Neural Network** (NN) to compute probabilities
2. **Learn NN parameters** from examples

Today

Exploit input image **structure** (rectangular grids of pixels) to build better classifiers

- Use **convolutions** to build feature extractors
- Features represent bigger and bigger **parts** of images
 - **deep** architectures: local pixels → neighbourhood → zone → global → prediction

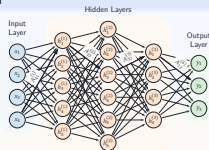
1 Build a Classifier for Images

From a set of images classified into **predefined** categories, we want to build: a system able to **classify new images**

Input



Output



dog 0.14

cat 0.85

hamburger 0.01

Two-step process

1. Compute a probability for **each** category
2. Return the **most probable** category

Data Mining perspective

1. Use a **Neural Network** (NN) to compute probabilities
2. **Learn NN parameters** from examples

Today

Exploit input image **structure** (rectangular grids of pixels) to build better classifiers

- Use **convolutions** to build feature extractors
- Features represent bigger and bigger **parts** of images
 - **deep** architectures: local pixels → neighbourhood → zone → global → prediction



2

Convolutions

Discrete Convolution between two vectors

Convolution of filter a over input v

$$f = a * v \text{ such that } f_n = \sum_{m=-\infty}^{+\infty} a_m v_{n-m}$$

Can be difficult to read. Let's rewrite it:

$$f_n = \sum_{p,q \text{ s.t. } p+q=n} a_p v_q$$

Example $a = [1, 2, 3]$ and $v = [4, 5, 6, 7]$ give $f = [4, 13, 28, 34, 32, 21]$

- $f_0 = a_0 v_0 = 4$
- $f_1 = a_0 v_1 + a_1 v_0 = 13$
- $f_2 = a_0 v_2 + a_1 v_1 + a_2 v_0 = 28$
- $f_3 = a_0 v_3 + a_1 v_2 + a_2 v_1 = 34$
- $f_4 = a_1 v_3 + a_2 v_2 = 32$
- $f_5 = a_2 v_3 = 21$

This is straightforward in numpy

```
import numpy as np
print(np.convolve([1,2,3], [4,5,6,7]))
```

```
[ 4 13 28 34 32 21]
```

2 Convolution Layers in Pytorch

In Pytorch this is called a 1D convolution.

Let's code this with PyTorch API

```
import torch
conv = torch.nn.Conv1d(1, #each input position in (4,5,6) is 1 number
                        1, #each output is 1 number (1 filter)
                        kernel_size=3, #convolution has size 3 (1,2,3)
                        padding=2, #at most vector a has 2 positions out
                        bias=False) #combination is linear, not affine
conv.weight.requires_grad=False # so we can change values by hand
#print(conv.weight.size()) # (1,1,3) (out,in,kernel)
conv.weight[0][0] = torch.tensor([3,2,1]) # in reverse order

print(conv(torch.tensor([4.,5.,6.,7.]).unsqueeze(0)).squeeze(0))
```

```
tensor([ 4., 13., 28., 34., 32., 21.])
```

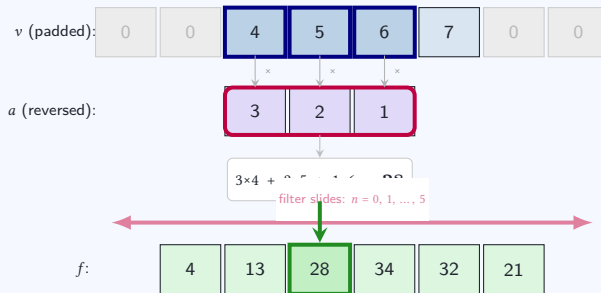
Remarks

- this can take a vector as each position and return a vector at each position (hence `unsqueeze/squeeze`)
 - the position wise multiplication is replaced by dot/biaffine products
- we can control the number of surrounding zeros (`padding`, see next slide)

to recover numpy's implementation: set to 2 with filter of size 3

2 1D Convolution: Sliding Window

At each step n , the filter computes a weighted sum with the aligned input segment

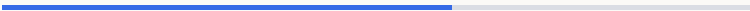


Convention

Mathematical convolution *reverses* the filter before sliding (hence $[3, 2, 1]$ above). In practice, CNNs use *cross-correlation* (no reversal) and call it “convolution” by convention — which is what `torch.nn.Conv1d` implements.

3

Convolutions For Image Processing



3 2D Convolutions for Images: From 1d to 2D

Why convolution?

Filter dot product operations can be seen as a [similarity function](#)

- convolution reports whether some part of the input [matches](#) the filter
- $\Rightarrow$ convolution = [feature detection](#)

Extension to Geometric Transformations

- instead of a single input value, a vector per input position
- instead of a single score per position, a [vector](#) score
- $\Rightarrow$ convolution = [feature detection](#) + [geometric transformation](#)

Vectors $\rightarrow$ Matrices

- Image as a matrix M , i.e. 32×32 , where $M[i][j]$ between 0 (black) and 1 (white) (colours later)
- filter A is also a (usually square) matrix $k \times k$, ($k = 3, 5 \dots$ odd)
 - will be applied to each patch (a $k \times k$ part of the input) to compute similarities and transformation

need to fix a small discrepancy between the *mathematical* convolution and the *computer vision* convolution.

if we adapt the previous formula, the output is a matrix f where:

$$f_{ij} = \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} A_{nm} M_{i-n, j-m}$$

little drawing shows this is not exactly what we want

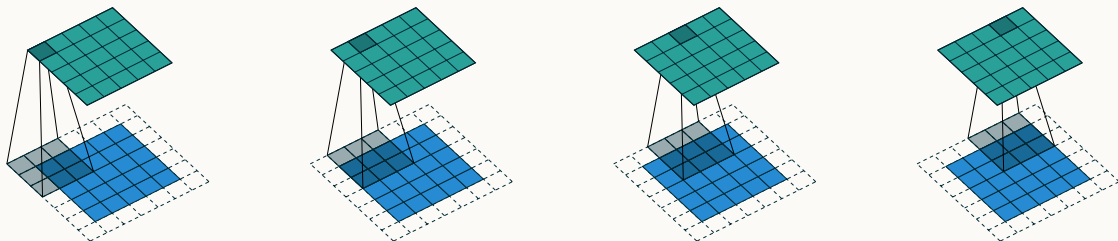
let's have the filter "centered" around position ij

Define $p = \lfloor \frac{k}{2} \rfloor$

$$f_{ij} = \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} A_{p+n, p+m} M_{i-n, j-m}$$

- convolution $\equiv$ transformation of a position ij using its neighbourhood of size p

from https://github.com/vdumoulin/conv_arithmetic



See also computation/animation on <https://cs231n.github.io/convolutional-networks/>

3 From Matrices to Tensors, add Depth

In practice

Images are not matrices, but 3D-Tensors

- each point has several pieces of information, called **channels**, encoded as vectors
- for instance:
 - 3 channels for RGB coding (red/green/blue)
 - as integer in $[0; 255]$ or float in $[0; 1]$
- Images are encoded as 3D tensors: $C \times W \times H$ with $C = 3$
- Full generalisation: add bias term b to compute affine functions

$$f_{ij} = b + \sum_{c=0}^{C-1} \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} A_{c,p+n,p+m} M_{c,i-n,j-m} = b + \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} \langle A[:,p+n,p+m], M[:,i-n,j-m] \rangle$$

Remark this return a matrix f , more precisely 3D tensor with depth 1

From Convolutions to Volume Transformation

Several Filters in parallel and concatenate $\equiv$ Convolution as a transformation $C \times W \times H \rightarrow D \times W \times H$

3 Hand-Crafted Convolution: Examples

What can we do with convolutions? Alter each pixel! Why?

Intuition

- Let's try to fill convolution parameters to implement some image transformations
- If we can do it, this means that learning parameters could implement these operations too!

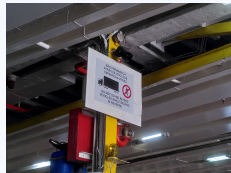
Example (preliminary: load picture)

First, load a picture

```
import torch
from torchvision import io
import matplotlib.pyplot as plt

img = io.read_image("../static/IMG_20220726_161509.jpg")

fig = plt.figure(figsize=(img.size(1)/ 150, img.size(2) /
    ↪ 150))
plt.imshow(img.permute(1,2,0))
```



Average (blurring effect)

Second, apply an average convolution

```
ks, n_ch = 20,3 # kernel size (patch size) and number of
↳ channels (RGB)
```

```
conv = torch.nn.Conv2d(n_ch, n_ch, ks, padding=ks-n_ch +
↳ 1, bias=False) #no bias
conv.weight.requires_grad=False #so we can replace values
↳ manually
```

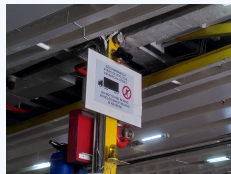
```
print(conv.weight.size()) #(3,3,20,20)
conv.weight[:, :, :, :] = torch.zeros(n_ch,n_ch,ks,ks)
↳ #reset all values to zero
```

```
patch = torch.ones(ks,ks) / (ks*ks) # uniform dist.
```

```
conv.weight[0][0] = patch # for R channel
conv.weight[1][1] = patch # for G channel
conv.weight[2][2] = patch # for B channel
```

```
img = img.to(torch.float)/256 # all values between 0 and
↳ 1
```

```
blurred = conv(img)
io.write_jpeg((256*blurred).to(torch.uint8),
↳ "blurred.jpg", 100)
```



Average (blurring effect)

Second, apply an average convolution

```
ks, n_ch = 20,3 # kernel size (patch size) and number of
↳ channels (RGB)
```

```
conv = torch.nn.Conv2d(n_ch, n_ch, ks, padding=ks-n_ch +
↳ 1, bias=False) #no bias
conv.weight.requires_grad=False #so we can replace values
↳ manually
```

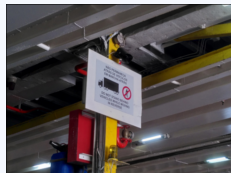
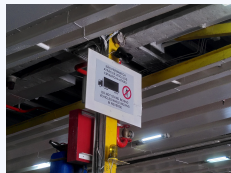
```
print(conv.weight.size()) #(3,3,20,20)
conv.weight[:, :, :, :] = torch.zeros(n_ch,n_ch,ks,ks)
↳ #reset all values to zero
```

```
patch = torch.ones(ks,ks) / (ks*ks) # uniform dist.
```

```
conv.weight[0][0] = patch # for R channel
conv.weight[1][1] = patch # for G channel
conv.weight[2][2] = patch # for B channel
```

```
img = img.to(torch.float)/256 # all values between 0 and
↳ 1
```

```
blurred = conv(img)
io.write_jpeg((256*blurred).to(torch.uint8),
↳ "blurred.jpg", 100)
```



Sharpening (edge detection)

```

ks, n_ch = 3, 3 # kernel size (patch size) and number of
↳ channels (RGB)

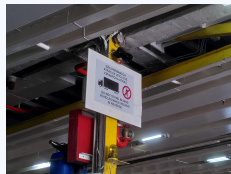
conv = torch.nn.Conv2d(n_ch, n_ch, ks, padding=ks - n_ch
↳ + 1, bias=False) # no bias
conv.weight.requires_grad = False # so we can replace
↳ values manually
print(conv.weight.size()) # (3,3,3,3)
conv.weight[:, :, :, :] = torch.zeros(n_ch, n_ch, ks, ks)
↳ # reset all values to zero

patch = (torch.tensor([[0, -1, 0],
                        [-1, 4, -1],
                        [0, -1, 0]])) / 4)

conv.weight[0][0] = patch # for R channel
conv.weight[1][1] = patch # for G channel
conv.weight[2][2] = patch # for B channel

img = img.to(torch.float)/256 # all values between 0 and
↳ 1
sharpened = conv(img)
io.write_jpeg((256*sharpened).to(torch.uint8),
↳ "sharpened.jpg", 100)

```



Sharpening (edge detection)

```

ks, n_ch = 3, 3 # kernel size (patch size) and number of
↳ channels (RGB)

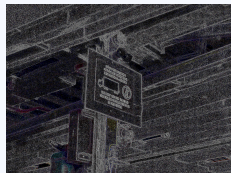
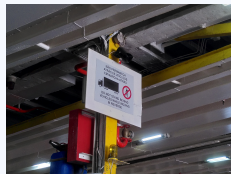
conv = torch.nn.Conv2d(n_ch, n_ch, ks, padding=ks - n_ch
↳ + 1, bias=False) # no bias
conv.weight.requires_grad = False # so we can replace
↳ values manually
print(conv.weight.size()) # (3,3,3,3)
conv.weight[:, :, :, :] = torch.zeros(n_ch, n_ch, ks, ks)
↳ # reset all values to zero

patch = (torch.tensor([[0, -1, 0],
                        [-1, 4, -1],
                        [0, -1, 0]])) / 4)

conv.weight[0][0] = patch # for R channel
conv.weight[1][1] = patch # for G channel
conv.weight[2][2] = patch # for B channel

img = img.to(torch.float)/256 # all values between 0 and
↳ 1
sharpened = conv(img)
io.write_jpeg((256*sharpened).to(torch.uint8),
↳ "sharpened.jpg", 100)

```



3 Learned Convolution: Example

From the 2012 ImageNet challenge Krizhevsky et al.



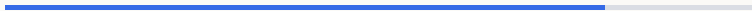
Visualisation of $3 \times 11 \times 11$ filters learned by the model after training.

Observations

- black/white/grey $\rightarrow$ same filter for all RGB channels
- some filters detect edges/ basic shapes
- others are more sensible to colour changes

4

Convolution Network Architectures



1/ Prepare Representation for Classification

Channels represent information about points (samples)

- at the beginning, just info about pixel colours (RGB): $C \times W \times H$
 - $C = 3$ and W, H so each image point is represented
- at the end $N \times 1 \times 1$
 - N = number of classes for classification
 - 1×1 one point representing the whole image

3/ Implementation:

- step 1 implemented by convolution
- step 2 implemented by pooling (see next)
- step 3 implemented by MLP

2/ Pipeline view

1. incorporate incrementally neighbours' info
2. expand from info about image points to zones/areas
3. to bigger areas until prediction on entire image

Steps 1 and 2 repeated, step 3 MLP classifier

A pipeline view of Convolution Network:

$$C_0 \times W_0 \times H_0 \rightarrow C_1 \times W_1 \times H_1 \rightarrow \dots \rightarrow C_n \times W_n \times H_n \rightarrow \text{Classifier}$$

where in general:

- $C_0 = 3, W_0 = W, H_0 = H$
- $C_{i+1} \geq C_i, W_{i+1} \leq W_i, H_{i+1} \leq H_i$
- $C_n = \text{NB_CLASSES}, W_n, H_n = 1$

Compute several filters at each position

Change depth, preserve dimension → increase volume

Can also be used to decrease the depth, but in practice we increase it

Change depth from 3 to 12

```
ks, n_in_ch, n_out_ch = 5, 3, 12 # kernel size, input channels, output channels
conv = torch.nn.Conv2d(n_in_ch, n_out_ch, ks) # compute n_out_ch filters
print(conv.weight.size(), conv.bias.size()) # (12, 3, 5, 5), (12)

print(img.size()) # (3, 640, 480) each pixel has 3 informations
img_w_neighbour_info = conv(img)
print(img_w_neighbour_info) # (12, 636, 476) whaaaat!!!
```

- Right number of output channels, but image is cropped (border of size 2) 🤖
- Convolutions are *centered*, so extreme points are ignored (cannot be centres)

Optional: Add a Non-Linearity

Convolutions is a linear (affine) operation: add ReLU or other activation after convolution to represent non-linear operations on channels

How many consecutive k positions in a vector of size n

- $n - k + 1$ (draw picture if need be)
- this means that each convolution *loses* some positions

Padding to control the output size

- use padding to add *fake* positions around image (zero by default)
- so every original point can be a centre

change depth from 3 to 12, preserve dimensions

```
ks, n_in_ch, n_out_ch = 5, 3, 12
conv = torch.nn.Conv2d(n_in_ch, n_out_ch, ks, padding=2)

print(img.size()) # (3, 640, 480) each pixel has 3 informations
img_w_neighbour_info= conv(img)
print(img_w_neighbour_info) # # (12, 640, 480)
```

Remark

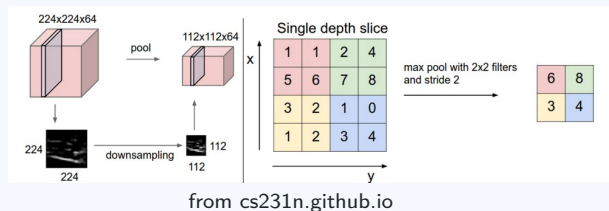
Padding can also be used to crop image (remove border points)

Pooling: Summarise Zone as Single Point

Change dimensions, Preserve depth → decrease volume

Example: Max-Pooling

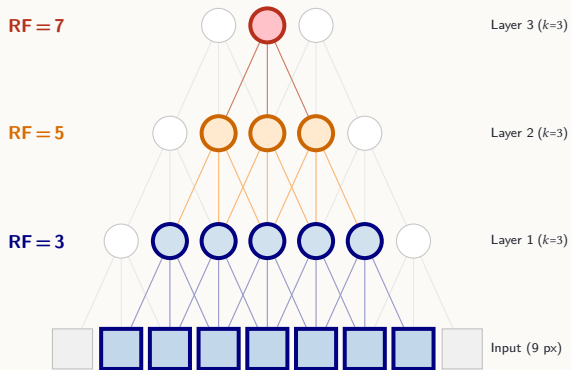
Implement pooling by returning for each channel the maximum value on a patch (zone).



```
print(img_w_neighbour_info)
# just a reminder, return (12, 640, 480)
mp = torch.nn.MaxPool2D(2)
img_small = mp(img_w_neighbour_info)
print(img_small.size()) # 12,320,240
```

Other Pooling Methods, Instead of max use:

- min
- average
- ... quizz: What properties does this operation need? 🤔



Takeaway

Stacking L conv-layers with kernel k gives **receptive field** $= 1 + L(k-1)$. With pooling, the spatial grid shrinks while depth grows: the highlighted centre neuron already sees 7 px after 3 layers; a few more layers and pooling cover the entire image, enabling global prediction from local operations.

4 Putting it all together

For MNIST digit classification:

- B&W images: original depth is 1
- 10 classes

```
class ConvNet1(torch.nn.Module):
    def __init__(self,):
        super().__init__()

        self.c1 = torch.nn.Conv2d(1, 10, 3, padding=1)
        self.m1 = torch.nn.MaxPool2D(2)
        self.c2 = torch.nn.Conv2d(10, 20, 5)
        self.m2 = torch.nn.MaxPool2D(2)
        self.c3 = torch.nn.Conv2d(20, 40, 3)
        self.m3 = torch.nn.MaxPool2D(3)
        self.fc = MLP(40, [30, 20], NB_CLASSES, torch.nn.ReLU)

    def forward(self, x):
        #print(x.size())
        x = torch.nn.functional.relu(self.c1(x))
        x = self.m1(x)
        x = torch.nn.functional.relu(self.c2(x))
        x = self.m2(x)
        x = torch.nn.functional.relu(self.c3(x))
        x = self.m3(x)
        x = x.view(-1) #quiz: why is this for?
        x = self.fc(x)
        return x
```


4 Putting it all together

For MNIST digit classification:

- B&W images: original depth is 1
- 10 classes

Exercise: write each *volume* $C_i \times W_i \times H_i$

```
class ConvNet1(torch.nn.Module):
    def __init__(self,):
        super().__init__()

        self.c1 = torch.nn.Conv2d(1, 10, 3, padding=1)
        self.m1 = torch.nn.MaxPool2D(2)
        self.c2 = torch.nn.Conv2d(10, 20, 5)
        self.m2 = torch.nn.MaxPool2D(2)
        self.c3 = torch.nn.Conv2d(20, 40, 3)
        self.m3 = torch.nn.MaxPool2D(3)
        self.fc = MLP(40, [30, 20], NB_CLASSES, torch.nn.ReLU)

    def forward(self, x):
        #print(x.size())
        x = torch.nn.functional.relu(self.c1(x))
        x = self.m1(x)
        x = torch.nn.functional.relu(self.c2(x))
        x = self.m2(x)
        x = torch.nn.functional.relu(self.c3(x))
        x = self.m3(x)
        x = x.view(-1) #quiz: why is this for?
        x = self.fc(x)
        return x
```



5

Conclusion

5 Recap

Main Idea

Exploit the specific structure of images (grid of pixels with meaning spatial distance) to define better data representation

- an example of *inductive bias*: put some task knowledge in the definition of the neural network

Convolutional Networks Architecture

3 components

- **convolutions** integrate informations from neighbours to new representations of points
- **pooling** summarises information on a patch (+ non-linearity)
- **MLP classifier** to transform image representation to class scores

Deep Architecture convolutions and pooling are repeated to:

1. build richer representations with convolutions
2. represent bigger and bigger image areas with pooling

From the 1990s to the 2020s

Old architecture but still state-of-the-art today (but Visual Transformers are coming strong!)

5 Extensions / Generalisations

Stride

Convolution can *skip* some positions (see lab session for examples)

Dropout

Randomly set neurons to zero during training, avoid overfitting, make the learned model more robust

Normalization

In image processing (ML in general), often a good idea to normalize input (center values around zero, with variance 1...)

Transformers

Can we develop an architecture where the $\text{kernel}_{\text{size}}$ can be as large as the size of the picture?

Generation

Can we use convolutions to generate image instead of classifying them? (Use Bayes theorem to transform back to classification)