The Abdus Salam International Centre for Theoretical Physics

College on Medical Physics 2026 | (smr 4232)

August, 26th, 2026

Practical AI COMPUTER SESSION 1 - Informatic Laboratory: AI basic concepts

Introduction to CNN in Medical Imaging (with GradCAM ex-post analysis)

Alessandro Bombini
Istituto Nazionale di Fisica Nucleare, Firenze

INFN

Fondazione ICSC
Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

Before the beginning

Material

The Hands-on session is freely, openly available at the GitHub repository:
https://github.com/androbomb/ICTP-College-on-Medical-Physics-2026-AI-basics-concepts

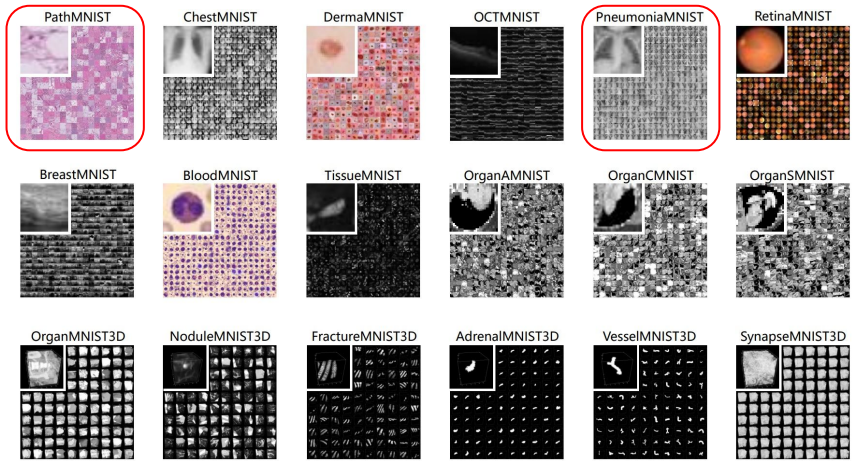
The dataset used for the Hands-on are from **MedMNIST** [1,2] (https://medmnist.com/)

We will start by a brief exploratory data analysis,

then we will see how to train a custom CNN model for the task,

and finally, we try to understand *why* the network predicts what it predicts, using Grad-CAM [3].

but before...



[1] Jiancheng Yang, Rui Shi, Donglai Wei, Zequan Liu, Lin Zhao, Bilian Ke, Hanspeter Pfister, Bingbing Ni. Yang, Jiancheng, et al. "MedMNIST v2-A large-scale lightweight benchmark for 2D and 3D biomedical image classification." Scientific Data, 2023.
[2] Jiancheng Yang, Rui Shi, Bingbing Ni. "MedMNIST Classification Decathlon: A Lightweight AutoML Benchmark for Medical Image Analysis". IEEE 18th International Symposium on Biomedical Imaging (ISBI), 2021.
[3] Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D. (2017). Grad-cam: Visual explanations from deep networks via gradient-based localization. In Proceedings of the IEEE international conference on computer vision (pp. 618-626).

Introduction

What is a CNN?

Learning spatial features on images

Neural Network Architectures: *a primer*

Convolutional neural networks

The number of parameters of MLP increase *quadratically* with the input/output size, and linearly with the number of layers

$$\#\text{Parameters} \sim O\left(\sum_{k=1}^{N_{layers}} \text{fan\_in}_k \cdot \text{fan\_out}_k + \text{fan\_out}_k\right)$$

Suppose we need to apply the MLP to an *image*, i.e. a bidimensional array of shape (h,w) and with multiple channels, i.e. a rank-3 tensor of shape (h,w,c) [numpy/TF notation, *channel last*] or (c,h,w) [PyTorch notation, *channel first*].

A simple, low res RGB image, e.g. (256,256,3) has **196.608 features**. A two-layer perceptron mapping image to itself (i.e., having each input size equal to the output size) would have (196.608x2 + 196.608)x2 free parameters, i.e. **1.179.648 parameters**.

This, back then, was a major issue, because it makes *going deep* computationally infeasible.

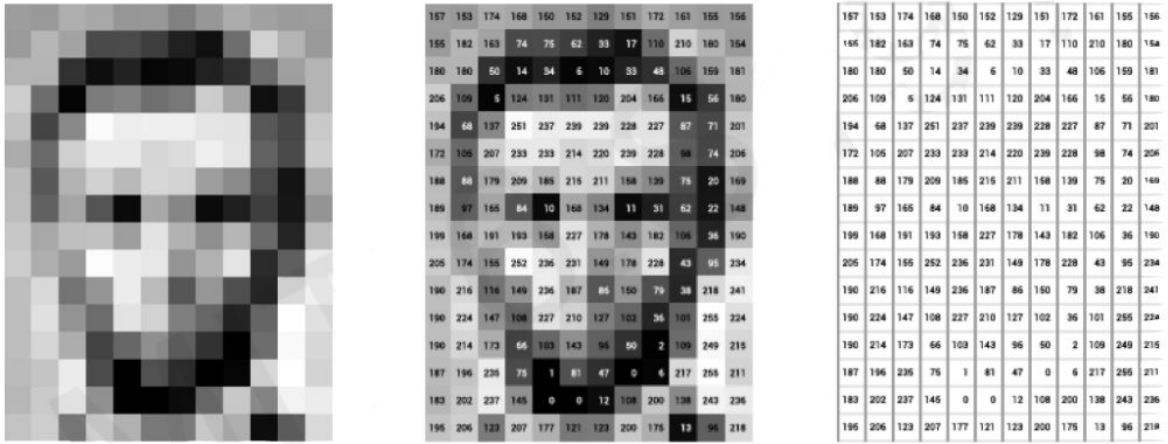
Furthermore, this kind of architecture is **not** well suited for certain specific tasks, like *image processing*.

Neural Network Architectures: *a primer*

Convolutional neural networks

For tasks like this we need a model with a more limited **expressive power**, and thus *fewer* parameters, but matching some properties of the input:

➢ symmetrical spatial structure of the input: pixels organised in a fixed size mesh



gray scale image with 8bit depth: 12x16x1 intensity ∈ [0,256]
color image with n-bit depth: $m_1 x m_2 x 3$ with each RGB intensity ∈ $[0,2^n]$

credit MIT AI course Link

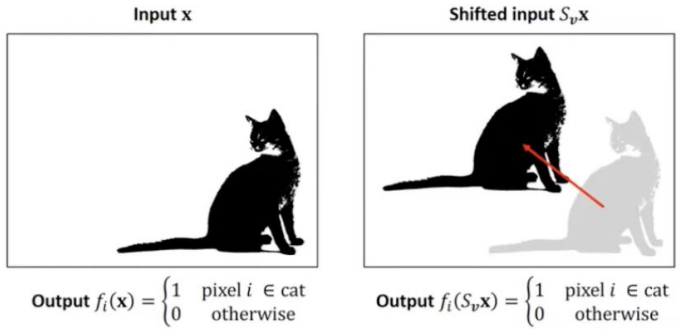
Neural Network Architectures: *a primer*

Convolutional neural networks

For tasks like this we need a model with a more limited **expressive power**, and thus *fewer* parameters, but matching some properties of the input:

- symmetrical spatial structure of the input: pixels organised in a fixed size mesh
- translation invariance (equivariance): sub-features in the image remain the same in different points of the image

Shift equivariance



- **'Cat segmentor'** $\mathbf{f}: \mathbb{R}^d \to \mathbb{R}^d$
- **Shift operator** $S_v: \mathbb{R}^d \to \mathbb{R}^d$ shifting the image by vector $\boldsymbol{v}$

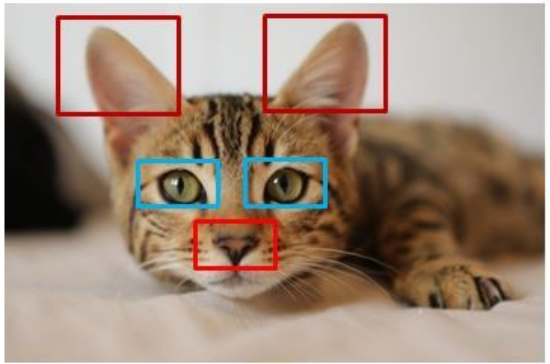
Shift equivariance: $S_v\mathbf{f}(\mathbf{x}) = \mathbf{f}(S_v\mathbf{x})$

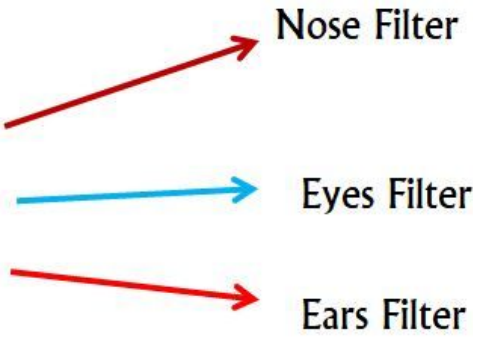
Neural Network Architectures: *a primer*

Convolutional neural networks

For tasks like this we need a model with a more limited **expressive power**, and thus *fewer* parameters, but matching some properties of the input:

- symmetrical spatial structure of the input: pixels organised in a fixed size mesh
- translation invariance (equivariance): sub-features in the image remain the same in different points of the image
- self-similarity: two or more identical sub-features present in the image can be recognized with a single filter that identifies one of the sub-features



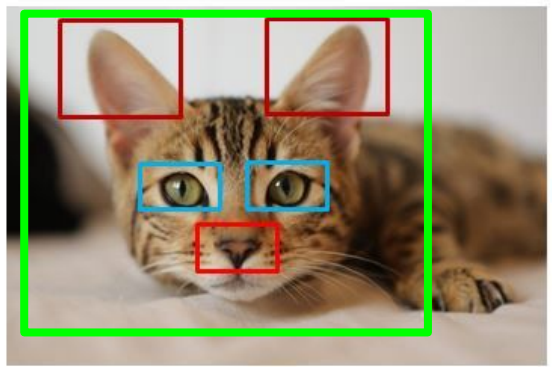


Neural Network Architectures: *a primer*

Convolutional neural networks

For tasks like this we need a model with a more limited **expressive power**, and thus *fewer* parameters, but matching some properties of the input:

- symmetrical spatial structure of the input: pixels organised in a fixed size mesh
- translation invariance (equivariance): sub-features in the image remain the same in different points of the image
- self-similarity: two or more identical sub-features present in the image can be recognized with a single filter that identifies one of the sub-features
- compositionality: a complex feature made of several sub-features can be recognized by identifying only few sub-features



face “map” from 1xNose + 2xEyes + 2xEars

Nose Filter

Eyes Filter

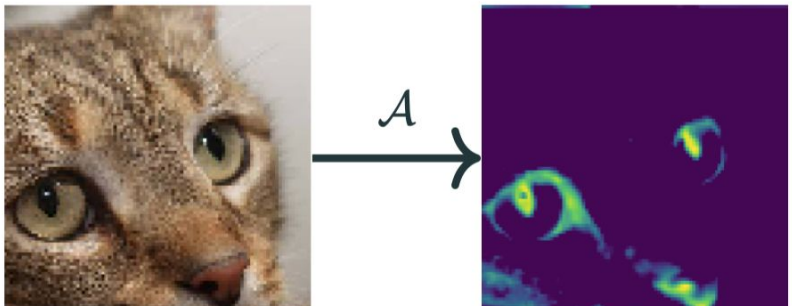
Ears Filter

Neural Network Architectures: *a primer*

Convolutional neural networks

For tasks like this we need a model with a more limited **expressive power**, and thus *fewer* parameters, but matching some properties of the input:

- symmetrical spatial structure of the input: pixels organised in a fixed size mesh
- translation invariance (equivariance): sub-features in the image remain the same in different points of the image
- self-similarity: two or more identical sub-features present in the image can be recognized with a single filter that identifies one of the sub-features
- compositionality: a complex feature made of several sub-features can be recognized by identifying only few sub-features
- locality of the features: to identify a sub-feature it often takes just a few pixels concentrated in a small portion of the image itself

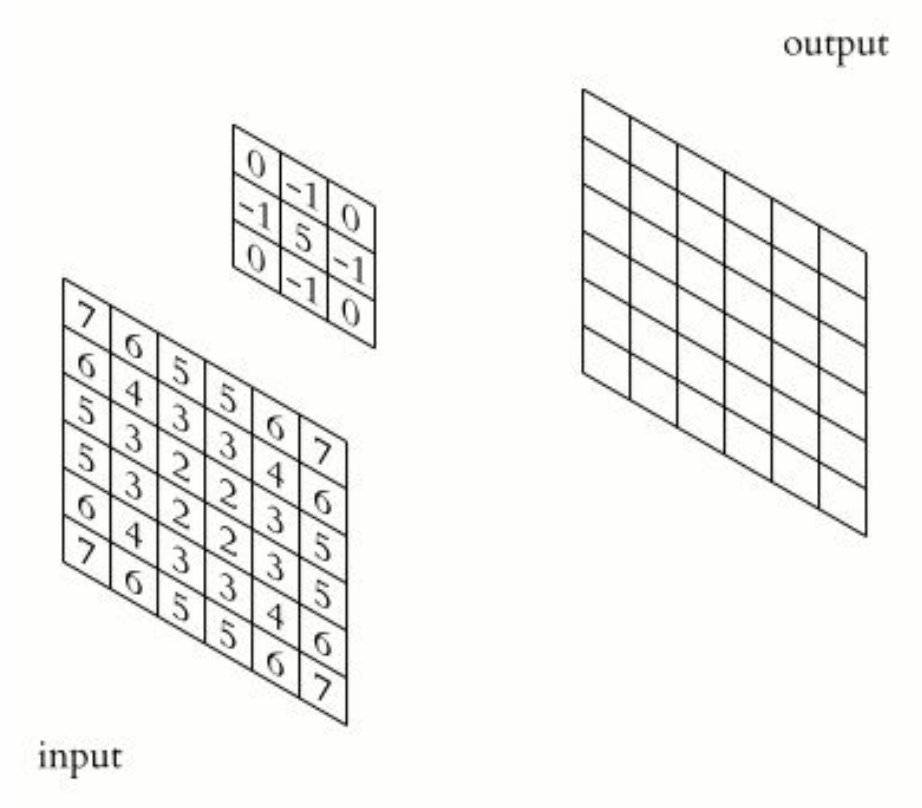


Neural Network Architectures: *a primer*

Convolutional neural networks

So the idea is: use convolutions. For Images:

- used to identify similar features that are present in different position of the image;
- based on three basic ideas:
  - *local receptive field*
    - Input neurons (one for each NxN pixels of the image) are NOT fully connected with all the neurons of the first hidden layer. Connections exist only for localised and small regions of the image called local receptive fields
    - the local receptive field is shifted through the whole image: for each shifted receptive field there will be an hidden neuron in the hidden layer
  - *shared-weights* (kernels) → **much smaller number of weights to learn**
  - [NEW]: *pooling layers* (later)



Neural Network Architectures: *a primer*

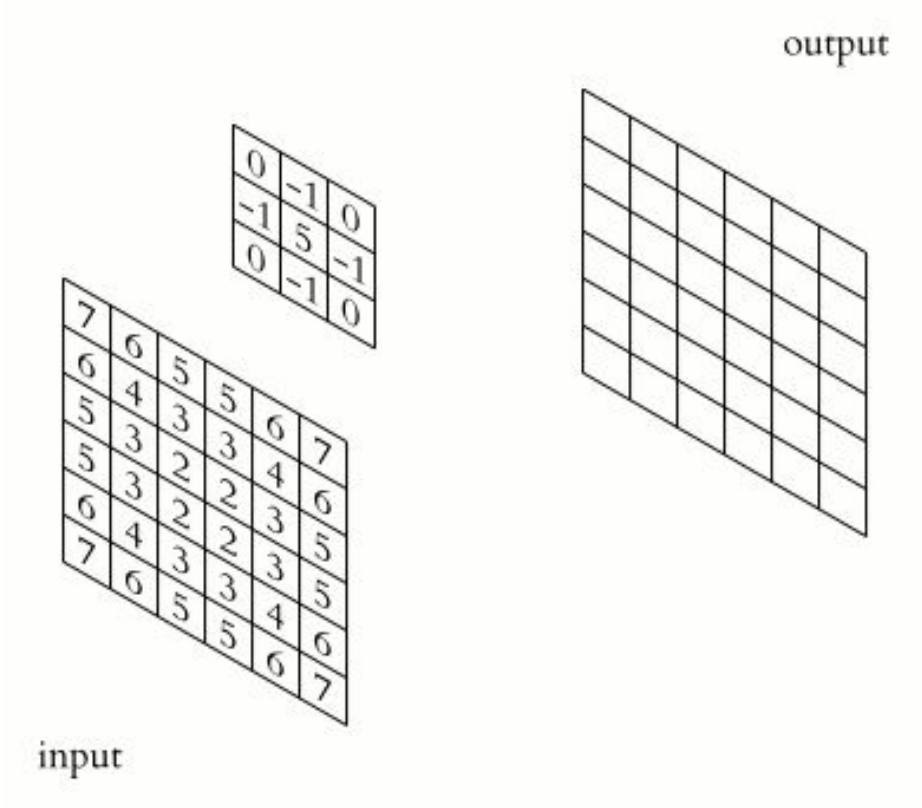
Convolutional neural networks

Code implementation:

```
from torch.nn import functional as F
x = torch.randn(16, 3, 32, 32)
w = torch.randn(64, 3, 5, 5)
F.conv2d(x, w, padding='same').shape
# [Out]: torch.Size([16, 64, 32, 32])
```

Box C.7.1: *Convolution in PyTorch. Note that the channel dimension is – by default – the first one after the batch dimension. The kernel matrix is organized as a (c', c, k, k) tensor. Padding can be specified as an integer or a string ('same' meaning that the output must have the same shape as the input, 'valid' meaning no padding).*

$$H_{ijz} = \sum_{i'=1}^{2k+1} \sum_{j'=1}^{2k+1} \sum_{d=1}^{c} [W]_{i',j',z,d} [X]_{i'+t(i),j'+t(j),d} \qquad \text{(E.7.7)}$$



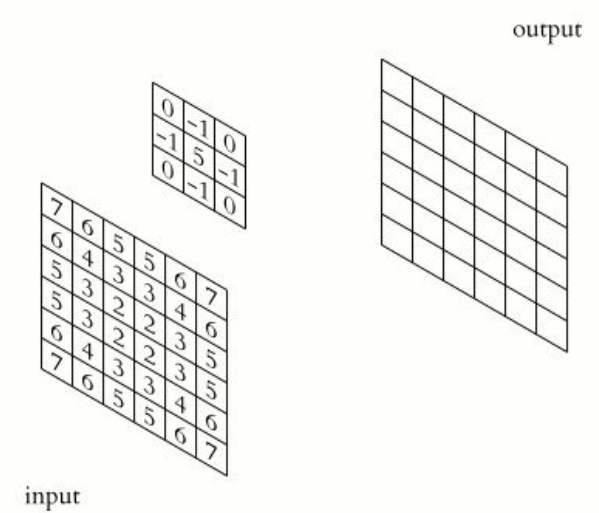
Neural Network Architectures: *a primer*

Convolutional neural networks - Pooling layers

We have seen that convolutional layers extract *local* features, in an equivariant (i.e., translational invariant) manner.

- ✓ symmetrical spatial structure of the input: pixels organised in a fixed size mesh
- ✓ translation invariance (equivariance): sub-features in the image remain the same in different points of the image
- ✓ self-similarity: two or more identical sub-features present in the image can be recognized with a single filter that identifies one of the sub-features
- ? compositionality: a complex feature made of several sub-features can be recognized by identifying only few sub-features
- ✓ locality of the features: to identify a sub-feature it often takes just a few pixels concentrated in a small portion of the image itself

what about compositionality?



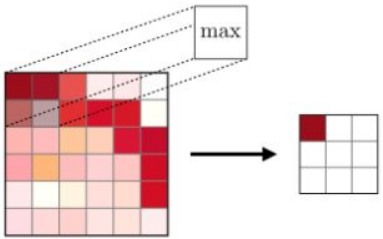
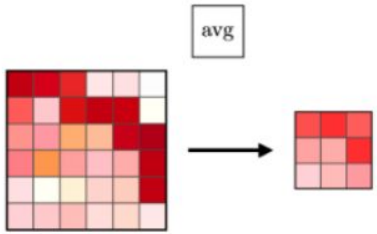
$$H_{ijz} = \sum_{i'=1}^{2k+1} \sum_{j'=1}^{2k+1} \sum_{d=1}^{c} [W]_{i',j',z,d} [X]_{i'+t(i),j'+t(j),d} \qquad \text{(E.7.7)}$$

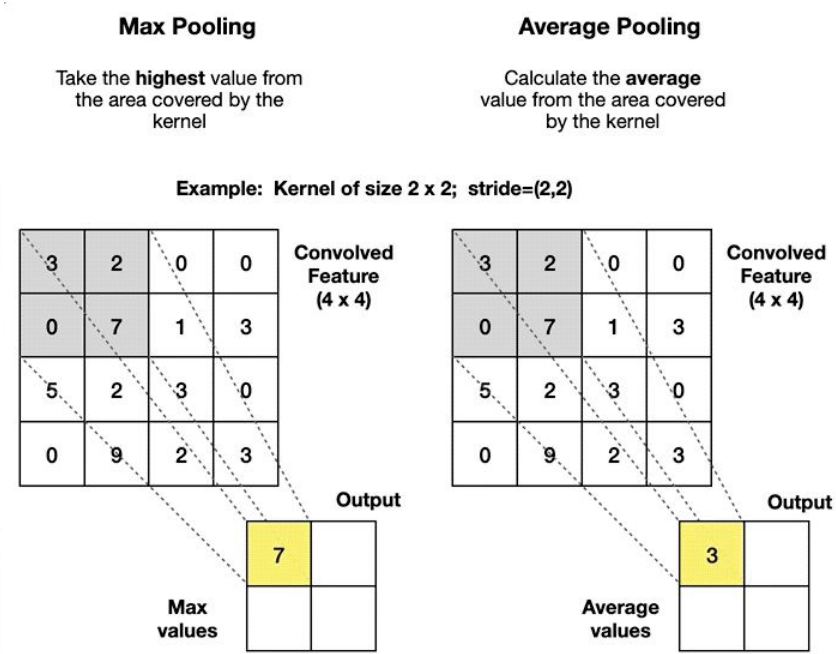
Neural Network Architectures: *a primer*

Convolutional neural networks - Pooling layers

To address compositionality, we need to introduce a new layer - the *Pooling layer* - to allow Convolutional *Networks* (not layers) to have such property

? compositionality: a complex feature made of several sub-features can be recognized by identifying only few sub-features

Type	Max pooling	Average pooling
Purpose	Each pooling operation selects the maximum value of the current view	Each pooling operation averages the values of the current view
Illustration		
Comments	• Preserves detected features • Most commonly used	• Downsamples feature map • Used in LeNet



Neural Network Architectures: *a primer*

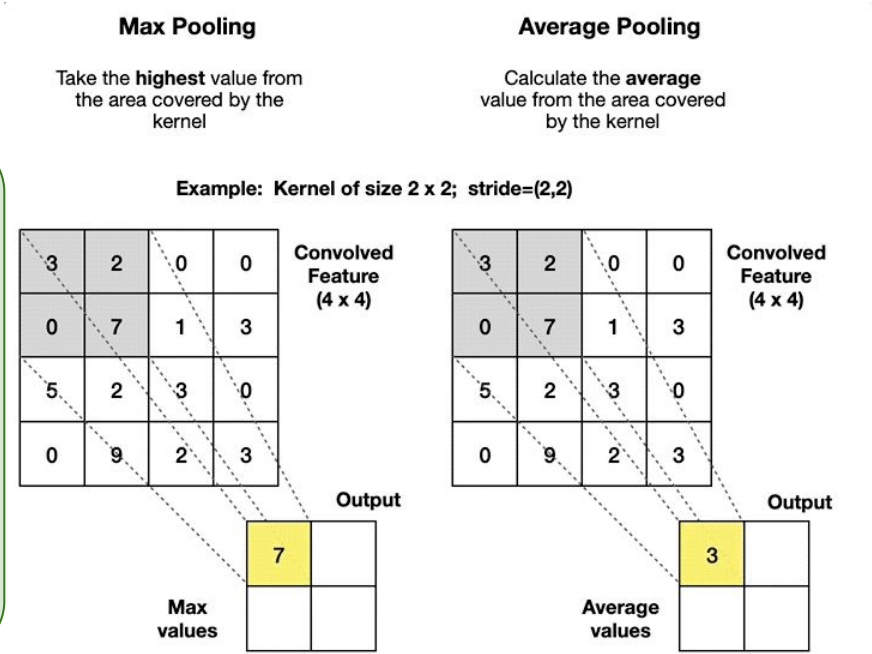
Convolutional neural networks - Pooling layers

To address compositionality, we need to introduce a new layer - the *Pooling layer* - to allow Convolutional *Networks* (not layers) to have such property

? compositionality: a complex feature made of several sub-features can be recognized by identifying only few sub-features

Pooling layers perform a ***downsampling operation***:

- ➢ simplifying the information in output from the convolutional layer (less weights) and making the network less sensitive to small translations of the image;
- ➢ motivated on the fact that once a sub-feature is found, to know the *exact* position is *not* as important as to know the **relative** position wrt the other sub-feature in the image;



Neural Network Architectures: *a primer*

Brief history of ConvNets - LeNet (1989)

The first Convolutional Neural Network was proposed by Y. LeCun in 1989, as LeNet-1. The final version was LeNet-5.

[LeNet-1] LeCun, Y.; Boser, B.; Denker, J. S.; Henderson, D.; Howard, R. E.; Hubbard, W.; Jackel, L. D. (December 1989). "Backpropagation Applied to Handwritten Zip Code Recognition". Neural Computation. 1 (4): 541–551. doi:10.1162/neco.1989.1.4.541

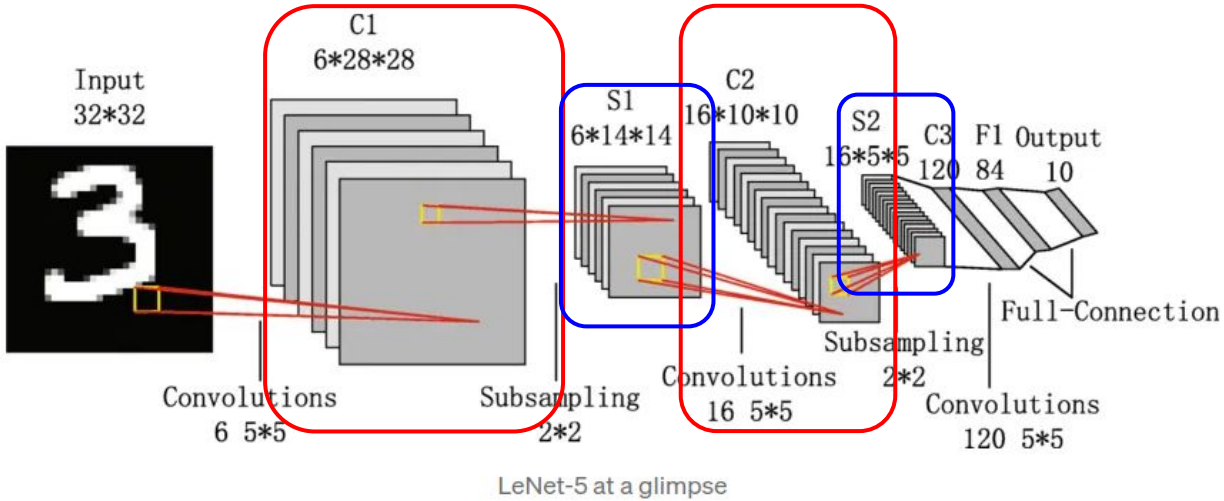
[LeNet-5] Lecun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. (1998). "Gradient-based learning applied to document recognition" (PDF). Proceedings of the IEEE. 86 (11): 2278–2324. doi:10.1109/5.726791

https://www.youtube.com/watch?v=HOoEr40YhrQ

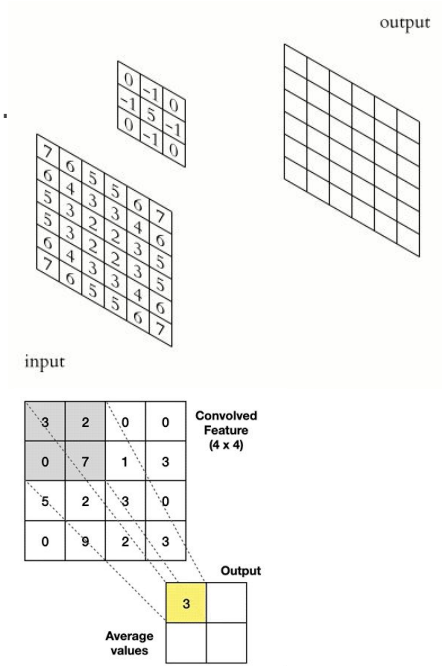
Neural Network Architectures: *a primer*

Brief history of ConvNets - LeNet (1989)

The first Convolutional Neural Network was proposed by Y. LeCun in 1989, as LeNet-1. The final version was LeNet-5.



LeNet-5 at a glimpse



[LeNet-1] LeCun, Y.; Boser, B.; Denker, J. S.; Henderson, D.; Howard, R. E.; Hubbard, W.; Jackel, L. D. (December 1989). "Backpropagation Applied to Handwritten Zip Code Recognition". Neural Computation. 1 (4): 541–551. doi:10.1162/neco.1989.1.4.541
[LeNet-5] Lecun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. (1998). "Gradient-based learning applied to document recognition" (PDF). Proceedings of the IEEE. 86 (11): 2278–2324. doi:10.1109/5.726791

Hands-on

How to train your ~~dragon~~ CNN

Using PyTorch

How to train your CNN

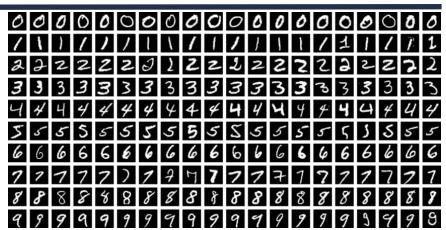
using Pytorch

In the notebook, we will perform 7 steps:

1. Download the data;
2. Perform a little Exploratory Data Analysis;
3. Define the Dataloaders;
4. Define the model (a simple LeNet5);
5. Define the training/evaluation process;
6. Test the trained model
7. [Optional] Use Grad-CAM to *interpret* the predictions;

How to train your CNN

using Pytorch



1. Download the data;
2. Perform a little Exploratory Data Analysis;
3. Define the Dataloaders;
4. Define the model (a simple LeNet5);
5. Define the training/evaluation process;
6. Test the trained model
7. [Optional] Use Grad-CAM to *interpret* the predictions;

It consists of images of *Colon Pathology*, and its 9 classes are:

1. Adipose (ADI),
2. background (BACK),
3. debris (DEB),
4. lymphocytes (LYM),
5. mucus (MUC),
6. smooth muscle (MUS),
7. normal colon mucosa (NORM),
8. cancer-associated stroma (STR),
9. colorectal adenocarcinoma epithelium (TUM).

Facts of **PathMNIST**

Data Modality: Colon Pathology

Task: Multi-Class (9)

Number of Samples: 107,180 (89,996 / 10,004 / 7,180)

Source Data:

Jakob Nikolas Kather, Johannes Krisam, et al., "Predicting survival from colorectal cancer histology slides using deep learning: A retrospective multicenter study," PLOS Medicine, vol. 16, no. 1, pp. 1–22, 01 2019.

License: *CC BY 4.0*

[1] LeCun, Yann and Corinna Cortes. "The mnist database of handwritten digits." (2005).
[2] Jiancheng Yang, Rui Shi, Bingbing Ni. "MedMNIST Classification Decathlon: A Lightweight AutoML Benchmark for Medical Image Analysis". IEEE 18th International Symposium on Biomedical Imaging (ISBI), 2021.
[3] Kather, J. N. et al. Predicting survival from colorectal cancer histology slides using deep learning: A retrospective multicenter study. PLOS Medicine 16, 1–22, https://doi.org/10.1371/journal.pmed.1002730 (2019).

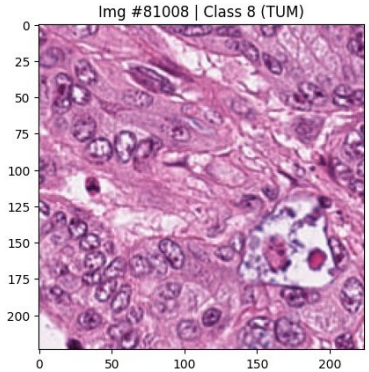
How to train your CNN

using Pytorch

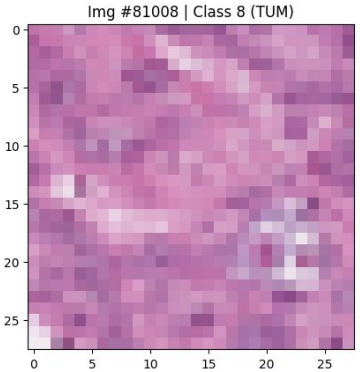
$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

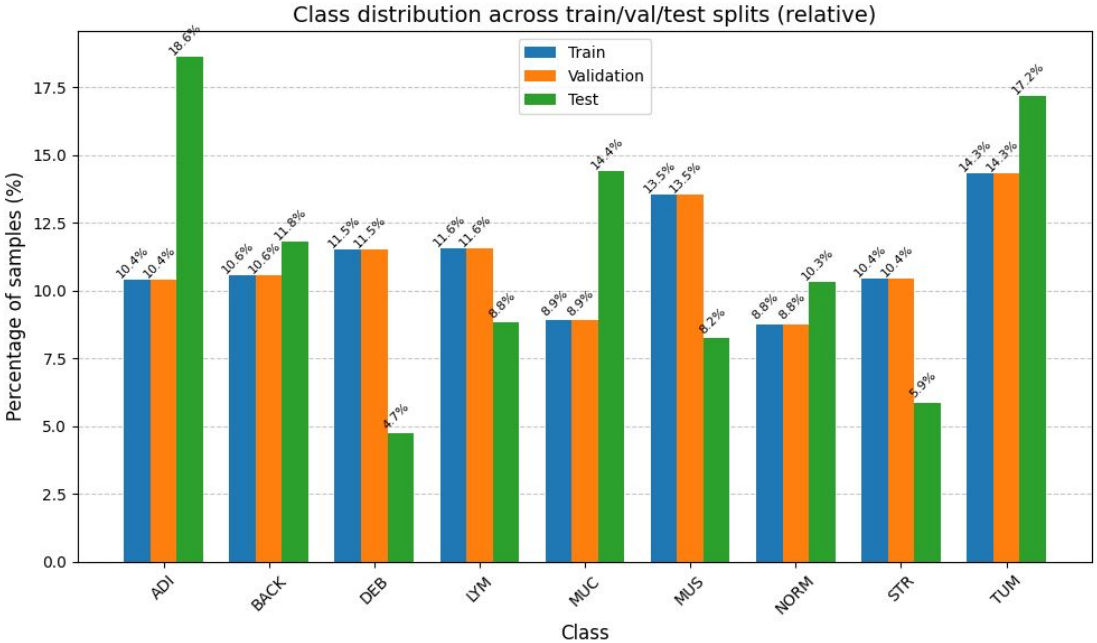
1. Download the data;
2. Perform a little Exploratory Data Analysis;
3. Define the Dataloaders;
4. Define the model (a simple LeNet5);
5. Define the training/evaluation process;
6. Test the trained model
7. [Optional] Use Grad-CAM to *interpret* the predictions;

224 x 224 px



28 x 28 px






```
Class freq   : [0.1  0.11 0.12 0.12 0.09 0.14 0.09 0.1  0.14]
```

```
Class freq   : [1.07 1.05 0.97 0.96 1.25 0.82 1.27 1.06 0.78]
```

How to train your CNN

using Pytorch

1. Download the data;
2. Perform a little Exploratory Data Analysis;
3. Define the Dataloaders;
4. Define the model (a simple LeNet5);
5. Define the training/evaluation process;
6. Test the trained model
7. [Optional] Use Grad-CAM to *interpret* the predictions;

```python
from torch.utils.data import Dataset
from torchvision.transforms import v2 as T2

class TransformDataset(Dataset):
    """Wraps tensor data and applies transforms."""
    def __init__(self, data, targets, transform=None):
        self.data = data          # expects (N, C, H, W) float tensor in [0,1]
        self.targets   = targets
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        x = self.data[idx]
        if self.transform:
            x = self.transform(x)
        return x, self.targets[idx]
```

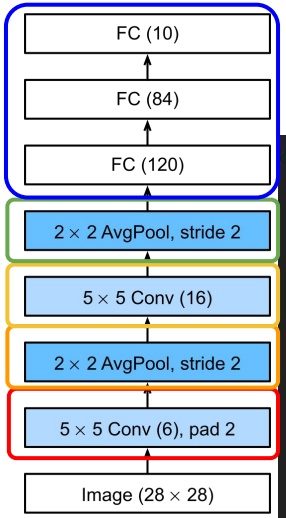
```python
train_transform = T2.Compose([
    T2.Normalize(mean=[.5], std=[.5]),
    T2.RandomHorizontalFlip(p=0.2),
    T2.RandomRotation(7),
])
val_transform = T2.Compose([
    T2.Normalize(mean=[.5], std=[.5])
])
```

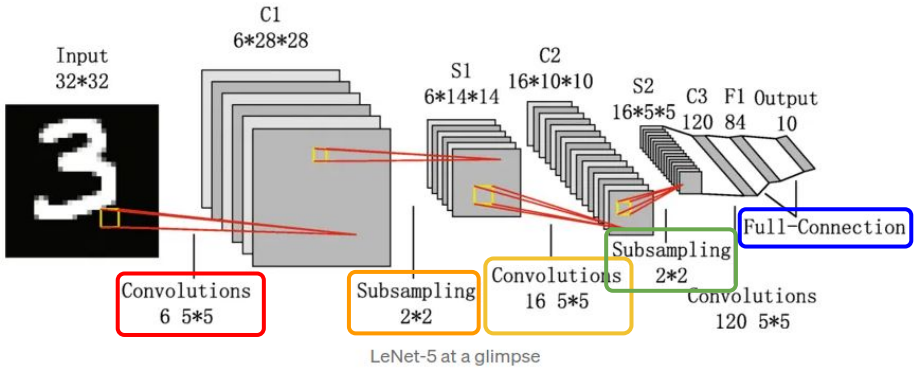
```python
# 2. Datasets (4 lines total)
train_ds = TransformDataset(X_train_t / 255.0, y_train_t, transform=train_transform)
val_ds   = TransformDataset(X_val_t   / 255.0, y_val_t,   transform=val_transform)
test_ds  = TransformDataset(X_test_t  / 255.0, y_test_t,  transform=val_transform)

# 3. Dataloaders (3 lines total)
batch_size = 64 #<==== important! =======
train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True)
val_loader   = DataLoader(val_ds,   batch_size=batch_size, shuffle=False)
test_loader  = DataLoader(test_ds,  batch_size=batch_size, shuffle=False)
```

How to train your CNN

using Pytorch

1. Download the data;
2. Perform a little Exploratory Data Analysis;
3. Define the Dataloaders;
4. Define the model (a simple LeNet5);
5. Define the training/evaluation process;
6. Test the trained model
7. [Optional] Use Grad-CAM to *interpret* the predictions;



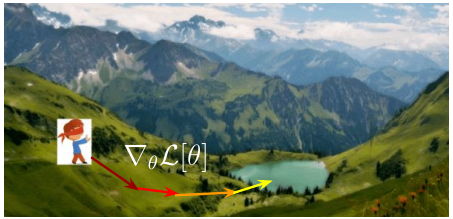


LeNet-5 at a glimpse

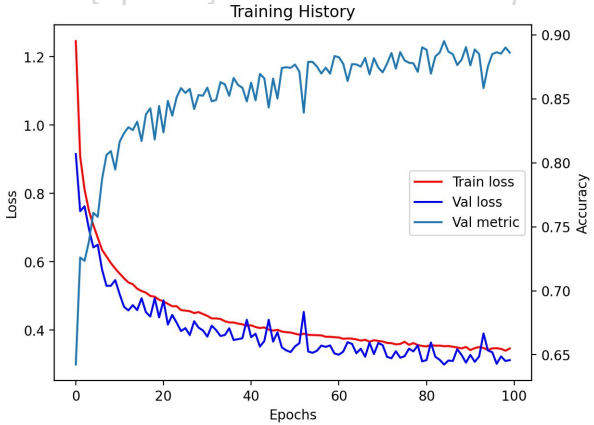
```python
class LeNet(nn.Module):
    def __init__(
        self,
        in_channels: int = 3,
        num_classes: int = 9
    ):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, 6, kernel_size=5, padding=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.fc1   = nn.Linear(16*5*5, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, num_classes)
        self.dropout = nn.Dropout(0.3)
    def forward(self, x):
        '''
        One forward pass through the network.
        Args:
            x: input
        '''
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x
```

How to train your CNN

using Pytorch



1. Download the data;
2. Perform a little Exploratory Data Analysis
3. Define the Dataloaders;
4. Define the model (a simple LeNet5);
5. Define the training/evaluation process;
6. Test the trained model
7. [Optional] Use Grad-CAM to *interpret* the

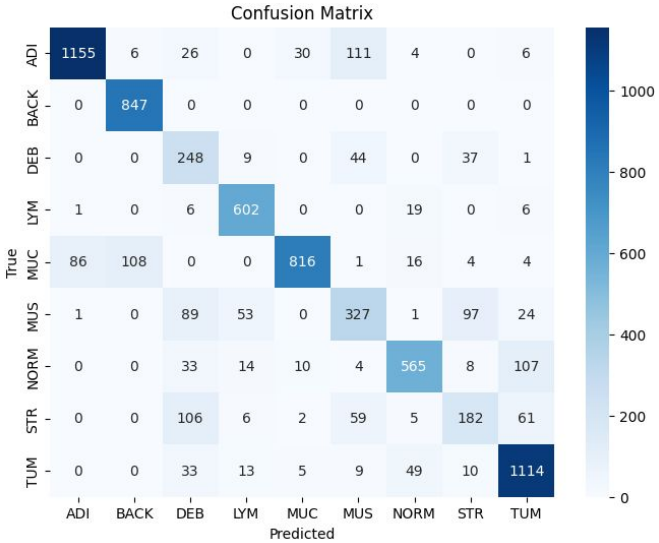


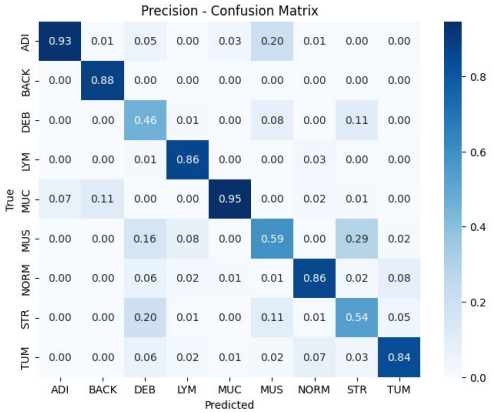

```python
def train(model, device, train_loader, optimizer, epoch, class_weights=None):
    model.train()
    avg_loss = 0
    pbar = tqdm.tqdm(enumerate(train_loader), desc='Train', total=len(train_loader))
    for batch_idx, (data, target) in pbar:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy( output, target, weight=class_weights)
        loss.backward()
        optimizer.step()
        # for data
        avg_loss += loss.item()
    avg_loss /= len(train_loader)
    print(f"Train Epoch: {epoch}; Loss: {avg_loss:.4e}")
    return avg_loss
```

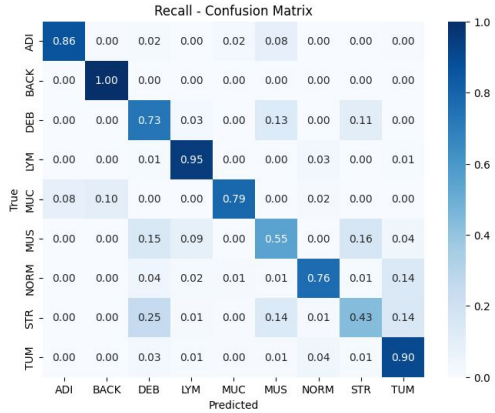
How to train your CNN

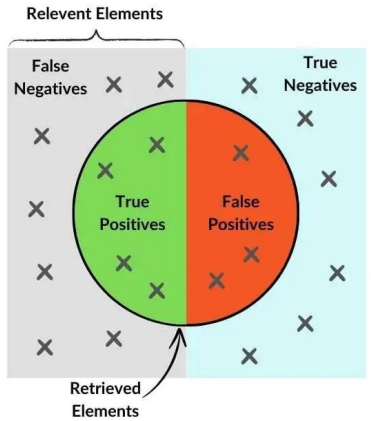
using Pytorch

1. Download the data;
2. Perform a little Exploratory Data Analysis;
3. Define the Dataloaders;
4. Define the model (a simple LeNet5);
5. Define the training/evaluation process;
6. Test the trained model
7. [Optional] Use Grad-CAM to *interpret* the predictions;







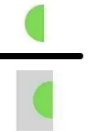


How many retrieved elements are relevant?

Precision =

How many relevant elements are retrieved?

Recall =



How to train your CNN

using Pytorch

1. Download the data;
2. Perform a little Exploratory Data Analysis;
3. Define the Dataloaders;
4. Define the model (a simple LeNet5);
5. Define the training/evaluation process;
6. Test the trained model
7. [Optional] Use Grad-CAM to *interpret* the predictions;

Advanced topics

Interpreting trained CNN with Grad-CAM

Using PyTorch

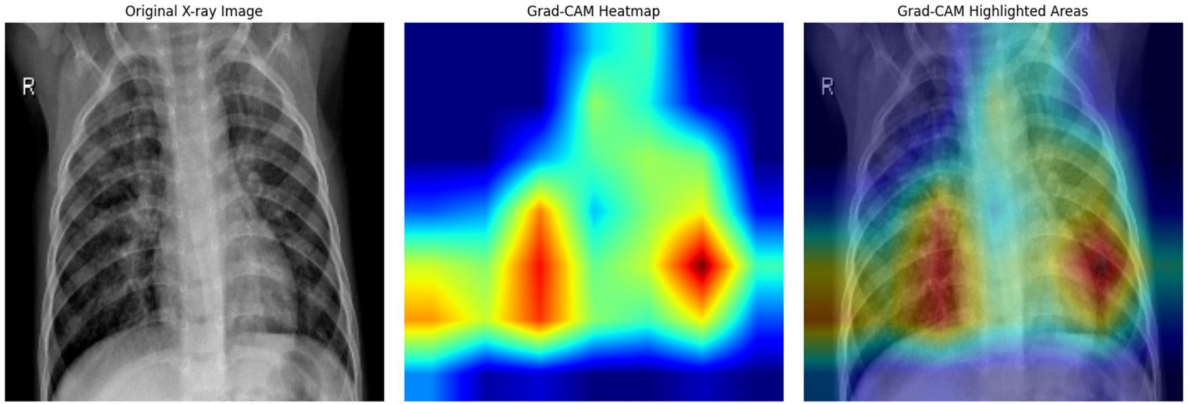
Interpreting CNN predictions with Grad-CAM

Why Grad-CAM? "Where did the CNN look?"

The main takeaway we need to stress is:

$$good\ predictive\ performance \neq clinically\ meaningful\ reasoning$$

Grad-CAM [1] is a tool which may help us *understand* which image regions *supported* the CNN prediction.



Grad-CAM heatmap visualization of chest X-ray images for pneumonia detection [2].

Grad-CAM (*Gradient-weighted Class Activation Mapping*):

- ✓ produces a class-specific localization map
- ✓ highlights regions that support the selected prediction
- ✓ requires no modification nor retraining of the CNN

[1] Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D. (2017). Grad-cam: Visual explanations from deep networks via gradient-based localization. In Proceedings of the IEEE international conference on computer vision (pp. 618-626).
[2] Colin J, Surantha N. Interpretable Deep Learning for Pneumonia Detection Using Chest X-Ray Images. Information. 2025; 16(1):53. https://doi.org/10.3390/info16010053

Interpreting trained CNN with Grad-CAM

How it works, in 6 steps

Grad-CAM is quite simple in its working; It needs 6 steps:

1. Forward pass the input X through the trained CNN f_θ;
2. Pick a class c (*predicted* or *target*); and select its pre-softmax score (logit) $y^c = f_\theta(X)$;
3. Extract the *feature maps* A^k_{ij} from the chosen convolutional layer (usually, the last one);
4. Compute the gradient of the score for the class w.r.t.. the feature maps; $\frac{\partial y^c}{\partial A^k_{ij}}$
5. Global average pool these gradients over pixels to obtain *channel importance weights*

How important is feature channel k for class c?

$$\alpha^c_k = \frac{1}{H \cdot W} \sum_{i,j} \frac{\partial y^c}{\partial A^k_{ij}}$$

6. Compute the weighted combination of the activation maps, apply ReLU, then upsample the resulting map to the input image size.

$$L^c_{\text{Grad-CAM}} = \text{ReLU}\left[\sum_k \alpha^c_k A^k_{ij}\right]$$

gradients tell us WHICH FEATURES matter + feature maps tell us WHERE they occur

The intuition behind the math: at each spatial position of the chosen layer, you are asking "if I increased the activation here, would the class score go up?"

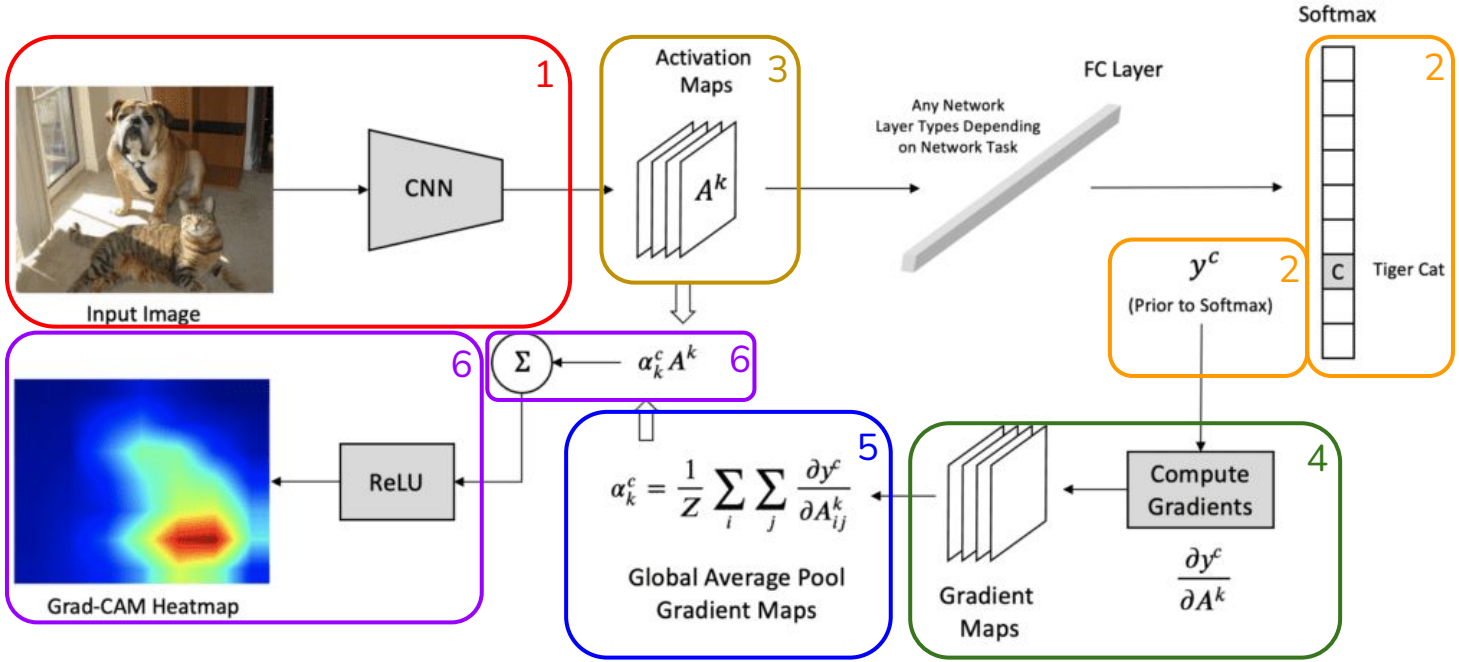
If yes, that position contributes *positively* to the heatmap, weighted by the channel's importance.

If no (or the contribution is negative), the ReLU suppresses it.

[1] Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D. (2017). Grad-cam: Visual explanations from deep networks via gradient-based localization. In Proceedings of the IEEE international conference on computer vision (pp. 618-626).
[2] M. Shaha, https://pages.cs.wisc.edu/~sharonli/courses/cs839_fall2020/slides/presentation6.pdf

Interpreting trained CNN with Grad-CAM

Grad-CAM under the hood



1. Forward the input through the CNN;

2. Pick a class and its *score*;

3. Extract Feature maps:

4. Compute gradient maps;

5. Get the channel weights;

6. Weight, combine, upsample

[1] https://learnopencv.com/intro-to-gradcam/

List of recommended resources

Books

[Scardapane] S. Scardapane, “Alice’s Adventures in a differentiable wonderland”, https://www.sscardapane.it/alice-book/
[Mehta18] Mehta, Pankaj, et al. "A high-bias, low-variance introduction to machine learning for physicists." Physics reports 810 (2019): 1-124. https://arxiv.org/abs/1803.08823
[DLB] I. Goodfellow, Y. Bengio and A. Courville, “Deep Learning” https://www.deeplearningbook.org/ (a.k.a. the *Bible of Deep Learning*)
[d2l] AAVV, “Dive into Deep Learning” https://d2l.ai/
[udl] S. Prince, “Understanding Deep Learning” https://udlbook.github.io/udlbook/
[LBDL] F. Fleuret, “The Little Book of Deep Learning”, https://fleuret.org/francois/lbdl.html
[DLwP] Francois Chollet, “Deep Learning with Python”, https://fchollet.com/

Lectures

[Giagu] S. Giagu, “Ann: Principles And Common Architectures” , slides
[Lizzi] F. Lizzi, “Introduction to CNN, Autoencoders and U-Net” slides
[cs231] Stanford CS231 Course “Convolutional Neural Networks for Visual Recognition” https://cs231n.github.io/

The Abdus Salam International Centre for Theoretical Physics

Thanks for your attention!

Alessandro Bombini
email: bombini@fi.infn.it

INFN

Fondazione ICSC
Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

The Abdus Salam International Centre for Theoretical Physics

Extra slides

Alessandro Bombini
email: bombini@fi.infn.it

INFN

Fondazione ICSC
Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

Neural Network Architectures: *a primer*

A brief introduction to training DNN

You have seen what a DNN is; I just use this slide to rephrase standard introduction in a way will be useful later.

DNNs are *universal function approximator*, i.e. we can approximate any function

$$f : z \in \Omega \subseteq \mathbb{R}^d \mapsto \mathcal{N} \subseteq \mathbb{R}^n$$

with an analytical function $\mathrm{DNN}_\theta : z \mapsto \mathrm{DNN}_\theta(z)$ in a way that

$$\mathrm{DNN}_\theta(z) \sim f(\theta)$$

i.e, we have to find the DNN parameters θ such that a certain *loss function*

$$\mathcal{L}(\theta) = \mathcal{L}\left[\mathrm{DNN}_\theta(z), f(z)\right]$$

is minimised, i.e.

$$\theta^* = \arg\min_\theta \mathcal{L}(\theta)$$

Neural Network Architectures: *a primer*

A brief introduction to training DNN

To achieve that, is commonly employed (a variation of) *gradient descent*, i.e., denoting with θ the DNN parameters,

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}[\mathrm{DNN}(\theta; z)]$$

this is usually facilitated by the fact that $\mathcal{L}[\mathrm{DNN}(z), f(z)]$ is a smooth function, with an explicit form for its derivative. Also, $\mathrm{DNN}(\theta; z)$ is itself a smooth function, arranged somehow in layers, and, using repeatedly the formula for the derivative of a composite function, its derivative in each point can be **analytically explicitly computed**.

For example, a standard MLP is

$$\begin{aligned}\mathrm{DNN}(\theta; z) &= \lambda_1[\theta_1] \circ \cdots \circ \lambda_N[\theta_N](z) \\ &= \lambda_N(\theta_N; \cdots \lambda_1(\theta_1; z))\end{aligned} \qquad \lambda_i(\theta_i; z) = f_i(W \cdot x_i + b)$$

so

$$\nabla_{W_i} \mathrm{DNN}[\theta; z] = \nabla_{W_{i+1}} \mathrm{DNN}[\theta; z] \cdot f_i'(W_i \cdot x_i + b_i) \cdot x_i$$

$$\nabla_{b_i} \mathrm{DNN}[\theta; z] = \nabla_{b_{i+1}} \mathrm{DNN}[\theta; z] \cdot f_i'(W_i \cdot x_i + b_i)$$

Neural Network Architectures: *a primer*

Gradient descent (and its variations)

We can use locality-based methods

how can we give the (almost) blind man directions? by letting him follow the nearby downhill, i.e., the (-) gradient direction

cost function (the mountains)

random starting point (the blind man)

argmin (the lake)

Neural Network Architectures: *a primer*

Gradient descent (and its variations)

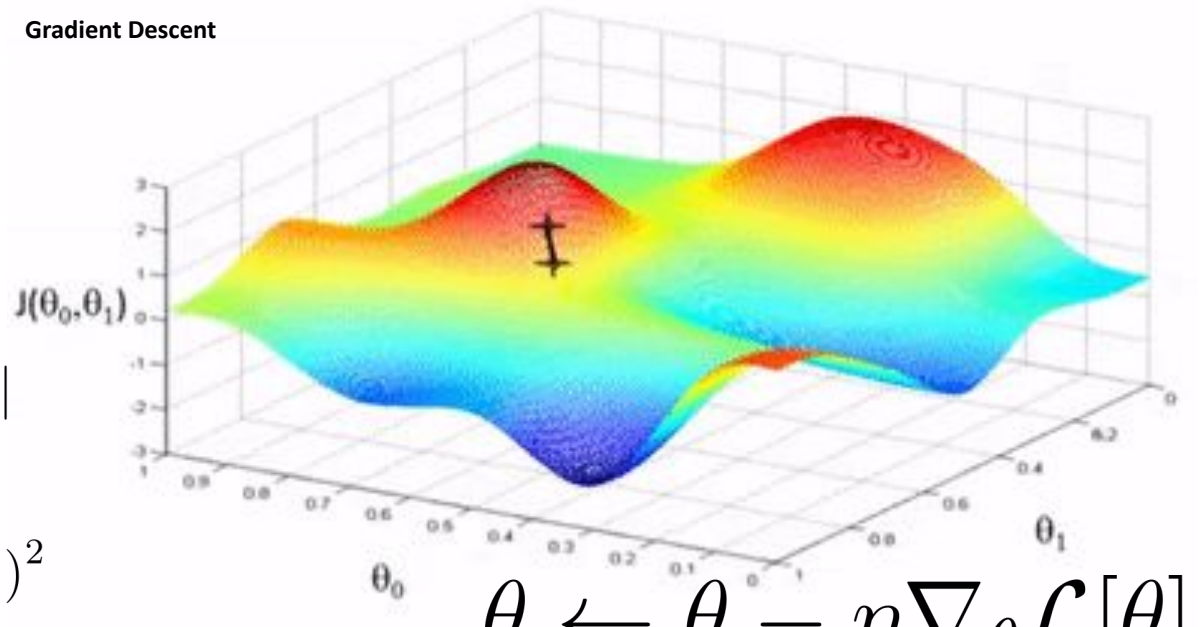
$$f_\theta : X \in \mathbb{R}^N \mapsto \mathbb{R}^m \ni Y$$

We can use locality-based methods

we need a *smooth*, differentiable, easily computable cost function to do this

$$\mathcal{L}_{MAE}[\theta] = \frac{1}{N}\sum_{i=1}^{N} |f_\theta(X_i) - Y_i|$$

$$\mathcal{L}_{MSE}[\theta] = \frac{1}{N}\sum_{i=1}^{N} (f_\theta(X_i) - Y_i)^2$$



$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}[\theta]$$

Neural Network Architectures: *a primer*

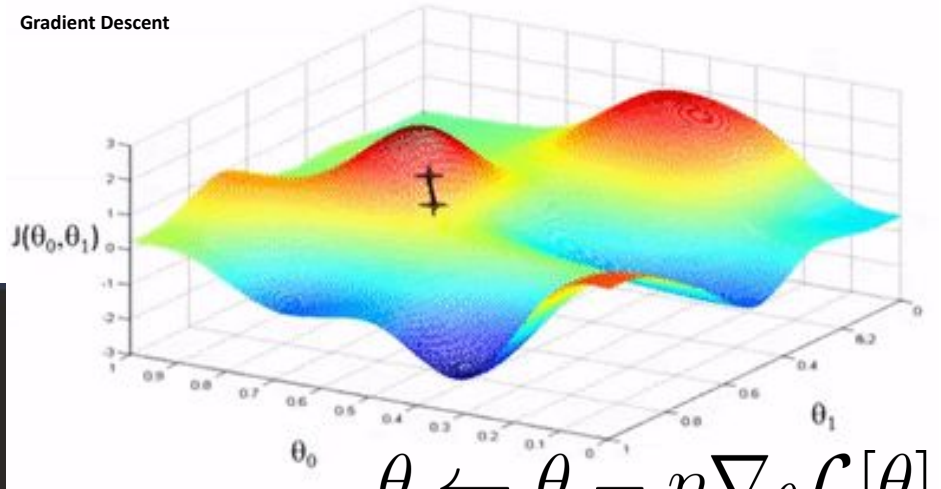
Gradient descent (and its variations)

$$f_\theta : X \in \mathbb{R}^N \mapsto \mathbb{R}^m \ni Y$$

We can use locality-based methods

it is an *iterative* process,
over so-called *epochs*

```
for epoch in range(epochs):
    # 1. Compute prediction
    y_pred = approx_func(x_true)
    # 2. Check prediction
    loss = loss_func(y_pred, y_true)
    # 3. Do Gradient descent
    approx_func.update_params(
        approx_func.params - lr * loss.grad(approx_func.params)
    )
    # 4. Stop iterations at convergece
    if convergece_reached:
        break
```



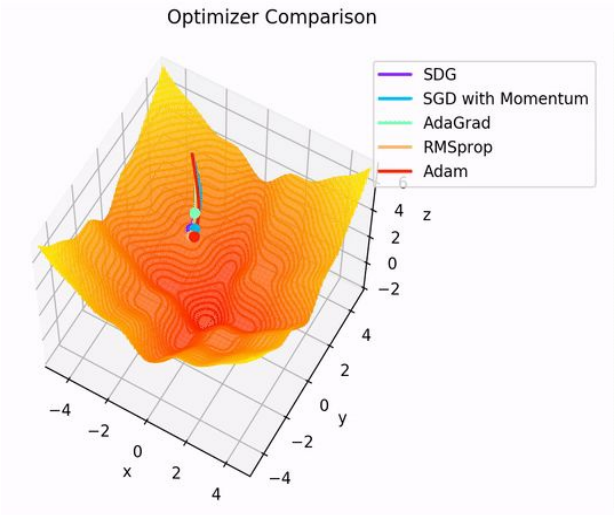
$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}[\theta]$$

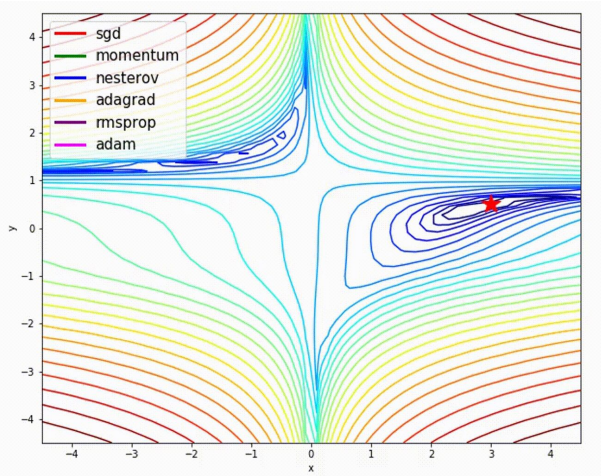
Neural Network Architectures: *a primer*

Mehta, Pankaj, et al. "A high-bias, low-variance introduction to machine learning for physicists." *Physics reports* 810 (2019): 1-124. https://arxiv.org/abs/1803.08823

Other *optimisers*

In practice, other optimisers are usually used; particularly, the ones using *second momentum of the loss*.
The most famous ones are RMSProp [Hinton12] and Adam [Adam].





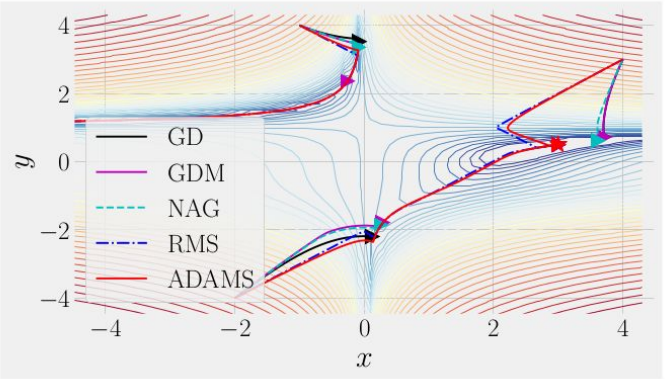


FIG. 9 **Comparison of GD and its generalization for Beale's function**. Trajectories from gradient descent (GD; black line), gradient descent with momentum (GDM; magenta line), NAG (cyan-dashed line), RMSprop (blue dash-dot line), and ADAM (red line) for $N_{\text{steps}} = 10^4$. The learning rate for GD, GDM, NAG is $\eta = 10^{-6}$ and $\eta = 10^{-3}$ for ADAM and RMSprop. $\beta = 0.9$ for RMSprop, $\beta_1 = 0.9$ and $\beta_2 = 0.99$ for ADAM, and $\epsilon = 10^{-8}$ for both methods. Please see corresponding notebook for details.

https://physics.bu.edu/~pankajm/MLnotebooks.html this notebook:

[Hinton12] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
[Adam] Kingma, Diederik P. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

Neural Network Architectures: *a primer*

How to use optimisers in PyTorch

```python
import torch

# instantiate model
model = ...

optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
# or
optimizer = torch.optim.adam(model.parameters(), lr=0.01, betas=(0.9, 0.999))
# or
optimizer = torch.optim.RMSprop(model.parameters(), lr=0.01, alpha=0.99)

# usage
for input, target in dataset:
    # reset optimiser state
    optimizer.zero_grad()
    # compute forward pass
    output = model(input)
    # compute loss
    loss = loss_fn(output, target)
    # compute backward pass
    loss.backward()
    # update model params
    optimizer.step()
```

https://pytorch.org/docs/stable/optim.html
https://github.com/pytorch/tutorials/blob/main/beginner_source/basics/optimization_tutorial.py