Deep learning

CNN and supervised learning

March 2019 @ TELECOM Nancy

T. Bagrel

Introduction

Interesting NN architectures (supervised)

Convolutional **N**eural **N**etwork (CNN)

- Supervised :(
- Most common architecture for image processing
- Position-invariant feature extraction (with shared weights)
- Very efficient

Introduction

Interesting NN architectures (unsupervised)

Restricted **B**oltzmann **M**achine (RBM)

- Unsupervised!
- General feature extraction
- Basic block of **D**eep **B**elief **N**etworks

Auto**e**ncoders (AE)

- Unsupervised!
- Can be used for denoising
- Can be used for general feature extraction

Autoencoders

Structure and principle

Principle

- Simple : reproduce the input at the output
- How : pass the input (image) through the NN, compute error from diff(I, O), adjust weights with GD

Feature extraction

Bottleneck : "center" of the NN where there is the lowest number of neurons. Activation in the "bottleneck" for a given input (image) represents its compressed form.

Autoencoders

Denoising

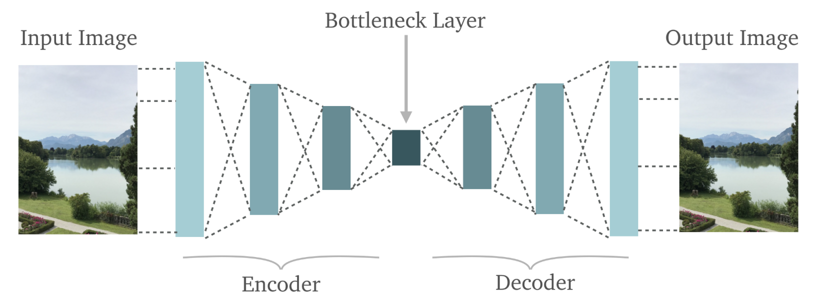


FIGURE – Autoencoder with feature extraction purpose

Denoising

Denoising can be achieved with roughly the same architecture

CNN Components

Gradient Descent

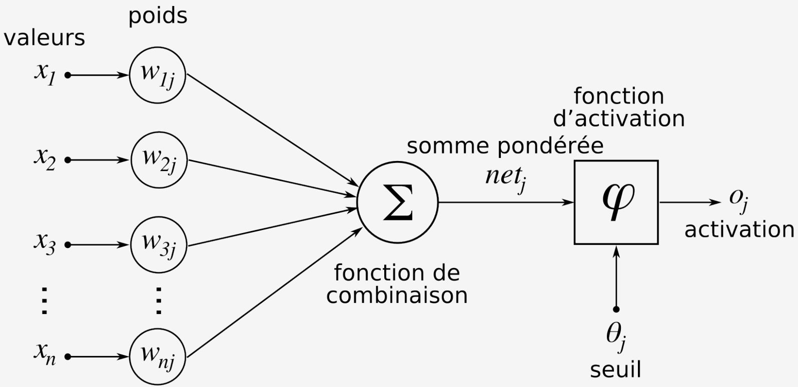


FIGURE – Neuron (perceptron) schema

CNN Components

Gradient Descent

Problem solved by GD

In which direction do we need to update our weights to minimize the error?

Last layer neuron n situation

E : error, L : loss function, e : expected value, o : obtained value, φ : activation function, $x_{n,\cdot}$: inputs for the last layer neuron, $w_{n,\cdot}$: weights for the last layer neuron

$$\mathrm{E} = \mathrm{L}(e, o) \tag{1}$$

$$= \mathrm{L}\left(e, \quad \varphi\left(\sum_k x_{n,k} w_{n,k}\right)\right) \tag{2}$$

CNN Components

Gradient Descent

What we need to know

- loss function $\mathrm{L}(e, o)$ and $\dfrac{\partial \mathrm{L}}{\partial o}$
- activation function $\varphi(r)$ and $\dfrac{\mathrm{d}\varphi}{\mathrm{d}r}$

What we can compute

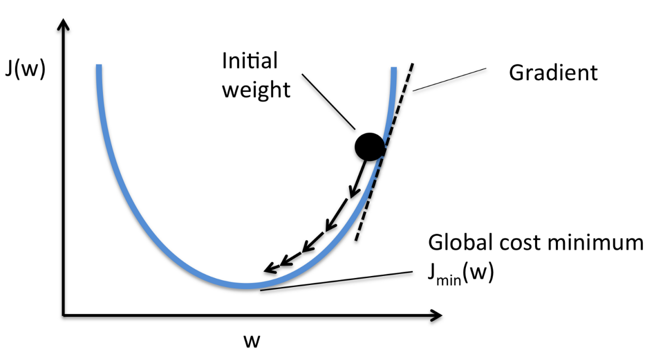
With the chain rule, we can compute $\dfrac{\partial \mathrm{E}}{\partial w_{n,k}}(e, o) \quad (\forall k)$

CNN Components

Gradient descent and adjustment made

Weights adjustment (α : learning rate)

$$w_{n,k} \longleftarrow w_{n,k} - \alpha \cdot \frac{\partial \mathrm{E}}{\partial w_{n,k}}(e, o)$$



CNN Components

Learning rate and advanced adjustment methods

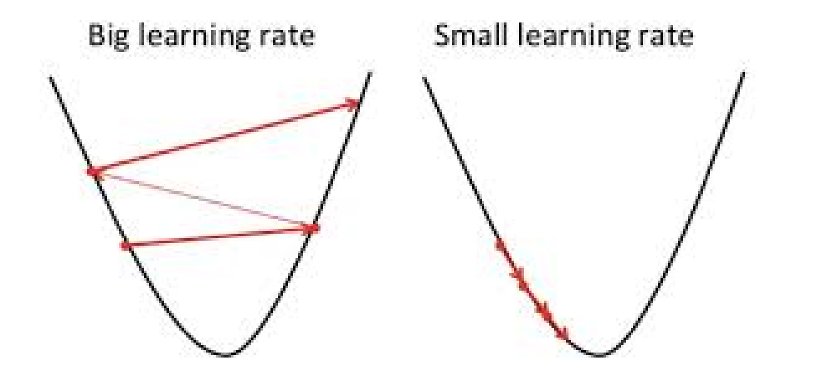


FIGURE – Importance of an appropriate learning rate

CNN Components

Advanced adjustment methods

Called optimizers in TensorFlow

Roles

- Try to prevent staying in a non-global local minima during learning
- Examples : Nesterov Momentum, Adam...

Used in my NNs

I use Nesterov's momentum :
`optimizers.SGD(nesterov=True, momentum=M)`
because it's recommended on most tutorials

CNN Components

GD vs SGD vs BGD

Each one is a different manner to apply Gradient Descent

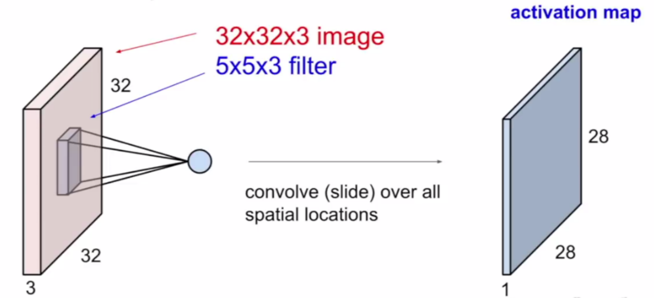
Differences

- (basic) Gradient Descent : weights are updated after every epoch (whole dataset pass) $\rightarrow$ very costly but more stable than SGD!
- Stochastic Gradient Descent : weights are updated after every sample $\rightarrow$ very fast!
- Batch Gradient Descent : weights are updated after N samples ($10 \leqslant \mathrm{N} \leqslant 100$: batch size) $\rightarrow$ best of both worlds!

CNN Components

Convolutional layer

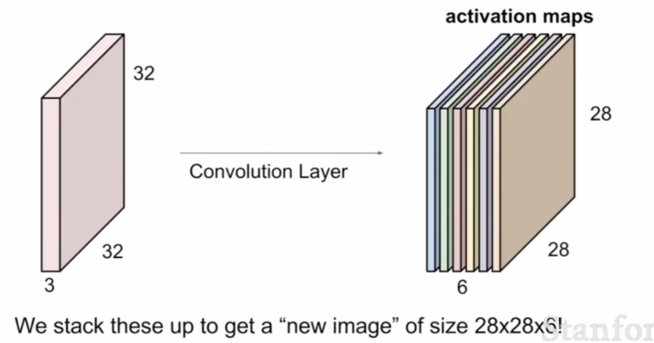
The (3D) dot product between the filter weights and an input image zone of the same size/shape gives an activation value. Doing this operation for all possible image zones (with **overlap**) gives the activation map for this filter.
It explains the position invariance of feature extraction for CNNs!



CNN Components

Convolutional layer

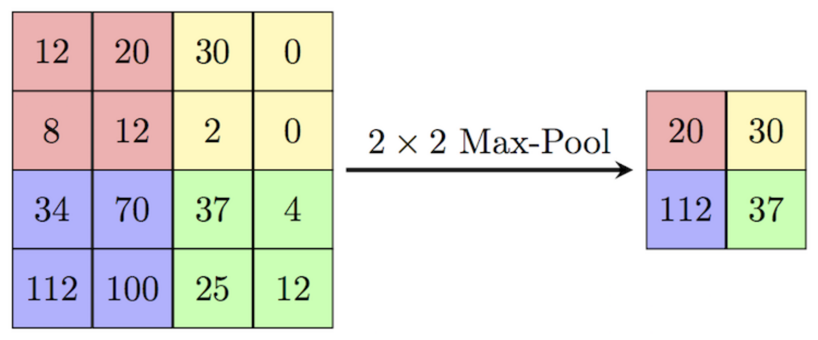
A convolutional layer is composed of several filters, each one with its own weights. Hence, the output of the convolutional layer is a stack of activation maps (one for each filter)



CNN Components

Pooling layer

A pooling layer is used to downsize/down-sample its input. All the values in a zone of the input volume results in one value in the output volume, by application of a simple function. Input zones **must not overlap** here!
A very common example is the *max pooling* 2 x 2 :



CNN Components

Output layer for classification

The output layer is used to extract classification information from a final dense/convolutional/pooling layer.

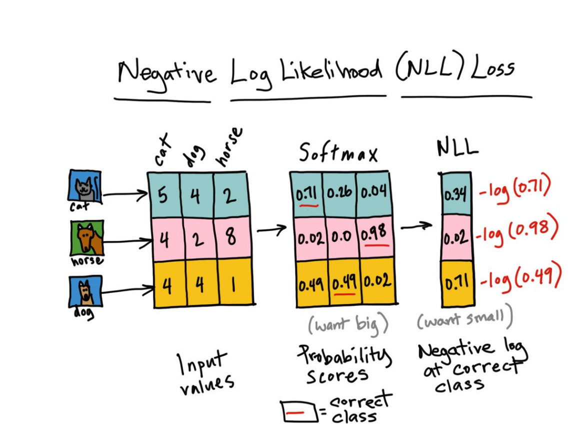
Types

- For binary classification : Dense layer with sigmoid activation function and 1 neuron (1 output value)
- For multiclass classification : Dense layer with softmax activation function and $Nb_{classes}$ neurons, followed by a arg_max$(\cdots)$ function

CNN Components

Loss function for classification

Cross entropy (aka. Negative Log Likelihood) is the loss function used in classification, which works well with the sigmoid or softmax activation functions (output layer)



CNN Components

CNN schema for multiclass

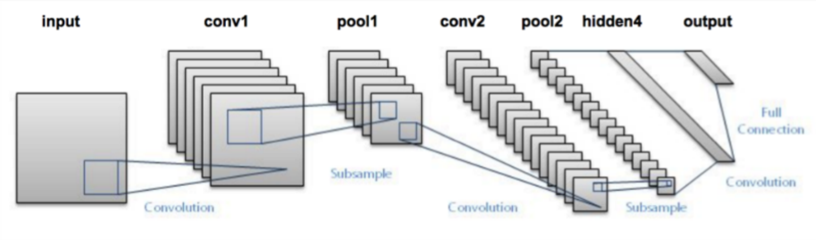


FIGURE – CNN architecture

Medical imaging

Availability

The issue

There are medical image datasets available online, but

- High resolution
- Difficult to find the differencies or the good class for an untrained eye
- File format and dataset organisation is not always normalized

Examples

- List of public datasets : `bit.ly/2EOziVp`, `bit.ly/2u0Da0Q`
- Datasets : `bit.ly/2F34jqb` (cancer cells), `bit.ly/2u99w9L` (neuroimaging), `bit.ly/2cGnNQL` (Alzheimer)...

Medical imaging

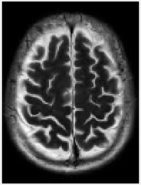
What to do so?

Requirements

- Large dataset
- Low-res images (max. 256 x 256)
- Something which looks like medical images

Idea

Generate brain 2D images and add random “tumors” on them



brain_tumor_factory

Let's get our hands dirty!

How to generate?

Home-made C program : `brain_tumor_factory`

- take a single base image
- add tumor(s) randomly depending on the chosen scenario
- alter the resulting image, to ensure that each output image is a bit unique

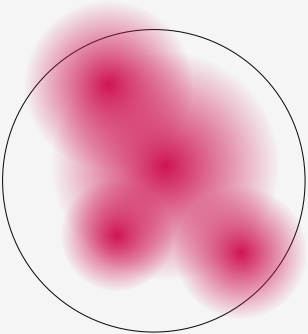
Program size : $\simeq$ 500 LOC
Output : $\simeq$ 1000 images/s
Image size : 96 x 128 x 1 (grayscale)

brain_tumor_factory

Creating tumors : `tumor` function

Instructions

Take a cirle of radius R_{area}, and put $n \in [\![n_{\text{min}}, n_{\text{max}}]\!]$ “soft discs” of radius $r_i \in [r_{\text{min}}, r_{\text{max}}]$ with their center placed randomly inside the chosen area



brain_tumor_factory

Altering the resulting image : alter_full function

Alterations

- shrink : shrink the image on x-axis and/or y-axis
- move : shift the image on x-axis and/or y-axis
- (ghost_blur $^{0.1\%}$) : simulate motion blur on a shot
- alter_contrast : ±[brightness range], with custom "brightness center"
- alter_shade : global flat ±[brightness]
- alter_rdc : per-pixel flat random ±[brightness]

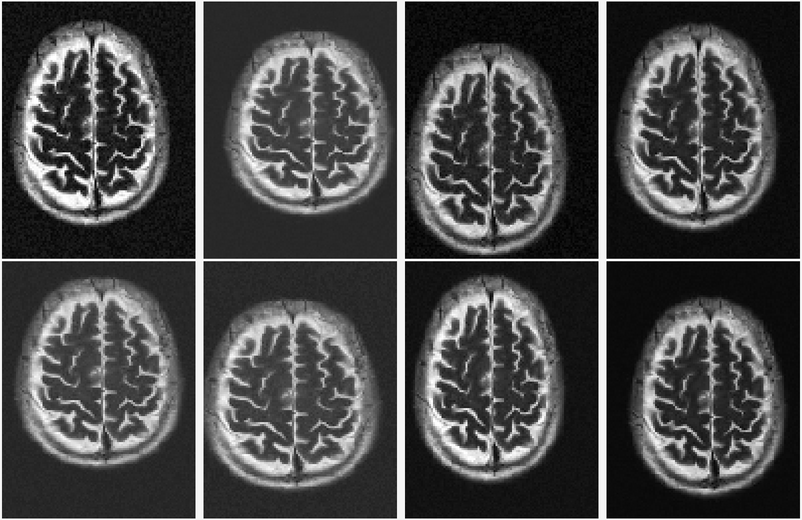
For each image, alterations parameters are random numbers taken from a uniform or normal distribution

brain_tumor_factory

Base image

brain_tumor_factory

Altered images



`brain_tumor_factory`

Syndrome A

Description

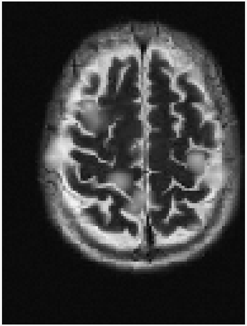
- $[\![1, 2]\!]$ big tumor(s)
- $[\![0, 2]\!]$ medium tumor(s)

brain_tumor_factory

Syndrome B

Description

- $[\![4, 8]\!]$ medium tumor(s)
- $[\![0, 3]\!]$ small/tiny tumor(s)



brain_tumor_factory

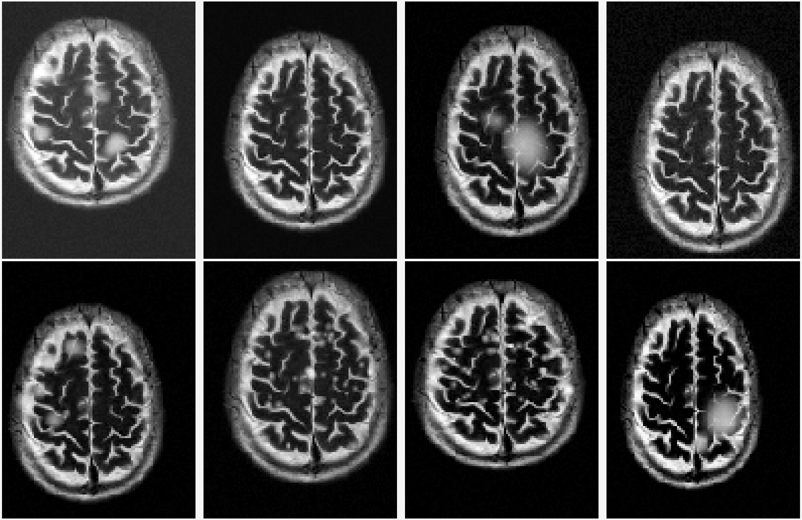
Syndrome C

Description

- only $[\![20, 48]\!]$ small/tiny tumor(s)

brain_tumor_factory

Guess game : EASY mode



brain_tumor_factory

Guess game : HARD mode

Observation

It turns out that it's kinda easy to find the correct syndrome type, isn't it ?
Let's try the HARD mode then (more realistic) 😈

HARD mode specifications

- Tumor opacity greatly reduced
- Tumor size reduced a bit

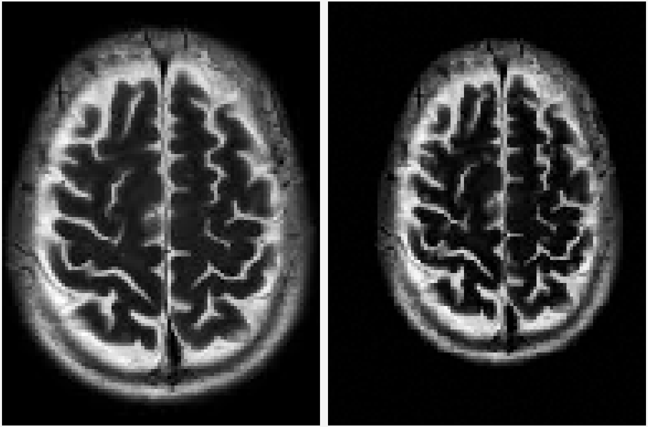
The base image will always be placed on the LEFT of the image whose type must be determined

brain_tumor_factory

Guess game : HARD mode

brain_tumor_factory

Guess game : HARD mode



Answer

Syndrome C

brain_tumor_factory

Guess game : HARD mode

brain_tumor_factory

Guess game : HARD mode

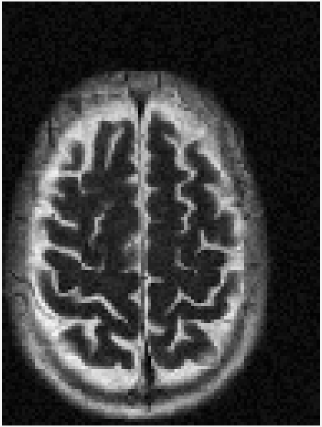
 

Answer

Syndrome A

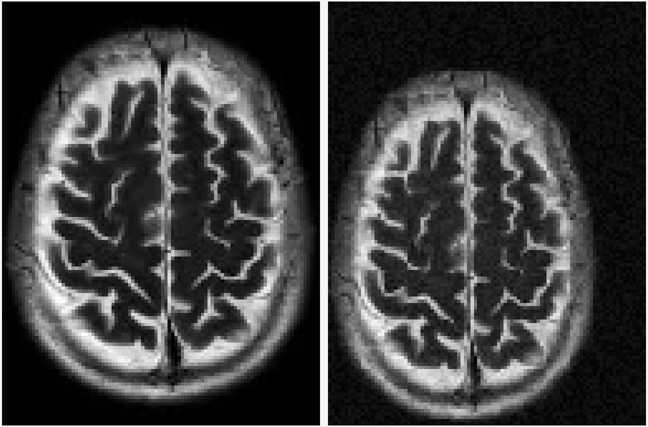
brain_tumor_factory

Guess game : HARD mode



brain_tumor_factory

Guess game : HARD mode



Answer

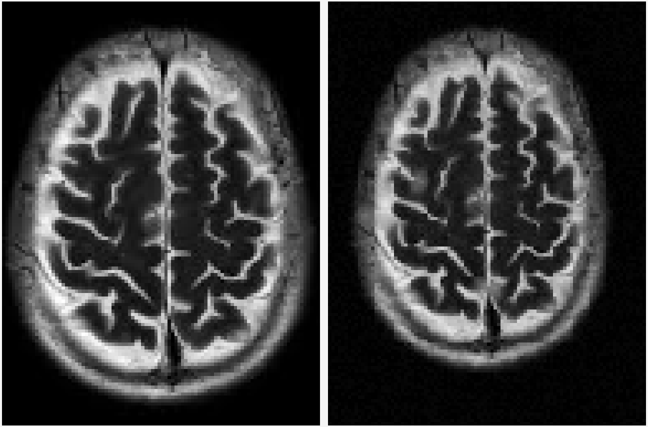
Z (healthy patient)

brain_tumor_factory

Guess game : HARD mode

brain_tumor_factory

Guess game : HARD mode

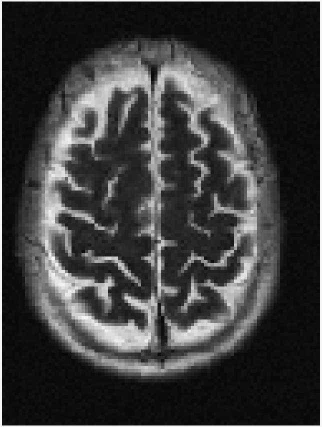


Answer

Syndrome B

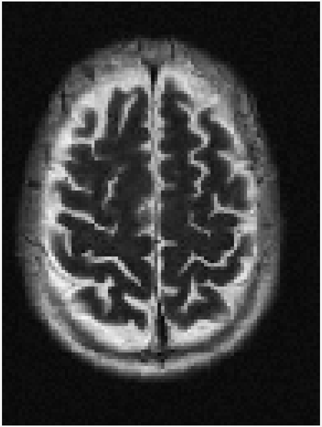
brain_tumor_factory

Guess game : HARD mode



brain_tumor_factory

Guess game : HARD mode



Answer

Z (healthy patient)

brain_tumor_factory

Guess game : HARD mode


brain_tumor_factory

Guess game : HARD mode


Answer

Syndrome C

brain_tumor_factory

Guess game : HARD mode

Observation

Not so easy this time? 😈

Let's see what deep learning can do

brain_tumor_nn

Description

Framework

- DL4j / ScalNet (Scala)
- not really fast (afterwards)
- Keras-like API

Architecture

- 3 `Conv2D` (`MaxPooling2D` inbetween) : 16 64 256
- `Dense` 1024
- GD (BGD) with Nesterov Momentum
- (adapted from MNIST NN examples)

brain_tumor_nn

Description

Issue (after 2 hours learning)

- Accuracy blocked at 0.25
- loss wasn't decreasing
- a single type predicted everytime

Why (afterwards)

- I didn't wait long enough
- 3 Conv layer NN + slow framework didn't help much neither
- in addition, learning on mobile CPU (i7) is still slow
- (but afterwards) architecture wasn't really the issue