Learn Computer Vision with CNN, TensorFlow, and PyTorch

Master Object Detection from basics to advanced

Intro

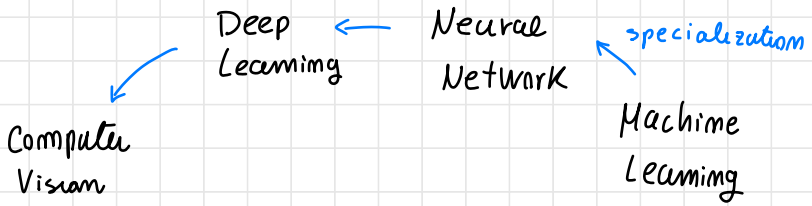
+ Python

set-up

COURSE INTRO

- Computer Vision is good way to learn programming + AI
  ⇓
  its Visual and Intuitive

- Computer Vision has constant demanding and improvements



⇓
specialization of deep learning...

- from Python the couse will focus on OpenCV, Deep Learning theory + algorithms and applications.

I
1. Python (recap)
2. Deep Learning Foundations (ANN from scratch, CNN, backpropagation...)
3. OpenCV (swiss army knife) → (image processing, transformation)

II
4. PyTorch (how models learn + optimize)
5. visualize CNN's (architecture view)
6. Image classification (build AI for object recognition + SOTA)
7. Data augmentation (flipping etc to enhance dataset)

III
8. Object Detection (find obj in images with YOLO + RCNN...)
9. Image segmentation (YOLO v11, other SOTA architecture)
10. Transformers (heart of NLP) and PROJECTS...

- Theory + Hands-on (PyTorch / Keras)
- SOTA Frameworks + Architectures

PYTHON PREREQUISITES (RECAP...)

Anaconda Installation

Open-source tool for sourcing, building and deploying DL tools
- best tool to work with multiple packages and environments

↑

easy managment of packages + environment
foundamental to work with many python versions... envs.

- Installation

On windows, remember to check "Add Anaconda 3 to my PATH..."
(important to create environment variables to properly run conda commands)

↓

you have Anaconda Navigatn GUI
providing many frameworks ready to use!
+ many venv created by default

+ Install vscode to set-up programming IDE

Using VSCode

IDE that allow easily to create envs and work with python code

1) create New environments... → using python 3.12 (Newest version)

↓

necessary in new projects to properly manage multiple libraries...
New features gets included always and maintenance to avoid conflict is required...

conda create -p <venv_name> python==<version>

2) activate environment: conda activate <venv_name>

3) install ipykernel to use jupyter notebooks in Vs code (with pip)

provide KERNEL to Notebook to run python code...

4) create a requirements.txt ... keep a note on required packages, responsible for future deployment

in future this will be responsible to install all requirements

Python Basics, Syntax

basics of SYNTAX: set of rules that defines symbols to program
= correct code arrangement

syntax errors are related to variables bad usage or other

SEMANTIC: meaning / interpretation → how to arrange to do a specific task... what it is supposed to do!

- Python delimit block of code with Indentation → required for code structure and any operation of flow execution (differently from other programming languages that rely on {} brackets)

...

coding practice with Jupyther Notebook on the main python concepts

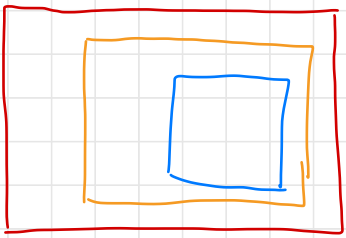
Data types, Data structures, functions, control flow, exception handling, modules (libraries), object oriented programming, generators, iterators, magic methods / decorators...
+ Numpy, Pandas and Logging

(Notes on the Jupyter Notebooks python_recap)

Deep Learning Theory

INTRODUCTION TO DEEP LEARNING

brief idea of what is DL, and how it is related to AI and ML



Artificial Intelligence (AI): create app that performs their own tasks (such as Netflix recom., self-driving car, etc.)

Machine Learning (ML)
subset of AI that provides tools to analyze, visualize data + forecast and prediction
→ supervised
↳ unsupervised

Deep Learning (DL)
subset of ML..
main task is to mimic the human brain → make a machine learn as humans.
⇓
Neural Network (multilayered) (focus of this course)
↙ supervised ↘ unsupervised

Deep Learning

1) Artificial Neural Networks (ANNs) → Classification, Regression ← main problems in supervised ML

2) Convolutional Neural Networks (CNNs) → classification
Input data: images/video frame
↖ CNN are optimal with this inputs

+ RCNN, Masked RCNN, YOLOvX
help in object detection/segmentation
↖ computer vision: object detection

3) Recurrent Neural Network (RNN) → for NLP (Natural Language Processing) tasks
Input data: text/time-series data (forecast etc..)
(sequence of data with dependency)

+ Word Embedding, LSTM RNN, GRU RNN, Bidirectional LSTM RNN, Encoder/Decoder

+Transformers, BERT... multiple use cases

FRAMEWORKS: • TensorFlow (made as opensource by Google brain) in end-to-end projects.

- big research is going on in NLP
- foundations are in ANN, CNN, RNN

Why DL is becoming so popular?

back in 2005... Facebook was launched ... then youtube, Whatsapp
... 2012 ... ↓ many online posts ↓ social media platform that created huge amount of DATA

⇓

DATA generated EXPONENTIALLY
how to store those Data effecently?
↳ {images, videos, texts...}
⇓
to be easily accessible
BIG DATA (Hadoop, Database...) → structured data
↘ unstructured data

⇓
2011 Big data frameworks created. store data and access effecently

2011-2012 DL starts to become more popular because of

1) HW requirements. GPU's cost decreasing
foundamental for model training
⇓ ↓
Train models possible NVIDIA GPU price decrease + Cloude GPU

2) Huge amount of Data generated → DL models perform well with huge data.

Using data in a smart way? → make product more similar to users

e.g. Netflix movie streaming platform
- how to keep users engaged? → based on Data available

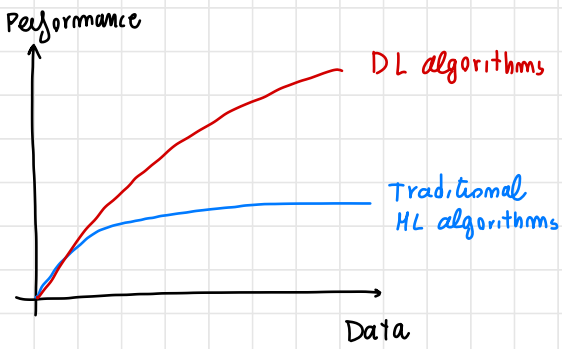
⇓

based on what I watch, it suggest similar product. ← Recommendation system

↑ my interaction with the platform is stored, and it is used to create efficient models ⇒ increase revenue

⇑

BIGGER DATA = more performative DL models



As Data increases, ML reach a limit, while DL continue to increase accuracy performance with amount of Data

3) DL has been used in many domains

↓

1) Medical
2) E-commerce
3) Retails
4) Marketing
...

extensive application thanks to the different techniques / models architecture

4) Open-source Frameworks for DL

↓

TensorFlow PyTorch

TensorFlow open-sourced by Google brain team

PyTorch open-sourced by Facebook

} CNN, ANN, RNN .. implementation

⇓

community size increase! ⇒ more research in DL thanks to an active community

↑

using Big Data with this new algorithms to make more revenue. Automating tasks impactfully

... Next → how DL works in a Neural Network model

DEEP LEARNING CONCEPTS

{ ANN, LOSS FUNCTIONS, ACTIVATION FUNC., CNN }

PERCEPTRON

ARTIFICIAL NEURON / NN UNIT ← this is the basic unit of ANN

~ simple layered NN

What are its PROs/CONs ... Why we move to MLP (Multi Layered Perceptron) MLNN, ANN

- Input Layer
- Hidden Layer
- Weights
- Activation Function

Example Perceptron (simple layer NN)

can be used for solving BINARY CLASSIFIER to solve Dataset problem of binary classification

DATASET

IQ	N° of study hours	% pass/Fail
95	3	0
110	4	1
100	5	1

three rows of DATASET how to use it in a PERCEPTRON and train it?

PERCEPTRON has the following architecture

↳ ...

DATASET

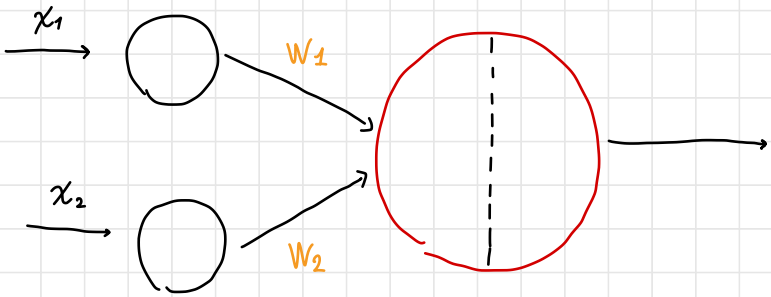
x_1 IQ	x_2 N° of study hours	y % pass/Fail
95	3	0
110	4	1
100	5	1

simple NN limit. Just one Neuron. (simple layer NN)

INPUT LAYER (same as input var.)

HIDDEN LAYER 1

OUTPUT



Neuron

[it performs two tasks and return an output]

you don't count I layer as separate layer

⇑

When input goes through Neuron, it is connected using weights. (W_1, W_2)

~ similarly to how our main neurons process signals from environment and we responds... Neurons process information from data

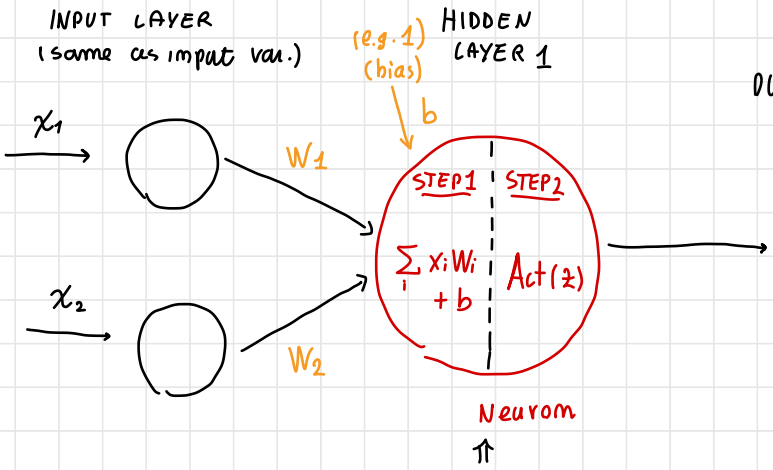
- IF something important occurs there are HIGH WEIGHTS on some part of the processing pipeline...

↓

and our neurons are well trained to handle data and responds weights are high is critical things occurs.

⇑

ENTIRE NN mechanism consists in a good training of the weights W_i to identify data information, possible by TRAINING!



this hidden layer consists of one hidden NEURON, where two simple processing occurs to give a response (OUTPUT)

STEP 1

summation of inputs weighted + Bias

$$z = w_1 x_1 + w_2 x_2 + b$$
$$= \sum_i w_i x_i + b$$

(I can have more inputs x_i = features)

BIAS is important in this STEP1. For example if w_i are randomly initialized (as usual) if $w_i = 0 \ \forall i \Rightarrow$ bias b need to be $b \neq 0$ to avoid initial deactivation of the NEURON

b: threated as a NOISE = the bias handle generalization of the model

(In Linear Regression problems the noise b is the intercept)

STEP 2

On top of z computed we apply an activation function

↑

It is the main aim of transforming the value.

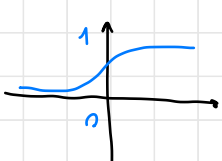
Transforms the output z between some values, such as $0 \div 1$, $-1 \div 1$

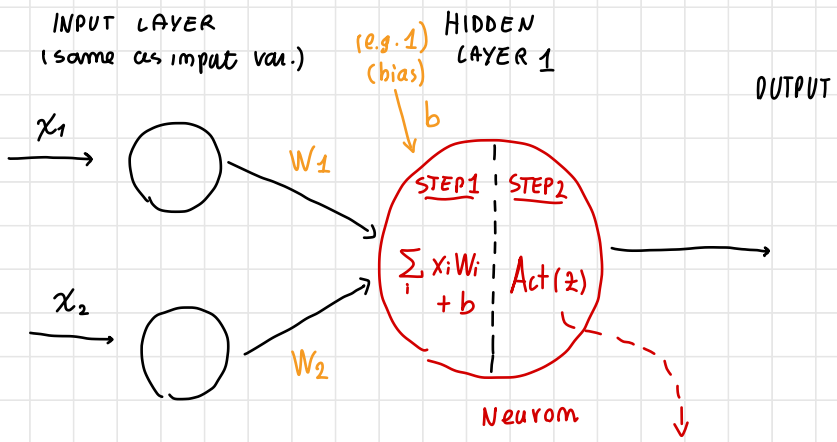
$0 \div 1$ I use SIGMOID Activation or a STEP Activation

↓

STEP function

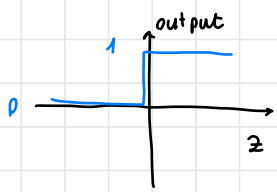
SIGMOID Function





the importance of ACTIVATION FUNCTION for ex. in BINARY CLASSIFICATION ← after performing STEP 1 and using data I want 0 ÷ 1 as output

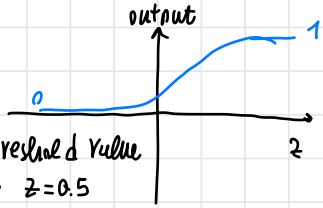
STEP Function



$$\begin{cases} 0 & z \le 0 \\ 1 & z > 0 \end{cases}$$

threshold value is $z = 0$ in STEP

SIGMOID Function



threshold value is $z = 0.5$

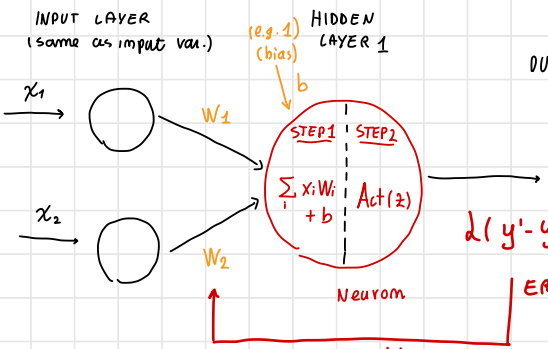
$$\begin{cases} 0 & z \le 0.5 \\ 1 & z > 0.5 \end{cases}$$

So the ACTIVATION FUNCTION transform output of STEP 1 in a 0 ÷ 1 value ← solve BINARY CLASSIFICATION require the mapping of $z = \sum_i x_i W_i + b$ in $[0, 1]$

ex in the first data $x_1 = 95$ $x_2 = 3$ $(y = 0)$

output = $Act(x_1 w_1 + x_2 w_2 + b) = ? \rightarrow 0$ if I get 0 as output of the single layer Perceptron it is CORRECT, then ERROR = 0 ↓ good weights initialization!

IF ERROR ↑↑ I need to update the weights ⇐



the weights are updated by going FOREWARD / BACKWARD (forewand / backwand propagation) ↑

⇐ the ERROR used in the architecture is the LOSS FUNCTION (or COST FUNCTION)

update weights based on e... → weights update until the prediction is satisfactory for all the records (data) ⇓ good accuracy with respect to our data

Step-by-Step:

STEP 1 $z = \sum_{i=1}^{m} w_i x_i + b = w_1 x_1 + w_2 x_2 + .. + w_m x_m + b$

↕ LINEAR PROBLEM statement

equation similar to $y = mx + c$ (LINEAR REGRESSION)

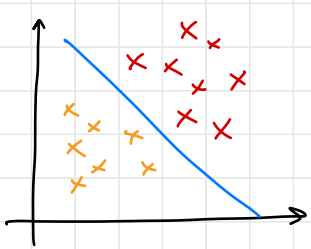
$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + .. + \beta_n x_n$

+ STEP 2 binary selection

PERCEPTRON is a linear classifier in step 1 + step 2 ←

overall with step 1 + step 2, in a classification problem

- Perceptron create a → line that separate the two classes!



At the end Simple PERCEPTRON is a LINEAR CLASSIFIER combining STEP1 + STEP2

↑ each ACTIVATION FUNCTION has its own functionality

↓

What are PROs and CONs of PERCEPTRON ?
+ how to move to Multi Layer Perceptron
+ how does a NN actually work

↖ Simple Neural Network are NOT feasible for complex problems

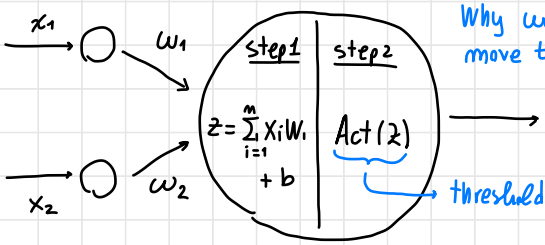
PROs AND CONs OF PERCEPTRON

PERCEPTRON MODELS

the one discussed before...

SINGLE LAYER PERCEPTRON ⇒ What are its PROs and CONs? Why we need to move to multi layer perceptron

MULTI LAYER PERCEPTRON = ANN (multi layered neural networks)



Feed Forward Neural Network

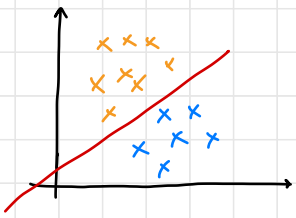
performing all operations = moving data from left to right

from a given input ↗ we assign weights ↗ we perform 2 steps... ↗ after Activation the boolean classification guarantee 1 OR 0 output

IF given x_1, x_2 I know $y = 1$ in dataset
↓ but I get $\hat{y} = 0$ in the current model
the weights are updated ← there is an ERROR!
and we perform again feed forward... until $\hat{y} \simeq y$ (error negligible, good performance)

NOT effient, but as PRO: linearly separable classification problems can be solved with Perceptron model

e.g. Linearly separable problem



using perceptron, the CLASSIFICATION problem can be solved finding a best Fit line.

BINARY CLASSIFIER

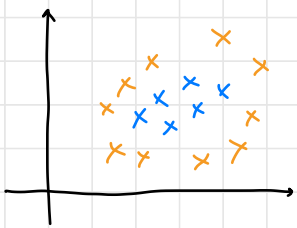
but changing w: randomly when $\hat{y} - y \neq 0$ is NOT efficient...

In MLP models we rely on LOSS FUNCTIONS, BACK PROPAGATION ← seamless weight update!

single layer NN perceptron are good just for simple binary classifier LINEARLY SEPARABLE

...

IF instead



NON LINEAR... a simple line doesn't exist SEPARABLE

⇓

MLP NN is required!

single layer is NOT enough

In MLP new techniques allow to achieve good performances

↓

1) Forward propagation
2) Backward propagation
3) Loss function
4) Various Activation functions → which one to use? when?
5) Optimizers!

After feed forward ⇒ a mechanism to update weights of the architecture model is required

↓

With (MLP) / ARTIFICIAL NEURAL NETWORK (ANN) we can create DEEP NN models to solve complex problems

MULTI LAYERED PERCEPTRON MODEL (ANN)

1) Forward Propagation
2) Backward Propagation → Geoffrey Hinton (thanks to his research on back propagation techniques we can create DeepNN)
3) Loss functions
4) Optimizers
5) Activation function

⇐ high level understanding of MLP concepts

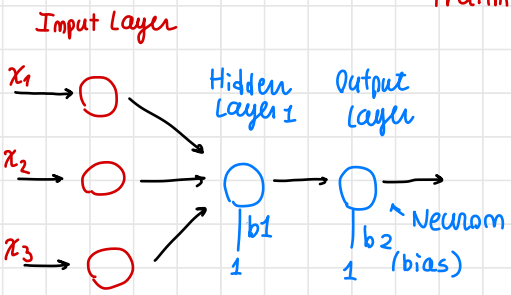
MLP → Multi Layered Neural Network

{ While in a Single Layer Perceptron model we had feed forward NN + thresholding (activation function) └ that allow for binary classification }

⇓

In MLP we focus on a two step process Forward and Backward propagation (FP and BP) ↑

in this 2 steps we can obtain effecient weight updates of the ANN, training the NN better



Multilayer NN (more than one hidden layer)

each layer can have multiple neurons ←

Dataset

IQ	Study Hours	Play Hours	output Pass/Fail
95	4	4	1
100	5	2	1
95	2	7	0
x_1	x_2	x_3	

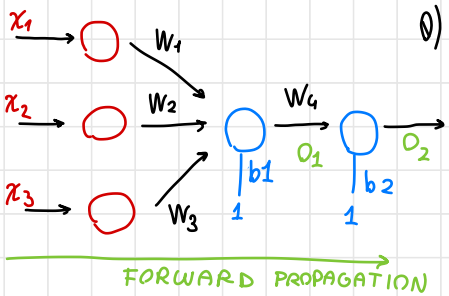
3 INPUT features that pass to my input layer

+ bias $b_i \times (+1/-1)$ to avoid zero initialization (random initialization)

⇑

2 Layered Neural Network (any number of hidden layers and neurons are feasible)

FORWARD PROPAGATION: sequence of steps through the NN



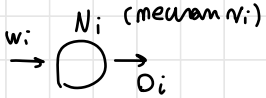
0) • Feed Forward NN steps INITIALIZATION

0.a) assign random bias b_i = rand (some constant)

0.b) assign random weights W_i (there are different initialization techniques) to each connection in the network require a weight

1) **FORWARD PROPAGATION** (FP)

Im the neuron, two steps occurs O_i os output of neuron N_i



$$\begin{cases} \text{step1)} \ \sum x_i W_i + b_i = z_i \\ \text{step2)} \ O_i = Act(z_i) \end{cases}$$

① $z_1 = x_1 w_1 + x_2 w_2 + x_3 w_3 + b_1 = \sum_{i=1}^{3} W_i x_i + b$ ② $O_1 = Act(z_1)$

Im FP, this operations is performed in ANY hidden neuron!

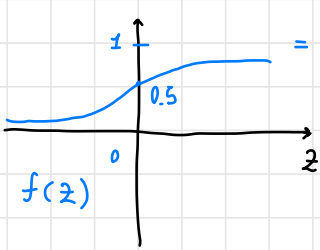
e.g. $W_1 = 0.01$, $W_2 = 0.02$, $W_3 = 0.03$ for DATA x_1 $x_{11} = 95$ $x_{12} = 4$ $x_{13} = 4$

$b_1 = 0.01$ HIDDEN LAYER 1

STEP1) $z_1 = 95 \cdot 0.01 + 4 \cdot 0.02 + 4 \cdot 0.03 + 1 \cdot 0.01 = 1.151$

STEP2) $O_1 = \text{Activation}(z_1) = \text{Activation}(1.151)$

consider for example SIGMOID



$= \frac{1}{1+e^{-z}} \in [0,1] \ \forall z$

many different activation functions exists (ReLu, ...)

the activation $0 \div 1$ represent an "ACTIVATION" state of the neuron based on the INPUT ... (therefore response required next ...)

important role in ACTIVATING and DEACTIVATING the Neuron (when it has no more reason to respond no role)

this is possible only if correct weights are used.

$f(z) = \frac{1}{1+e^{-1.151}} = 0.759$ hidden layer 1 $O_1 = 0.759$

Hidden layer 2:

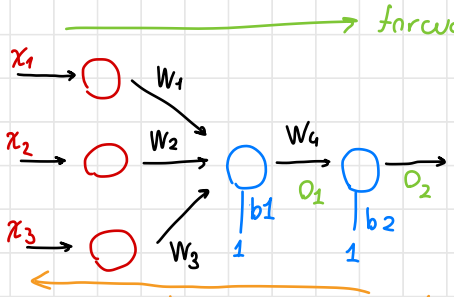
initialized $w_4 = 0.02$ $b_2 = 0.03$

STEP1) $z_1 = O_1 \cdot w_4 + b_2 = 0.759 \cdot 0.02 + 0.03 \cdot 1 = 0.04518$

STEP2) $O_2 = \text{Activation}(z_1) = \frac{1}{1+e^{-0.04518}} = 0.51129$

$\hat{y} = 0.51129$

PREDICTED output of the model



LOSS FUNCTION

↑ after we get the output $O_2 = \hat{y}$ of the MLP ANN, we have the loss function

$\hat{y}_1$ is a prediction of y_1

LOSS FUNCTION = $(y_1 - \hat{y}_1) = (1 - 0.51129) \simeq 0.49$ ERROR WE GOT IN PREDICTION

this is a big error!

$\hat{y} = 1$ would be $y - \hat{y} = 1$ OK!

my main objective is to reduce this error! ⇒ MINIMIZE THE ERROR OF PREDICTION

to reduce the error (and minimize it)

we update all the weights w_i: backward propagation process

from $w_4 \to w_3, w_2, w_1 \ldots$

we try to update all the weights...

once backward propagation finish, w_i are updated for $x_1 = (x_{11}, x_{12}, x_{13})$ we update w_i $\forall i$

the next INPUT will be sent... $x_2 = (x_{21}, x_{22}, x_{23})$ and again forward/backward propagation occurs!

loss = $y_2 - \hat{y}_2$ and w_i updates... ↑ the updates of w_i occurs if the loss needs to be reduced!

FP/BP process continue to reduce $\mathcal{L}$ to satisfactory value.

LOSS FUNCTION vs **COST FUNCTION**

computing $(y_i - \hat{y}_i)$ for every point

↑ calculate in each step updating the weights.

computing the error overall!

$$\sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

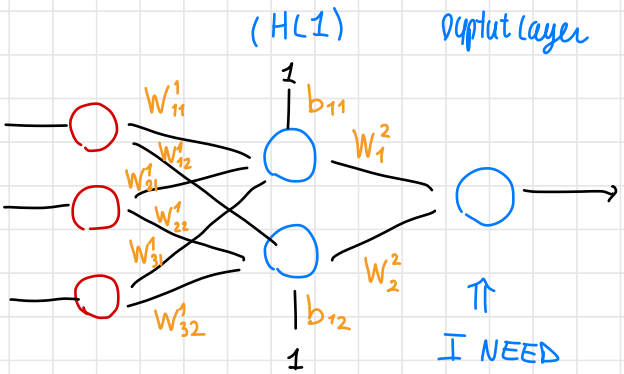
for all the dataset

calculate the error once + update weights just once.

In order to update w_i after the loss function is computed, during backward propagation **OPTIMIZERS** are used.

↑ helps in BP to reduce loss and updates weights

BACK PROPAGATION AND WEIGHT UPDATE FUNCTION



DATASET

IQ	Study Hours	Play Hours	Pass/Fail
95	4	4	1
100	5	2	1
95	2	7	0
x_1	x_2	x_3	

INPUTS = independent features

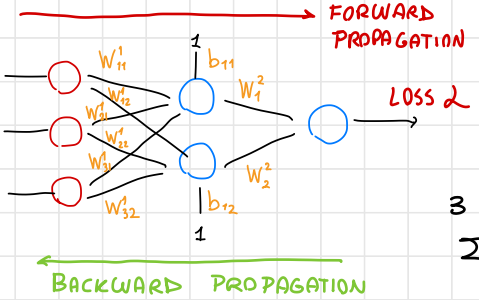
↑ If this change the output change (dependent)

OUTPUT = dependent feature

I NEED to have one Neuron in the OUTPUT layer IF I'm solving a binary classification problem $y = \{0, 1\}$

the number of neurons in the output layer depend on the problem solved ←

IF multi-class classification problem I can use more neurons



having more neurons and input, now the operation during forward propagation changes.

3 input layers × 2 hidden layers

$\sum_i x_i W_{ij}^e = x_1 W_{11}^1 + x_1 W_{12}^1 + \dots$

$x_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \end{bmatrix} \quad W^1 = \begin{bmatrix} W_{11}^1 & W_{12}^1 \\ W_{21}^1 & W_{22}^1 \\ W_{31}^1 & W_{32}^1 \end{bmatrix}$

$\leftarrow \quad x_i^T W^i : \quad 1 \times 2 = 2_1 \dots$

than we forward until $\hat{y}_i$

and being BINARY CLASSIFICATION PROBLEM the **loss function** depend

ℒ

REGRESSION

- Mean Squared Error (MSE)
- Mean Absolute Error (MAE)
- Huber Loss

CLASSIFICATION

- Binary Cross Entropy
- Categorical Cross Entropy

basic loss: $(y-\hat{y})^2 \Rightarrow$ main aim is to reduce this Loss

if this is BIG → we updates w_i during BACKWARD PROPAGATION

Weight update formula

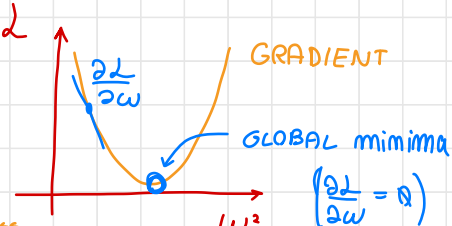
after loss computation, I start weight updates from the last layer W_1^2, W_2^2

$$W_{1\,NEW}^2 = W_{1\,OLD}^2 - \eta \boxed{\frac{\partial \mathcal{L}}{\partial w_{1\,OLD}^2}}$$

$\boxed{\frac{\partial \mathcal{L}}{\partial w_{1\,OLD}^2}}$ ← SLOPE of the GRAPH $\mathcal{L}(W_{1\,OLD}^2)$

⇓ positive or negative loss

↑ we update weights as this



Usually the loss has a parabolic shape ↑ GRADIENT DESCENT

Where 0 slope $w_{new} = w_{old}$, we don't need to update weights

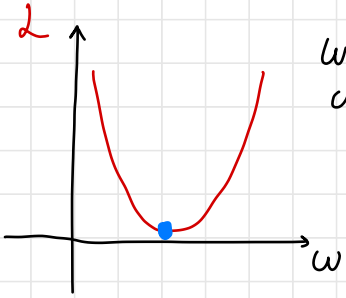
im the same way $w^2_{2\,NEW} = w^2_{2\,OLD} - \eta \frac{\partial \mathcal{L}}{\partial w^2_{2\,OLD}}$

↓

this formula work in updating the Wi effecently to reduce $\mathcal{L}$

↑ LEARNING RATE (hyperparameter)

$$W_{new} = W_{old} - \eta \frac{\partial \mathcal{L}}{\partial W_{old}}$$ Weight update formula



↙

With the $\mathcal{L}$ used as MSE, MAE, ... we get "gradient descent" shape $\mathcal{L}(w)$

↓

main AIM is to update all weights w_i With BACK PROPAGATION

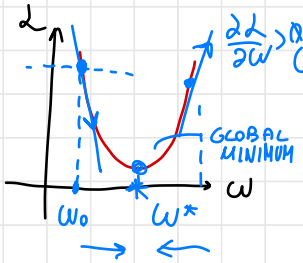
⇓

Im order to minimize the loss function $\mathcal{L} = (y-\hat{y})^2$ we use OPTIMIZERS

↓

GRADIENT DESCENT optimizer (for example ... other optimizers exists)

the ROLE of optimizers is to reduce the loss value $\mathcal{L}\downarrow\downarrow$ by updating the weights during BACKWARD PROPAGATION



$W_{new} = W_{old} - \eta \left(\frac{\partial \mathcal{L}}{\partial W_{old}}\right)$

$\frac{\partial \mathcal{L}}{\partial w_0}$:= slope in w_0 of $\mathcal{L}$

1) understand

if slope $\gtrless 0$

right side of tangent point downward ⇒ slope < 0

$\frac{\partial \mathcal{L}}{\partial w_0} < 0$ than ⇒ $W_{new} = W_{old} - \eta\left(-\left|\frac{\partial \mathcal{L}}{\partial W_{old}}\right|\right)$

$= W_{old} + \eta \left|\frac{\partial \mathcal{L}}{\partial W_{old}}\right|$

$W_{new} > W_{old}$

INCREASING!

$\frac{\partial \mathcal{L}}{\partial w_0} < 0$

we move towards the GLOBAL MINIMUM w^*

In the same way if $w_{old} > w^*$ $\quad \frac{\partial \mathcal{L}}{\partial w_{old}} > 0$:

move on the left towards w^* $\Leftarrow$ $w_{new} = w_{old} + \left(-\eta \left| \frac{\partial \mathcal{L}}{\partial w_{old}} \right|\right) < w_{old}$

↓

the main aim is to move toward w^* with update

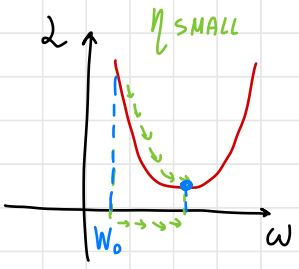
The **LEARNING RATE** η is an important HYPERPARAMETER

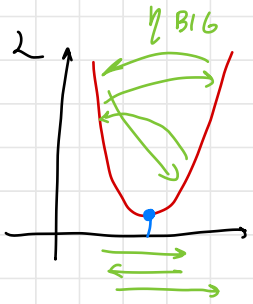
↓

usually small initialization e.g $\eta \simeq 0.001$ SMALL STEP

define the "STEP SIZE" towards global minimum → HOW FAST CONVERGING during ITERATIVE weight update

If η is BIG ⇒ we move with big steps! It can lose convergence





it may never converge

Good practice to use η SMALL

The optimizer know that it reached the optimum when

$\frac{\partial \mathcal{L}}{\partial w} = 0 \Rightarrow$ $w_{new} = w_{old}$ → w REACHES GLOBAL MINIMUM

$\mathcal{L} \simeq 0$ NO need to further change w_i !!

↑ IDEA of GRADIENT DESCENT OPTIMIZER

(other OPTIMIZERS for weights update exists...)

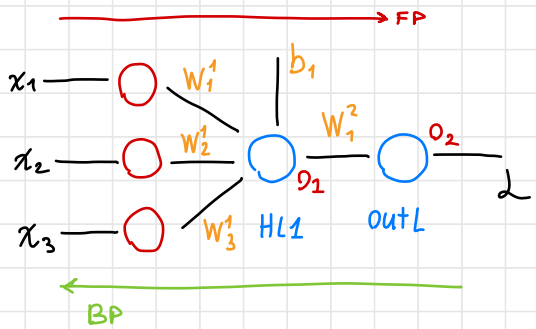
BUT notice that w_i depends on w_j $\forall j \neq 1$, all weights are somehow connected since

$$\mathcal{L}(w) = \mathcal{L}(w_1, w_2, \dots, w_i, \dots, w_N)$$

and each $\frac{\partial \mathcal{L}}{\partial w_j}(w_i, w_k \dots)$ → a specific technique is considered

↓ CHAIN RULE OF DERIVATIVE (used to compute those weights)

CHAIN RULE OF DERIVATIVES



related to w updates formula and how it can be computed

$$W_{new} = W_{old} - \eta \frac{\partial \mathcal{L}}{\partial W_{old}}$$

the weights update start from last ...

$$W^2_{1\,NEW} = W^2_{1\,OLD} - \eta \left[\frac{\partial \mathcal{L}}{\partial W^2_{1OLD}} \right]$$

↳ by expanding it $\mathcal{L}$ depends on all w_i

$$\frac{\partial \mathcal{L}}{\partial W^2_{1OLD}} = \frac{\partial \mathcal{L}}{\partial O_2} \cdot \frac{\partial O_2}{\partial W^2_{1OLD}}$$

I can apply chain rule of derivation and introduce O_2

CHAIN RULE of DERIVATIVES

↓

this allow to compute each component of the weights update

In the other terms

$$W^1_{1\,NEW} = W^1_{1OLD} - \eta \frac{\partial \mathcal{L}}{\partial W^1_{1OLD}}$$

$O_1(w_1)$ as output of first hidden layer + $O_2(w_1^2)$ depends on O_1 also...

↓

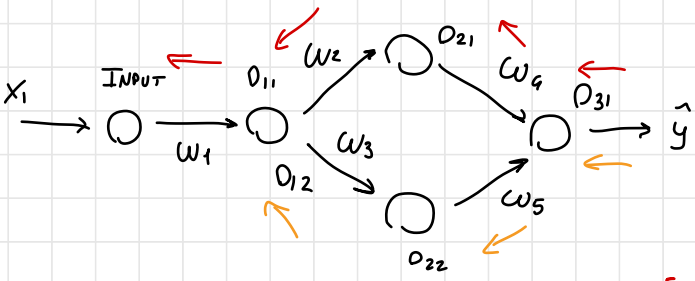
Chain rule apply to expand.

$$\frac{\partial L}{\partial W^1_{1old}} = \frac{\partial L}{\partial O_2} \cdot \frac{\partial O_2}{\partial O_1} \cdot \frac{\partial O_1}{\partial W^1_{1old}}$$ CHAIN RULE

$$L(O_2) = L(O_2(W^2_1)) = L(O_2(W^2_1(O_1)) = L(O_2(W^2_1(O_1(W^1_1))) \ldots$$

↑ dependency chain!

Example



$$W_{1new} = W_{1old} - \eta \left[\frac{\partial L}{\partial w_{1old}}\right] \rightarrow \frac{\partial L}{\partial w_{1old}} = \left[\frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{21}} \frac{\partial O_{21}}{\partial O_{11}} \frac{\partial O_{11}}{\partial W_{1old}}\right] +$$

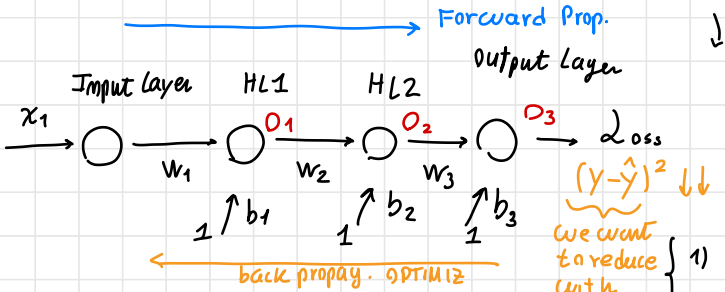
$$+ \left[\frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial O_{12}} \cdot \frac{\partial O_{12}}{\partial W_{1old}}\right]$$

to update W_{1new} this chain rule equation will be used

This always happens with help of optimizer

$\frac{\partial L}{\partial O_i}$ can be computed efficiently as loss wrt outputs of neurons

VANISHING GRADIENT PROBLEM AND ACTIVATION FUNCTIONS



with multiple hidden layers ⇒ Two layered NN (here with just one Neuron per layer)

$$z = \sum_{i=1}^{N} w_i x_i + b$$

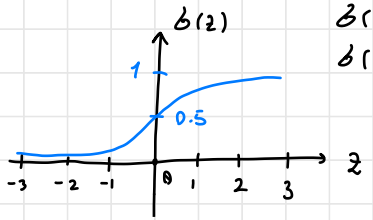
Sigmoid activation $\sigma(z) \Rightarrow 0 \div 1$ (sometimes $\phi(z)$ Notation)

Sigmoid Activation function

Main aim is to transform its value as $\sigma(z) \in [0,1)$ ← usefull for example in binary classification where the output should be between 0 and 1

⇓ mathematically

$$\sigma(z) = \frac{1}{1+e^{-z}}$$



$\sigma(0) = 0.5$

$\sigma(\infty) \to 1$ $\sigma(-\infty) \to 0$

always map z in $(0,1)$

here to update the weights

back propagation we start from behind ⇑ to update pattern.

$$W_{1new} = W_{1old} - \eta \left[\frac{\partial L}{\partial W_{1old}}\right]$$

can be expanded with chain rule

$$\frac{\partial L}{\partial W_{1old}} = \frac{\partial L}{\partial O_3} \cdot \left[\frac{\partial O_3}{\partial O_2}\right] \cdot \frac{\partial O_2}{\partial O_1} \cdot \frac{\partial O_1}{\partial W_{1old}}$$

$\eta = 0.01$ random small initializ of the LEARNING RATE

how can I solve this?

$$O_3 = \sigma(W_3 \cdot O_2 + b_3) = \sigma(z)$$

$z = W_3 O_2 + b_3$

$$\frac{\partial O_3}{\partial O_2} = \frac{\partial(\sigma(z))}{\partial z} \cdot \frac{\partial z}{\partial O_2}$$

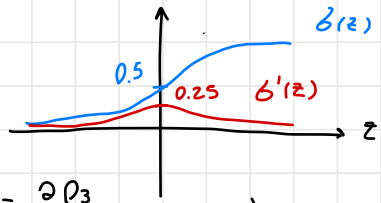
Sigmoid derivative

chain rule again. to split $\partial O_3 / \partial O_2$

by applying this chain since $\sigma(z) \in (0,1) \to 0 \le \sigma(z) \le 1$ always when apply activation

$\frac{\partial(\sigma(z))}{\partial z} \in (0, 0.25)$! IMPORTANT $\Rightarrow$ $0 \leq \sigma'(z) \leq 0.25$ (math proof exists)

While $\frac{\partial z}{\partial o_2} = w_{3oed}$



$\Downarrow$

$\frac{\partial \sigma(z)}{\partial z} \cdot \frac{\partial z}{\partial o_2} = (0 \leq \sigma'(z) \leq 0.25) \cdot \underbrace{w_{3oed}}_{\in (0,1)} = \frac{\partial o_3}{\partial o_2} \in (0, 0.25)$

$\in (0,1)$ initialization in small value distribution

What happens is that every derivative $0 \div 0.25$

$\frac{\partial L}{\partial w_{1oed}} = \frac{\partial L}{\partial o_3} \cdot \frac{\partial o_3}{\partial o_2} \cdot \frac{\partial o_2}{\partial o_1} \cdot \frac{\partial o_1}{\partial w_{1oed}}$

$0 \div 0.25 \quad 0 \div 0.25 \quad 0 \div 0.25 \quad 0 \div 0.25$

small value

small rate

small

$\Downarrow$

$w_{1new} = w_{1oed} - \eta \frac{\partial L}{\partial w_{1oed}}$

almost null

$\Downarrow$

$w_{1new} \simeq w_{1oed}$

Negligible update of your weights! You are not moving towards convergence $w_{i new} \simeq w_{i oed}$ ...

This "stack" occurs because of the sigmoid activation function

$\Rightarrow$ PROBLEM of VANISHING GRADIENT

Very important problem in caused by sigmoid because $\sigma(z)$ is s.t $\sigma'(z) \in (0, 0.25)$

Im a Deep NN, with more layers ... the chains in the chain rule is longer $\rightarrow$ even more vanishing

$\frac{\partial L}{\partial w_{i oed}} \simeq 0$ $w_{i new} \simeq w_{i oed}$ ....

Im the same way to compute $w_{2new} = w_{2old} - \eta \frac{\partial L}{\partial w_{2old}}$

$$\frac{\partial L}{\partial w_{2old}} = \frac{\partial L}{\partial o_3} \boxed{\frac{\partial o_3}{\partial o_2}} \frac{\partial o_2}{\partial w_{2old}}$$

(∂L/∂o3 ≈ 0.20, ∂o3/∂o2 ≈ 0.01, ∂o2/∂w2old ≈ 0.. --)

→ SMALL VALUE keep appending

$$\frac{\partial o_3}{\partial o_2} = \frac{\partial(\sigma(z))}{\partial z} \cdot \frac{\partial z}{\partial o_2}$$

$z = (o_2 \cdot w_3) + b_3$ in this case

$\simeq [0 \div 0.25] \cdot w_3$ SMALL VALUE (since $w_3 \simeq 0 \div 1$)

as initialization techniques

$W_{2new} \simeq W_{2old}$ → NO weight update!
We are back propagating but NOT converge!
it get stucked..

SIGMOID

PRO

- suitable for Binary classification: (0,1) output
- clear prediction since it has output close to 0 OR 1

CONS

- prone to VANISHING GRADIENT problem
- Function output is NOT zero-centered*..
- The Exponential e^{-z} in $\sigma(z)$ is demanding for computers (relatively time consuming)

this allow efficient Weight update!!

the zero-centering is done when training models with gradient descent

this helps in weight updates

* ZERO-CENTERED

0 mean ≡ zero-centered

⇒ efficient Weights-update

In gradient descent optimizers

this zero centering ⇐ In ML linear Regression and classification we perform STANDARDIZATION

⇓ zero-centering

$\mu = 0$ mean
$\sigma = 1$ std. dev

To Fix this problem ⇒ Researchers worked in Exploring other Activation functions

- ReLu
- Tanh
- Pre ReLu
- Swiss

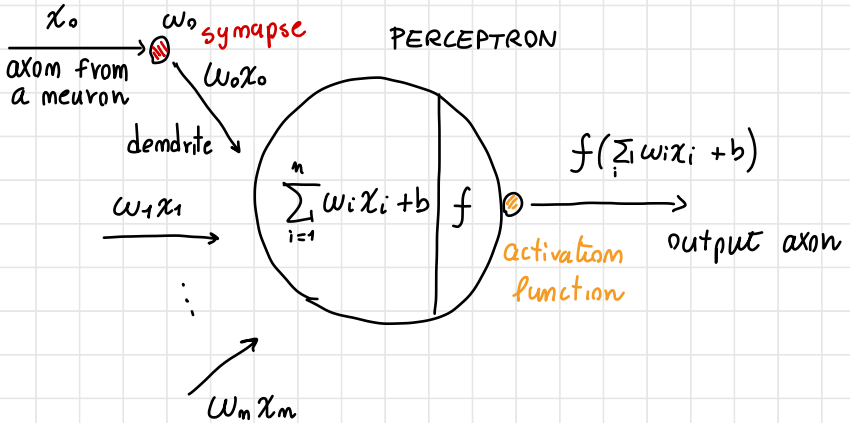
⇒ Which Activation should be used in HL? and which one in output layer.

Sigmoid can be good in small NN, BUT in a Deep NN you get a small $\partial L/\partial w$ that keep $w_{new} \simeq w_{old}$ due to gradient vanishing

ACTIVATION FUNCTION

Attached to each neuron in the network, determines whether it should activate or not, based on whether each neuron's input is relevant for the model's prediction

+ helps to normalize output in $0 \div 1$ or $-1 \div 1$ for each neuron

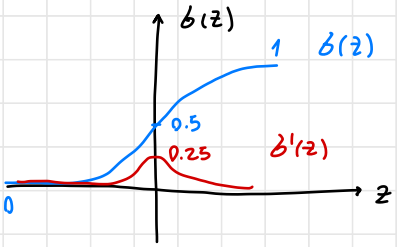


Sigmoid function

(RECAP)

main aim

transform $z \to \hat{\sigma}(z) \in (0,1)$ as $\sigma(z) = \frac{1}{1+e^{-z}}$



$\sigma'(z) \in (0, 0.25)$ cause of vanishing Gradient problem

⇓

during back propagation the value of $\partial L / \partial W$ is small and the update is stopped

the chain rule of chained small number cause this issue

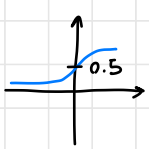
because of this reason we rely on other activations...

- It was the most used originally, for SMALL NN

↓

but WITH DNN is a PROBLEM! but in principle is smooth and easy to derive...

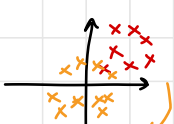
- Sigmoid ~ firing rate of a Neuron (how much active/inactive)
- function output is NOT centered in 0! Reducing efficiency of Weight update



NO 0-centered

0-centered function ⇒ the mean is 0

In ML algorithms from a dataset



We apply standard (CENTERING) scalar

STANDARD SCACAR guarantee $\begin{cases} \mu = 0 \\ \sigma = 1 \end{cases}$

↘ having a 0-centered function is good since, as proved, if points are zero centered (function zero center) There will be an efficent update in weights

⇓

helping us to compute the derivative quickly

- the SIGMOID function performs EXPONENTIAL operations which is slow for computers.
  ↳ takes time!

but $\sigma(z)$ has advantages

⇓

- it is smooth, prevent "jumping" in output
- good output normalization in (0,1)
- clear predictions (very close to 1 OR 0)

The main disadvantages

- Prone to gradient vanishing
- Not-zero centered output
- time consuming exponential operation

⇓

because of this other possible activation functions exists, again with their pros and cons

TANH ACTIVATION FUNCTION

While sigmoid $\sigma(z) \in (0,1)$, $\sigma'(z) \in (0, 0.25)$ ← backpropagation face the gradient vanishing + disadvantages discussed...

⇓

tanh solves some of the gradient descent problems

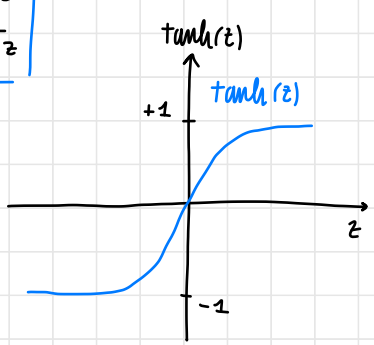
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

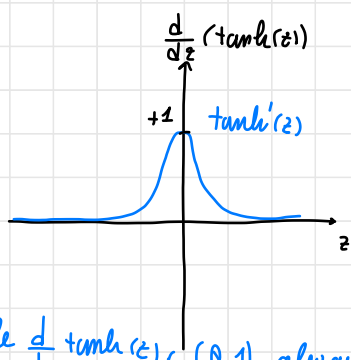
$(z = \sum_{i=1}^{m} w_i x_i + b)$ ↑

$\tanh(z) \in (-1,1)$

↑

$\forall z$, the activation by tanh map between -1 and 1





While $\frac{d}{dz}\tanh(z) \in (0,1)$ always

$\tanh'(z) \in (0,1)$ → advantage with respect to vanishing gradient problem

tanh is an hyperbolic tangent function
and it has a curve similar to the sigmoid, but with -1, 1 value
+ tanh is 0-centric (better than sigmoid)
mean 0

being $\frac{d}{dz}\tanh \in (0,1)$ in a medium side DNN you don't face gradient vanishing... but in a very Deep NN you face some vanishing...
↓
multiply a chain of smaller values...

For generic binary classifier tanh is used for hidden layers, while sigmoid for output layer

+ Notice as disadvantage that
- e^z is computationally expensive
- vanishing problem steel present in DNN

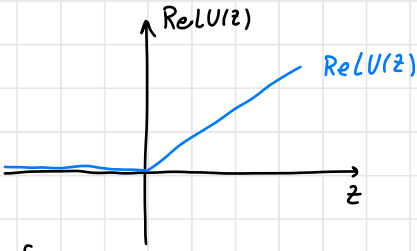
advantage
- 0-centric = efficient weight update

to fix some of this problems, other activation function exists... to solve the MAJOR problem related to VANISHING GRADIENT PROBLEM the ReLu activation is the solution...

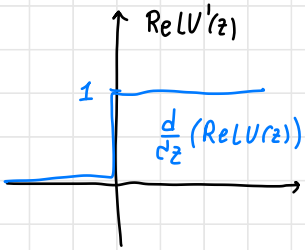
ReLU ACTIVATION FUNCTION

~> this will solve the Vanishing gradient BUT still one problem remain...

$$ReLU(z) = \max(0, z)$$



$$\begin{cases} \text{IF } z \le 0 \Rightarrow ReLU(z) = 0 \\ \text{IF } z > 0 \Rightarrow ReLU(z) = z \end{cases}$$



While

$\frac{d}{dz} \tanh \in (0,1)$

$\frac{d}{dz} \sigma \in (0, 0.25)$

$\frac{d}{dz} ReLU = \{0, 1\}$

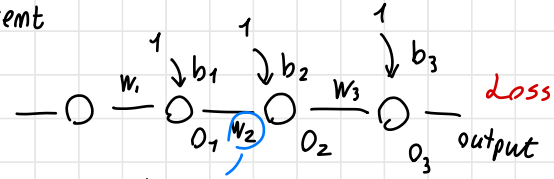
during backward propagation

$$\frac{d}{dz} ReLU(z) = \begin{cases} 0 & \text{IF } z \le 0 \\ 1 & \text{IF } z > 0 \end{cases}$$

↓

this solves Vanishing gradient

in a deep NN



to update w_2... using previous equations

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial O_3} \frac{\partial O_3}{\partial O_2} \frac{\partial O_2}{\partial w_2}$$

$\{0,1\}$

$$\hookrightarrow \frac{\partial O_3}{\partial O_2} = \frac{d(ReLU(z))}{dz} \frac{dz}{dO_2} = \{0,1\} \cdot w_3$$

IF $\frac{d}{dz} ReLU(z) = 1$: weight update will happen!

$w_{new} \gtrless w_{old}$ NOT stacked!

IF $ReLU(z) = 0 \Rightarrow$ this create a DEAD NEURON ($\frac{\partial L}{\partial w} = 0$)

neuron without functionality...

If derivative of $ReLU(z)$ is 0: $w_{new} \approx w_{old}$

$\Downarrow$

$w_{new} = w_{old}$

the entire neuron will pass the information, without updating its value...

the $\frac{\partial ReLU(z)}{\partial z} = 0$ if $z \leq 0$

While if $z > 0$: $ReLU(z) = z \rightarrow \frac{\partial ReLU(z)}{\partial z} = 1$

↓

- $z > 0$: OK ✓
- $z \leq 0$: dead Neuron

even if this solve the vanishing gradient, when $z \leq 0$ it create a dead neuron..

ReLU: advantages
- It is solving the vanishing gradient problem
- Activation $\max(0,x)$ has a FAST calculation + the ReLU function has a linear Relationship
- It is much faster than sigmoid or tanh

disadvantages
- It leads to dead neuron problems
- ReLU function output is $(0,z)$

⇓

It is NOT zero-centric

there are scenarios in which it is possible to handle that $z \leq 0$, but als other Activation functions can handle it

The next version of ReLU is Leaky ReLU...

ReLU := "Rectified Linear Unit"

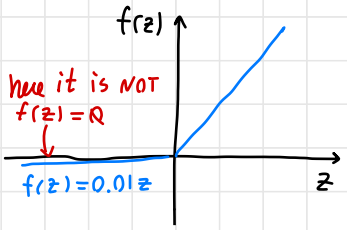
↑

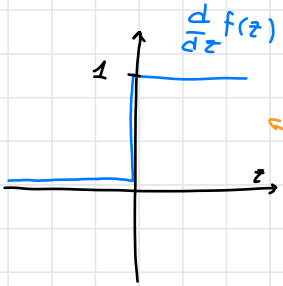
Specific type of Linear Activation..

Another is ELU (Exponential Linear Unit)

Leaky ReLU and Parametric ReLU

$f(z) = \max(0.01z, z)$





← $\frac{d}{dz}$ Activation is used for Back Propagation

- since ReLU problem was related to the Dead Neurom Problem

  ↓

  Dead ReLU Problem (entire neurom deactivated)

  ⇓

  Solved by Leaky ReLU and parametric ReLU

by using a small constant instead of $\max(0, z)$ we use $\max(Kz, z)$

In PARAMETRIC ReLU this small $K \simeq 0.01$

$\max(\alpha z, z) = f(z)$

hyperparameter to tune ← by hyperparameter tuning

$\alpha = 0.01, 0.02, \ldots$ depending on the specific problem

When $\alpha = 0.01$ is Leaky ReLU

Thanks to this parameter α (OR 0,01 fixed) $\frac{d}{dz} f(z) \neq 0 \quad \forall z$

↓

always $\frac{d}{dz} f(z) > 0$ even for $z \leq 0$ ...

↓

Therefore in backpropagation we never get the dead neurom!

↑

fixed this problem!

Advantages
- Leaky ReLU has all the advantages of ReLU
- There is NO dead ReLU problem
  (it remove this problem thanks to small constant)
  + it solves also vanishing gradient problem

⇓

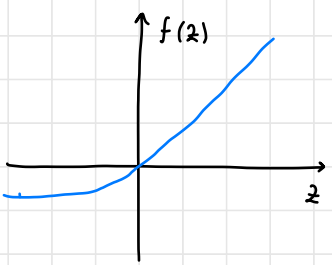
other Activations solves other problem...
Another Linear Unit is the Exponential Linear Unit activation...

Some Disadvantage of Leaky and Parametric ReLU
↓
still NOT zero-centric ! => weight update will NOT be done efficiently

↓ with ELU we try to solve the problem of ReLU about dead Neuron + additional advantages...

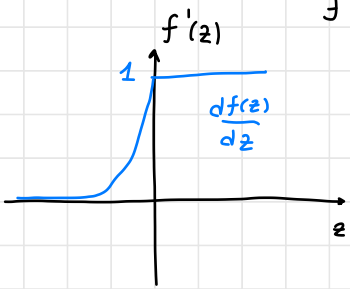
ELU Activation Function (Exponential Linear Unit)

$$f(z) = \begin{cases} z & z > 0 \\ \alpha(e^z - 1) & z \leq 0 \end{cases}$$



(Forward propagation)

$$f'(z) = \begin{cases} 1 & z > 0 \\ \alpha e^z & z \leq 0 \end{cases}$$



(Backward Propagation)

$f'(z) \neq 0 \ \forall z$ it solves dead Neuron problem!

Advantage
- No dead ReLU issues
- zero-centric Activation (the mean is 0)

Disadvantage
- Mathematically ... is slightly more intense computationally (Respect ReLU)

⇑

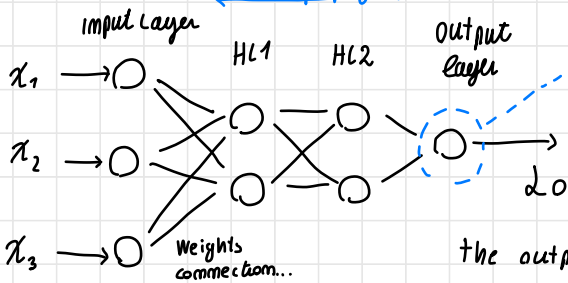
Used to solve ReLU Problems...

the usage of the Activation function is based on the problem and dataset

... the best combination of ReLU, sigmoid, etc... is a "magic"

SOFTMAX ACTIVATION FUNCTION

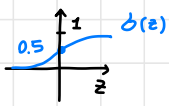
Specifically used in the output layer for a multi-class classification problem



With one output layer neuron

↓

BINARY CLASSIFICATION problem...



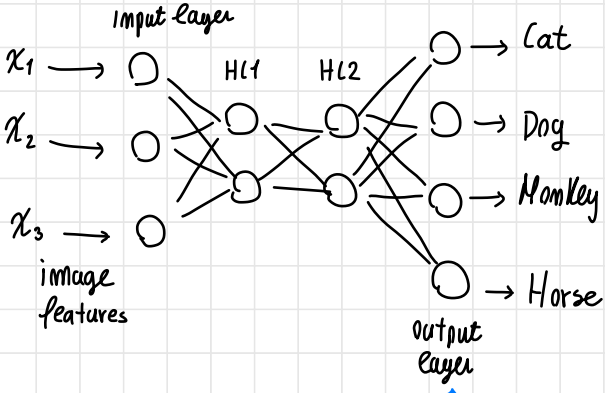
the output layer in binary classification has a SIGMOID ACTIVATION that map the values in 0 to 1 map

It transform $\sigma(z) \in (0,1)$ ⇐ the output of last layer...

↓

I can set a threshold. $\begin{cases} \sigma(z) \geq 0.5 \Rightarrow 1 \\ \sigma(z) < 0.5 \Rightarrow 0 \end{cases}$

BUT if instead I want to have multiple outputs:



multi-class classification problem (in the four output)

I need to predict if the input features (x_1, x_2, x_3, x_4) are related to one of the output classes

↙ In this scenario SIGMOID ACTIVATION is NOT the correct choice... (sigmoid give BINARY output)

↓

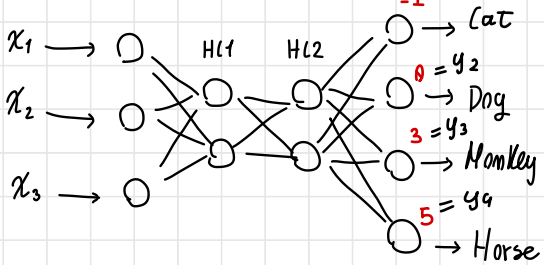
having 4 nodes in output layer each neuron give Prob(Cat)....

We want the PROBABILITY of each class...

⇓ we need to use

Softmax activation function:

before apply this softmax, each of the last layer node has a value



⇓

after this I need to apply softmax activation function!

$$\text{Softmax} = \frac{e^{y_i}}{\sum_{k=0}^{m} e^{y_k}}$$

$i :=$ node of the output layer

$m :=$ # output node

In this case $i = \{1,2,3,4\}$ $m = 4$

$$\text{Softmax: } Cat\,(y_1) := \frac{e^{y_1}}{\sum_{k=0}^{4} e^{y_k}} = \frac{e^{-1}}{e^{-1} + e^{0} + e^{3} + e^{5}} \cong 0.00033$$

(y_i = output * W + b)

$$\text{Dog: } \text{Softmax}(y_2) = \frac{e^{0}}{e^{-1} + e^{0} + e^{3} + e^{5}} = 0.0024$$

$$\text{Monkey} = \frac{e^{3}}{e^{-1} + e^{0} + e^{3} + e^{5}} = 0.0183$$

$$\text{Horse} = \frac{e^{5}}{e^{-1} + e^{0} + e^{3} + e^{5}} = 0.1353$$

↑ this is the maximum...

⇓

than the probability is found by normalizing

$\mathbb{P}(\text{Horse}) = 0.1353 / (0.00033 + 0.0024 + 0.0183 + 0.1353) \simeq 86\%$

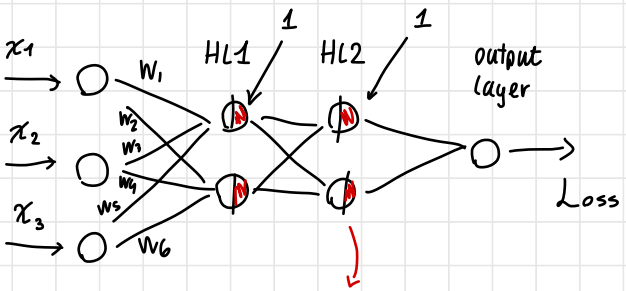
$\mathbb{P}(\text{cat}) = 0.00033 / (\dots) \dots$ ↑ highest! → the MLP predict Horse

the Softmax is applyed at output layer to calculate multi-class classification probability with respect to all categories present

Binary Classification ⇒ sigmoid Multi-Class Classification ⇒ Softmax

↓ But how to chose in the Hidden layers? and in other problems...

WHICH ACTIVATION FUNCTION TO USE ?



Usually after weights initialization.. and bias added, we apply activation function..

Input layer

When apply activation function...

When the NN is deep (many hidden layers...)

HL i ⇒ Sigmoid OR tanh ⇒ Vanishing Gradient Problem!

No

(In HL we avoid sigmoid / tanh)

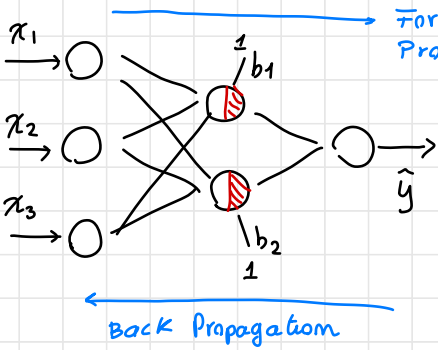
⇓

ReLU or its variant (Parametric ReLU, Leaky ReLU, ELU) ⇒ Which solve Vanishing Gradient

While in OUTPUT Layer:
- binary classification: sigmoid
- Multi class classification: softmax

(This holds in any ANN of any depth/lenght with any HL)

LOSS FUNCTION AND COST FUNCTION



$Loss = (y-\hat{y})^2$ ↓ goal is reduce the Loss! (error on prediction)

this is reduced with OPTIMIZERS... gradient descent for example update the weights during backpropagation

What are the different types of LOSS FUNCTION ? (and what about the COST FUNCTION)

Given data points: (dataset)	x_1	x_2	x_3	output
	//	//	//	0
	//	//	//	1
	//	//	//	0
	//	//	//	1

Loss function

we take ALL data points and do Forward + Backward propagation

e.g MSE (mean square Error)

$MSE = (y-\hat{y})^2$ ↙ this happens ∀ datapoint i

$MSE_i = (y_i-\hat{y}_i)^2$

Cost function

in this case we refer to a batch

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2$$

instead of one point we pass all / batch of points of dataset...

We propagate through the net multiple datapoints x simultaneously

↓ all $\hat{y}_i$ are used to compute it...

so ∀ output $\hat{y}$ of datapoint the error is averaged... and again the objective is cost function reduction

So.. In Loss function ∀ datapoint $x_i = (x_{i1}, x_{i2}, .., x_{if})$ in dataset
we compute $\hat{y}_i$ and calculate
$\mathcal{L} = (y_i - \hat{y}_i)^2$ and backpropagate + update weights...

update for every point

While in the cost a batch $i = k_1 \dots k_b$ of points $x_{k_1}, x_{k_2}, \dots, x_{k_b}$
and estimate $\hat{y}_{k_1}, \hat{y}_{k_2}, \dots, \hat{y}_{k_b}$ then the cost is
$\frac{1}{b} \sum_{i=k_1}^{k_b} (y_i - \hat{y}_i)^2$ and backpropagating we update the weight once

update for set of points

So, what are the main Loss and Cost functions?
(focus on Regression / Classification)

With the help of ANN, it is possible to solve → Classification, Regression

in this two main problems we use different Loss functions..

REGRESSION

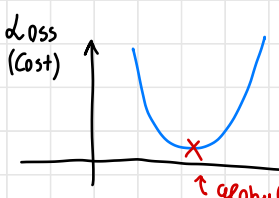
- Mean square Error (MSE)
- Mean Absolute Error (MAE)
- Huber Loss
- Root Mean Square Error (RMSE)

$\hat{y}$ is the prediction output of the ANN after forward propagate

MSE

Loss function := $(y - \hat{y})^2$ Cost function := $\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$

QUADRATIC EQUATION!
this is very important, meaning full choice



you have a paraboloid curve...
which has a GLOBAL MINIMUM

And the objective is to update the weight s.t. the loss decrease towards global minimum

Parabola curve ~ Gradient Descent ⇐ has ONE GLOBAL MINIMUM
+ Knows how to update formula

In COST FUNCTION also I get multi-dimensional paraboloid to update w_i ... the same in higher dimension

Advantages

- MSE is Differentiable
(@ any time $d\text{Loss}/dw$ can be computed)
as done during weight update

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

slope of loss

IF differentiable we can compute it

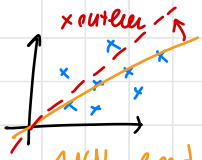
- MSE have ONE Local OR Global Minimum

I can get stucked to local since $\frac{\partial L}{\partial w} = 0$

- MSE converges faster thanks to the loss shape

Disadvantages

- NOT Robust to outliers



ANN find best fit line...

$(y_i - \hat{y}_i)^2$ error is squared..

because of outliers error is increasing by squaring out this x_{NOISE} ... PENALIZING the COST FUNCTION

the line move a lot!

(penalizing cost with errors)

MAE

Loss function = $|y_i - \hat{y}_i|$ Cost function = $\frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$

(Not squared, just absolute value..)

Advantages

- it is ROBUST to outliers

(NO more penalizing errors... even if I have outliers

NOT too much increased, since NOT squared

↓

therefore MAE is better than MSE if outliers

Disadvantages

- We don't get a paraboloid curve..

We cannot use Gradient Descent (NOT differentiable)

↓

we rely on SUBGRADIENT concept, where we compute SCOPE in different parts...

Convergence usually takes time in MAE

(IF you don't have outliers MSE is better..) ←

for big outliers you can remove it? but lose data

HUBER LOSS (Combination of MSE + MAE)

Cost function := $\begin{cases} \frac{1}{n}\sum_{i=1}^{m} (y-\hat{y})^2 & \text{if } |y-\hat{y}| \le \delta \\ \delta\,|y-\hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$

threshold hyperparameter ← if NOT outlier (low error)

← if outlier you scale up

↑

IF NO outlier you use MSE, otherwise you use a modification of MAE

RMSE is an update of MSE with square root

Cost function = $\sqrt{\frac{1}{n}\sum_{i=1}^{m} (y_i - \hat{y}_i)^2}$

Loss / Cost for CLASSIFICATION PROBLEMS

that we will minimize with optimizers

Classification → Binary / Multi-Class → we use a technique called CROSS-ENTROPY

modified based on the specific classification problem

- Binary Cross Entropy (BINARY)
- Categorical Cross Entropy (MULTI-CLASS)
- Sparse Categorical Cross Entropy (MULT-CLASS)

BINARY CROSS ENTROPY (used in Logistic Regression...)

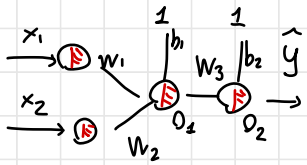
$$Loss = -y \cdot \log(\hat{y}) - (1-y)\log(1-\hat{y})$$ ← log-loss formula

{ y := Actual value
$\hat{y}$:= Predicted value

$$Loss = \begin{cases} -\log(1-\hat{y}) & \text{if } y=0 \\ -\log(\hat{y}) & \text{if } y=1 \end{cases}$$

two scenario in BINARY CASE

↓ example with ANN



$O_2 = \sigma(z_2)$ SIGMOID since BINARY CLASSIFICATION (output can be 0/1)

$$O_2 = \sigma(\underbrace{O_1 * W_3 + b_2}_{z}) = \hat{y}$$

back prop.

$$\hat{y} = \frac{1}{1+e^{-z}}$$

that loss is computed on this $\hat{y}$ of simple SIGMOID..

CATHEGORIKAL CROSS ENTROPY (used in multi-class classification)

from a dataset with features f:

f_1	f_2	f_3	Output
2	3	4	Good
5	6	7	Bad
8	9	10	Neutral

multi-class! (more than two)

the cathegorical CROSS ENTROPY start from output processing ⇒ apply OHE (One Hot Encoding) on the output data

Output	j=1 Good	j=2 Bad	j=3 Neutral
Good	1	0	0
Bad	0	1	0
Neutral	0	0	1

(OHE) y is encoded than the loss function

$$\mathcal{L}(x_i, y_i) = -\sum_{j=1}^{c} y_{ij} * \ln(\hat{y}_{ij})$$

$i = 1, \dots, N$ (all data points N)

c := # of cathegories in the OHE

Where i := datapoints index

$$y_{ij} = \begin{cases} 1 & \text{if the element is in the class } j \\ 0 & \text{otherwise} \end{cases}$$

actual value from the dataset...

Good Bad Neutral

$y_{1j} = [y_{11} \; y_{12} \; y_{13}]$ i=1, j=1...c cathegory

$y_{1j} = [y_{11} \; y_{12} \cdots y_{1c}]$

$y_{2j} = [y_{21} \; y_{22} \cdots y_{2c}]$

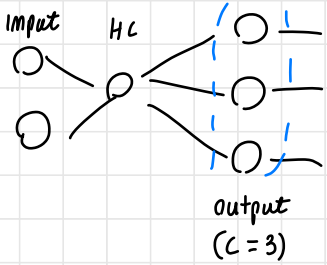
⋮

How I compute $\hat{y}_{ij}$? (the PREDICTION) → in a multi-class classification the Activation Function applied is Softmax activation ≃ $\text{Soft}(z) = \frac{e^{z_i}}{\sum_{j=1}^{c} e^{z_j}}$

therefore

...

Softmax activation function in output



output:

$\hat{y}_{ij} \in [0,1]$:= probability of data i to be of class j

$\Rightarrow [0.2, 0.3, 0.5]$

categorical cross entropy output...

If $c = 5$ I get $[0.1, 0.2, 0.3, 0.2, 0.2]$ for example.. $\left(\sum_{j=1}^{c} \hat{y}_{ij} = 1\right)$

↑ probability of belonging to category j

⇓

this also gives the probability of other categories

(this differentiate from Sparse..)

SPARSE CATHEGORICAL CROSS ENTROPY

Disadvantage: Losing info about the probability of other category

Still probabilities output.. $[0.2, 0.3, 0.5]$ (0st, 1st, 2nd) category

highest index...

⇓

2nd Index output

We focus just on the class with the highest probability (NOT in the overall)

from $c = 5$ $[0.2, 0.3, 0.1, 0.2, 0.2]$ (0, 1st, 2nd, 3rd, 4th)

↑ final output tell me $\mathbb{P}_{max}$ is given by 1st prob.

this differentiate from Categorical standard...

To know the overall probability on all classes we rely on categorical cross-entropy...

To get only the highest probability output you rely on Sparse one

WHICH LOSS FUNCTION TO USE...

What about Right Combination? of the problem statement and ANN architecture...

Hidden Layer	activation → Output Layer	Problem statement	Loss function
ReLU or its variants	Sigmoid	BINARY CLASSIFICATION	Binary Cross Entropy
ReLU or its variants	Softmax	Multi-Class Classific.	Cathegorical / Sparse Cross Entropy (CE)
ReLU or its variants	Linear (specific activ.)	Regression	MSE, MAE, Huber loss, RMSE

Used to solve Regression

{activation function applied in output layer}

... optimizers, weight initialization, ... other components to describe in DL..

OPTIMIZERS

→ once defined an computed the COSS FUNCTION both in Regression / classification problems

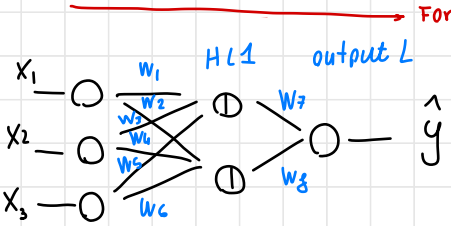
How the weights are updated to minimize the loss? optimizers are responsible of that

- Gradient Descent
- stochastic Gradient Descent (SGD)
- Mini Batch SGD
- SGD with Momentum
- Adagard and RMSPROP
- Adam optimizer

→ start optimizers description...

GRADIENT DESCENT OPTIMIZER

Optimizers are responsible of reducing the loss function! Foundamental part



main aim to reduce loss !
↳ such that predictions will match the output

Loss → reduced thanks to optimizers

↙ optimizers reduce loss Function during backward propagation

optimizer will update all the weights

Weight Update

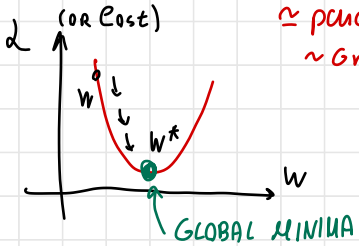
$$W_{new} = W_{old} - \eta \frac{\partial \mathcal{L}}{\partial W_{old}}$$

each optimizer update the weights in a different way... different weight update

η = learning rate ≈ speed of convergence to GLOBAL MINIMUM

⇓

plotting $\mathcal{L}(w)$ after Forward



≃ parabola curve
~ Gradient Descent Curve

during weight update W changes such that W reaches W^* minimum with respect to different number of epochs

⇓

When calculate $\frac{\partial \mathcal{L}}{\partial w}$ @ global minima $\frac{\partial \mathcal{L}}{\partial W_{old}} = 0 \rightarrow W_{new} = W_{old}$ CONVERGENCE to optimal point

• What does Gradient Descent do?

Considering MSE Loss := $(y-\hat{y})^2$ Cost func = $\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2$

related to COST FUNCTION (Not Loss)

In GRADIENT DESCENT:

two foundamental concepts are { Epochs, Iterations

forward ⟶ $\hat{y}$ ⇒ cost function.. take more data points $\vec{x}_i$ $(\vec{x}_1 \dots \vec{x}_n)$
compute $\hat{y}$ $(\hat{y}_1 \dots \hat{y}_n)$

Having 1000 Datapoints in Dataset

1 Epoch { 1000 Datapoints ⟶ $\hat{y}_i$ $(i=1\dots1000)$ ⇒ cost function C

backward propagation
weights update ⊗

than to Reduce this we rely on GRADIENT DESENT optimizer

{ one forward propagation with all datapoints + backward + weights update } = 1 Epochs

continue this process until $C\downarrow\downarrow$ reduce enough

2nd Epoch { ⟶ ⟵⊗ }

100 Epoch ⟶ ⟵⊗

perform multiple epochs... → in each epoch $C\downarrow$ updating weights...

GRADIENT DESCENT change the weights with same weight update formula

In any Epoch ⟵ ⊛ $C\downarrow\downarrow$ decrease...

Gradient Descent optimizers works in this way!

Iteration

∀ Epoch I perform back-propagation ... In gradient descent as defined above 1 epoch = 1 iteration

If Instead I divide datapoints in 10% batch

$Data_1$ = 100 records = 1000 · 0,1
$Data_2$ = 100
:
$Data_{10}$ = 100

1 epoch = ~~10~~ iterations

NOT in Gradient Descent, where you take all dataset in each epoch

What are PROs and CONs ?

Advantage

1) Convergence will happen (due to how defined Wnew)

Disadvantage

1) you require huge resource RAM, GPU since you need to compute Forward for all datapoints ! ↓ huge RAM to store and update all !....

NOT good in Limited computational resources system...
Risource intensive !

STOCHASTIC GRADIENT DESCENT (SGD)

Optimizer

This helps solving the "Resource Intensive" problem.

by addressing EPOCHS / ITERATIONS

in STOCHASTIC

Epoch 1 { ITERATION 1 — 1st data-point → $\hat{y}$ → Loss function ↓ minimized by SGD optimizers; update weights

ITERATION 2 — 2nd data-point → $\hat{y}$ → $\mathcal{L}$

⋮

ITERATION n — n^{TH} data ...

⋮

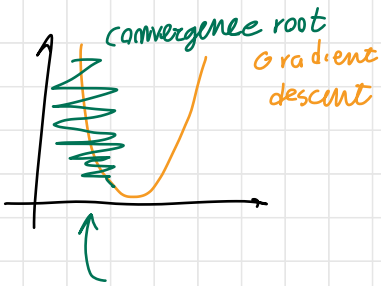
Epochs Ne ← we can have as many epochs until Loss value is satisfactory

Advantage

- Now even if limited RAM, GPU, you don't allocate all simultaneously...
  SOLVE Resource Issue

Disadvantage

- Time complexity is Huge! (take lots time to finish..)
  1000000 datapoints × 100 epochs... take LOTS of time
- convergence will take more time
- NOISE GETS INTRODUCED



as Disadvantage of SGD

While classical GD smoothly converge towards minimum, since you use entire RECORDS of dataset, guarantee smoothing

↓

while in SGD taking single data points at time, you update less smoothly

due to the SINGLE data usage

Time complexity increases + introduce noise

↓

Model training and convergence takes time...

↓

HOW to REDUCE this noise? → concept of BATCH

↙

Instead of taking single datapoints you date batch of datapoints and perform forward/backward propagation

The noise can be easily reduced through the concept of another optimizer

MINI BATCH SGD Optimizer

↑

while SGD is very slow, with lots of iterations and noisy update since each iteration take one data

"Noisy" because noise in the weight convergence...
Path not smooth
(lot of ZIG ZAG towards minimum)

↳ In Mini Batch SGD with Epoch and Iterations we introduce BATCH and Batch-size

having # Data points = 100 000 and batch-size = 1000

In Epoch 1: Iteration 1 → 1000 → Cost func ↙ $\sum_{i=1}^{1000} (y_i - \hat{y}_i)^2$ ← 1000 ← optimizer Mini batch SGD (MSE) ↓↓

optimize weight ← changing it

∀ epoch I take batch-size and use it in each iteration

iteration = $\frac{\text{Data points}}{\text{batch size}} = \frac{10^5}{10^3} = 100$

since with 1 data - iterat I have lot of NOISE... this concept of batch size help to reduce the noise

Iteration 2 → 1000 → $C = \sum_{i=1}^{1000} (y_i - \hat{y}_i)^2$ ← 1000 ← Mini batch SGD

⋮

Iteration 100

↑ I use cost function w.r.t the batch size data

$$C = \sum_{i=1}^{\text{batch}} (y_i - \hat{y}_i)^2$$

Epoch 2 Iteration 1

⋮ ⋮

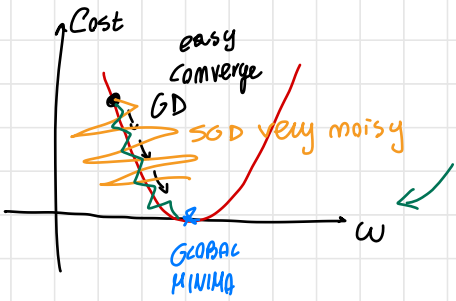
Iteration 100

⋮

With this approach... Having 8GB RAM,
with 1000 data-points it will NOT be a problem in resources

While 5000 batch-size RAM↑↑

With batch-size ↑↑ the # iteration ↓↓ and Reduce the NOISE!



thanks to batch of batch SGD the noise get reduce
⇓
No more single record at each iteration! ⇒ reduce noise towards min.

Advantages

- Convergence speed increase w.r.t SGD (thanks to batch size)
- Noise will be less compared to SGD
- It solves efficency resource problem efficent resource usage (RAM)

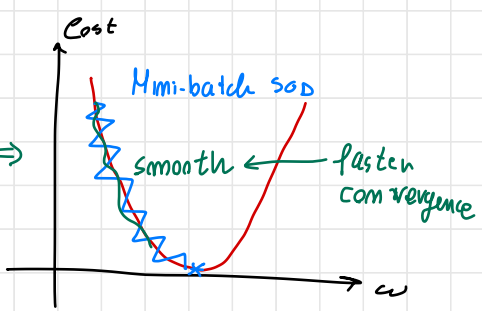
Disadvantage

- The Noise still exists! still slow down a bit convergence

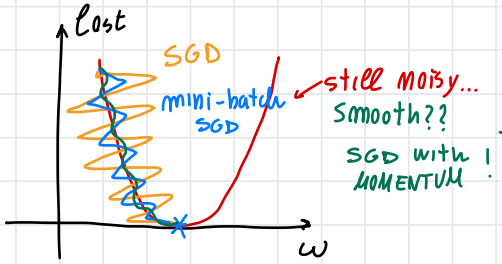
↓

the next objective is to smooth the noise help to reach global minima faster

this is achieved by SGD with MOMENTUM (smoothening) ⇒



SGD with MOMENTUM Optimizer



How the smoothening can happen?

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

↑ Change this formula to incorporate momentum (smoothness)

Also, during back propagation the bias gets updated

$$b_{new} = b_{old} - \eta \frac{\partial L}{\partial b_{old}}$$

rewrite in time series

Weight update formula $W_t = W_{t-1} - \eta \frac{\partial L}{\partial W_{t-1}}$

by using time dependency we can understand how to smooth

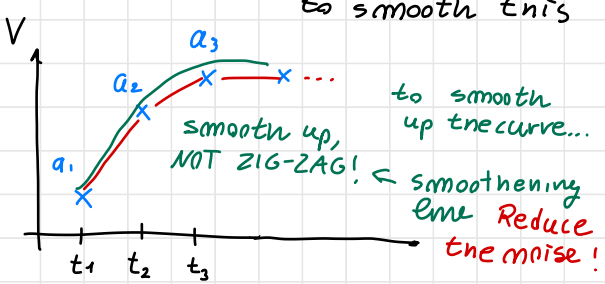
to guarantee smoothening, we rely on...

EXPONENTIAL WEIGHT AVERAGE (SMOOTHENING)

Time:	t_1	t_2	t_3	t_4 ...	t_n
Value:	a_1	a_2	a_3	a_4 ...	a_n

↖ this is a "time-series" problem... we want to smooth this

⇓ Exp Weighted Average } $V_{t_1} = a_1$ smoothening



$$V_{t_2} = \beta V_{t_1} + (1-\beta) a_2$$

β CONTROL the smoothening function ex. $\beta = 0.95$ $\beta \in [0,1]$

$$V_{t_2} = 0.95 * a_1 + 0.05\, a_2$$

give control to the a_1 value ← a_2 has less effect

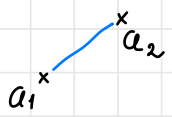
With this $\beta \sim 1$ the value a_1 control MORE, if $\beta \gg 0.5$ a_1 control the smoothening

Exp weighted avg allow smoothening → REDUCE THE NOISE!

$V_{t_1} = a_1$

V_{t_2} ... NOT a_2 ← you smooth giving weight to a_1/a_2

⋮



the β value control how the smoothness is controlled by current - previous point...

↓

If β is big, a previous control more..

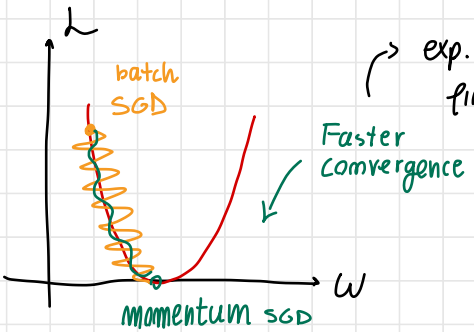
at same way $V_{t_3} = \beta(V_{t_2}) + (1-\beta)a_3 = 0.95[0.95\,a_1 + 0.05\,a_2] + 0.05\,a_3$

$V_{t_2} = \beta V_{t_1} + (1-\beta)a_2$

↑ the second point control smoothening of third point

↑

this Expomential weighted average can be used on $\partial L/\partial W_{t-1}$ → smoothening the entire curve



→ exp. weighted average helps in finding out the weights... its introduced in weight update

$$W_t = W_{t-1} - \eta \frac{\partial L}{\partial W_{t-1}}$$

∀ chain Rule we apply exp. weighted avg

smoothening guarantee ⇓

Advantage

- Reduces noise
- Quick convergence

this exp weighted avg happens With techniques as ARIMA, SARIMAX... for time series

- importance of Learning rate η ? $\Rightarrow$ we want a dynamic Learning rate...

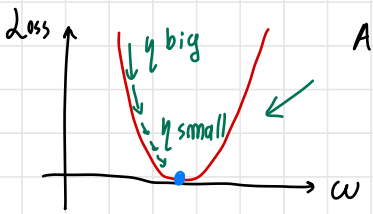
ADAGRAD: ADAptive GRADient Descent Optimizer

until now $W_t = W_{t-1} - \eta \frac{\partial L}{\partial W_{t-1}}$ update weight

↳ learning rate $\in [0,1]$ slow speed of convergence, usually initialized small at beginning

η = ~~FIXED~~ during convergence in GD, SGD...

↑ DYNAMIC ? what if η is dynamic?



As the convergence happen the learing Rate should change!

⇑

initially η big, fast, than we reduce η when close to min

ADAGRAD make η_t dynamic controlling convergence speed

$W_t = W_{t-1} - \eta' \frac{\partial L}{\partial W_{t-1}}$

η' : dynamic LEARNING RATE (Not Fixed!)

⇓

$$\eta' = \frac{\eta}{\sqrt{\alpha_t + \varepsilon}}$$

ε := small value (NEVER $\varepsilon = 0$, even when $\alpha_t = 0$ is fine!) $\alpha_t + \varepsilon \neq 0$ always

$$\alpha_t := \sum_{i=1}^{t} \left(\frac{\partial L}{\partial W_t} \right)^2$$

as back propagating w.r.t t, $\alpha_t \uparrow\uparrow$ therefore $\eta' \downarrow\downarrow$

at beginning α_t is small and η' is big, than opposite...

η' get decreasing with a finer small step convergence

$t=1$	$t=2$		$t=K$
$\eta=0.01$	$\eta=0.005$	...	$\eta=0.0001$
initialize	decrease		

↳ as Advantage you get dynamic η ...

but Disadvantage in a very deep NN, $d_t \uparrow\uparrow$ and $\eta' \downarrow\downarrow$ very small!

- η' can became a very small value approximatively equal to 0
  $W_t \approx W_{t-1}$ in this bad case... Weight update doesn't occurs

↑ this disadv. is addressed by a new optimizer...

ADADECTA and RMSPROP optimizers

While ADAGRAD uses a dynamic Learning rate $\eta' = \eta / \sqrt{d_t + \varepsilon}$ ← $d_t \uparrow$ with chain rule and $\eta' \downarrow$ keep decreasing dynamically

$$\sum_{i=1}^{t} \left(\frac{\partial L}{\partial w_t}\right)^2$$

but in Deep Layered Neural Network $d_t \uparrow\uparrow$ since chain rule is long...
$\Sigma \frac{\partial L}{\partial w_t}$ on entire depth... $\Rightarrow \eta' \downarrow\downarrow \to 0 \Rightarrow w_t \approx w_{t-1}$ No weight update!

To fix the problem on $\eta' \to 0$ converging to 0 and weight update stucked... New formula

$$\eta' = \frac{\eta}{\sqrt{Sdw_t + \varepsilon}}$$

exponential weighted average SMOOTHENING the update of d_t

$$Sdw_t = \beta \cdot Sdw_{t-1} + (1-\beta)\left(\frac{\partial L}{\partial w_{t-1}}\right)^2$$

IF $\beta = 0.95$, even if $\partial L/\partial w_t \uparrow\uparrow$ restricting by $(1-\beta)=0.05$ I restrict Sdw_t ← restricting $\partial L/\partial w_{t-1}$ with β...

With $Sdw_t = 0$ at beginning .. then update

AdaDelta and RMSPROP has dynamyc learning rate η

Exponential Weighted Average

Restricting α_t to be small $\alpha_t \equiv Sdw_t$ + smoothening with EWA

AdatDelta / RMSPROP ≡ Dynamic LR + smoothening EWA

two optimizers with slight differences but same concept

again same update

$$W_t = W_{t-1} - \eta' \frac{\partial \mathcal{L}}{\partial W_{t-1}} \qquad \eta' = \frac{\eta}{\sqrt{Sdw + \varepsilon}}$$

still this technique doesn't have momentum in weights etc... combination of more advantages

⇒ Adam Optimizer bring many more advantages considered very good

ADAM OPTIMIZER

It collect all the advantages of previous discussed optimizers

Architecture..

SGD with MOMENTUM ⊗ {less noise} + RMSPROP {dynamic LR + smoothening} (so α_t never too big)

Adam optimizer mix this two features...

Weight update

weight update

$$\begin{cases} W_t = W_{t-1} - \eta' V_{dw} \\ b_t = b_{t-1} - \eta' V_{db} \end{cases}$$

as SGD with Momentum you do EWA

bias update

(major updates) dynamic learning rate

$$\eta' = \frac{\eta}{\sqrt{Sdw_t + \varepsilon}}$$

(Dynamic LR) ⊗

$$\begin{cases} Sdw_0 = 0 \text{ initialize} \\ Sdw_t = \beta \cdot Sdw_{t-1} + (1-\beta)\left[\frac{\partial \mathcal{L}}{\partial W_t}\right]^2 \end{cases}$$

EWA

SMOOTHENING (by Sdw_t) ⊗

$$V_{dw_t} = \beta V_{dw_{t-1}} + (1-\beta)\frac{\partial \mathcal{L}}{\partial W_{t-1}}$$ smoothening of weights

(EWA)

$$V_{db_t} = \beta V_{db_{t-1}} + (1-\beta)\frac{\partial \mathcal{L}}{\partial b_{t-1}}$$ bias smoothening

this reflect SGD with MOMENTUM

Momentum

↝ All combined in weight update formula and this V_{dx_t} allow MOMENTUM → smoothening weight update

also RMSPROP with smoothening is applyed on Adam rely on EWA

EWA updates in weights/bias update

Adam optimizer helps in weight convergence quickly, since it mix the best properties

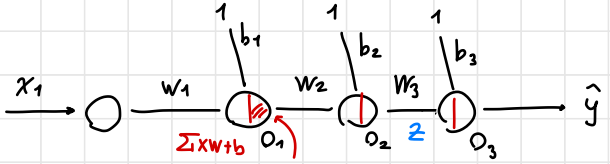
↓

quick convergence + batch size + learning Rate dym + smoothening

best in MOST of use cases!!

EXPLODING GRADIENT PROBLEM

↖ It is related to the weights and can be "bad"...



Activation .... e.g ReLU

here BINARY CLASSIFICATION, one output.. sigmoid

Loss function ↓↓
Optimizers
↗ train loss and back propagate...

⇓ to update

$$w_{1\,new} = w_{1old} - \eta \left| \frac{\partial L}{\partial w_{1old}} \right|$$

↘ where solving the $\partial L / \partial w$ using chain Rule

$$\frac{\partial L}{\partial w_{2old}} = \frac{\partial L}{\partial o_3} \cdot \frac{\partial o_3}{\partial o_2} \cdot \frac{\partial o_2}{\partial o_1} \cdot \frac{\partial o_1}{\partial w_{2old}}$$

↓ another chain rule..

As next step...

$z = o_2 w_3 + b_3$

then Activation function applied on z $\sigma(\cdot)$ sigmoid

$$\frac{\partial o_3}{\partial o_2} = \underbrace{\frac{\partial \sigma(z)}{\partial z}}_{\in [0, 0.25]} \cdot \underbrace{\frac{\partial z}{\partial o_2}}_{= w_3} = [0 \div 0.25]\, w_3$$

↑ vanishing previously was related to $\partial \sigma / \partial z$... now not issue

← ✓ VANISHING ✓ :)

What if w_3 is BIG

$w_3 = 100 \div 1000$ ... in any part of chain Rule because weight initialization ⇒ high value

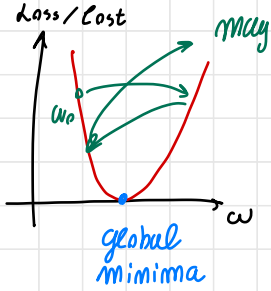
$$\frac{\partial L}{\partial w_{2old}} = \frac{\partial L}{\partial o_3} \cdot \frac{\partial o_3}{\partial o_2} \cdot \frac{\partial o_2}{\partial o_1} \cdot \frac{\partial o_1}{\partial w_{2old}}$$

high high high high ← big numbers..

↑↑ with big wi initialization...

big value! $w_{inew} = w_{iold} - \eta \frac{\partial L}{\partial w_{iold}}$ ← big value

When this update by big step happen ← { $w_{inew} \gg w_{iold}$ or $w_{inew} \ll w_{iold}$



maybe it diverge! and never reach mimimum

EXPLODING GRADIENT DESCENT

due to weight initialization problem!

IF w_i are too big ...

⇓

different w_i initialization techniques allow to avoid the EXPLODING GRADIENT PROBLEM!

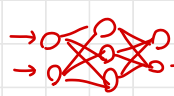
↙

weight initialization techniques allow to guarantee convergence of weights (scientifically proven)

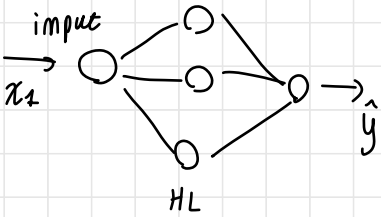
WEIGHT INITIALIZATION TECHNIQUES

1) Uniform Distribution
2) Xavier / Glorot Initialization
3) Kaiming He Initialization

Key Points

- Weights should be small ← if big you face gradient explosion
- Weights should Not be same ←  $w_i \neq w_j$ and small! otherwise they "think" in the same way and does the same processing ↓ you should give different processing with different weights
- Weights should have good variance ↖ $w_1, w_2 \ldots$ variance changes as different with big diff

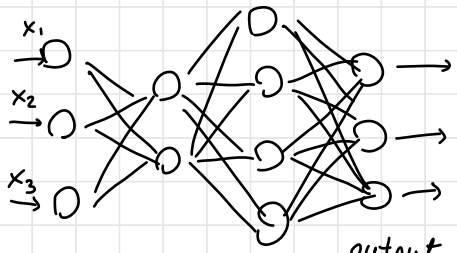
Important from one input $x_1 \rightarrow$ 3 output $\rightarrow$ then one output $\hat{y}$



$\begin{cases} \text{input} = 1 & x_1 \\ \text{output} = 1 & \hat{y} \end{cases}$

Multi Layer

IF multiple input features



input = 3
output = 3

output
(multi class)
↑
SoftMax activation

UNIFORM DISTRIBUTION

In weight initialization $w_1, w_2, w_3, \ldots$
we initialize it with UNIFORM DISTRIBUTION

$w_{ij} \sim$ Uniform Distribution (a, b) = Uniform Distribution $\left[-\frac{1}{\sqrt{\text{input}}}, \frac{1}{\sqrt{\text{input}}}\right]$

i: input layer, j: hidden layer

inside range a, b

$a = -1/\sqrt{\text{input}}$
$b = 1/\sqrt{\text{input}}$

input = # input features

e.g if input = 3 $\quad w_{ij} \sim$ Uniform $\left[-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right]$

XAVIER/GLOROT INITIALIZATION

(avoid exploding gradient and proof the validity...)

Invented by Researcher → Xavier Glorot

1) Xavier Normal Initialization

$W_{ij} \sim \mathcal{N}(0, \sigma)$ Normal distribution

with $\sigma = \sqrt{\frac{2}{input + output}}$

2) Xavier Uniform Initialization

$W_{ij} \sim$ Uniform Distribution (a, b)

$a = -\frac{\sqrt{6}}{\sqrt{input + output}}$ $b = \frac{\sqrt{6}}{\sqrt{input + output}}$

KAIMING HE INITIALIZATION

1) He Normal initialize.

$W_{ij} \sim \mathcal{N}(0, \sigma)$

$\sigma = \sqrt{\frac{2}{input}}$

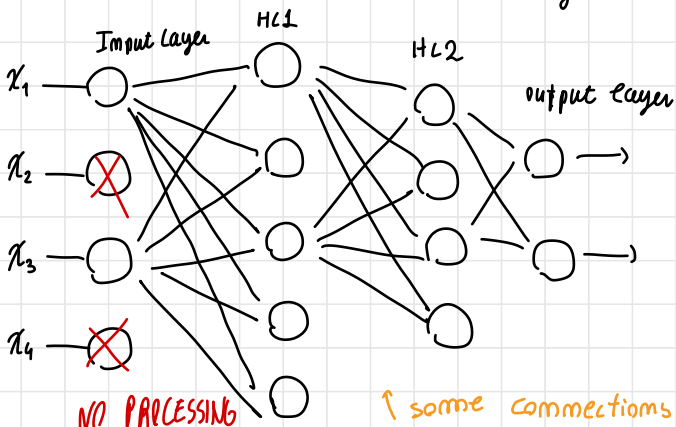
2) He Uniform initialize

$W_{ij} \sim$ Uniform Distribution (a, b)

$a = -\sqrt{\frac{6}{input}}$ $b = +\sqrt{\frac{6}{input}}$

DROPOUT LAYERS

Multi layered NN



NO PROCESSING skipp this feat

↑ some connections are MISSING
it is on purpose as dropout

it is related to OVERFITTING ⇒ When creating a deep learning model, accurate during training...
but than in testing data it performs very BAD!

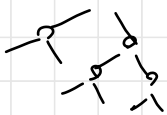
Model) training accuracy ~ 90%.
Test accuracy ~ 60% <<training

← the model has overfitt! we want a generalized model

we should focus on generalizing the model ← that performs well also in test data

°In machine learning algorithms such as Decision tree
created to its depth...

D.T. ... leads to OVERFITTING..
we do pruning technique OR Random Forest



Im Random Forest → for every decision tree we do Feature sampling
↑ from all dataset I take only a sub-set of features
and Raw sampling we get only some datapoints...

so Im Random Forest we create subset of many decision trees built on its depth with diffent data subset...
↓
during construction, the output of DT is compared and the max occurence is output... create generalized model with Random Forest

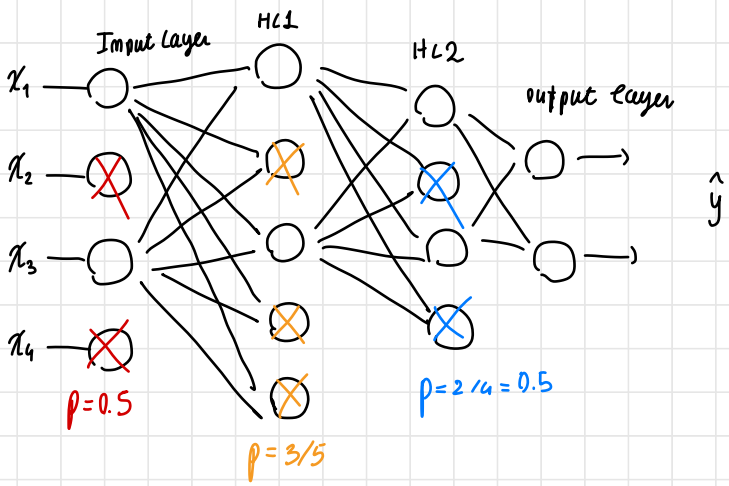
↳ similarly im a MLNN with MANY WEIGHTS... many w_{ij}, b_{ij} initialized on hidden layers / neurons...
⇓
the ANN tends to overfit, if fully connected!

to address OVERFITTING you rely on DROPOUT LAYER

we define a DROPOUT RATE $p \sim 0.5$ (e.g) $(0 \leq p \leq 1)$

we define the dropout probability of a specific layer (input/hidden)

e.g. $p_{input} = 0.5$ ← Feature sampling (subset of features (input) are used)
with 50% prob. you ignore the node..
↓
with 4 input ~ 2 node are discarded during first epoch



3/5 of neurons are skipped → this p value must be tuned!

during FORWARD PROPAGATION. based on p some neurons are Deactivated than in Backpropagation this steps happen again ⇝ neurons may be update

In new Iteration / Epoch (next Forward propagation) the new dropout neurons change → randomly selected

↓ this continue until loss satisfactory and weight converges

purpose: deactivate some neurons / features s.t. model does not overfit

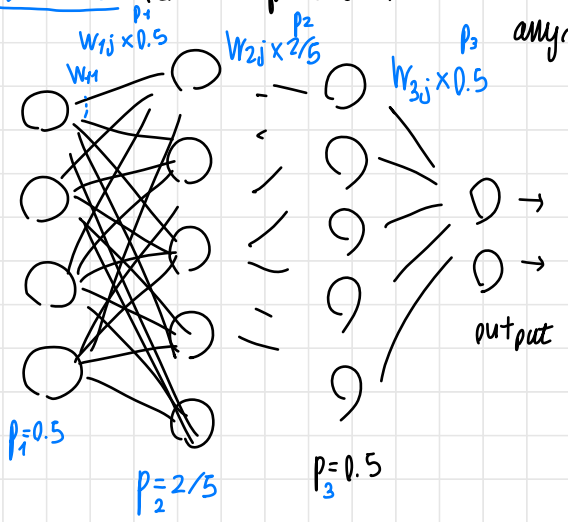
↑ (during training) ↓ after fixing all weights / bias

than for testing you CONNECT all (Fully connected)

(the probability of dropout $p \in [0,1]$ depends on hyperparameter)

Test data (do the prediction)

the dropout doesn't hold anymore for testing



in test data we don't apply dropout, we need to predict.. instead we multiply all weights by prob.

No deactivation... to connect the NN you multiply all weights by probability initialized before

CNN INTRODUCTION

covered ANN, Optimizers, Activation Functions... → used in CNN theory

Remember...

- ANN → supervised Learning { Classification / Regression }

↓

Dataset: Inputs features (numerical / cathegorical values) — Output feature

↳ Im

CNN — Input: Images → you can solve many different problems

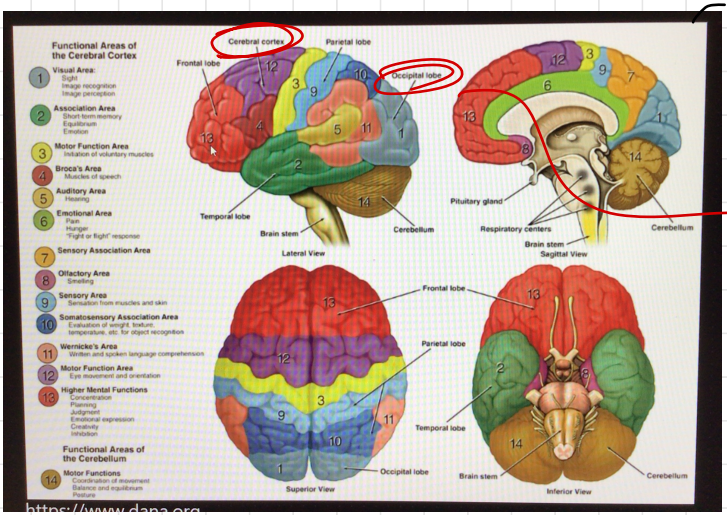
- Image classification
- Object Detection
- segmentation
- ...

How does a CNN works?
How the CNN process the data? take images and solve complex problem statements

CNN take images as INPUTS → it's working principle is influenced on how human brain works w.r.t images

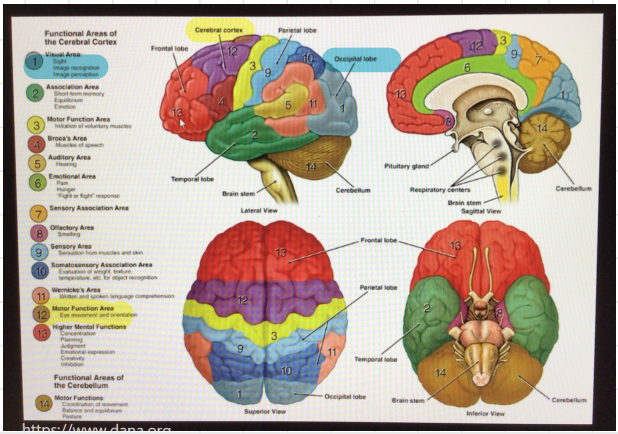
↓

at the high level:



→ Important parts of the brain that NN try to emulate

→ for CNN we focus on { • cerebral cortex • occipital lobe }



⇐ thes two parts
- CEREBRAL CORTEX
- OCCIPITAL LOBE

have sub-tasks responsible for different stimulae

1) Occipital lobe ⇒ Visual Area { sight, Image recognition, Image perception

12) Cerebral cortex ⇒ Motor Function Area { Eye movement and orientation

↑ process in different layers to overall recognize pattern in images (12) take the image, (1) process it

these two parts are replicated in CNN

(12) pass signals to Visual cortex → subdivide layers of image than in (1) the image is analyzed in details..

⇑

Also other brain functionalities are applied in Robotics idea

(12) the eye see something, pass signals to multiple layers in the VISUAL CORTEX

↓

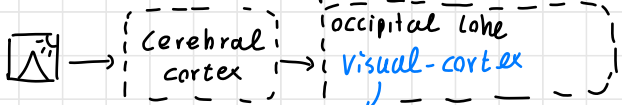
each signal is then processed an passed on (1) to extract and visualize details

(1) here the image is visualized + processed etc!

HUMAN BRAIN and CNN

Cerebral Cortex and Visual Cortex (inside occipital lobe) → this help to understand the working primaple of CNN

- How a human brain process a visual image ?



VISUAL CORTEX (V1-V5) : Region of the brain that receive, integrates and processes visual information relayed from the retinas

{ divided into 5 layers }

⇓

signal (retina)

↓

V1: primary Visual Cortex (orientation of Edges and Lines)

im V1 the edges/lines are processd/analyzed

{ PROCESSING is going on }

V2: Differences in color (red, green?), complex patterns, Object orientation

V3 V4 V5 Object Recognition Detection ....

(round · squared · shape.. ?)

⇓

Visualize the image

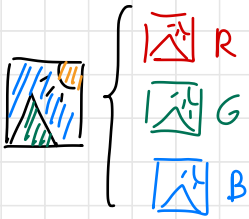
signal goes from the retinas to various layers and get processed... finally after V3-V4-V5 ⇒ we visualize the image characteristics

↳ Im CNN this functionality is replicated! ↑ pre-processing and peform this task im a computational way

each layer of network CNN process signal using FILTERS to extract features (edges/lines...)

ALL YOU NEED KNOW ABOUT IMAGES

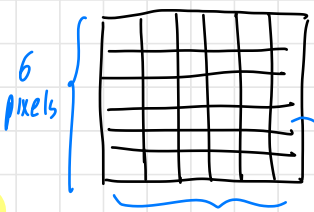
RGB Images and Grayscale Images



to understand how CNN works...
Images are divided in 3 channels, combined to get any colored image.

In the image representation you have

GRAYSCALE IMAGE



pixels division

6x6 with values in 0÷255

0÷255

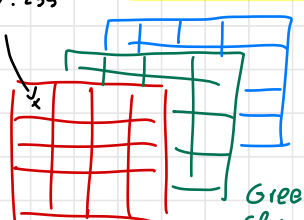
+ you specify the # channels

{ 6x6x1 := Grayscale
{ 6x6x3 := colored Image

you can extract a gray channel (B&W) of each channel

(8 bit) each value in 0÷255

COLORED IMAGE



6x6x3

overall when combined, it forms the colored image

↑ different treatment by a CNN if the image is Grayscale or Colored

CONVOLUTION OPERATION IN CNN

First type of operation in CNN

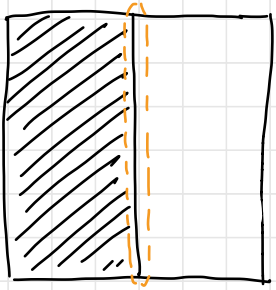
example:

0	0	0	255	255	255
0	0	0	255	255	255
0	0	0	255	255	255
0	0	0	255	255	255
0	0	0	255	255	255
0	0	0	255	255	255

6x6x1 image (grayscale)

image ⇒

0 := black 255 := white



convolution operation process the image and find useful information

↑

↑ "edge" between B&W

as VISUAL CORTEX has many layers... every layer process some information (V1 := edges orientation..)

The first step of CONVOLUTION is pixel normalization

STEPS

1) Normalize: convert all pixels in [0,1] range while help in the next processing steps

↓

Normalize as pixel / 255 (as maximum)

as dividing by 255 (max pixel intensity)

0	0	0	255	255	255
0	0	0	255	255	255
0	0	0	255	255	255
0	0	0	255	255	255
0	0	0	255	255	255
0	0	0	255	255	255

/255 ⇒

0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1

0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1

$*$

+1	0	-1
+2	0	-2
+1	0	-1

FILTER 3×3

$=$

0	-4	-4	0
0	-4	-4	0
0	-4	-4	0
0	-4	-4	0

Step 2: convolution operation

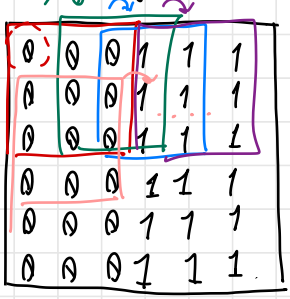
as the visual cortex process information in each layer (V1, V2...) in the same way CNN layered process occurs through FILTERS (it can be of different sizes)

Its main objective is to take out information from the image (edges horiz/vert., vertical plane)

6×6×1 img $*$ 3×3×1 Filter → 4×4 output by convolution operation

during convolution we take the 3×3 filter and slide it on top of 6×6×1 image

stride



$*$

1	0	-1
2	0	-2
1	0	-1

convolution 1 centered on first 3×3 block

→

$1\cdot0 + 0\cdot0 - 1\cdot0$
$+ 2\cdot0 + 0\cdot0 - 2\cdot0$
$+ 1\cdot0 + 0\cdot0 - 1\cdot0 = 0$

element wise product and summation..

→

0	-4	-4	0
0	-4	-4	0
0	-4	-4	0
0	-4	-4	0

next convolution 2 step you jump by one (STRIDE) on the right and multiply for the same filter

$1\cdot0 + 0\cdot0 - 1\cdot1 + 0\cdot0 + 0\cdot2 - 2\cdot1$
$+ 0\cdot1 + 0\cdot0 - 1\cdot1 = -4$

"stride" is a jump to right/left/bottom here STRIDE = 1

another stride 2 on step 3

than final stride on first "row" of the output and for next step I go to next row (NO less right stride)

What does this output represents??

normalized image

0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1

$*$

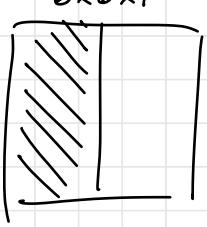
+1	0	-1
+2	0	-2
+1	0	-1

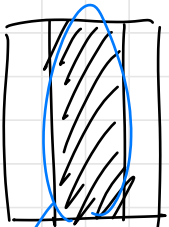
FILTER 3×3

$=$

0	-4	-4	0
0	-4	-4	0
0	-4	-4	0
0	-4	-4	0

6×6×1





4×4×1

this Filter 3×3 has find out the VERTICAL EDGE on original image "edge detection"

+1	0	-1
+2	0	-2
+1	0	-1

vertical edge filter

any image passed through this filter will get extracted the vertical edges (here B/W difference)

⇐

255	0	0	255
255	0	0	255
255	0	0	255
255	0	0	255

white black

↓ de-normalization based on min/max

here { 0 := 255 max
-4 := 0 min

original format

(NOT computed in CNN, here just for visual purpose)

→ in VISUAL CORTEX V1 layer has the responsability on edges orientation

↓

in the same way in CONVOLUTION OPERATION edges are extracted by FILTERS.

this behave as the V1 layer

similarly I can have any # of filters

- Round edge
- Horizontal edge
- ...

e.g Horiz Edge Filter

+1	+2	+1
0	0	0
-1	-2	-1

↖ extract horiz.

Here these filters are pre-defined

Usually in a CNN we don't define filters values... we randomly initialize these values

↓

Then through Forward and Back Ward propogation, the FILTERS values are updated

With different filter size and values... computed by different CNN techniques, (e.g in transfer learning)

SO... a convolution operation pass a filter through an image and you get an output that highlight image informations
(vertical/horiz/round edges etc..)

→ Notice that from 6×6 image size $n = 6$ and filter size f is 3×3 $f = 3$ while output size u $u = 4$

⇓

Usually a formula is valid on this input/output size

$$n - f + 1 = u \qquad (6 - 3 + 1 = 4)$$

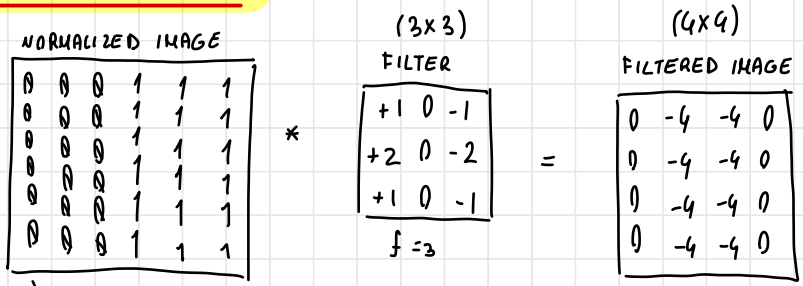
Be carefully from 6×6 image I get a 4×4...
loss of information! (less pixels/resolution)

↓

It is possible to prevent this loss of information during filtering by applying PADDING.

Convolution is used to process the image and find patterns that will help the model in its task

PADDING IN CNN



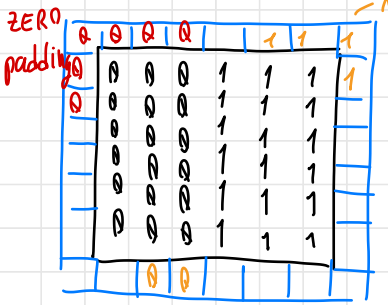
$n - f + 1 = 4$ (output)

here from 6x6 to 4x4 we lose some amount of informations...

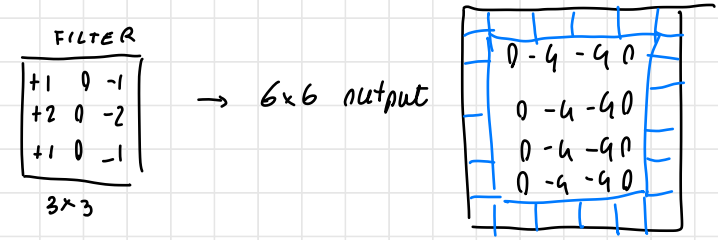
⇓

How to prevent it? to get as output same size as input we apply padding

padding := adding additional layer



here one layer you avoid loss of data by PADDING in CNN



↑ how much padding I need to apply? s.t at output you have same size of input

⇓

$n - f + 2p + 1 = \text{output}$ (desired)

$6 - 3 + 2p + 1 = 6 \rightarrow 2p = 2 \rightarrow p = 1$

padding of 1 layer is necessary

↳ what values you use in the padding??

1) ZERO padding := apply 0 in all the additional layer

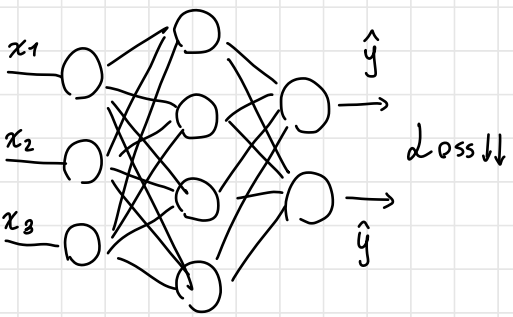
2) Neighbour padding := based on the close pixels, you use same value

Level of padding is computed based on desired output size
+ you need to decide also padding type (zero / neighbour)

e.g. input $n=7$ (7×7) filter $f=3$ (3×3) output desired $q=7$ (7×7)

$n-f+2p+1=q \Rightarrow 7-3+2p+1=7 \quad 2p=2 \rightarrow p=1$

OPERATION OF CNN vs ANN



In ANN (multi layered NN)
← 2 layered NN
(2 input, 1 HL, 1 output)

$z = W^T x_i + b$
$Act(z)$ (e.g. ReLU(z))
then backward, etc...

Similarly in a CNN

this is a (SINGLE) CONVOLUTION LAYER
↓
we can add more convolution layer in sequence on the CNN (horizontally)

NORMALIZED IMAGE

0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1

$*$

(vertical edge) FILTER f_1

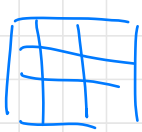
+1	0	-1
+2	0	-2
+1	0	-1

$=$

FILTERED IMAGE

0	-4	-4	0
0	-4	-4	0
0	-4	-4	0
0	-4	-4	0

f_2



I can have multiple filters...

→ when training CNN from scratch this filters are Randomly initialized...

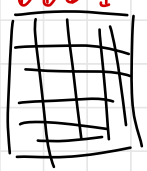
Image pass through the Filter...
as in the VISUAL CORTEX layered processing of information coming from the Retina.

(Random)
the Filter is applied and img output w.r.t 1 is computed

multiple fi: filters can be applied to extract patterns (shape/edges/...)

each filter play an important role in understanding the image ⇒ the initialized filters (Randomly) during training initialization

then with forward / backward propagation we update the filters values...

Image * filter =  output

↓↓↓↓ to all values ReLU is applied $\max(0,x)$ → to transform the values on the filtered image

⇒ when back propagate, filter values must be updated ... to update it the derivative should be computed... (make it differentiable) ↳ easily update filter weights (+avoid vanishing gradient with ReLU)

Same training / learning approach as ANN

Forward: we apply filters on input image

$$\underset{n \times n}{\text{img}} * \begin{matrix} f_1 \; f_1 \times f_1 \\ f_2 \; f_2 \times f_3 \\ \vdots \\ f_n \; f_n \times f_n \end{matrix} = \begin{matrix} out1 & n - f_1 + 2p_1 + 1 \\ out2 & n - f_2 + 2p_2 + 1 \\ & \vdots \\ outn & n - f_n + 2p_n + 1 \end{matrix}$$

on all values of output image we apply ReLU

↳ because on backpropagation we should be able to find the derivative... ⇐

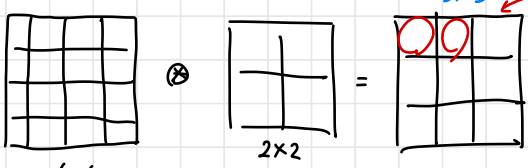
the differentiation is required so that the filters can be computed (learn parameters)

after convolution, ReLU is required (or other variation)
so that it derivable...
than other operations are applied after ReLU

allow the learning of filter based on the problem task,
proper output for classification/detection...

after convolution, ReLU → max pooling is applied
(average)

MAX POOLING, MIN POOLING, MEAN POOLING



ReLU on each output cell
(activation s.t in backward process this is differentiable and allow update)

$p = 1$ (padding = 1) $q = m + 2p - f + 1 = 4 + 2 - 2 + 1 = 3$

$S = 1$ (stride = 1)

CONVOLUTION LAYER

IF after convolution

max pooling ?
chose a filter size

ReLU output

1	2	3
4	5	6
2	8	4

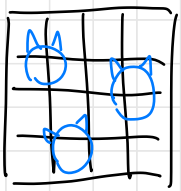
I apply Max pooling on top of the FILTER

this filter is applied on top of the output

max pooling := applyed for LOCATION INVARIANT

LOCATION INVARIANT

since we use multiple FILTERS to determine different edges / shapes...



I have 3 cat in img ⇒ location invariant

Any number of cat imgs present...
IF FILTER is responsible to find cat faces

↓

when Apply max pooling on output layer I can extract faces much clearly!

MAX POOLING

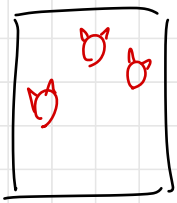
It concentrate completely on the part of interest to get output

allow to focus on the most important informations irrespective of the particular filter

e.g. multiple cars in image... given a FILTER that extract the wheels then max pooling focus on cars wheels to take out the specific information

+LOCATION INVARIANT → wherever those img shapes / features are present, max pooling can detect those parts

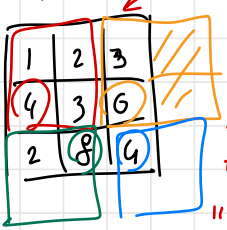
overall max pooling act in this way:



2×2

=

ReLU...



MAX Pooling

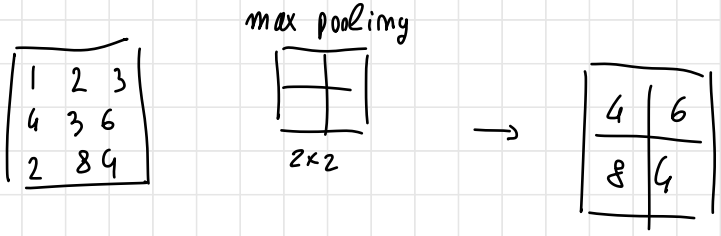
2×2

max pooling act as in the convolutions...

"MAX" pooling since it extract the max value in the filter size

stride of 2!

then make stride = 2, since it take max intensity (NOT all pixels)



highest intensity (most important feature) is extracted...

↓

output is more focused on the part of interest

In CNN architectures: convolutional layer + max pooling are stacked horizontally multiple times...

Input → convolution → max pooly → convolution → max pooly ....

applied to all output of the filtered image

other pooling techniques are MIN / MEAN POOLING

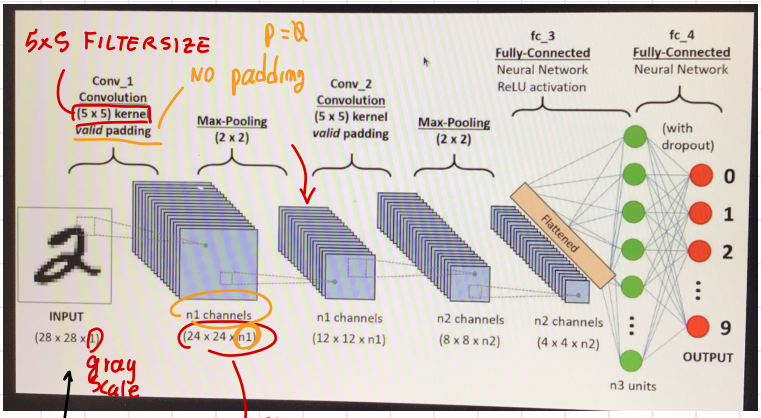
MIN: take the minimum element ← interest in the minimum intensity pixel available

MEAN: Average of all the pixel values

After all this CNN architecture combine convolution + max pooling to extract information

+ place as final layer a fully connected layer on the CNN output

FULLY CONNECTED LAYER IN CNN



gray scale

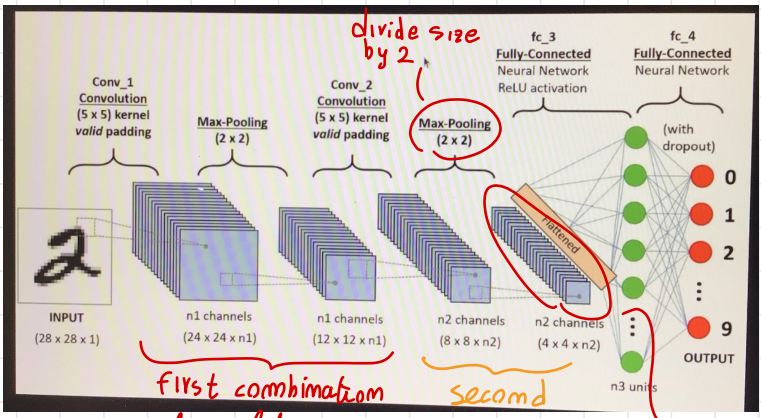
handwritten MNIST dataset 0-9 digits

after first convolution layer ... → then ReLU applied and max pooling 2×2 applied to all channel of 1... n1 and reduce img size → focus on LOCATION INVARIANT... extract more intensity path

With n1 channels meaning the amount of filters applied on the image

28-5+1=24 out of convolution layer

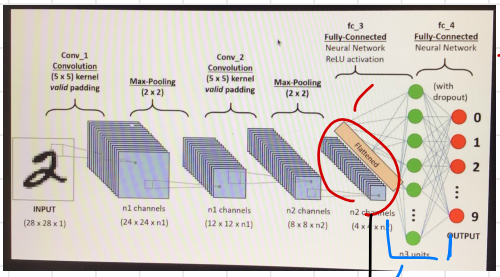
conv := filter + ReLU ... applyed horizontally in sequence



first combination of convolution and max pooling

second layer of conv + max pool

after all conv + max pooly sequence the FLATTENING LAYER is applied



FLATTENED layer

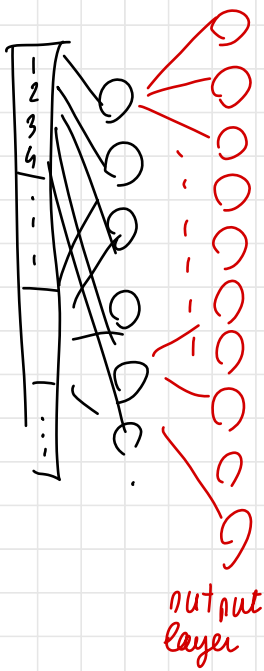
⊛ then the last piece is a fully connected layer (the same as ANN)

on the last output...

flattening

⊛ fully connected layer

then, depend on the problem e.g in MNIST you have 0÷9 (10 outputs) classes



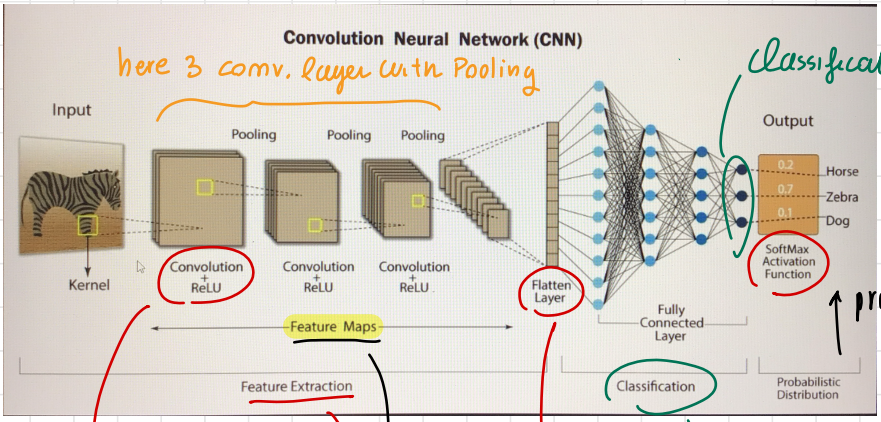
loss ↓↓ (optimizers)

→ prediction of class

so... Forward + backward update FILTERS VALUES (differentiable thanks to ReLU activ.)

output layer

CNN with RGB



here 3 conv. layer with Pooling

classification of 3 classes

probability in output

one convolution layer

flattening output

from flattening to MLP apply classification

define the process of feature

the entire process is the FEATURE EXTRACTION

Im forward propag... we pass through the cell Network...
then im backward you should be able to derivate... why you use ReLU

then loss $(\hat{y}-y)^2$ ... back propogate + optimizers...

updaty FILTERS
weight and ANN weights.