TUTORIAL 1: INTRO, ML BASICS, PERCEPTRON

BASICS:

- LEARNING FROM DATA:

  WE WANT TO LEARN FROM DATA BECAUSE WE DON'T KNOW HOW TO DESIGN A RULE FOR SOLVING THE PROBLEM.

  EXAMPLE:

  HOW TO DESIGN A RULE THAT TELLS IF AN IMAGE IS A CAT?

- FOR LEARNING, YOU NEED

  - DATA

    SHOWS EXAMPLES OF PROBLEM YOU CARE ABOUT

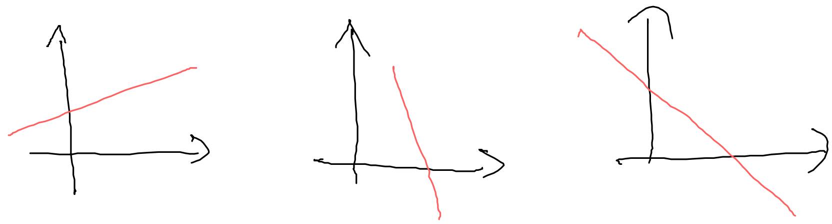
  - MODEL

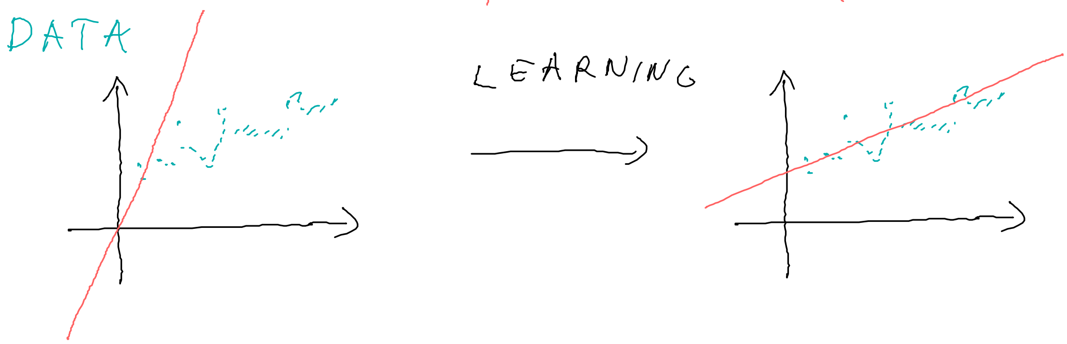
    A SET OF EQUATIONS / ALGORITHM THAT CAN REPRESENT AN APPROXIMATE SOLUTION FOR THE PROBLEM.

    ITS BEHAVIOUR IS CHANGED BASED ON DATA, TO BETTER MATCH THE PROBLEM WE CARE ABOUT. THIS CAN BE DONE THROUGH ITS PARAMETERS

EXAMPLE:

LINEAR REGRESSION: THE MODEL IS A LINE





- LOSS FUNCTION:

  SHOWS HOW GOOD/BAD THE CURRENT SOLUTION IS. EXAMPLE: MEAN SQUARED ERROR

- LEARNING ALGORITHM

  SHOWS HOW TO CHANGE THE MODEL PARAMETERS TO REDUCE THE LOSS.

  EXAMPLE: GRADIENT DESCENT

- TYPES OF OUTPUTS

  - CLASSIFICATION.

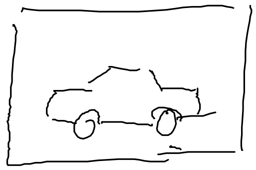
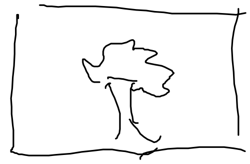
    DISCRETE: CAT OR DOG

  - REGRESSION

    CONTINOUS: PREDICT THE STOCK PRICE

- TYPES OF LEARNING:
  - SUPERVISED

    INPUT / OUTPUT EXAMPLES ARE GIVEN

     → CAR  → TREE

  - UNSUPERVISED

    NO LABELS ARE GIVEN, JUST RAW DATA.

    GOAL: DISCOVER REGULARITIES.

    USUALLY SOME FORM OF COMPRESSION

  - REINFORCEMENT LEARNING

    AGENTS ACT IN ENVIRONMENT

    ↑ EG: PLAYER ↑ EG: GAME

    AT SOME TIMESTEPS A SCALAR REWARD IS GIVEN TO THE AGENT.

    THE AGENT WANTS TO MAXIMIZE THIS REWARD, BY CHANGING THE ACTIONS IT TAKES.

TYPES OF PARAMETERS

- WEIGHTS
  USUALLY THE ONES OPTIMIZED BY GRADIENT DESCENT (GD)
- HYPERPARAMETERS
  NOT OPTIMIZABLE BY GD. USUALLY OPTIMIZED BY SOME FORM OF RANDOM SEARCH.

  EXAMPLE: LEARNING RATE, TYPES OF NONLINEARITIES, NUMBER OF UNITS, NUMBER OF LAYERS

GENERALIZATION:

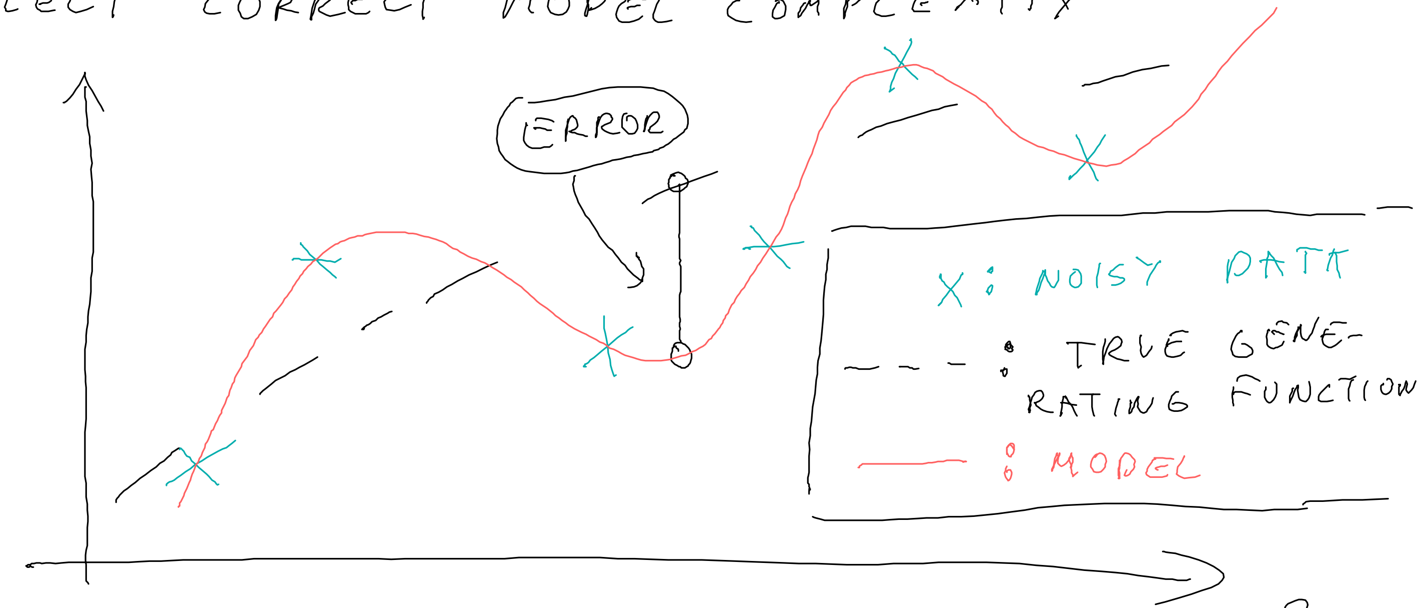
WE WANT OUR MODEL TO WORK FOR EXAMPLES NOT IN THE TRAIN SET, BUT SIMILAR. THIS IS GENERALIZATION

UNDERFITTING / OVERFITTING

- PROBLEM: THE DATA IS ALWAYS NOISY OR INCOMPLETE. BECAUSE IT IS FINITE, WE CANNOT KNOW WHAT IS THE NOISE AND WHAT THE INTERESTING DATA.

WE WANT TO REDUCE THE MODELS ABILITY TO REPRESENT IRRELEVANT REGULARITIES

- SELECT CORRECT MODEL COMPLEXITY



DO YOU THINK THIS IS A GOOD MODEL?

THIS IS OVERFITTING

- TRADEOFF

  WE WANT HIGH CAPACITY MODELS TO BE ABLE TO REPRESENT THE DATA
  WE WANT LOW CAPACITY TO PREVENT OVERFITTING

- SOLUTION

  USE REGULARIZERS

  EXAMPLE : L1 OR L2 LOSS ON WEIGHTS (WEIGHT DECAY)

  MEASURE OVERFITTING AND STOP WHEN PERFORMANCE DEGRADES (EARLY STOPPING)

  USE ARCHITECTURAL BIASES

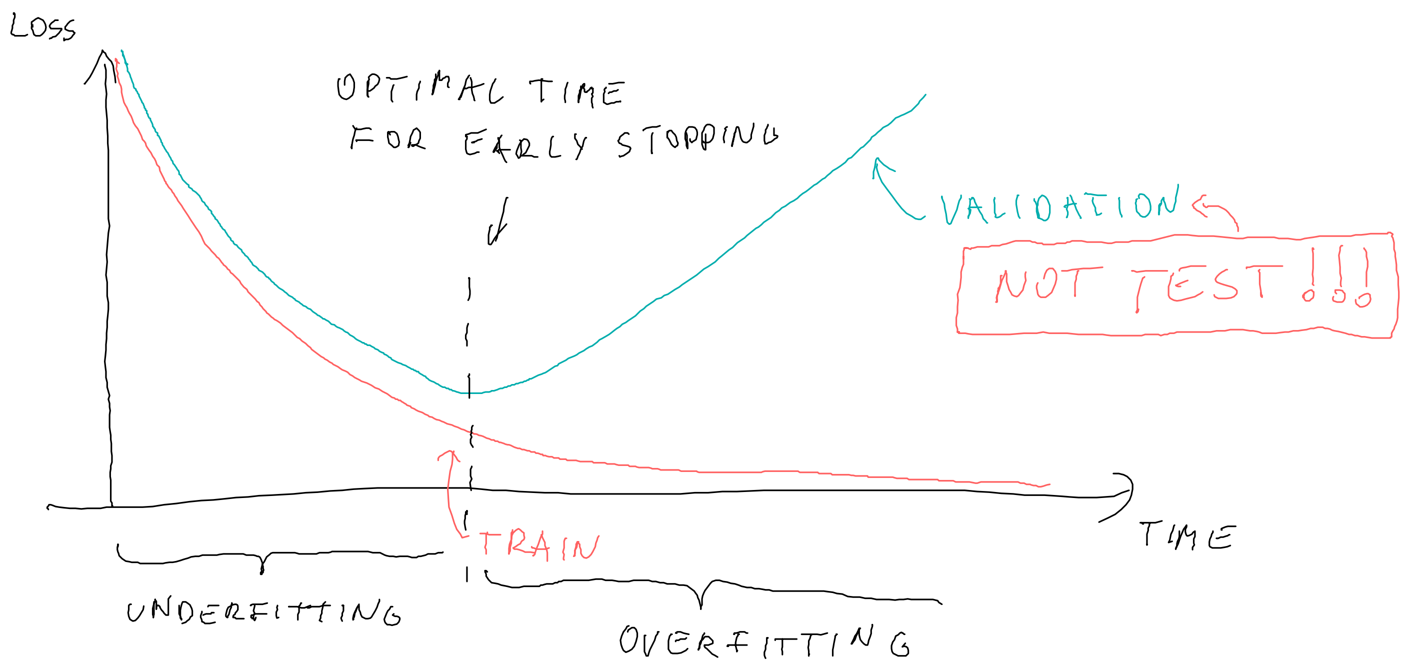
  EXAMPLE : CONVOLUTION FOR IMAGES

- MEASURING NETWORK PERFORMANCE CALLED VALIDATION

  DATASET IS DIVIDED TO 3 PARTS:

  - TRAIN SET - USED FOR TRAINING THE MODEL
  - VALIDATION SET - USED FOR SELECTING HYPERPARAMETERS, EARLY STOPPING
  - TEST SET - USED ONLY $\boxed{\text{ONCE}}$ TO CHECK AND REPORT HOW GOOD A MODEL IS. DECISION IS NEVER MADE BASED ON THIS.

- MEASURE TRAIN AND VALIDATION ACCURACY PERIODICALLY



THE LOSS/TIME PLOT IS THE TRAINING CURVE.

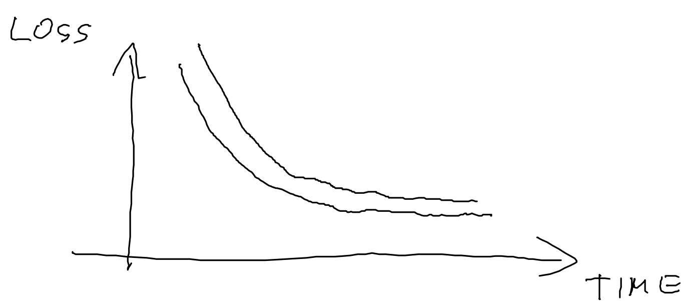
VALIDATION LOSS $\geq$ TRAIN LOSS

- BIAS - VARIANCE TRADEOFF
  - IF THE MODEL IS TOO SIMPLE, IT HAS TOO MUCH <u>BIAS</u> TOWARDS SIMPLICITY

    TRAIN LOSS IS HIGH
    VALIDATION LOSS IS HIGH
    VALIDATION LOSS DOES <u>NOT</u> INCREASE

    UNDER-FIT

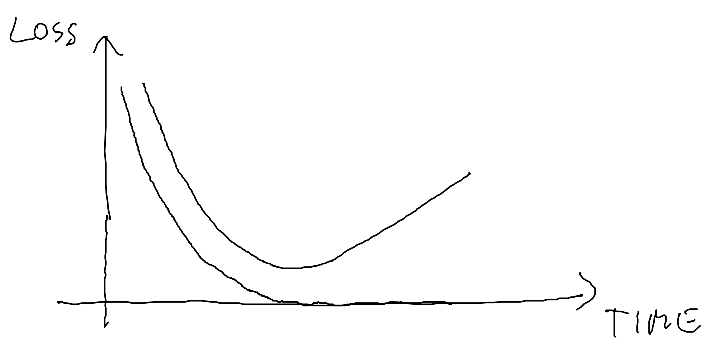
    TRAINING CURVE:



  - IF THE MODEL IS TOO COMPLEX, IT WILL OVERFIT - WILL HAVE HIGH VARIANCE

    TRAIN LOSS IS LOW
    VALIDATION LOSS IS HIGH
    VALIDATION LOSS INCREASES

    OVERFIT



INDUCTIVE BIASES

- BIAS IS NOT ALWAYS BAD. THE <u>CORRECT</u> BIAS IS GOOD. IT ENCOURAGES THE MODEL TO LEARN THE „RIGHT" PARAMETERS OF THE MANY POSSIBILITIES DESCRIBING THE TRAIN SET EQUALLY WELL.
- IT IS A WAY TO BUILD IN OUR KNOWLEDGE TO THE NETWORK
- SIGNIFICANT PART OF THE ML RESEARCH FOCUSES ON COMING UP WITH THE RIGHT INDUCTIVE BIASES.

PERCEPTRON

BASIC BUILDING BLOCK OF NON-CONVOLUTIONAL NEURAL NETWORKS. ALSO CALLED FULLY CONNECTED OR LINEAR LAYER (LATTER WITHOUT THE ACTIVATION FUNCTION).

$$y = \sigma(x_1 w_1 + x_2 w_2 + \dots + x_n w_n + b)$$

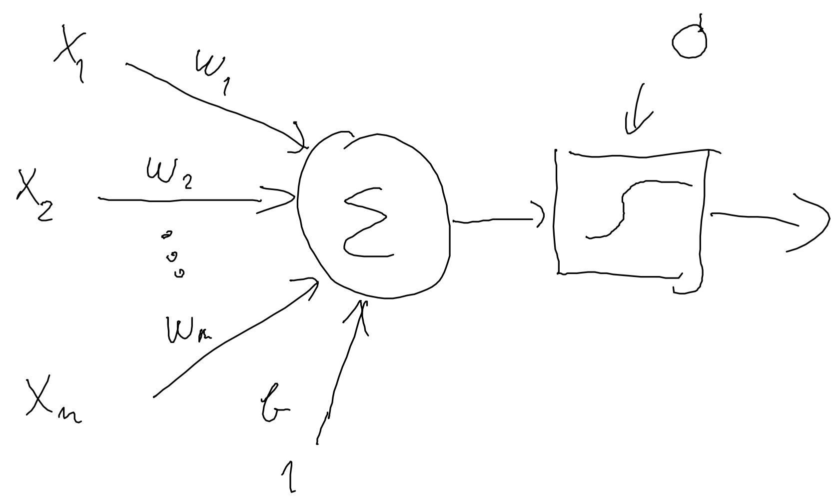
$\sigma(x)$ – ACTIVATION FUNCTION

x_i – INPUT FEATURES

w_i – PARAMETERS (WEIGHTS) – LEARNED

b – BIAS – LEARNED

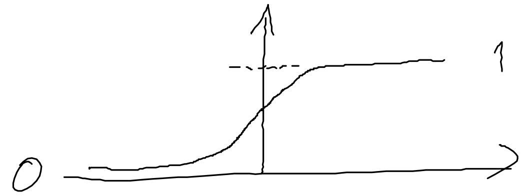
y – OUTPUT



ACTIVATION FUNCTIONS

- SIGMOID

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

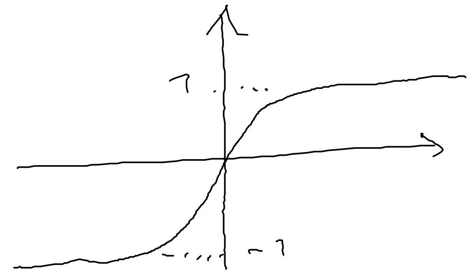


USED:

- BETWEEN LAYERS
- GATES
- BINARY CLASSIFICATION

- TANH

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

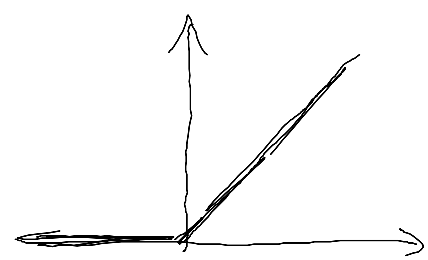


USED:

- BETWEEN LAYERS

- RELU

$$\mathrm{relu}(x) = \max(x, 0)$$



USED:

- BETWEEN LAYERS
  (MOST POPULAR IN CONVNETS)

- SOFTMAX

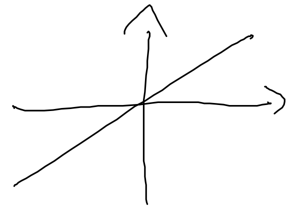
$$\mathrm{softmax}(\bar{x})_i = \frac{e^{x_i}}{\sum_j e^{x_i}}$$

USED:

- MULTICLASS CLASSIFICATION

- IDENTITY

$$i(x) = x$$



USED:

- REGRESSION OUTPUT

VECTORIZATION

$$\bar{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \bar{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \qquad y = \sigma\left(\bar{w}^T \bar{x} + b\right)$$

$\bar{w}^T \bar{x}$ — MATRIX MULTIPLICATION

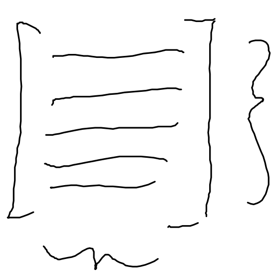
IF $\bar{a}$ AND $\bar{b}$ ARE COLUMN VECTORS,

$\bar{a}^T \bar{b}$ IS THE DOT PRODUCT

$$\bar{a}^T \bar{b} = \mathrm{dot}(\bar{a}, \bar{b}) = \sum_i a_i b_i$$

IN PRACTICE WE ARE USING (MINI) BATCHES OF DATA. A MINIBATCH IS JUST M DIFFERENT BATCHES CONCATENATED TO A MATRIX, WHERE EACH ROW IS A DATA POINT.

$\underline{X} =$



M - NUMBER OF DATA POINTS

N - NUMBER OF INPUT FEATURES

WE CAN ALSO HAVE MULTIPLE (K) OUTPUT FEATURES - EQUIVALENT OF HAVING MORE PERCEPTRONS.

$$\underline{y} = \underbrace{\begin{bmatrix} \text{———} \\ \text{———} \\ \text{———} \\ \text{———} \end{bmatrix}}_{K} \Big\} M$$

IN THIS CASE WE WRITE:

$$\underline{y} = \sigma\left(\underbrace{\underline{X}\,\underline{W}} + \bar{1}_M \bar{b}^T\right)$$

BIAS IS NOW A VECTOR OF K

NOTE THE SWAPPED ORDER

$\bar{1}_M$ - A VECTOR WITH M 1s: $\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \Big\} M$

IN IMPLEMENTATION THIS IS HANDLED BY BROADCASTING AUTOMATICALLY, SO IN FUTURE WE WILL JUST WRITE

$$\underline{X}\,\underline{W} + \bar{b}$$

NOTE: EACH DATA SAMPLE WAS PREVIOUSLY REPRESENTED AS A COLUMN VECTOR. WHEN BATCHING, THEY BECAME ROWS OF THE DATA MATRIX. IT IS COMMON IN LITERATURE TO IGNORE THE BATCH DIMENSION IN THE DEFINITIONS, BUT CARE SHOULD BE TAKEN WHEN IMPLEMENTING THEM.

NOTE 2: BATCHING SERVES MULTIPLE PURPOSES:

1. COMPUTE MULTIPLE SAMPLES IN PARALLEL. GPUs HAS MANY CORES, SO THEY CAN PROCESS THEM IN PARALLEL. (TO SOME UPPER LIMIT GIVEN BY THE GPU).
2. EVEN WHEN THE LIMIT OF THE GPU IS REACHED, OR A CPU IS USED, MATRIX OPERATIONS ARE VERY FAST, WELL OPTIMIZED IMPLEMENTATIONS EXISTS.
3. DURING THE TRAINING, THE LOSS OF ALL SAMPLES IN THE DATA IS AVERAGED, OBTAINING A LESS NOISY ESTIMATE OF THE GRADIENT

IN GENERAL WE WORK WITH TENSORS. THEY ARE N DIMENSIONAL ARRAYS (FOR MATRIX, $N=2$. FOR A BATCH OF GRAY-SCALE IMAGES, $N=3$. FOR A BATCH OF COLOR IMAGES, $N=4$; FOR VIDEOS $N=5$).

EVALUATING THE MODEL

IN ORDER TO KNOW HOW TO IMPROVE THE MODEL, WE NEED TO MEASURE THE ERROR.

FOR REGRESSION AND SINGLE CLASS CLASSIFICATION WE OFTEN USE MEAN-SQUARED ERROR (MSE), BECAUSE ITS DERIVATIVES ARE SIMPLE.

$$L(y_i, \hat{y}_i) = \frac{1}{2}(y_i - \hat{y}_i)^2$$

LOSS   NET OUT   GROUND TRUTH FROM THE DATASET.

i - SAMPLE INDEX FROM DATASET

IF y_i IS A VECTOR, JUST AVERAGE ITS ELEMENTS:

$$L(y_i, \hat{y}_i) = \frac{1}{m}\sum_{j}^{m} L(y_{ij}, \hat{y}_{ij})$$

WE ALSO AVERAGE OVER ELEMENTS OF THE (MINI) BATCH.

IN CASE OF MULTIPLE NETWORK OUTPUTS, MULTIPLE LOSSES MIGHT BE PRESENT. USUALLY A WEIGHTED AVERAGE IS TAKEN OF THEM.

WE MUST HAVE A SINGLE SCALAR LOSS FOR THE GD TO WORK! AFTER ALL, HOW COULD THE NETWOR DECIDE WHICH OUTPUT IS MORE IMPORTANT TO YOU?

LEARNING

- START WITH RANDOM WEIGHTS

  SPECIAL CARE SHOULD BE TAKEN ABOUT THE RANGE - SEE XAVIER INITIALIZATION

- COMPUTE THE GRADIENT OF WEIGHTS WITH RESPECT TO THE LOSS

  $\frac{\partial L}{\partial w}$ GRADIENTS: VECTORS OF ELEMENT-WISE DERIVATES

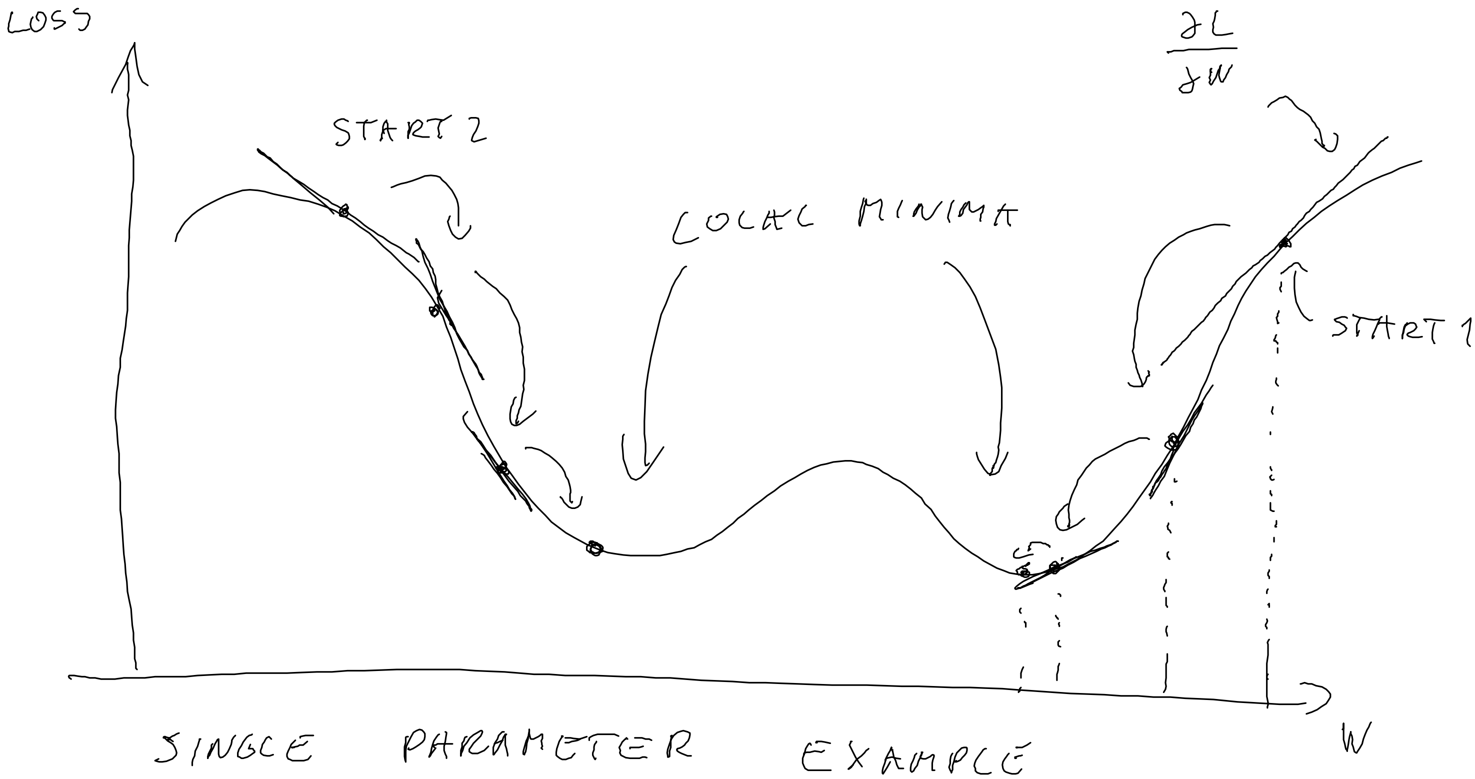
  THEY SHOW IN WHICH DIRECTION TO CHANGE THE WEIGHTS TO INCREASE THE LOSS THE MOST. THAT'S WHE THE - SIGN.

- UPDATE THE WEIGHTS BASED ON GRADIENTS:

$$\underline{w}_i^{NEW} = \underline{w}_i^{OLD} - \eta \frac{\partial L}{\partial \underline{w}}$$

η - LEARNING RATE (USUALLY 0.1 ... 0.0001)

(MORE ADVANCED OPTIMIZERS USUALLY TUNE η SEMI-AUTOMATICALLY. MORE IN TUTORIAL 2)



SINGLE PARAMETER EXAMPLE

PROBLEM: NO WAY TO ESCAPE LOCAL MINIMA. DIFFERENT STARTING POSITIONS CONVERGE TO DIFFERENT SOLUTIONS

WHY THIS IS NOT A PROBLEM

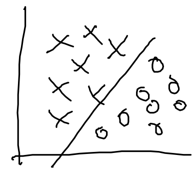
INTUITION ONLY. NO PROOF EXISTS TO MY BEST KNOWLEDGE.

WE ARE WORKING IN HIGH DIMENSIONAL SPACES. AS THE NUMBER OF PARAMETERS (DIMENSIONS) INCREASE, IT BECOMES EXPONENTIALLY MORE UNLIKELY THAT A LOCAL MINIMUM ON ALL THE DIMENSIONS EXISTS IN THE SAME POINT. THUS THE OPTIMIZATION CAN "SLIP OUT" OF LOCAL MINIMUM ON THE

OTHER DIMENSIONS.

MULTI-LAYER PERCEPTRON

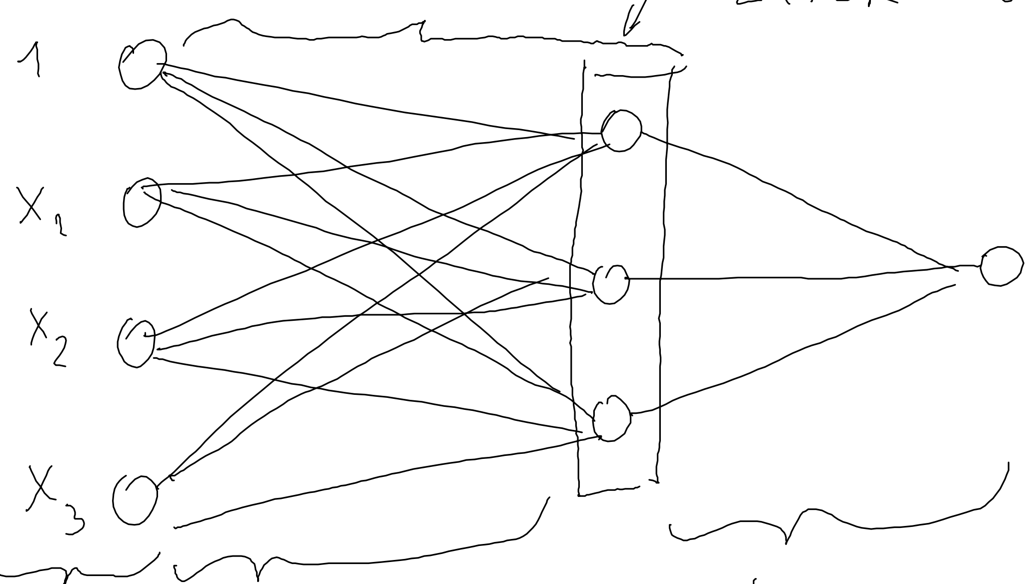
- PERCEPTRON IS LIMITED. IT CAN ONLY REPRESENT LINEAR DECISION BOUNDARIES;



- A 3 LAYER NETWORK IS ALREADY AN UNIVERSAL FUNCTION APPROXIMATOR
- IDEA: PUT MULTIPE PERCEPTRONS AFTER EACH OTHER

ACTIVATION VECTOR: LAYER 1 OUTPUT

LAYER1



$\sigma(\underline{X}\,\underline{W})$

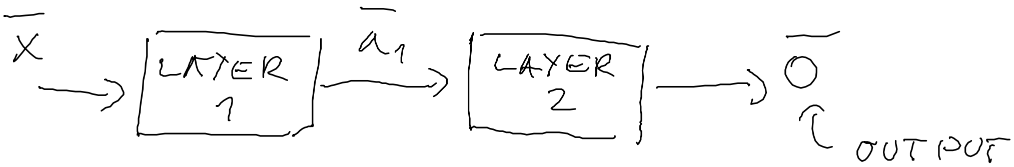
LAYER 2
OUTPUT LAYER

INPUT LAYER. JUST A VECTOR OF DATA

ACTIVATION VECTOR

THE TERM „LAYER" IS ILL DEFINED.

- WE LIKE TO THINK ABOUT LAYERS AS A COMPUTATION THAT TRANSFORMS AN ACTIVATION TO ANOTHER. FOR MLP, ACTIVATIONS ARE VECTORS.



$\to \boxed{} \to$ IS A COMPUTATION/TRANSFORMATION

- FOR ANY NETWORK WE CAN DRAW A COMPUTATION GRAPH.
  - CAN BE ANY GRANULARITY
  - CAN BE USED TO COMPUTE BACKPROP EASILY
  - THIS IS HOW PYTORCH AND TENSORFLOW WORKS.