Machine LearNiNg

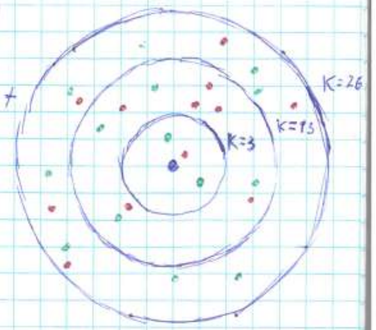
Types of ML Problems:

- **Unsupervised Learning** - the machine did not use marked down data. Could find **Hidden patterns**.

- **Supervised Learning** - the machine use marked down data as training one. Could **predict output** due to previous training data expirience.

- **Reinforcement Learning** - the machine use real-time simulation (system) where it have „rewards" and „punishments". Could simulate **complicated behaviour** and **learn by own mistakes**

K.N.N. („K" - Nearest Neighbors)



Idea: Consiter input Data as point <vector> in n-dimentional space. Then use „K" as Hyperparamiter to find K-Nearest Neighbours. Simply count amount of points of each class among neighbours, so the answer is the greater amount class (for classification). Use mean ($\bar{X}$) of all neighbours (for Regression)

Problems:

- **Curse of Dimensionality** - The more dimentionality is, the greater point Dispercy is → less points would be in „K" area.

- **Feature Scale** - Features have different Metrics, so one feature might be much more important then the other: [0...2000](m), [0...2](Km) → 2m ~ 2Km **Wrong!**

Distance Metrics:

Euclidean: $D(\bar{a}, \bar{b}) = \sqrt{\sum_{i=1}^{n} (\bar{b}_i - \bar{a}_i)^2}$

Manhattan: $D(\bar{a}, \bar{b}) = \sum_{i=1}^{n} |\bar{a}_i - \bar{b}_i|$

Minkowski: $D(\bar{a}, \bar{b}) = \left(\sum_{i=1}^{n} |\bar{a}_i - \bar{b}_i|\right)^{\frac{1}{p}}$

Learning

Accuracy Metrics:

It is a performance indecators (functions) that helps to evaluate the correctness of the model:

Classificator:

$$\boxed{\text{Precision} = \frac{\text{Correct Pred. (class „n")}}{\text{total (class „n")}}}$$

$$\boxed{\text{Recall} = \frac{\text{correct Pred. (class „n")}}{\text{total Pred. (class „n")}}}$$

$$\boxed{\text{Accuracy} = \frac{\text{Pred. Correct}}{\text{total pred.}}}$$

$$\boxed{F1 = \frac{2\,\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}}$$

Regressor:

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

$$\boxed{MAE = \frac{1}{m}\sum_{i=1}^{m}|y_i - \hat{y}_i|}$$

$$\boxed{R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \text{mean}(y))}}$$

Loss $L(y, \hat{y})$:

It is a function that shows how did prediction „y" differs from correct answer „$\hat{y}$". The lessier it is, the better predicted result you have.

Overfitting:

Situaation when model loss on training data → 0 and loss on real data → 100.

Split Data:

Data Normalization:

ALWAYS Normalize The Features!

$$\boxed{x_{new} = \frac{x - \mu}{\sigma}}$$

μ - mean value of feature,

σ - Standart deviation of feature.

Regularization:

A method of adding special **constraints** to the loss function to **decrease model complexity** and **Prevent overfitting**.

$$L_1 \text{ (Lasso)} = \ldots + \beta \sum_{i=1}^{m} |\omega_i|$$

$$L_2 \text{ (Ridge)} = \ldots + \beta \sum_{i=1}^{m} \omega_i^2$$

$$\text{Elastic Net} = \ldots + \beta_1 \sum_{i=1}^{m} |\omega_i| + \beta_2 \sum_{i}^{m} \omega_i^2$$

Linear Regression

INPUT:

$$X_{(m \times n)} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1n} \\ 1 & x_{21} & \cdots & x_{2n} \\ 1 & x_{31} & \cdots & x_{3n} \\ 1 & x_{41} & \cdots & x_{4n} \\ \vdots & \vdots & & \vdots \\ 1 & x_{m1} & \cdots & x_{mn} \end{bmatrix} \quad Y_{(m \times 1)} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix}$$

features — correct answers

Model:

$$W_{(k \times 1)} = \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \\ \vdots \\ \omega_k \end{bmatrix}$$

weights

Output:

$$\hat{Y}_{(m \times 1)} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \vdots \\ \hat{y}_m \end{bmatrix}$$

Prediction result

Train:

$$X \cdot W = \hat{Y}$$

$$L(W) = \frac{1}{m}(XW - Y)^T (XW - Y) = \frac{1}{m} \cdot \sum_{i=1}^{m} (\hat{y}_i - y)^2$$ #Mean Square Error

Analitical Method (not used):

The main idea is to minimize loss function „L" by finding such weights „W" to $\frac{\partial L}{\partial W} = 0$

$$\frac{\partial L}{\partial W} = \frac{2}{m} X^T (XW - Y) = 0$$

$$\boxed{W = (X^T X)^{-1} X^T Y \sim X^+ Y}$$

Gradient descent:

The same thing but consider derivative Vector as gradient of the Loss function. (Repeat iterativly till the acceptable result)

$$\frac{\partial L}{\partial W} \sim grad(L)$$

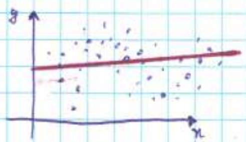
$$\boxed{W_{new} = W - \alpha \cdot \frac{\partial L}{\partial W}}$$

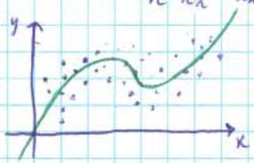
α - Learning Rate

If dependencies are Non-Linear:

Simply add more Polynomial features and normalize them.

$$X = \begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \quad Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \qquad X' = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & x_n^3 \end{bmatrix} \quad Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$





Logistic Regression

The IDEA:

quiet the same as in Linear Regression, But this one is used in Classification and have belonging class probability as an output. Also, for each class there is own Weights Vector.

$$X_{(m \times n)} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1n} \\ 1 & x_{21} & \cdots & x_{2n} \\ \vdots & \vdots & & \vdots \\ 1 & x_{m1} & \cdots & x_{mn} \end{bmatrix} \text{individuals}$$

features

$$W_{(K \times L)} = \begin{bmatrix} \omega_{00} & \omega_{01} & \cdots & \omega_{0L} \\ \omega_{10} & \omega_{11} & \cdots & \omega_{1L} \\ \vdots & \vdots & & \vdots \\ \omega_{K0} & \omega_{K1} & \cdots & \omega_{KL} \end{bmatrix} \text{for each feature}$$

for each Class

$$\hat{Y}_{(L \times 1)} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_L \end{bmatrix}$$

Probab. of Belon[ging] to each cl[ass]

Train:

$$logits(X_{individ}) = X_{individ} \cdot W \sim [1\, x_1\, x_2 \cdots x_n] \cdot \begin{bmatrix} \omega_{00} & \cdots & \omega_{0L} \\ \vdots & & \vdots \\ \omega_{K0} & \cdots & \omega_{KL} \end{bmatrix}$$

$$Softmax(logits(X_{individ})) = \hat{Y}$$

The most possible class type is one with the highest probability.

$$\begin{bmatrix} 1.3 \\ 5.1 \\ 2.2 \\ 0.7 \\ 1.1 \end{bmatrix} \rightarrow \boxed{\text{SoftMax} \; \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}} \rightarrow \begin{bmatrix} 0.02 \\ 0.9 \\ 0.05 \\ 0.01 \\ 0.02 \end{bmatrix}$$

Loss:

$$L(W) = -\frac{1}{m} \sum_{i=1}^{m} \begin{cases} \log(1 - \hat{y}_i) & \text{if } y_i = 0 \\ \log(\hat{y}_i) & \text{if } y_i = 1 \end{cases}$$

$$L(W) = -\frac{1}{m} \sum_{i=1}^{m} \left(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right)$$

LogLoss - return huge Value if Prediction „$\hat{y}_i$" complet[ely] dismatch the correct class answer „[y_i]"

Gradient descent:

$$\frac{\partial L}{\partial W} = \frac{1}{n} \cdot X_{individ}^T (\hat{Y} - Y)$$

Decision Trees.

Idea:

it is a logical algorithm that based on a binary tree. Each **node** contains a **question** (condition

connections - all possible **results** of a question,

leafs - terminal node ready to predict.

After training (tree is complete) we could predict the result by simple tree traverse.

Training:

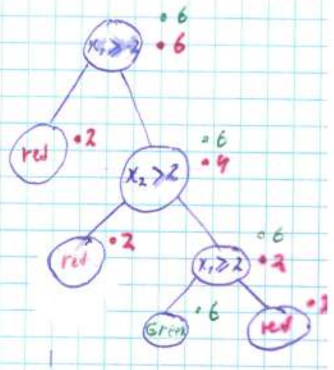
- Check all the leafs and calculate the **Entropy** or **Gini Index** for every possible combination of every feature value split ($x_1 \leq 1, x_2 \geq 3 \dots$)

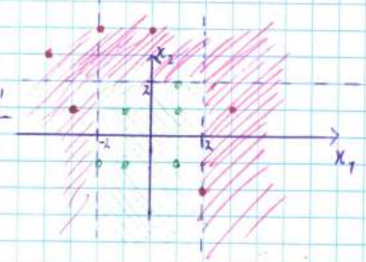
- Calculate **Information Gain** for each possible split By:

$$\boxed{IG = entropy(parent) - \sum_{i=0} \omega_i \cdot entropy(child_i)}$$

where „ω_i" - $\frac{size(child)}{size(parent)}$

- Choose split with the highest IG value and repeat untill „max_tree_depth" or every final node become **leaf**.





$$\boxed{entropy(X) = -\sum_{i=1} P(x_i) \cdot \log_2 P(x_i)}$$

where „p()" - probability of a class.

$$\boxed{P_{(class)} = \frac{\text{Total objects}}{\text{class amount}}}$$

$$\boxed{gini\_index(X) = 1 - \sum_{i=1} P(x_i)^2}$$

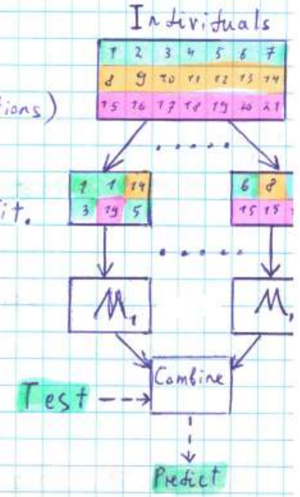
Ensembles.

Voting:

Use different algorithms with Prediction probabili greater than $\frac{1}{2}$ (50%) and simply get the mean Value of all the results for **Regression.**
For Classification problem the idea is the same, but we Choose the most frequent class.

Bagging (parallel):

Firstly, get some random sampes (with repetitions) and train some models to get "Weak learners" so they would not overfit.
Then, pass the test data through these models and mean the result.



Random Forest:

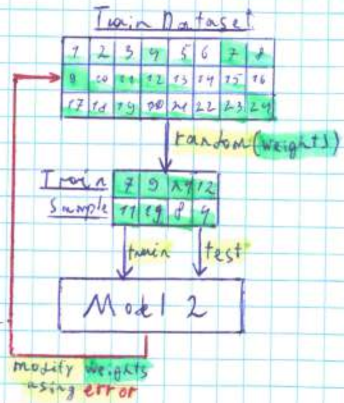
This is a variation of **Bagging** technique where we randomly sample not only **individuals** but their **features** too.
Also it is used **Decision Tree** as a model.

Could learn very deep and hard dependencies.

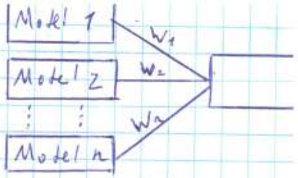
Boosting (sequential):

It is a process that uses a set of models (**Weak Lerners**) combined into single **Strong Learner** model in order to increase the accuracy.

Adaptive Boosting (Adaboost):



- Firstly add "$\frac{1}{n}$" weight to each individ (or sample) in Training Dataset.
- Randomly (depending on weights) choose data to create training sample and train clear model (weak Lerner).
- Test this model on the same sample and mark up individuals were predicted with error.
- increase weight at those individuals in Training dataset.
- Repeat this steps and combine all weak lerners into one strong learner using weights, which are depends on weak learner accuracy.

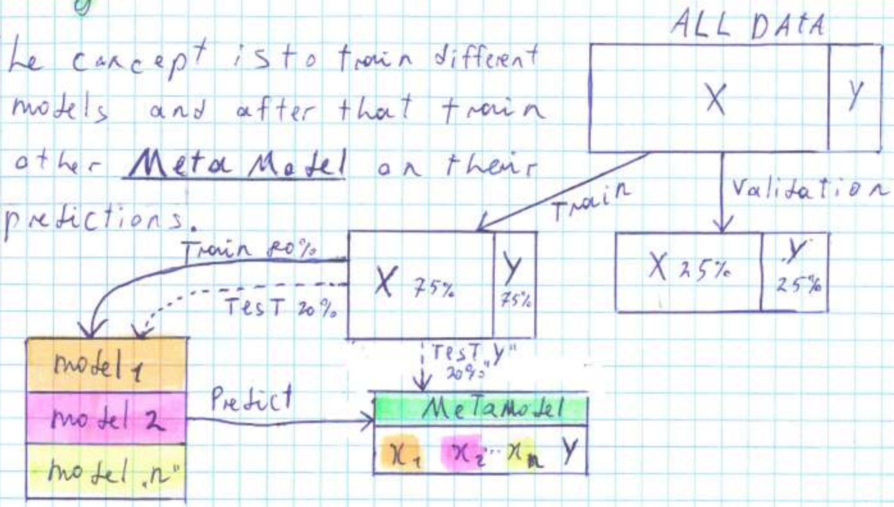


$$\boxed{W_i = \frac{1}{2} \ln\left(\frac{1-error_i}{error_i}\right)}$$

Gradient Boosting:

Blending:

The concept is to train different models and after that train other **Meta Model** on their predictions.

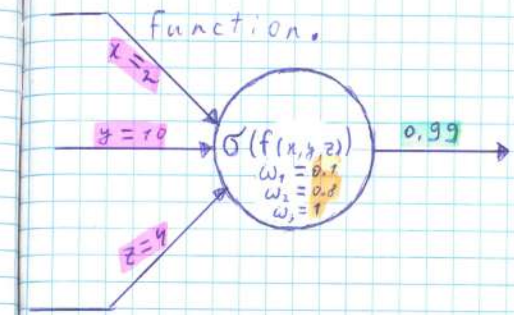


Stacking:

DEEP LEARNING

Neuron:

The main structure element of ANN's. The main idea behind it is to <u>linearly transform</u> some input factor (constant, Vector, Matrix, tensor...) using neuron function and then apply <u>non-linear activation</u> function.

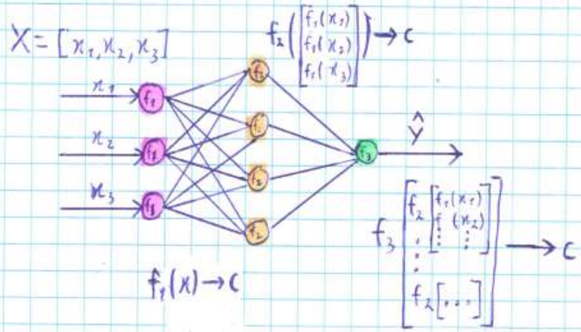


$$f(x, y, z) = \omega_1 \cdot x + \omega_2 \cdot \frac{y}{2} + \omega_3 \cdot z$$
$$\sigma(x) = \text{Sigmoid}(x) = \frac{1}{1+e^{-x}}$$

$$\sigma(f(2,10,4)) = \text{Sigmoid}\left(0.1 \cdot 2 + 0.8 \cdot \frac{10}{2} + 1 \cdot 4\right) =$$
$$= \text{Sigmoid}(8.2) = 0.99$$

Layer:

The way to group neurons of same type into one object.



- Input layer
- Output layer
- other

Activation Functions

Idea:

The goal of any activation function is to prevent collapse of all linear function into one. Mostly, it also normalize data and should be differentiable.

Consider sequential linear functions without activation

$$\underbrace{\boxed{\omega_1(k_1 x + b_1)}}_{f_1(x)} \rightarrow \underbrace{\boxed{\omega_2(k_2 x + b_2)}}_{f_2(x)} \rightarrow \underbrace{\boxed{\omega_3(k_3 x + b_3)}}_{f_3(x)}$$

$$f_3(f_2(f_1(x))) = \omega_3(k_3(\omega_2(k_2(\omega_1(k_1 x + b_1)) + b_2)) + b_3) \sim \boxed{\omega(kx + b)}$$

could be simplifi into this expression (linear)

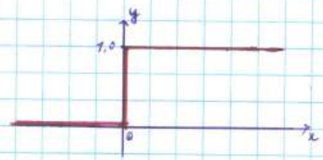
Same but using activation:

$$\sigma f_3(\sigma f_2 \ldots)) = \sigma(\omega_3(k_3(\sigma \omega_2 \ldots . + b_3)) \rightarrow$$ Could not be simplified because „σ"-function!

Binary Step:

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

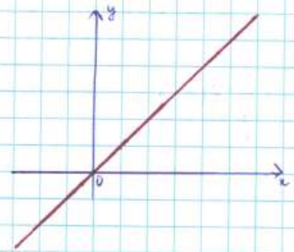
$$f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{none} & \text{if } x = 0 \end{cases}$$



Linear:

$$f(x) = x$$

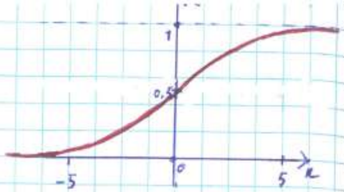
$$f'(x) = 1$$



Sigmoid (logistic):

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

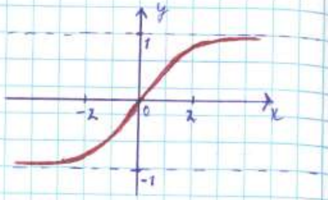
$$f'(x) = f(x) \cdot (1 - f(x))$$



Tanh:

$$f(x) = \frac{2}{1+e^{-2x}} - 1$$

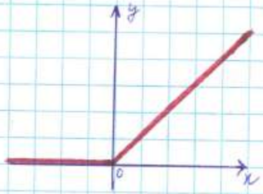
$$f'(x) = 1 - f(x)^2$$



ReLU (Reflected Linear Unit):

$$f(x) = \max(0\,;x) \sim \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

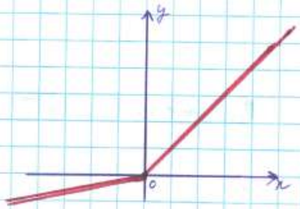
$$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$



Leaky ReLU:

$$f(x) = \max(\alpha x\,;x) \sim \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

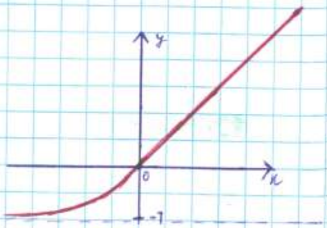
$$f'(x) = \begin{cases} \alpha & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$



ELU (Exponential Linear Unit):

$$f(x) = \begin{cases} \alpha\ e^{x} - 1 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

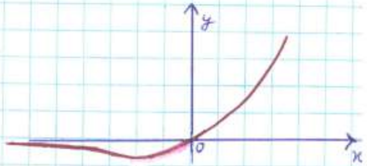
$$f'(x) = \begin{cases} f(x) + \alpha & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$



Swish:

$$f(x) = x \cdot \text{sigmoid}(x) \sim \frac{x}{1+e^{-x}}$$

$$f'(x) = f(x) + \text{sigmoid}(x) \cdot (1 - f(x))$$

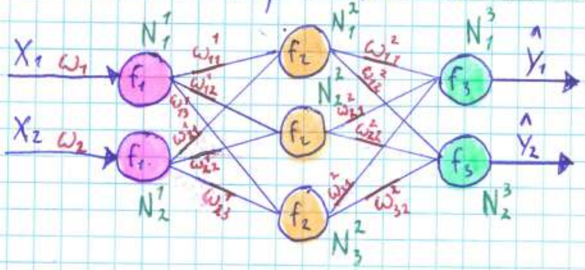


Idea:

The main idea is to update all the parameters (weights) using gradient descient to minimize Loss function.

Example:

Consider Simple Neural Network:



$f_1(x) = \tanh(\omega x)$

$f_2(x_1, x_2) = \sigma(\omega_1 x_1 + \omega_2 x_2)$

$f_3(x_1, x_2, x_3) = 1 \cdot (\omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3)$

Forward Pass:

$$\hat{Y} = f_3(\, f_2[\, f_1\{X\}, \dots], \dots, \dots)$$

$$\hat{Y} = 1 \cdot (\omega_1^2 (\sigma[\omega_1^1[\tanh(\omega_1 X)] + \omega_2^1 \dots]) + \omega_2^2 \dots + \omega_3^2 \dots)$$

Backward Pass:

Consider Loss function as MSE

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

Use **LocalGradients** to find **Weight correction** factors and further LocalGradients for each Neuron and it's connection.

To simplify, our goal is to find every partial derivative by each model parameter (weight).

for f_3:

$$\frac{\partial L}{\partial \omega_{11}^2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f_3} \cdot \frac{\partial f_3}{\partial \omega_{11}^2} =$$

$$= \underbrace{-\frac{2}{m}(y - \hat{y})}_{\substack{L' \\ \text{(Loss)}}} \cdot \underbrace{1}_{\substack{f_3' \\ \text{(Activation)}}} \cdot \underbrace{f_2(f_1[X], \dots)}_{\text{unweight previous result from } \omega_{11}^2}$$

For output Layer:

$$\boxed{\text{LocalGrad}(N_m^n) = L' \cdot f_n'}$$

$$\boxed{\omega_{new} = \omega_{old} - \alpha \cdot \text{LocalGrad}(N_m^n) \cdot \text{res}(\omega}$$

for f_2:

$$\frac{\partial L}{\partial \omega_{11}^1} = \sum_{i=1} \left(\omega_i \cdot \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f_{3_i}}\right) \cdot f_2' \cdot \frac{\partial f_2}{\partial \omega_{11}^1} =$$

$$= \underbrace{\sum_{i=1} (\omega_i \cdot \text{LocalGrad}(N_i^3))}_{\text{sum of weighted Previous Local gradients,}} \cdot \underbrace{\sigma(\dots)(1 - \sigma(\dots))}_{\substack{f_2' \\ \text{(Activation)}}} \cdot \underbrace{f_1(X)}_{\text{unweight prev. result from } \omega_{11}^1}$$

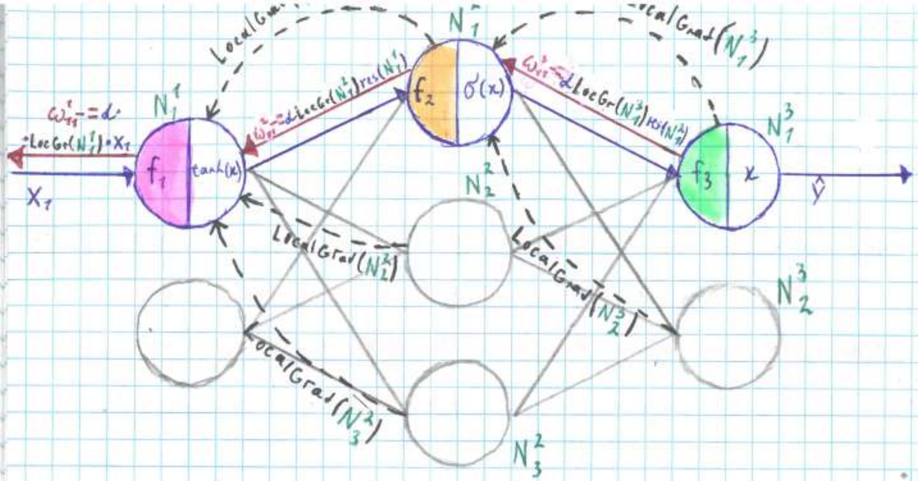
For Hidden Layers:

$$\boxed{\text{LocalGrad}(N_m^{n-1}) = \sum_{i=1} (\omega_i \cdot \text{PrevLocalGrad}(N_i^n)) \cdot f_{n-1}'}$$

for f_1:

$$\frac{\partial L}{\partial X_1} = \sum_{i=1} \left(\omega_i \cdot \sum_{j=1} \left(\omega_j \cdot \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f_{3_j}}\right) \cdot f_{2_i}'\right) \cdot f_1' \cdot \frac{\partial f}{\partial \omega_1} =$$

$$= \underbrace{\sum_{i=1} (\omega_i \cdot \text{localGrad}(N_i^2))}_{\text{sum of weighted Previous Local Gradients.}} \cdot \underbrace{1 - \tanh^2(\dots)}_{\substack{f_1' \\ \text{(activation)}}} \cdot \underbrace{X_1}_{\text{unweight prev. res from } \omega_1} \quad - \text{\underline{Global Gradient}}$$



N_1^1
f_1
$\tanh(x)$
X_1
N_1^2
f_2
$\sigma(x)$
N_1^3
f_3
x
$\hat{y}$
N_2^2
N_3^2
N_2^3
LocalGrad(N_1^3)
LocalGrad(N_2^2)
LocalGrad(N_2^3)
LocalGrad(N_3^2)

CNN

Convolutional Neural Networks

Idea:

It is a NN that contains special <u>convolution layers</u>. Mostly it is used for processing images.

Convolution2D:

An operation that used to extract some locational information into <u>Feature maps</u> using <u>Kernels (filters)</u>

Kernel: square matrix $n \times n$ that represent a <u>filter</u> to „Detect" some patterns in image. Mostly, it consists of <u>parameters (weights)</u> which automatically change in train loop.

Convolution:

$$\text{Stride} = 2 \; ; \; \text{Kernel} = \underbrace{\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}}_{3\times 3}$$

Image 7×7

$$\begin{bmatrix} 1 & 9 & 8 & 4 & 4 & 5 & 7 \\ 4 & 8 & 6 & 7 & 9 & 1 & 7 \\ 4 & 0 & 5 & 9 & 3 & 8 & 4 \\ 7 & 3 & 5 & 9 & 0 & 5 & 4 \\ 7 & 4 & 1 & 1 & 8 & 1 & 2 \\ 7 & 6 & 6 & 9 & 8 & 7 & 6 \\ 3 & 6 & 3 & 5 & 4 & 2 & 7 \end{bmatrix}$$

$$Out_{11} = Sum\left(\begin{bmatrix} 1 & 9 & 8 \\ 4 & 8 & 6 \\ 4 & 0 & 5 \end{bmatrix} \circ \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}\right)$$

$$Out_{12} = Sum\left(\begin{bmatrix} 8 & 4 & 4 \\ 6 & 7 & 9 \\ 5 & 9 & 3 \end{bmatrix} \circ \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}\right)$$

$$\downarrow \text{Output}$$

$$\begin{bmatrix} 21 & 7 & -24 \\ -1 & 30 & 12 \\ 7 & 25 & 18 \end{bmatrix}$$

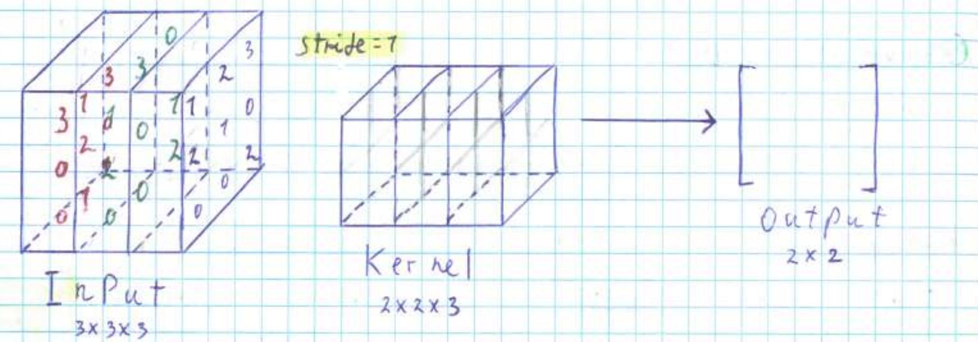
Stride: amount of pixels on which kernel would be shifted during Convolution.

Padding: add frame of mean value of pixels to the image. Increases output image size after convolution.

Feature Map: Simply shows if **Kernel was activated** (When Feature Map numbers are high with Abs()) **or not** (when F.M. Abs(nums) are **low**). The one represents if there is a kernel pattern on the image.

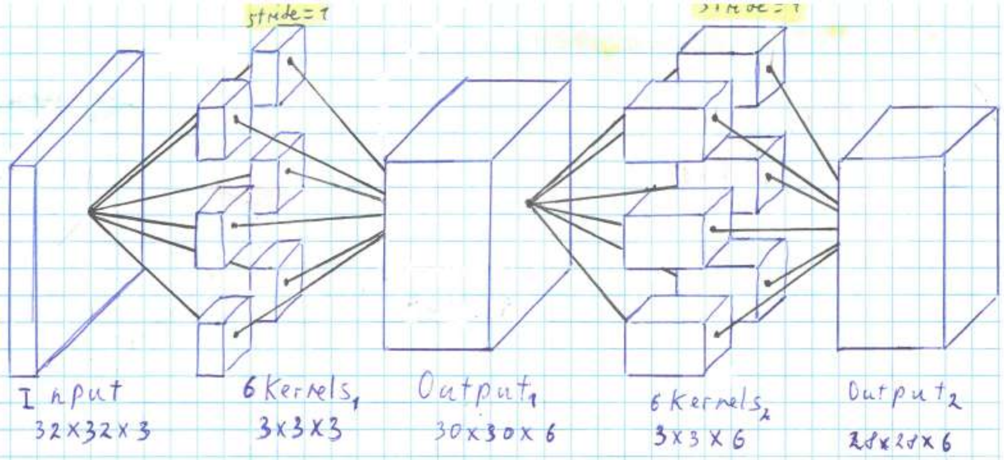
Convolution 3D:

This one is the same as **2D** but the **Kernel and input** are now **Tensors** and **feature map** is still **matrix**.



Multiple Convolutions:

Consider convolutional layer as multiple kernels applied to single **input** and then stacking the results into one tensor. So this tensor could be the input to other convolutional layer with kernels of Depth = stacked results.



Pooling:

A technique of **size reduction** of feature map By splitting matrix into pieces and getting „**Max**" or „**Average**" or „**Sum**" Value. Used to **decrease sensitivity** of next layers. Helps to recognize objects in different positions and sizes.

$$\begin{bmatrix} 1 & 7 & 2 & 6 \\ 2 & 4 & 0 & 2 \\ 1 & 4 & 0 & 8 \\ 0 & 0 & 4 & 9 \end{bmatrix} \xrightarrow{\text{Max Pooling}} \begin{bmatrix} 7 & 6 \\ 4 & 9 \end{bmatrix}$$

Alex Net as example :