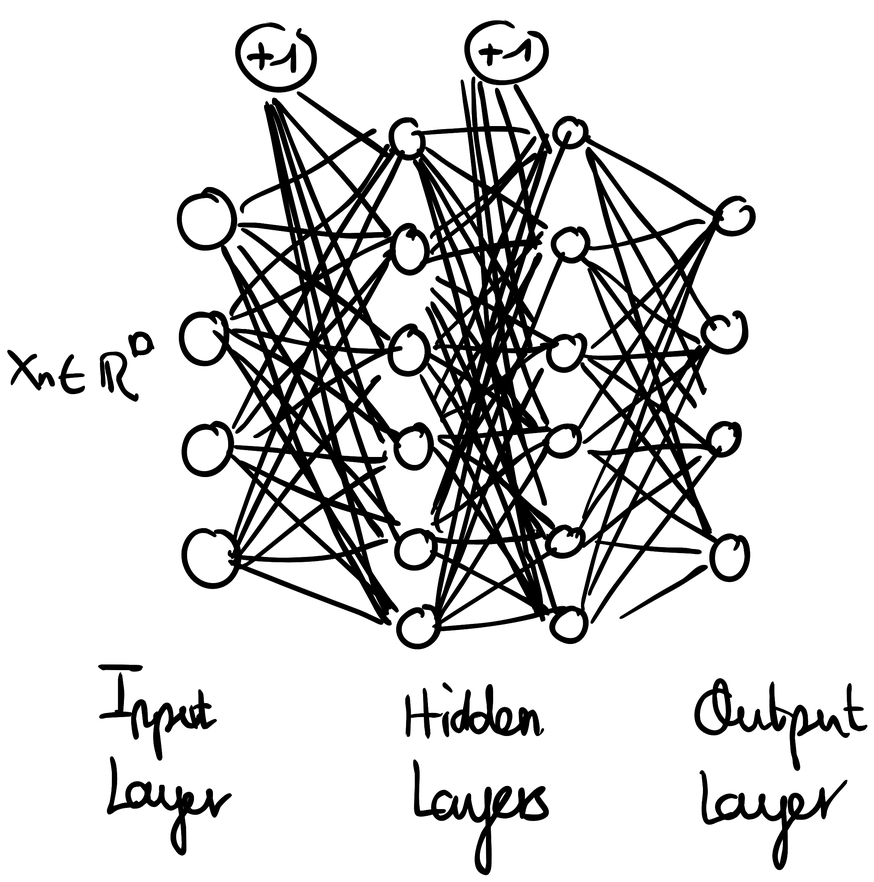
Neural Networks (NN)

A typical NN (feed-forward) consists of a neurone input layer of size D, one or more hidden layers (number of neurones per layer is tunable), and an output layer (size depends on problem).



"Learning" for a neural net can be reduced to determining :

- weights
- biases

of the model.

↳ Such values are determined by estimations using "maximum likelihood": (MLE).

Maximum Likelihood estimation (MLE).

Consider $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^{N}$ samples used to train the neural net.

In supervised learning contexts, MLE determines parameters which maximizes probability of seeing $\mathcal{D}$.

↳ $\hat{\theta}^{MLE} = \underset{\theta}{\operatorname{argmax}}\ \mathbb{P}(Y|X, \theta)$ where $\theta = \{W^{(l)}, b^{(l)}\}_{l=1}^{L}$

$\hat{\theta}^{MLE} = \{\hat{W}^{(l)}, \hat{b}^{(l)}\}_{l=1}^{L}$

Assuming $y_n \overset{iid}{\sim} \mathbb{P}(Y|X, \theta)$, $\hat{\theta}^{MLE} = \underset{\theta}{\operatorname{argmax}} \prod_n \mathbb{P}(y_n | x_n, \theta)$

- Classification.

Labels follow Bernouilli Distribution (dice, coin ...)

$\hat{\theta}^{MLE} = \underset{\theta}{\operatorname{argmax}} \prod_n \prod_k \mathbb{P}(y_n = k | x_{n,k}; \theta)^{t_{nk}}$ ← t is a one-hot encoded y (true label)

where: $\sum_k \mathbb{P}(y_n = k | x_n; \theta) = 1$ (a sample always belong to a class)

Convert to logs in order to compute sums instead of products:

$\hat{\theta}^{MLE} = \underset{\theta}{\operatorname{argmax}} \sum_n \sum_k t_{nk} \log(\mathbb{P}(y_n = k | x_n; \theta))$

We wish to minimize the error (Negative log likelihood):

$$NLL(\theta) = -\ell(\theta) = -\sum_n \sum_k t_{nk} \log \mathbb{P}(y_n = k \mid x_n, \theta)$$

For a single sample, this is the cross entropy loss:

$$Error_n = -\sum_k t_{nk} \log \mathbb{P}(y_n = k \mid x_n, \theta)$$

- Regression

$\mathcal{L}(\theta) = \prod_n \mathbb{P}(y_n \mid x_n, \theta)$ where $y_n \in \mathbb{R}$

We assume that $\mathbb{P}(y_n \mid x_n, \theta) = \mathcal{N}(y_n \mid \hat{y}(\theta, x), \sigma^2)$

Substitute into likelihood:

$$\mathcal{L}(\theta) = \prod_n \mathcal{N}(y_n \mid \hat{y}(\theta, x), \sigma^2) = \prod_n \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\left(\frac{(y_n - \hat{y}_n)^2}{2\sigma^2}\right)}$$

Take log-likelihood:

$$\ell(\theta) = \sum_n \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \left(\frac{(y_n - \hat{y}_n)^2}{2\sigma^2}\right)$$

$$\ell(\theta) = -\frac{1}{2}\sum_n (y_n - \hat{y}_n)^2$$

The error becomes: $NLL(\theta) = -\ell(\theta) = \frac{1}{2}\sum_n (y_n - \hat{y}_n)^2$

And for a single sample, it is: $\frac{1}{2}(y_n - \hat{y}_n)^2$ which is the RSS

The MLE is an unbiaised estimator. $\mathbb{E}(\hat{\theta}^{MLE}) = \theta^*$.

↳ For infinite training data: $\hat{\theta}^{MLE} \to \theta^*$

We now need to find $\hat{\theta}$ that minimizes the cost. ⇒ Gradient descent.

Stochastic gradient descent

$$w^{(t+1)} = w^{(t)} - \eta \frac{\delta E}{\delta w}$$

$$b^{(t+1)} = b^{(t)} - \eta \frac{\delta E}{\delta b}$$

→ Find these gradients using a 2 step:
- Forward propagation
- Back propagation.

↳ Notations

x_n datas/input

$w_{ji}^{(l)}$ weight on edge connecting i^{th} neurone on layer $l-1$ and j^{th} neurone on layer l.

$b_i^{(l)}$ weight connecting bias neurone with i^{th} neurone on layer l.

n_l number of neurones on layer n.

$u_j^l = w_{ji}^l a_i^{l-1} + b_j^l$

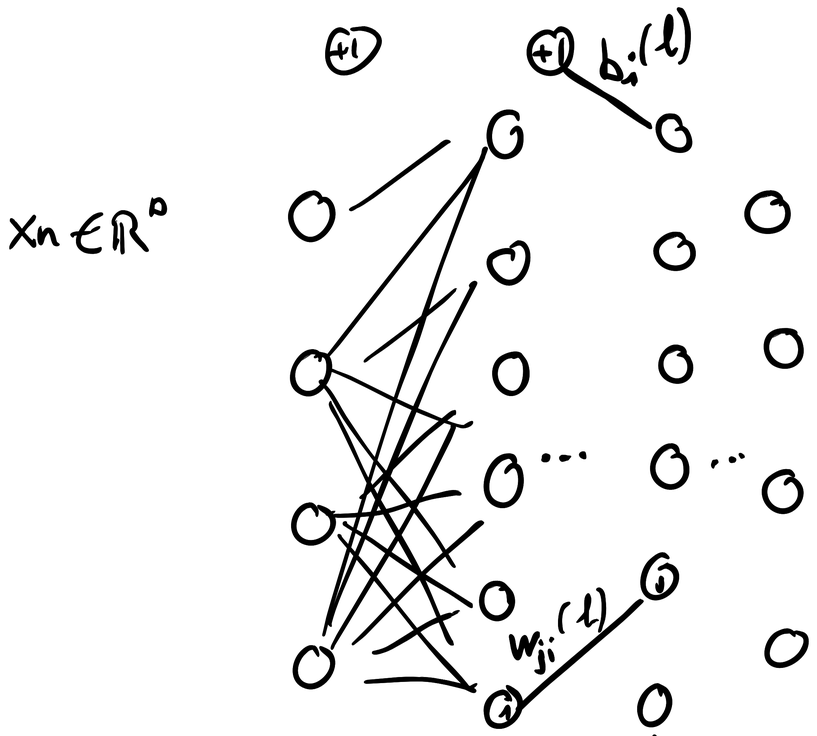
$a_j^l = f^l(u_j^l)$ an activation function, typically non-linear.

where $a^{(0)} = x_n$, output of 0^{th} layer is the input to the network

↳ $f^L(x) = \begin{cases} x & \text{regression} \\ \text{softmax}(x) & \text{classification} \end{cases}$ for the last layer L.

$a^L = \hat{y}$ for the last layer.

In regression $a^L = f^L(u^L) = u^L$



Notations now being defined, we can start the neuron training process.

- Forward Propagation.

1) Randomly initialize w_{ji}^l and $b_j^{(l)}$ for every layer.
2) Pick a sample (x_n, y_n).
3) Find $u^{(1)}, a^{(1)}, u^{(2)}, a^{(2)} \dots, u^{(L)}, a^{(L)}$ in this order using these two formulas alternately:

$$u_j^l = w_{ji}^l a_i^{l-1} + b_j^l$$

$$a_j^l = f^l(u_j^l).$$

4) Compute the prediction $\hat{y}_n$ and sample error E_n:

$$E_n^{\text{classification}} = -\sum_k y_{nk} \log \mathbb{P}(\hat{y}_n = k \mid x_n, \theta)$$

$$E_n^{\text{regression}} = \frac{1}{2}(y_n - \hat{y}_n)^2$$

↳ $E = \sum_n E_n$.

Now that we have the cost, we need to compute the gradient of the cost w.r.t the weights and bias. This gradient computation is done with back propagation.

Main idea: use $\frac{\partial E}{\partial a}$, $\frac{\partial E}{\partial u}$ to determine $\frac{\partial E}{\partial w}$, $\frac{\partial E}{\partial b}$.

• Back propagation.

a) Find: $\frac{\partial E}{\partial a^{(L)}}$, $\frac{\partial E}{\partial u^{(L)}}$, $\frac{\partial E}{\partial a^{(L-1)}}$, $\frac{\partial E}{\partial u^{(L-1)}}$ ... $\frac{\partial E}{\partial a^{(1)}}$, $\frac{\partial E}{\partial u^{(1)}}$ in this order.

↳ Can be simplified to:

$$\frac{\partial E}{\partial a_k^{(L)}} = \frac{\partial E}{\partial \hat{y}_{n,k}} = \begin{cases} y_{nk} - \hat{y}_{nk} & \text{regression} \\ -y_{nk}/\hat{y}_{nk} & \text{classification} \end{cases}$$

↳ Compute $\delta_k^{(l)} = \frac{\partial E}{\partial u_k^{(l)}}$ the cost derivative wr.t u for all neurons in the last layer.

$$\delta_k^{(L)} = \frac{\partial E}{\partial u_k^{(L)}} = \frac{\partial E}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial u_k^{(L)}} = \frac{\partial E}{\partial a_k^{(L)}} \times f'^{(L)}(u_k^{(L)}) \quad \text{using chain rule}$$

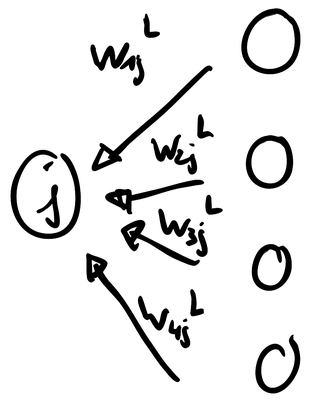
We know both terms, δ_k can be computed.

How do we find $\frac{\partial E}{\partial a^{(L-1)}}$ now? Using chain rule.

$$\frac{\partial E}{\partial a_j^{L-1}} = \sum_k \frac{\partial E}{\partial u_k^L} \frac{\partial u_k^L}{\partial a_j^{L-1}}$$

$\frac{\partial E}{\partial u_k^L}$: change in the cost wrt the change in the activation of the j^{th} neuron in $l-1$ layer.

$\Rightarrow$ We need to sum over k, since changes in the j^{th} neuron affect all the neurons in next layer.



$$\frac{\partial E}{\partial a_j^{L-1}} = \sum_k \delta_k^L w_{kj}^L$$

Generalizing this for any layer: $\frac{\partial E}{\partial a_j^l} = \sum_k \delta_k^{l+1} w_{kj}^{l+1}$

Then, $\frac{\partial E}{\partial u_j^l} = \frac{\partial E}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial u_j^l} = \frac{\partial E}{\partial a_j^l} \cdot f'^{(l)}(u_j^{(l)})$

$\frac{\partial E}{\partial a_j^l}$: known from previous step; $f'^{(l)}(u_j^{(l)})$: also known.

$$= \sum_k \delta_k^l w_{kj}^l \cdot f'^{(l)}(u_j^{(l)}) = f'^{(l)}(u_j^{(l)}) \sum_k \delta_k^{l+1} w_{kj}^{l+1}$$

Recall that $\delta_h^{(l)} = \frac{\partial E}{\partial u_h^{(l)}}$

Up $\delta_j^{\,l} = f'^{(l)}(u_j^{(l)}) \cdot \sum_h \delta_h^{\,l+1} w_{hj}^{\,l+1}$

To summarize, we have introduced two similar notations:

$$\frac{\partial E}{\partial w_j^{\,l}} = \frac{\partial E}{\partial a_j^{\,l}} \frac{\partial a_j^{\,l}}{\partial w_j^{\,l}} \qquad \text{and} \qquad \frac{\partial E}{\partial w_j^{\,l}} = \frac{\partial a_j^{\,l}}{\partial w_j^{\,l}} \sum_h \frac{\partial E}{\partial u_h^{\,l+1}} \frac{\partial u_h^{\,l+1}}{\partial a_j^{\,l}}$$

$$\frac{\partial E}{\partial a_j^{\,l}} = \sum_h \frac{\partial E}{\partial u_h^{\,l+1}} \frac{\partial u_h^{\,l+1}}{\partial a_j^{\,l}}$$

2) Compute costs gradients w.r.t weights and biases

i.e find $\frac{\partial E}{\partial w_{ji}^{(l)}}$, $\frac{\partial E}{\partial b_j^{(l)}}$ for every layer in the network.

Still applying chain rule:

$$\frac{\partial E}{\partial w_{ji}^{(l)}} = \frac{\partial E}{\partial u_j^{(l)}} \frac{\partial u_j^{(l)}}{\partial w_{ji}^{(l)}}$$

$$\frac{\partial E}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}$$

$$\frac{\partial E}{\partial b_i^{(l)}} = \frac{\partial E}{\partial u_i^{(l)}} \frac{\partial u_i^{(l)}}{\partial b_i^{(l)}}$$

$$= \delta_i^{(l)} \times 1 = \delta_i^{(l)}$$

Once we computed all weights and biases in the network, we perform the stochastic gradient update and update weights and biases:

$$w_{ji}^{(l)(t+1)} = w_{ji}^{(l)(t)} - \eta \frac{\partial E}{\partial w_{ji}^{(l)(t)}}$$

$$b_j^{(l)(t+1)} = b_j^{(l)(t)} - \eta \frac{\partial E}{\partial b_j^{(l)(t)}}$$

NB: In case of a mini batch stochastic gradient descent:

For a batch size m, the cost gradient w.r.t weights becomes

$$\frac{\partial E}{\partial w_{ji}^{(l)(t)}} = \frac{1}{m} \sum_m \frac{\partial E_m}{\partial w_{ji}^{(l)(t)}}$$

Mini batch weight and bias update:

$$w_{ji}^{(l)(t+1)} = w_{ji}^{(l)(t)} - \frac{\eta}{m} \sum_m \frac{\partial E_m}{\partial w_{ji}^{(l)(t)}} \quad ; \quad b_j^{(l)(t+1)} = b_j^{(l)(t)} - \frac{\eta}{m} \sum_m \frac{\partial E_m}{\partial b_j^{(l)(t)}}$$

Summary:

For i in range(epoch): ← usually until convergence

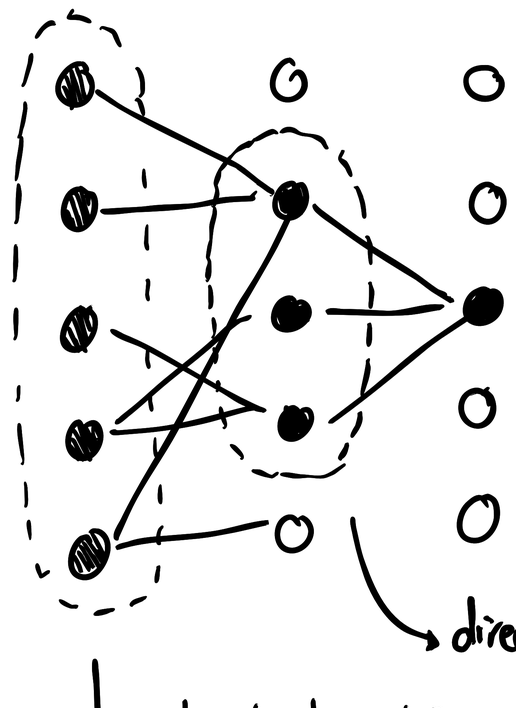
- Pick a random sample $x_n \in \mathbb{R}$
- Determine $u^{(1)}, a^{(1)}, \dots, u^{(L)}, a^{(L)}$ (forward prop.)
- Find E_n
- Determine $\frac{\partial E}{\partial a^{(L)}}, \frac{\partial E}{\partial u^{(L)}}, \dots, \frac{\partial E}{\partial a^{(1)}}, \frac{\partial E}{\partial u^{(1)}}$ (back propagation)
- Compute $\frac{\partial E}{\partial w_{ij}^{(l)}}, \frac{\partial E}{\partial b_j^{(l)}}$
- Update $w_{ij}^{(l)}$ and $b_i^{(l)}$ with the stochastic gradient descent update rule.

Convolutional Neural Networks. (CNN)

CNN are special NN for processing data with grid like topology.
CNN have 3 fundamental features that reduce parameters in NN.:

- Sparse interactions between layers
  In NN, every neurone in layer $l-1$ is connected to every neurone in layer l.
  ↳ Need much datas, convergence time increases, and we overfit.

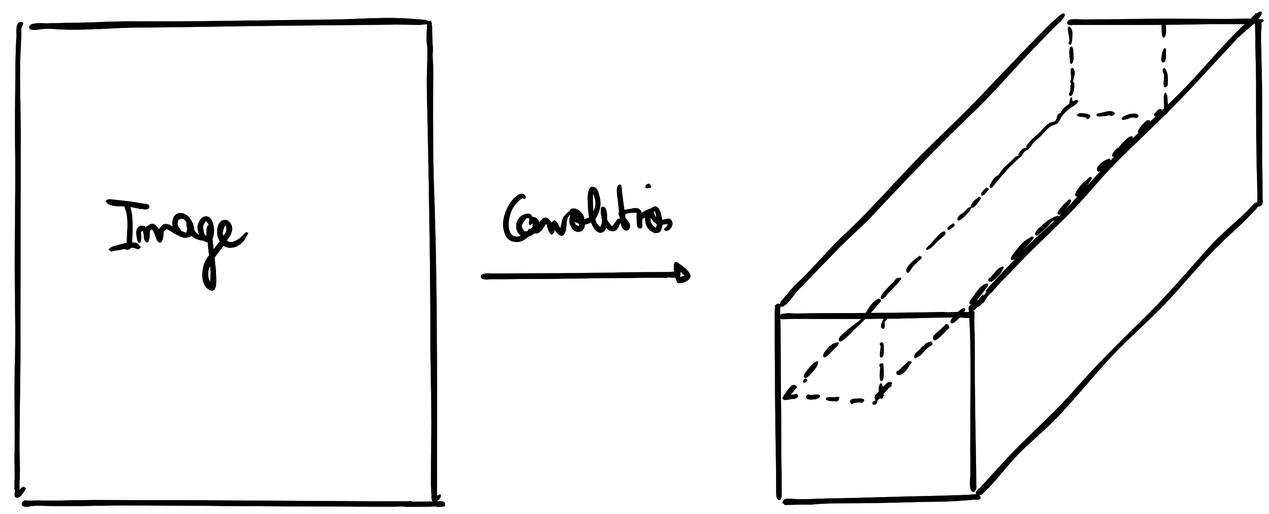
We introduce the notion of direct and indirect interactions:



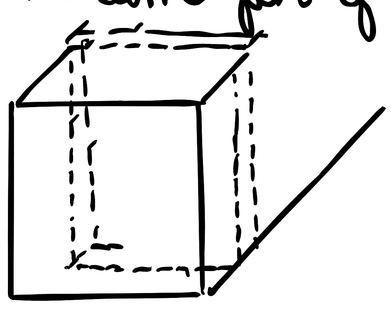
We do not need all neurons to be connected for the information to transit through the network. Sparse interaction between layers should suffice.

↳ indirect interactions.

- Parameter sharing: CNN create spatial features



A section of this volume taken through the depth will represent features of the same part of the image



Each feature in the same depth layer is generated by the same filter that convolves the image.

⇒ The individual points of the same depth of the feature map (the output 3D volume) are created from the same kernel. They are created using the same set of shared parameters.

- Equivariant representations:

A function f is said to be equivariant w.r.t another function g if:

$$f(g(x)) = g(f(x))$$

In the CNN, we can first convolve the image and then translate it, or vice versa.

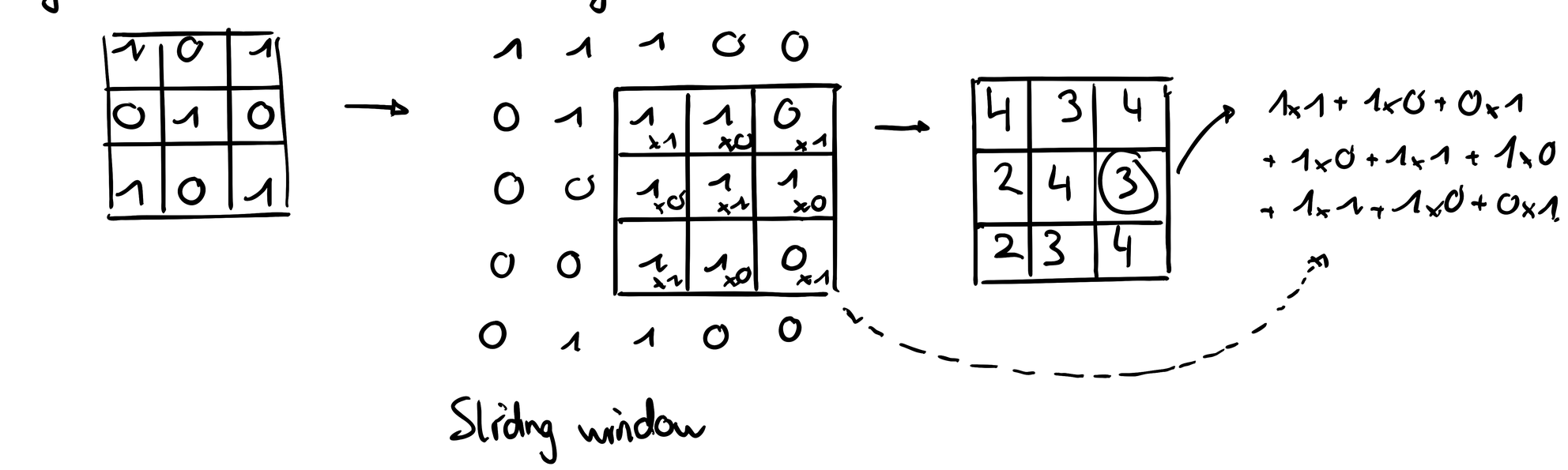
In image data, the first convolution layer usually focuses on edge detection. However, similar edges may occur throughout the image, so it makes sens to represent them with the same parameters

What are the different layers in a CNN ?

- Convolution layer
- Activation layer
- Pooling layer
- Fully connected layer

I. Convolution layer.

Usually 1st layer, where we convolve the image / data using filters or kernels.
Filters are small units that we apply across the data through a sliding window.



Mathematically, convolution in 1 dimension (input f and kernel g are both functions of 1D datas (e.g time series)), then convolution is given by:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau$$

The convolution represents the percentage of area of the filter g that overlaps the input f at a time τ over all time t

We notice that $\tau < 0$ is meaningless, and $\tau > t$ means predicting future.

We can apply tighter bounds:

$$(f * g)(t) = \int_{0}^{t} f(\tau)\, g(t-\tau)\, d\tau \quad \Rightarrow \text{For 1 } t \text{. Need to take all } \tau\text{s. in } f * g$$

In multiple dimensions:

$$(I * h)(x,y) = \int_{0}^{x}\int_{0}^{y} I(i,j) \cdot h(x-i, y-j)\, di\, dj \quad \text{(slide image over kernel)}$$

image input — kernel

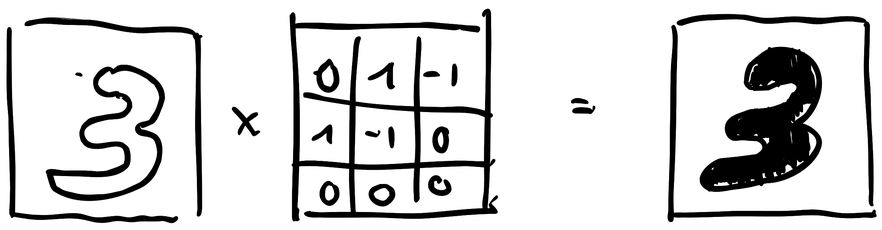
$$= \int_{0}^{x}\int_{0}^{y} I(x-i, y-j) \cdot h(i,j)\, di\, dj \quad \text{(slide kernel over image)}$$

↳ Typicall less values of x and y in a kernel than on image. Use 2nd form.

The result will be a scalar value. We repeat the convolution (x, y) for all set of points (x, y), and store it in a convolved matrice noted $I * h$.

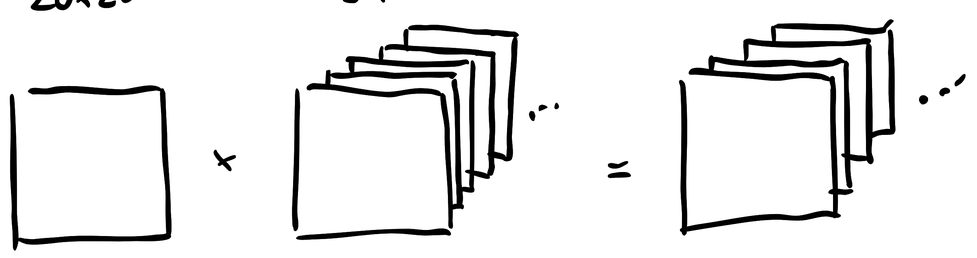
Example:

- Image 28×28 × Filter 3×3 = Convolved Output 26×26 ↳ $(28-3+1)$



For example.

- Image 28×28 × Filters 32 3×3 = Convolved output $26 \times 26 \times 32$



We are creating an output volume: the feature map.

How do we define the filters ?

- Some can be defined to find curves
- others sharp edges
- others textures
- ...

Most of the features used especially in the deeper layers, are not interpretable by humans

II. Activation layer.

Only non-linear activation functions are used for learning.
(Otherwise, $A_1 * (A_2 * X) = (A_1 * A_2) * X = A * X$, and several layers are as effective as a single one)

Typically, **reLU** and **Leaky reLU** (avoiding dying reLU problem) are used for activation functions

- A rectified linear unit (ReLU) is an activation function of the form:
  $f(x) = x^+ = \max(0, x)$ ⇒ All negative values become 0 immediatly.
- Leaky ReLU: $f(y) = y$ when $y > 0$, αy when $y < 0$.
- Sigmoid function: $\sigma(z) = 1/(1+e^{-z})$
- Tan H: $f(x) = \tanh(x) = 2/(1-e^{-2x}) - 1$.
- Softplus: $f(x) = \log_e(1+e^x)$

Activation functions are used to **determine the output of neural network** by **mapping the resulting values in between two finite values**. (typically 0,1 or -1,1).

NB: Order of layers can be switched:

reLU(MaxPool(Conv(M))) = MaxPool(reLU(Conv(M)))

III. Pooling layer.

Pooling involves a down sampling of features so that we need to learn less parameters when training. Typically, two hyperparameters introduced with the pooling layer:

- dimension of spatial extent : value of n for which we can take a $n \times n$ feature representation and map to a single value
- stride : how many features the sliding window skips along width and height

A common pooling layer uses a 2×2 max filter with a stride of 2. This is a non-overlapping feature

Single depth slice

1	1	2	4
5	(6)	7	(8)
(3)	2	1	0
1	2	3	(4)

→

6	8
3	4

Since pooling is applied to every layer within the 3D volume, the depth of the feature map after pooling will remain unchanged.

Performing pooling reduces the chances of overfitting, as there are less parameters we can also use an average filter, but max works better in practice.

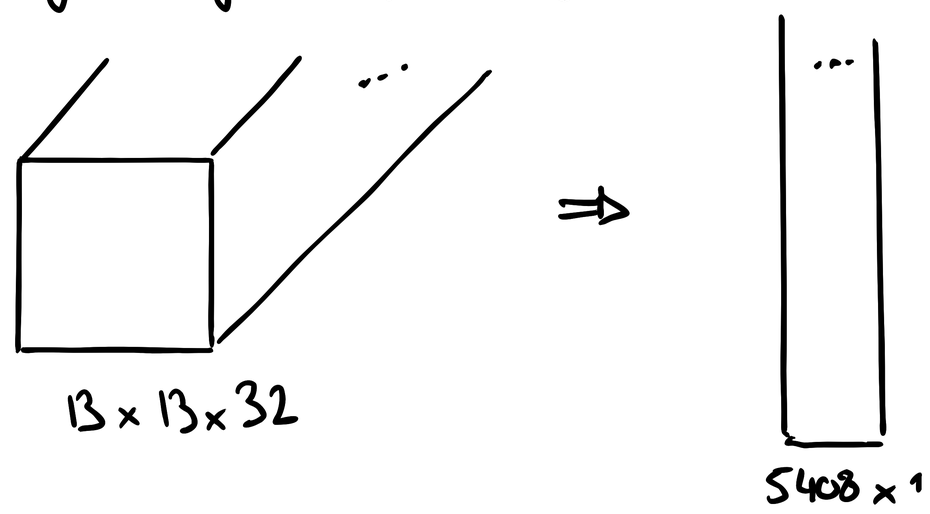
IV. Fully connected layer.

The output from the convolution layers represent high level features in data.
While that output could be flatened and connected to the output layer, adding a fully connected layer is usually a cheap way of learning non-linear combinations of these features.

The convolutional layers provide meaningful, low dimensional, invariant feature space, and fully connected layer is learning a possibly non-linear function in that space.

Question is: How to convert the 3D feature map pooling output to a 1D feature vector?

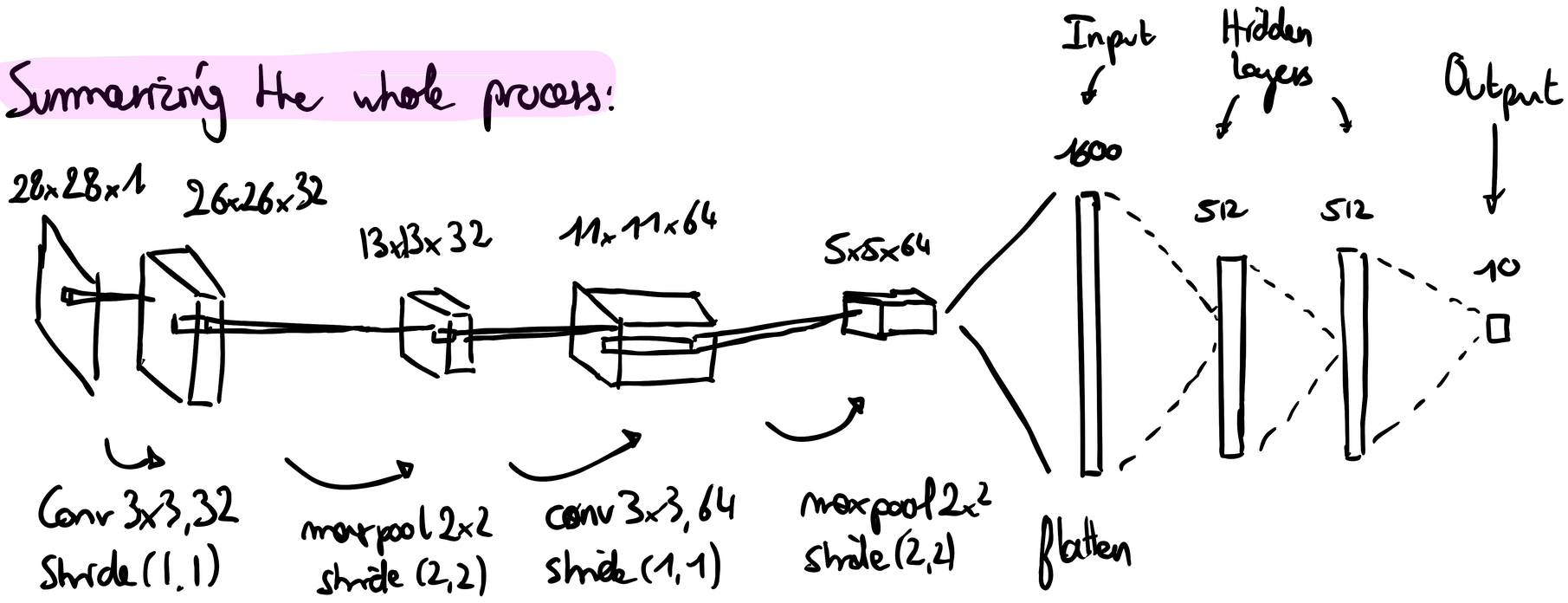
The 1D feature vector will be the input of the FC layer.
The 3D volumes are usually very deep at this point, because of the increased number of kernels that are introduced at every convolution layer.
We go through a *flattening 3D volume conversion*.



Is now in an admissible form for FC layer.

From now on, we can build a classical (FC) neural network.

Summarizing the whole process:



$$Out_{width} = \frac{Input_{width} - kwidth + 1}{Stride}$$

If not Integer: Keras has 2 parameters in MaxPool 2D:

- Valid: no padding for image border
- Same: Padding for image border.

Masked R-CNN

Masked R-CNN is an approach to instance segmentation. It does combine 2 subproblems:

- Object detection : detect the number of objects per image
- Semantic segmentation: understanding of an image at a pixel level.
  ↳ Assign object class to each pixel in image.

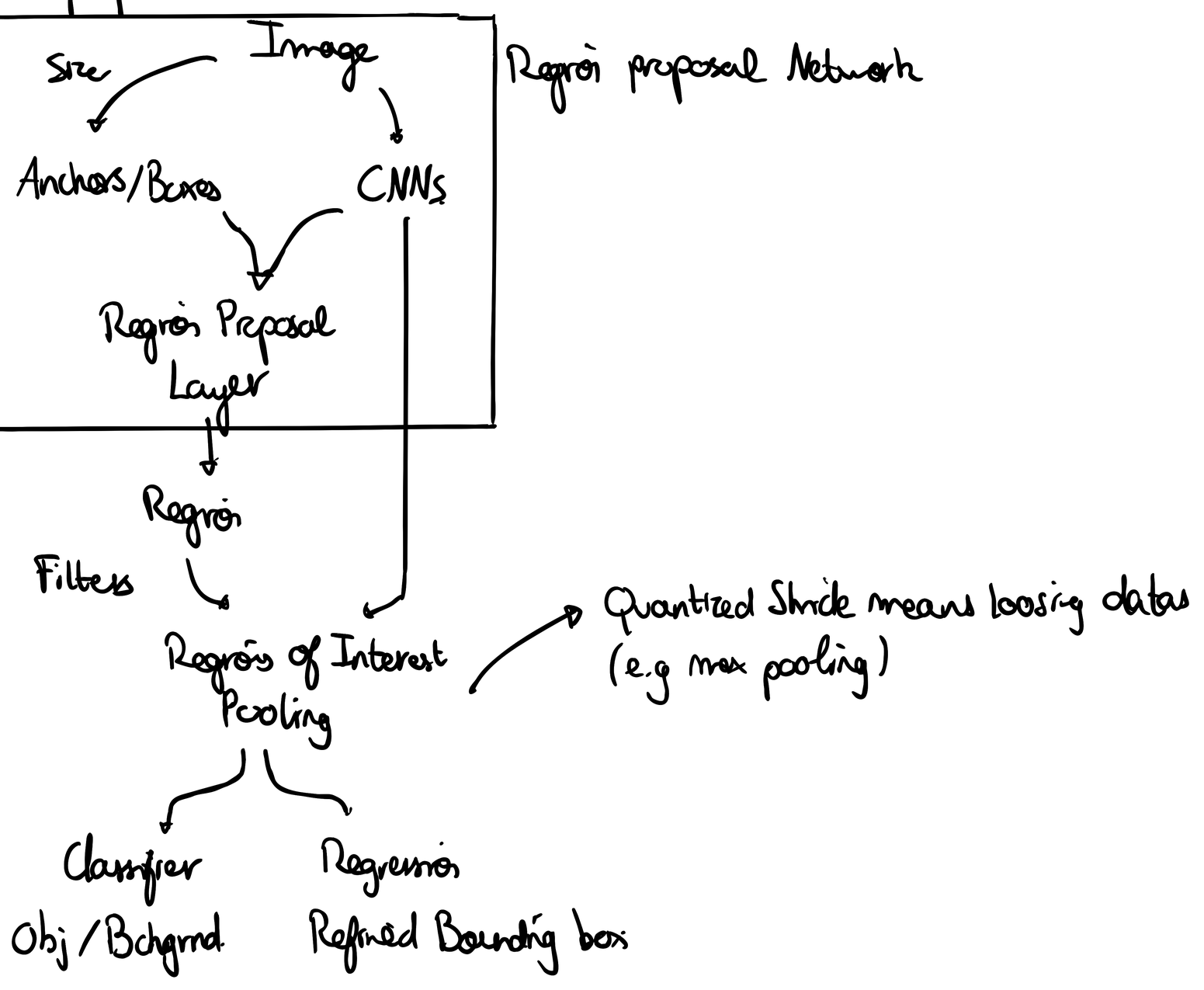
I. Object detection

Object detection is usually done using FasterR-CNN. We should start by defining what R-CNN (Region CNN) is

R-CNN is an approach to bounding box object detection that scans a number of object regions or regions of interest (ROI)

The Faster R-CNN performs a better job by incorporating an attention mechanism using a region proposal network (RPN)

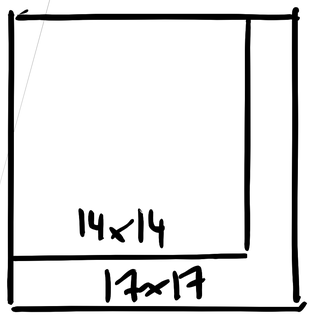
Process:



Pooling enables a model to become invariant to image rotation for example.
We introduced the notion of a stride being quantized. What does it mean ?

- Image 17×17 $\xrightarrow{\text{ROI pooling}}$ 7×7 $\Rightarrow$ Stride $\approx \frac{17}{7} = 2,42$

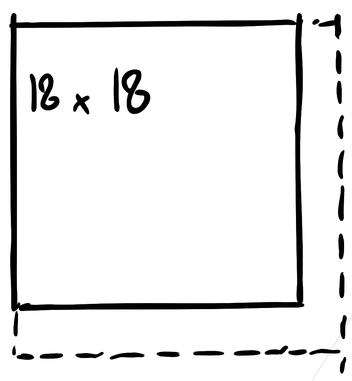
  Need to map it ↗ $\quad$ ↳ $\text{stride}_{\text{ROI pooling}} = [2,42] = 2$



Leads to loss of data.

- Image 18×18 $\xrightarrow{\text{ROI}}$ 7×7 $\Rightarrow$ Stride $\approx \frac{18}{7} = 2,57$

  ↳ $\text{stride}_{\text{ROI pooling}} = [2,57] = 3.$



Leads to mis-alignment.

To avoid these issues, ROI Align is used. Strides are not quantized.
How do we handle the decimals ?

- Each cell is divided into a 2×2 bin, and each of the subcell is pooled through bi-linear interpolation
- Final cell value computed by either an average or the maximum over the 4 sub-values

II. Semantic segmentation.

Output object mask (shape drawn) using pixel to pixel alignment. This mask is a binary mask outputted for each region of interest.
For each ROI, a binary mask $m \times m$ is built. While computing the mask, say it could be k objects, a loss of km^2 is occured.. FCN are used to predict mask, as it retains spatial position of objects.