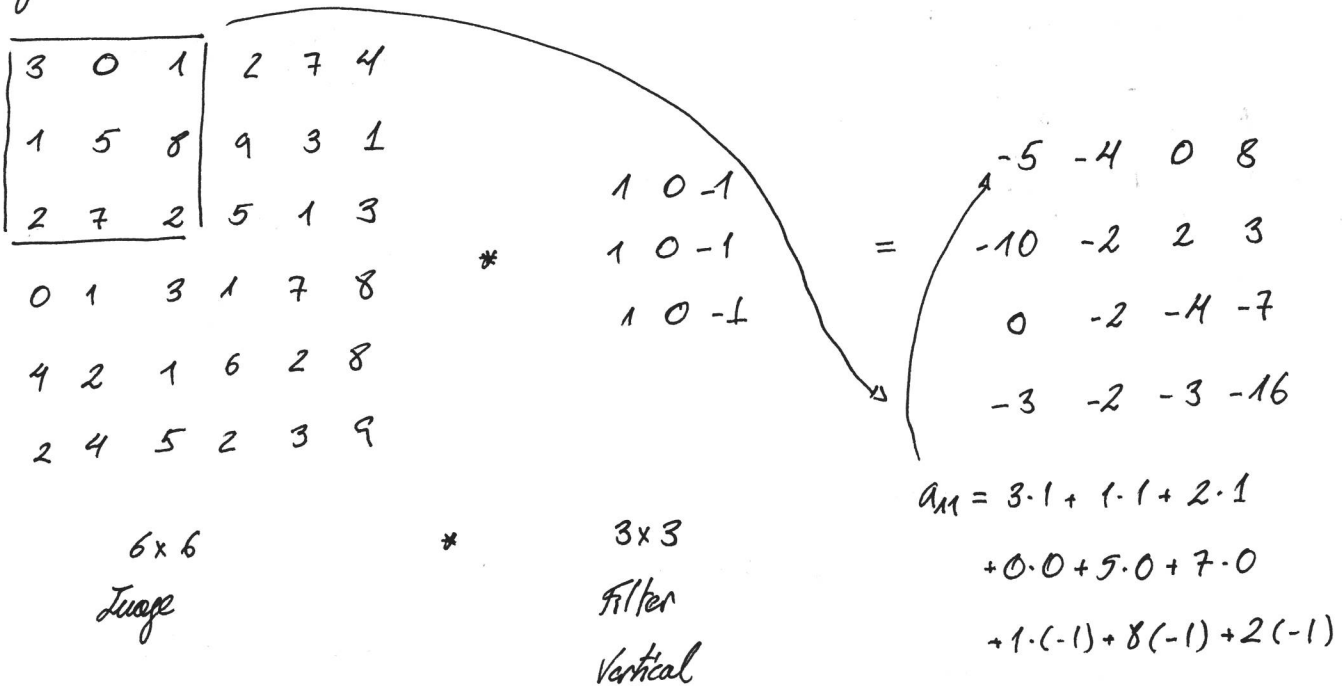
Convolutional Neural Networks

* Edge detection →



6×6 Image * 3×3 Filter Vertical

Horizontal Filter 3×3	Sobel Vert Filter 3×3	Schorr Vert Filter 3×3	Learned Param NN Edge detect.
1 1 1	1 0 -1	3 0 -3	w_1 w_2 w_3
0 0 0	2 0 -2	10 0 -10	w_4 w_5 w_6
-1 -1 -1	1 0 -1	3 0 -3	w_7 w_8 w_9

↓ Not necessarily 0°, 90°
but also 45°, 70°, 85°

rgn vert vs Horz.

* Padding →

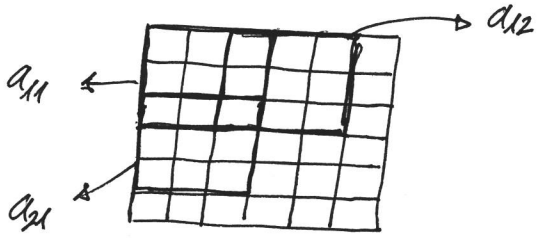
- If padding is not used, pixel at borders are not used as much.
- Prevents image shrinking, also keeps information of borders.
- Valid convolution: "No padding" → $n \times n * f \times f \rightarrow (n-f+1) \times (n-f+1)$
- Same convolution → Out size = In size.

$(n+2p-f+1) \times (n+2p-f+1) \rightarrow n+2p-f+1 = n \rightarrow p = \frac{f-1}{2}$

f = usually odd number.

* Strided Convolution →

Stride = 2



Resulting dimension →

$$\left\lfloor \frac{n+2p-f}{s}+1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s}+1 \right\rfloor$$

$\lfloor z \rfloor = \text{floor}(z)$

↳ If the convolution doesn't fit the matrix, convention is not to do it.

* Mathematical convolution →

A. The filter is flipped both vertically.

B. In here we skip this step, which technically is cross-correlation.

* Convolution over volume →

RGB images

6 × 6 × 3 * 3 × 3 × 3 = 4 × 4 → a_{11} = Red conv + Green conv + Blue conv

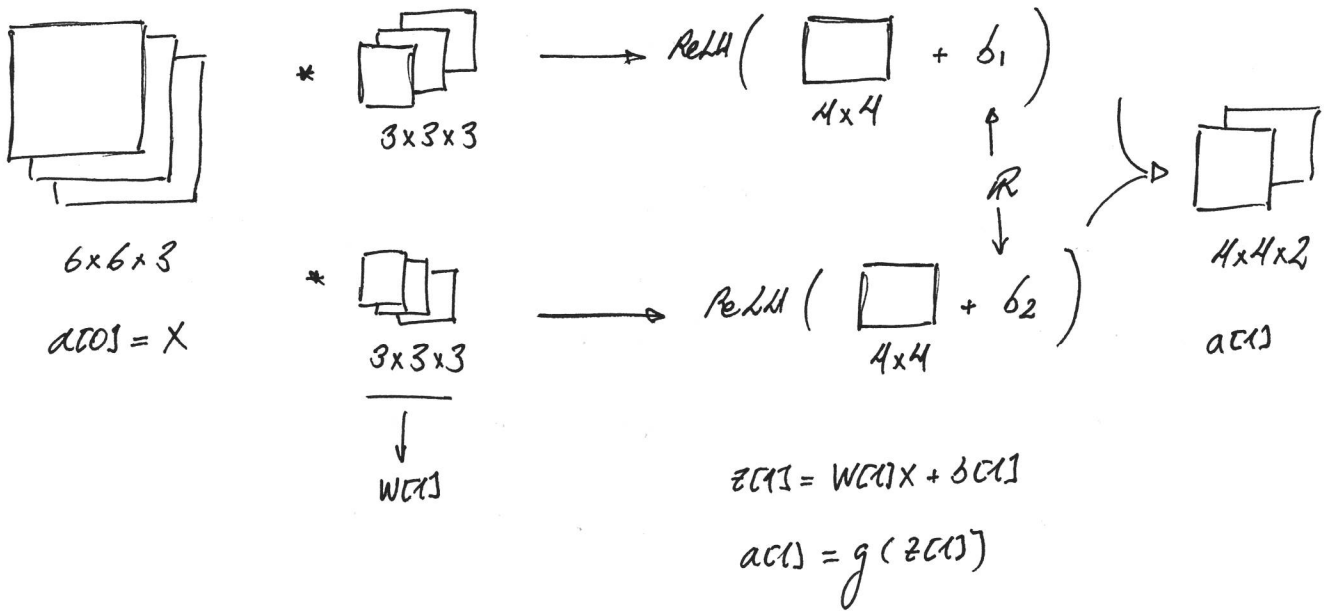
27 num = 3^3

- Multiple filters →

6 × 6 × 3 * 3 × 3 × 3 = 4 × 4

* 3 × 3 × 3 = 4 × 4 → 4 × 4 × 2 (Number of filters.)

- One layer CNN →



$z^{[1]} = W^{[1]} X + b^{[1]}$

$a^{[1]} = g(z^{[1]})$

2 filters ; each 3×3×3 → Parameters?

3×3×3 + bias / filter = 28 param / filter ⟹ 28 · 2 = 56 parameters.

* Notation →

- layer $l \equiv$ convolutional layer.
- $f^{[l]} \equiv$ filter size
- $p^{[l]} \equiv$ padding
- $s^{[l]} \equiv$ stride
- $n_c^{[l]} = \#$ filters.

Input →

$n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$

Output →

$n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

$$n_H^{[l]} = n_W^{[l]} = \left\lfloor \frac{n^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

Filter →

$f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$

Weights →

$f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$

Activations →

$a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

$A^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]} \times m$

Bias →

$n_c^{[l]}$

- Pooling layers →

1	3	2	1	3
2	9	1	1	5
1	3	2	3	2

Hyperparameter → $f = 3$; $s = 1$.

9	9	5
9	9	5
8	6	9

$\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$

8	3	5	1	0
5	6	1	2	9

$3 \times 3 \times 2$ → filter.

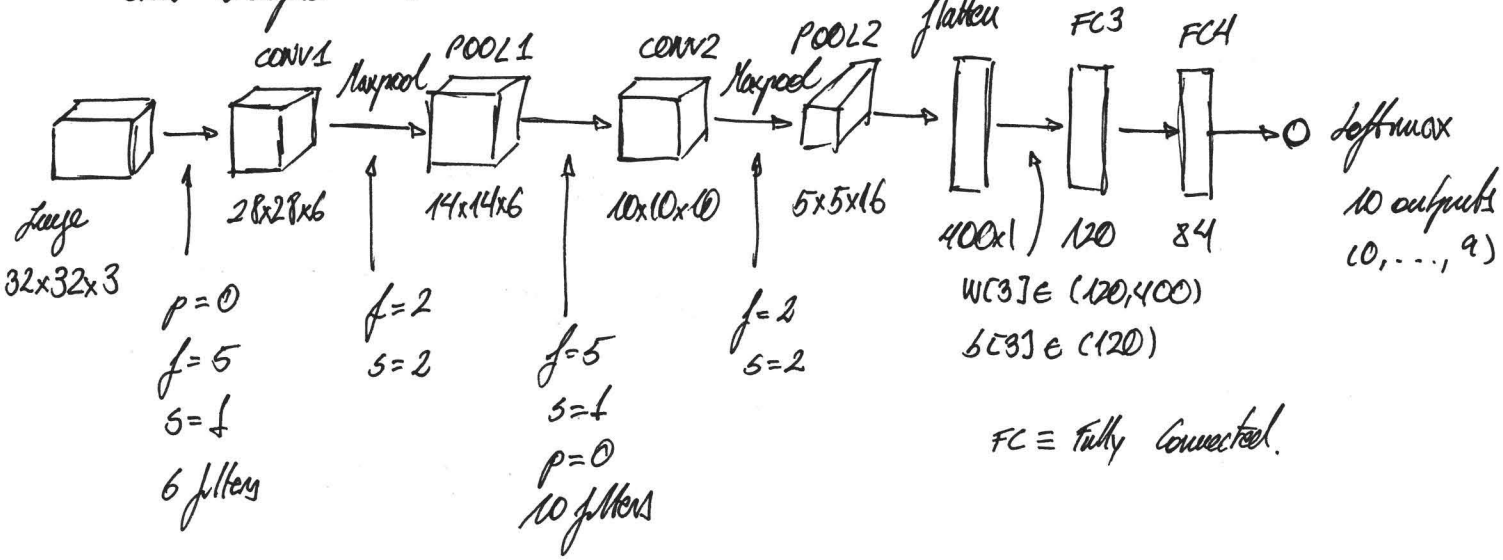
$5 \times 5 \times 2$ → $3 \times 3 \times 2$ → resulting.

* Good for detecting determinate features.
* No parameter to learn. → But it affects backpropagation calculation.
* Average pooling → Take average instead of max.

$n_H \times n_W \times n_c \rightarrow \left\lfloor \frac{n_H - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_W - f}{s} + 1 \right\rfloor \times n_c$

$\left\lfloor \frac{n_H - f}{s} + 1 \right\rfloor$ $\quad$ $\left\lfloor \frac{n_W - f}{s} + 1 \right\rfloor$

- CNN Example →



* n_H, n_W ↓
* n_c ↑

	Activation Shape	Activation size	# parameters
Input	(32, 32, 3)	3,072	0
conv 1	(28, 28, 8)	6,272	208
Pool 1	(14, 14, 8)	1,561	0
Conv 2	(10, 10, 16)	1600	416
Pool 2	(5, 5, 16)	400	0
FC 3	(120, 1)	120	48,001
FC 4	(84, 1)	84	10,081
Softmax	(10, 1)	10	841

↓ Decreases.

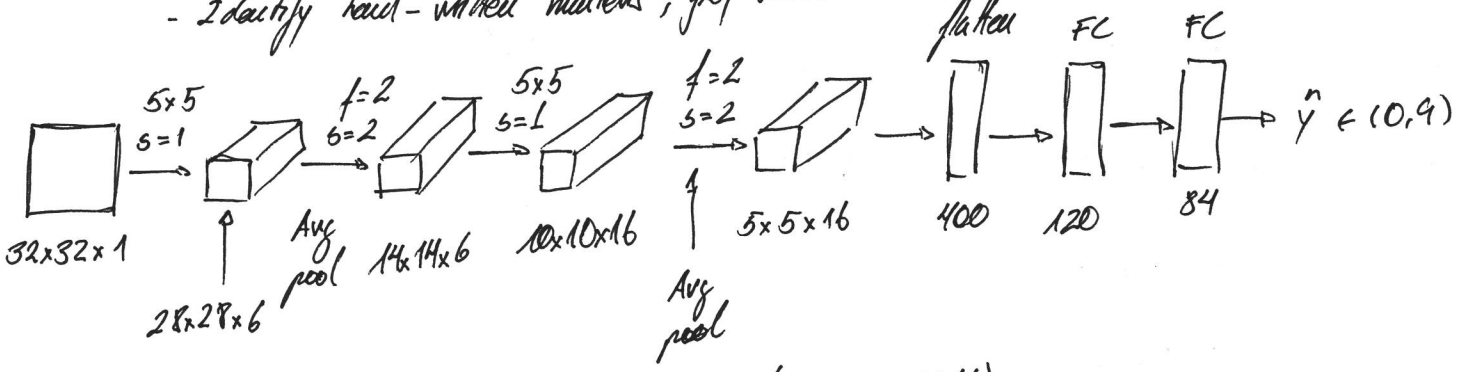
→ Most of the parameters.

- Parameter sharing →
  * Convolutional NN favors reducing the total number of parameters, reducing overfitting.
  * It also allows a feature detector to be used in multiple locations throughout the whole input image/volume.
- Sparsity of connections →
  * Each activation in the next layer depends only a small number of activations from the prev. layer.

- Case Studies →

* LeNet 5 →

- Classify hand-written numbers; grey scale.



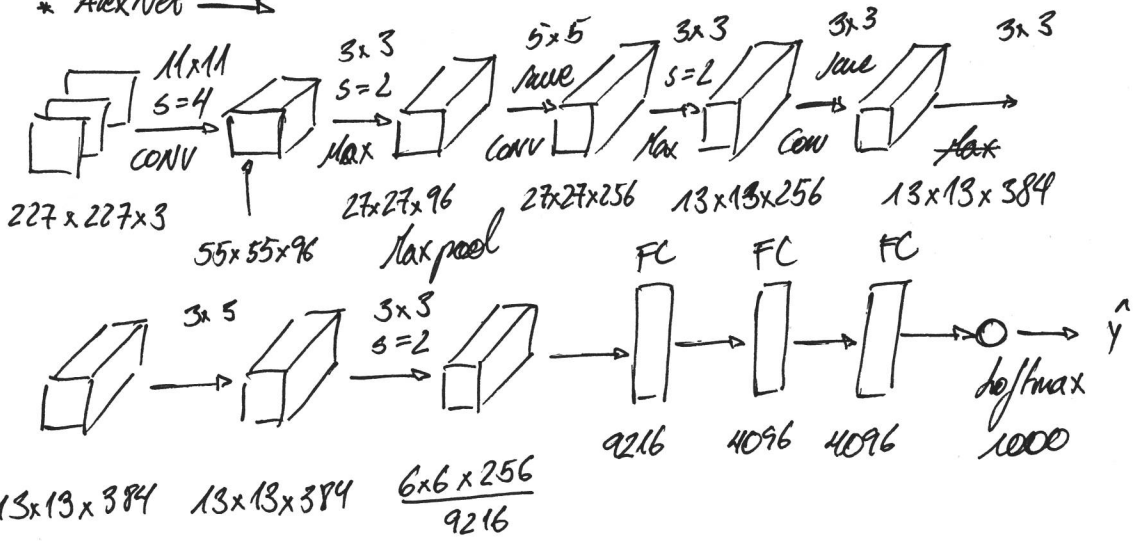
- 60k parameters → small compare to now (10M).

- $n_H, n_W \downarrow$; $n_C \uparrow$

- One or more → conv + pool

- Activation function → typically tanh ← → ReLU (now)

* AlexNet →



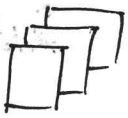
- $16 \cdot 10^6$ parameters.

- Similar but layer LeNet.

- Activation → ReLU.

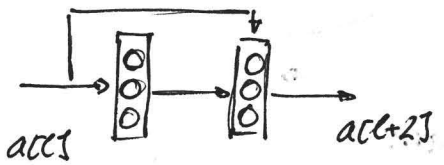
* VGG-16 →

- 16 layer with weights.
- Conv → 3×3 filter, s=1, same
- Max pool → 2×2, s=2
- $138 \cdot 10^6$ parameters.



224×224×3 → CONV 64 ×2 → 224×224×64 → POOL → 112×112×64 → CONV 128 ×2 → 112×112×128 → POOL → 56×56×128 → CONV 256 ×3 → 56×56×256 → POOL → 28×28×256 → CONV 512 ×3 → 28×28×512 → POOL → 14×14×512 → CONV ×3 → 14×14×512 → POOL → 7×7×512 → FC 4096 → FC 4096 → softmax 1000

- ResNet ≡ Residual Network →



(Other layer skips 3)

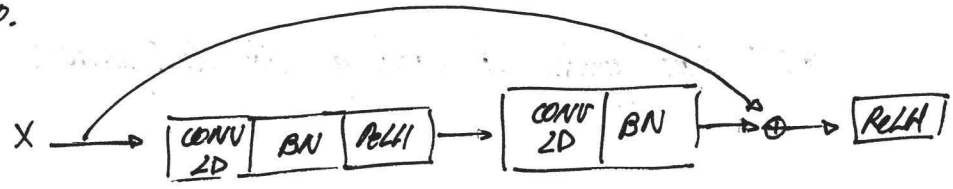
$$a[l+2] = g(z[l+2] + a[l])$$

$$z[l+2] = W[l+2]\, a[l+1] + b[l+2] =$$

$$= W[l+2]\, g(W[l+1]\, a[l] + b[l+1]) + b[l+2]$$

* If we assume $W[l+2] = b[l+2] = 0 \rightarrow a[l+2] = g(a[l])$

- easy for the ResNet to learn the identity function.
- it doesn't impact much the performance.
- it is not more difficult to learn the useful identity function as deep as you go.



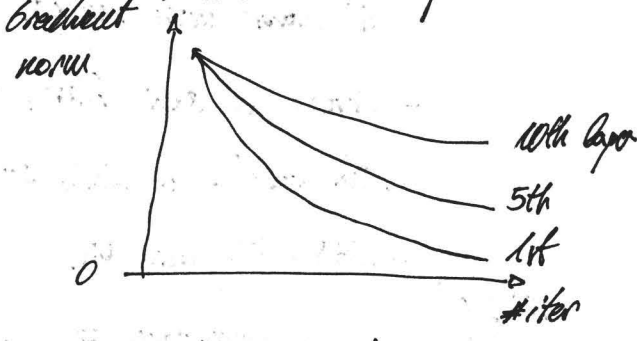
- Assumption $z^{[l+2]}$ & $a^{[l]}$ have same dimension ⟶ same convolution.

- In any other case ⟶

$$g(z^{[l+2]} + W_s a^{[l]})$$

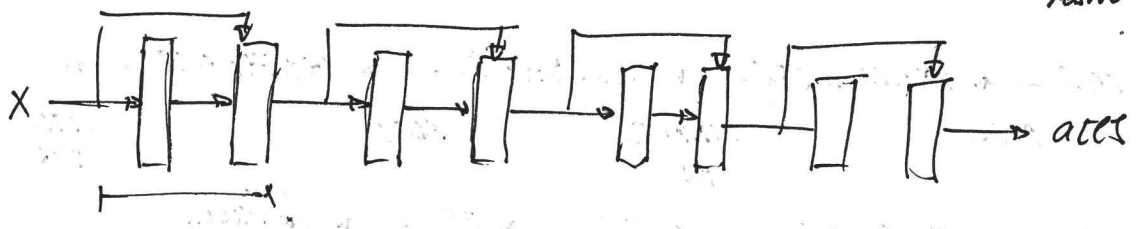
$z^{[l+2]}$: 256 ; W_s: $\mathbb{R}^{256 \times 128}$; $a^{[l]}$: 128

Problem with deep CNN



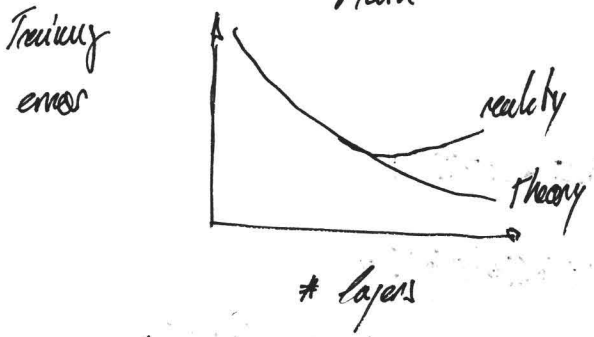
Vanishy gradient.

ResNet to address this.

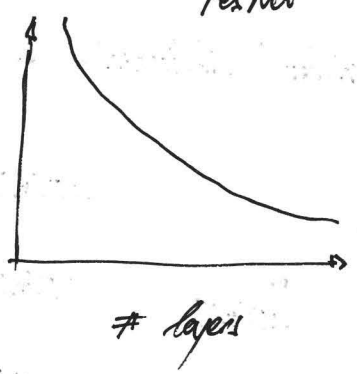


Residual block.

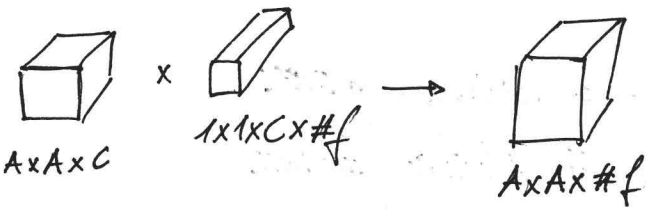
Plain



ResNet



- 1x1 Convolution ⟶



Pelll (fun (1x1 pro each of the channels).

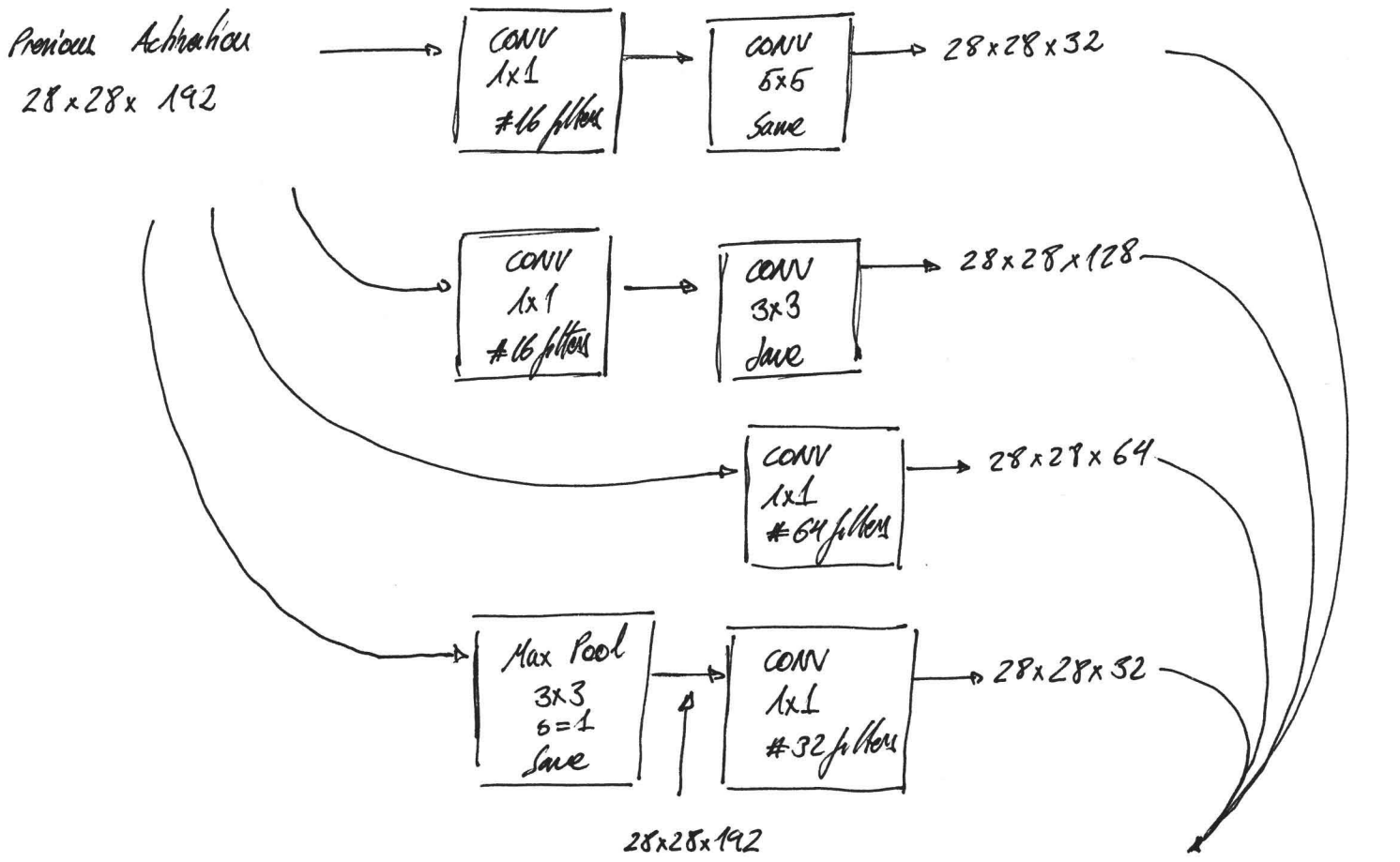
* Keeps n_W, n_H and can shrink # channels.

* If you keep # channels, it allows you to add non-linearity to the model.

* FC that applies to each of the AxA pictures, by the # filters.

- Inception Network →

* Inception block →



* Inception network as a NN based on these.

* This networks may have different branches with softmaxs →

helps predict an output cost and prevent overfitting.

* GoogLeNet

* Computational cost why and not why 1x1 →

28x28x192 → 28x28x32

CONV
5x5
Same
32 #f

(28x28x192)x(5x5x192) = 120M

28x28x192 → 28x28x16 → 28x28x32

CONV
1x1
16
1x1x192

(28x28x16)x(192) = 2.4M

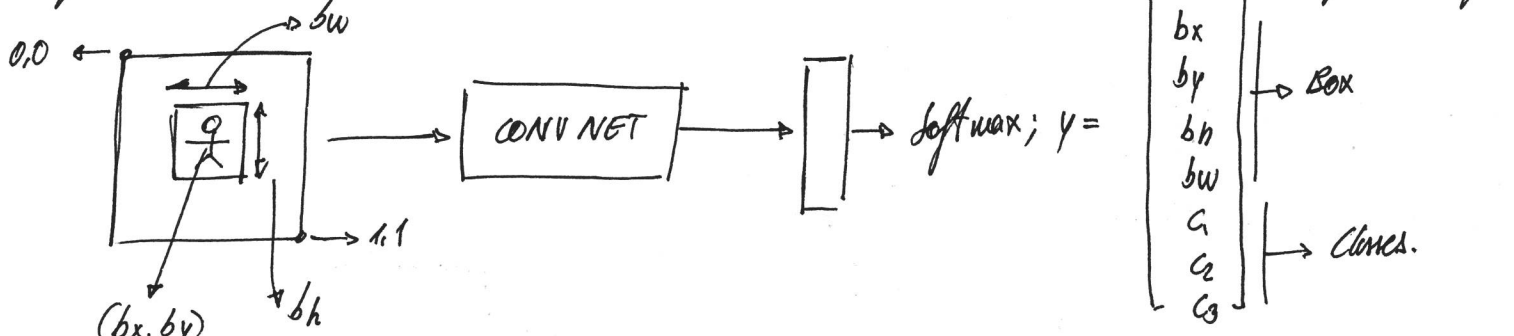
CONV
5x5
32
5x5x16

(28x28x32)x(5x5x16) = 10M

12.4M

Detection Algorithms

* Object detection →



* If $P_c = 0$; rest don't care.

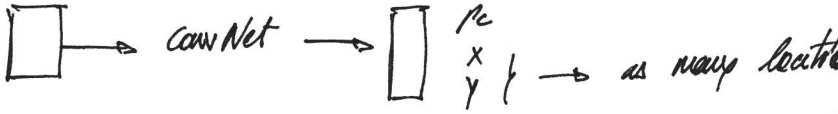
- If we define the loss function as squared error →

$$\text{If } y_1 = 1 \rightarrow \sum_{i}^{8} (y_8 - \hat{y}_8)^2$$

$$\text{If } y_1 = 0 \rightarrow (y_1 - \hat{y}_1)^2$$

* Landmark detection →

- Uses a NN to extract x, y locations in an image.
- Basic node for face recognition ⇒ Need to label training set by hand.

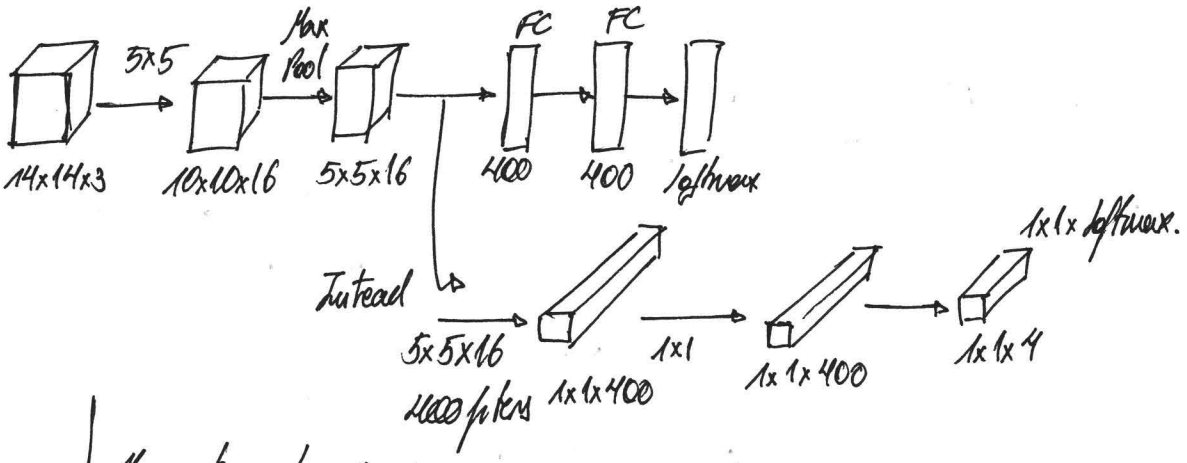


* Object detection →

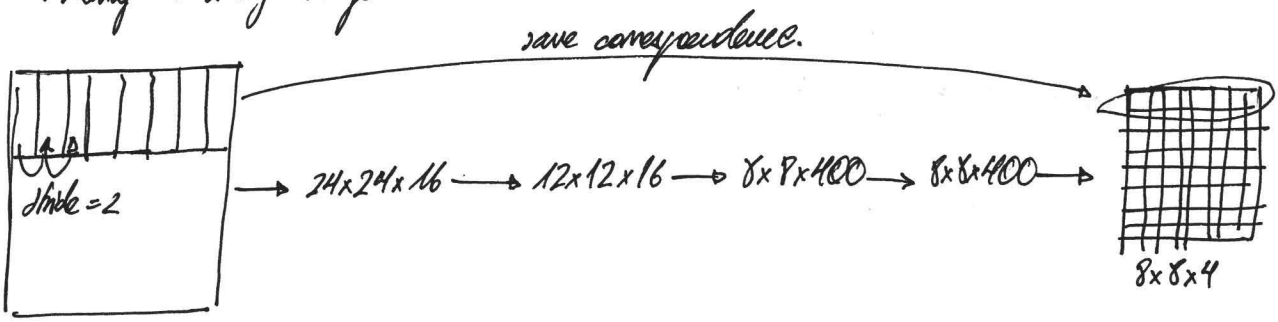
- On training set feed close cropped images, mostly image of object.
- Then use sliding window detection → This has a huge computational cost.

* Convolutional implementation of sliding windows →

- 1st idea → Turn FC into a CONV layer.



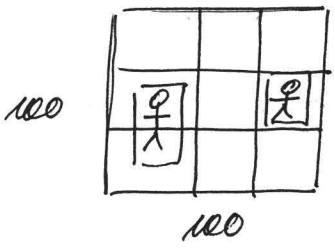
↳ Having to a big image



* Most of the calculation has already been done, highly repetitive.
* Instead passes the whole image.

* Bounding box prediction →

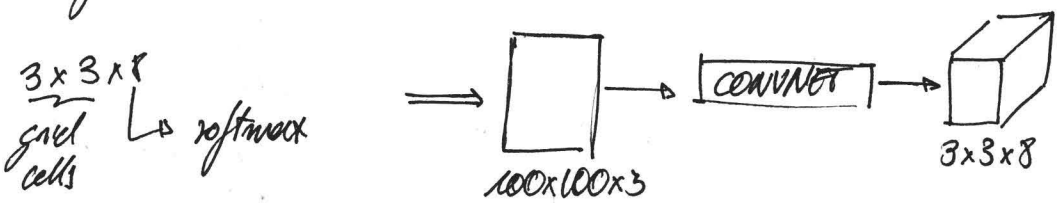
- YOLO Algorithm →



* Divide image in grid of cells and object localization for each cell
* if assign objects to a cell
* For each grid cell label y →

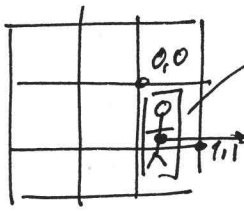
$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_H \\ b_W \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

* So the resulting volume should be →



* Encoding the bounding boxes →

$$y = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

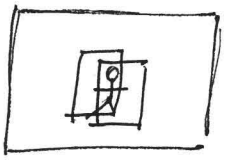


$b_w = 0.5$; $b_H = 1.1$ can be greater like in this case.

$b_x \geq 0.5$
$b_y \geq 0.1$ } → always between 0,1

- Intersection over Union →

* Used in YOLO as a way to evaluate object detection.



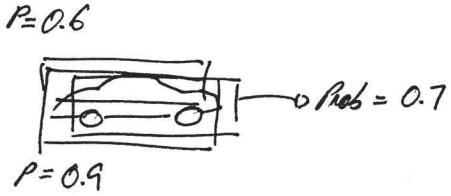
* For each of the anchor boxes, it will be used with the one of highest prob → Removing all boxes that have a large overlap.

 Union  Intersection → $IoH = \frac{\text{Area Intersection}}{\text{Area Union}}$

If IoH > iou_threshold → Remove box

- Non-max Suppression →

* Scenario → One object with several bounding boxes



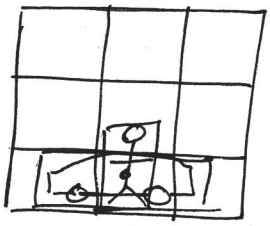
* Algorithm →

1. Takes the bbox with highest prob.
2. For each remaining →
   - Calculates IoH, discard if > threshold

* Each cell of the grid → ~~(One object)~~ output $(p_c, b_x, b_y, b_H, b_w)$

- Anchor boxes →

* Overlapping objects →



Center both objects in same grid cell

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_H \\ b_W \\ c_1 \\ c_2 \\ c_3 \\ p_c \\ b_x \\ b_y \\ b_H \\ b_W \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

→ Anchor 1

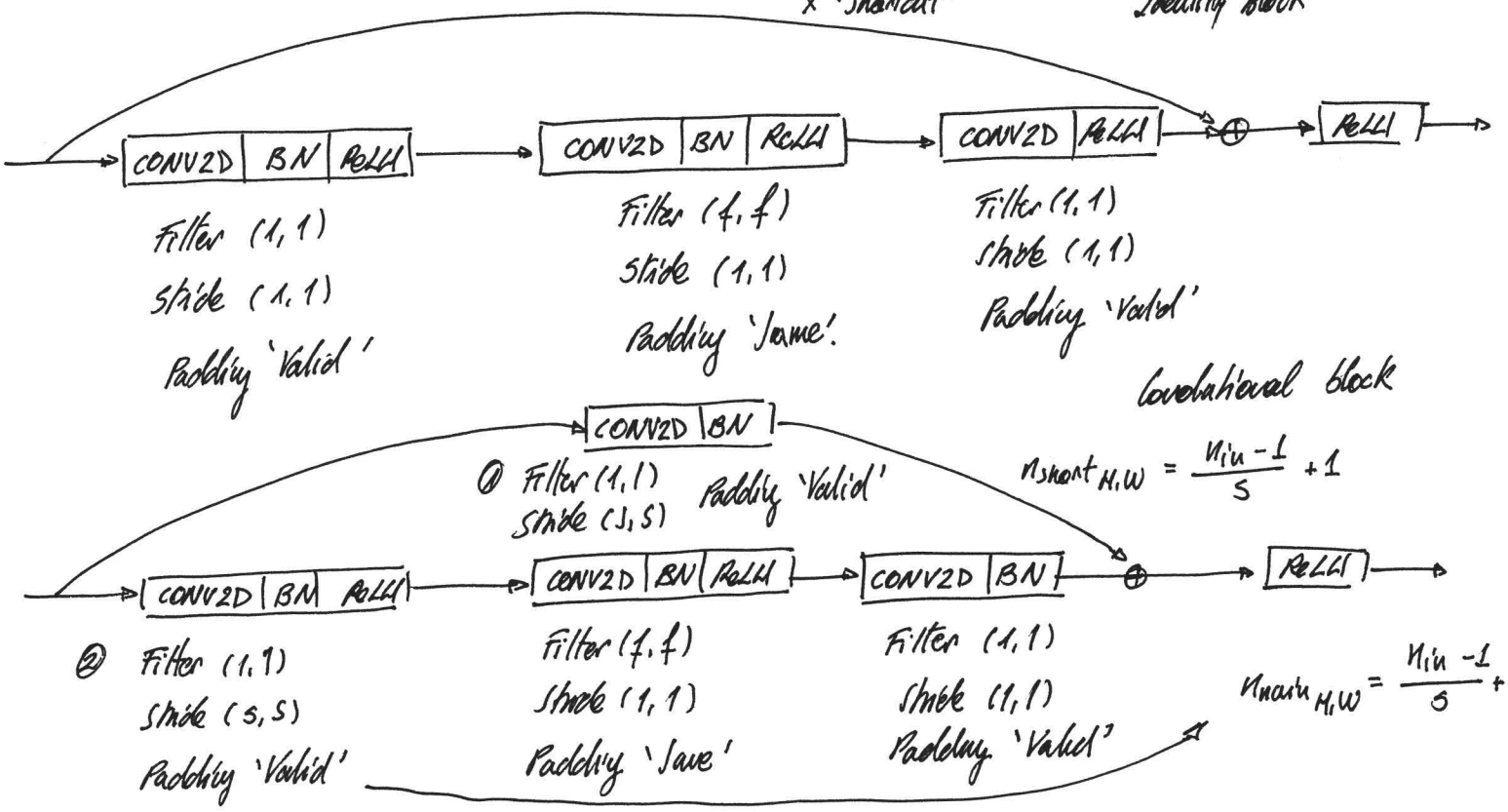
→ Anchor 2

→ Can encode information of 2 detections & Can.

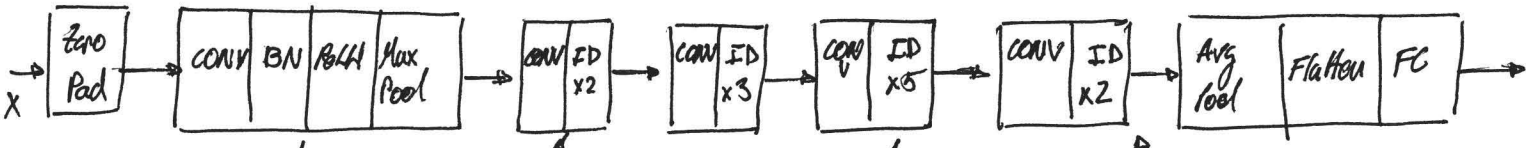
* Now each grid can have more than one output associated.

* Only works if 2 different object, each associated to different anchor boxes.

ResNet Implementation →



① Used to match up the addition, since filter path ② changes shape.



zero padding (3,3)

F_1 (7,7), s(2,2), #64

Max Pool (3,3), s(2,2)

F_{2C} (64×64×256)

$f=3, s=1$

F_{2ID} (64×64×256)

$f=3$

F_{3C} (128×128×512)

$f=3, s=2$

F_{3ID} (128×128×512)

$f=3$

F_{4C} (256×256×1024)

$f=3, s=2$

F_{4ID} (256×256×1024)

$f=3$

F_{5C} (512×512×2048)

$f=3, s=2$

F_{5ID} (512×512×2048

$f=3$

Avg Pool (2,2)

ResNet 50 layers.

Face Recognition

* Verification vs recognition →

Verification

Input image vs name, ID ⇒ 1:1

Output whether the image is of that person

⟷

Recognition

DB with K persons, but input image } 1:K

More prob of making mistake.

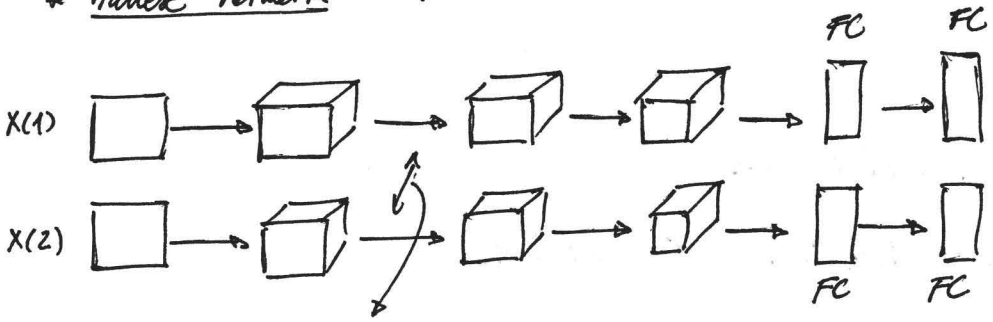
* One shot learning →

- Objective → Learn from one instance to recognize the person again.
- Use a similarity function →

$d(img1, img2)$ ≡ degree of difference between images

$d(img1, img2) \leq \tau$

* Siamese network →



$f(x(1))$ ≡ Input of encoding $x(1)$

$f(x(2))$

Same architecture.

$d(x(1), x(2)) = \| f(x(1)) - f(x(2)) \|_2^2$

* Train network so it encodes the similarity function.

* Parameter of NN define the encoding. →

If $x(i), x(j)$ are same person → $\| f(x(i)) - f(x(j)) \|_2^2$ is small.

* Paper → DeepFace, closing the gap to human level performance.

* Triplet loss function →

- Uses Anchor, Positive, Negative → $\|\ \|_2^2$ → Square Euclidean distance

$$\|f(A) - f(P)\|_2^2 \leq \|f(A) - f(N)\|^2 \qquad d = (f(A_1) - f(P_1))^2 + (f(A_2) - f(P_2))^2 + \dots$$

$d(A,P)$ $\qquad$ $d(A,N)$

- To prevent the network to put $f(\text{image}) = \vec{0}$ →

$\alpha \equiv$ margin ; forces the network to learn to fix a gap between P, N.

$$L(A,P,N) = \max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0)$$

↑ The NN shouldn't care how much negative if it is for no match cases, force 0.

- Training set → 10K pictures of 1K persons.

$$J = \sum_{i=1}^{m} L(A(i), P(i), N(i))$$

- Triplets →

* If A, P, N are chosen randomly →

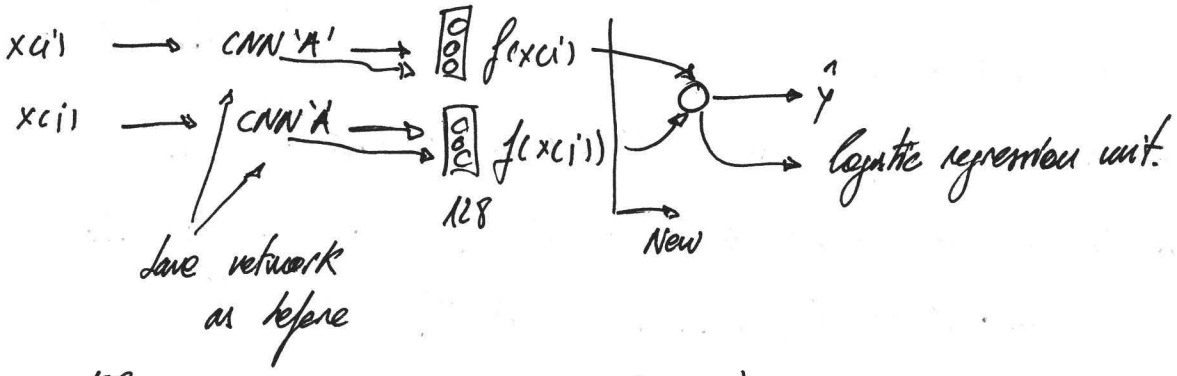
$d(A,P) + \alpha \leq d(A,N)$ is easily satisfied.

* Choose triplets 'hard' to train on →

$d(A,P) \geq d(A,N)$ → Network forced to learn this cases.

* Face verification & Binary classification →

- Another way of learning the similarity function.



$$\hat{y} = \sigma\left(\sum_{k=1}^{128} w_i \left| f(x^{(i)})_k - f(x^{(j)})_k \right| + b\right)$$

$$\chi^2 \text{ similarity} = \frac{\left(f(x^{(i)})_k - f(x^{(j)})_k\right)^2}{f(x^{(i)})_k + f(x^{(j)})_k}$$

- If the 2nd image belongs to a db, you can have those already computed. $f(x^{(j)})$

Neural Style Transfer →

* Deep ConvNet learning →

Image → 110 x 110 x 96 → 55 x 55 x 96 → 26 x 26 x 256 → 13 x 13 x 384 →
224 x 224 x 3

→ 13 x 13 x 384 → 6 x 6 x 256 → FC (4096) → FC (4096) → ŷ AlexNet like network.

- Paper → Visualizing and understanding convolutional networks. 2013, Zeiler and Fergus.

- When you visualize activation function outputs →

  * On the first layer, it look for edges, small features.

  * As you go deeper it identifies larger portions.

* Cost function →

Content C Style S

$J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G)$

Generated G.

- Find generated G →

1. Initiate randomly → G: 100 × 100 × 3 (Or whatever shape).
2. Use gradient descent to minimize G → $G = G - \frac{\alpha}{2G} J(G)$

Update on G values (Image output).

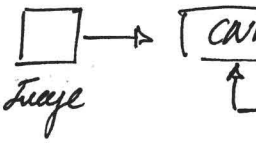
- Content cost function. →

* Use hidden layer l to compute cost → neither shallow, too deep.
* Pretrained CNN → E.g. VGG
* Let $a^{[l]}(C)$ and $a^{[l]}(G)$ be the activation function of layer l activations

$J_{content}(C,G) = \frac{1}{2} \| a^{[l]}(C) - a^{[l]}(G) \|^2$ → Measures how similar the a are.

element-wise, unroll them into vectors.

- Style cost function →



Use likely to find correlated ones in later layers. (Zones).

* Define the correlation between activation from different channels.
* Style matrix → $a^{[l]}_{i,j,k}$ ≡ Activation at (i, j, k). $G^{[l]}$ is $n_C^{[l]} \times n_C^{[l]}$

i: n_H, j: n_W, k: #C

$$G^{[l]}_{kk'} = \sum_{i}^{n_H^{[l]}} \sum_{j}^{n_W^{[l]}} a_{ijk}^{[l]} a_{ijk'}^{[l]}$$

$G^{[l]}_{kk'}$ ≡ Measures how correlated are activations in k with k'.

Point of this → be how correlated are the style in the generated image.

* Construct Gram matrix for style and generated ⟶

$G_{KK'}[l][l](S)$; $G_{KK'}[l][l](G)$; $G \equiv$ Gram Matrix.

$$J_{style}[l](S,G) = \| G[l](S) - G[l](G) \|_F^2 = \frac{1}{(2 n_H[l] n_W[l] n_c[l])^2} \left(G_{KK'}[l](S) - G_{KK'}[l](G) \right)^2$$

* For multiple layers ⟶

$$J_{style}(S,G) = \sum_l \lambda[l] J_{style}(S,G)$$

Neural style transfer example →

A. Neural style starts from a pretrained network and builds on top

B. Paper → VGG-19, VGG with 19 layers.

* Keras from model → dictionary with model in pretrained
'layer name' = object.

* feed image → model['input'].assign(image) ; image 600×800

* access activations of particular layer →

res.run(model['conv4_2']) → same convention as in prev assg.

C. Build algorithm on this →

hyperparameters that control relative weight of content and style

$J_{content}(C,G)$; $J_{style}(S,G)$ → $J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G)$

* Image generated image G match content image C (fixed image).

- choose some layer in between, not too shallow or deep.

- Image C → VGG forward. ↓ activation function of layer $a^{(C)}[l]$

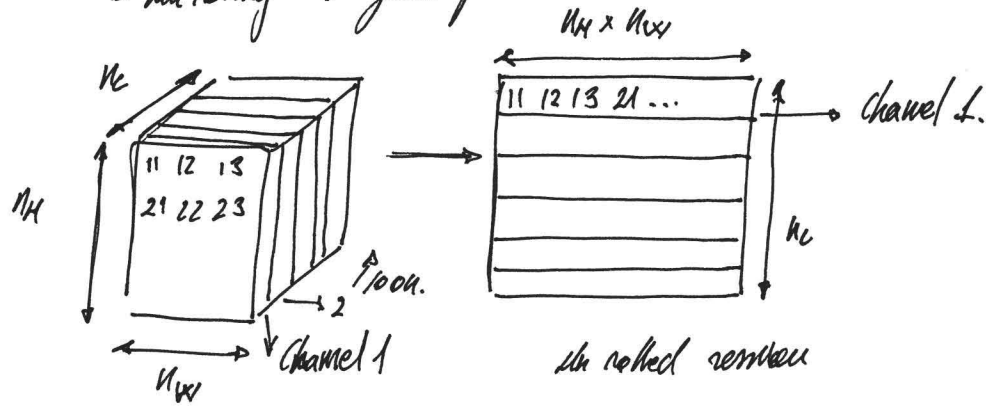
- Image G → VGG forward. ↓ $a^{(G)}[l]$

$$J_{content}(C,G) = \frac{1}{n_H \times n_W \times n_C} \sum_{\text{all entries}} (a^{(C)}[l] - a^{(G)})^2$$

1. measures how different $a^{[l]}(C)$ & $a^{[l]}(G)$ are.

2. Minimize cost → make G to have similar content as C.

- Use nothing → good practice.



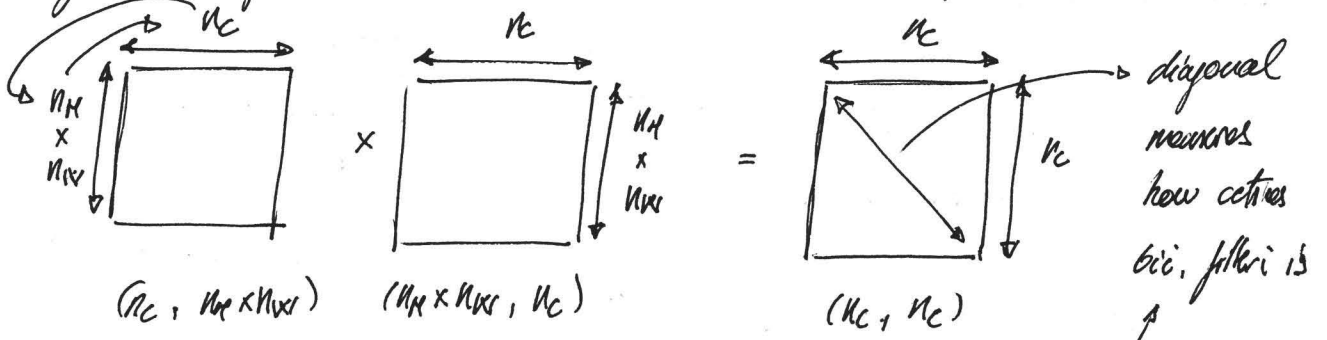
* Style Matrix.

- Gram Matrix ⟶ G of vectors $(v_1, \dots, v_n)$ is the matrix of the dot products.

$G_{ij} = v_i^T v_j = np.dot(v_i, v_j)$ ⟶ G from Gram, not to take for Generated G.

* Measures how similar are the vectors. As similar, as large.

- We generate it from the unrolled matrixes, G and G^T (transposed of G).



- Measures how similar the activation functions are.

* Style cost ⟶

- Goal is to minimize the distance between Gram matrix of the Style matrix and the Generated image.

$$J_{style}^{[l]}(S, G) = \frac{1}{4 \cdot n_C^2 \cdot (n_H \cdot n_W)^2} \sum_{i=1}^{n_C} \sum_{j=1}^{n_C} \left(G_{ij}^{(S)} - G_{ij}^{(G)} \right)^2$$

$[l]$ ⟶ layer; $G^{(S)}$ ⟶ Gram Style; $G^{(G)}$ ⟶ Gram Generated.

D. Style Weights ⟶

* So far, only for one layer. Better results for several different layers.

$$J_{style}(S, G) = \sum_l \lambda^{[l]} J_{style}^{[l]}(S, G)$$

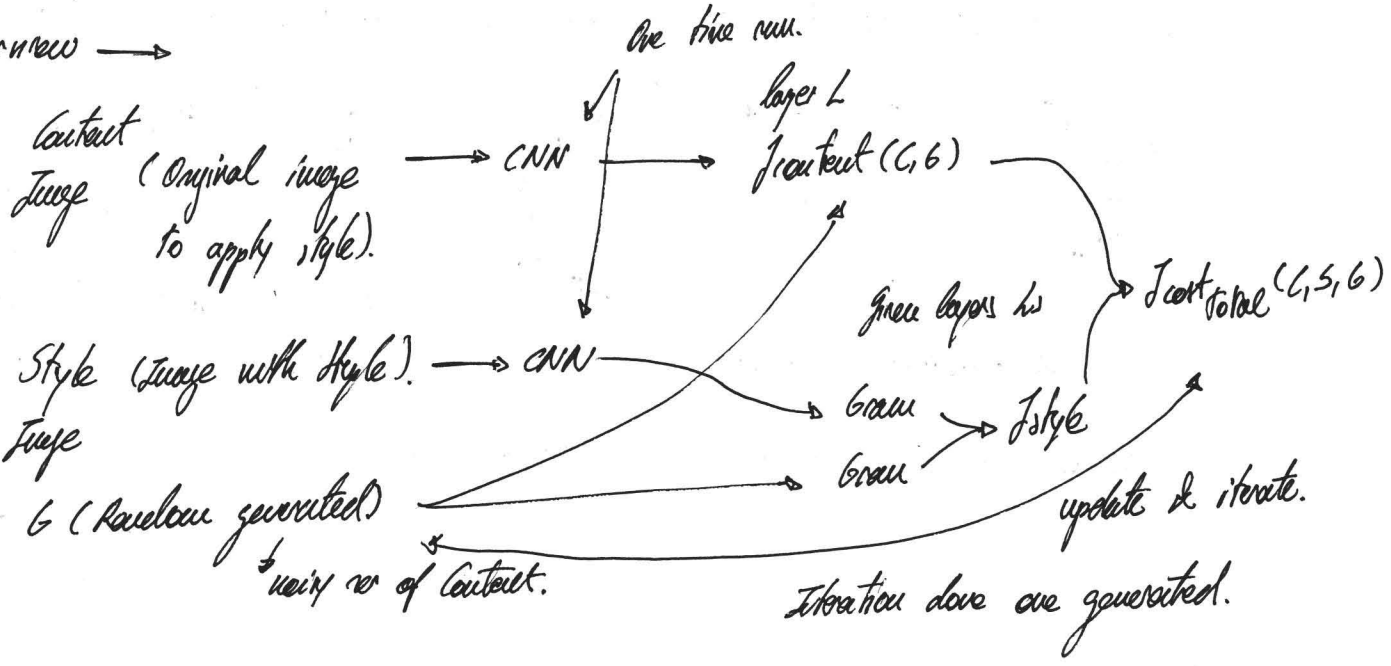
* Assign weights per layer

* Style of an image can be represented by the gram of a hidden layer function.
  Combining a few gives a better result.

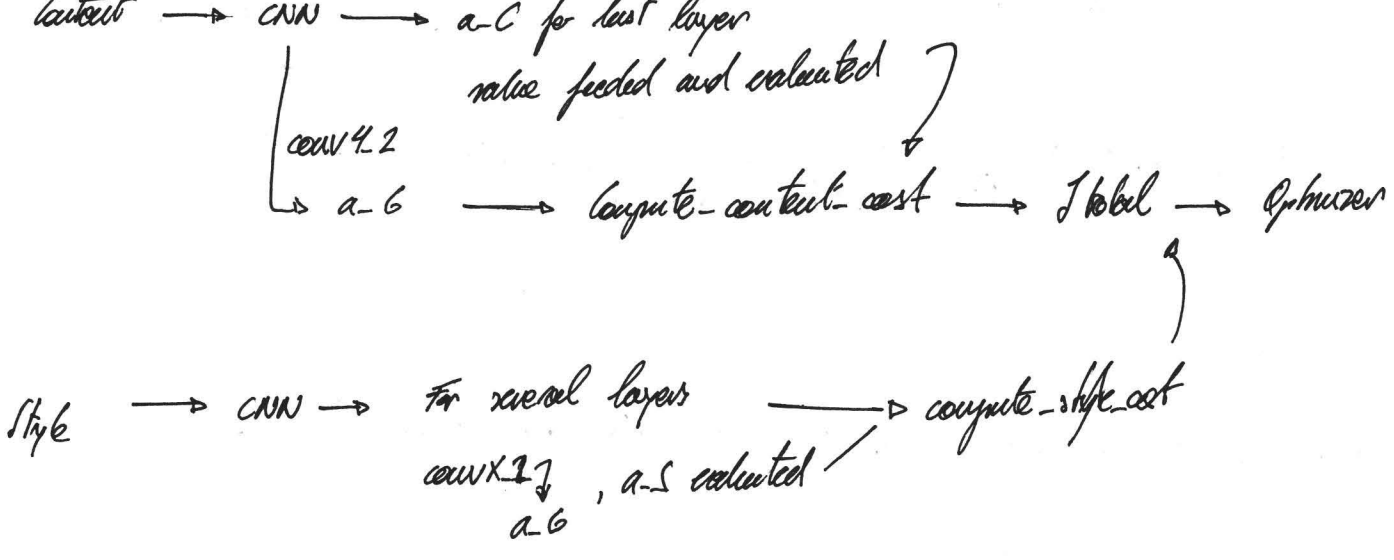
* For the content representation, just one layer is sufficient.

* Minimizing the style function → makes G to then follow the style of S.

D. Overview →



E. Tensor flow graph →



Feed generated image → model → Gets values at the layers
& run.

← run(Optimizer); runs gradient descent and updates generated.
for X iterations

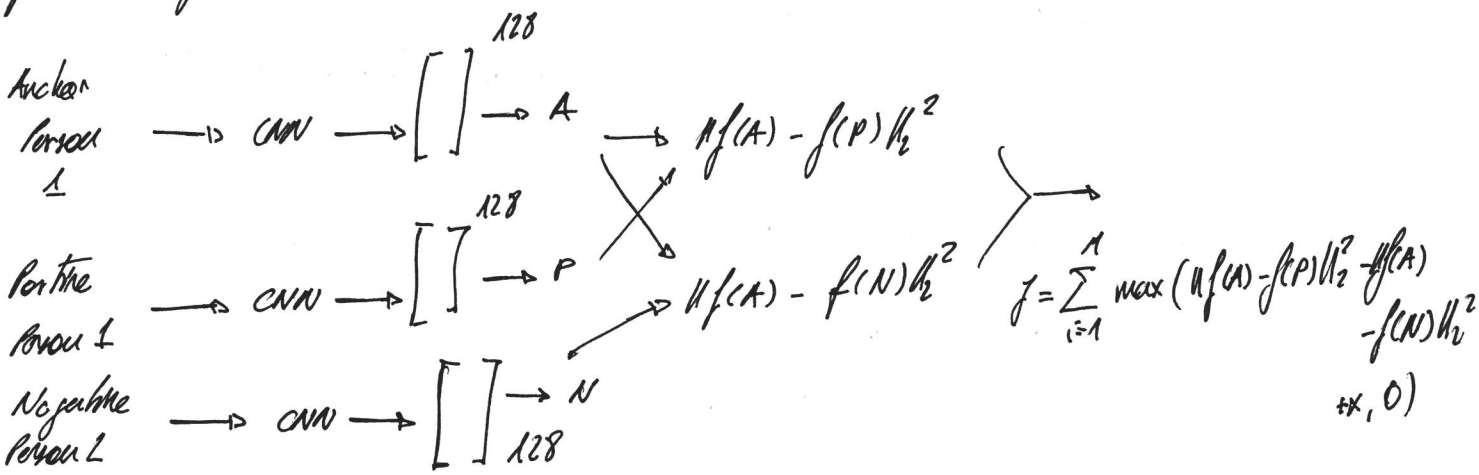
Face Recognition

* Using FaceNet model →

Image Face → [CNN inception model] → [ ] 128 Bit

- This model recognizes features from faces

* Triplet loss function →



$\|f(A) - f(P)\|_2^2 = \sum_{i=1}^{dim} (f(A(i)) - f(P(i)))^2$ = figure euclidean distance.

* Once define the loss function → Training.

- Objective is to pick images that are similar (P, N) so the network is trained on the hard cases.

Notes →

A. Tensorflow conv2d →

padding = 'same' →

new_height = ceil(float(in_height) / float(strides[1]))

new_width = ceil(float(in_width) / float(strides[2]))

padding = 'valid' →

new_height = ((input_height - filter_height + 2·P)/S) + 1

new_width = ((input_width - filter_width + 2·P)/S) + 1

B. Tensorflow conv2d_transpose →

padding = 'same' →

out_height = ceil(float(in_height) * float(strides[1]))

out_width = ceil(float(in_width) * float(strides[2]))

padding = 'valid' →

out_height = in_height * stride + filter_height - 1

out_width = in_width * stride + filter_width - 1.