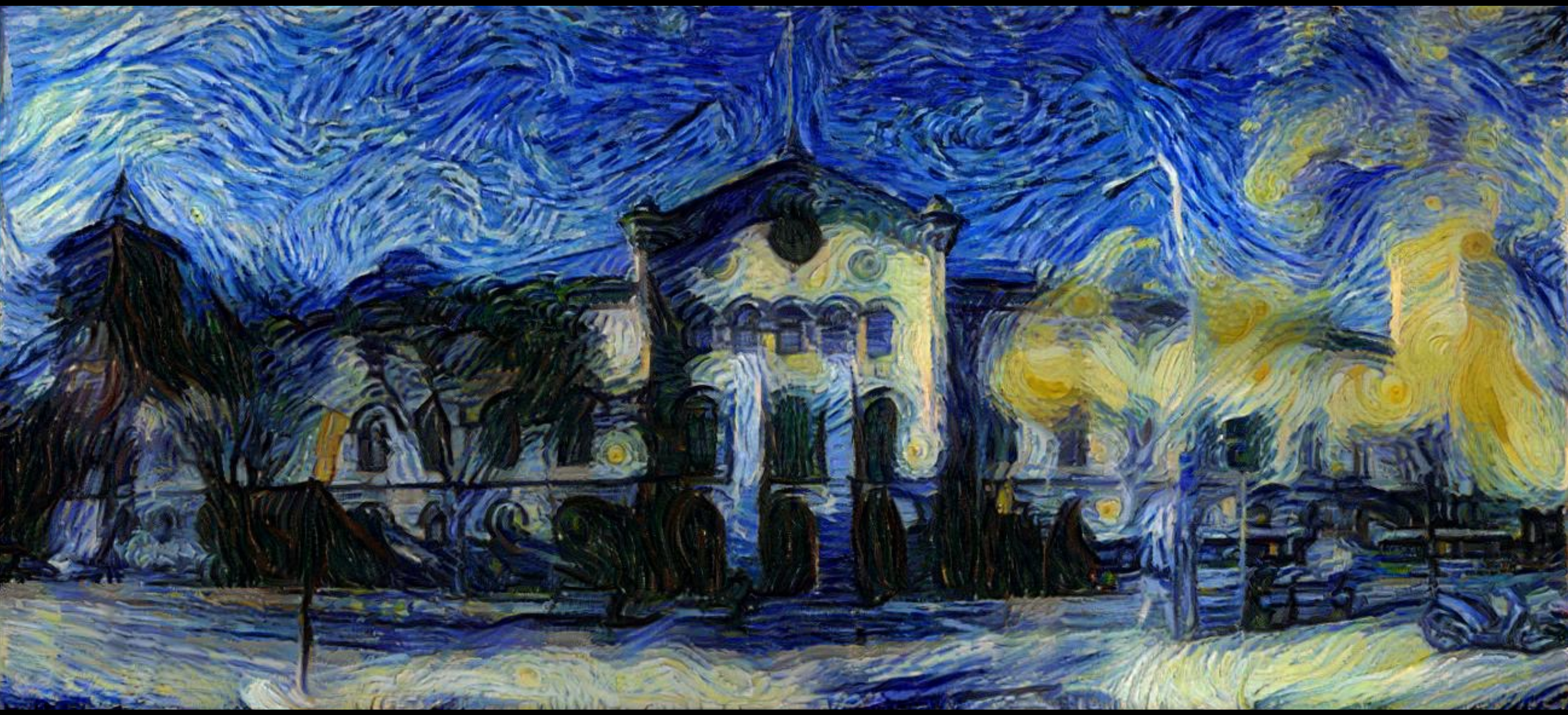
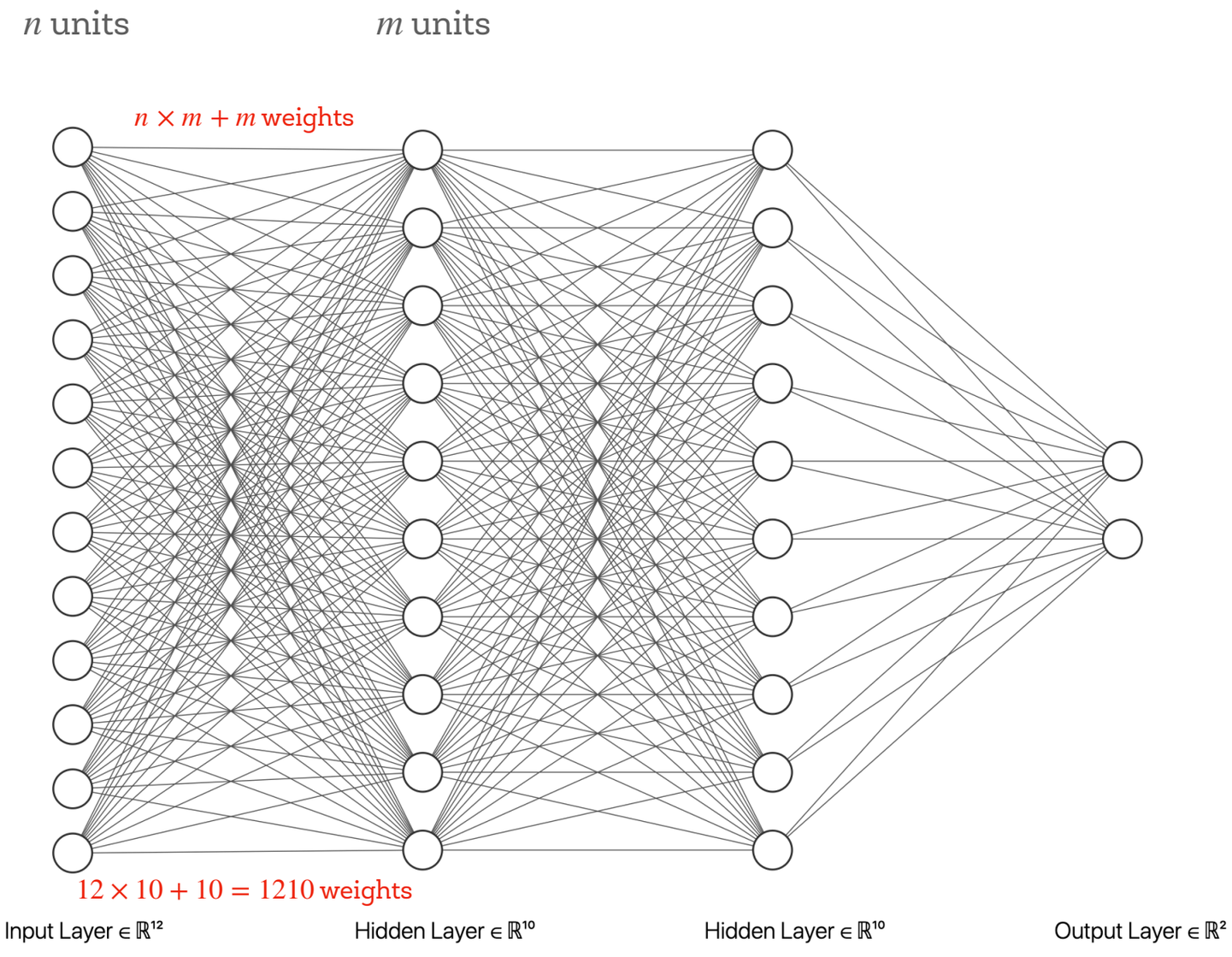


Deep Learning

Convolutional Neural Networks

Images and FC neural networks

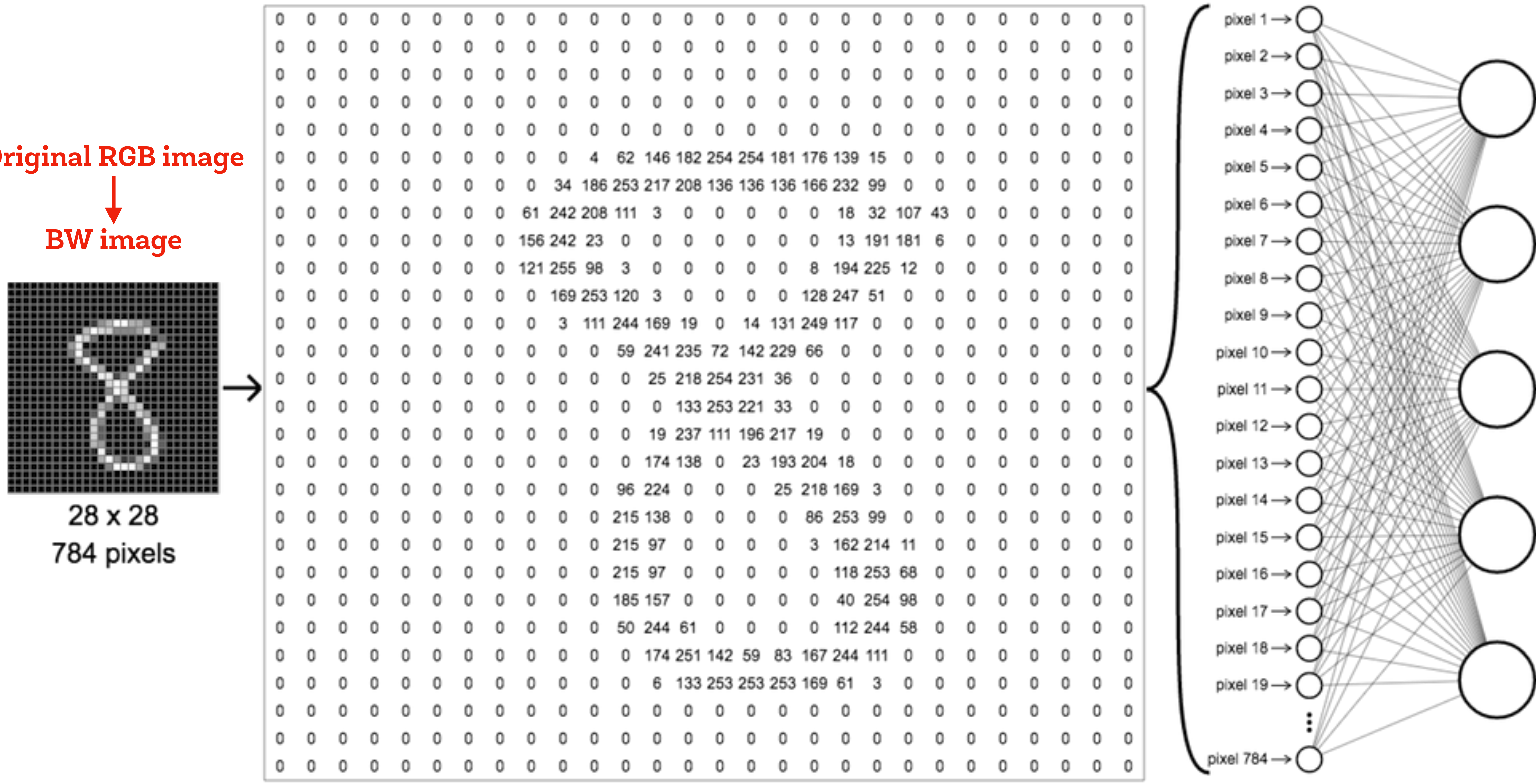


https://alexlenail.me/NN-SVG/index.html

Images and FC neural networks

784 × 5 + 5 weights

Original RGB image
↓
BW image



Flattening operation.
This function is used to get a copy of a given tensor collapsed into one dimension.

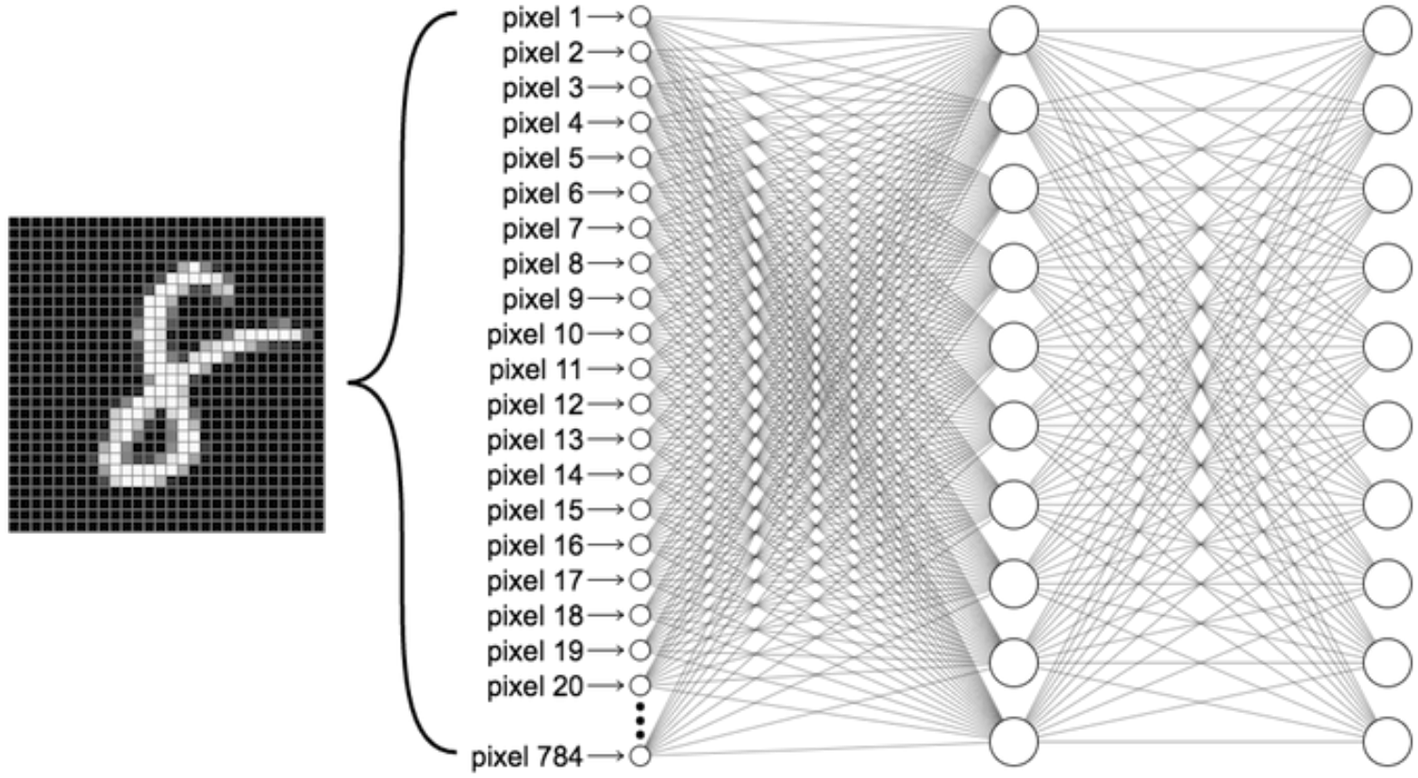
What about processing a 1024x1024 image?

https://mc.ai/what-is-deep-learning/

Part I: CNN layers

Images and FC neural networks

784 → 10 is a very aggresive dimensionaly reduction!

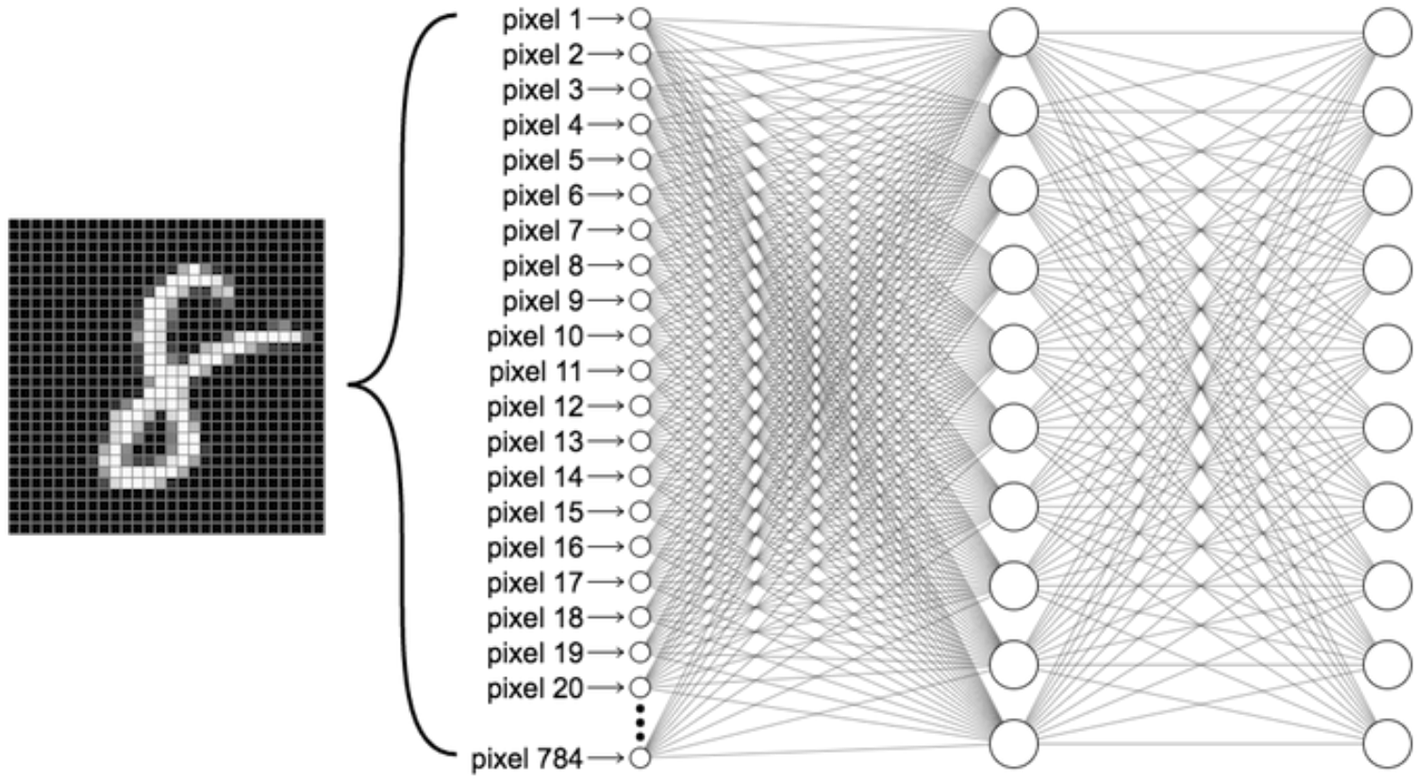


https://www.google.com/url?sa=i&url=https%3A%2F%2Fml4a.github.io%2Fml4a%2Flooking_inside_neural_nets%2F&psig=AOvVaw0UknUai0JapuIndBrY_BpX&ust=1604588386364000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCOiB-NmT6ewCFQAAAAAdAAAAABAD

Say we have a dataset of one-megapixel photographs we need to classify. This means that each input to the network has **one million dimensions**.

Even an **aggressive reduction** to one thousand hidden dimensions would require a fully-connected layer characterized by $10^6 \times 10^3 = 10^9$ **parameters**. Moreover, learning a classifier by fitting so many parameters might require collecting an **enormous dataset**.

Images and FC neural networks



https://www.google.com/url?sa=i&url=https%3A%2F%2Fml4a.github.io%2Fml4a%2Flooking_inside_neural_nets%2F&psig=AOvVaw0UknUai0JapuIndBrY_BpX&ust=1604588386364000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCOiB-NmT6ewCFQAAAAAdAAAAABAD

Convolutional neural networks (CNNs) are one creative way that machine learning has embraced for *exploiting some of the known structure in natural images* in order to (i) **minimize the number of parameters** and also (ii) to build interesting **invariances/inductive biases in the model**.

Inductive Bias

We can study **models** from two aspects:

- Whether a solution is **realisable** for the model, i.e., there is at least one set of weights that leads to the optimal solution.
- Whether a solution is **learnable** for the model, i.e., it is possible for the model to learn that solution within a reasonable amount of time and computational resources.

Inductive Bias

In many cases in deep learning, we are dealing with models that have enough (potential) expressive power to solve our problems.

However, they have **different biases regarding learnability**.

Optimal solutions are realisable for all of them, but depending on the task at hand it is easier for some of them to learn the solutions compared to the others.

Inductive Bias

Neural networks are good function approximation methods for **some kinds of data** (images, texts, etc.) because they can leverage interesting **invariances** that are embedded in their architectures.

This property is called **inductive bias**.

Inductive bias describes the tendency for a system to **prefer a certain set of generalizations over others** that are equally consistent with the observed data.

Inductive biases are the characteristics of learning algorithms that influence their generalization behaviour, **independent of data**.

Inductive Bias

Suppose an unbiased algorithm is trying to learn a mapping from numbers in the range one to 100 to the labels TRUE and FALSE.

> It observes that “2”, “4” and “6” are labeled TRUE, while “1”, “3” and “5” are labeled FALSE.

What is the label for “7”?

> To the **unbiased learner**, there are 2^{94} possible labelings (each one implemented by a different classifier) of the numbers seven through 100, all equally possible, with “7” labeled TRUE in one half and “7” labeled FALSE in the other.

Inductive Bias

X	Y
1	0
2	1
3	0
4	1
5	0
6	1
7	?
8	?
9	?
10	?
...	...
99	?

$y = f(x)$

The X space has 100 elements. We have fixed the outcome for 6 elements $(x_1, x_2, x_3, x_4, x_5, x_6)$, and we want to learn a function to predict the outcome for the rest.

In principle, all these classifiers ($n = 2^{94}$) are equally possible!
Each one of them represents **a function that is compatible with the training data**.

Which one is the right one?

1	0
2	1
3	0
4	1
5	0
6	1
7	1
8	1
9	1
10	1
...	...
99	1

$\hat{y} = f_1(x)$

1	0
2	1
3	0
4	1
5	0
6	1
7	0
8	1
9	1
10	1
...	...
99	1

$\hat{y} = f_2(x)$

1	0
2	1
3	0
4	1
5	0
6	1
7	0
8	0
9	1
10	1
...	...
99	1

$\hat{y} = f_3(x)$

. . .

1	0
2	1
3	0
4	1
5	0
6	1
7	0
8	0
9	0
10	0
...	...
99	0

$\hat{y} = f_n(x)$

Inductive Bias

To a human, of course, there is a clear pattern of even numbers labeled TRUE and odds labeled FALSE, so a label of FALSE for "7" seems likely.

This type of **belief**, that certain hypotheses like

"*evens are labeled TRUE*"

seem likely, and other hypotheses like

"*evens are labeled TRUE up through "6", and the rest of the labeling is random*"

seem unlikely, is **inductive bias**.

Inductive Bias

This likely solution can be seen as a "**simple solution**" from different points of view.

For example:

- It can be written in a concise way (if $x \,\%\, 2 == 0$: label $= 1$; else label $= 0$).
- Most of the 2^{94} solutions cannot be written in a way simpler than enumerating the outcome for each input value.

"**Simplicity**" **(parsimony) is one strong inductive bias**!

Inductive Bias

Fully-connected neural networks have an inherent inductive bias that makes them inclined to learn **generalizable hypotheses** and **avoid memorization** (non parsimonious solutions).

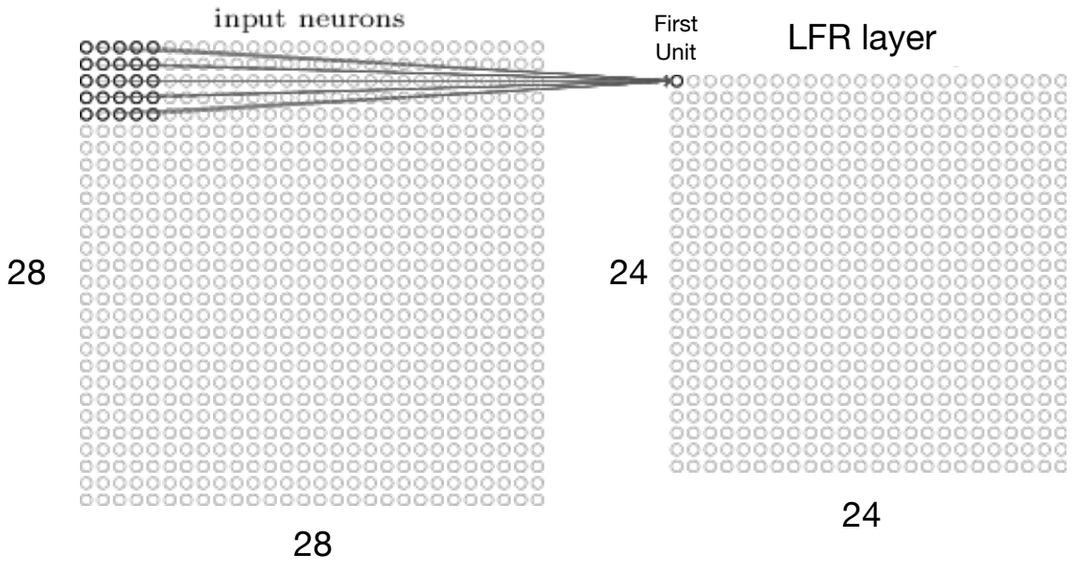
In this respect, some results suggest (this is an hypothesis) that the inductive bias stems from neural networks being lazy: they tend to learn **simple rules** first.

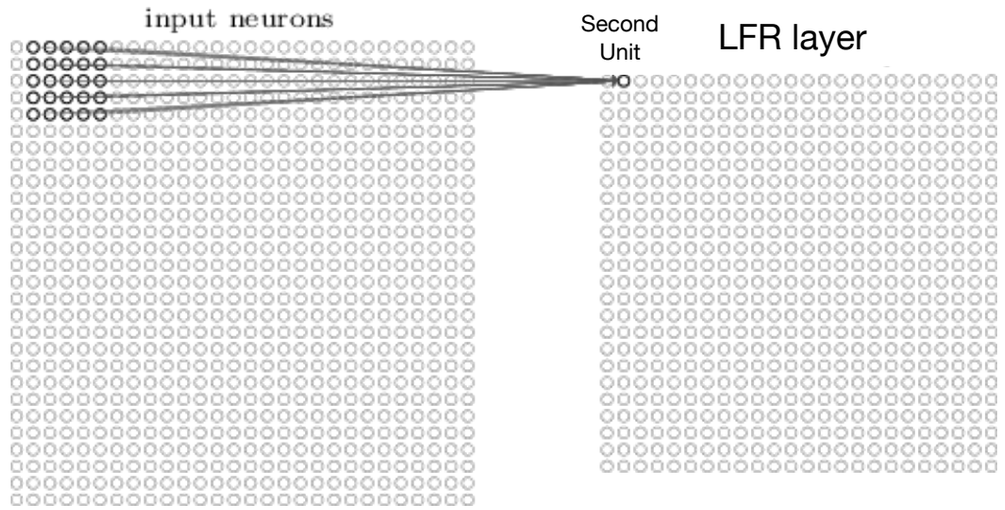
Moreover, deep learning architectures (beyond fully connected layers) add new inductive biases to these models.

Minimizing the number of parameters: LRF's

One possible to minimize the number of parameters is to consider **local receptive fields** (LRF) (an inspiration that comes from neuroscience).

A local receptive field is a standard MLP with units that have a **local view** of the image: its input is a submatrix $M \times M$ of the original $N \times N$, where $M << N$.

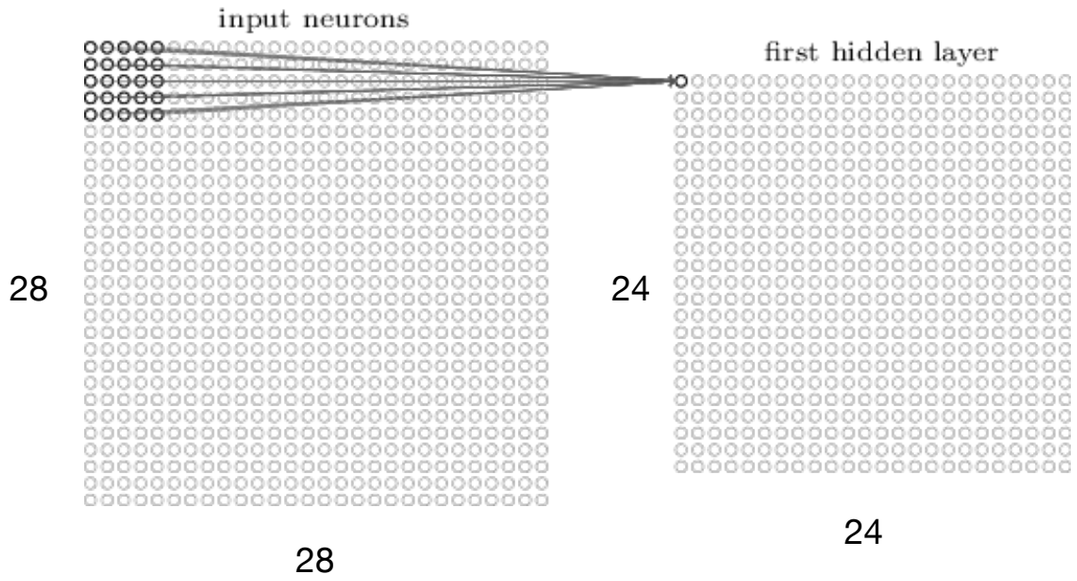




No flattening operation!

http://neuralnetworksanddeeplearning.com/chap6.html

Minimizing the number of parameters: LRF's



This LFR layer has one neuron for each pixel (with the exception of the borders of the image)

Note that if we have a 28×28 input image, and 5×5 local receptive fields, then there will be 24×24 neurons in the hidden layer.

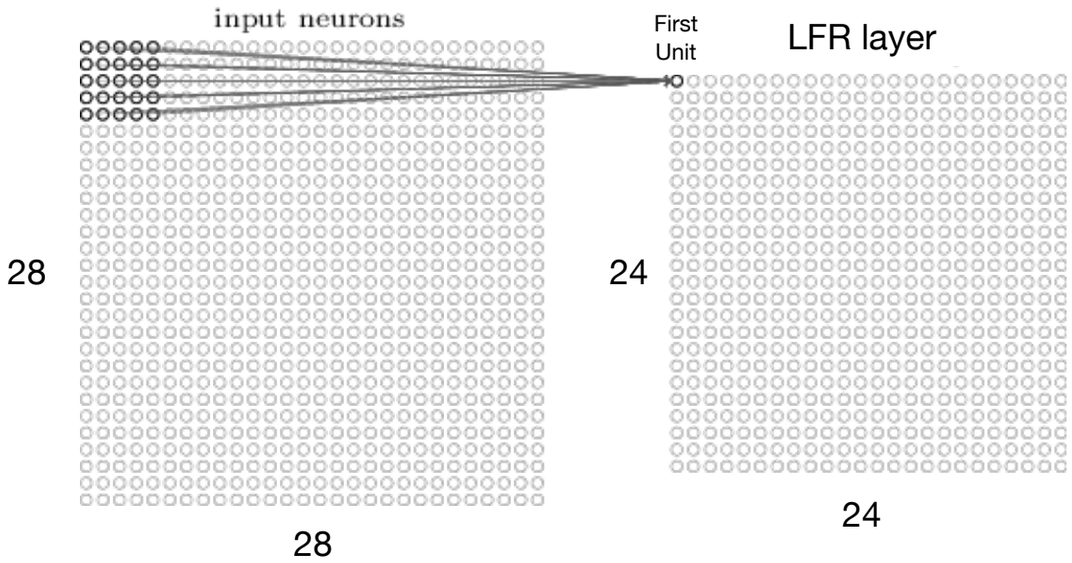
This is because we can only move the local receptive field 24 neurons across (or 24 neurons down), before colliding with the right-hand side (or bottom) of the input image.

http://neuralnetworksanddeeplearning.com/chap6.html

Minimizing the number of parameters: LRF's

One possible to minimize the number of parameters is to consider **local receptive fields** (LRF) (an inspiration that comes from neuroscience).

A local receptive field is a standard MLP with units that have a **local view** of the image: its input is a submatrix $M \times M$ of the original $N \times N$, where $M << N$.



MNIST image size= 28 x 28 pixels = 784 dimensions

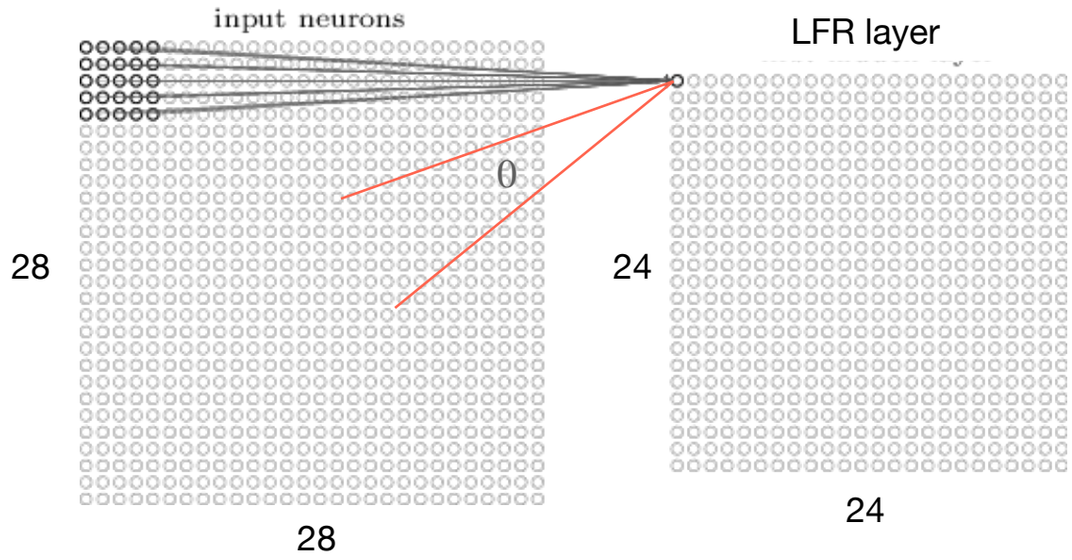
A FC layer with 100 units has (784 x 100 + 100) parameters = 78500 parameters

This LRF layer with 24x24 units has 25 x 24 x 24 + (24 x 24) parameters = 14976 parameters

http://neuralnetworksanddeeplearning.com/chap6.html

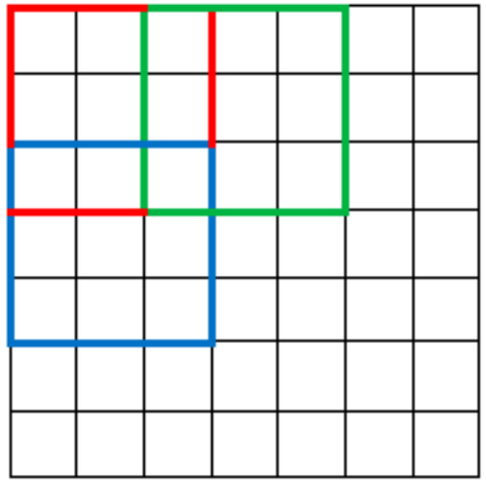
Minimizing the number of parameters: LRF's

A LRF network can be seen as a fully connected layer where a large subset of weights are 0.

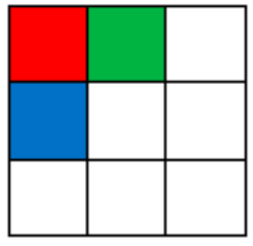


Minimizing the number of parameters: LRF's

Stride = 2



Input 7x7



Output 3x3

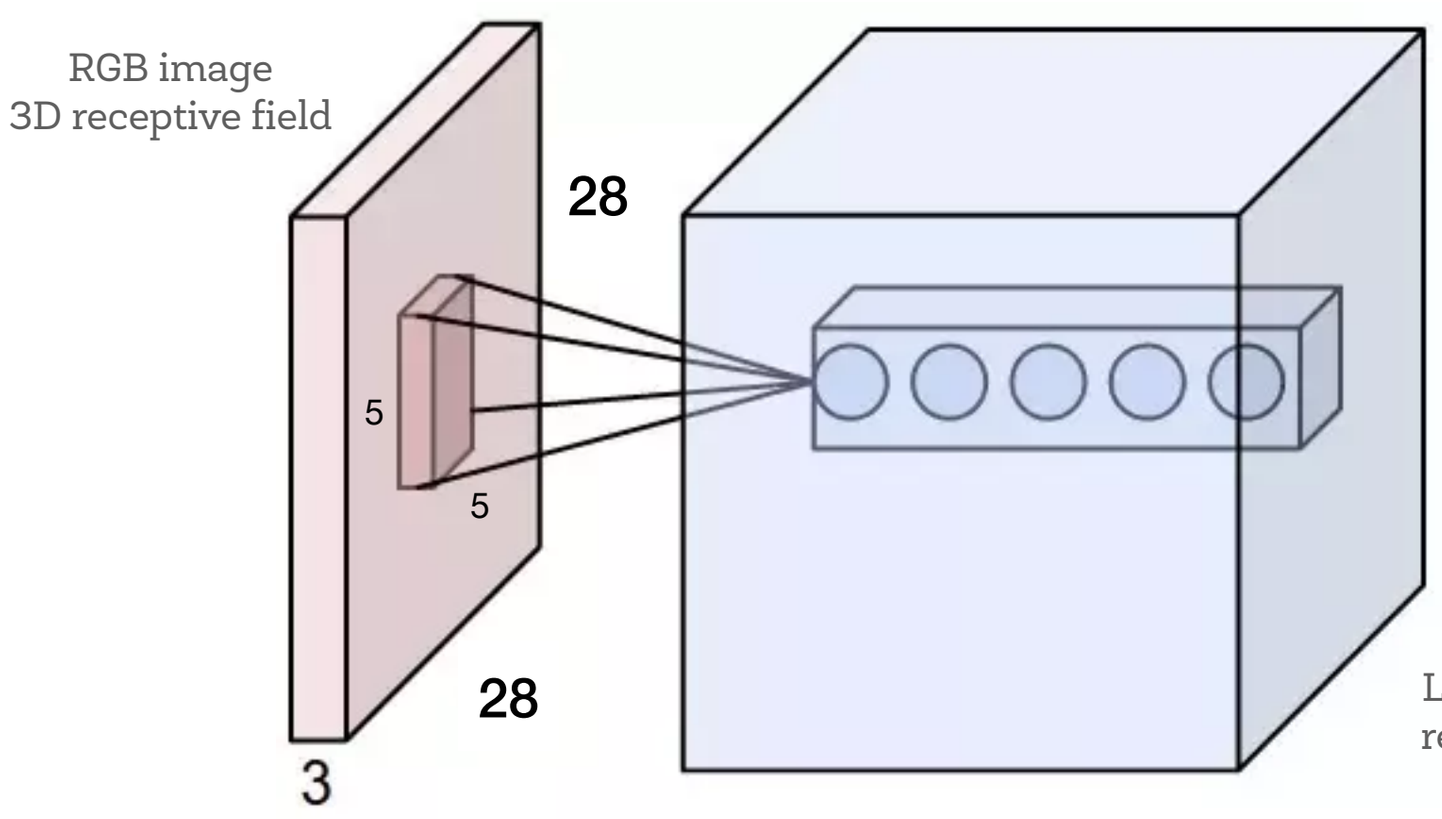
We have shown the local receptive field being moved by one pixel at a time, but this can be **redundant** in the case of natural images.

In fact, sometimes a different **stride** length is used. For instance, we might move the local receptive field 2 pixels to the right (or down), in which case we'd say a stride length of 2 is used. This also **reduces ($\times 4$) the number of units of the first layer** and consequently the number of parameters.

http://neuralnetworksanddeeplearning.com/chap6.html

Minimizing the number of parameters: LRF's

To have more capacity, we could compute **several local receptive fields** per pixel and also to consider **3D local receptive fields** for color images:



Layer with multiple local receptive field responses (24 x 24 x 5)

number of receptive fields per pixel

This LRF layer has $\underline{25 \times 3} \times \underline{5 \times 24 \times 24} + \underline{(24 \times 24 \times 5)} = 218880$ parameters

parameters per receptive field

number of units in the first layer

biases

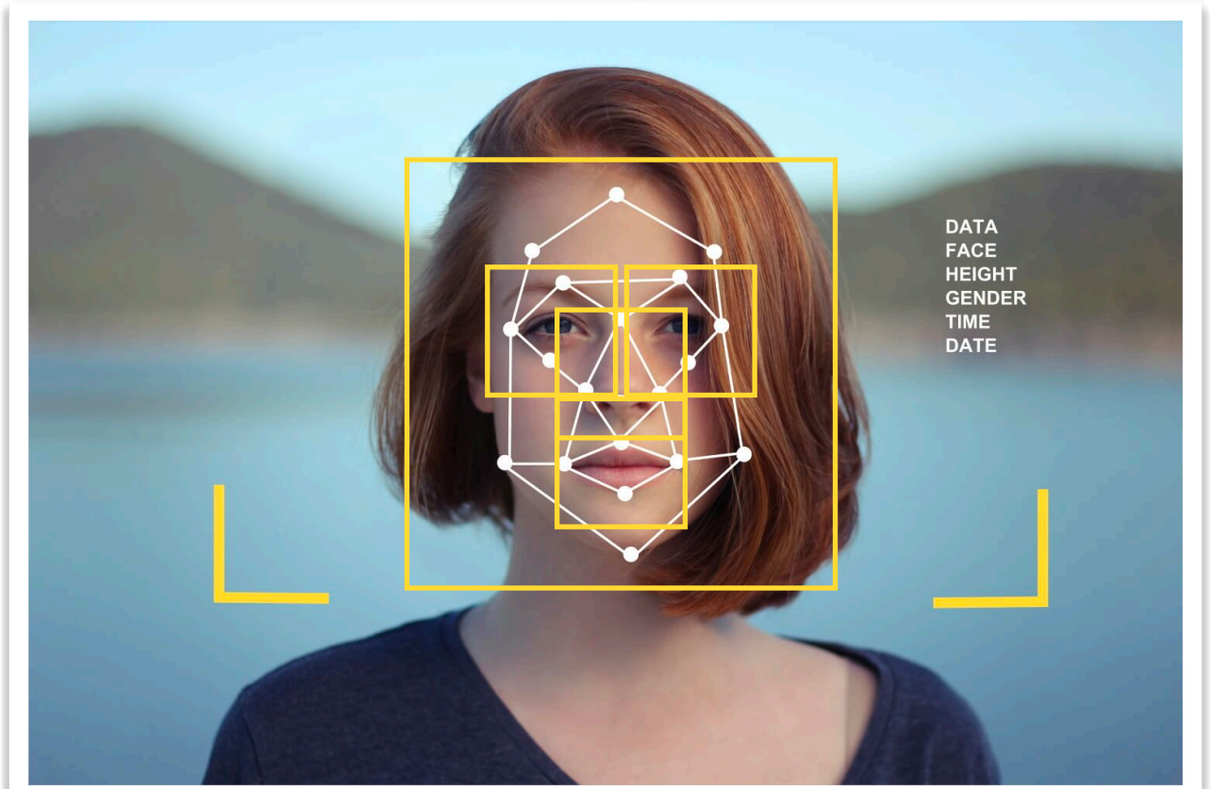
Too many!

From LRF's to Convolutions

Using LRF is like decomposing the problem of image analysis in a set of local experts that are specialized in certain parts of the image.

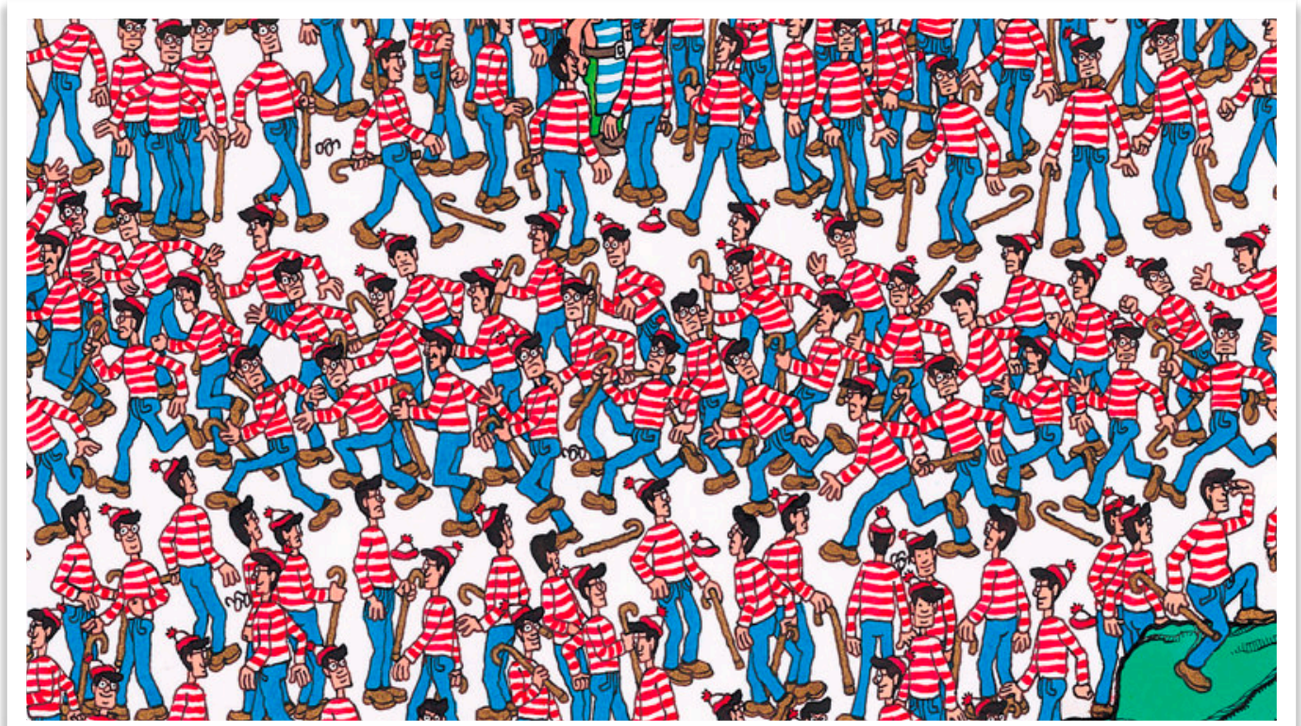
The outcome of these specialists can be integrated at a high level of the model by **stacking** layers (receptive fields over receptive fields).

This can be useful for some applications (where we dot not need translation-invariance): f.e. face recognition from detected faces.



From LRF's to Convolutions

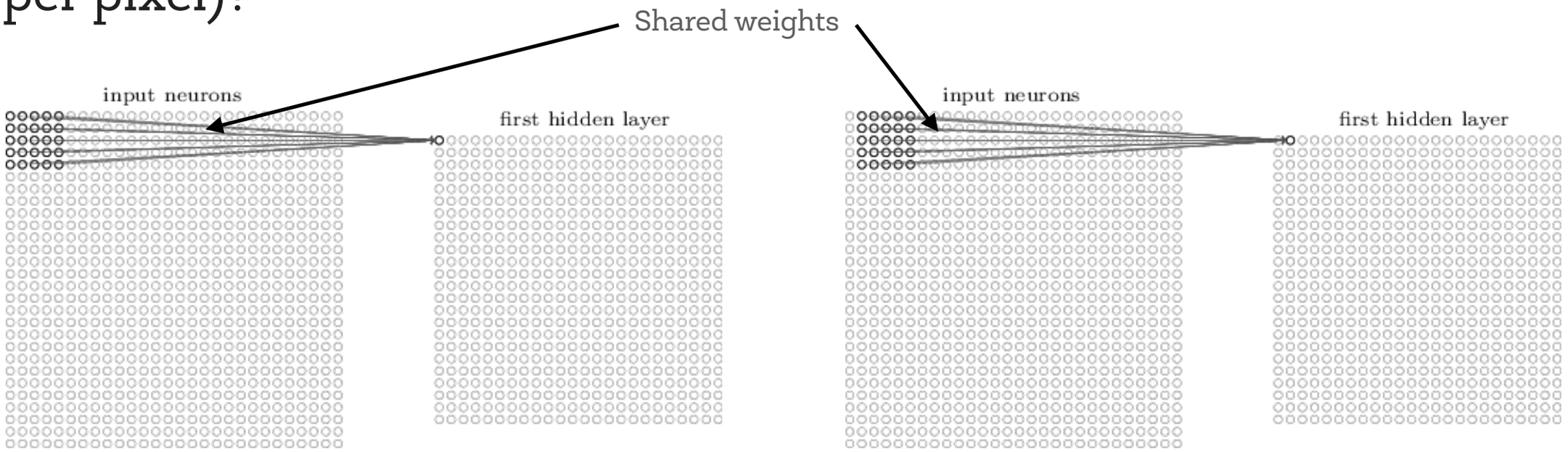
But in **natural images**, an object may appear at any image coordinates! Object detection must be invariant to translations in the image plane.



In this case **LRF are not suited for this task** because in order to learn a detector we should provide the model with a lot of examples in order to consider all possible object localizations!

Froom LRF's to convolutional layers

What about **sharing** LRF weights (and using several receptive fields per pixel)?



This can be expressed as a **convolution**. Now, for the (j, k) hidden unit, the output is of "shared" a receptive field can be expressed as:

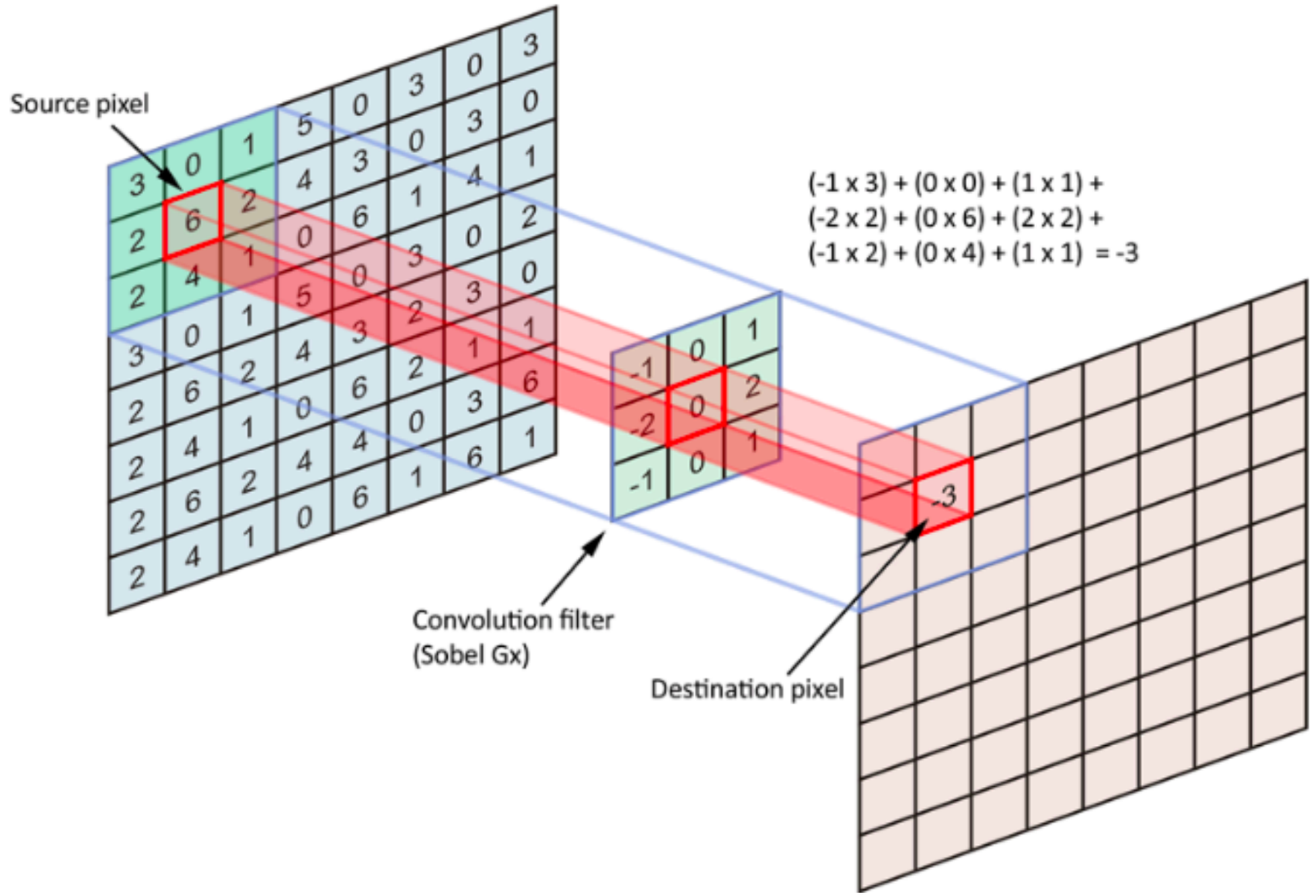
The non-linear activation function is necessary!

$$\sigma\left(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{l,m} a_{j+l,k+m}\right)$$

where b is the shared value for the bias, $w_{l,m}$ is a 5x5 array of shared weights and $a_{x,y}$ represents the input value at position x, y .

Convolutional layers

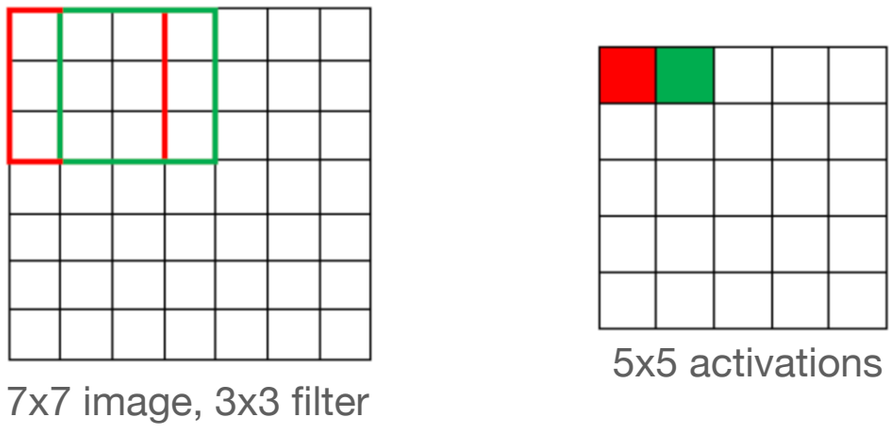
The term **convolution** refers to the mathematical combination of two functions to produce a third function. It merges two sets of information.



https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

Padding

For a gray scale (n x n) image and (f x f) filter/kernel, the dimensions of the image resulting from a convolution operation is (n – f + 1) x (n – f + 1).



Thus, the image shrinks every time a convolution operation is performed. This places an upper **limit to the number of times such an operation could be performed (due to stacking) before the image reduces to nothing** thereby precluding us from building deeper networks.

To overcome these problems, we use **padding**.

Padding

Padding is simply a process of adding “dimensions” to our input images so as to avoid the problems mentioned above.

Input: 6x6 image

0	0	0	0	0	0	0	0
0	1	2	3	1	3	5	0
0	2	2	5	4	2	5	0
0	0	6	9	6	2	2	0
0	2	0	1	9	4	0	0
0	5	5	4	6	7	6	0
0	6	1	3	7	1	5	0
0	0	0	0	0	0	0	0

0-padding

3x3 Filter

1	0	-1
1	0	-1
1	0	-1

*

=

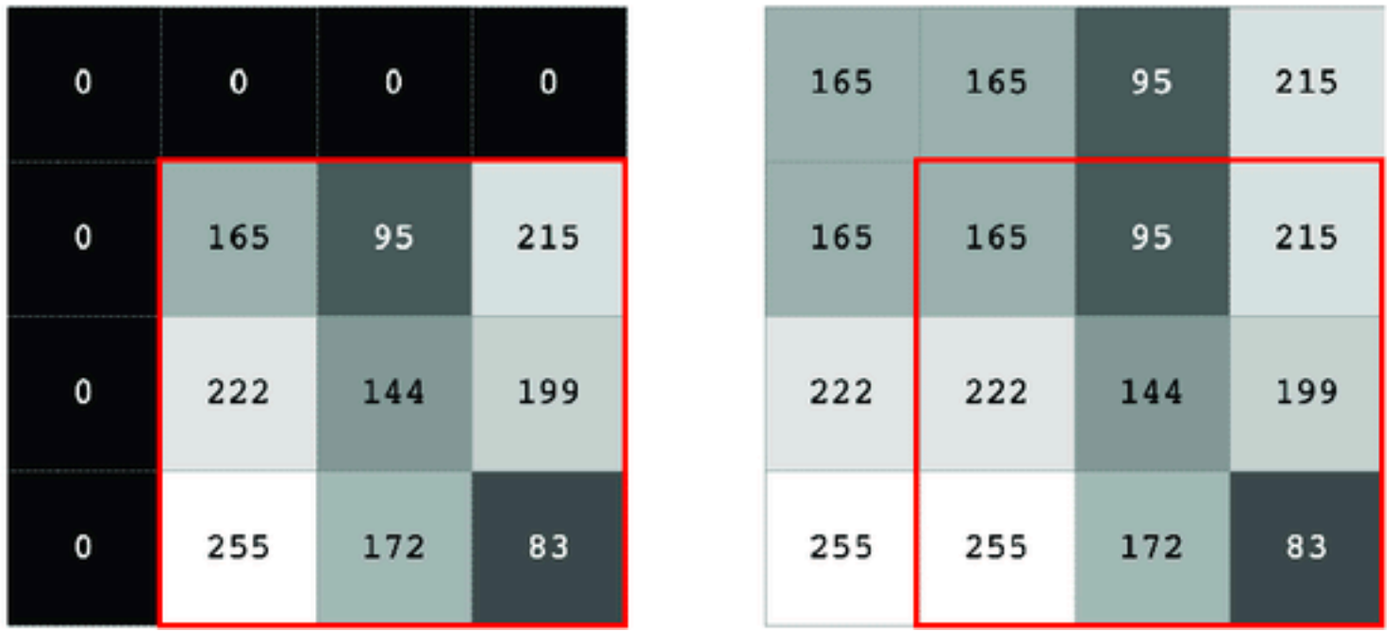
Output: 6x6 activations

-4	-5	-1	3	-5	5
-10	-14	-1	10	-1	7
-8	-11	-11	7	12	8
11	-7	-10	1	13	13
-6	5	-16	-4	10	12
-6	4	-7	-1	2	8

The non-linear activation function is not shown for simplicity!

https://www.brilliantcode.net/1584/convolutional-neural-networks-1-convolution-layer-stride-padding-kernel/

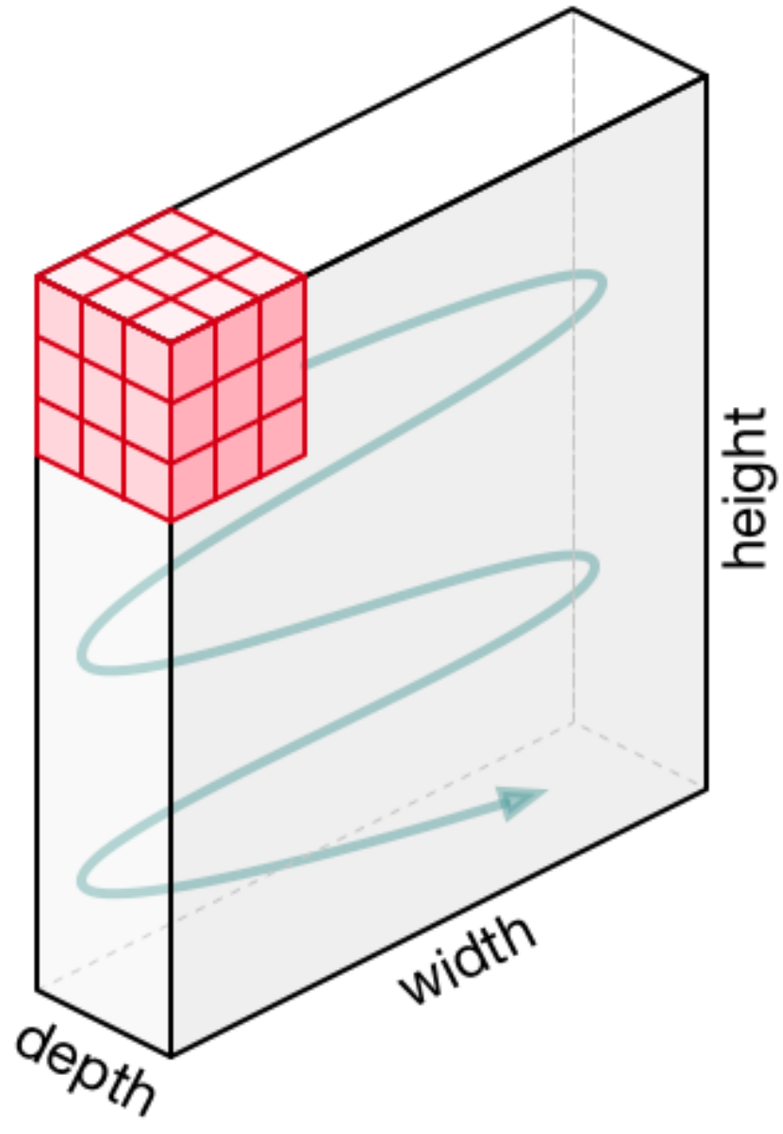
Padding



0-padding

Replication-padding

Convolutional layers (tensors)



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Convolutional layers

0	0	0	0	0	0	...
0	156	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...	...	...	...	...	...	...

Input Channel #1 (Red)

0	0	0	0	0	0	...
0	167	166	167	169	169	...
0	164	165	168	170	170	...
0	160	162	166	169	170	...
0	156	156	159	163	168	...
0	155	153	153	158	168	...
...	...	...	...	...	...	...

Input Channel #2 (Green)

0	0	0	0	0	0	...
0	163	162	163	165	165	...
0	160	161	164	166	166	...
0	156	158	162	165	166	...
0	155	155	158	162	167	...
0	154	152	152	157	167	...
...	...	...	...	...	...	...

Input Channel #3 (Blue)

Padding

-1	-1	1
0	1	-1
0	1	1

Kernel Channel #1

1	0	0
1	-1	-1
1	0	-1

Kernel Channel #2

0	1	1
0	1	0
1	-1	1

Kernel Channel #3

Model parameters

$308 + (-498) + 164 + 1 = -25$

Bias = 1

The non-linear activation function is not shown for simplicity!

Output

-25				...
				...
				...
				...
...	...	...	...	...

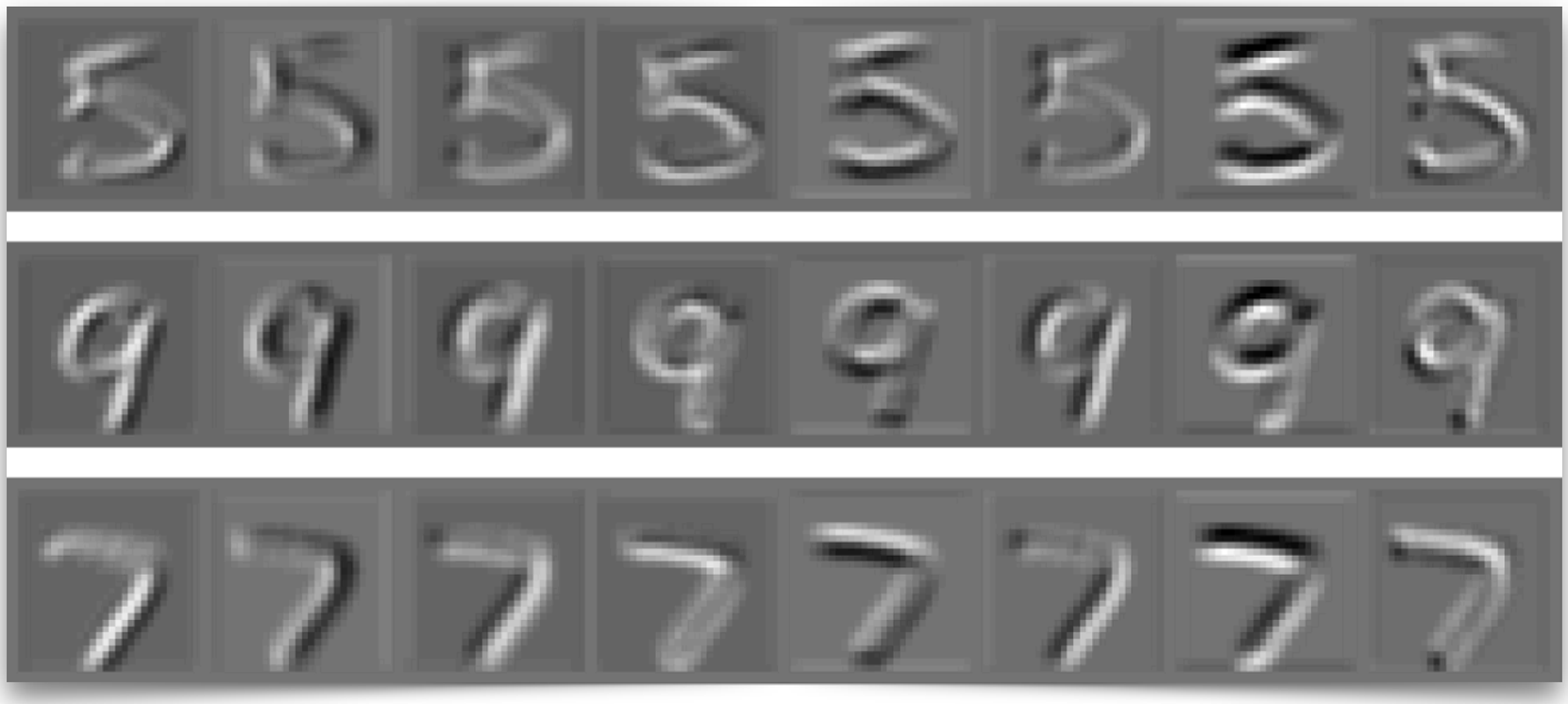
Input of the next layer

https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Convolutional layers

The weights of the first layer of a CNN can be visualized!
They are $n \times n \times 3$ (color images) or $n \times n \times 1$ (BW images) tensors.

Weights of conv1 layer, trained with BW MNIST, which has shape [8, 1, 5, 5] (8 outer channels, 1 inner channel, 5x5 grids)

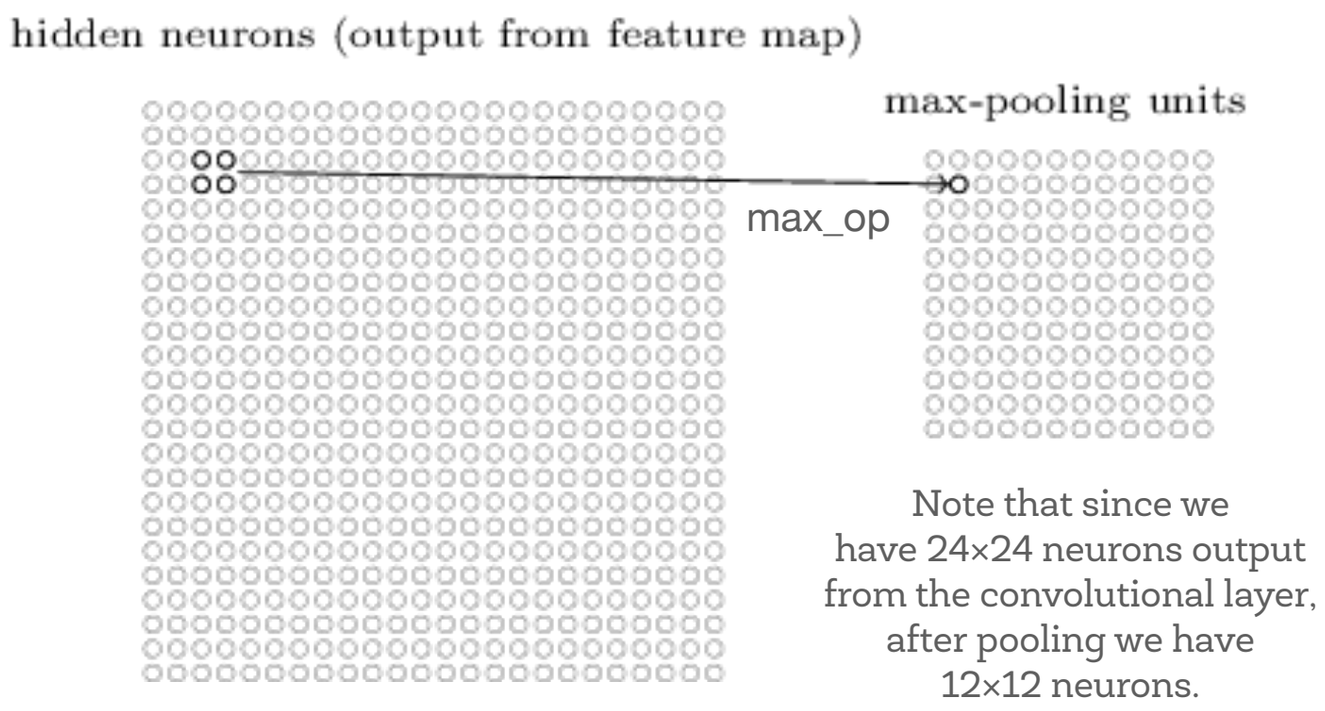


Filter output values on random inputs from MNIST.

Convolutional layers

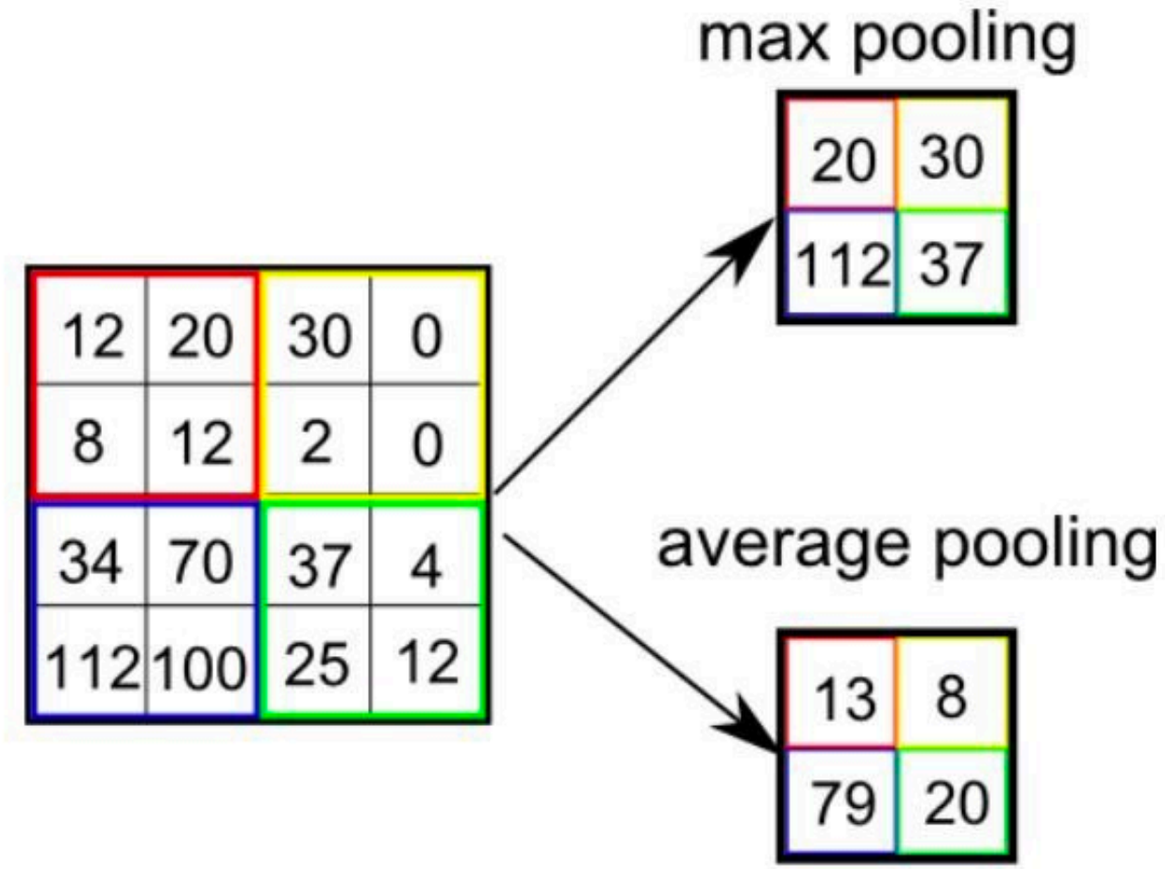
In addition to the convolutional layers just described, convolutional neural networks also contain **pooling layers**. Pooling layers are usually used immediately after convolutional layers. What the pooling layers do is **simplify the information in the output** from the convolutional layer.

As a concrete example, one common procedure for pooling is known as ***max-pooling***. In max-pooling, a pooling unit simply outputs the maximum activation in the 2×2 input region, as illustrated in the following diagram:



https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

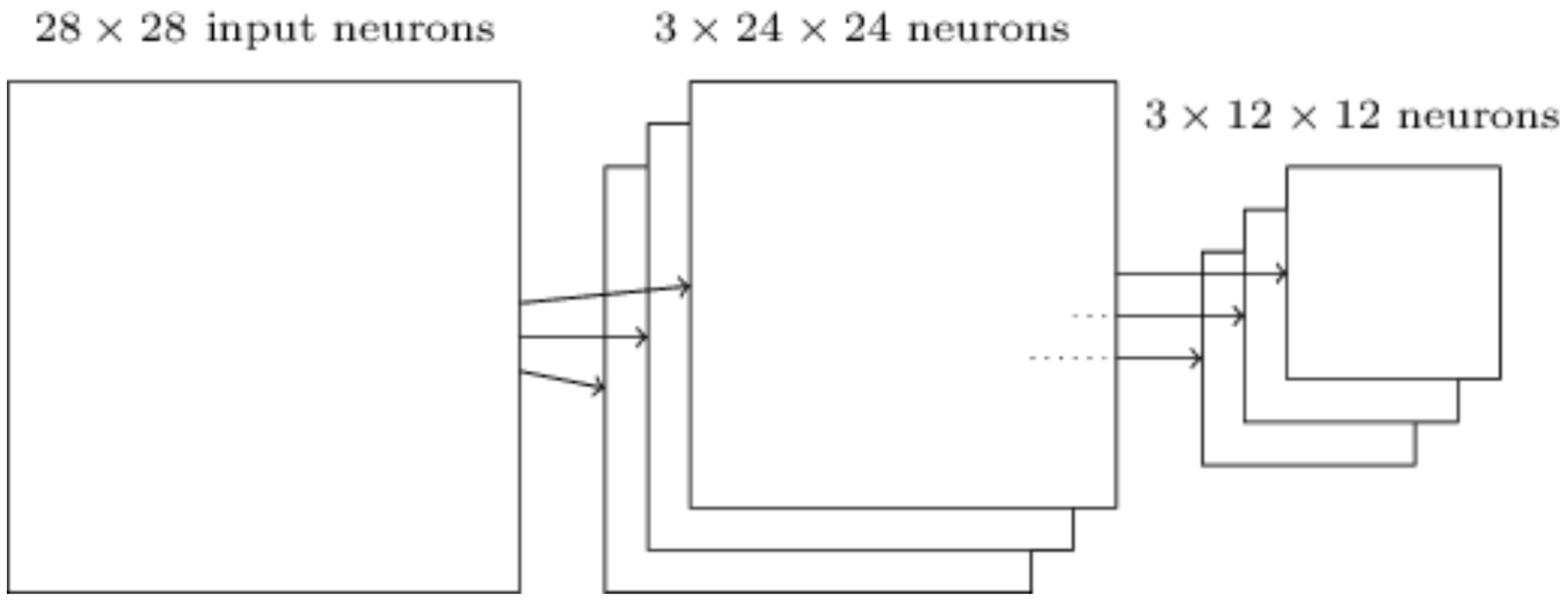
Convolutional layers



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Convolutional layers

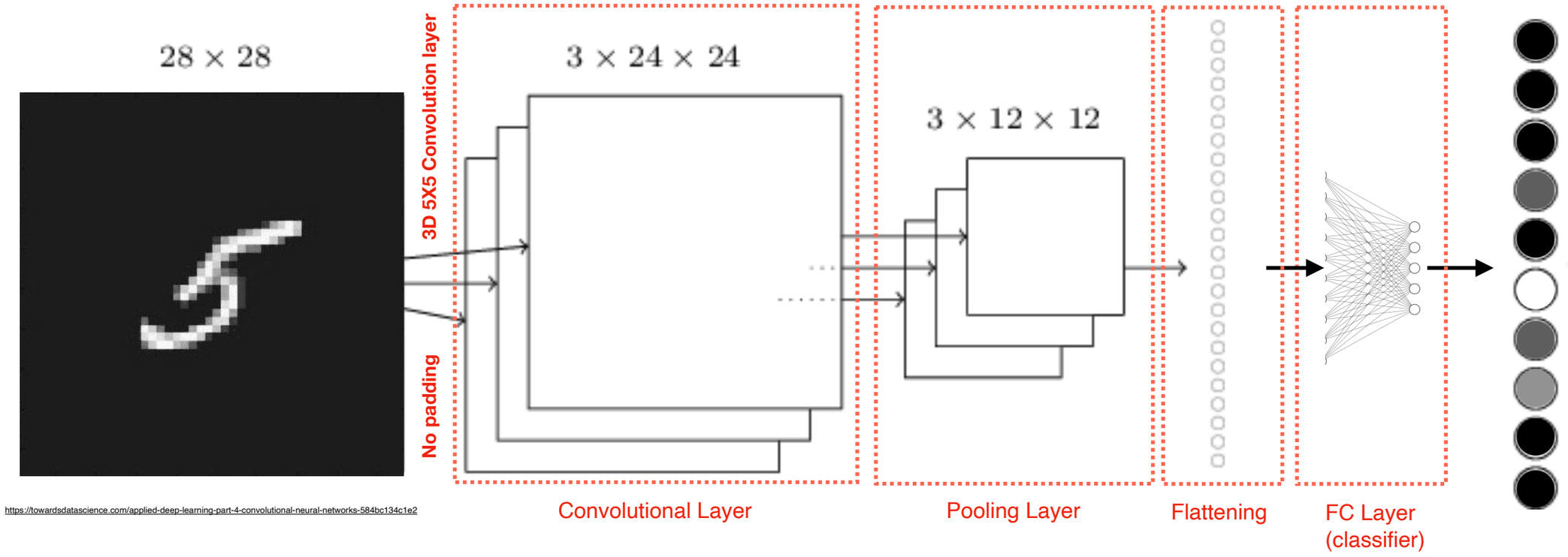
If there were 3 feature maps/convolution filters with no padding, the combined convolutional and max-pooling layers would look like:



https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

Convolutional layers

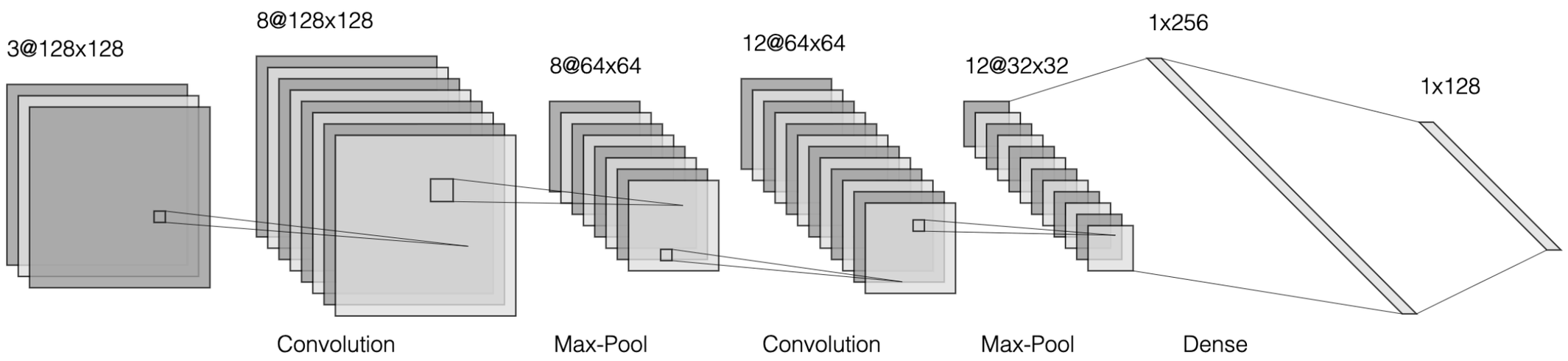
We can now put all these ideas together to form a complete **1-layer convolutional neural network**:



The network begins with 28×28 input neurons, which are used to encode the pixel intensities for the MNIST image. This is then followed by a convolutional layer using a 5×5 local receptive field and 3 feature maps. The result is a layer of 3×24×24 hidden feature neurons. The next step is a max-pooling layer, applied to 2×2 regions, across each of the 3 feature maps. The result is a layer of 3×12×12 hidden feature neurons. Finally, there is a fully connected layer with 3×12×12x10 + 10 parameters.

Convolutional layers and stacking

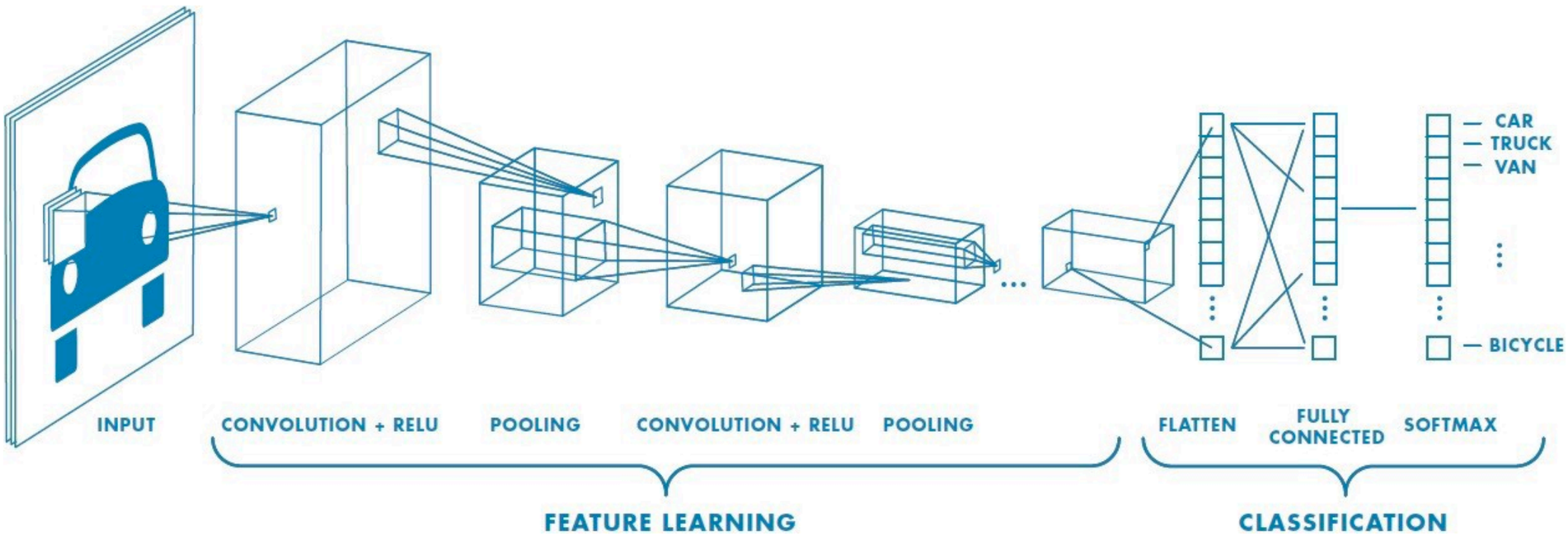
The **stacking** of convolutional layers allows a **hierarchical decomposition** of the input.



Convolutional layers and stacking

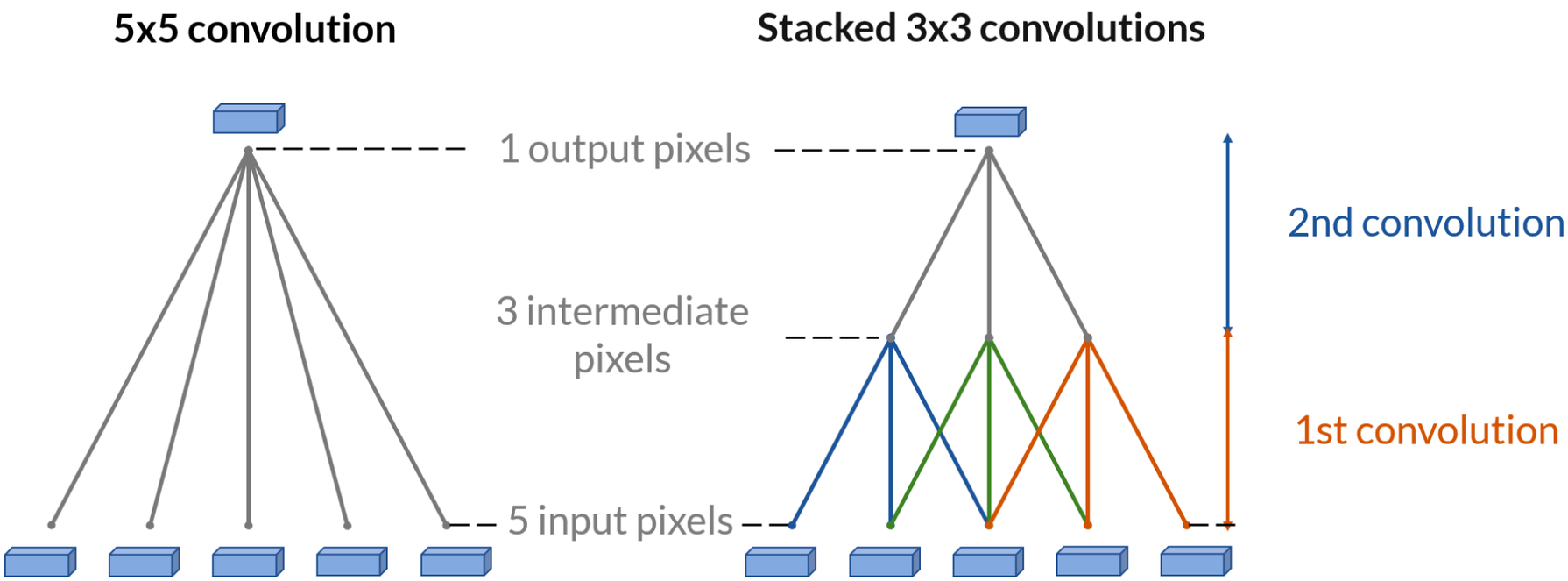
Consider that the filters that operate directly on the raw pixel values will learn to extract low-level features, such as lines and small patterns. The filters that operate on the output of the first line layers may extract features that are combinations of lower-level features, such as features that comprise multiple lines to express shapes.

This process continues until very deep layers are extracting faces, animals, houses, and so on.



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Convolutional layers and stacking



We can replace large convolution kernels with multiple 3x3 convolutions on top of one another.
This is good for two reasons: deeper is better and less computational cost.

Convolutional layers in Keras

```
1
2 model = keras.Sequential(
3     [
4         keras.Input(shape=input_shape),
5         layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
6         layers.MaxPooling2D(pool_size=(2, 2)),
7         layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
8         layers.MaxPooling2D(pool_size=(2, 2)),
9         layers.Flatten(),
10        layers.Dropout(0.5),
11        layers.Dense(num_classes, activation="softmax"),
12    ]
13 )
14
15
16
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1600)              0
_________________________________________________________________
dropout (Dropout)            (None, 1600)              0
_________________________________________________________________
dense (Dense)                (None, 10)                16010
=================================================================
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
```

```
Test loss: 0.027600124478340 15
Test accuracy: 0.9900000095367432
```

What about the inductive bias of a CNN?

```
# create model
model = kmod.Sequential()

# add model layers
model.add(klay.Conv2D(
    filters=32, kernel_size=3, activation='relu',
    input_shape=x_train.shape[1:],
    trainable=False))
model.add(klay.MaxPool2D(trainable=False))
model.add(klay.Conv2D(
    filters=32, kernel_size=3, activation='relu',
    kernel_initializer="glorot_uniform",
    trainable=False))
model.add(klay.MaxPool2D(trainable=False))
model.add(klay.Conv2D(
    filters=32, kernel_size=3, activation='relu',
    kernel_initializer="glorot_uniform",
    trainable=False))
model.add(klay.Flatten(trainable=False))
model.add(klay.Dense(
    units=10, activation='softmax',
    kernel_initializer="glorot_uniform"))
```

```
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_30 (Conv2D)           (None, 26, 26, 32)        320

max_pooling2d_20 (MaxPoolin  (None, 13, 13, 32)        0
g2D)

conv2d_31 (Conv2D)           (None, 11, 11, 32)        9248

max_pooling2d_21 (MaxPoolin  (None, 5, 5, 32)          0
g2D)

conv2d_32 (Conv2D)           (None, 3, 3, 32)          9248

flatten_10 (Flatten)         (None, 288)               0

dense_10 (Dense)             (None, 10)                2890

=================================================================
Total params: 21,706
Trainable params: 2,890
Non-trainable params: 18,816
```

```
Random Feature Accuracy = (91.49 +/- 0.58)%
```

What about the inductive bias of a CNN?

CNN architecture leverages some natural invariances (e.g. translations, small scale changes, small deformations) for free.

From Images to Computer Vision

We have seen how to implement image classification, but there are other Computer Vision tasks:

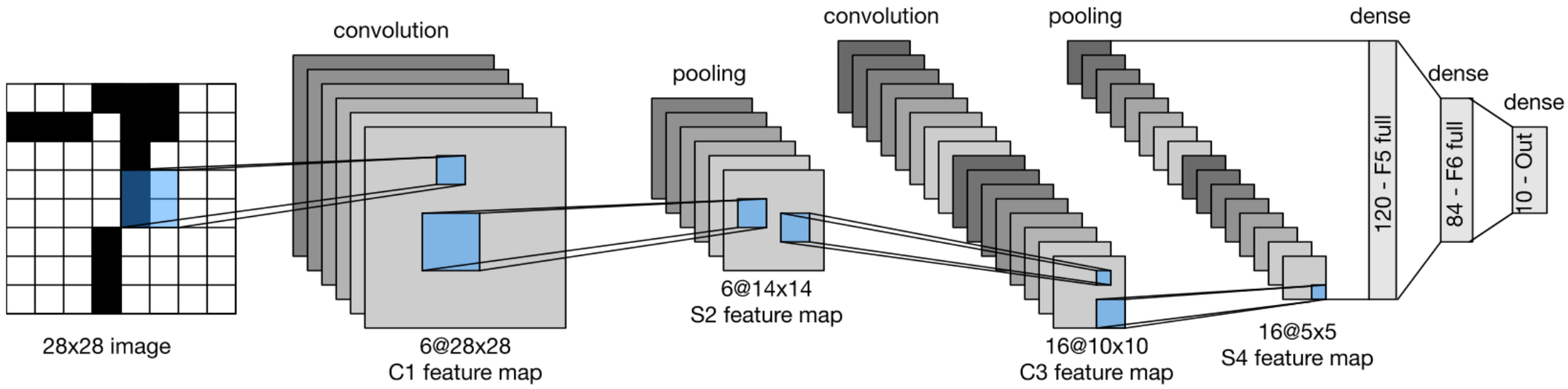
- Reconstruction (recover what is "not seen", f.e. 3D)
- Visual Control, tracking,...
- Segmentation
- Detection (bounding box)
- Etc.

CNNs can play an important role to solve this tasks, but we must reformulate them in terms of learning tasks (losses and architectures).

Part II: CNN architectures

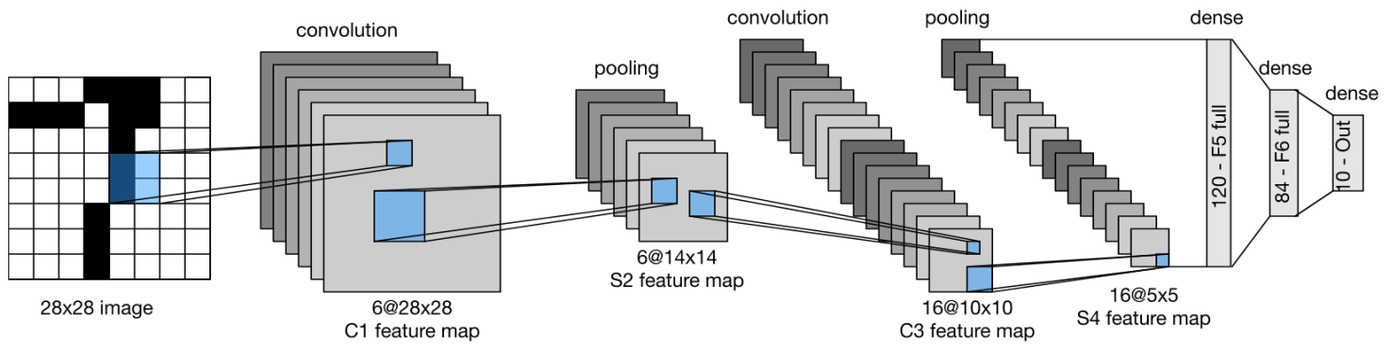
LeNET (1989)

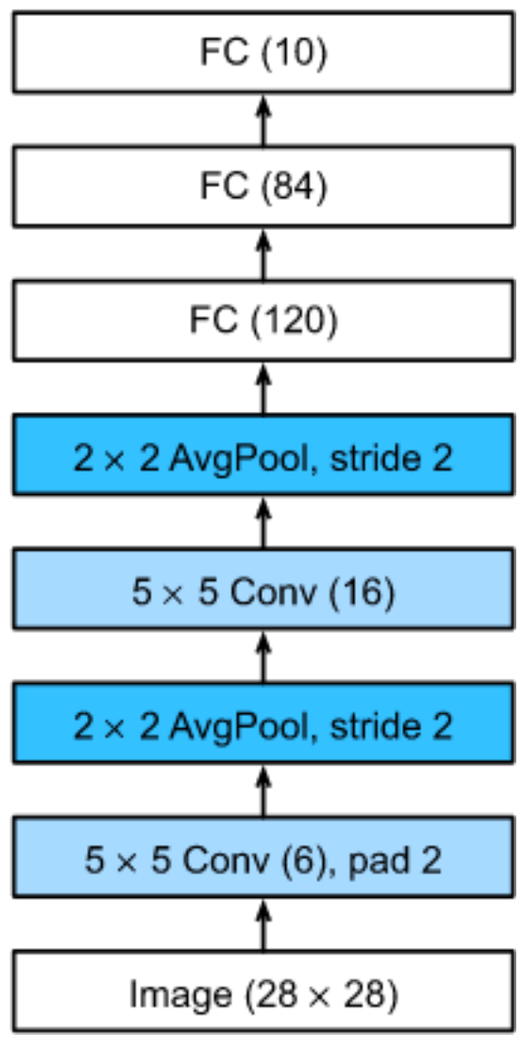
LeNet was the first published CNNs to capture wide attention for its performance on computer vision tasks. The model was introduced by (and named for) Yann LeCun, then a researcher at AT&T Bell Labs, for the purpose of recognizing handwritten digits in images.



https://d2l.ai/chapter_convolutional-neural-networks/lenet.html

LeNET (1989)



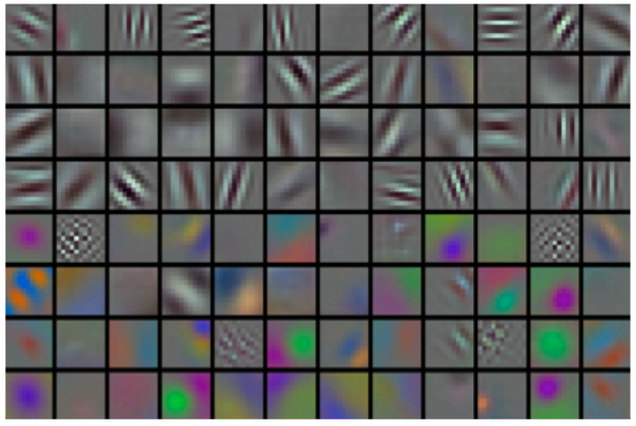


https://d2l.ai/chapter_convolutional-neural-networks/lenet.html

AlexNet (2012)

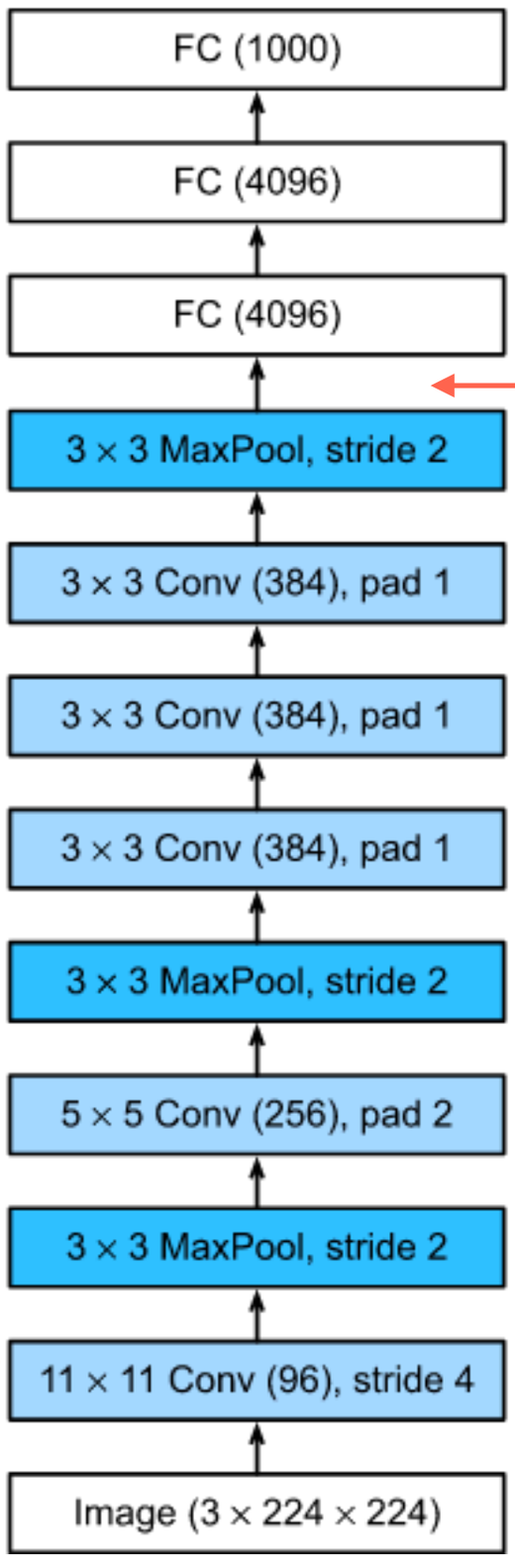
Rebirth of Neural Networks

1 Filters that operate in raw pixels can be visualized because they are like images (**11 x 11 x 3**).



1000 classes

0 Clocking in at **150 GB**, ImageNet holds 1,281,167 images for training and 50,000 images for validation, organized in 1,000 categories.



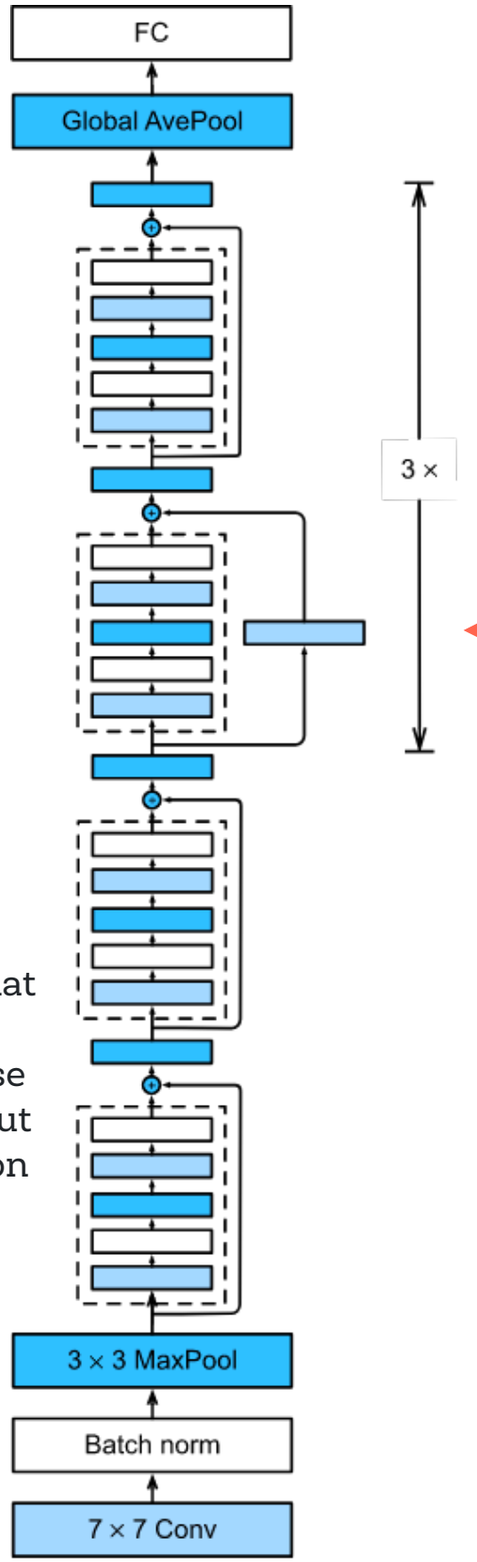
3 Ultimately, the final hidden state learns a **compact representation of the image that summarizes its contents** such that data belonging to different categories be separated easily.

2 Higher layers in the network might **build upon these representations to represent larger structures**, like eyes, noses, blades of grass, and so on. Even higher layers might represent whole objects like people, airplanes, dogs, or frisbees.

1 The lowest layers of the network, the model learned feature extractors that resembled some traditional filters.

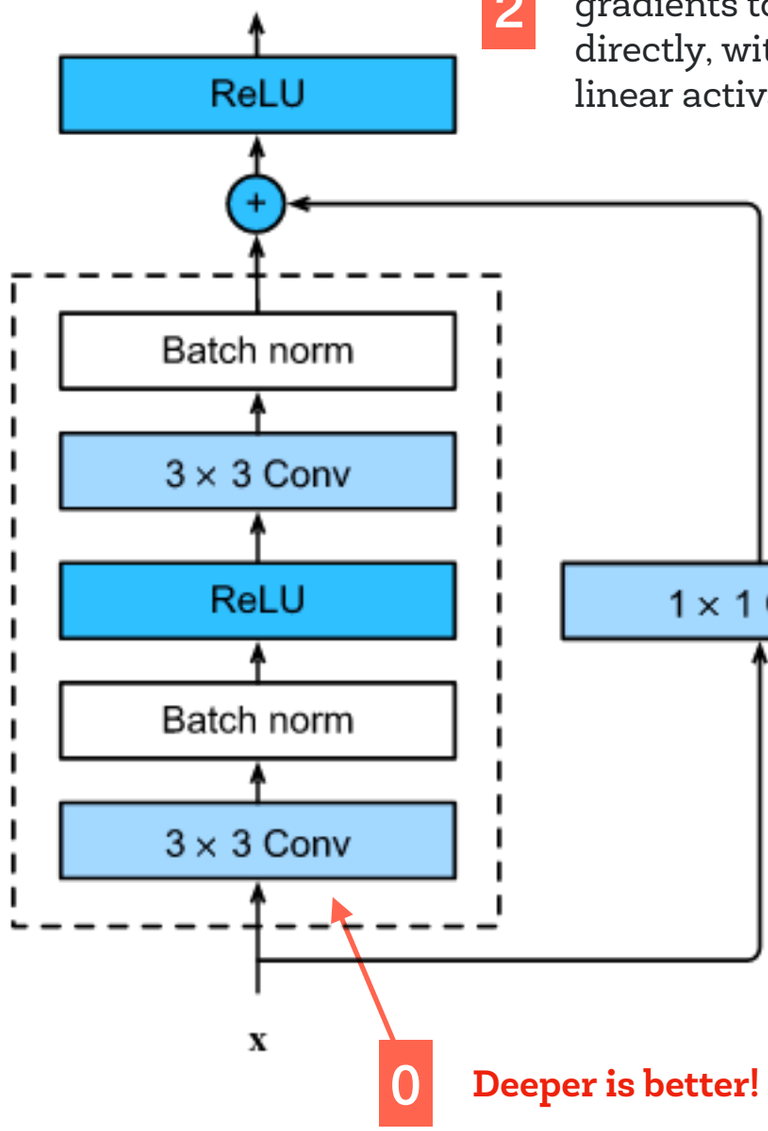
https://d2l.ai/chapter_convolutional-neural-networks/lenet.html

ResNet (2016)

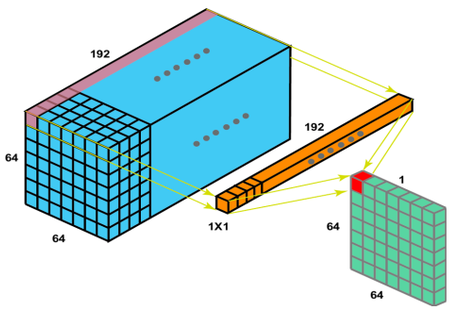


3 **Batch normalization** applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

2 **Residual connections** are used to allow gradients to flow through a network directly, without passing through non-linear activation functions.



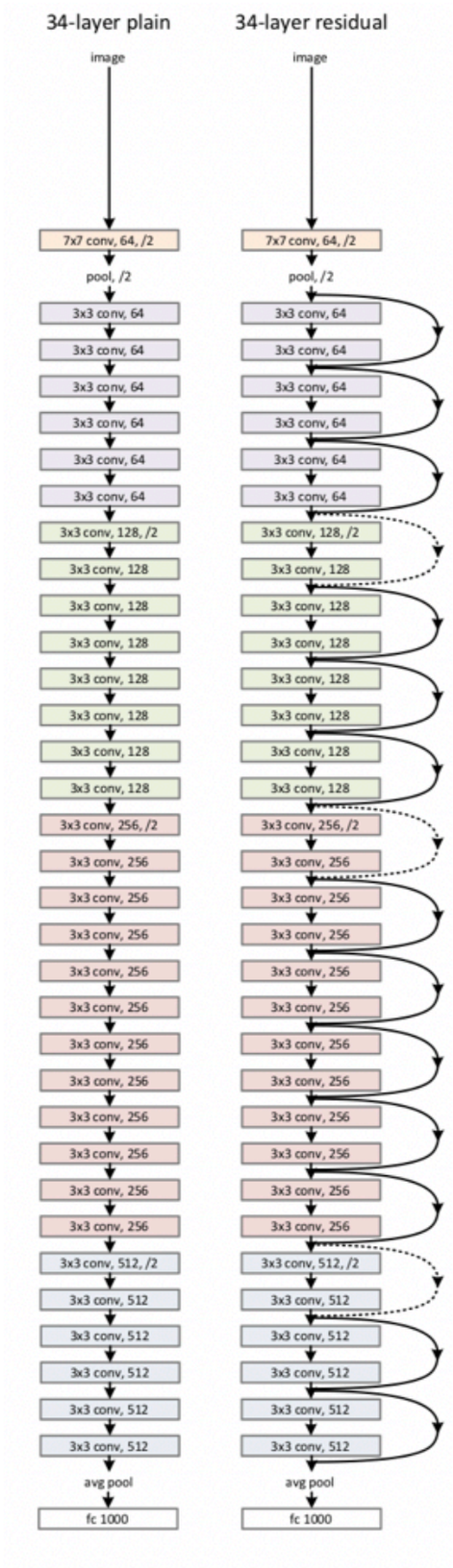
1 **1x1 convolutional filters** can be used to **change the dimensionality** in the filter space.



0 **Deeper is better!**

https://d2l.ai/chapter_convolutional-neural-networks/lenet.html

ResNet (2016)



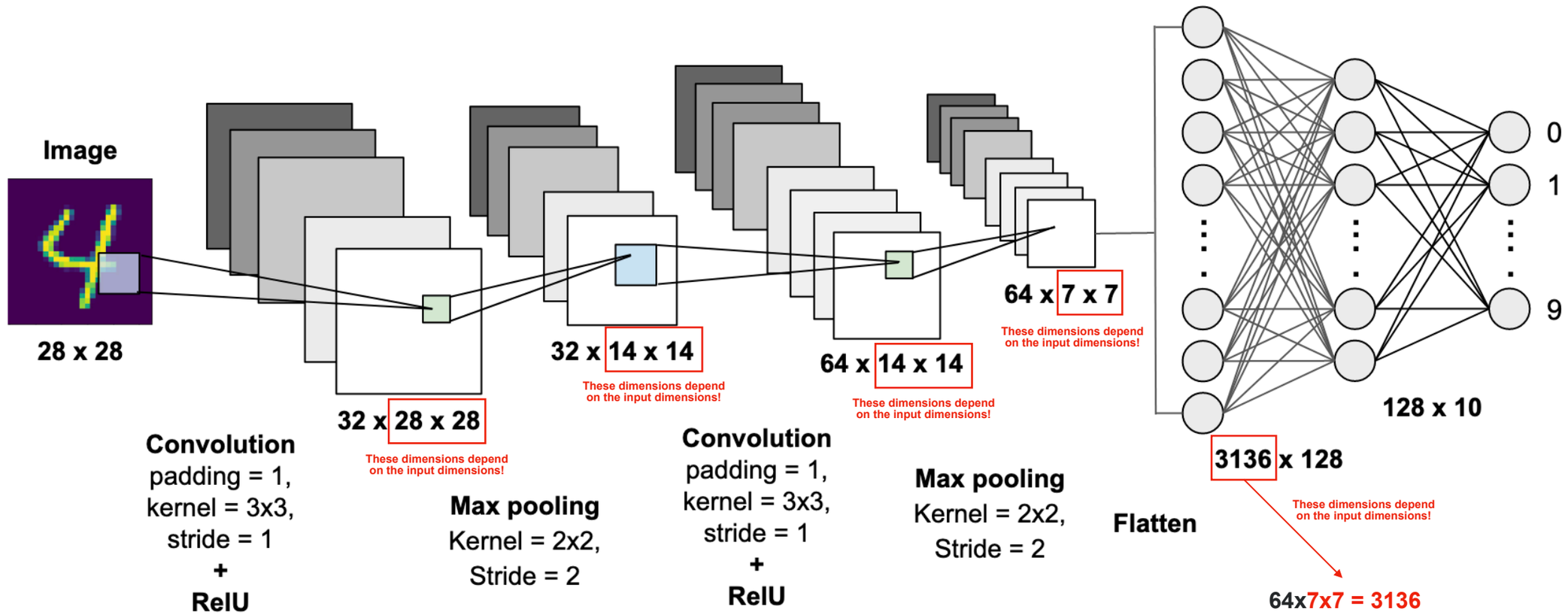
Residual Connections

Gradient information can be lost as we pass through many layers, this is called vanishing gradient.

Advantages of skip connection are they pass feature information to lower layers so learning becomes easier.

Fully Convolutional Networks (2015)

The use of a fully connected layer at the end of a model represents a severe limitation!



https://becominghuman.ai/building-a-convolutional-neural-network-cnn-model-for-image-classification-116f77a7a236

We can only classify images of this size! What about larger images?

Fully Convolutional Networks (2015)

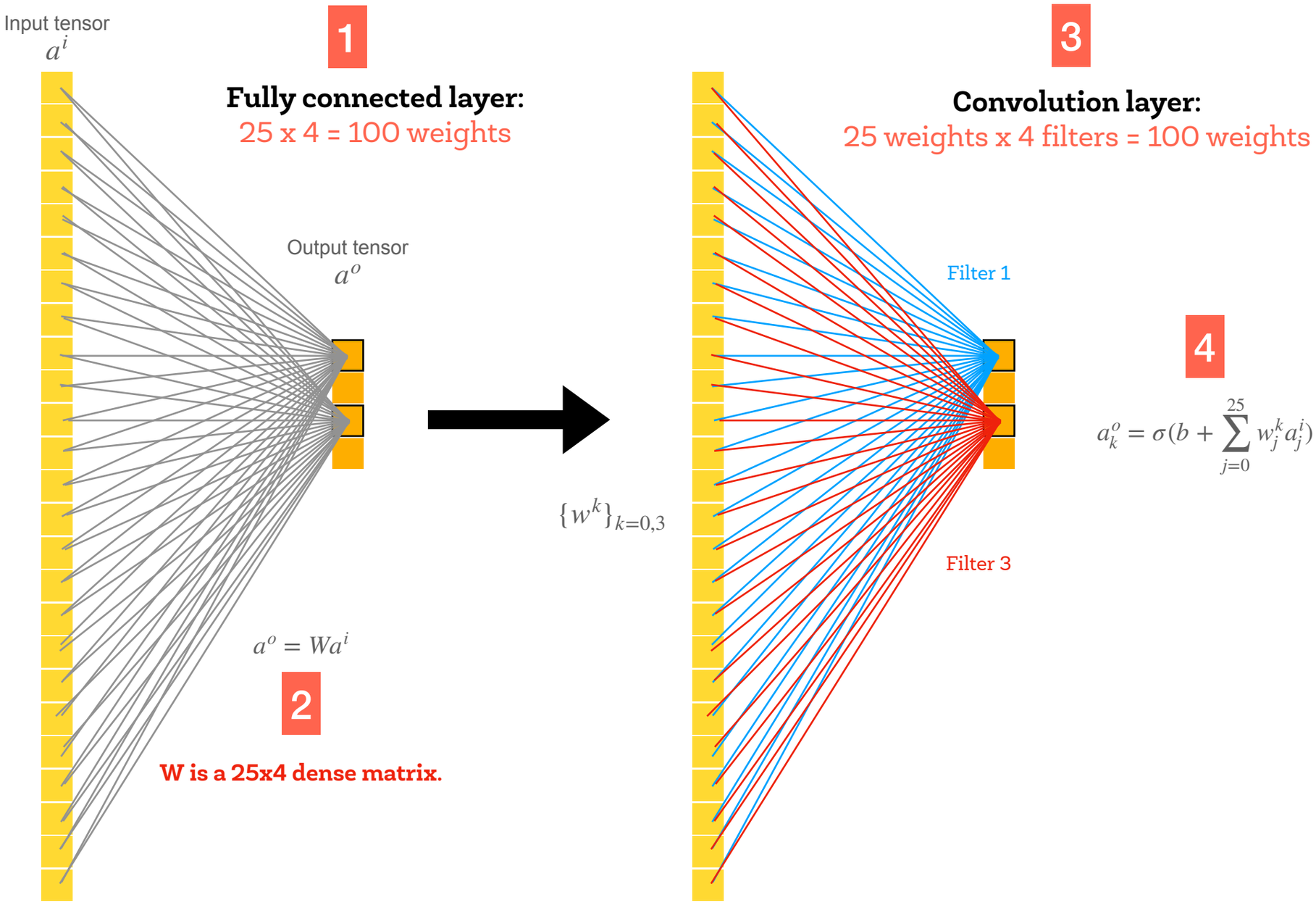
The only difference between Fully Connected (FC) and Convolutional (CONV) layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters.

However, **the neurons in both layers still compute dot products, so their functional form is identical**.

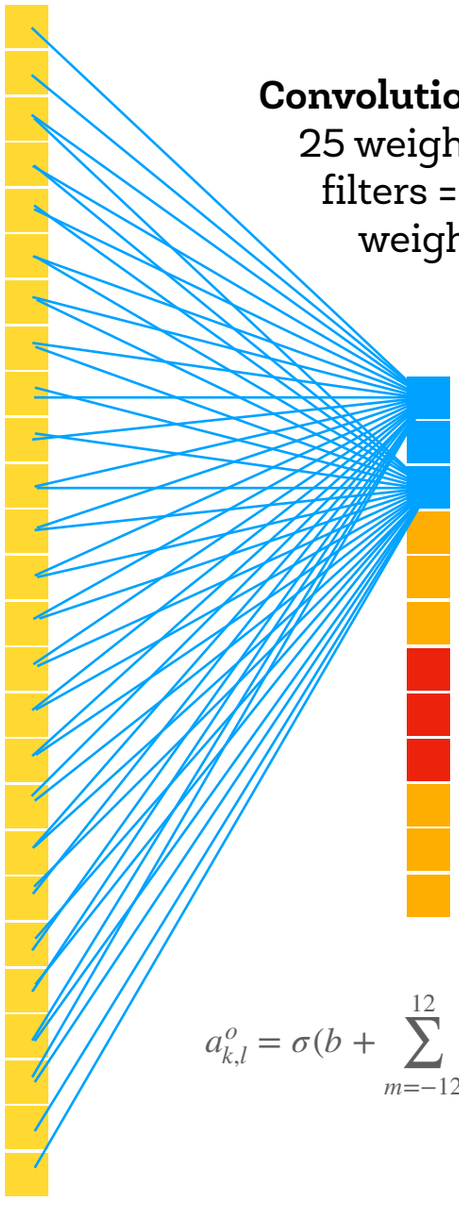
Then, it is easy to see that **for any FC layer there is an CONV layer that implements the same forward function**.

Fully Convolutional Networks (2015)

Any FC layer can be converted to a CONV layer.



Fully Convolutional Networks (2015)



$$a^o_{k,l} = \sigma(b + \sum_{m=-12}^{12} w^k_{m+12} a^i_{k+m+l})$$

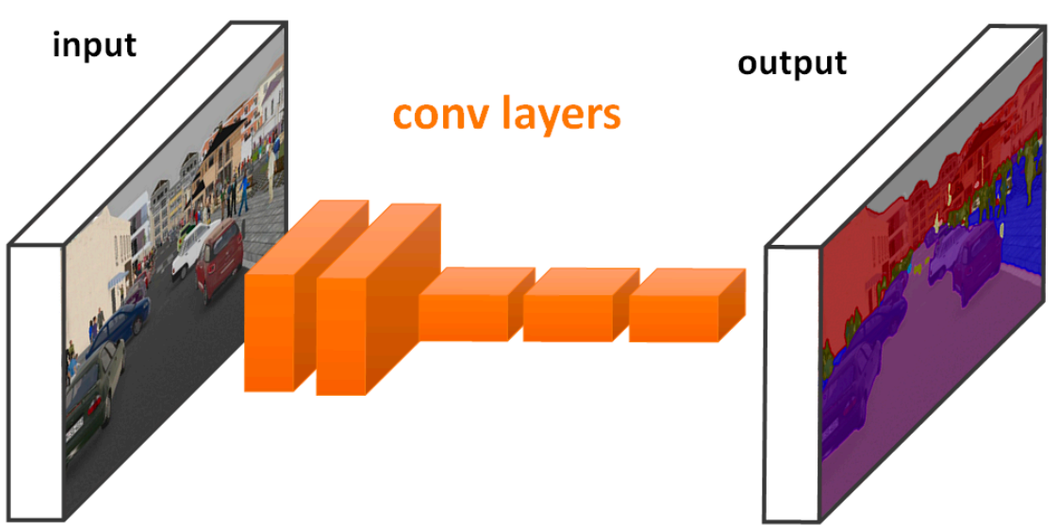
We can transform the first fully connected layer in a convolutional one for training. But during inference, **we can process bigger input tensors**!

If the input is bigger we get a set of additional output dimensions that can be understood as the **output of a sliding window classifier**.

Fully Convolutional Networks (2015)

It turns out that this conversion allows us to “slide” the original ConvNet very efficiently across many spatial positions in a larger image, in a single forward pass.

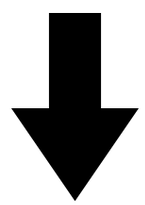
Now the model is independent of the input image size!



This is a fully convolutional network that has been trained to classify image patches. It can be applied to larger images to segment them.

Fully Convolutional Networks

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1600)              0
_________________________________________________________________
dropout (Dropout)            (None, 1600)              0
_________________________________________________________________
dense (Dense)                (None, 10)                16010
=================================================================
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
```



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 1, 1, 10)          16010
_________________________________________________________________
global_average_pooling2d_2 ( (None, 10)                0
=================================================================
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
_________________________________________________________________
```

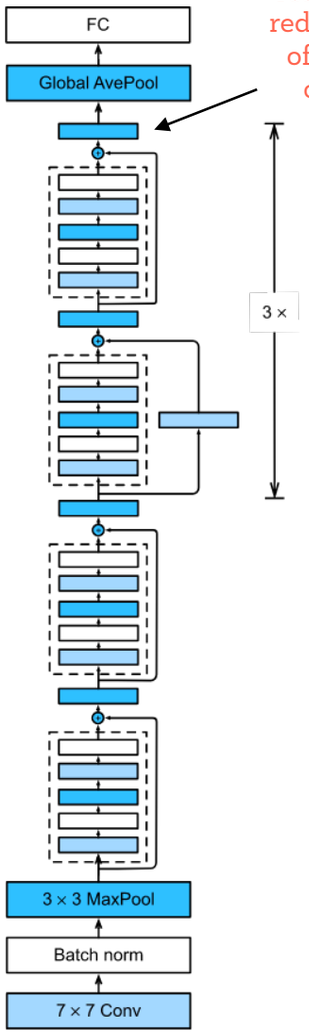
Semantic Segmentation

The layers we introduced so far for convolutional neural networks, including convolutional layers and pooling layers, often **reduce the input width and height**, or keep them unchanged.

Applications such as semantic segmentation, however, require to predict values for each pixel and therefore needs to increase input width and height. Transposed convolution, also named **deconvolution**, serves this purpose.

Semantic Segmentation

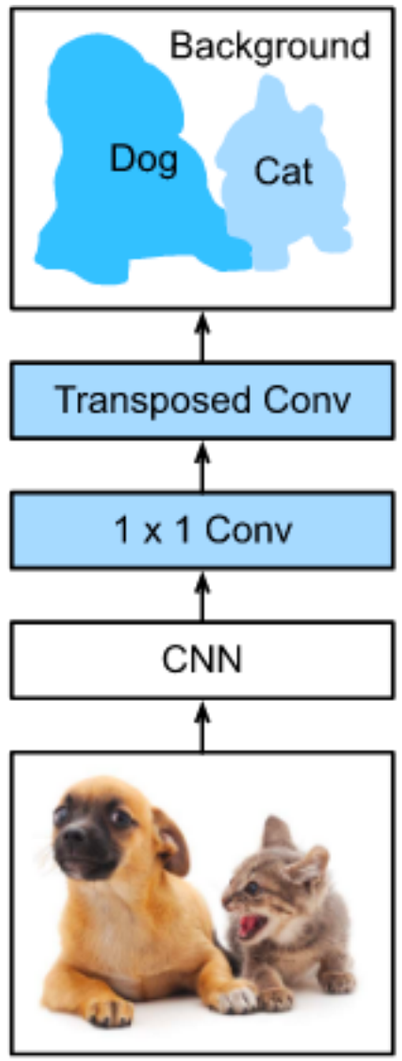
We can use a fully convolutional network. First we will use a convolutional neural network to extract **image features**, then we will transform the number of channels into the number of categories through the 1x1 convolution layer, and finally we will transfer the height and width of the feature map to the size of the input image by using a deconvolution layer.



Given an input of a height and width of 320 and 480 respectively, ResNet18 will reduce the height and width of the input to 1/32 of the original, i.e., 10 and 15.

1

1. We can use **ResNet18** to extract image features (last convolutional layer output before global average pooling layer).



2

2. Next, we transform the number of output channels to the number of object categories of through the 1×1convolution layer, building a classification map.

3

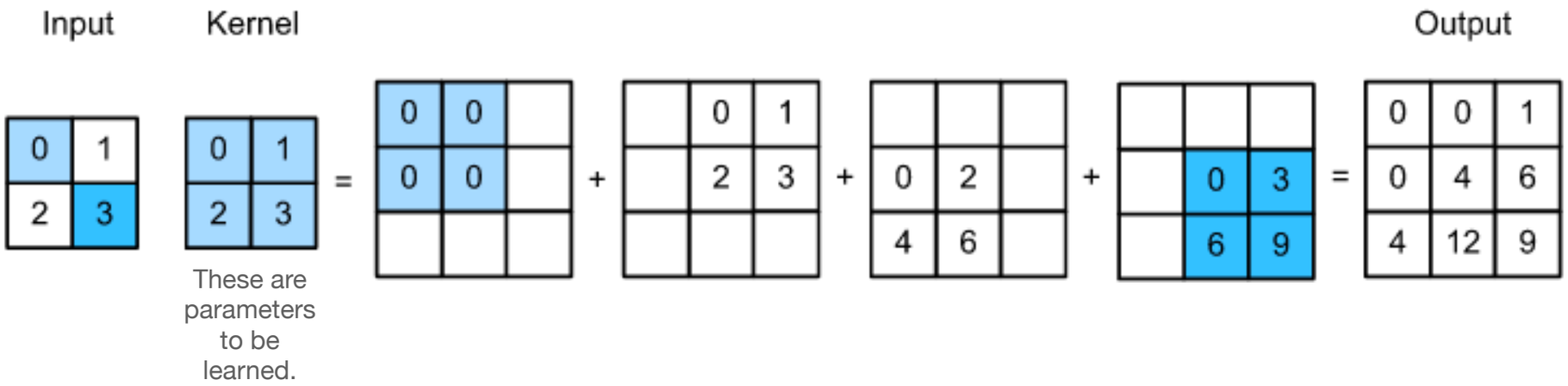
3. Finally, we need to magnify the height and width of the feature map by a factor of 32 to change them back to the height and width of the input image.

https://d2l.ai/chapter_computer-vision/fcn.html

Deconvolution

Let us consider a basic case that both input and output channels are 1, with 0 padding and 1 stride.

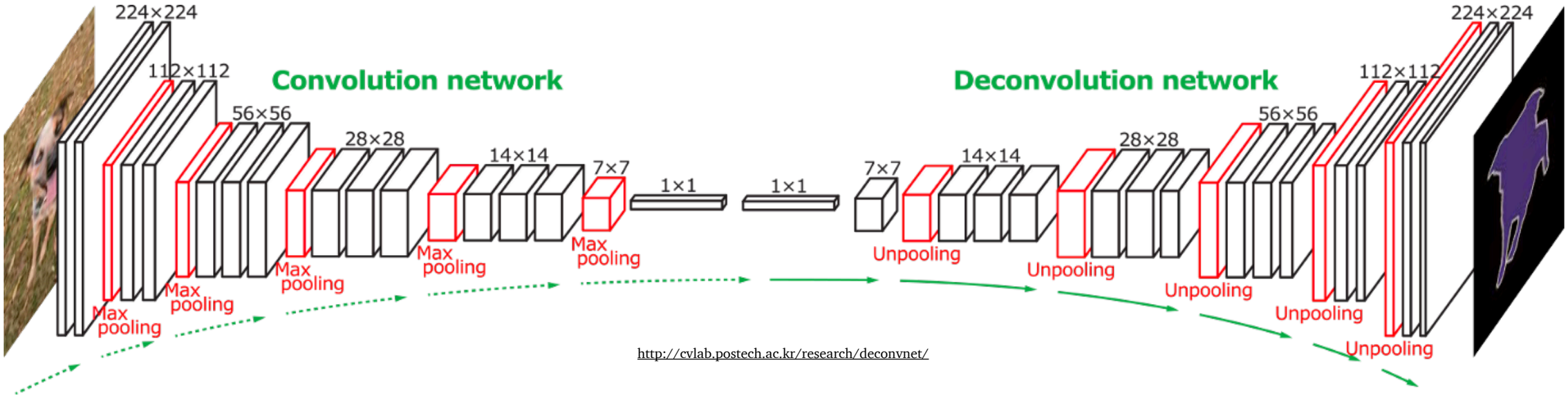
This is how transposed convolution with a 2×2 kernel is computed on the 2×2 input matrix:




```
def trans_conv(X, K):
    h, w = K.shape
    Y = np.zeros((X.shape[0] + h - 1, X.shape[1] + w - 1))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Y[i: i + h, j: j + w] += X[i, j] * K
    return Y
```

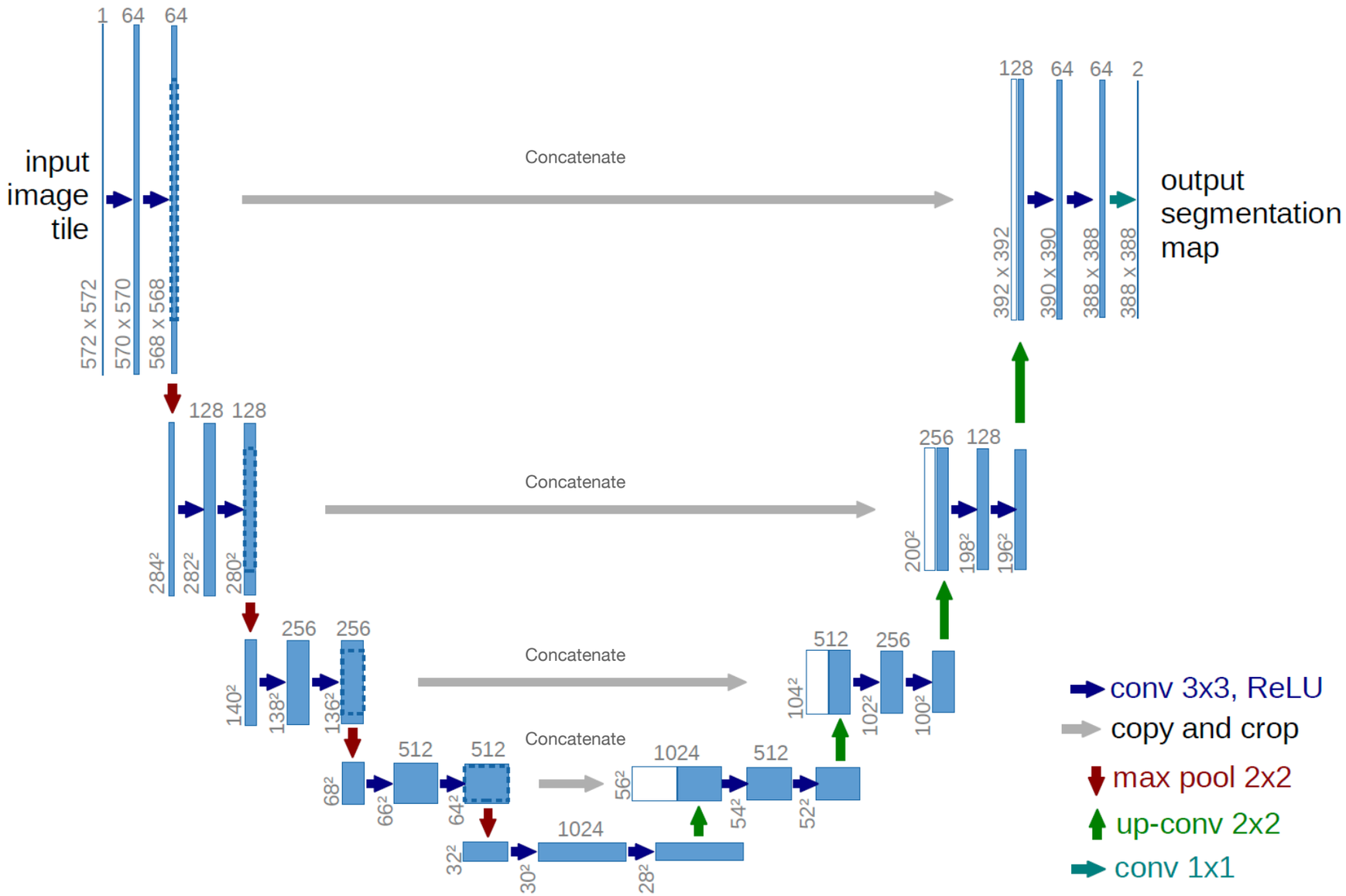
https://d2l.ai/chapter_computer-vision/fcn.html

Semantic Segmentation



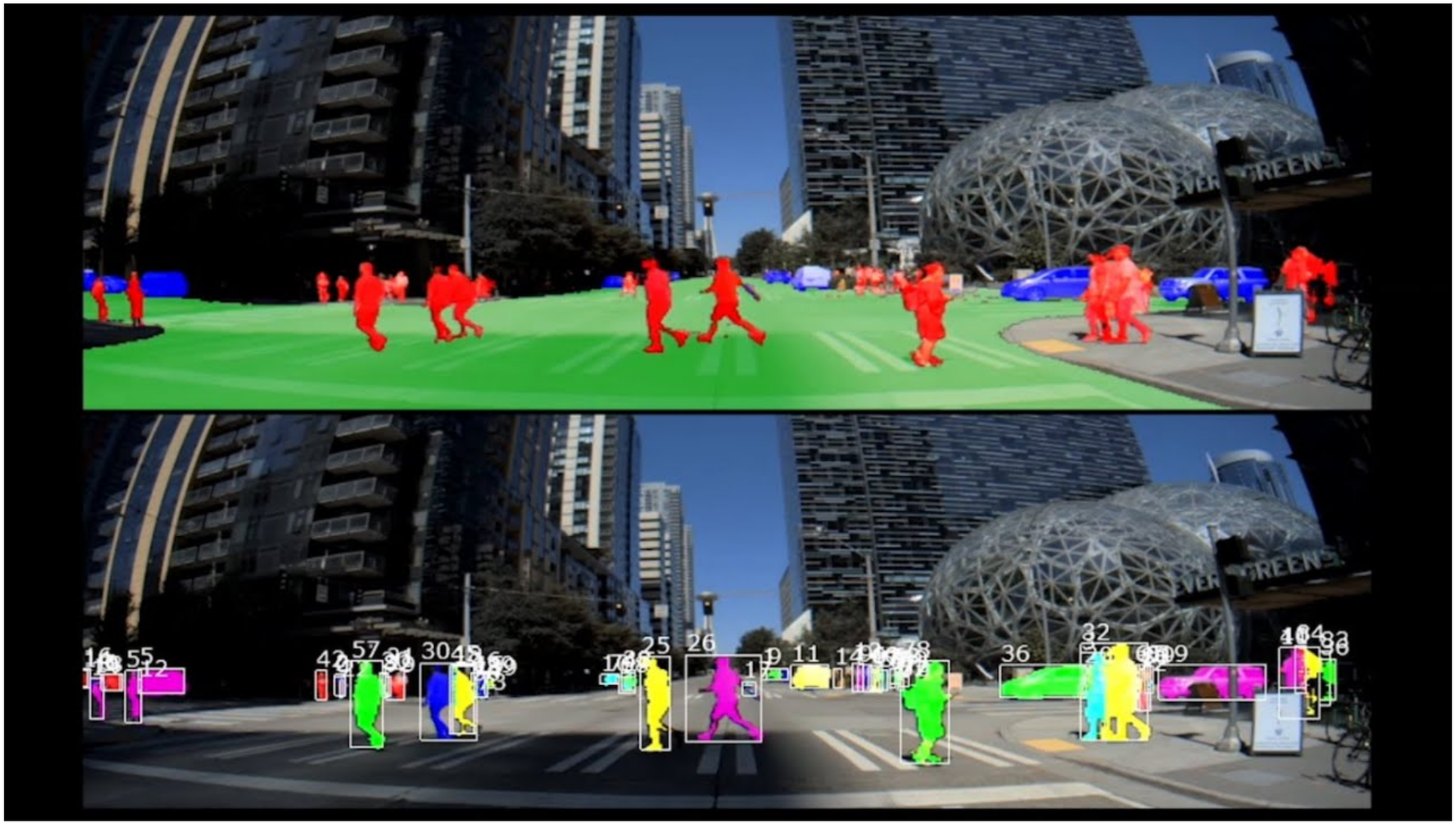
http://cvlab.postech.ac.kr/research/deconvnet/

Semantic Segmentation: U-Net



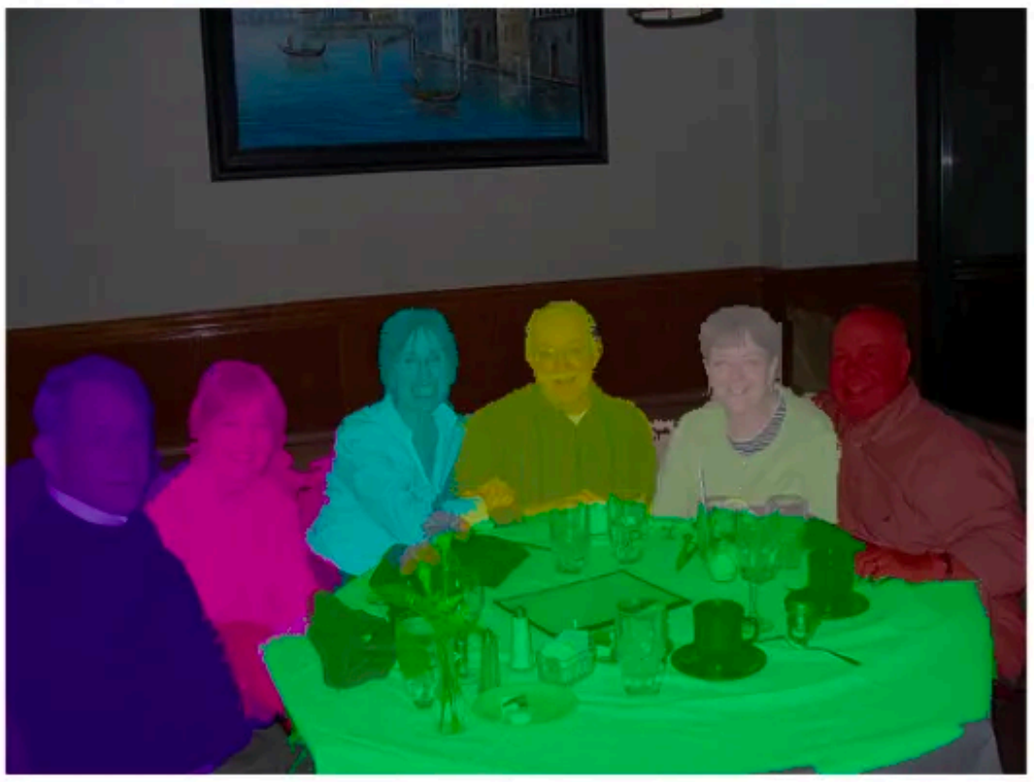
https://medium.com/@nishanksingla/unet-with-resblock-for-semantic-segmentation-dd1766b4ff66

Semantic Segmentation



Semantic Segmentation

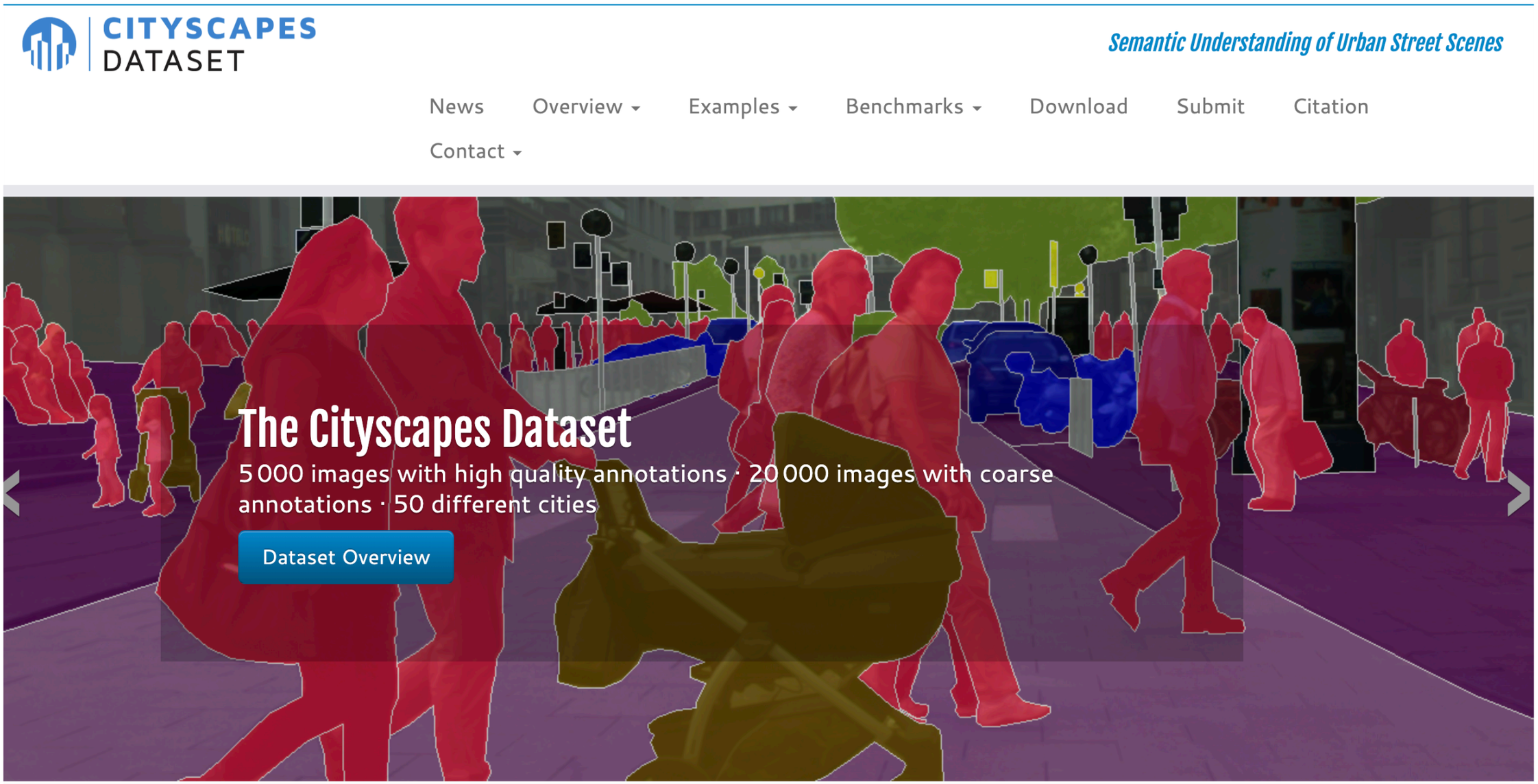
Semantic Segmentation



Instance Segmentation

http://www.robots.ox.ac.uk/~tvg/publications/2017/CRFMeetCNN4SemanticSegmentation.pdf

Semantic Segmentation: Databases



Assignment: Facial Point Detection

> You will need **GPU computation**. Consider the use of Colab!