

# Machine Intelligence:: Deep Learning

## Week 2

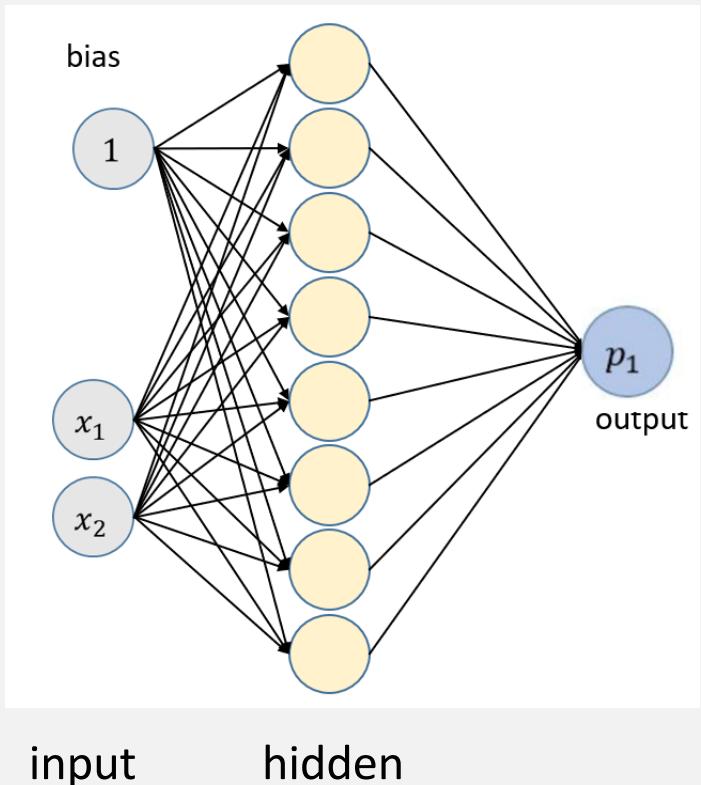
*Beate Sick, Oliver Dürr*

Institut für Datenanalyse und Prozessdesign  
Zürcher Hochschule für Angewandte Wissenschaften

# Topics of today

- A second look on fully connected Neural Networks (fcNN)
  - Classification loss if more than one class
  - ReLU
  - Training Testset
- Convolutional Neural Networks (CNN) for images
  - Motivation for switching from fcNN to CNNs
  - Introduction of convolution
  - ReLu and Maxpooling Layer
  - Biological inspiration of CNNs
  - Building CNNs

# Besprechung



input

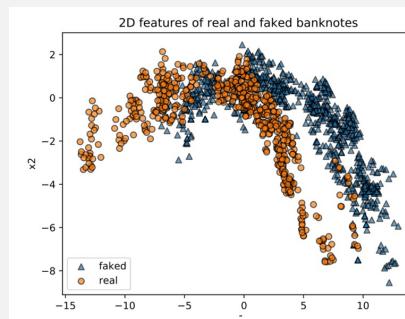
hidden

Open NB [01\\_simple\\_forward\\_pass.ipynb](#) and do the Keras part

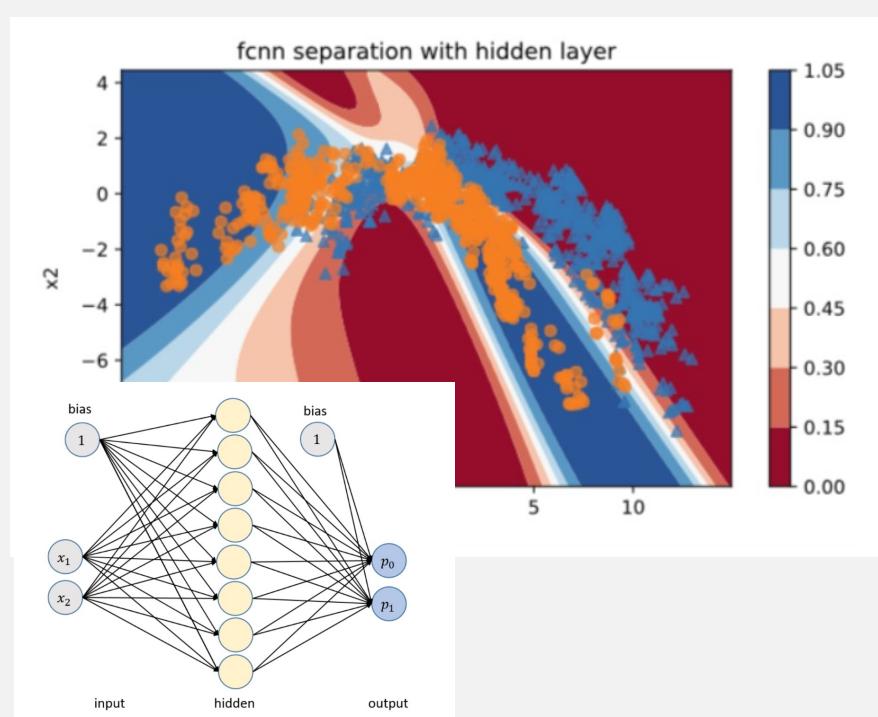
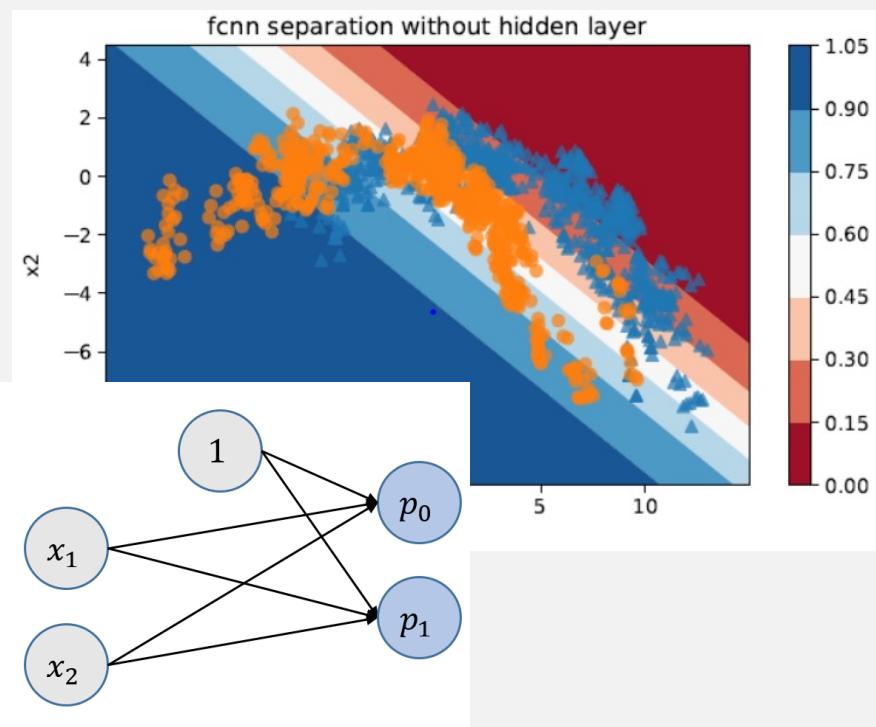
# Additional Homework

Do exercise NB 02:

Use Keras for modeling the banknotes data with binary outcome and 2 continuous input features ( $x_1, x_2$ )

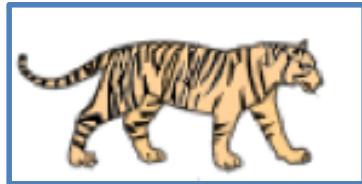


Fit the a fcNN w/o and with hidden layer to discriminate fake bank notes from real bank notes: NB02



# Recap Training

Input  $x^{(i)}$



True class  $y^{(i)}$

Tiger



Tiger



Seehorse

...

Typical 1 Mio. Trainingsdaten

Suggested class  
(Show is most likely class)

→ Seal 🤢

→ Tiger 👍

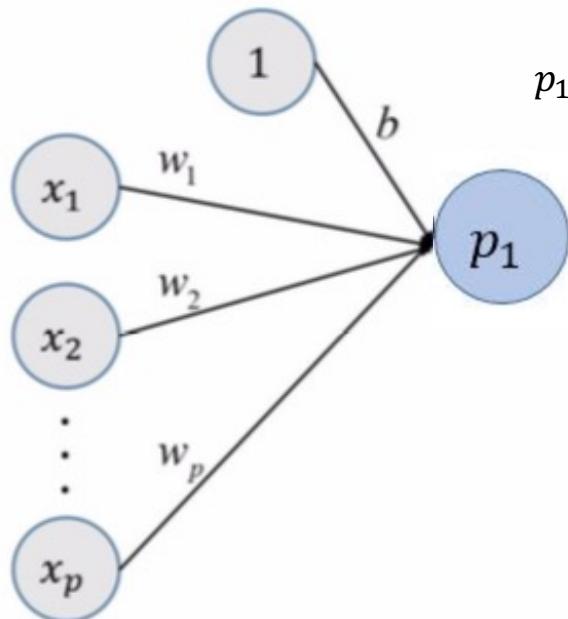
→ Seehorse 👍

Neural network with many weights  $W$

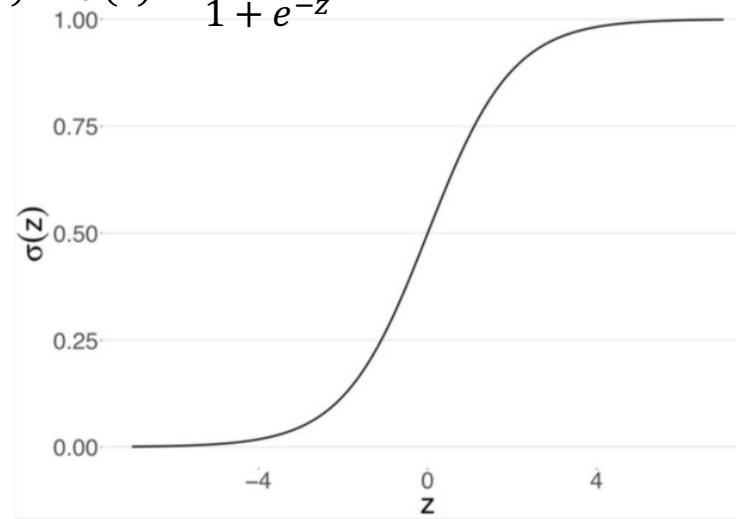


Trainingsprinciple:  
Weights are tuned so a loss function gets minimized.  
 $loss = loss(\{y^{(i)}, x^{(i)}\}, W)$

# Recap Loss for Single Output



$$p_1 = P(Y = 1|z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



## Loss

$p_1 = P(Y = 1|x)$  if the observed class is  $y_i = 1$

$p_0 = P(Y = 0|x)$  if observed is  $y_i = 0$

$$\text{LogLikelihood} = \sum_{i=1}^n [y_i \log(p_{1i}) + (1 - y_i) \log(1 - p_{1i})]$$



# Classification

- Predict class
- Usually in DL the model predicts a probability for each possible class
- Example:
  - Banknote from exercise – classes: “real” or “fake”
  - Typical example Number from hand-written digit – classes: 0, 1, ..., 9

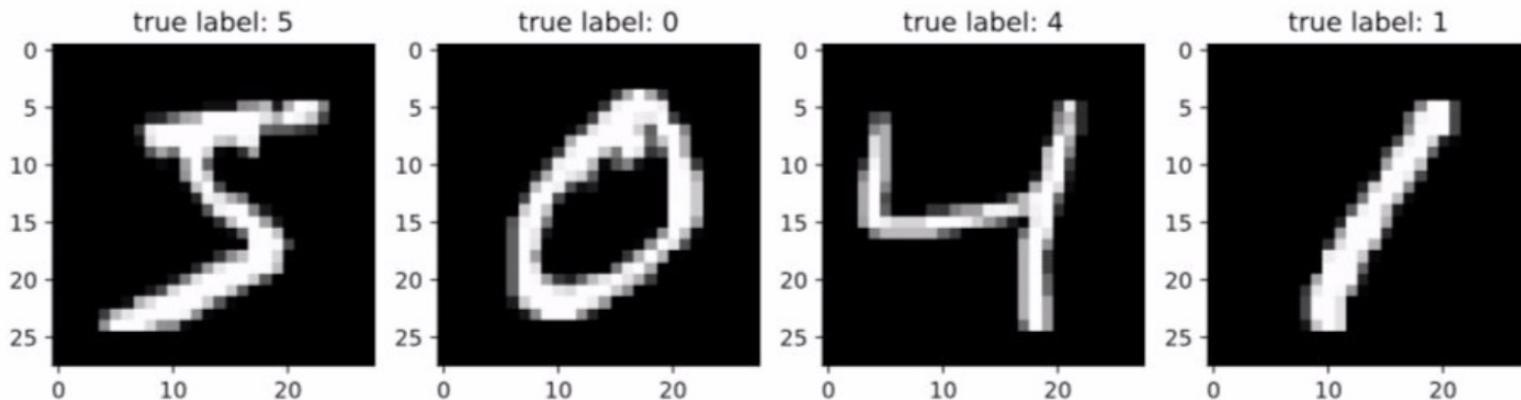
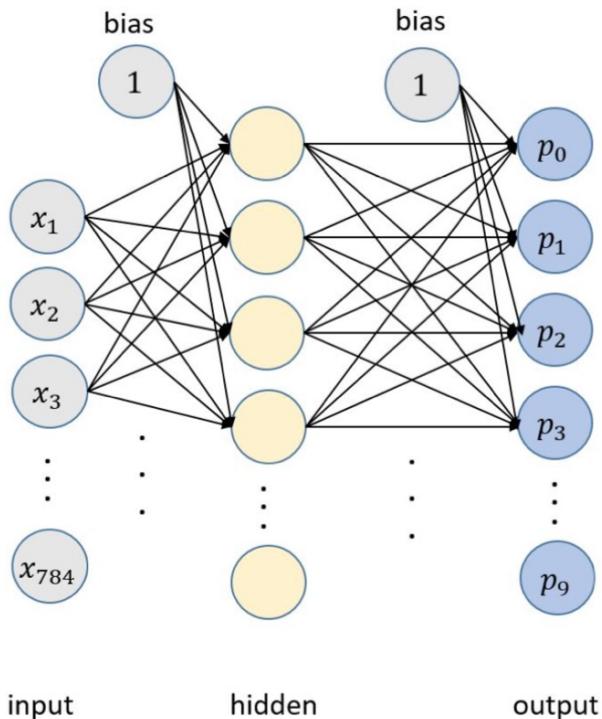


Figure 2.11 The first four digits of the MNIST data set—the standard data set used for benchmarking NN for images classification

# Classification: Softmax Activation



$p_0, p_1 \dots p_9$  are probabilities for the classes 0 to 9.

Activation of last layer  $z_i$  incoming

$$p_i = \frac{e^{z_i}}{\sum_{j=0}^9 e^{z_j}}$$

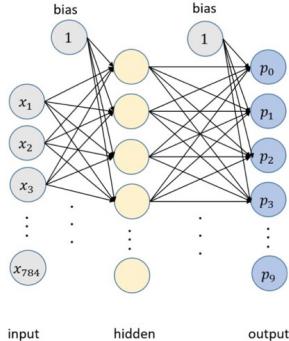
Makes outcome positive

Ensures that  $p_i$ 's sum up to one

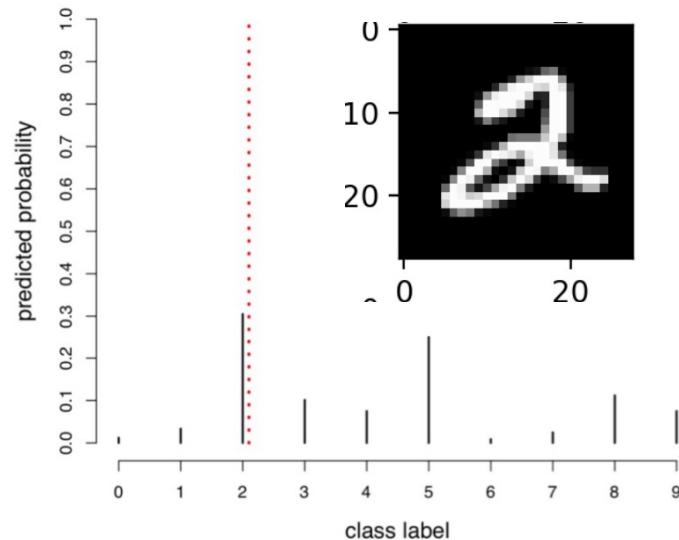
This activation is called softmax

Figure 2.12: A fully connected NN with 2 hidden layers. For the MNIST example, the input layer has 784 values for the 28 x 28 pixels and the output layer out of 10 nodes for the 10 classes.

# Loss for classification ('categorical cross-entropy')



$p_0, p_1 \dots p_9$  are probabilities for the classes 0 to 9.



Definition (Negative Log-likelihood NLL / cat. crossentropy):

The loss  $l_i$  of a single training example  $x^{(i)}$  with true label  $y^{(i)}$  is

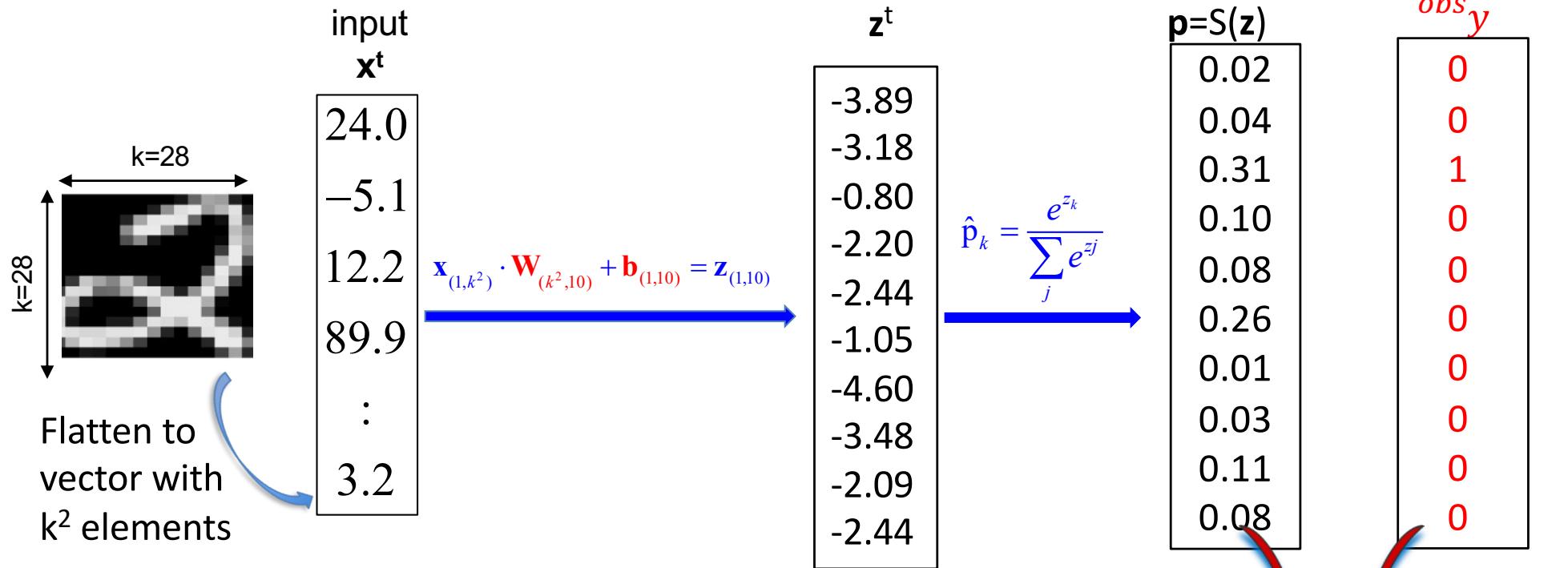
$$l_i = -\log(p_{model}(y^{(i)}|x^{(i)}))$$

Notation: if true label is 2 then  $p_{model}(y^{(i)}|x^{(i)}) = p_2$

- Perfect, i.e. predicts class of training example  $y^{(i)}$  with probability 1  $\Rightarrow l_i = 0$
- Worst, i.e. predicts class  $y^{(i)}$  with probability 0  $\Rightarrow l_i = \infty$

For more all examples, just average loss =  $\frac{1}{N} \sum l_i$

# What is going on in a 1 layer fully connected NN?

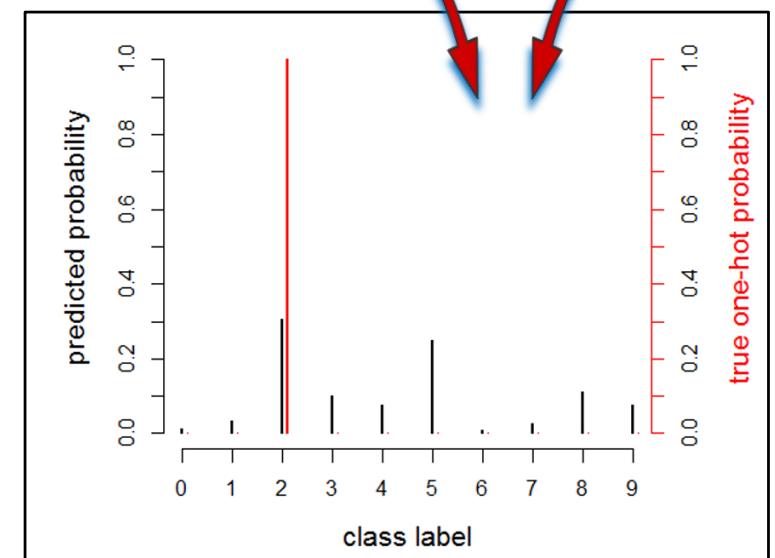


Cost C or Loss = cross-entropy averaged over all images in mini-batch

$$C = \frac{1}{N} \sum_i D(\mathbf{p}_i, \mathbf{y}_i)$$

Cross-Entropy

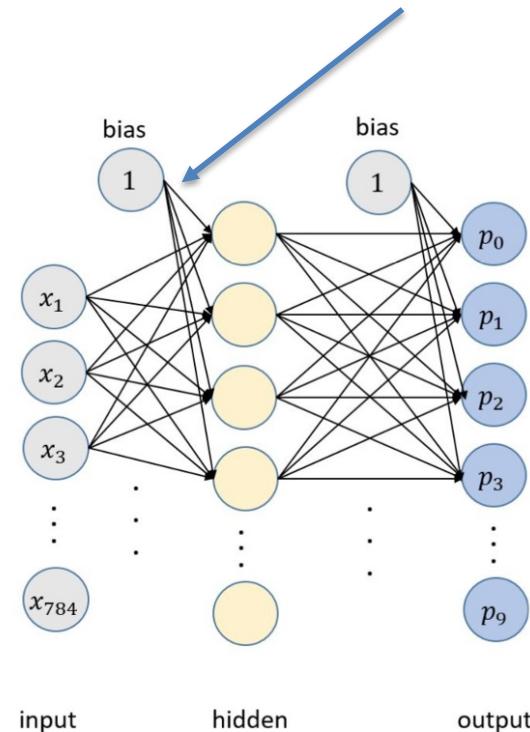
$$D(\mathbf{p}, \mathbf{y}) = -\sum_{k=1}^{10} {}^{obs} y_k \cdot \log(p_k)$$



# Optimization in DL

- DL many parameters
  - Optimization loss by simple gradient descent
- Algorithm: Stochastic Gradient Descent (SGD)\*
  - Take a random batch of training examples  $\{y^{(i)}, x^{(i)}\}_{i=1,\dots,B}$
  - Calculate the loss of that batch  $loss(\{y^{(i)}, x^{(i)}\}, W)$
  - Tune the weights so that loss gets minimized a bit (gradient descent)
  - Repeat

Parameters of the network are the weights.



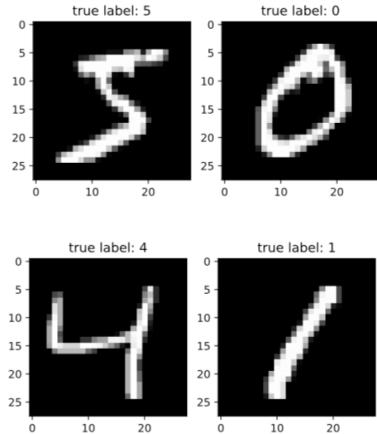
Modern Networks have Billions ( $10^9$ ) of weights.

Record 2020:  $175 \cdot 10^9$  (GPT-3)

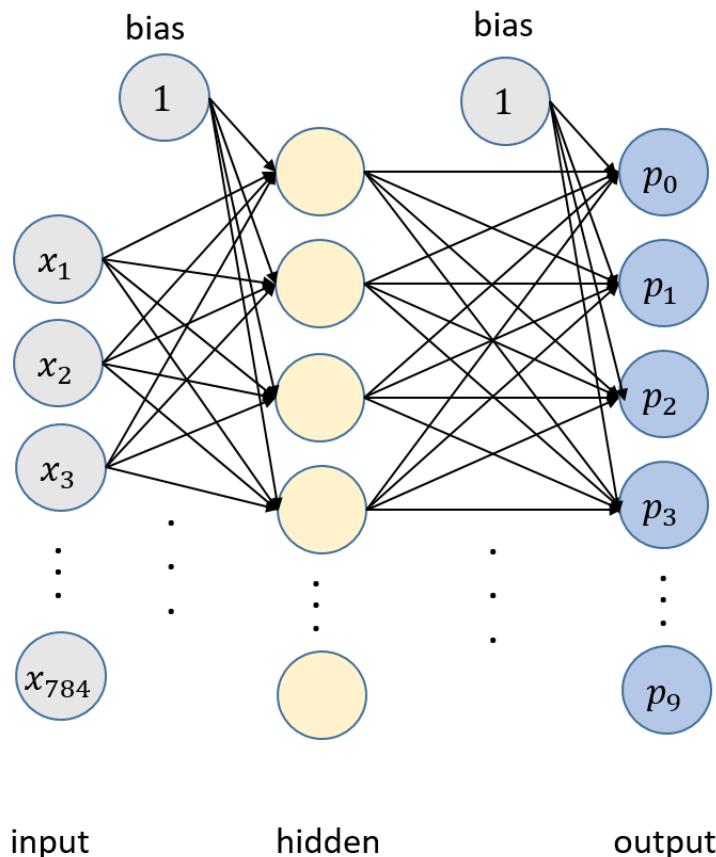
\*aka minibatch gradient descent.

Fully connected NN for image data  
Good Idea?

# A fcNN for MNIST data



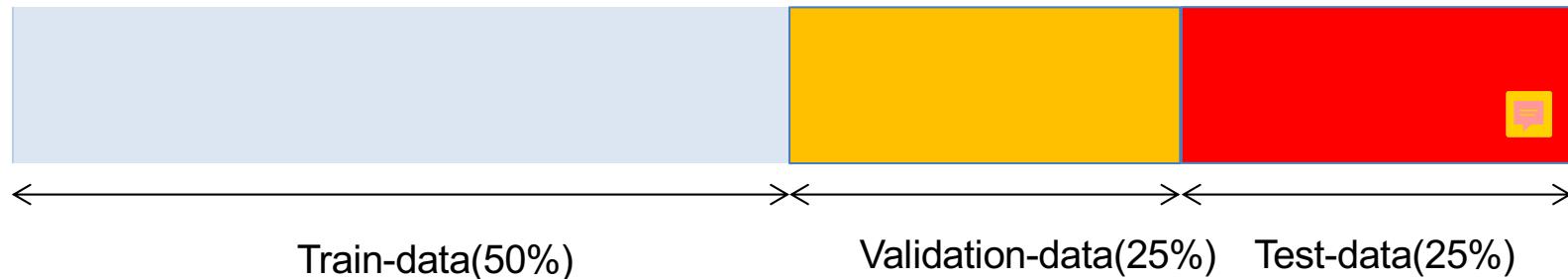
The first four digits of the MNIST data set - each image consisting of  $28 \times 28 = 784$  pixels



A fully connected NN with 2 hidden layers.

For the MNIST example, the input layer has 784 values for the  $28 \times 28$  pixels and the output layer has 10 nodes for the 10 classes.

# Best practice: Split in Train, Validation, and Test Set

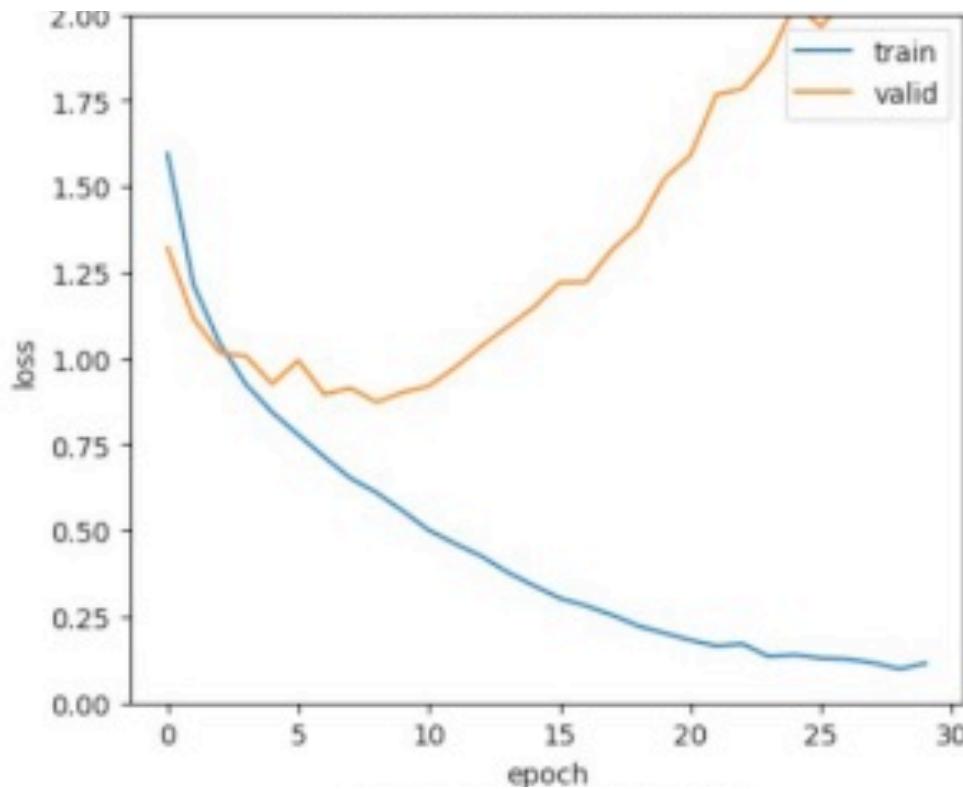


Best practice: Lock an extra **test data set** away, and use it only at the very end, to evaluate the chosen model, that performed best on your validation set.  
Reason: **When trying many models, you probably overfit on the validation set.**

Determine performance metrics, such as MSE, to evaluate the predictions **on new validation or test data**

## What can loss curves tell us?

Very common check: Plot loss in train and validation data vs epoch of training.



- If training loss does not go down to zero: model is not flexible enough
- In case of overfitting (validation loss >> train loss): regularize model

# Typical Training Curve / ReLU

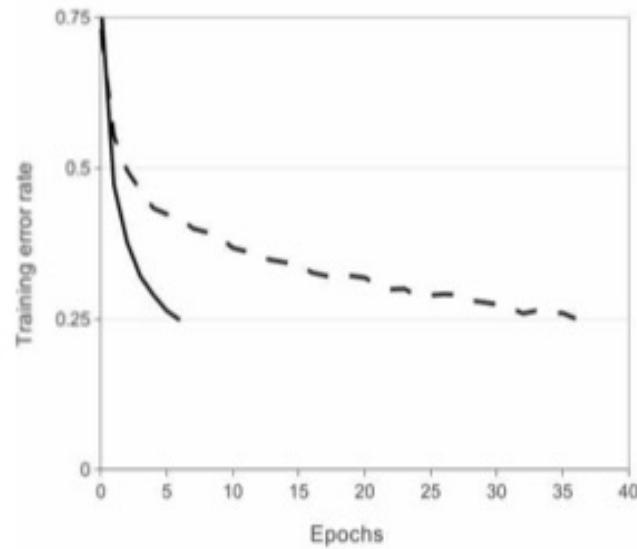
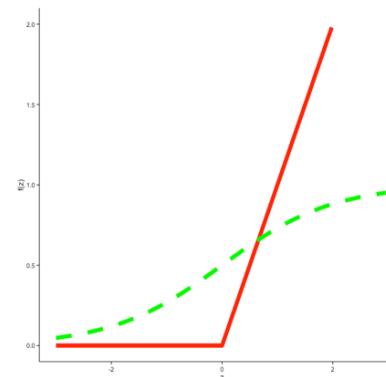
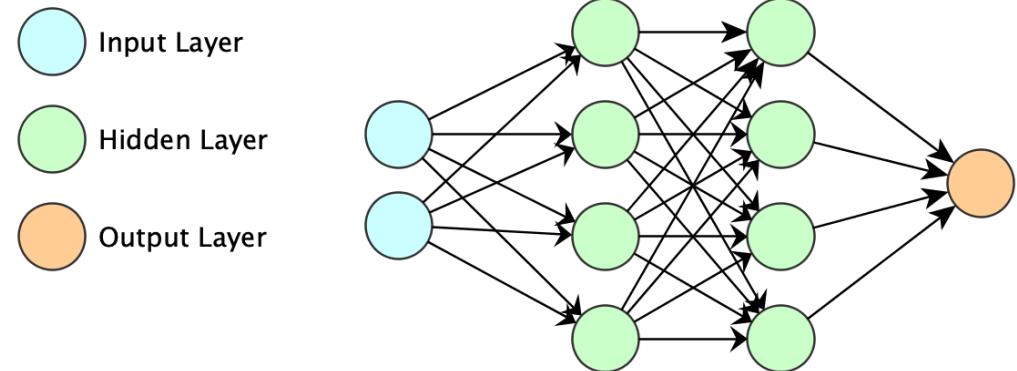


Figure 1: A four-layer convolutional neural network with ReLUs (solid line) reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons

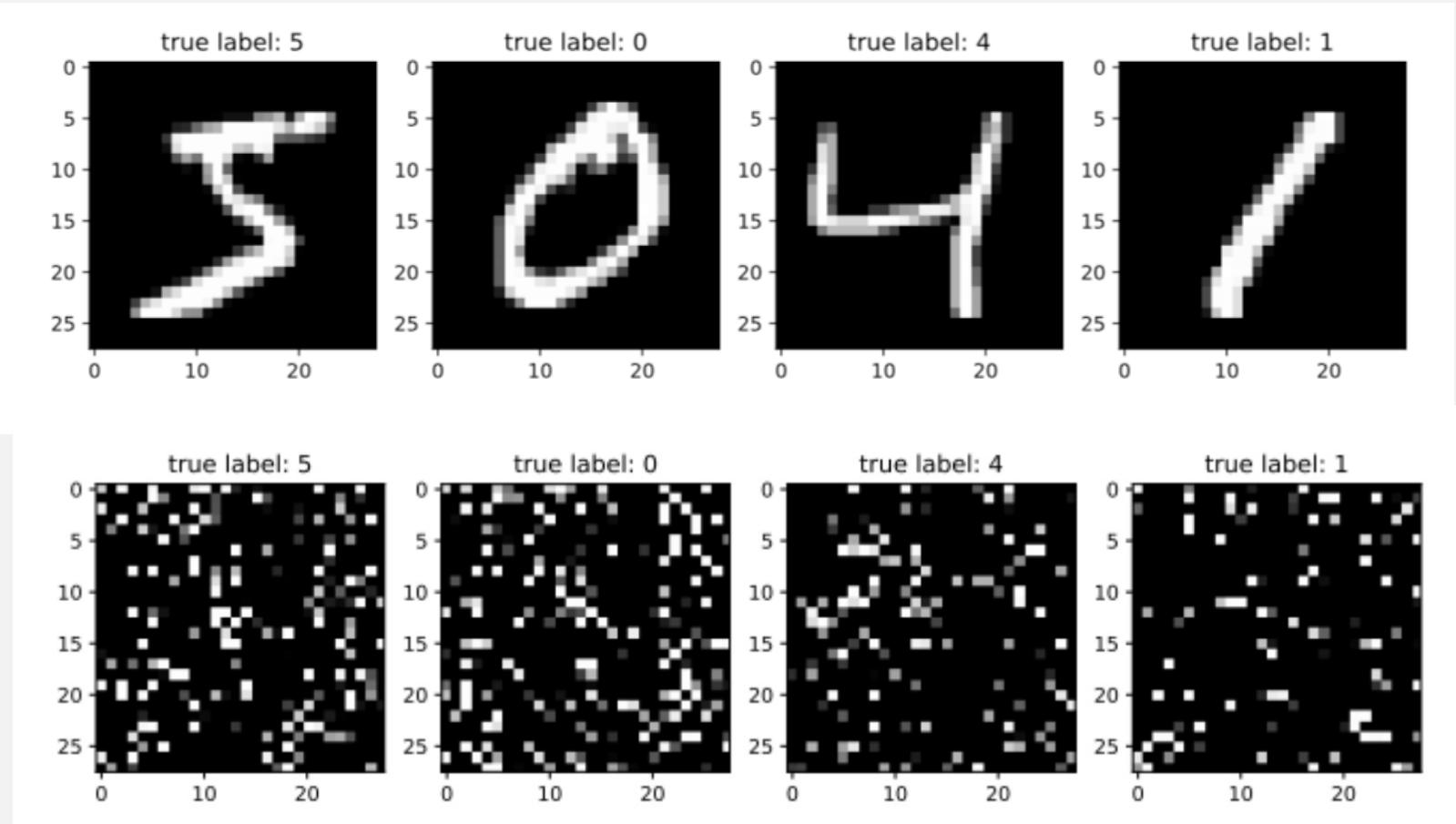
Source:  
Alexnet  
Krizhevsky et al 2012



Motivation:  
**Green:**  
sigmoid.  
**Red:**  
ReLU faster  
convergence

Epochs: "each training examples is used once"

## Exercise: Does shuffling disturb a fcNN?



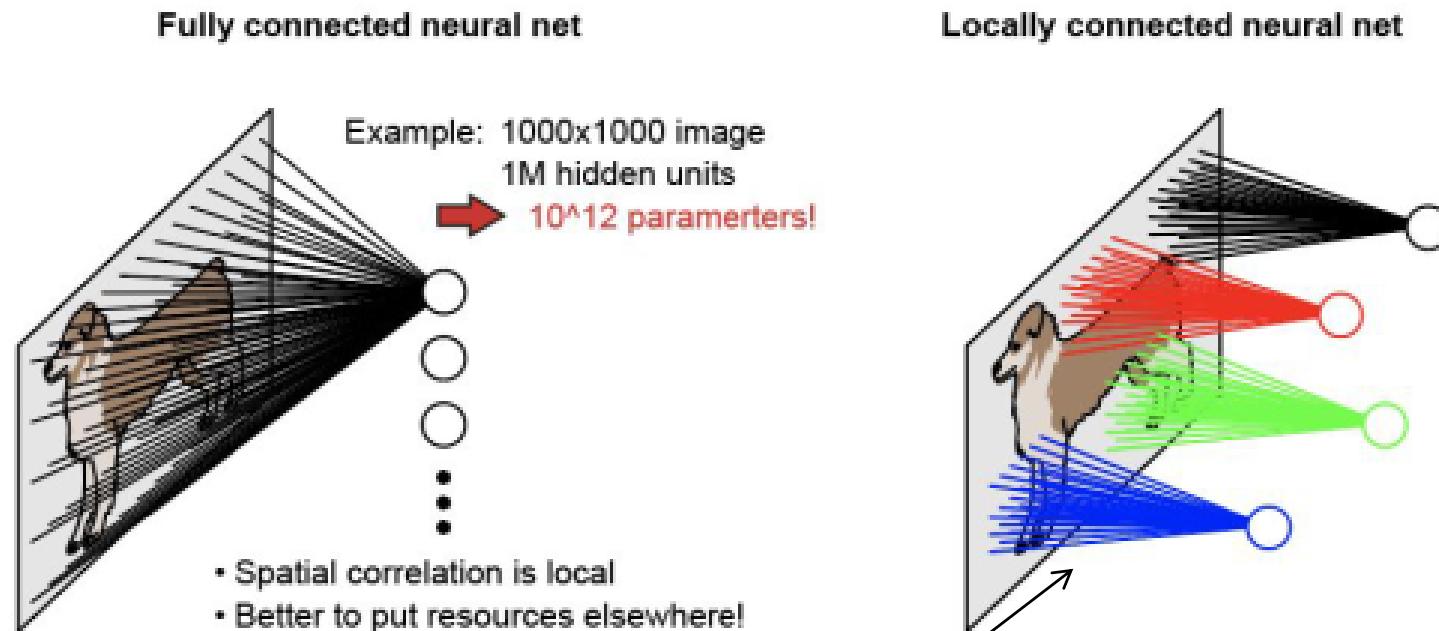
Use fcNN for MNIST:  
[03\\_fcnn\\_mnist.ipynb](#)

If time  
[04\\_fcnn\\_mnist\\_shuffled.ipynb](#)

# Convolutional Neural Networks

## SOTA for image data

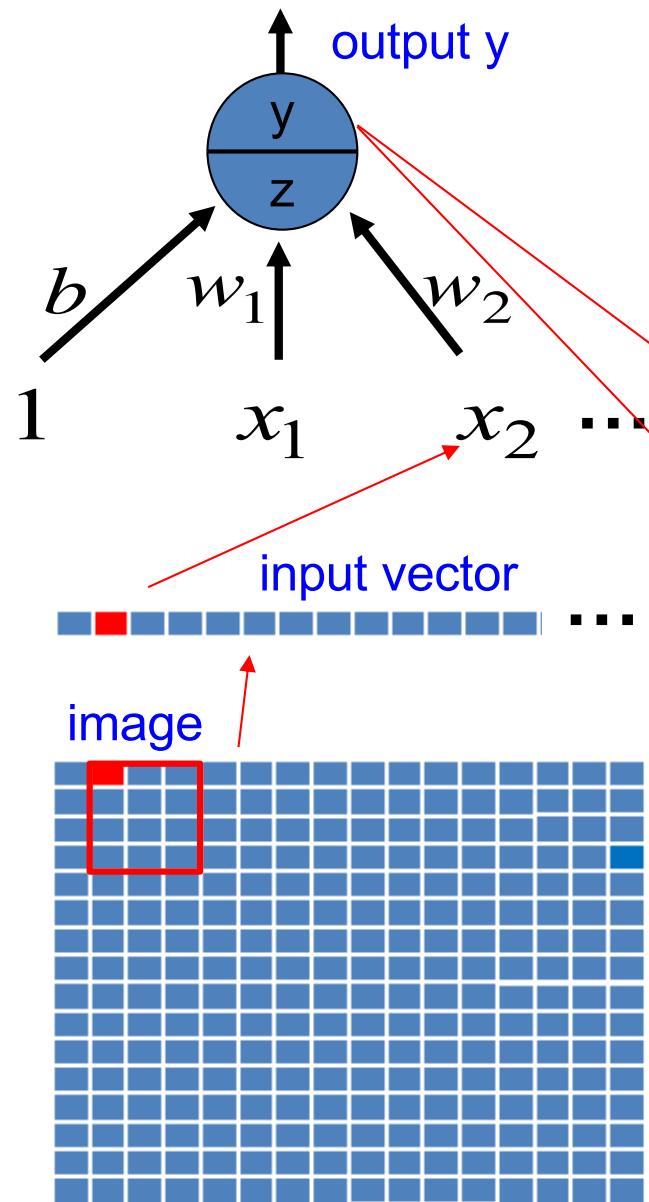
# Convolution extracts local information using few weights



Shared weights:

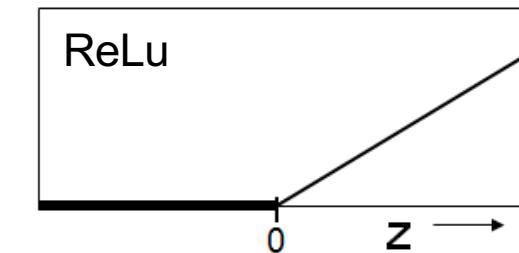
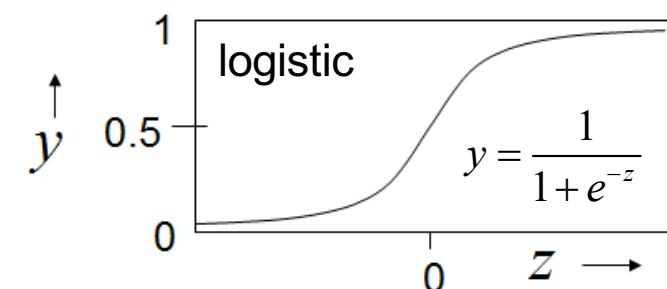
by using the **same weights for each patch** of the image we need much **less parameters** than in the fully connected NN and get from each patch the same kind of **local feature information** such as the presence of a edge.

# An artificial neuron



$$z = b + \sum_i x_i w_i$$

Different non-linear transformations are used to get from  $z$  to output  $y$



Convolutional networks use neighborhood information and replicated local feature extraction

In a locally connected network the calculation rule

$$z = b + \sum_j x_i w_i$$

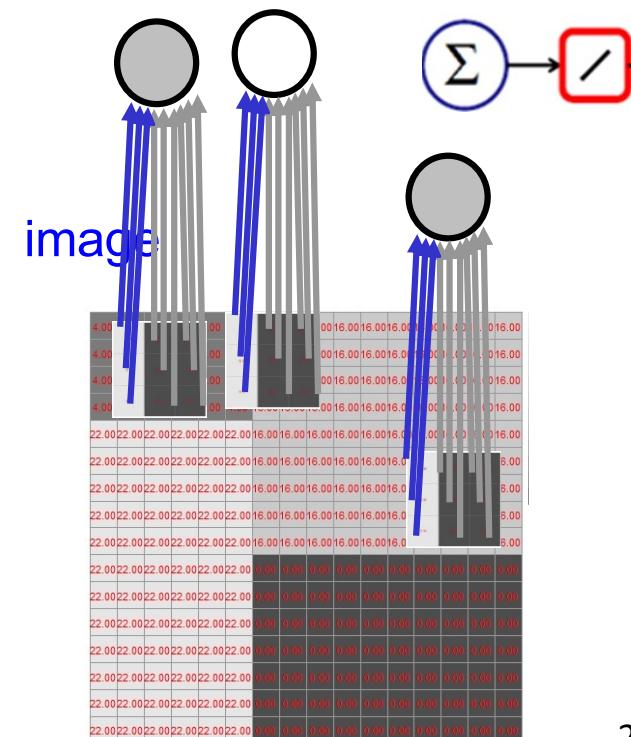
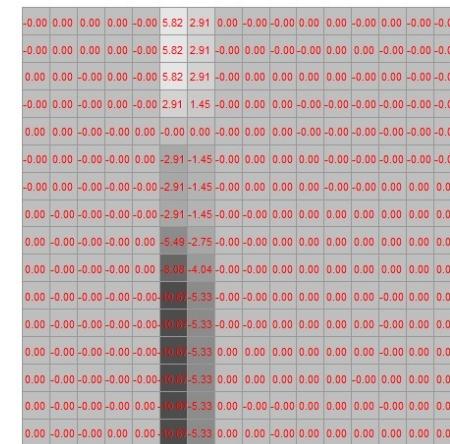
Pixel values in a small image patch are element-wise multiplied with weights of a small filter/kernel:



The filter is applied at each position of the image and it can be shown that the result is maximal if the image pattern corresponds to the weight pattern.

The results form again an image called **feature map** (=activation map) which shows at which position the feature is present.

## feature/activation map



## Exercise: Do one convolution step by hand

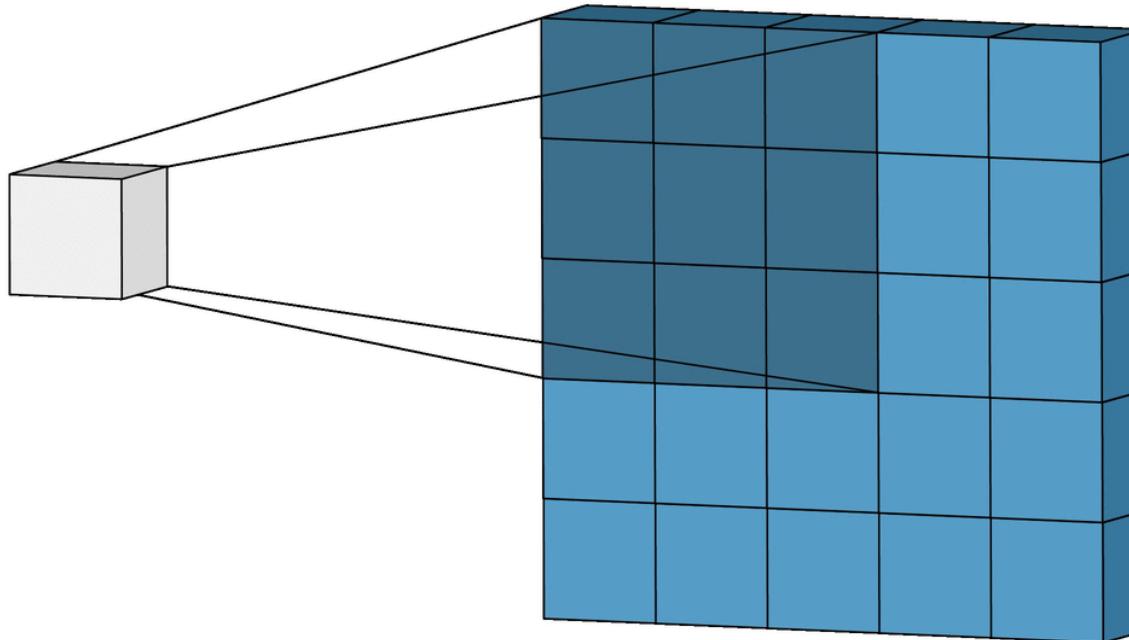


The kernel is 3x3 and is applied at each valid position  
– how large is the resulting activation map?

The small numbers in the shaded region are the kernel weights.  
Determine the position and the value within the resulting activation map.

3	3	2	1	0
0 <sub>0</sub>	0 <sub>1</sub>	1 <sub>2</sub>	3	1
3 <sub>2</sub>	1 <sub>2</sub>	2 <sub>0</sub>	2	3
2 <sub>0</sub>	0 <sub>1</sub>	0 <sub>2</sub>	2	2
2	0	0	0	1

## Applying the same $3 \times 3$ kernel at each image position



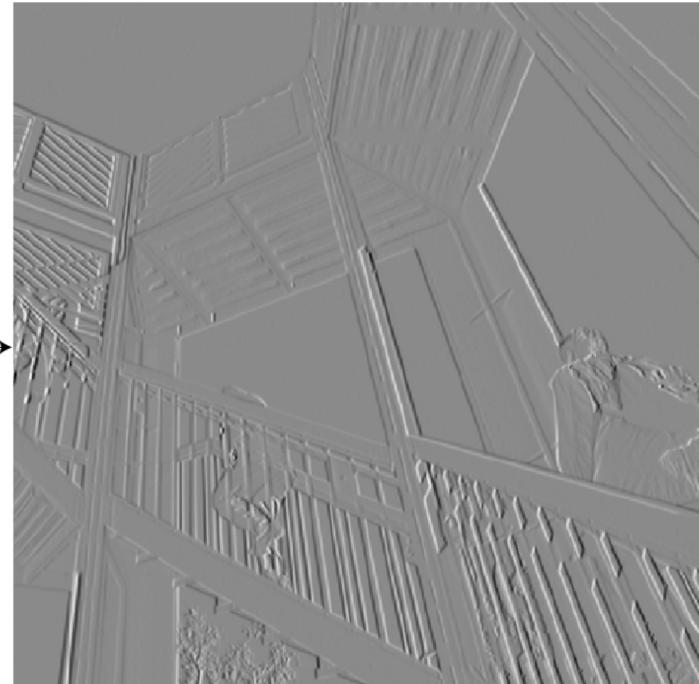
Applying the  $3 \times 3$  kernel on a certain position of the image yields one pixel within the activation map where the position corresponds to the center of the image patch on which the kernel is applied.

# Example of designed Kernel / Filter



$$\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}$$

Horizontal Sobel kernel



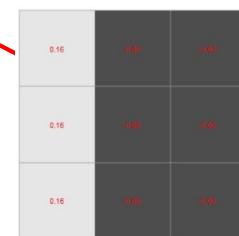
Applying a vertical edge detector kernel

Taken from the great website on convolution: <https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1>

Convolutional networks use neighborhood information and replicated local feature extraction

filtering = convolution      kernel 1

# image

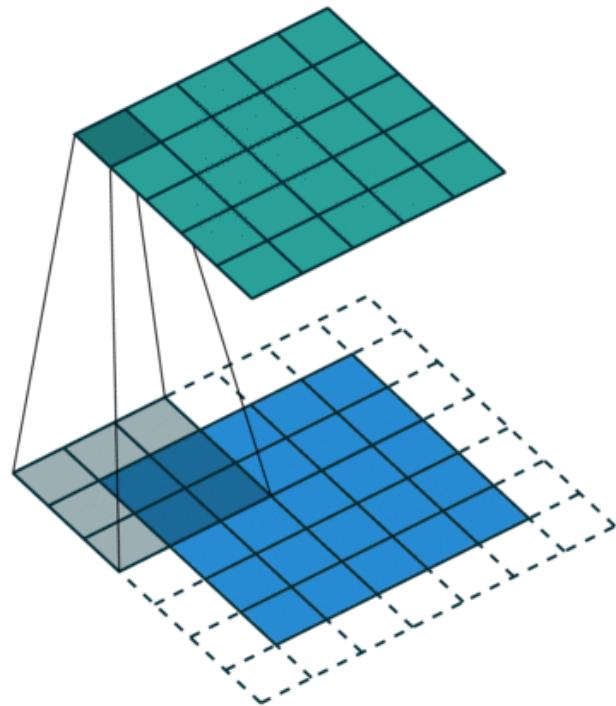


# feature map 1

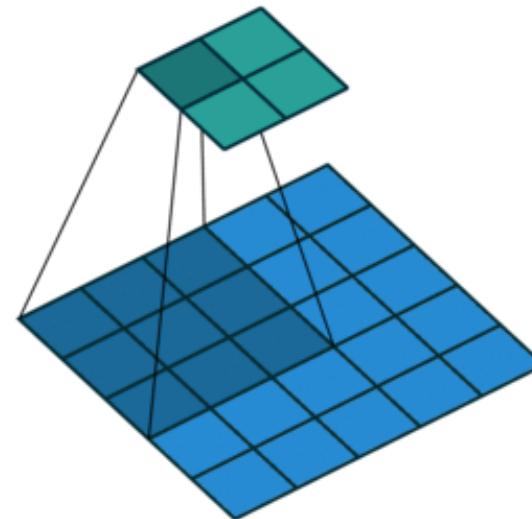
0.00 0.00 -0.00 5.82 2.91 0.00 -0.00 -0.00 0.00 0.00  
feature map 2

The weights of each filter are randomly initiated and then adapted during the training.

# CNN Ingredient I: Convolution



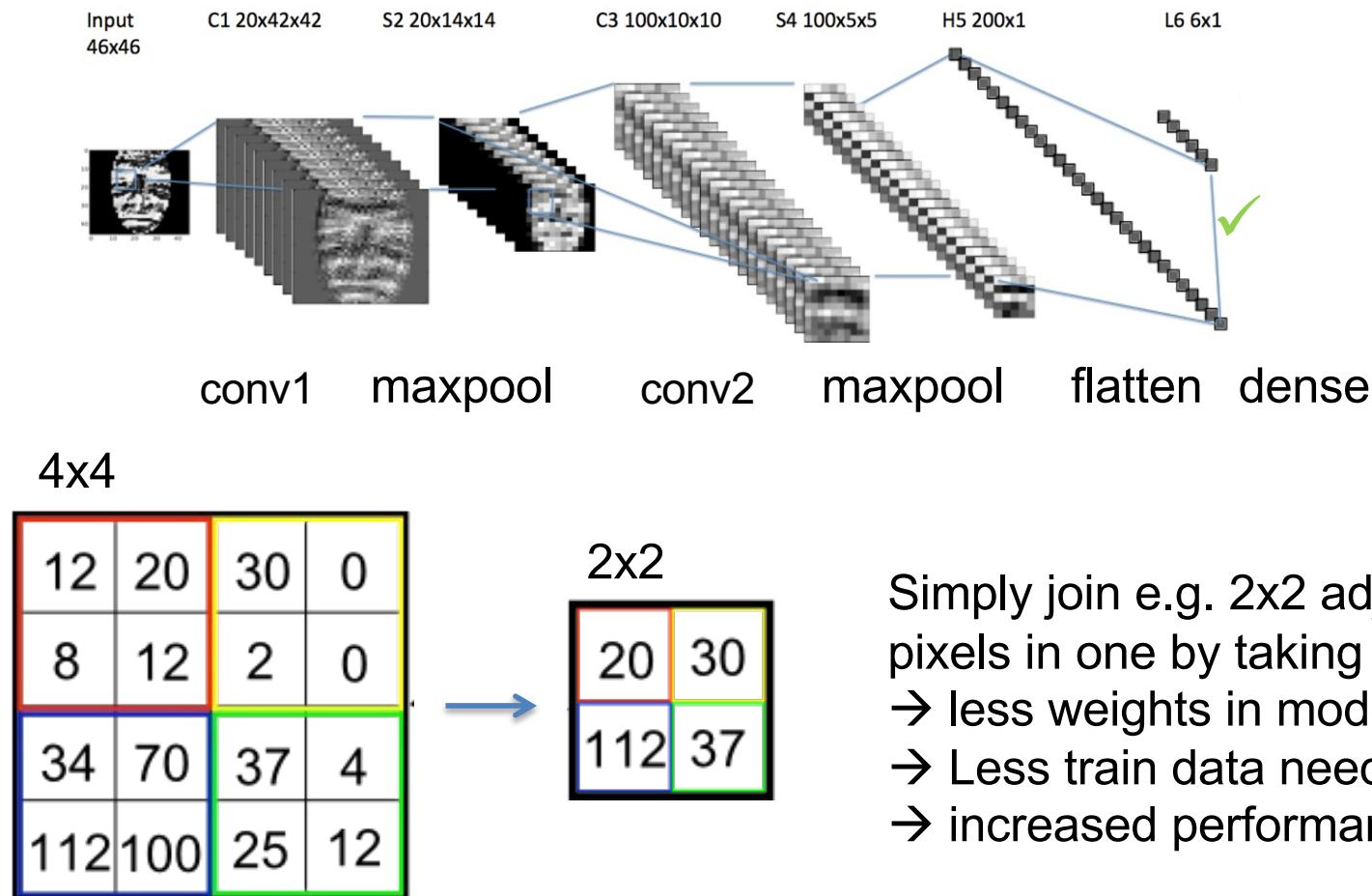
Zero-padding to achieve  
**same** size of feature and input



no padding to only use  
**valid** input information

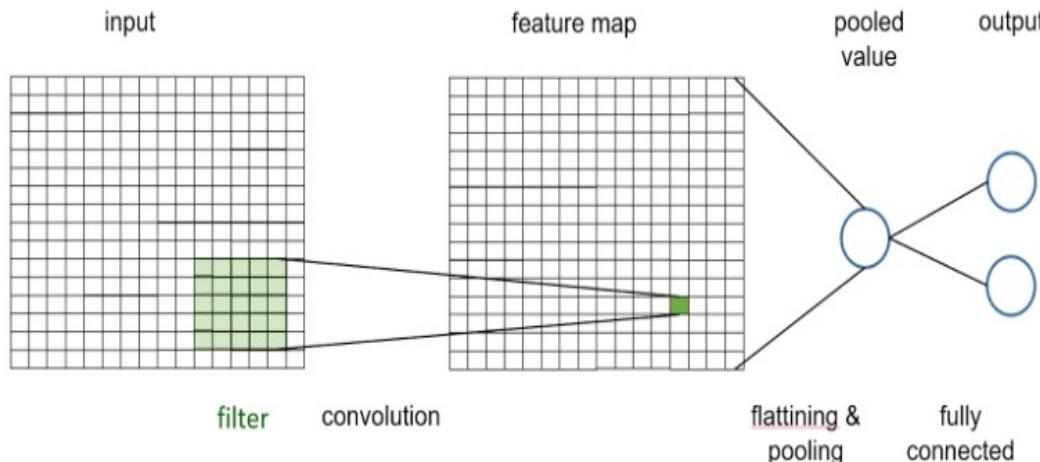
The *same* weights are used at each position of the input image.

## CNN ingredient II: Maxpooling Building Blocks reduce size



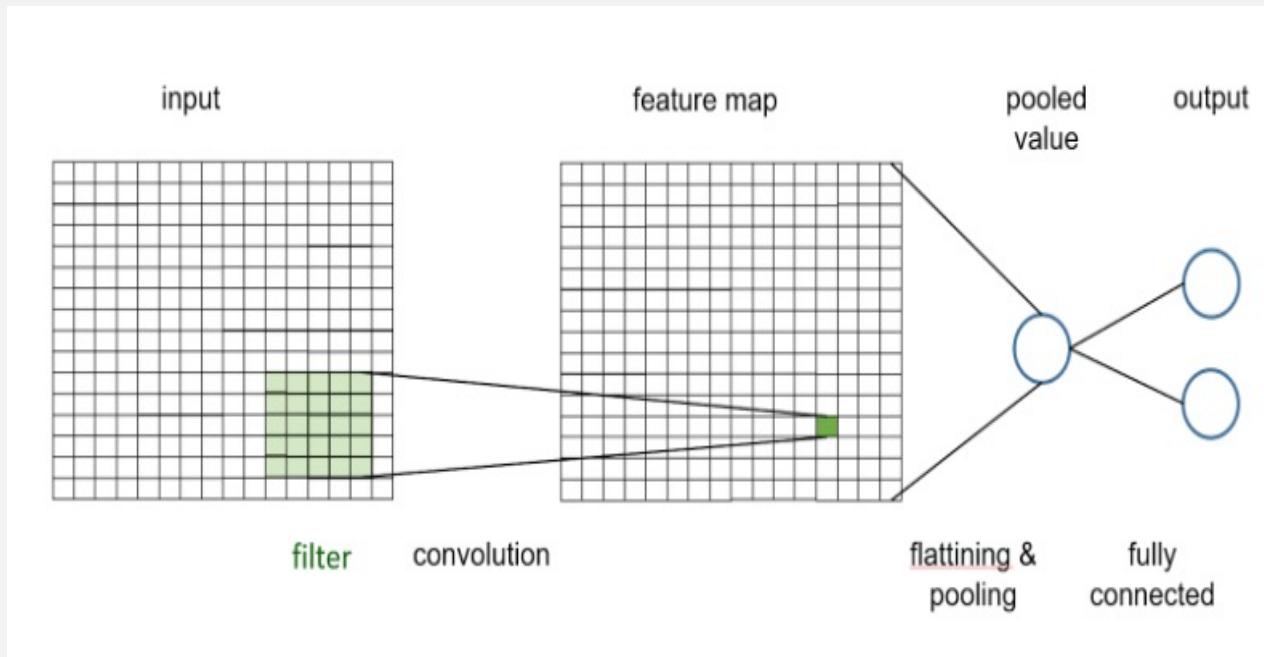
Hinton: „The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster“

# Building a very simple CNN with keras



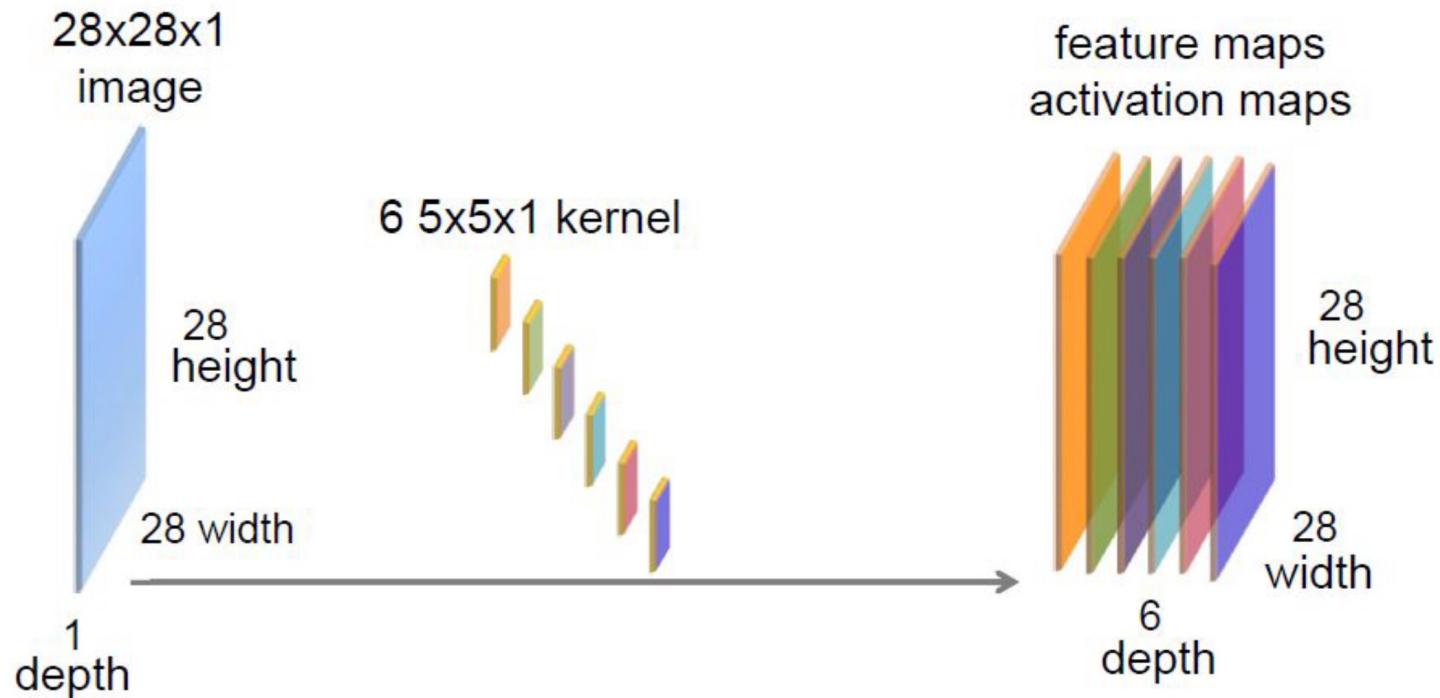
```
model = Sequential()
model.add(Convolution2D(1, (5,5), # one 5x5 kernel
                      padding='same',           # zero-padding to preserve size
                      input_shape=(pixel,pixel,1)))
model.add(Activation('linear'))
# take the max over all values in the activation map
model.add(MaxPooling2D(pool_size=(pixel,pixel)))
model.add(Flatten())
model.add(Dense(2))
model.add(Activation('softmax'))
```

## Exercise: Artstyle Lover



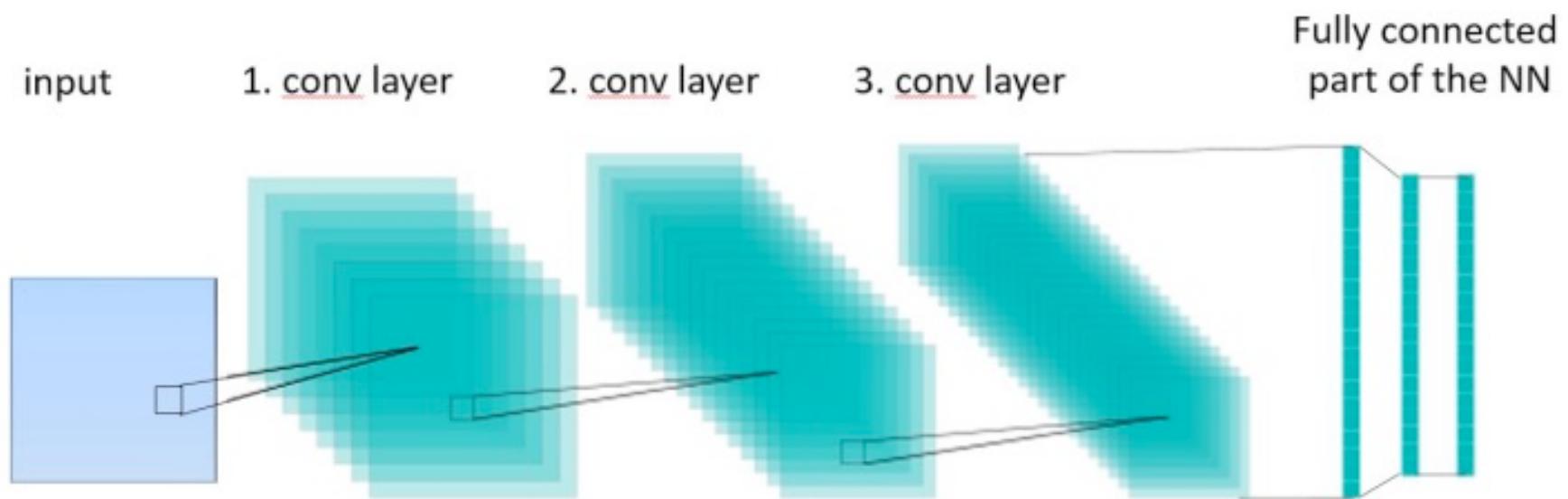
Open NB in: [https://github.com/tensorchiefs/dl\\_course\\_2024/blob/master/notebooks/05\\_cnn\\_edge\\_lover.ipynb](https://github.com/tensorchiefs/dl_course_2024/blob/master/notebooks/05_cnn_edge_lover.ipynb)

## Convolution layer with a 1-channel input and 6 kernels

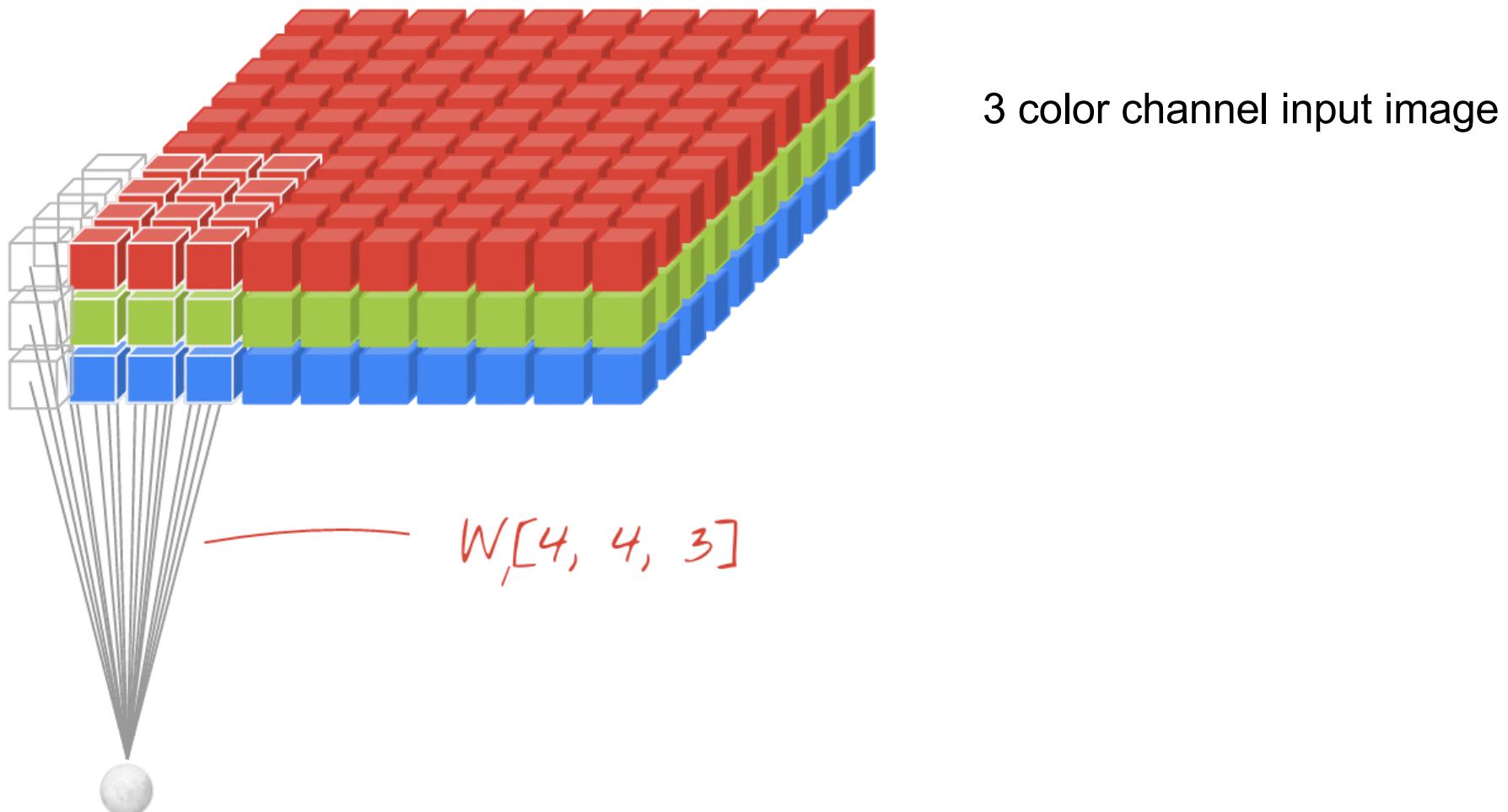


Convolution of the input image with 6 different kernels results in 6 activation maps.  
If the input image has only one channel, then each kernel has also only one channel.

# A CNN with 3 convolution layers



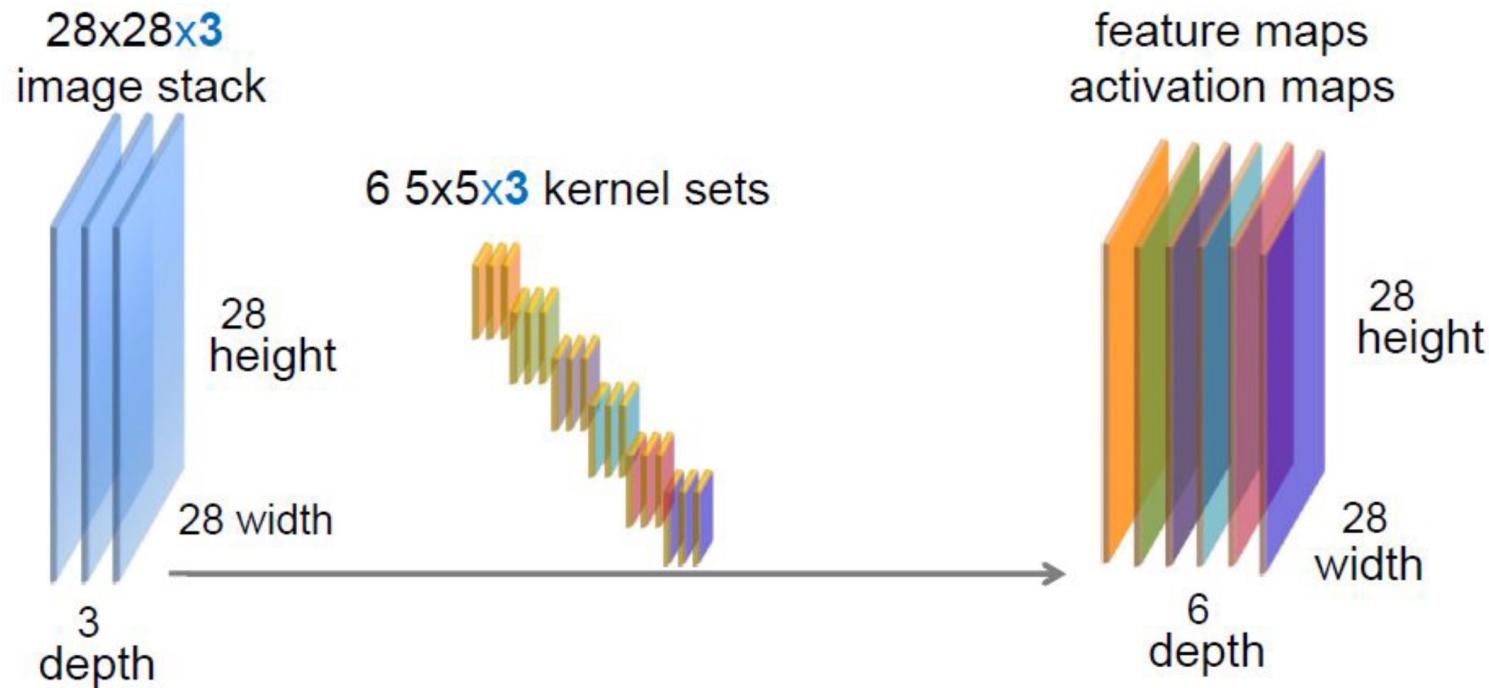
# Animated convolution with 3 input channels



The value of neuron  $j$  in the  $k$ -th featuremap are computed from the weights in the  $k$ -th filter  $w_{ki}$  and the input values  $x_{ji}$  at the position  $j$ :

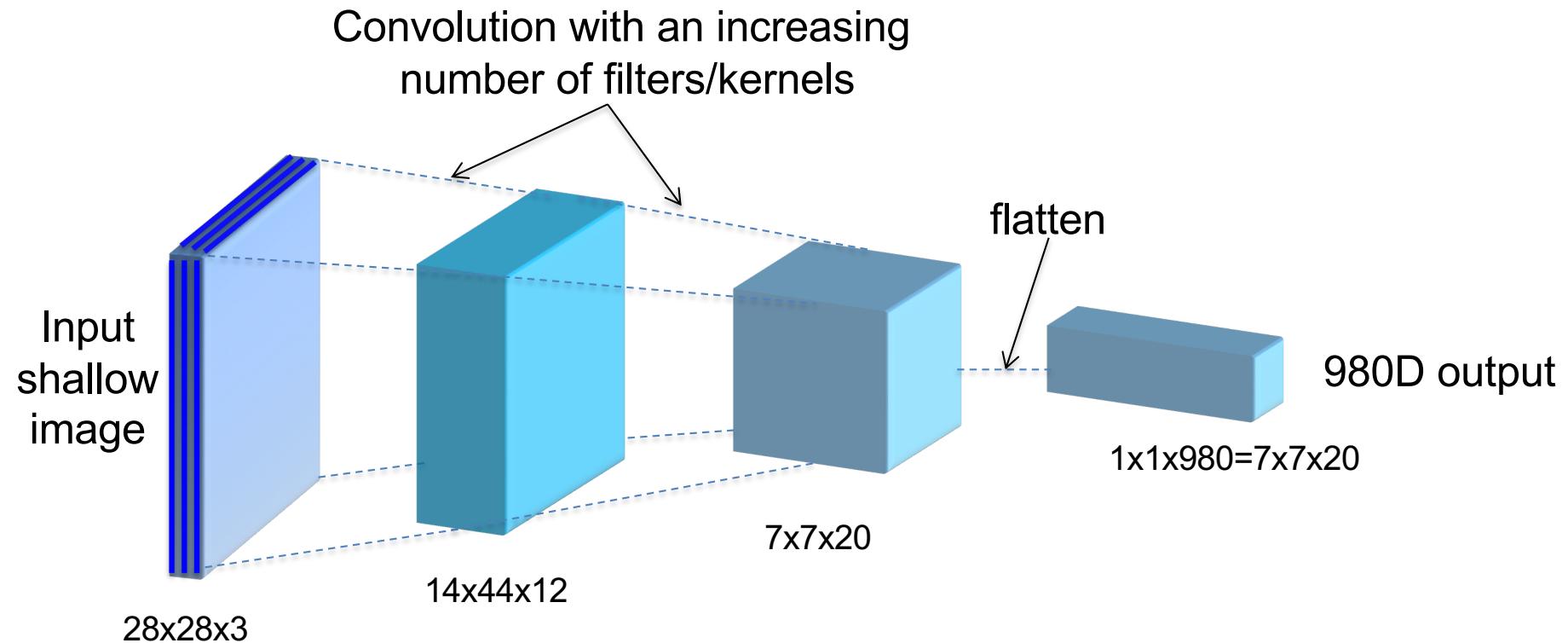
$$y_{jF_k} = f(z_{jF_k}) = f(b_k + \sum x_{ji} \cdot w_{ki})$$

## Convolution layer with a 3-channel input and 6 kernels



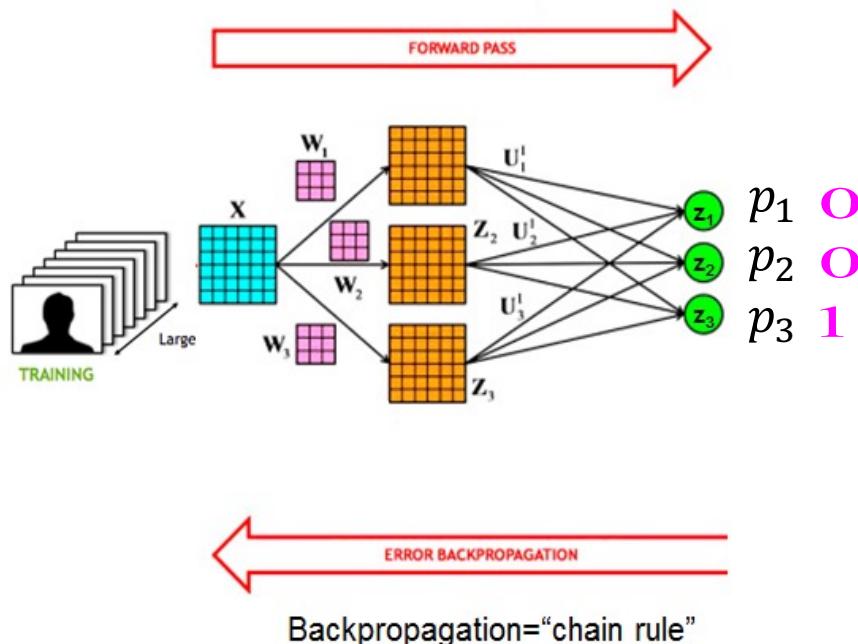
Convolution of the input image with 6 different kernels results in 6 activation maps.  
If the input image has 3 channels, then each filter has also 3 channels.

# Typical shape of a classical CNN



Spatial resolution is decreased e.g. via max-pooling while more abstract image features are detected in deeper layers.

# Training of a CNN is based on gradient backpropagation



Learning is done by weight updating:

For the training we need the observed label for each image which we then compare with the output of the CNN.

We want to adjust the weights in a way so that difference between true label and output is minimal.

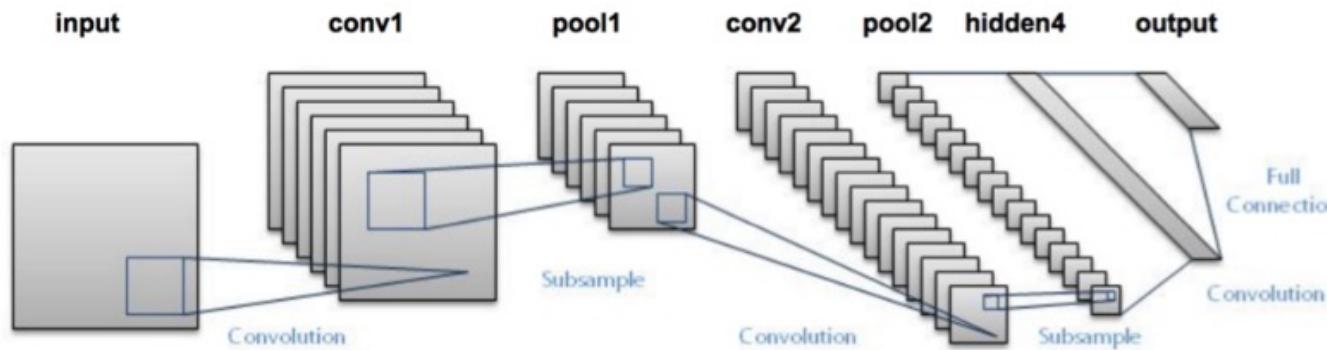
Minimize Loss-function:

$$L = \text{distance}(\text{observed}, \text{output}(w))$$

$$w_i^{(t)} = w_i^{(t-1)} - l^{(t)} \left. \frac{\partial L(w)}{\partial w_i} \right|_{w_i=w_i^{(t-1)}}$$

↑  
learning rate

# CNN for MNIST



```
num_classes = 10                      # 10 classes: 0, 1,...,9
input_shape = (28, 28, 1)    # 28x28 pixels, 1 channel (grey value)

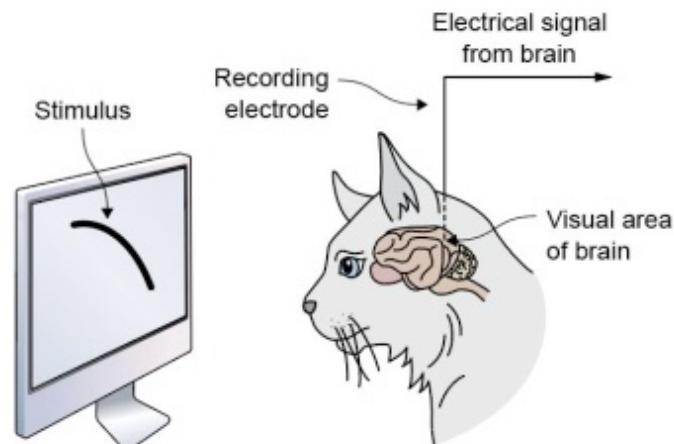
model = Sequential()
model.add(Convolution2D(32, (3, 3), #32 filters of size 3x3
                      activation='relu',
                      input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Convolution2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(40, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

## fcNN versus CNNs - some aspects

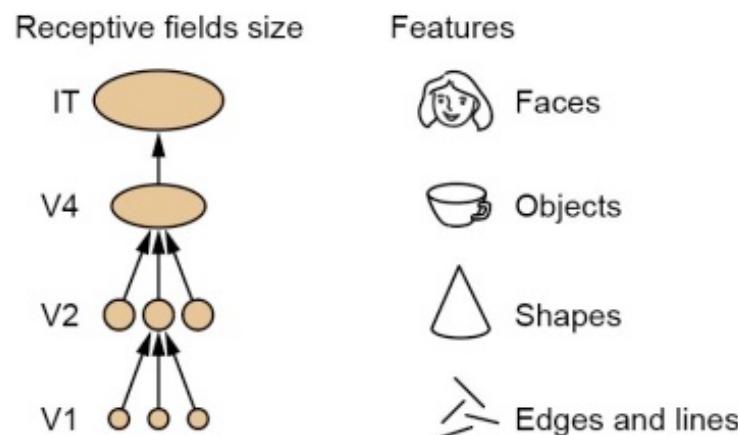
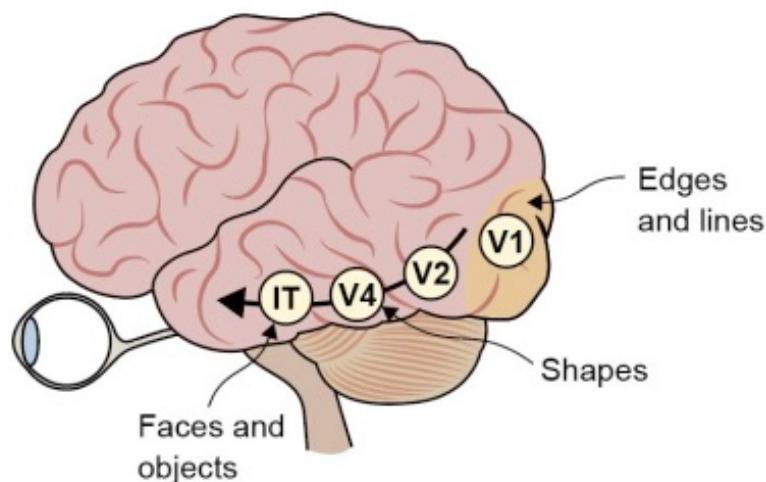
- A fcNN is good for tabular data, CNNs are good for 2D data (eg images).
- In a fcNN the order of the input does not matter, in CNN shuffling matters.
- A fcNN has no model (inductive) bias, a CNN has the model bias that neighborhood matters.
- A node in one layer of a fcNN corresponds to one feature map in a convolution layer:
- In each layer of a fcNN connecting p to q nodes, we learn q linear combinations of the incoming p signals, in each layer of a CNN connecting p channels with q channels we learn q filters (each having p channels) yielding q feature maps

# Biological Inspiration of CNNs

# How does the brain respond to visually received stimuli?

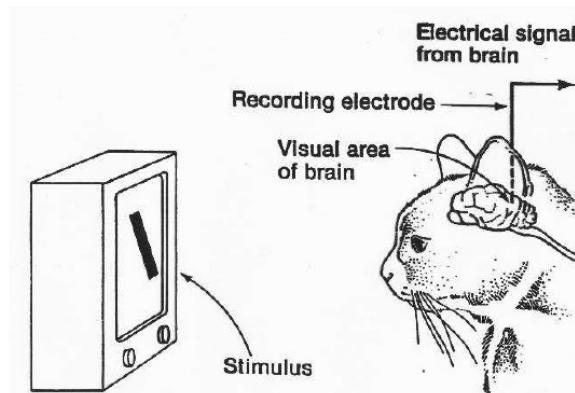


Setup of the experiment of Hubel and Wiesel in late 1950s in which they discovered **neurons** in the visual cortex that **responded** when moving **edges** were shown to the cat.

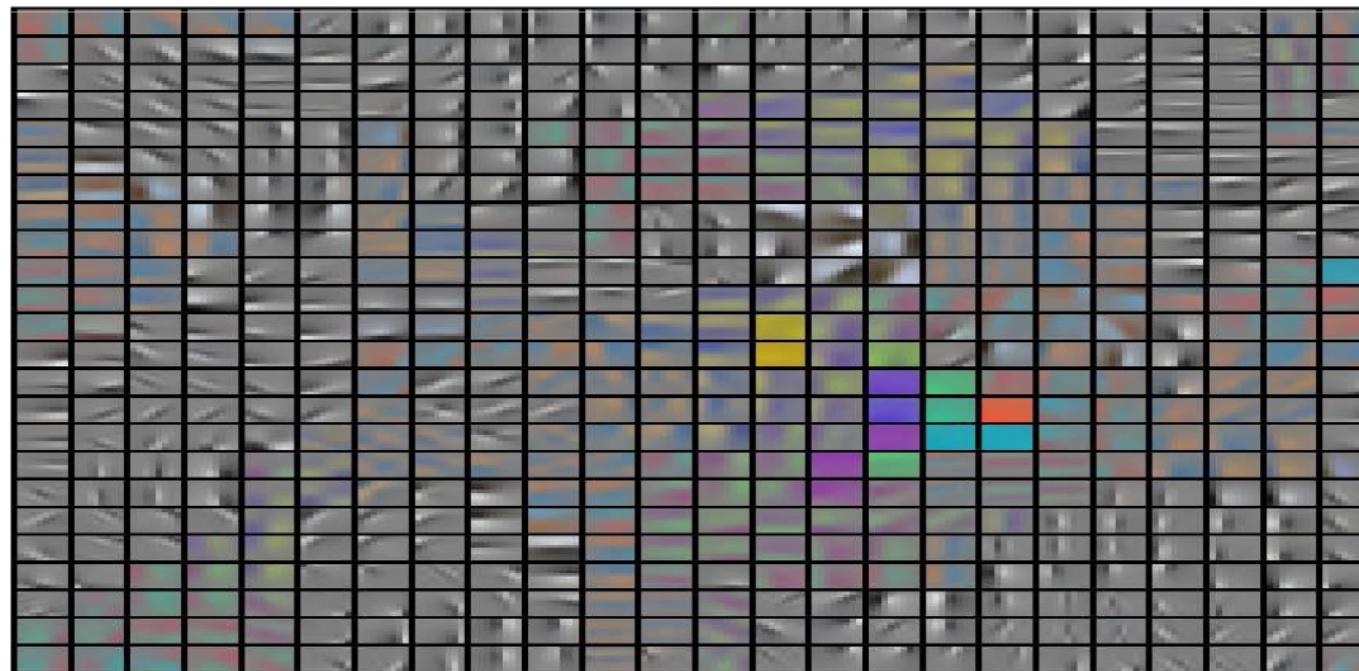
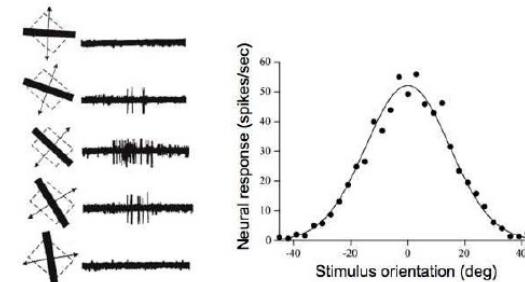


Organization of the visual cortex in a brain, where neurons in different regions respond to more and more complex stimuli

# Compare neurons in brain region V1 in first layer of a CNN



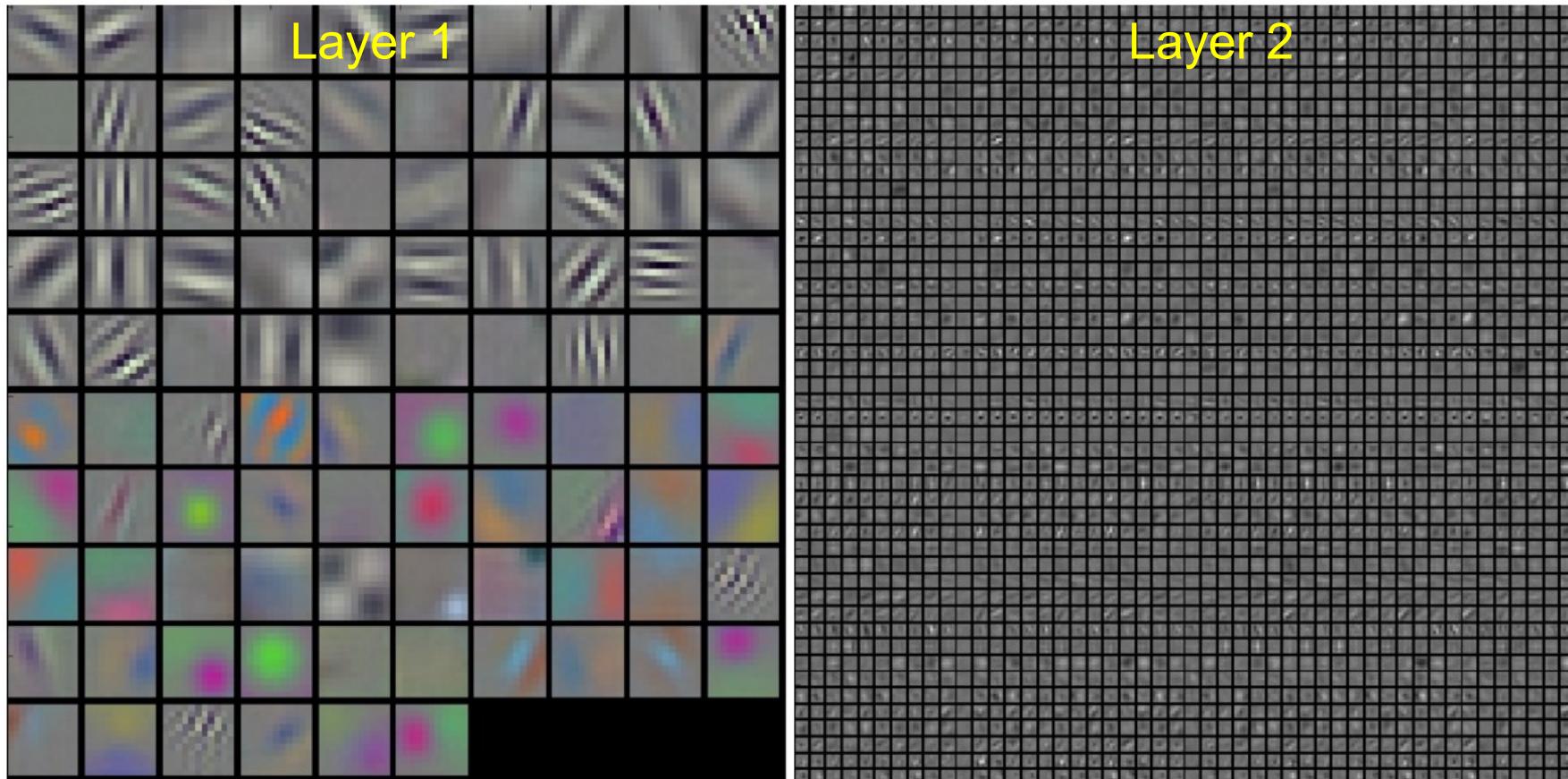
V1 physiology: orientation selectivity



Neurons in brain region V1 and neurons in 1. layer of a CNN respond to similar patterns

# Visualize the weights used in filters

Filter weights from a trained Alex Net

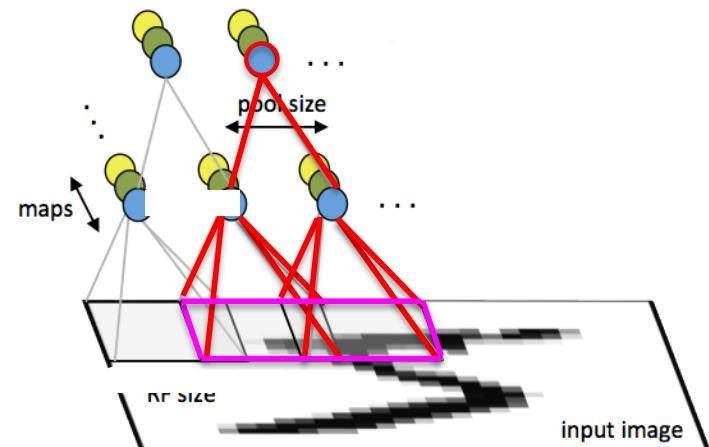
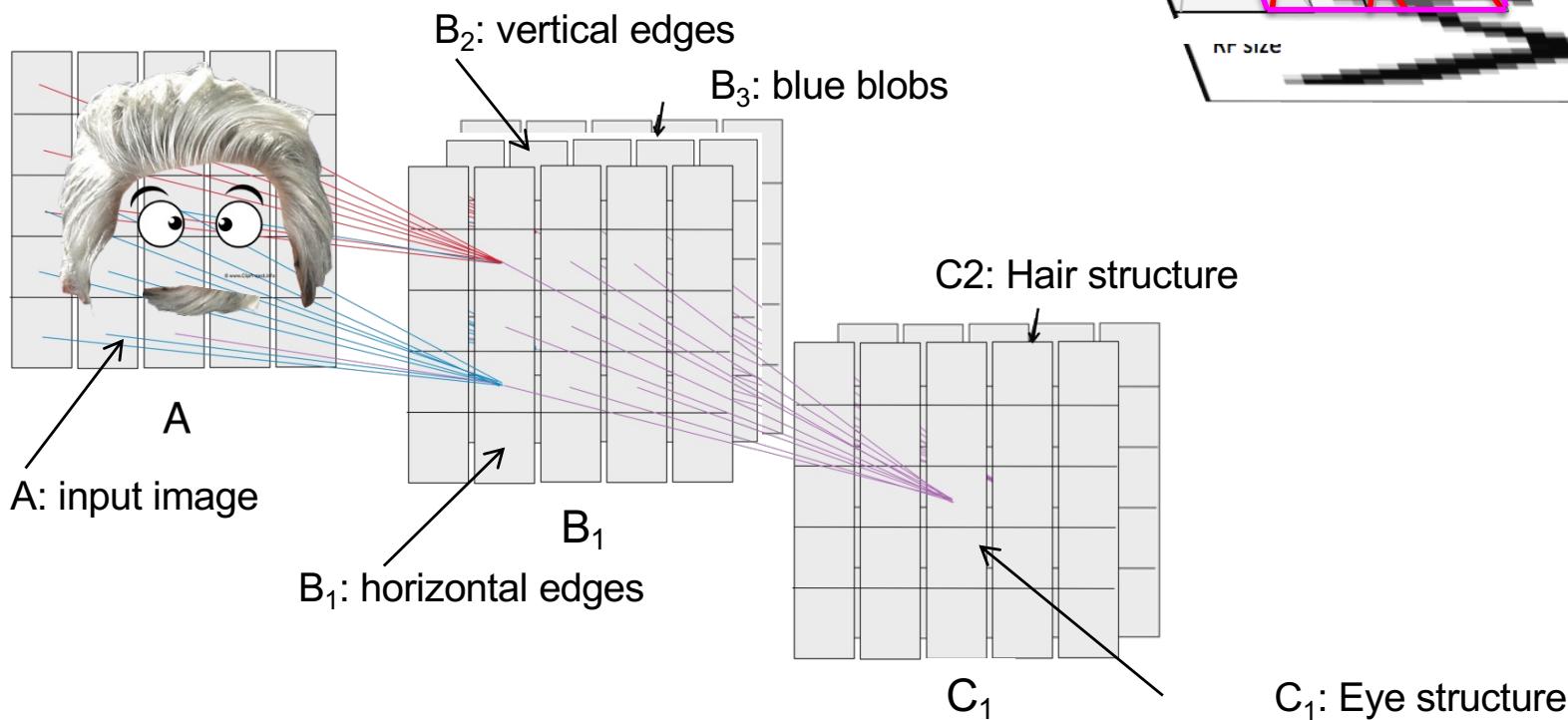


Only in layer 1 the filter pattern correspond to extracted patterns in the image.

In higher layers we can only check if patterns look noisy, which would indicate that the network that hasn't been trained for long enough, or possibly with a too low regularization strength that may have led to overfitting.

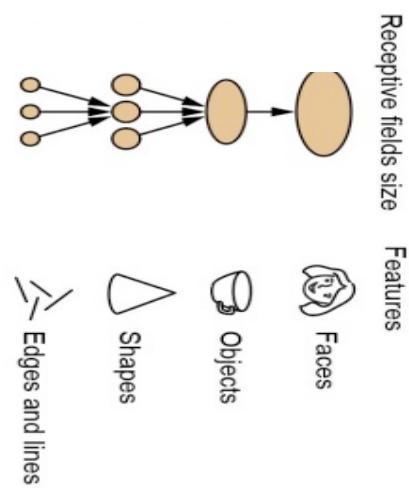
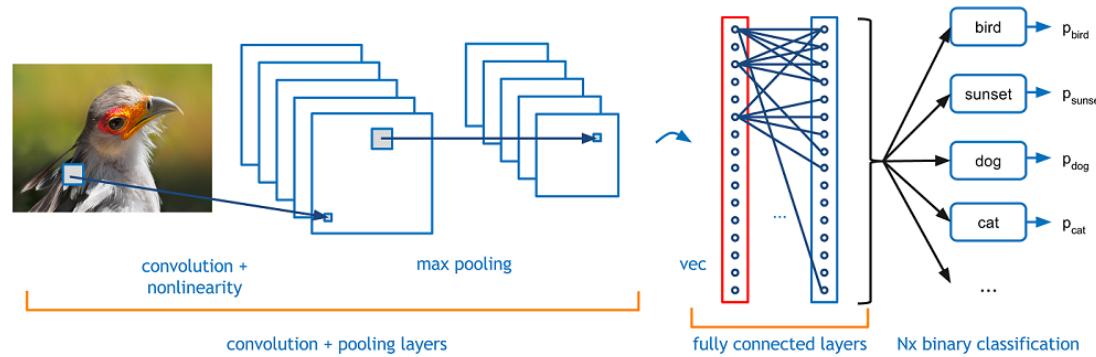
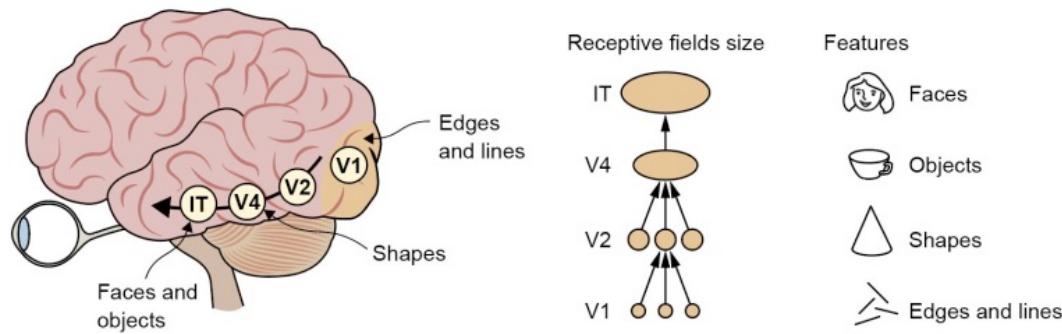
# The receptive field

For each pixel of a feature map we can determine the connected area in the input image – this area in the input image is called receptive field.



A feature map gets activated by a certain structure of the feature maps one layer below, which by itself depends on the input of a preceding layer etc and finally on the input image. Activation maps close to the input image are activated by simple structures in the image, higher maps by more complex image structures. 47

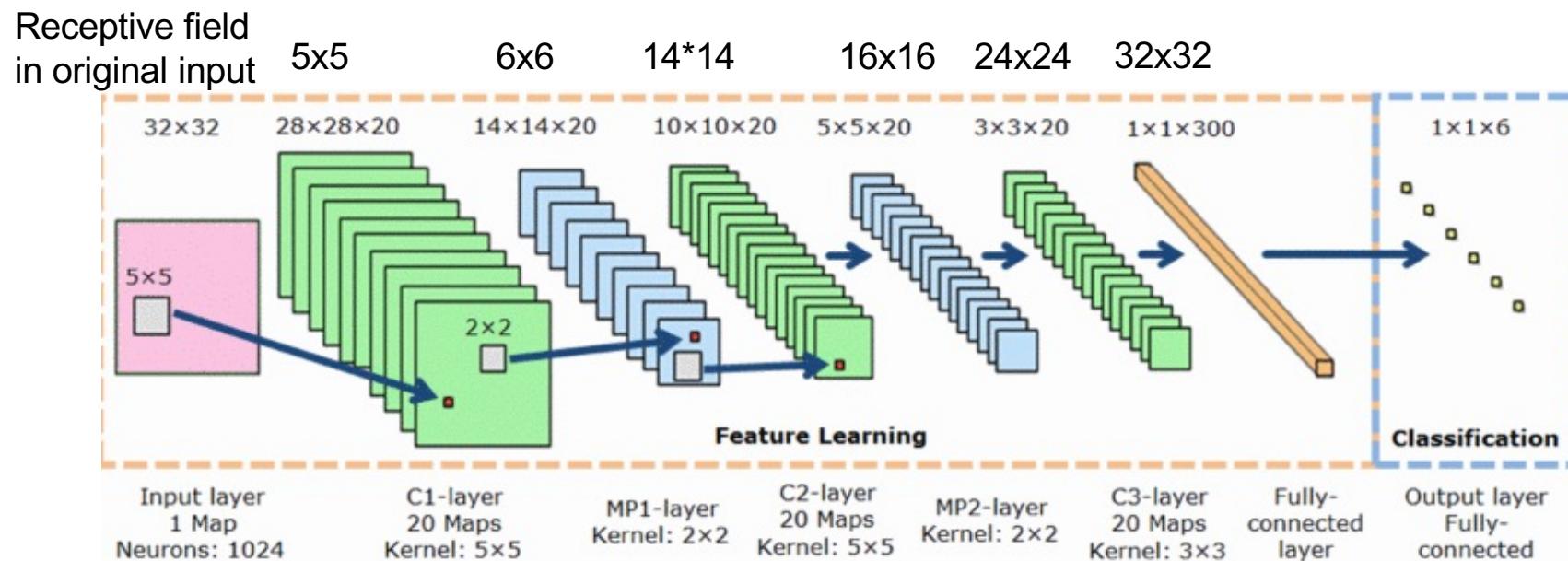
# Weak analogies between brain and CNNs architecture



What input does activate a feature map in the CNN part or a neuron in the last layer?

# The receptive field is growing from layer to layer

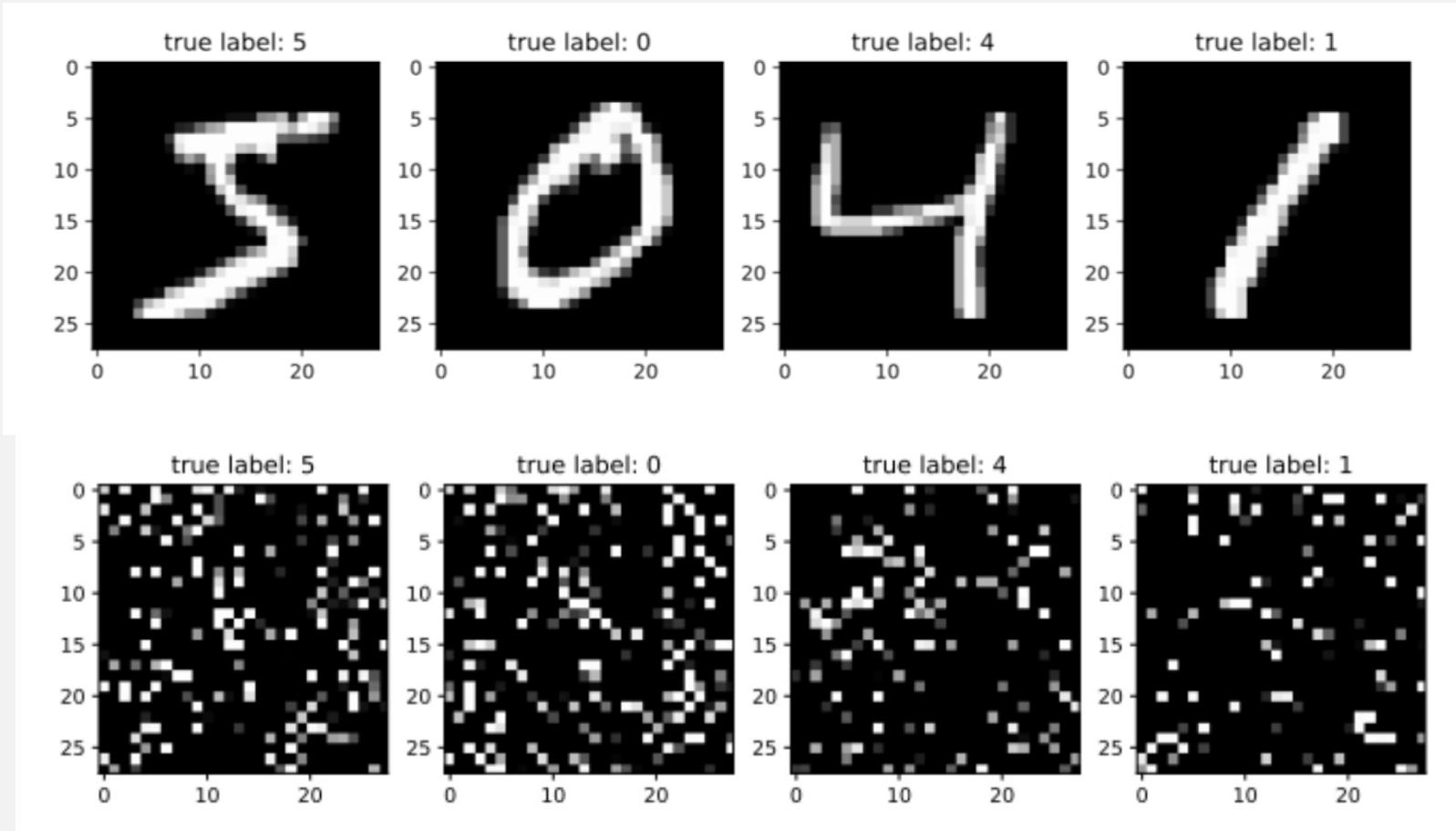
The receptive field of a neuron is the area in the original input image that impact the value of this neuron – “that can be seen by this neuron”.



Neurons from feature maps in higher layers have a larger receptive field than neurons sitting in feature maps closer to the input.

Code to determine size of receptive field: <http://stackoverflow.com/questions/35582521/how-to-calculate-receptive-field-size>

# Ufzgi: Does shuffling disturb a fcNN or CNN?



Investigate if shuffling disturbs the fcNN for MNIST:

[04\\_fcnn\\_mnist\\_shuffled.ipynb](#)

Investigate if shuffling disturbs the CNN for MNIST:

[06\\_cnn\\_mnist\\_shuffled.ipynb](#)

# Summary

- Use loss curves to detect overfitting or underfitting problems
- NNs work best when respecting the underlying structure of the data.
  - Use fully connected NN for tabular data
  - Use convolutional NN for data with local order such as images
- CNNs exploit the local structure of images by local connections and shared weight (same kernel is applied at each position of the image).
- Use the ReLu activation function for hidden layers in CNNs.
- NNs are loosely inspired by the structure of the brain.
  - When going deep the receptive field increases (~layer 5 sees whole input)
  - Deeper layer respond to more complex feature in the input