Machine Learning for Physicists

Lecture 5

Summer 2017

University of Erlangen-Nuremberg
Florian Marquardt

Handwriting recognition

"MNIST" data set (for postal code recognition)

Handwriting recognition

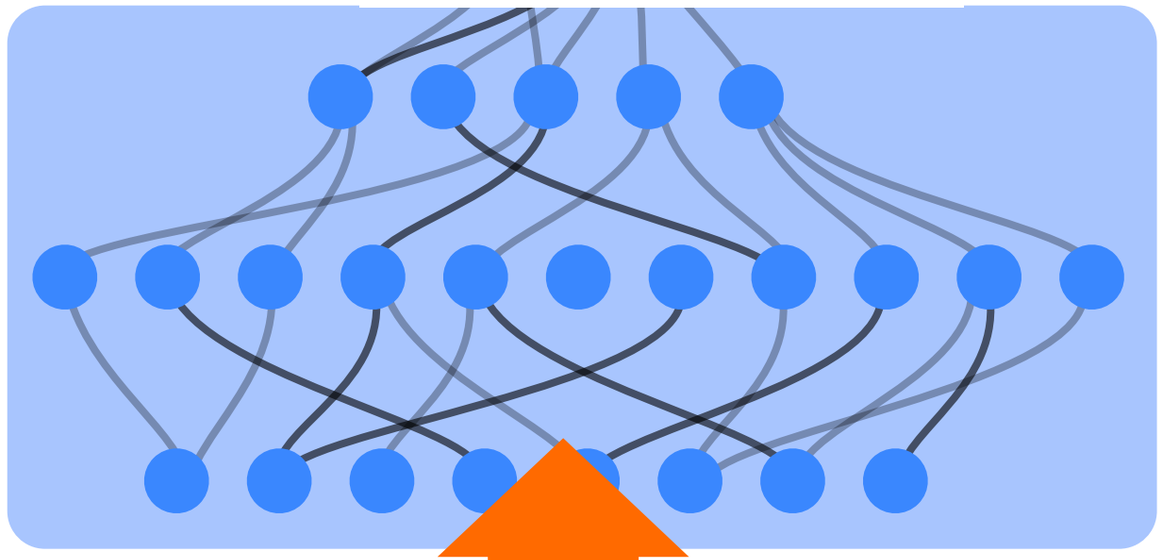
Will learn:

- distinguish categories
- "softmax" nonlinearity for probability distributions
- "categorical cross-entropy" cost function
- training/validation/test data
- "overfitting" and some solutions

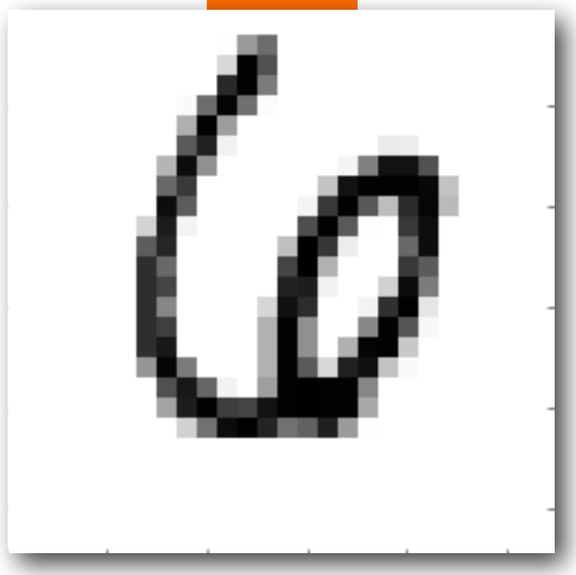
0 1 2 3 4 5 6 7 8 9

0 0 0 0 0 0 1 0 0 0

output: category
classification
“one-hot encoding”



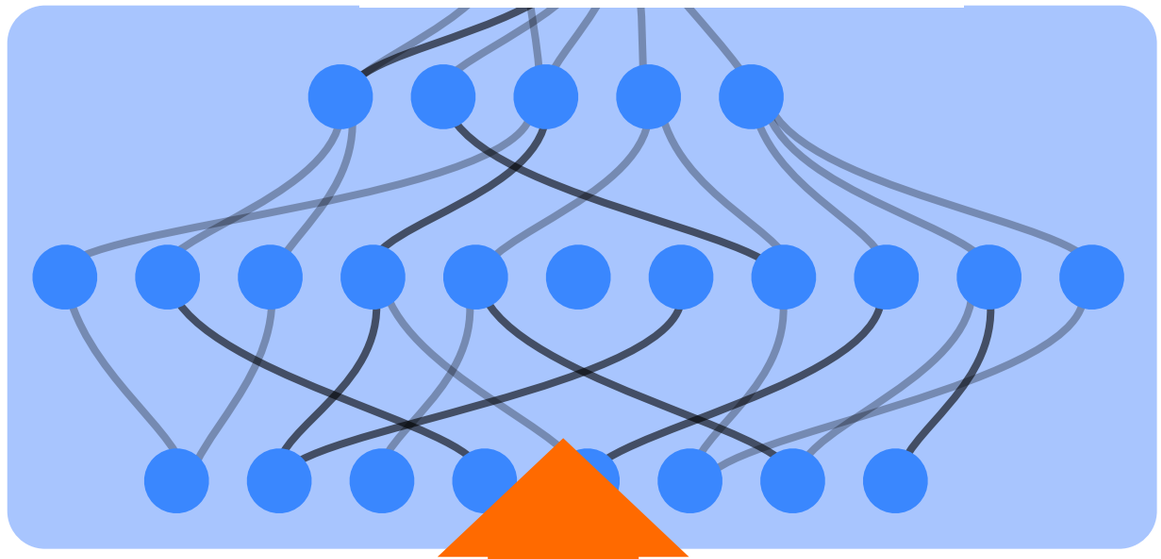
28x28 input pixels
(=784 gray values)



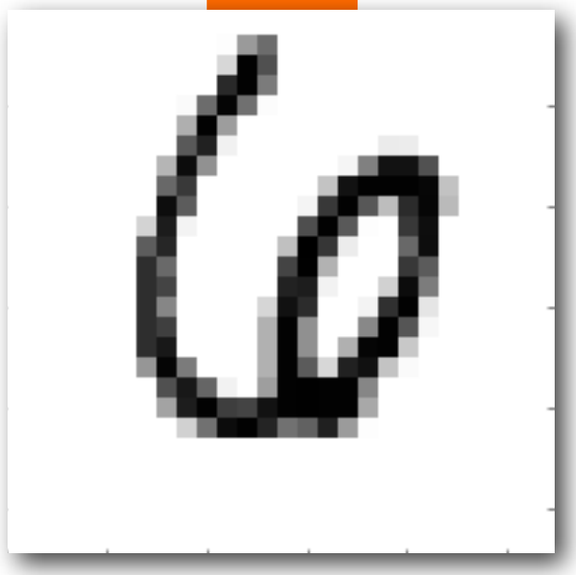
0 1 2 3 4 5 6 7 8 9

.1 0 0 0 .1 .1 .7 0 0 0

output: probabilities
(select largest)

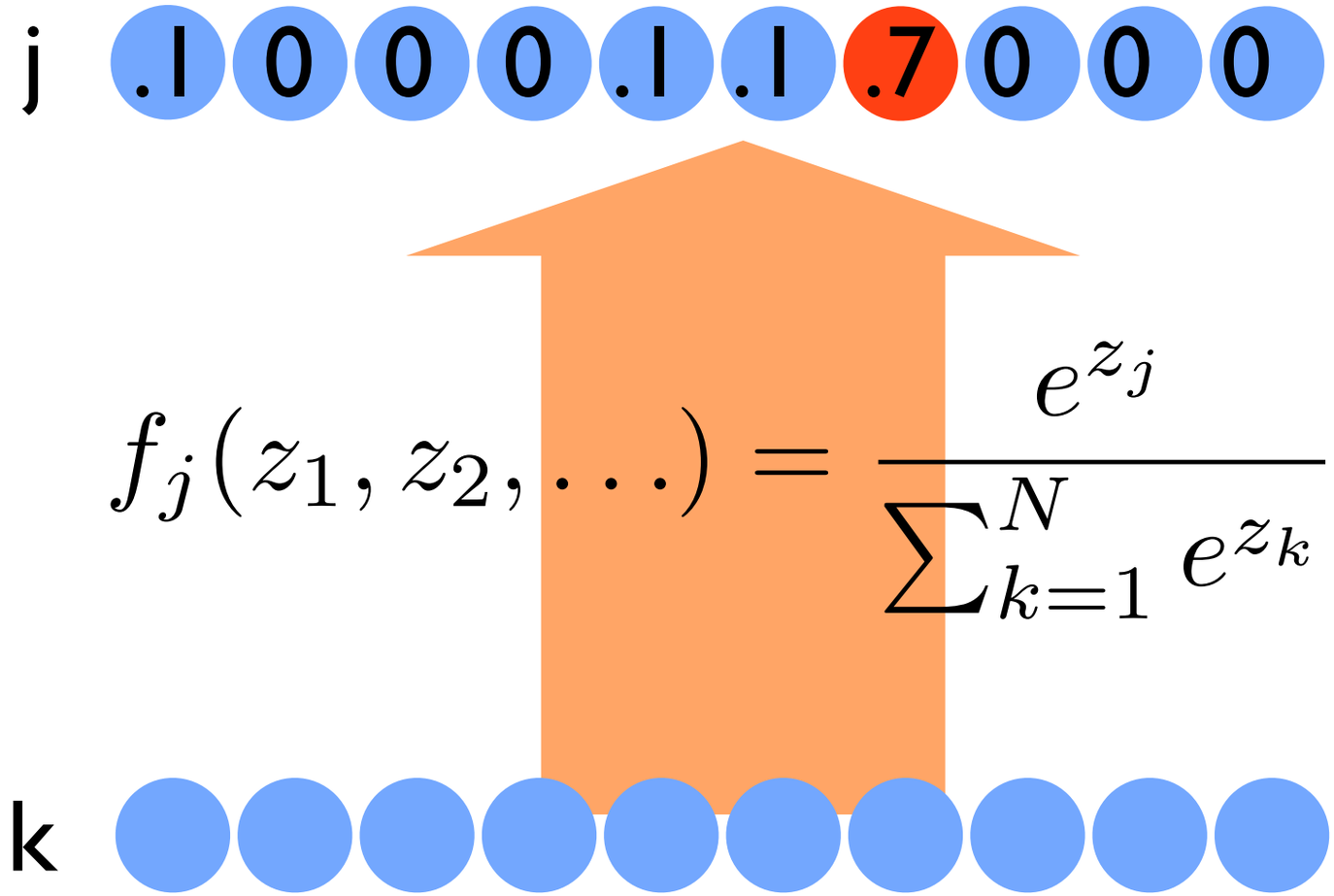


28x28 input pixels
(=784 gray values)



“Softmax” activation function

Generate normalized probability distribution,
from arbitrary vector of input values



(multi-element generalization of sigmoid)

"Softmax" activation function

$$f_j(z_1, z_2, \ldots) = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}$$

in keras:

```
net.add(Dense(10,activation='softmax'))
```

Categorical cross-entropy cost function

$$C = -\sum_j y_j^{\text{target}} \ln y_j^{\text{out}}$$

where $y_j^{\text{target}} = F_j(y^{\text{in}})$

is the desired "one-hot" classification,

in our case

Check: is non-negative and becomes zero for the correct output!

in keras:

```
net.compile(loss='categorical_crossentropy',
optimizer=optimizers.SGD(lr=1.0),
metrics=['categorical_accuracy'])
```

Categorical cross-entropy cost function

$$C = -\sum_j y_j^{\text{target}} \ln y_j^{\text{out}}$$

Advantage: Derivative does not get exponentially small for the saturated case (where one neuron value is close to 1 and the others are close to 0)

$$f_j(z_1, z_2, \ldots) = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}$$

$$\ln f_j(z) = z_j - \ln \sum_k e^{z_k}$$

$$\frac{\partial \ln f_j(z)}{\partial w} = \frac{\partial z_j}{\partial w} - \frac{\sum_k \frac{\partial z_k}{\partial w} e^{z_k}}{\sum_k e^{z_k}}$$

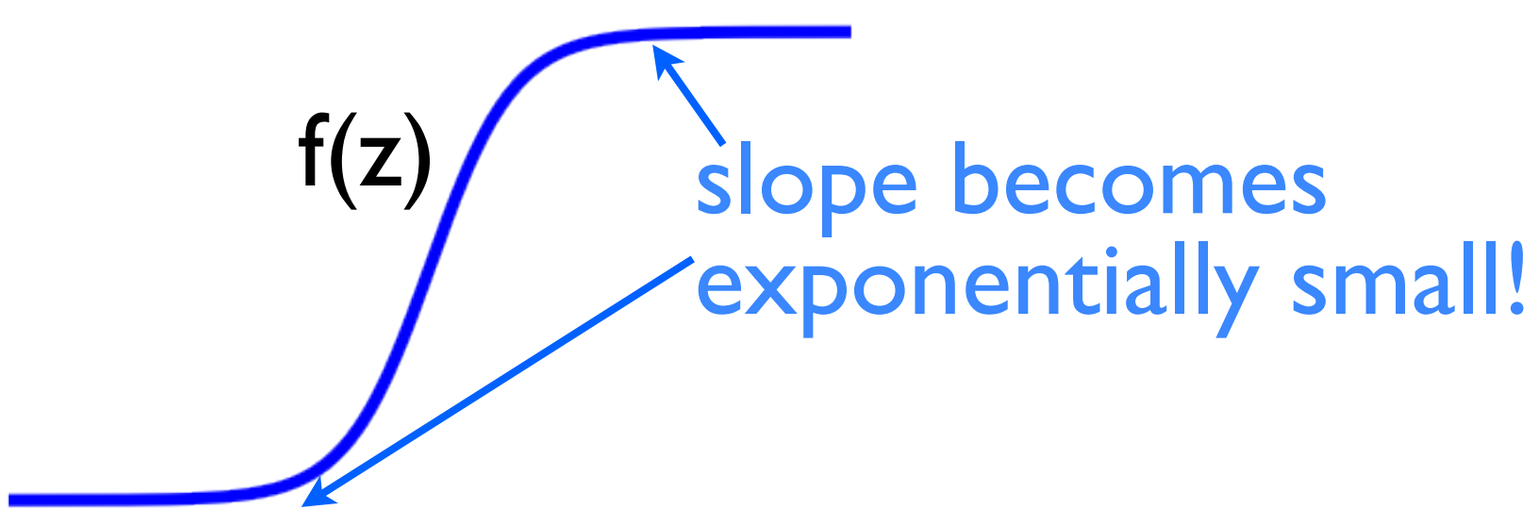
derivative of input values

Compare situation for quadratic cost function

$$f_j(z_1, z_2, \ldots) = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}$$

$$\frac{\partial}{\partial w} \sum_j (f_j(z) - y_j^{\text{target}})^2 =$$

$$= 2 \sum_j (f_j(z) - y_j^{\text{target}}) \frac{\partial f_j(z)}{\partial w}$$



Training on the MNIST images

(see code on website)

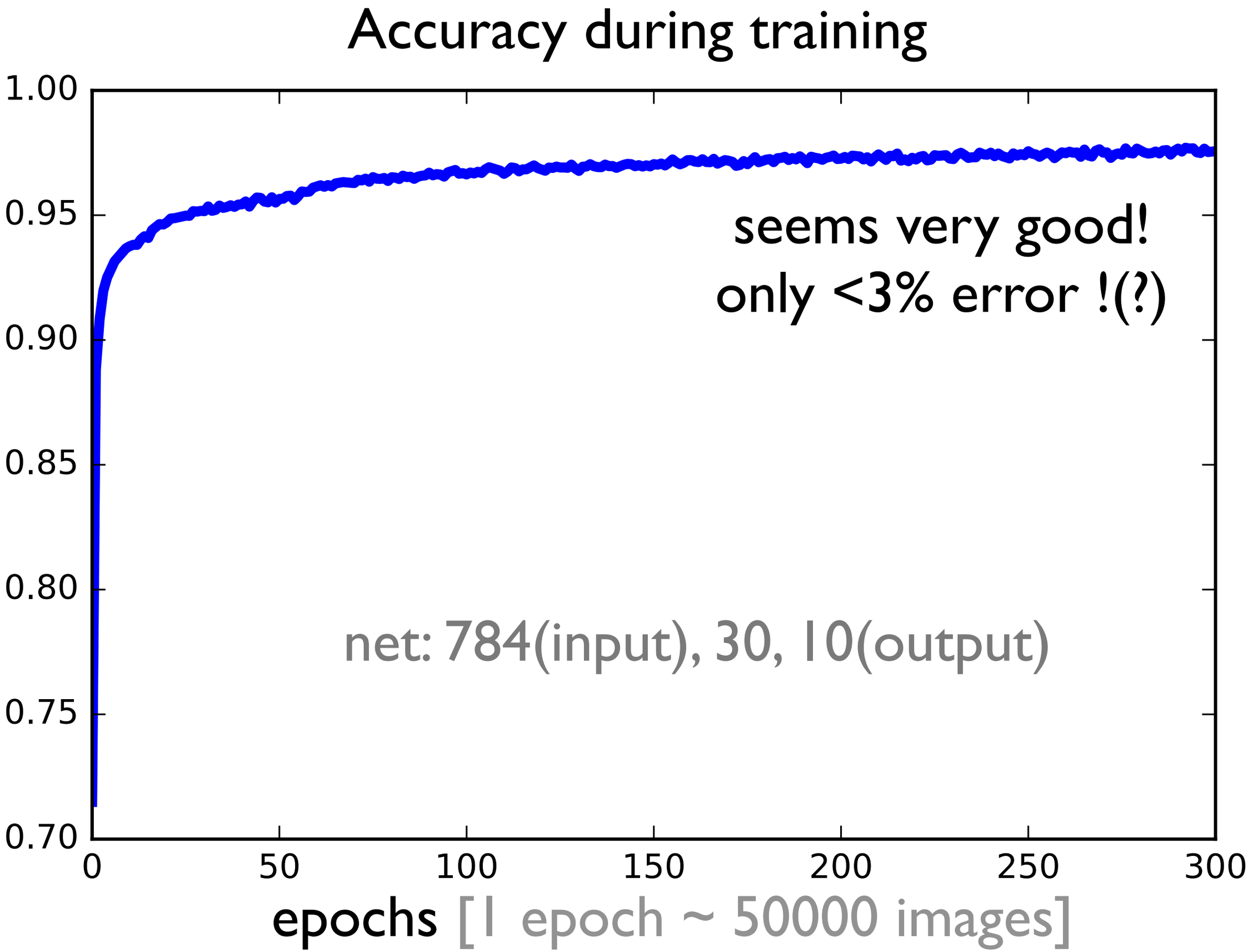
training_inputs array num_samples x numpixels

training_results array num_samples x 10
("one-hot")

in keras:

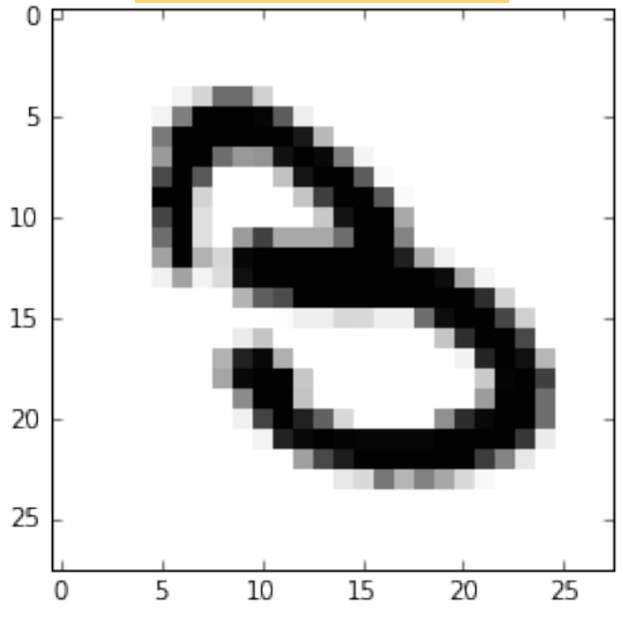
```
history=net.fit(training_inputs,
training_results,batch_size=100,epochs=30)
```

One "epoch" = training once on **all** 50000 training images, feed them into net in batches of size 100
Here: do 30 of those epochs

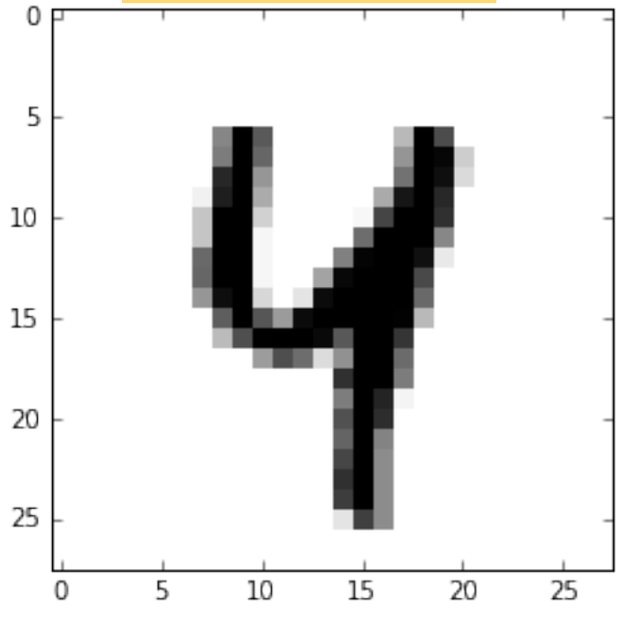

Accuracy during training
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0
50
100
150
200
250
300
seems very good!
only <3% error !(?)
net: 784(input), 30, 10(output)
epochs [1 epoch ~ 50000 images]

But: About **7 %** of the test samples are labeled incorrectly!

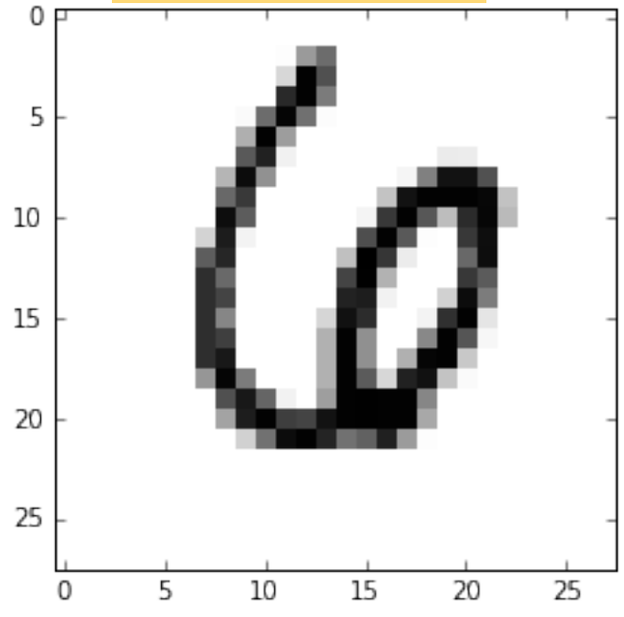
"8" (3 !)



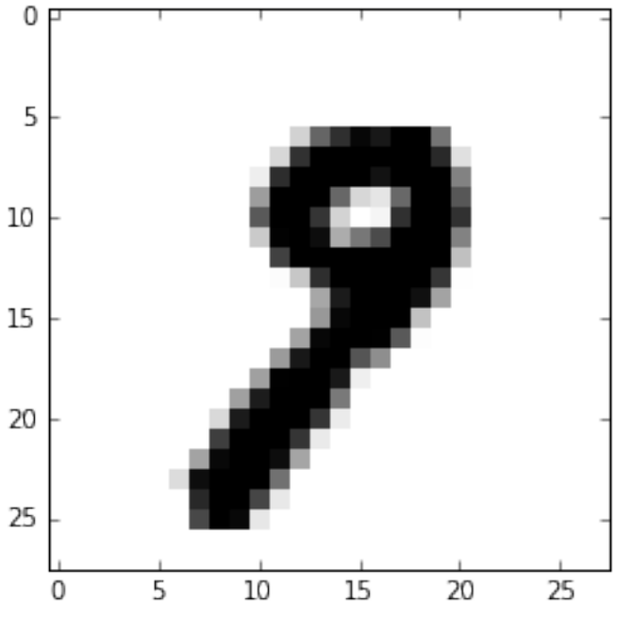
"7" (4 !)



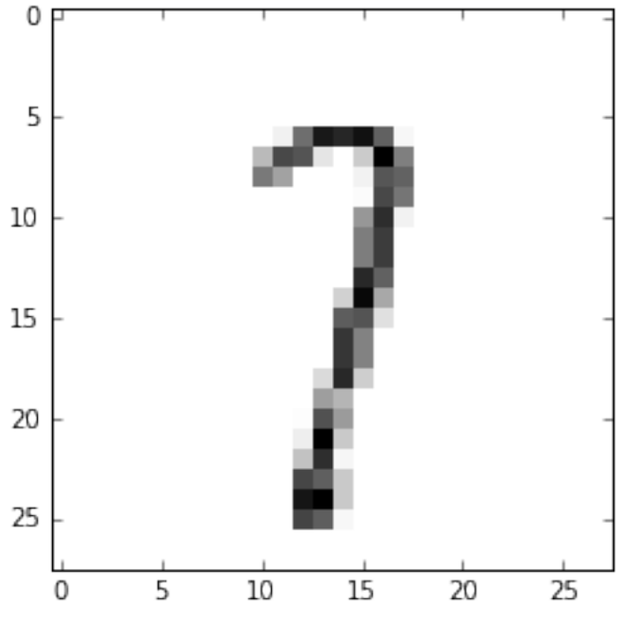
"5" (6 !)



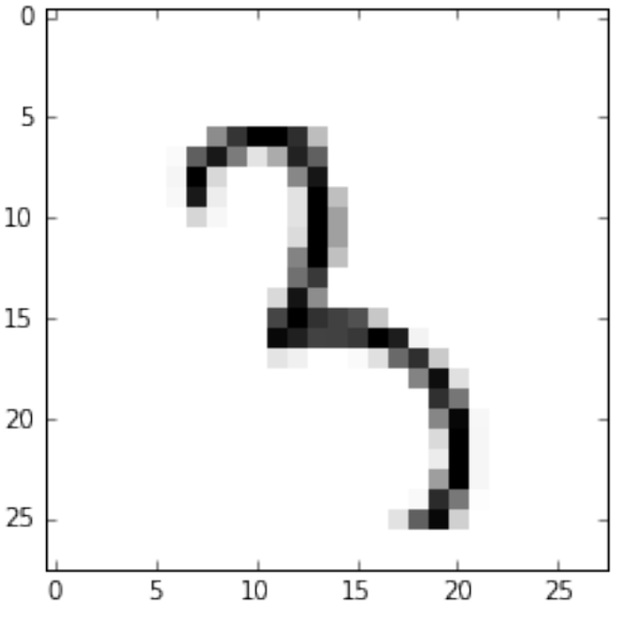
"7" (9 !)



"3" (7 !)



"5" (3 !)



Problem: assessing accuracy on the training set may yield results that are too optimistic!

Need to compare against samples which are **not** used for training! (to judge whether the net can ‘generalize’ to unseen samples)

How to honestly assess the quality during training

5000 images

Validation set

(never used for training, but used during training for assessing accuracy!)

Training set

(used for training)

45000 images

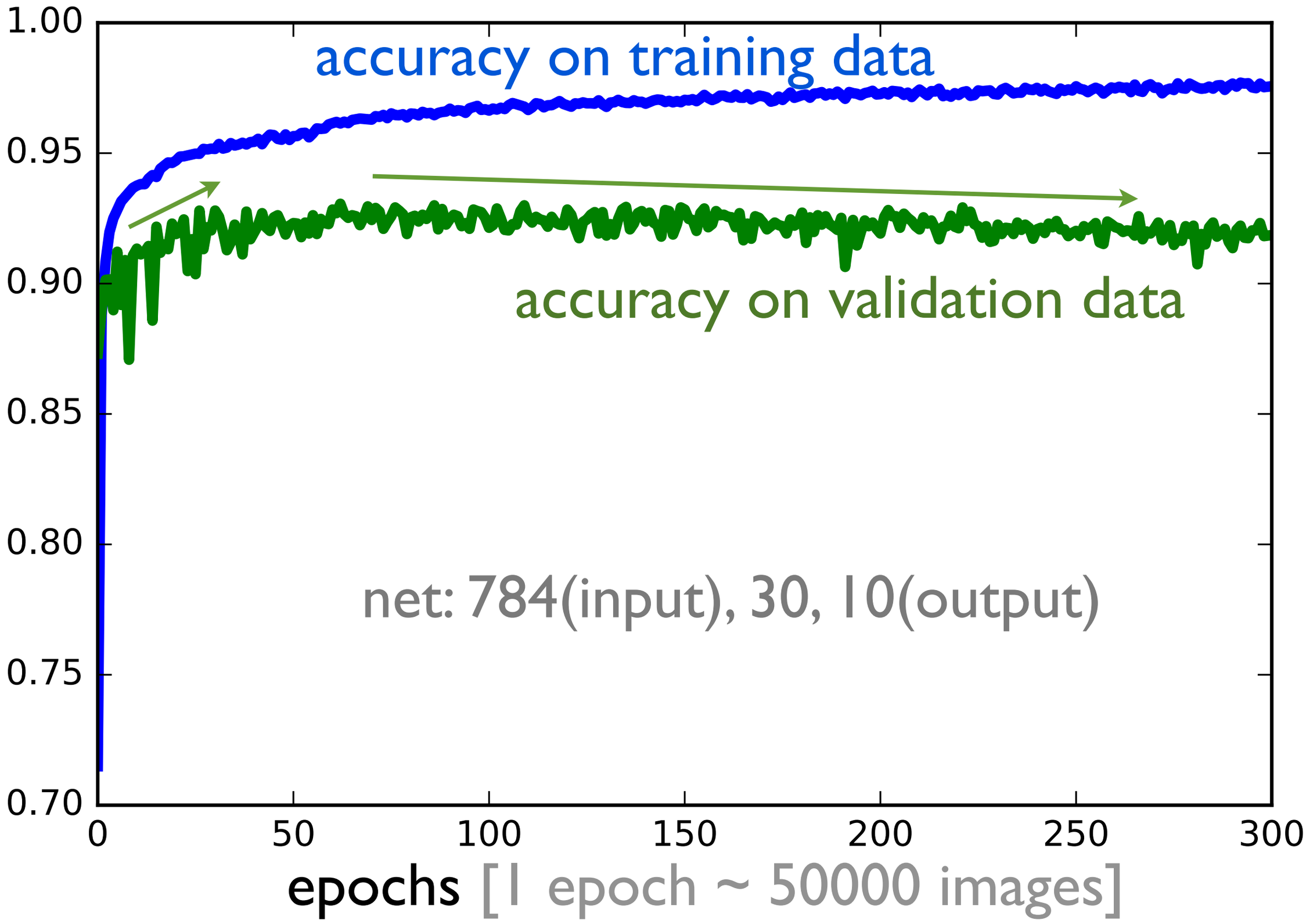
10000 images

Test set

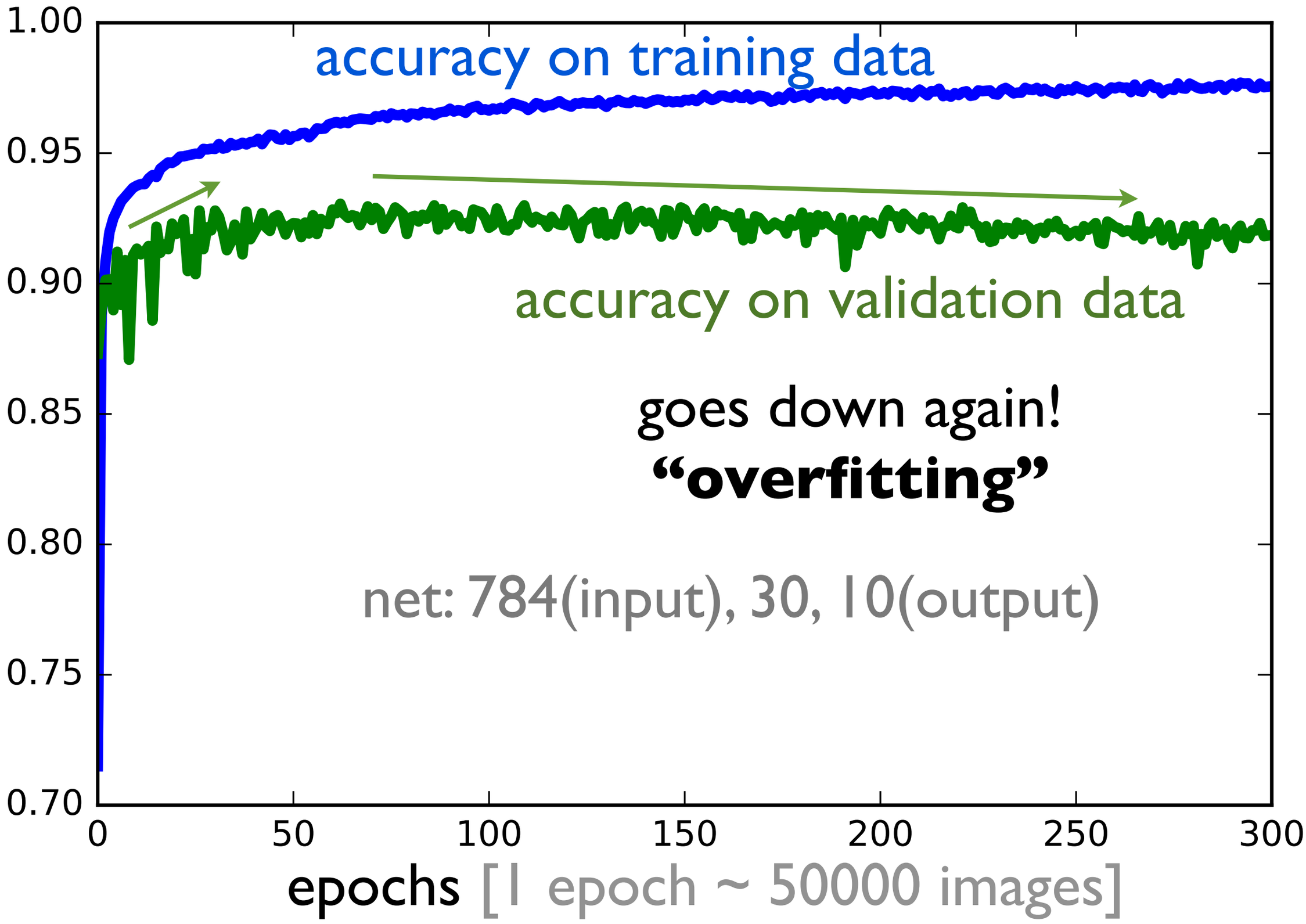
(never used during training, only later to test fully trained net)

(numbers for our MNIST example)

Accuracy during training



Accuracy during training



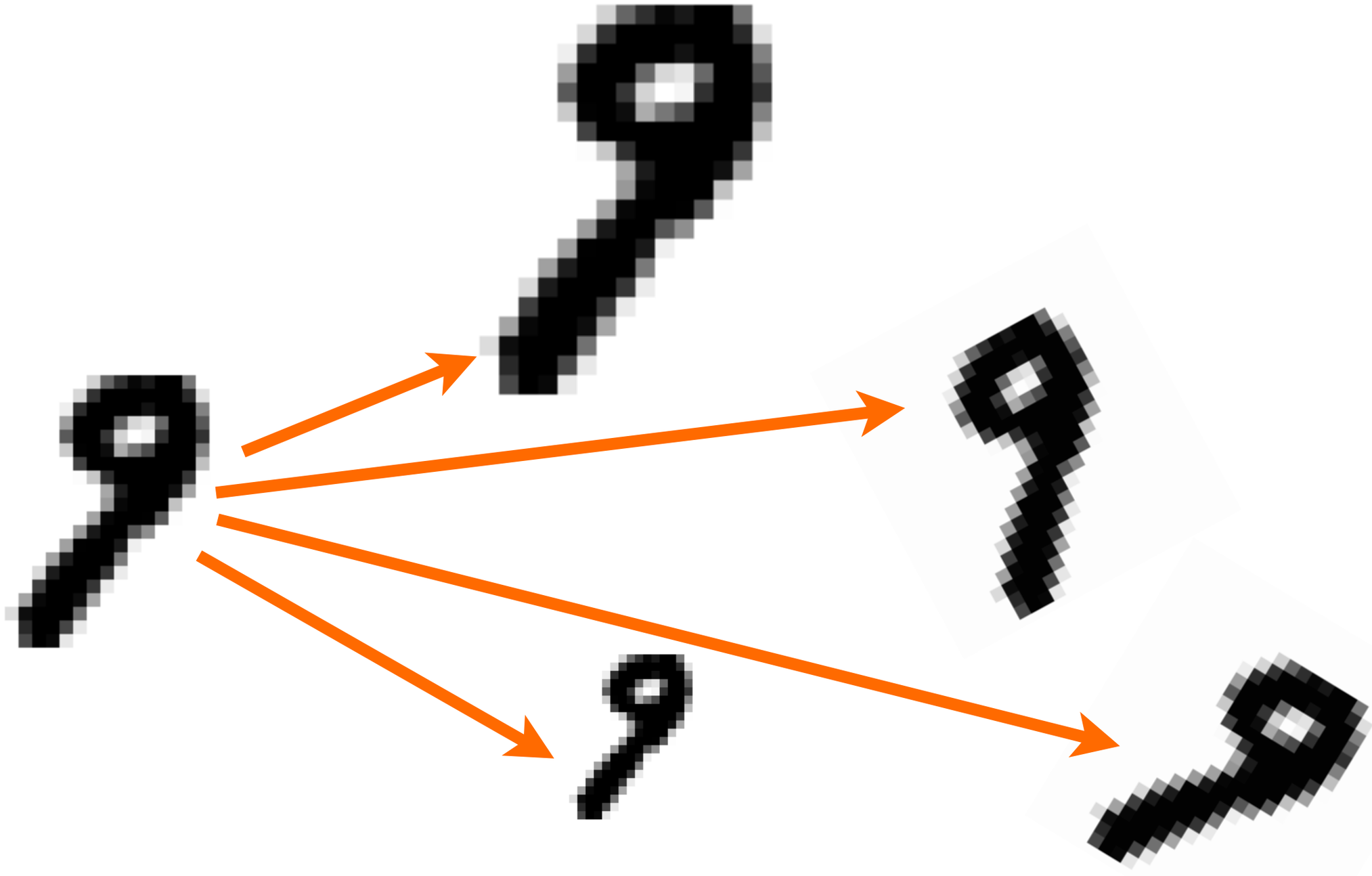
"Overfitting"

- Network "memorizes" the training samples (excellent accuracy on training samples is misleading)
- cannot generalize to unfamiliar data

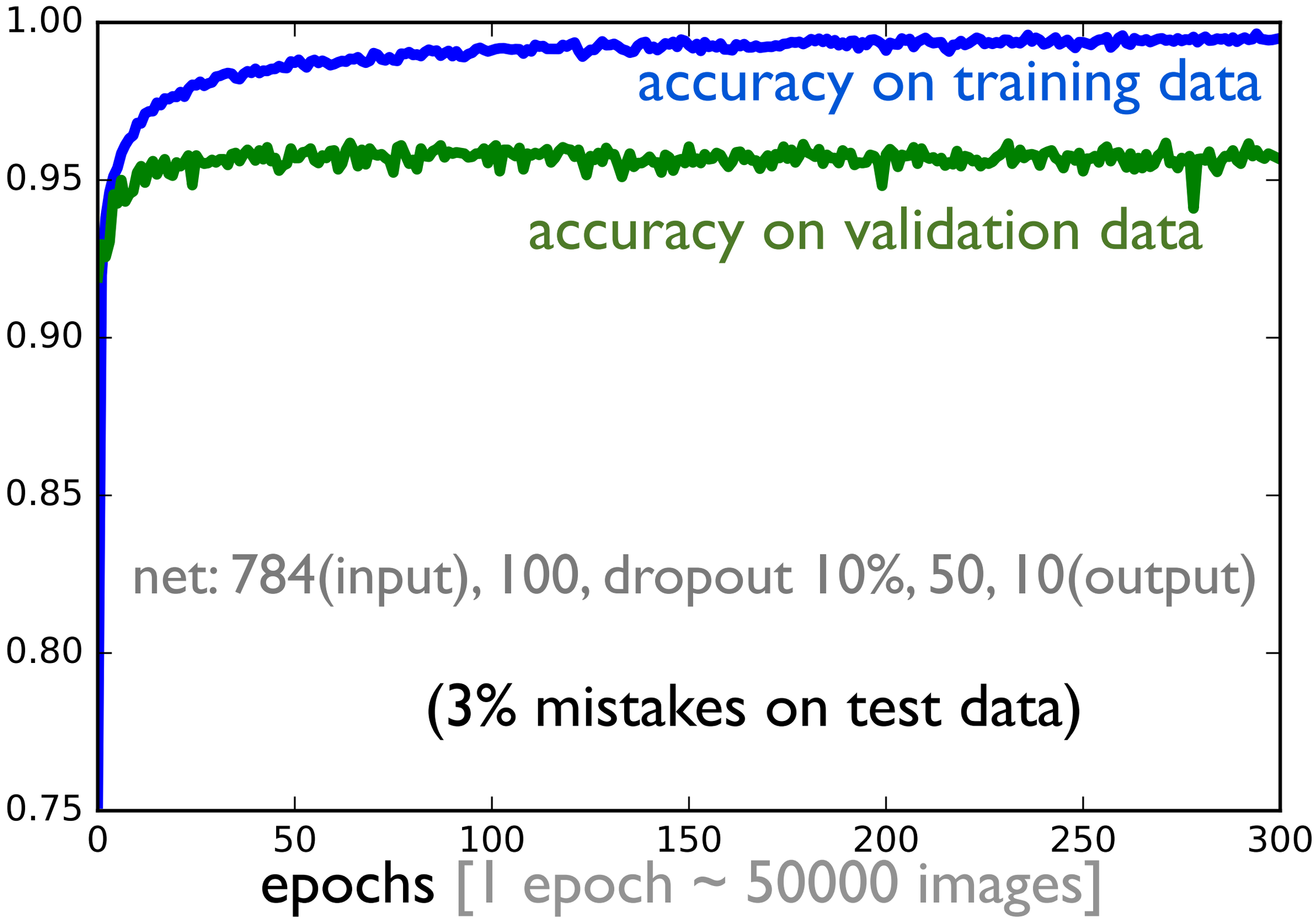
what to do:

- always measure accuracy against validation data, independent of training data
- strategy: stop after reaching maximum in validation accuracy ("early stopping")
- strategy: generate fresh training data by distorting existing images (or produce all training samples algorithmically, never repeat a sample!)
- strategy: "dropout" – set to zero random neuron values during training, such that the network has to cope with that noise and never learns too much detail

Generating new training images by transformations

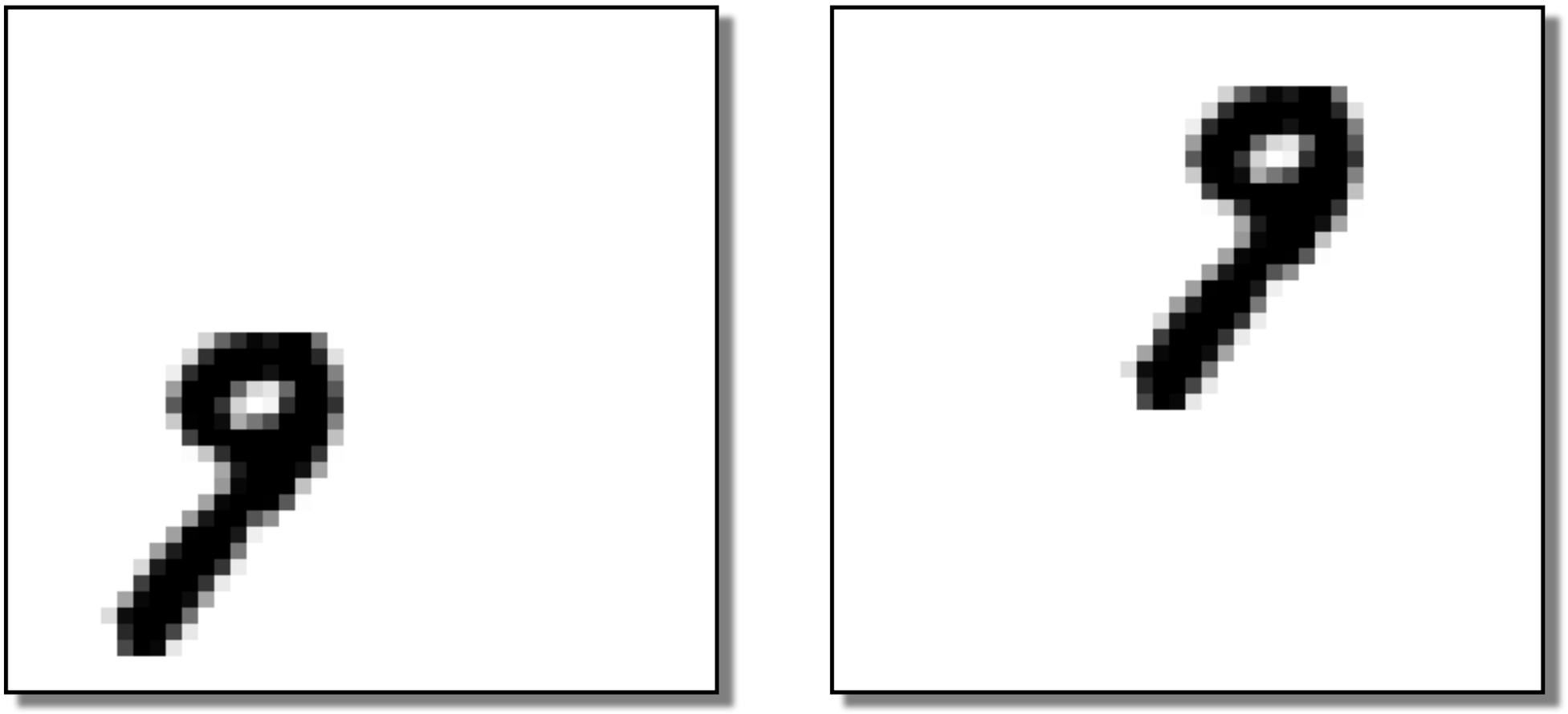


Accuracy during training



Convolutional Networks

Exploit translational invariance!

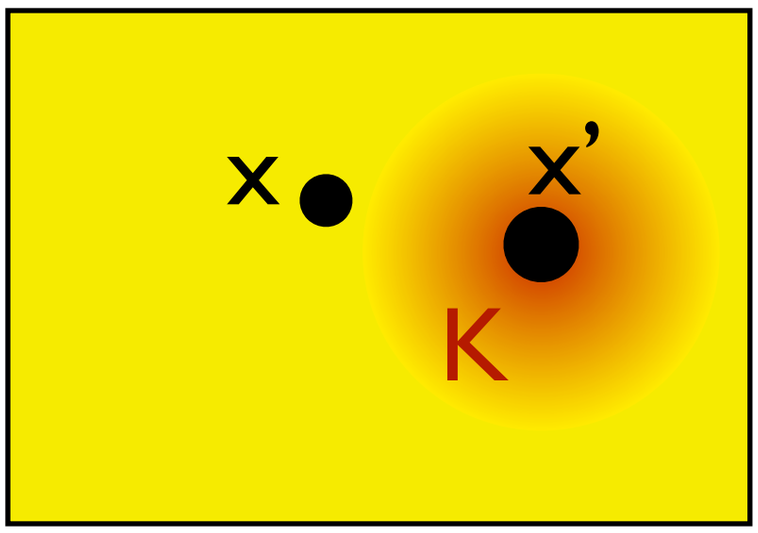


different image, same meaning!

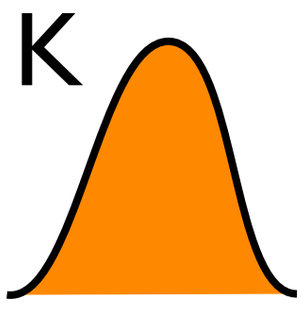
Convolutions

$$F^{\text{new}}(x) = \int K(x - x')F(x')dx'$$

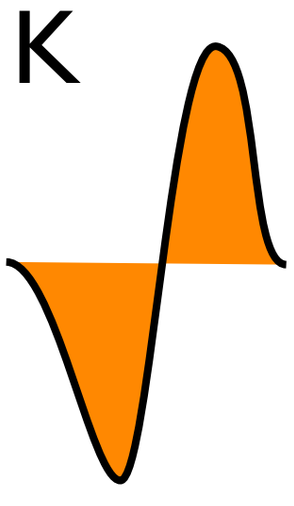
"kernel"



In physics:
- Green's functions for linear partial differential equations (diffusion, wave equations)
- Signal filtering



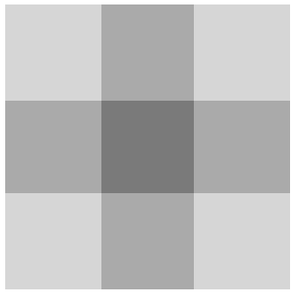
smoothing



(approx.) derivative

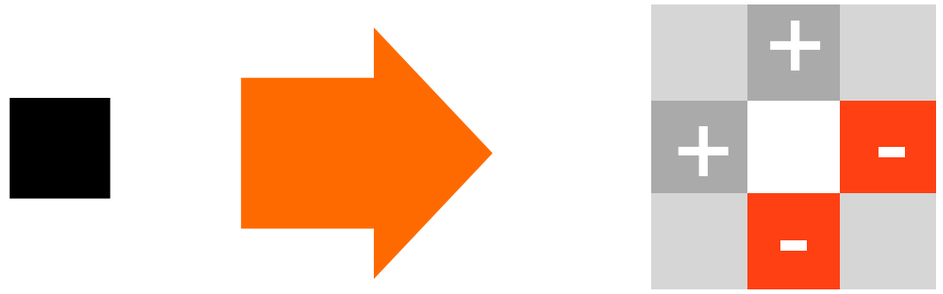
Image filtering: how to blur...

original pixel



resulting pattern

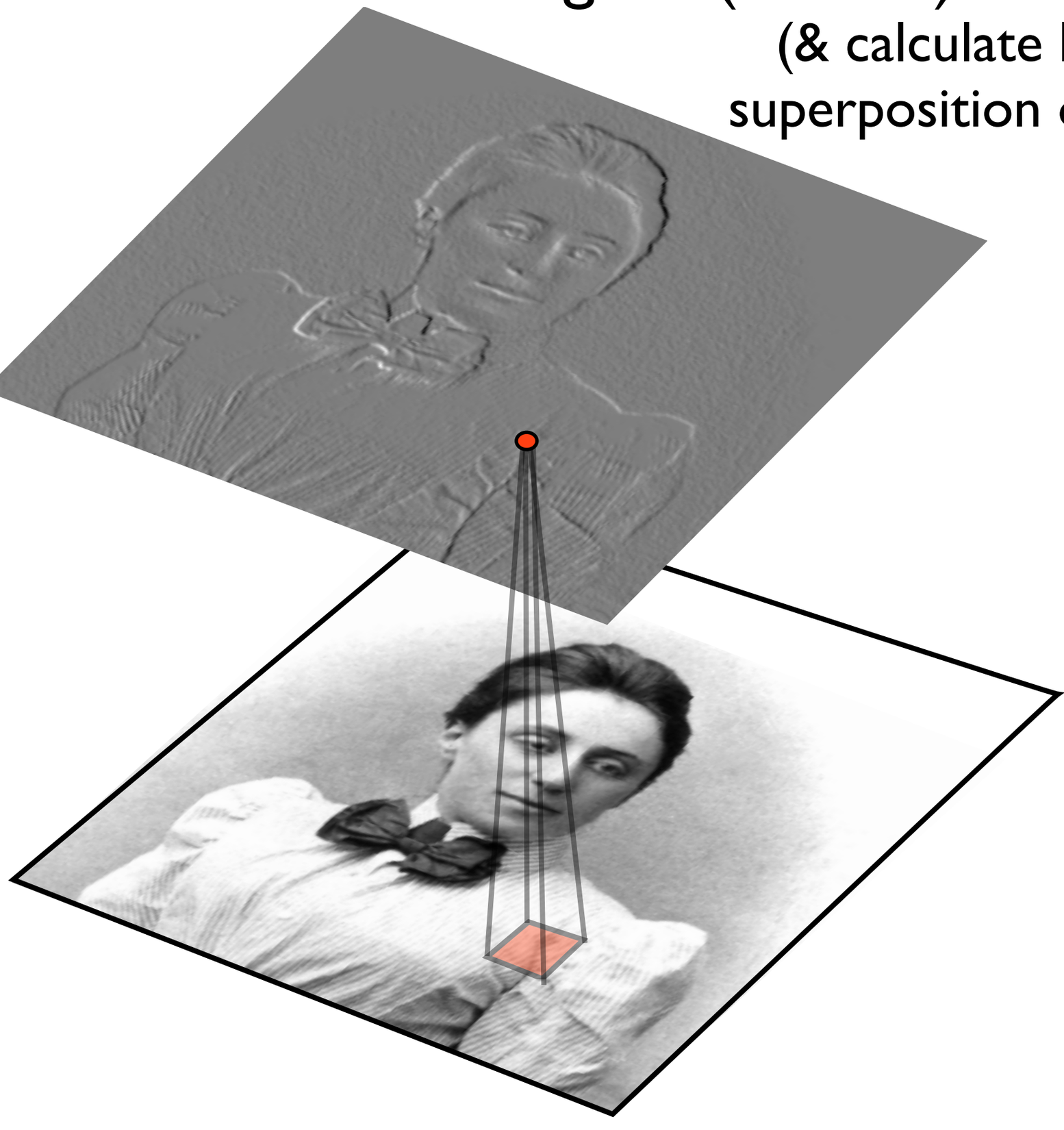
Image filtering: how to obtain contours...



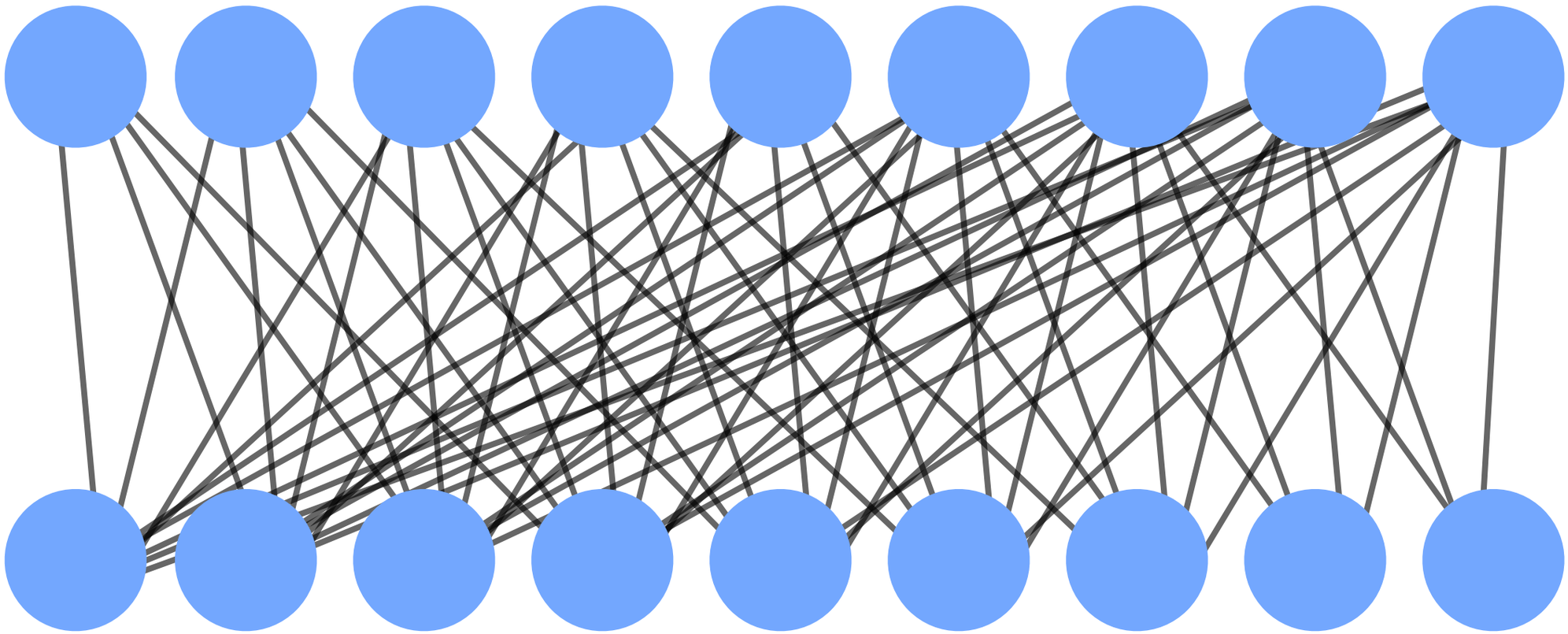
Alternative view:

Scan kernel over original (source) image

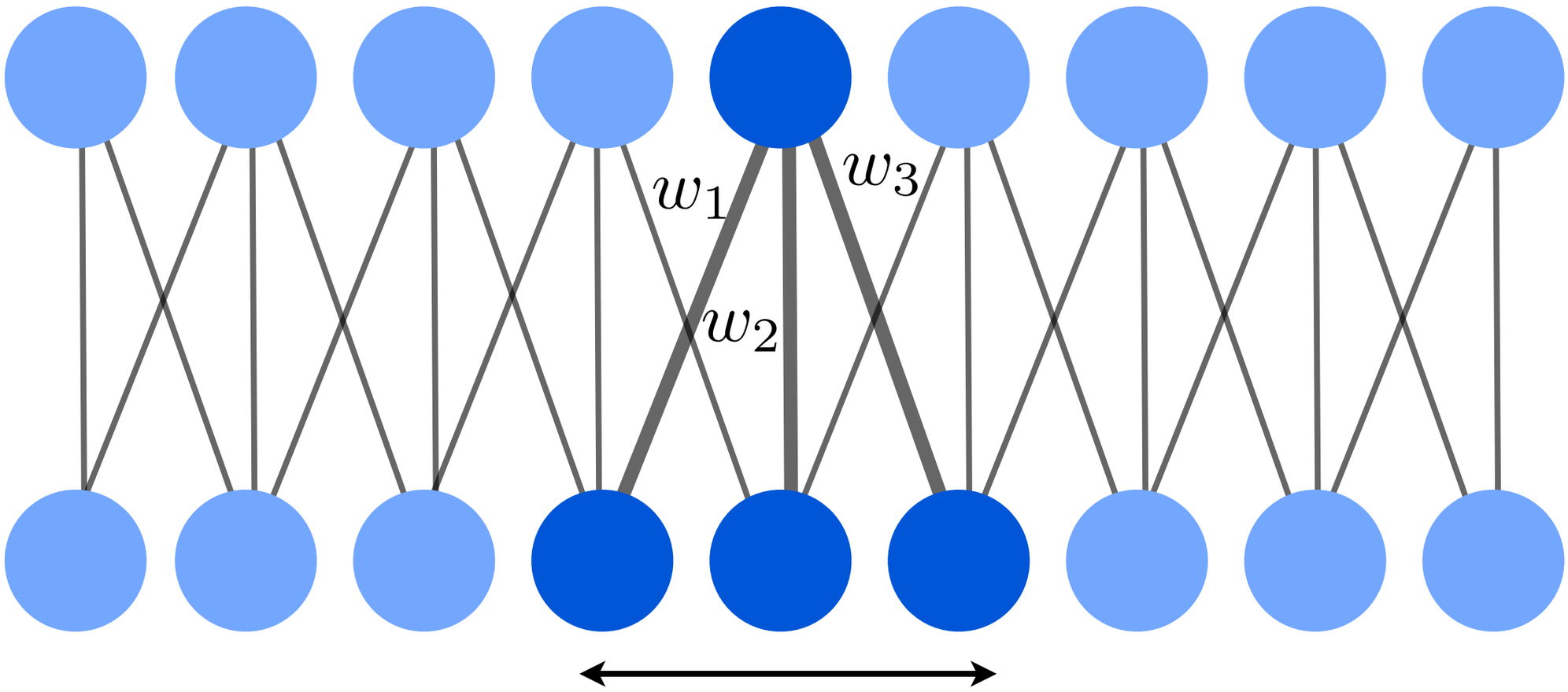
(& calculate linear, weighted superposition of original pixel values)



“Fully connected (dense) layer”

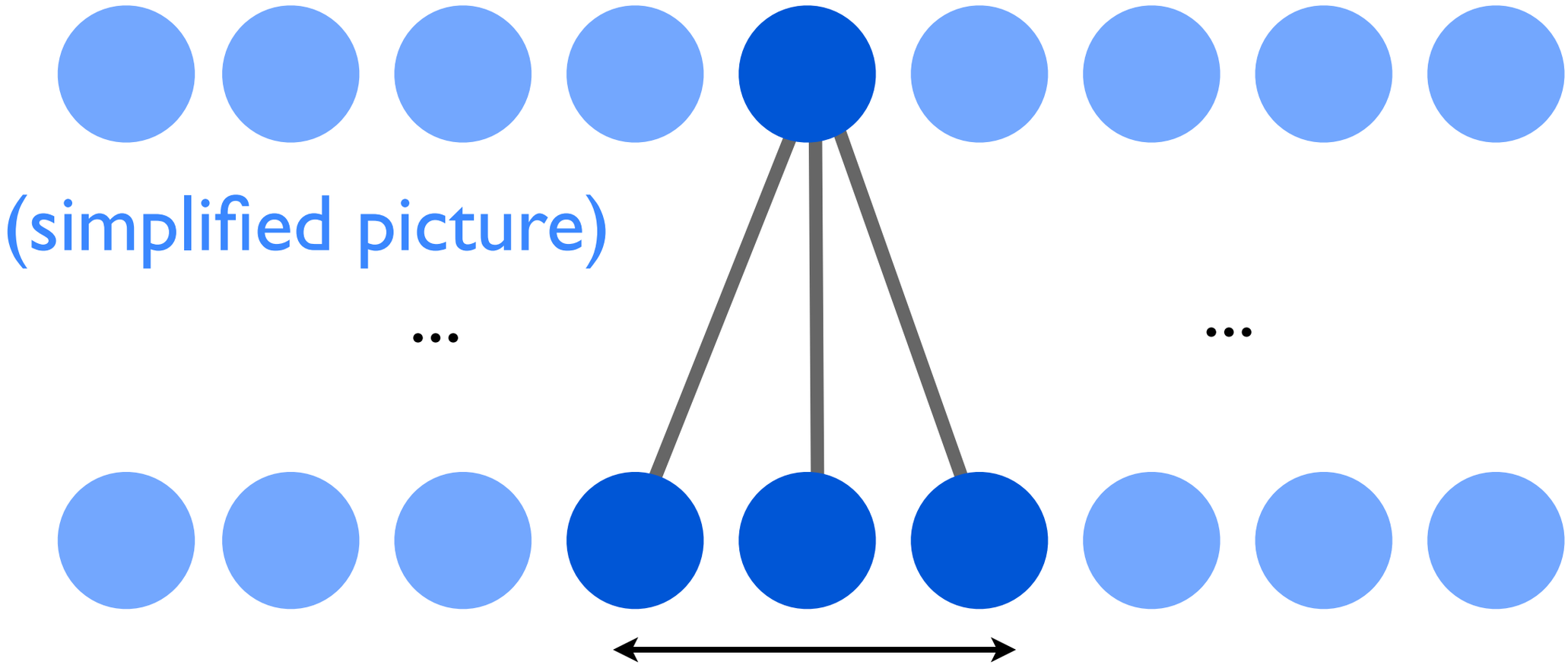


“Convolutional layer”



Same weights (=”kernel”=”filter”) used for each neuron in the top layer!

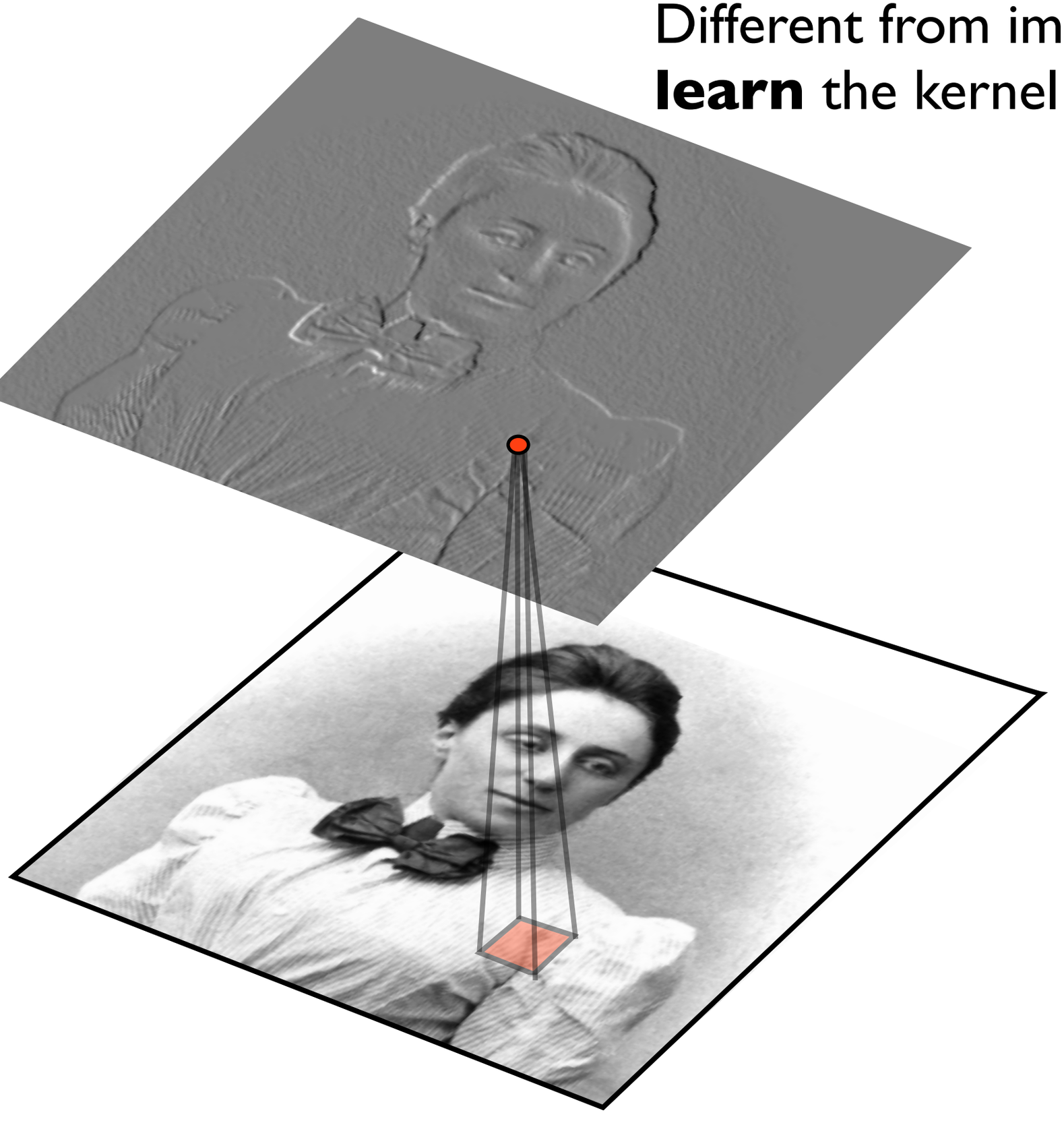
“Convolutional layer”



Same weights (=”kernel”=”filter”) used for each neuron in the top layer!

Scan kernel over original (source) image

Different from image processing:
learn the kernel weights!



Convolutional neural networks

Exploit translational invariance (features learned in one part of an image will be automatically recognized in different parts)

Drastic reduction of the number of weights stored!

fully connected: N^2 (N=size of layer/image)

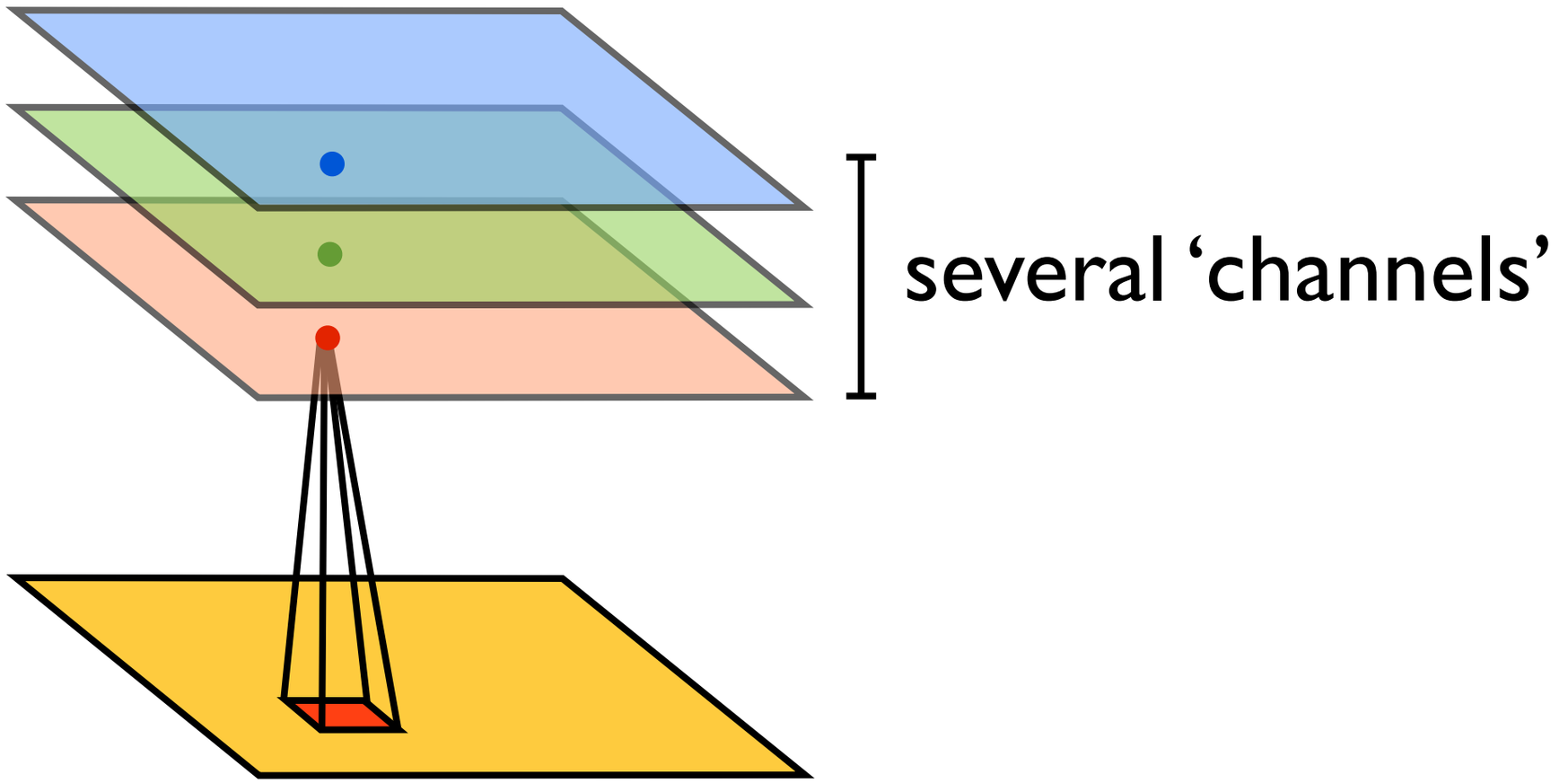
convolutional: M (M=size of kernel)

independent of the size of the image!

lower memory consumption, improved speed

Several filters (kernels)

e.g. one for smoothing, one for contours, etc.



in keras:

2D convolutional layer

input: NxN image, only 1 channel [need to specify this only for first layer after input]

```
net.add(Conv2D(input_shape=(N,N,1),
  filters=20, kernel_size=[11,11],
  activation='relu',padding='same'))
```

next layer will be NxNx20 (20 channels!)

kernel size (region)

what to do at borders (here: force image size to remain the same)