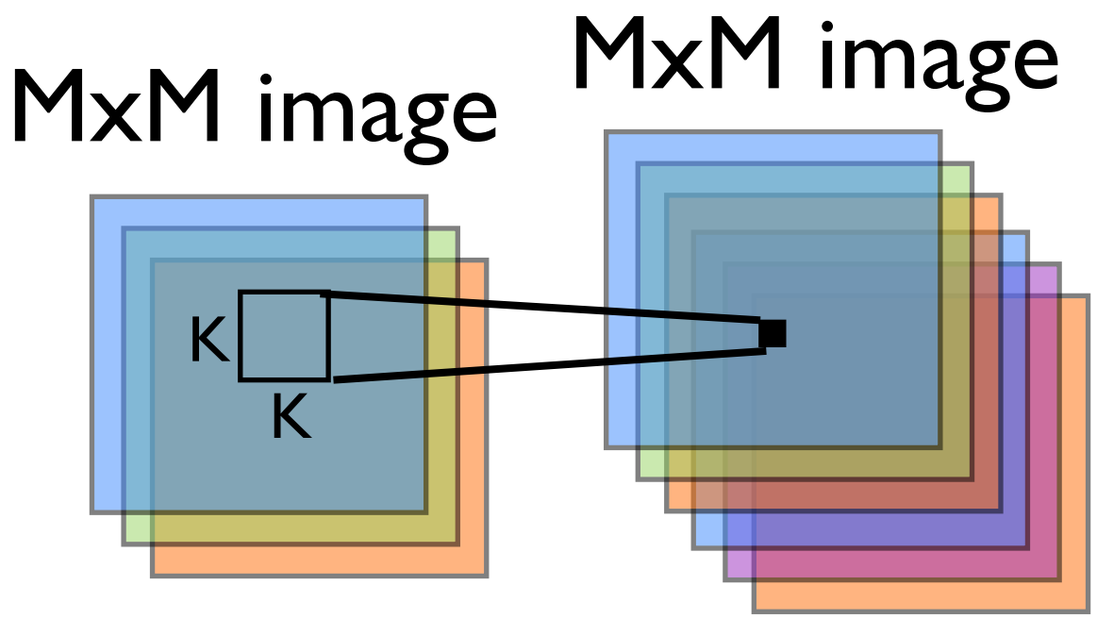
Machine Learning for Physicists

Lecture 6

Summer 2017

University of Erlangen-Nuremberg
Florian Marquardt

“Channels”

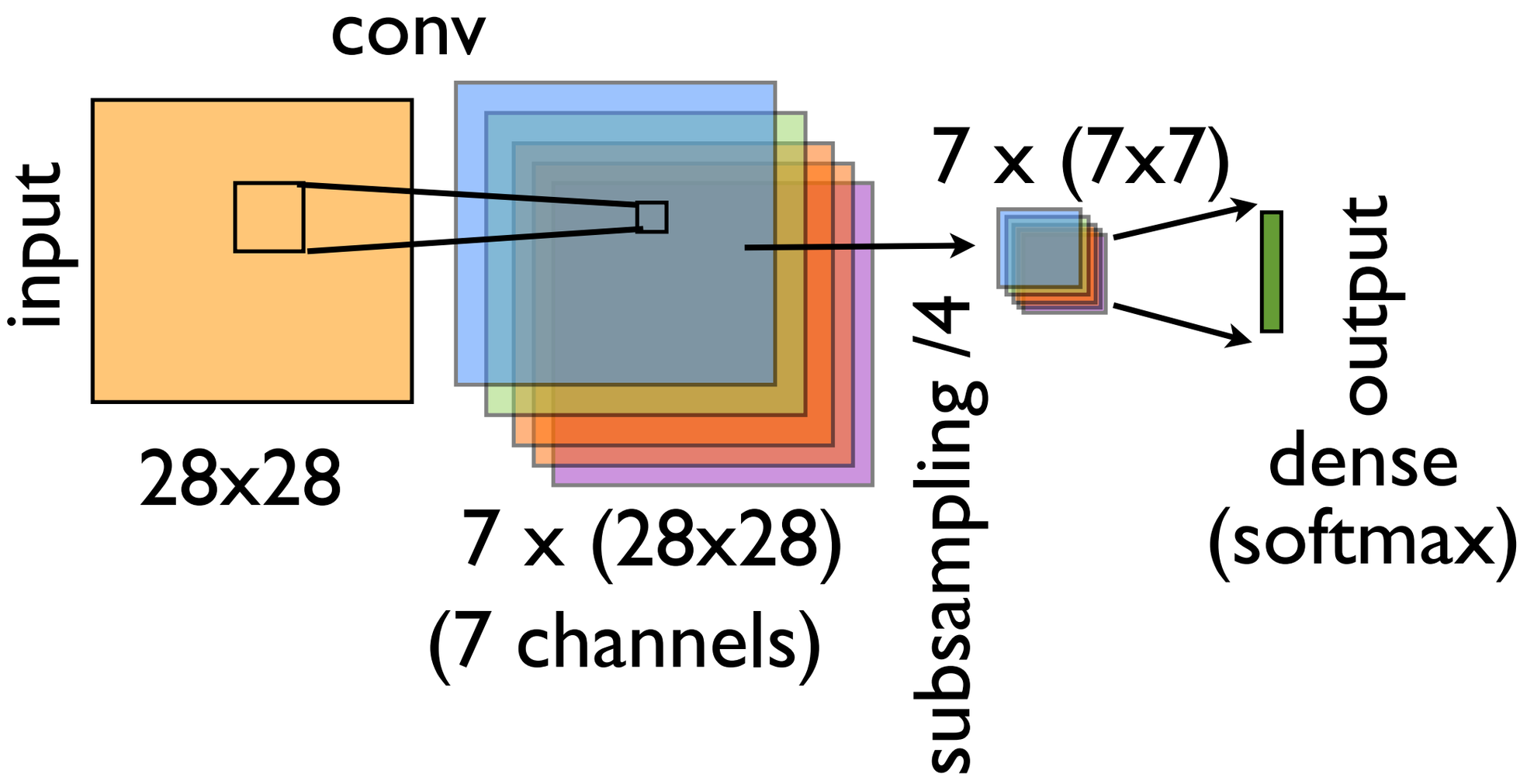


in any output channel, each pixel receives input from KxK nearby pixels in ANY of the input channels (each of those input channel pixel regions is weighted by a different filter); contributions from all the input channels are linearly superimposed

in this example: will need 6x3=18 filters, each of size KxK (thus: store 18xKxK weights!)

Note: keras automatically takes care of all of this, need only specify number of channels

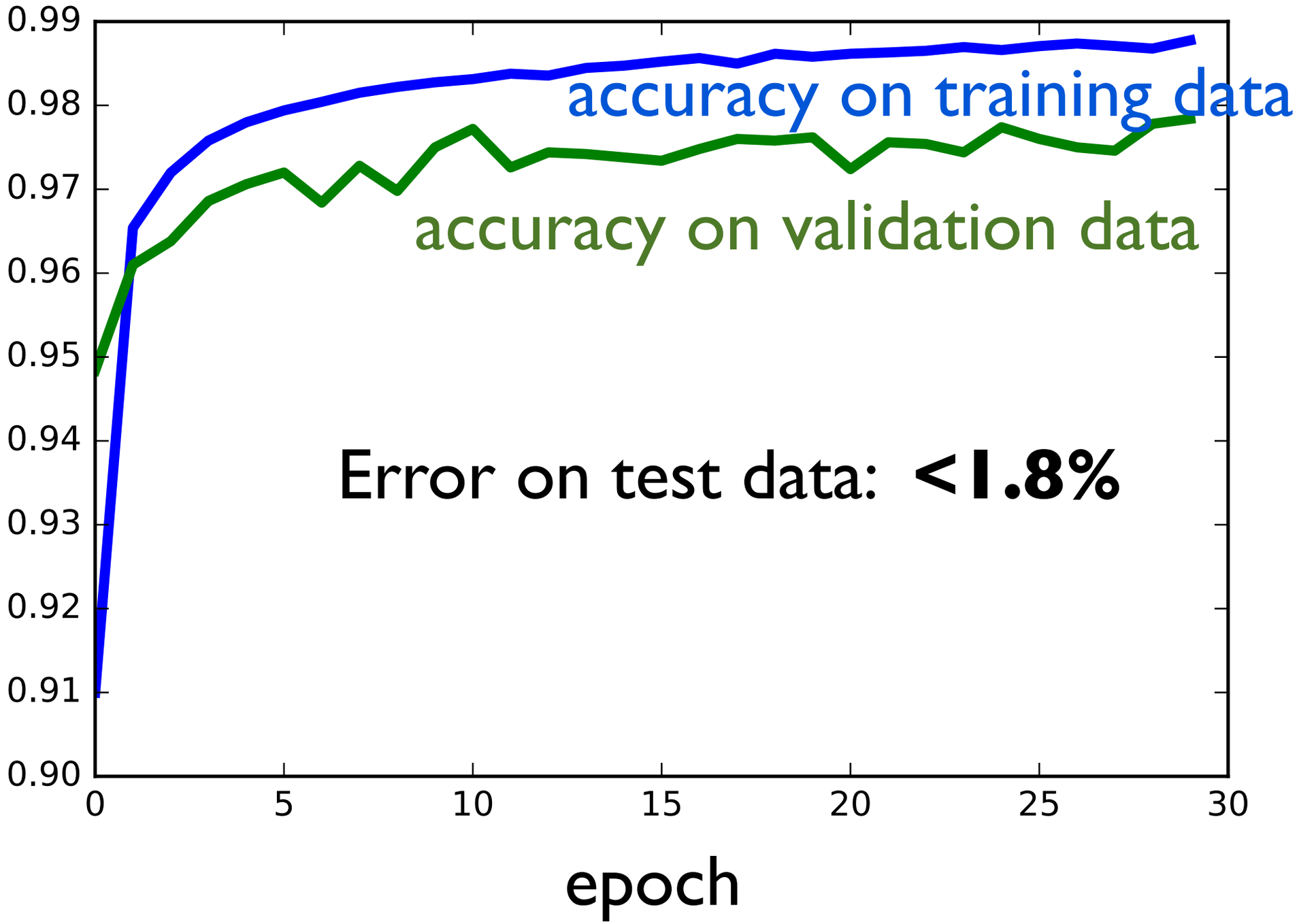
Handwritten digits recognition with a convolutional net



```
# initialize the convolutional network
def init_net_conv_simple():
    global net, M
    net = Sequential()
    net.add(Conv2D(input_shape=(M,M,1), filters=7,
kernel_size=[5,5],activation='relu',padding='same'))
    net.add(AveragePooling2D(pool_size=4))
    net.add(Flatten())
    net.add(Dense(10, activation='softmax'))
    net.compile(loss='categorical_crossentropy',
optimizer=optimizers.SGD(lr=1.0),
metrics=['categorical_accuracy'])
```

← needed for transition to dense layer!

note: M=28 (for 28x28 pixel images)

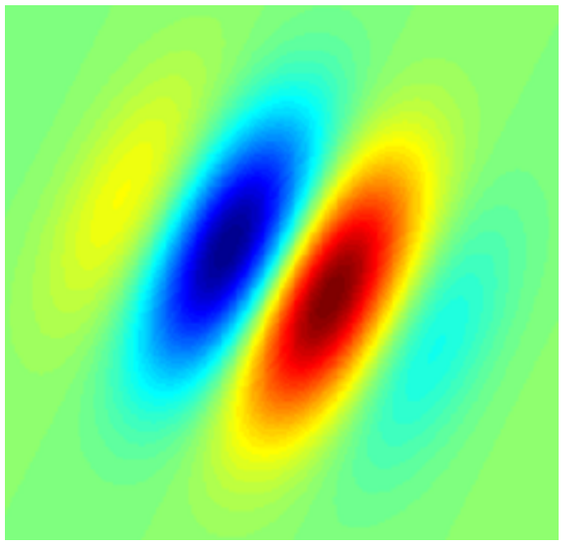

0.99
0.98
0.97
0.96
0.95
0.94
0.93
0.92
0.91
0.90
0
5
10
15
20
25
30
accuracy on training data
accuracy on validation data
Error on test data: **<1.8%**
epoch

The convolutional filters

Interpretation: try to extract common features of input images!

"diagonal line", "curve bending towards upper right corner", etc.

An aside: Gabor filters



(Image: Wikipedia)

2D Gauss times sin-function

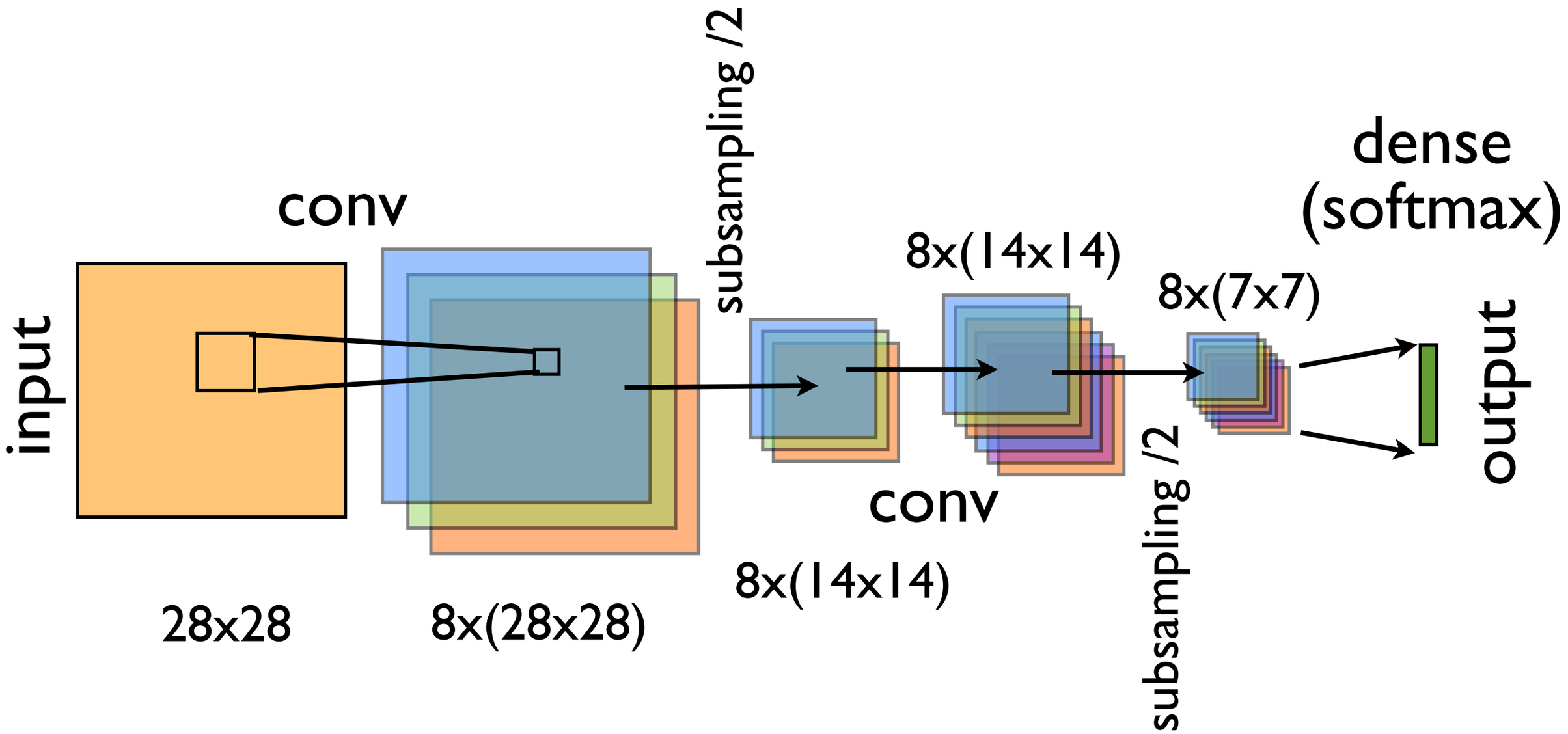
encodes orientation and
spatial frequency

useful for feature extraction in images
(e.g. detect lines or contours of certain
orientation)

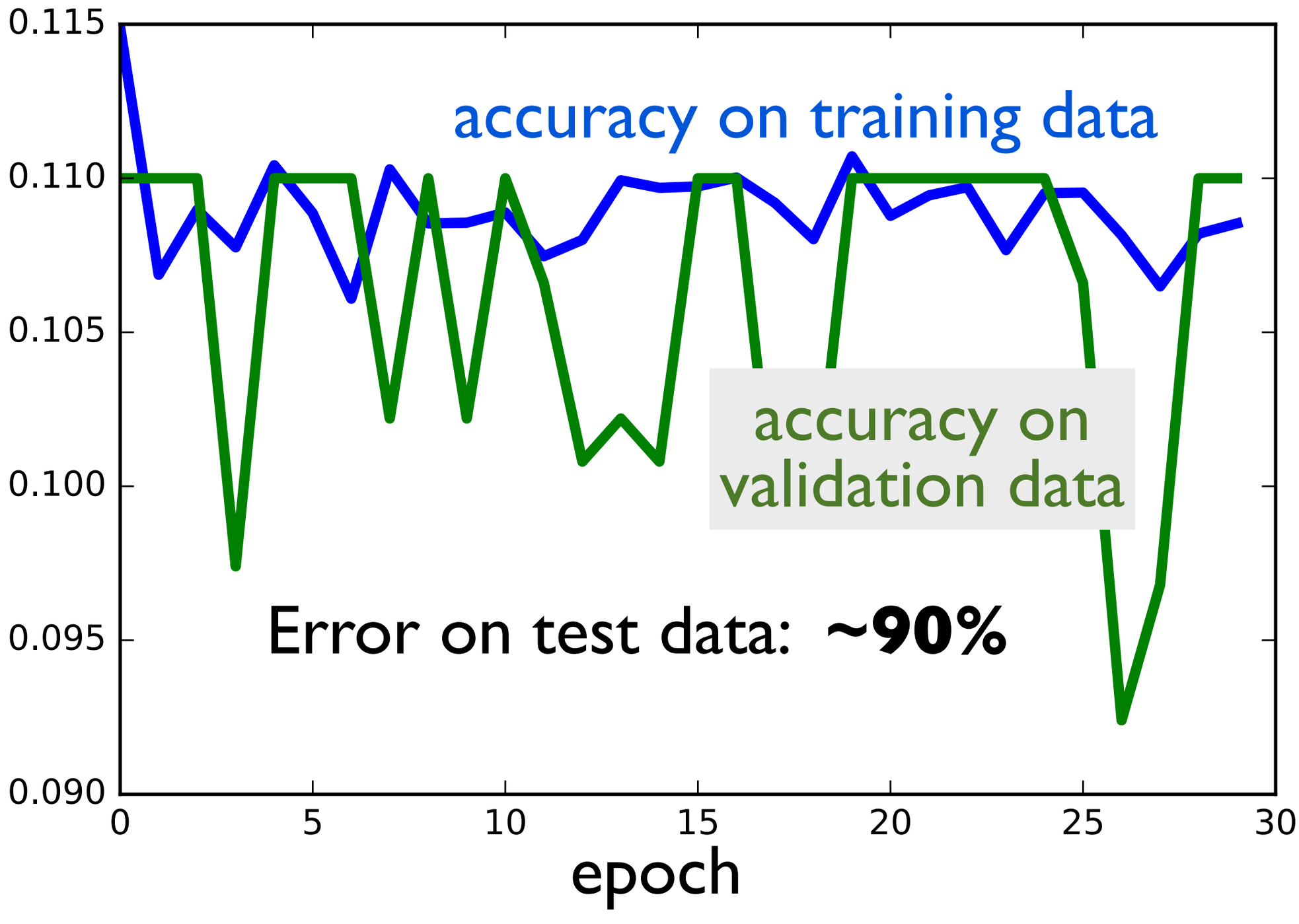
believed to be good
approximation to first
stage of image processing
in visual cortex

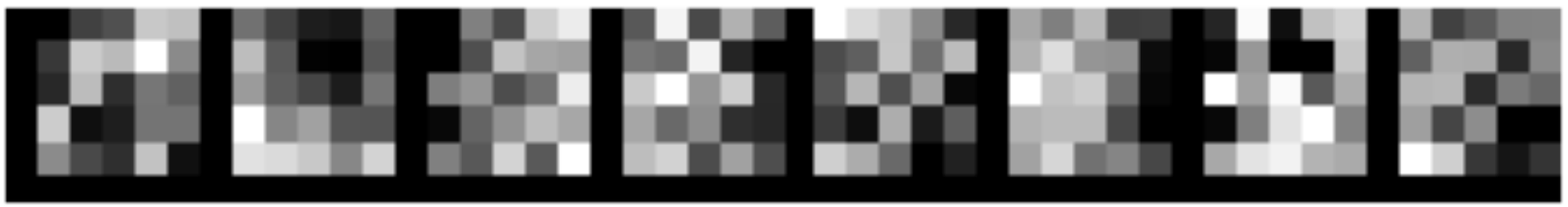
Handwritten digits recognition with a convolutional net

Let's get more ambitious! Train a two-stage convolutional net!

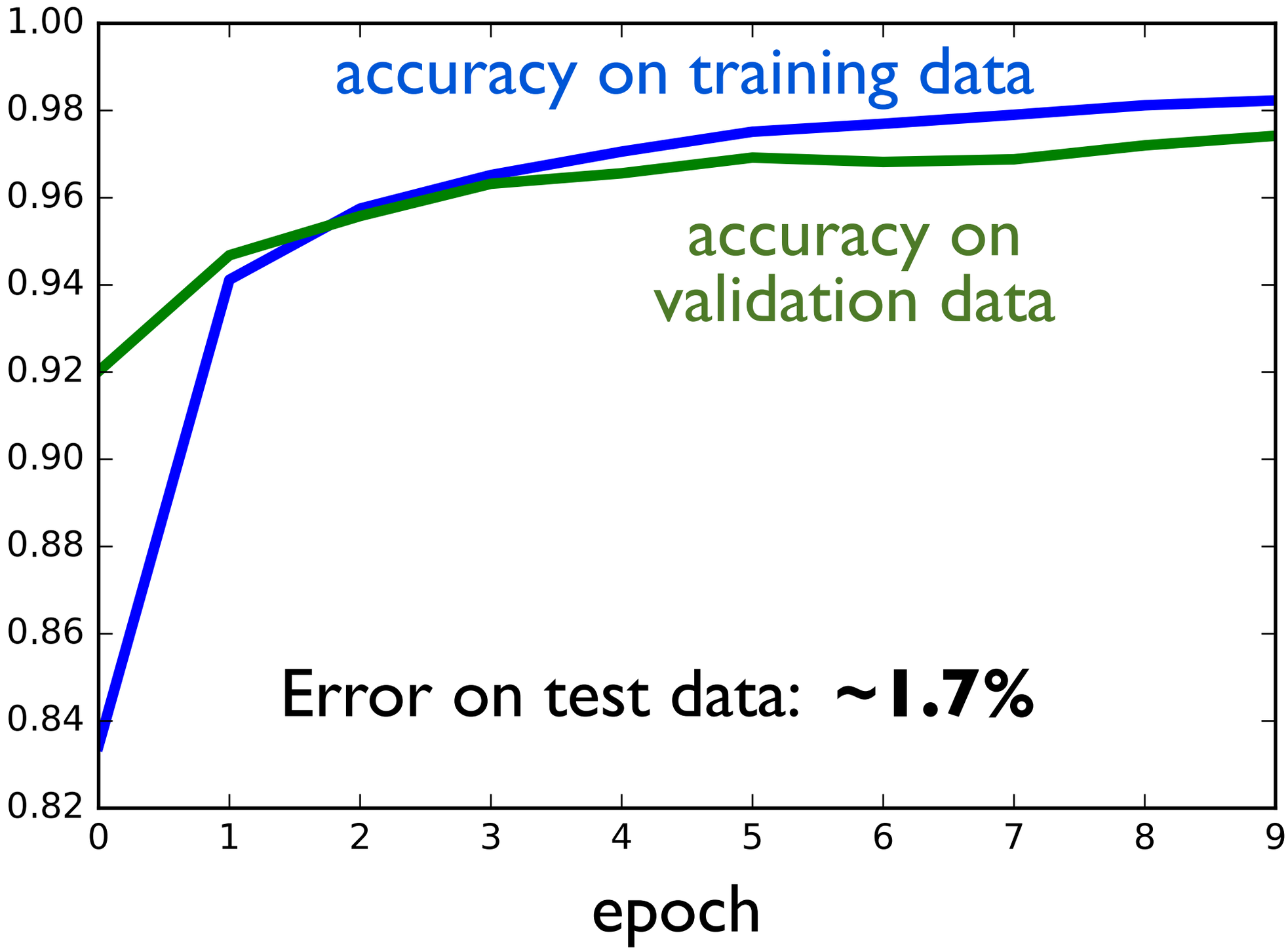


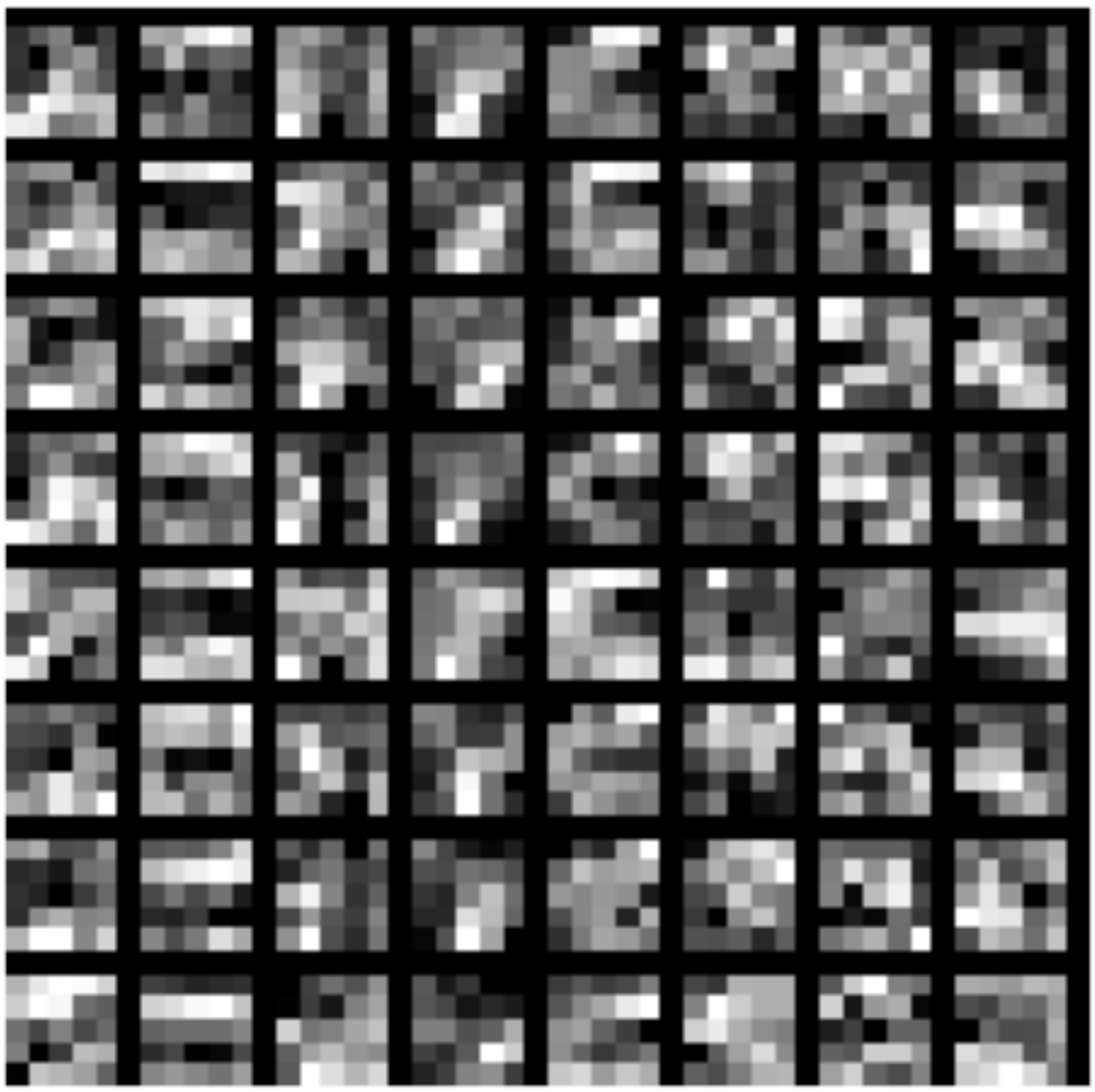
Does not learn at all! Gets 90% wrong!





same net, with adaptive learning rate
(see later; here: 'adam' method)





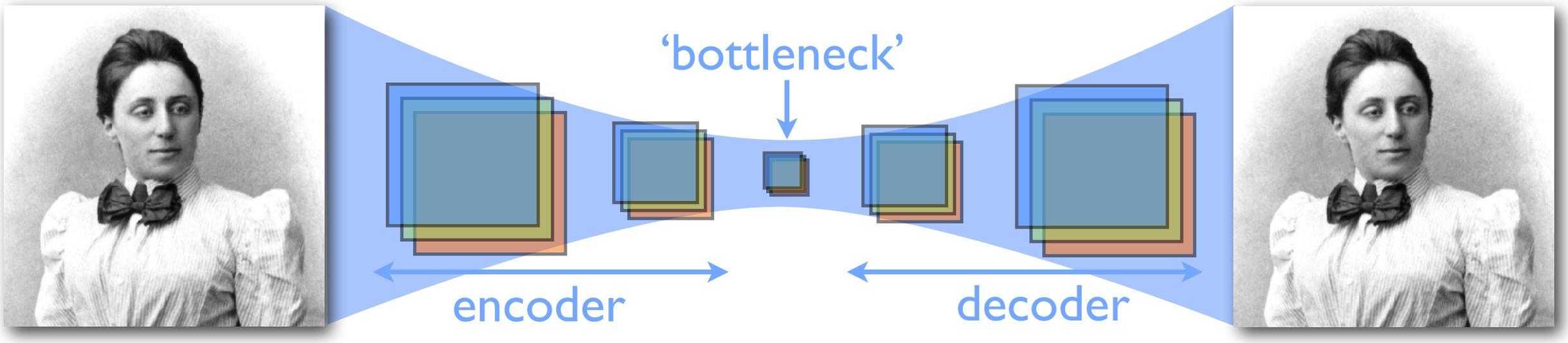
Homework

try and extract the filters after longer training (possibly with enforcing sparsity)

Unsupervised learning

Extracting the crucial features of a large class of training samples without any guidance!

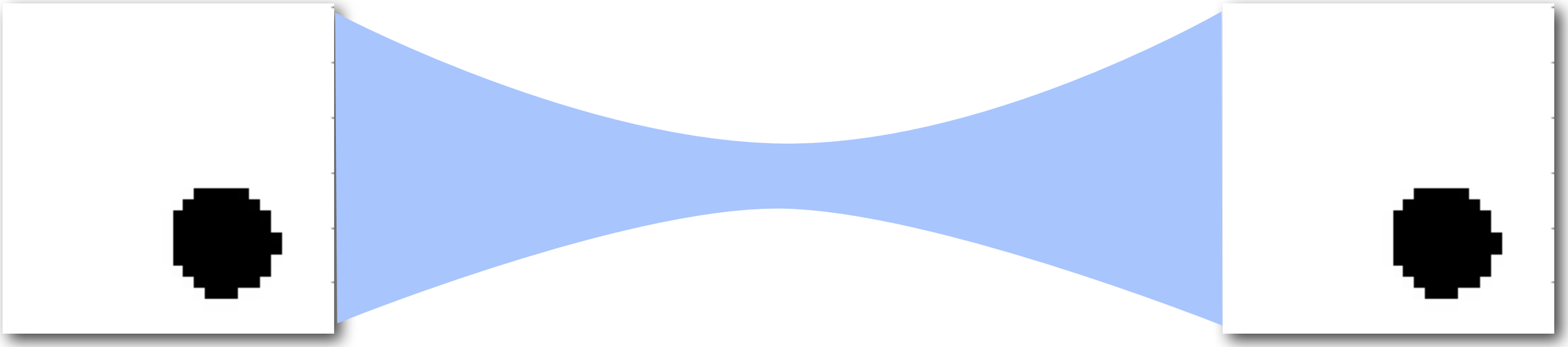
Autoencoder



- Goal: reproduce the input (image) at the output
- An example of unsupervised learning (no need for 'correct results' / labeling of data!)
- Challenge: feed information through some small intermediate layer ('bottleneck')
- This can only work well if the network learns to extract the crucial features of the class of input images
- a form of data compression (adapted to the typical inputs)

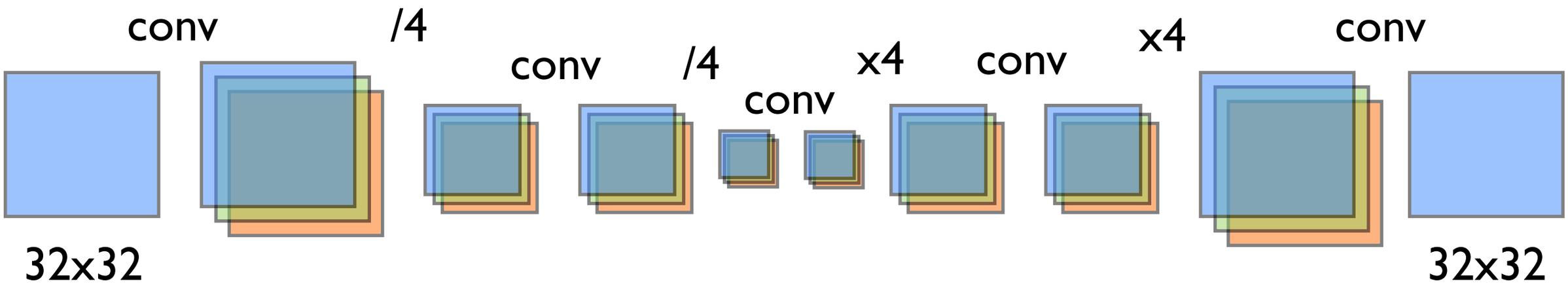
Still: need a lot of training examples

Here: generate those examples algorithmically



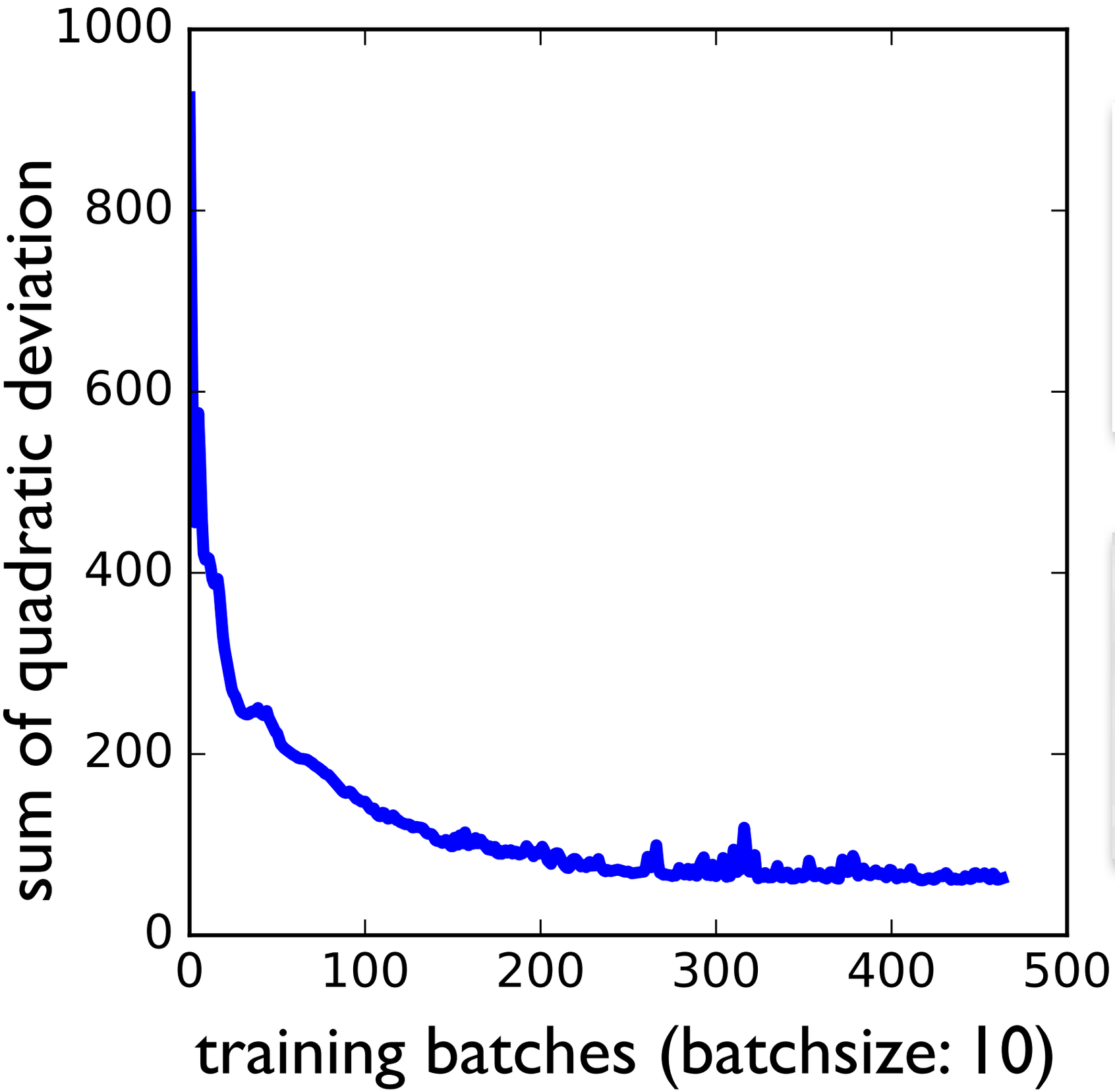
for example: randomly placed circle

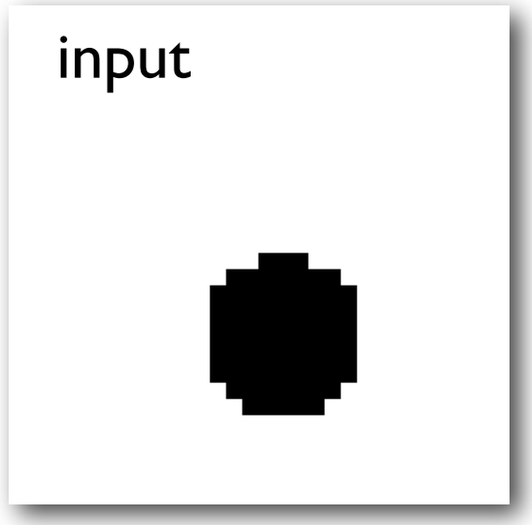
Our convolutional autoencoder network

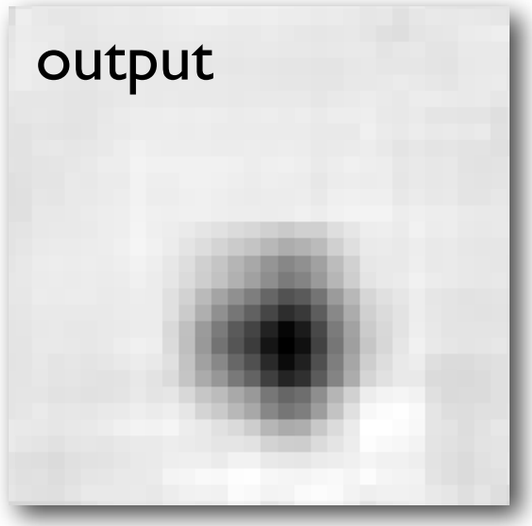


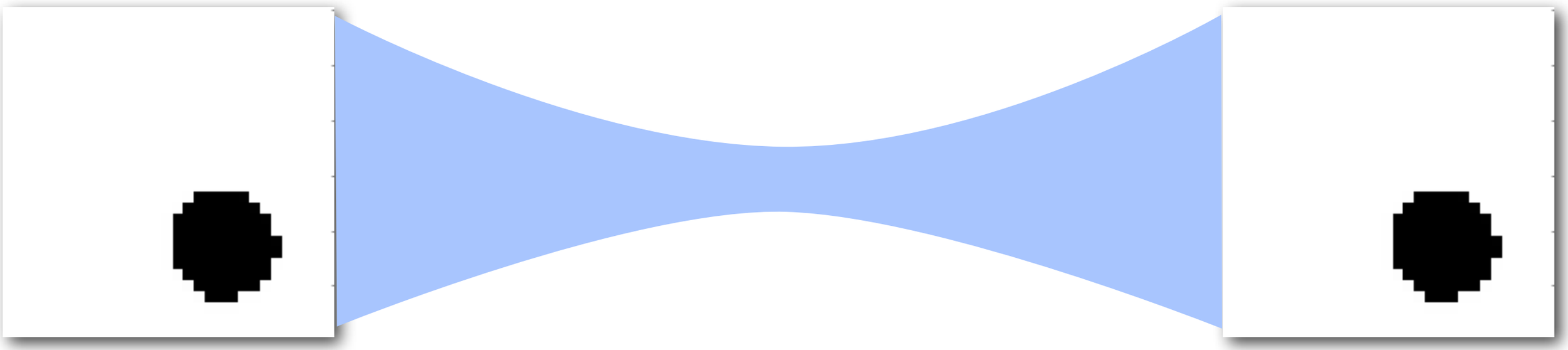
(20 channels in all intermediate steps)

cost function for a single test image

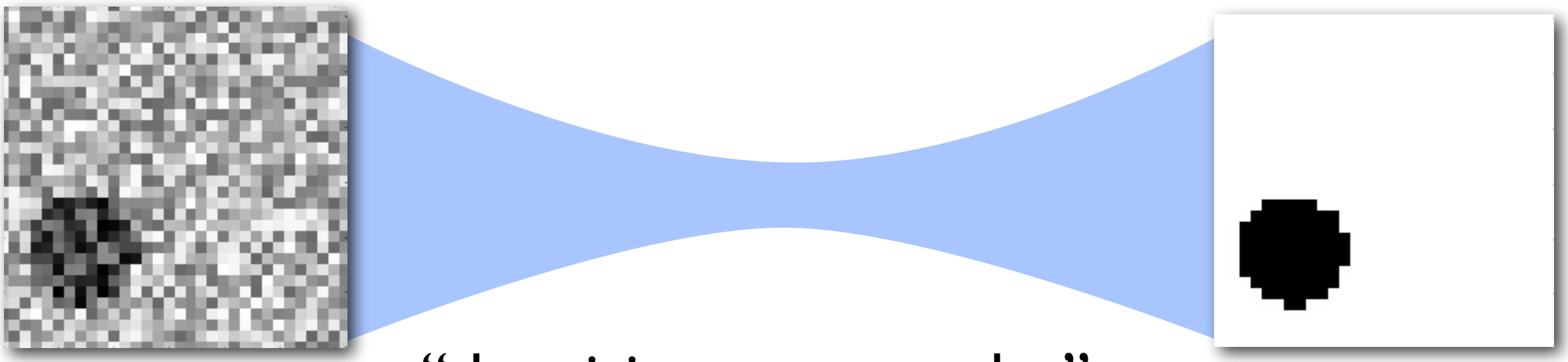






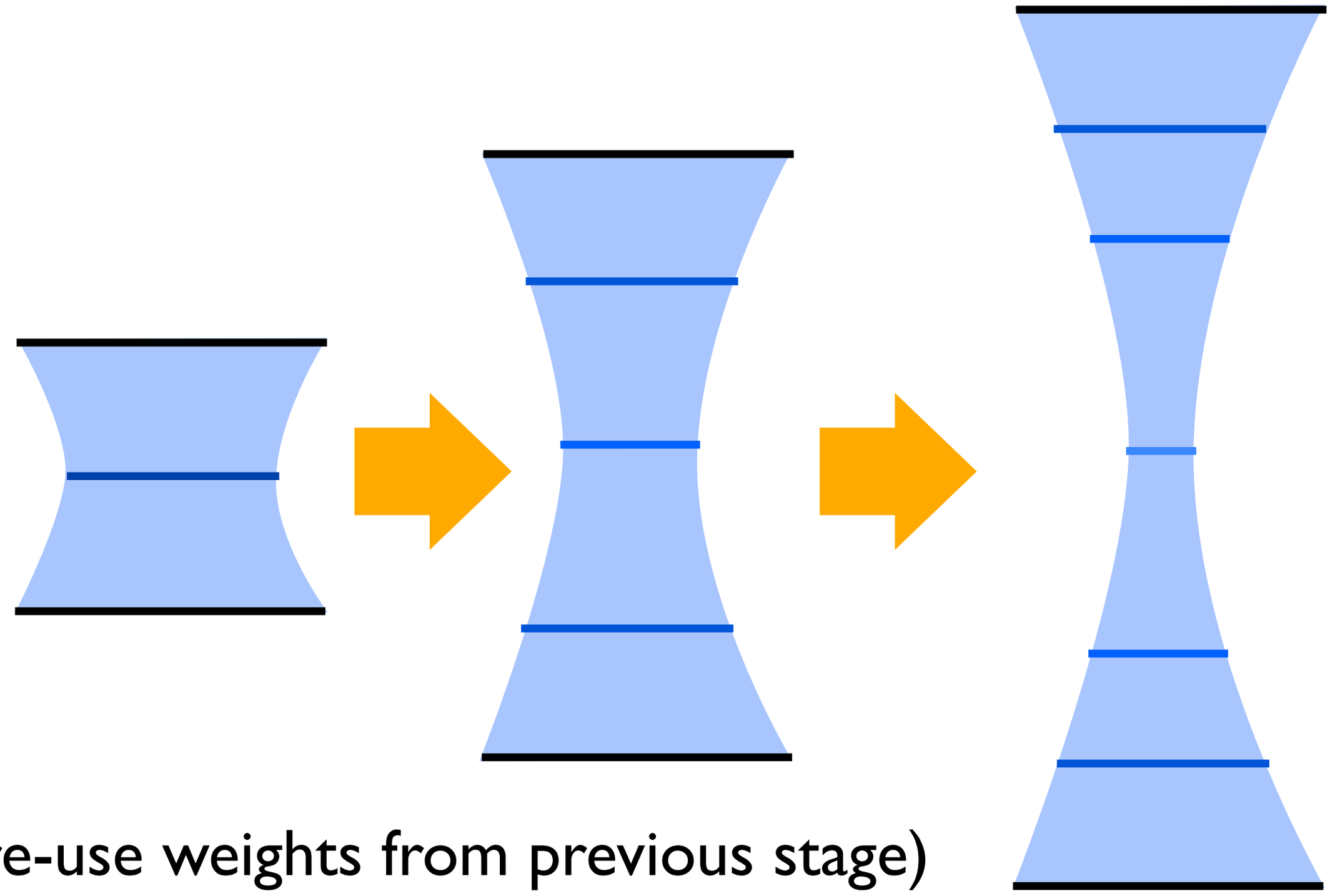


Can make it even more challenging: produce a cleaned-up version of a noisy input image!



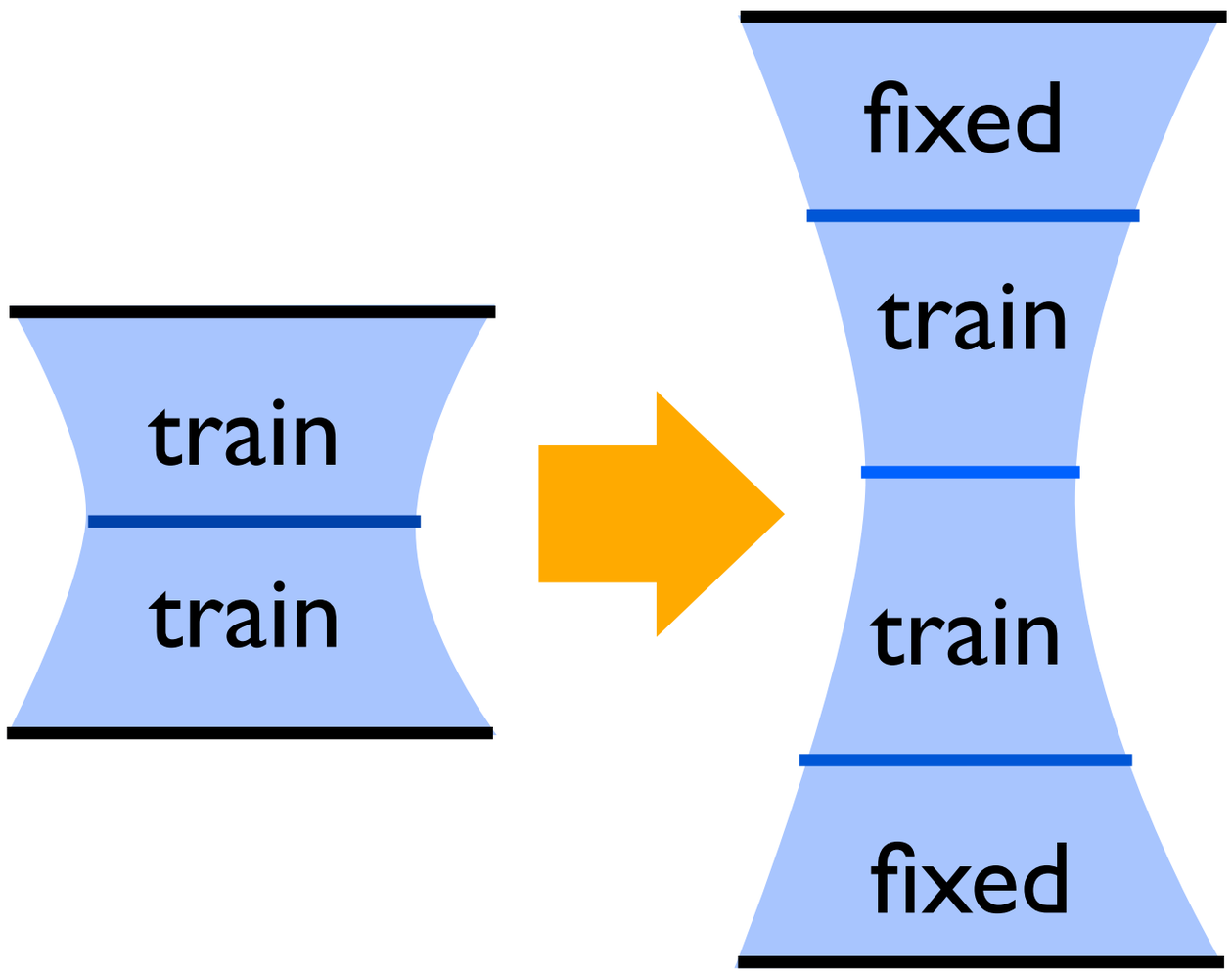
"denoising autoencoder"

Stacking autoencoders



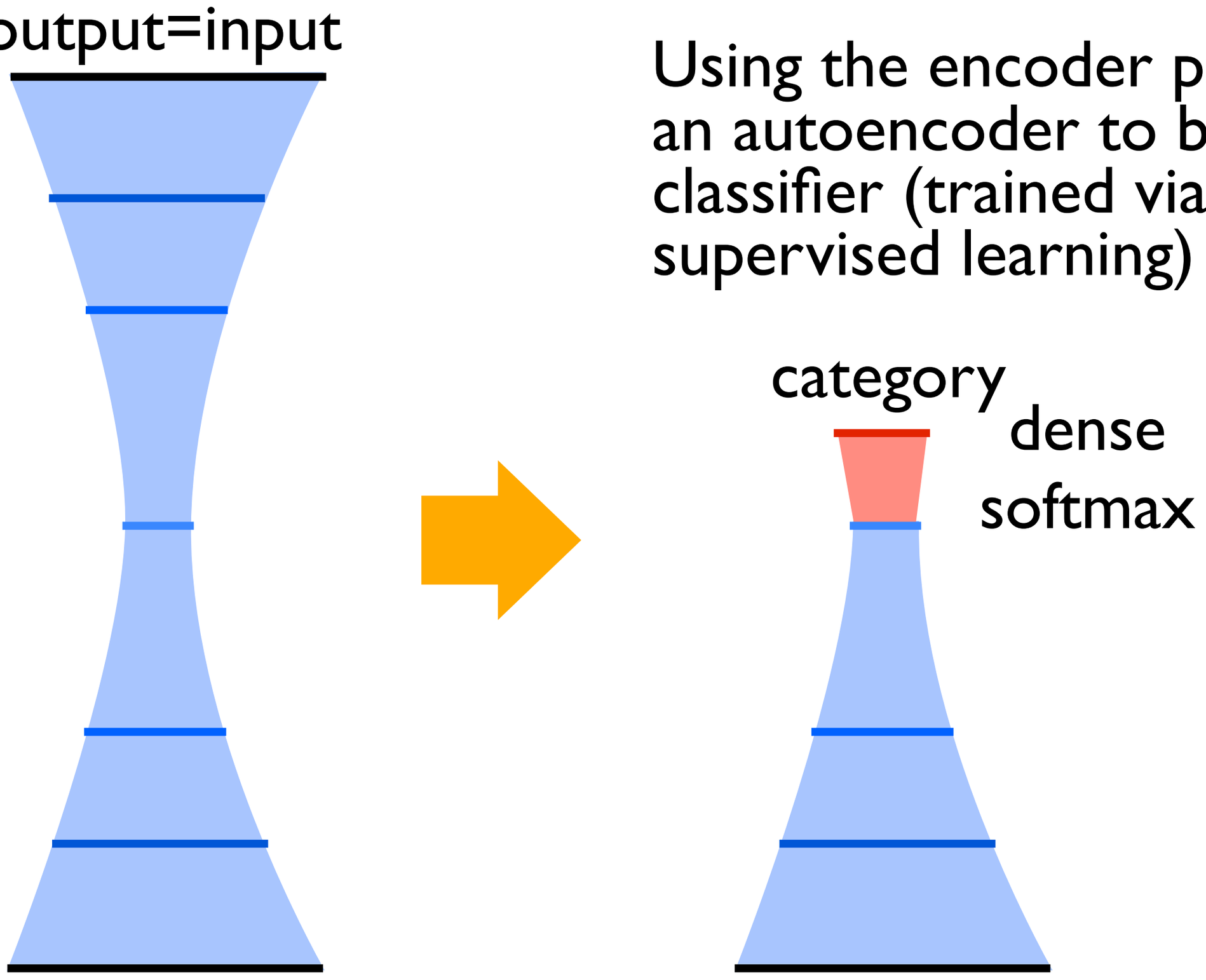
(re-use weights from previous stage)

“greedy layer-wise training”



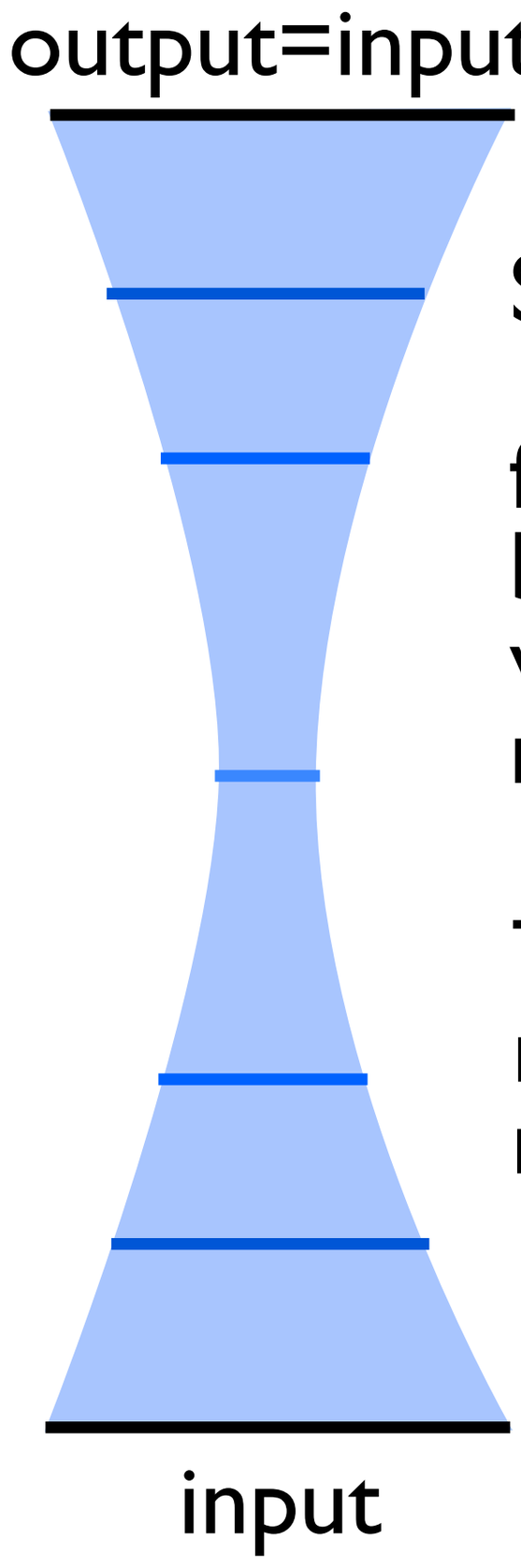
(and so on, for more and more layers)

afterwards can ‘fine-tune’ weights by training all of them together, in the large multi-layer network



Using the encoder part of an autoencoder to build a classifier (trained via supervised learning)

training the autoencoder = “pretraining”



Sparse autoencoder:

force most neurons in the inner layer to be zero (or close to some average value) most of the time, by adding a modification to the cost function

This forces useful higher-level representations even when there are many neurons in the inner layer

(otherwise the network could just 1:1 feed through the input)

What are autoencoders good for?

- Autoencoders are useful for pretraining, but nowadays one can train deep networks (with many layers) from scratch

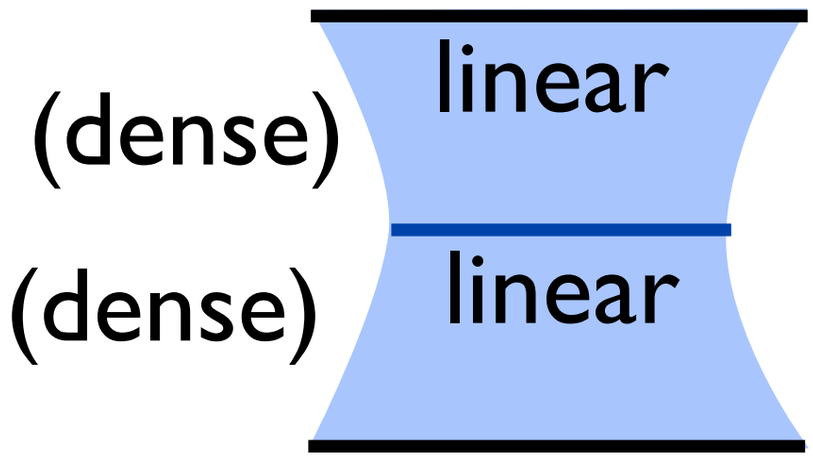
- Autoencoders are an interesting example of unsupervised (or rather self-supervised) learning, but detailed reconstruction of the input (which they attempt) may not be the best method to learn important abstract features

- Still, one may use the compressed representation for visualizing higher-level features of the data

- Autoencoders in principle allow data compression, but are nowadays not competitive with generic algorithms like e.g. jpeg

An aside: Principal Component Analysis (PCA)

Imagine a purely linear autoencoder: which weights will it select?

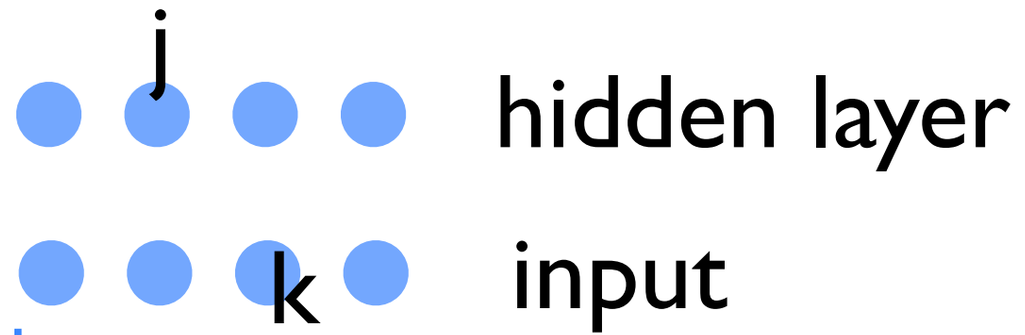


Challenge: number of neurons in hidden layer is smaller than the number of input/output neurons

Each inner-layer neuron can be understood as the projection of the input onto some vector (determined by the weights belonging to that neuron)

set $w_{jk} = \langle v_j | k \rangle$

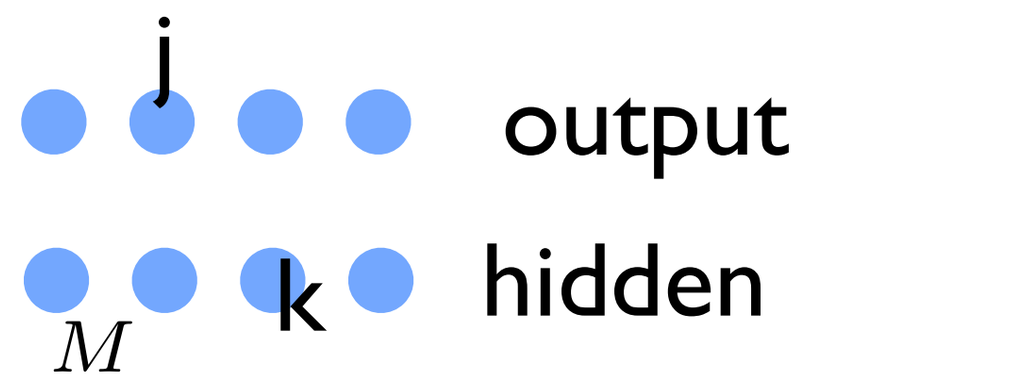
for the input-hidden weights



the hidden layer neuron values will be the amplitudes of the input vector in the "v" basis!

set $w_{jk} = \langle k | v_j \rangle$

for the hidden-output weights



Set restricted projector $\hat{P} = \sum_{j=1} |v_j\rangle \langle v_j|$

where M is the number of neurons in the hidden layer, which is smaller than the size of the Hilbert space, and the vectors form an orthonormal basis (that we still want to choose in a smart way)

The network calculates: $\hat{P} |\psi\rangle$

Mathematically: try to reproduce a vector (input) as well as possible with a restricted basis set!

Note: in the following, for simplicity, we assume the input vector to be normalized, although the final result we arrive at (principal component analysis) also works for an arbitrary set of vectors

We want: $|\psi\rangle \approx \hat{P}\,|\psi\rangle$

"...for all the typical input vectors"

Note: We assume the average has already been subtracted, such that $\langle|\psi\rangle\rangle = 0$

Choose the vectors "v" to minimize the **average** quadratic deviation

$$\left\langle \left\| |\psi\rangle - \hat{P}\,|\psi\rangle \right\|^2 \right\rangle$$

average over all input vectors $|\psi\rangle$

$$= \left\langle \langle\psi|\psi\rangle - \left\langle \psi|\hat{P}\psi \right\rangle \right\rangle$$

Solution: Consider the matrix

$$\hat{\rho} = \langle |\psi\rangle \langle \psi| \rangle = \sum_j p_j \left|\psi^{(j)}\right\rangle \left\langle \psi^{(j)}\right|$$

$$\rho_{mn} = \langle \psi_m \psi_n^* \rangle$$

p: probability of having a particular input vector

This characterizes fully the ensemble of input vectors (for the purposes of linear operations)

[this is the covariance matrix of the vectors]

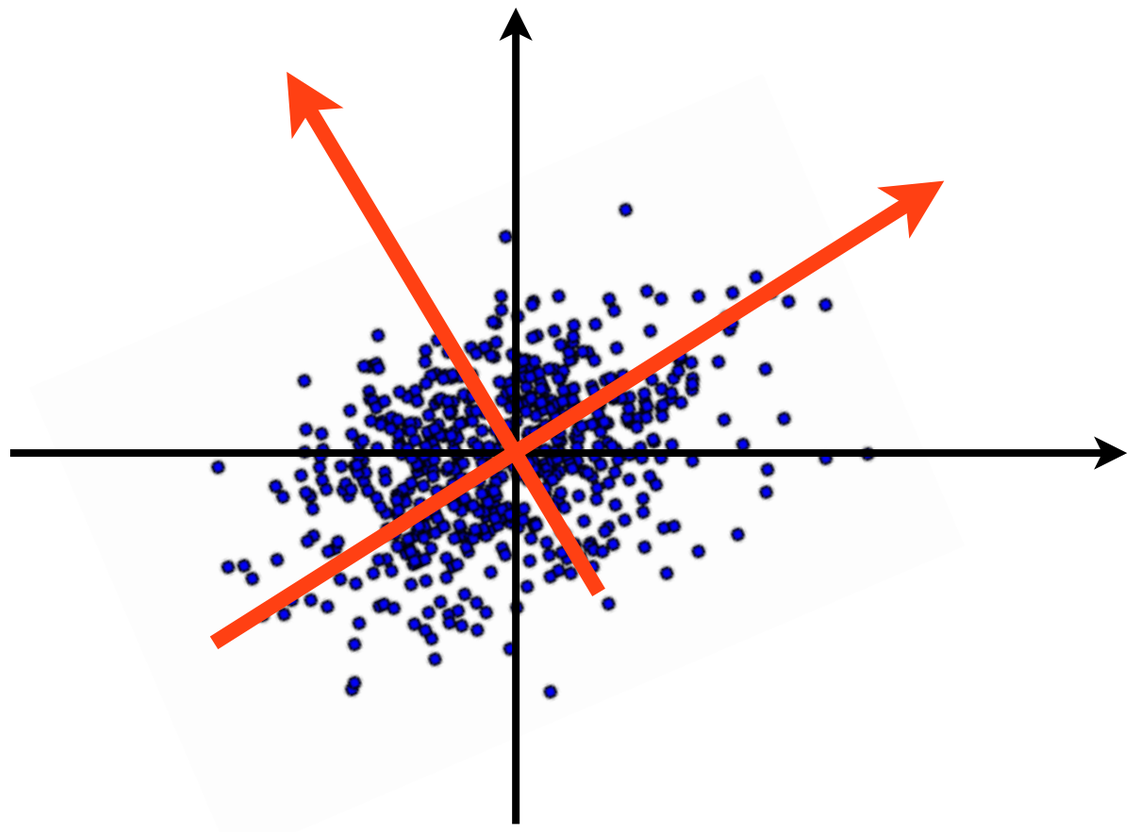
[compare density matrix in quantum physics!]

Claim:

Diagonalize this (hermitean) matrix, and keep the M eigenvectors with the largest eigenvalues. These form the desired set of "v"!

An example in a 2D Hilbert space:

the two eigenvectors of $\hat{\rho}$



(points=end-points of vectors in the ensemble)

Application to the MNIST database

```
shape(training_inputs)
```

the MNIST images

```
(50000, 784)
```

subtract average

```
psi=training_inputs-sum(training_inputs,axis=0)/num_samples
```

(note: we do not do normalization here, in this example, although we could)

```
rho=dot(transpose(psi),psi)
```

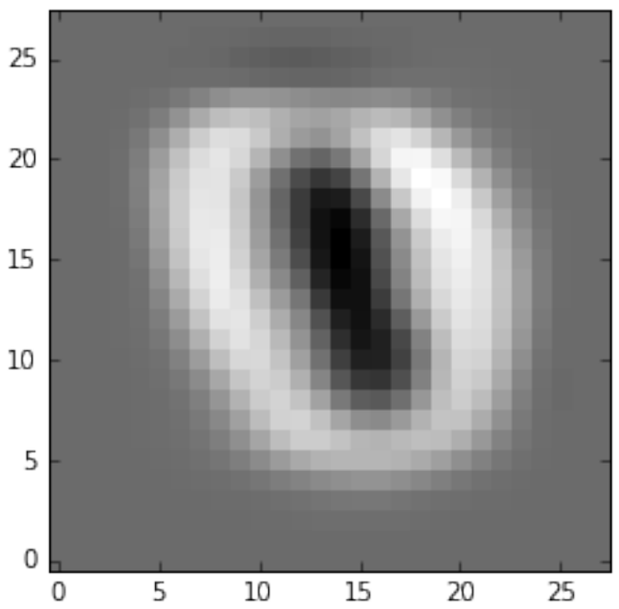
rho will be 784x784 matrix

```
vals,vecs=linalg.eig(rho)
```

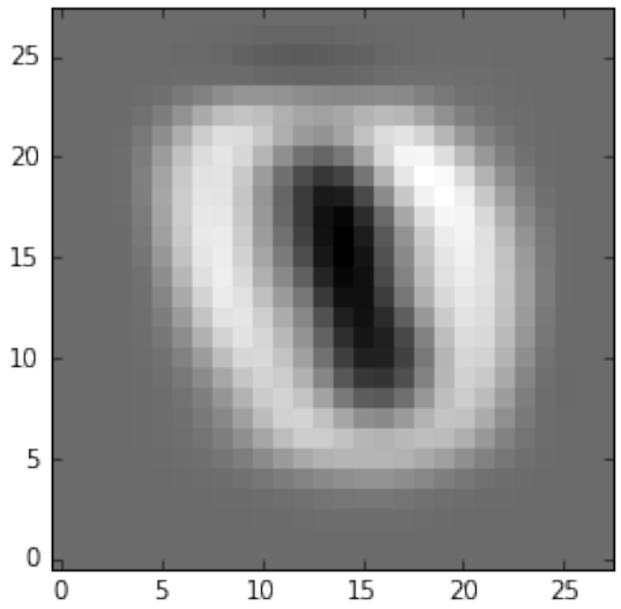
get eigenvalues- and vectors (already sorted, largest first)

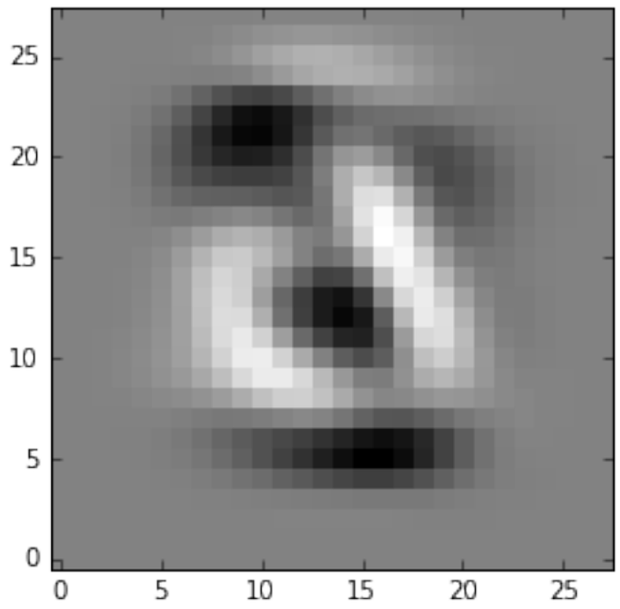
```
plt.imshow(reshape(-vecs[:,0],[28,28]),
origin='lower',cmap='binary',interpolation='nearest')
```

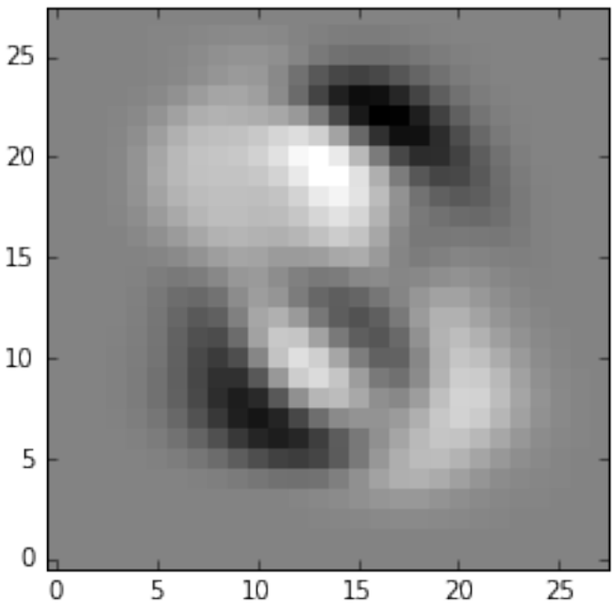
display the 28x28 image belonging to the largest eigenvector

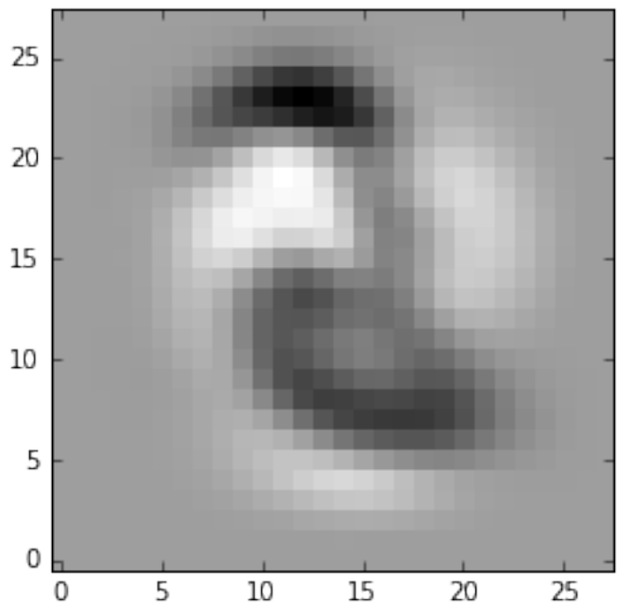


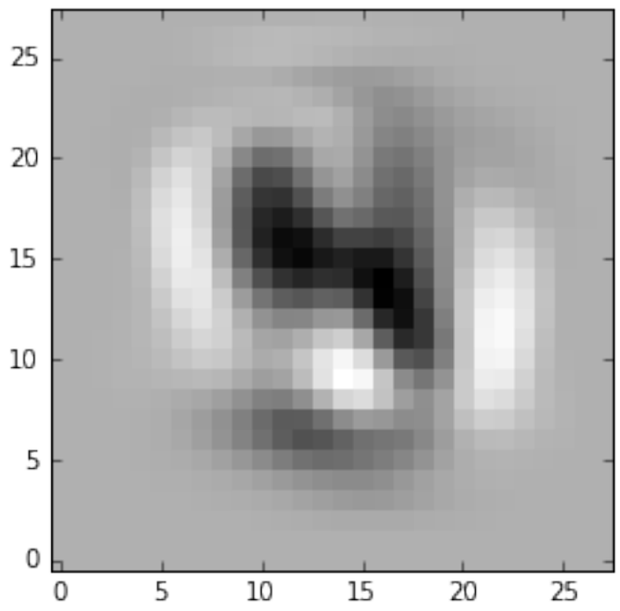
The first 6 PCA components (eigenvectors)

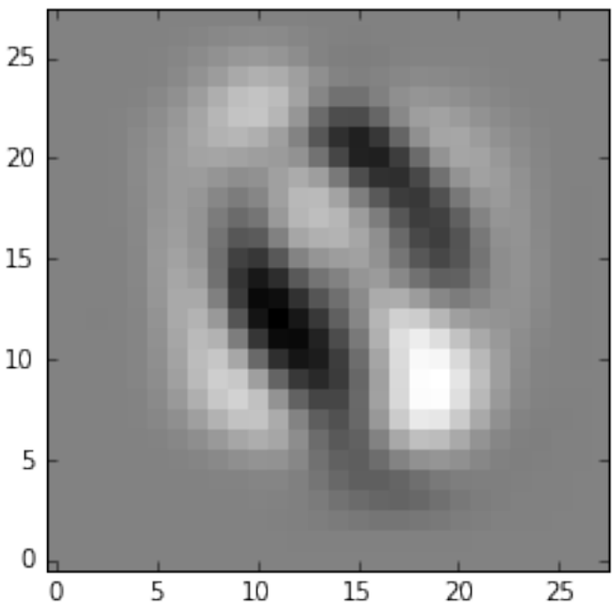






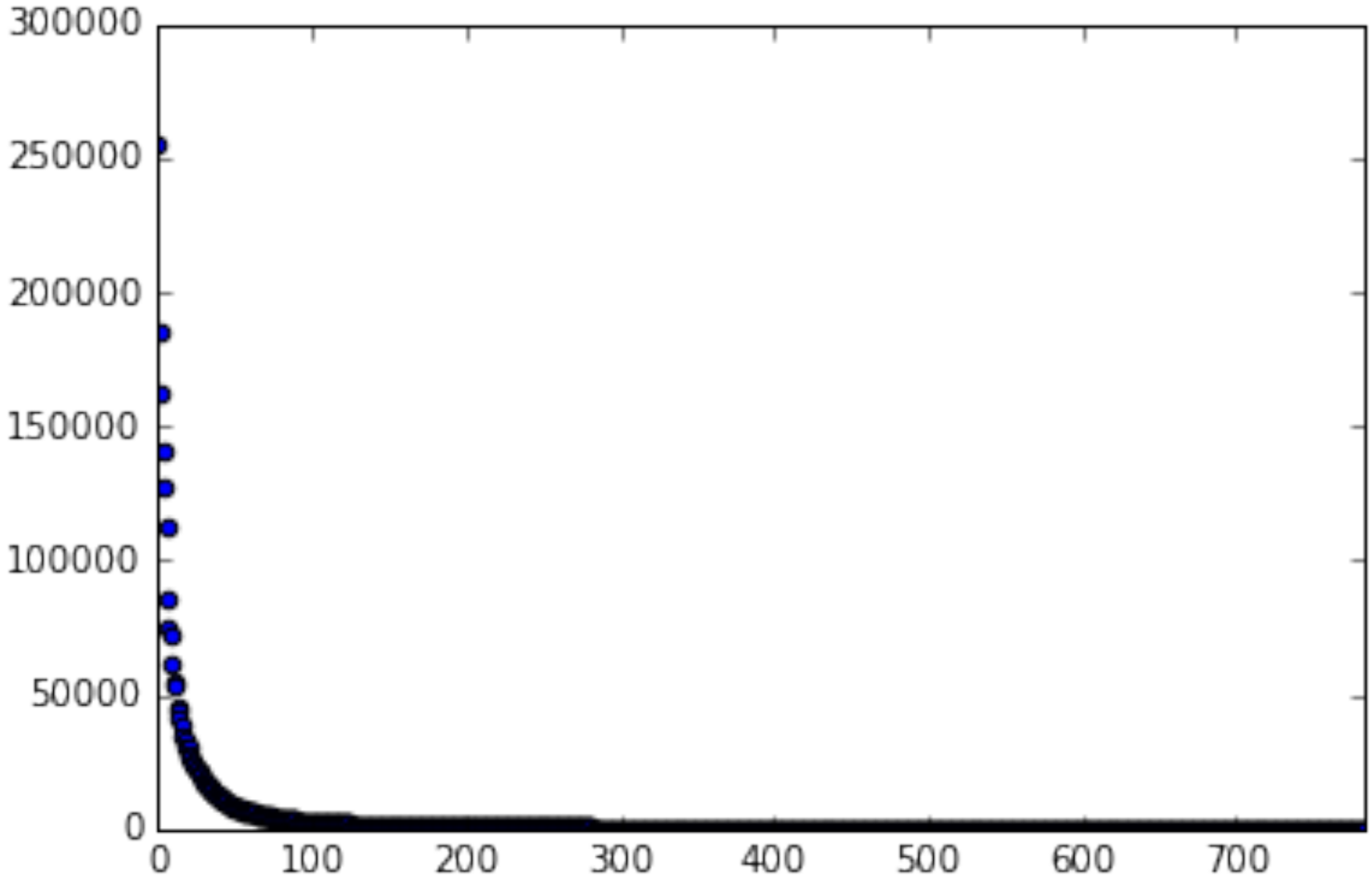






Can compress the information by projecting only on the first M largest components and then feeding that into a network

All the eigenvalues



The first 100 sum up to more than 90% of the total sum