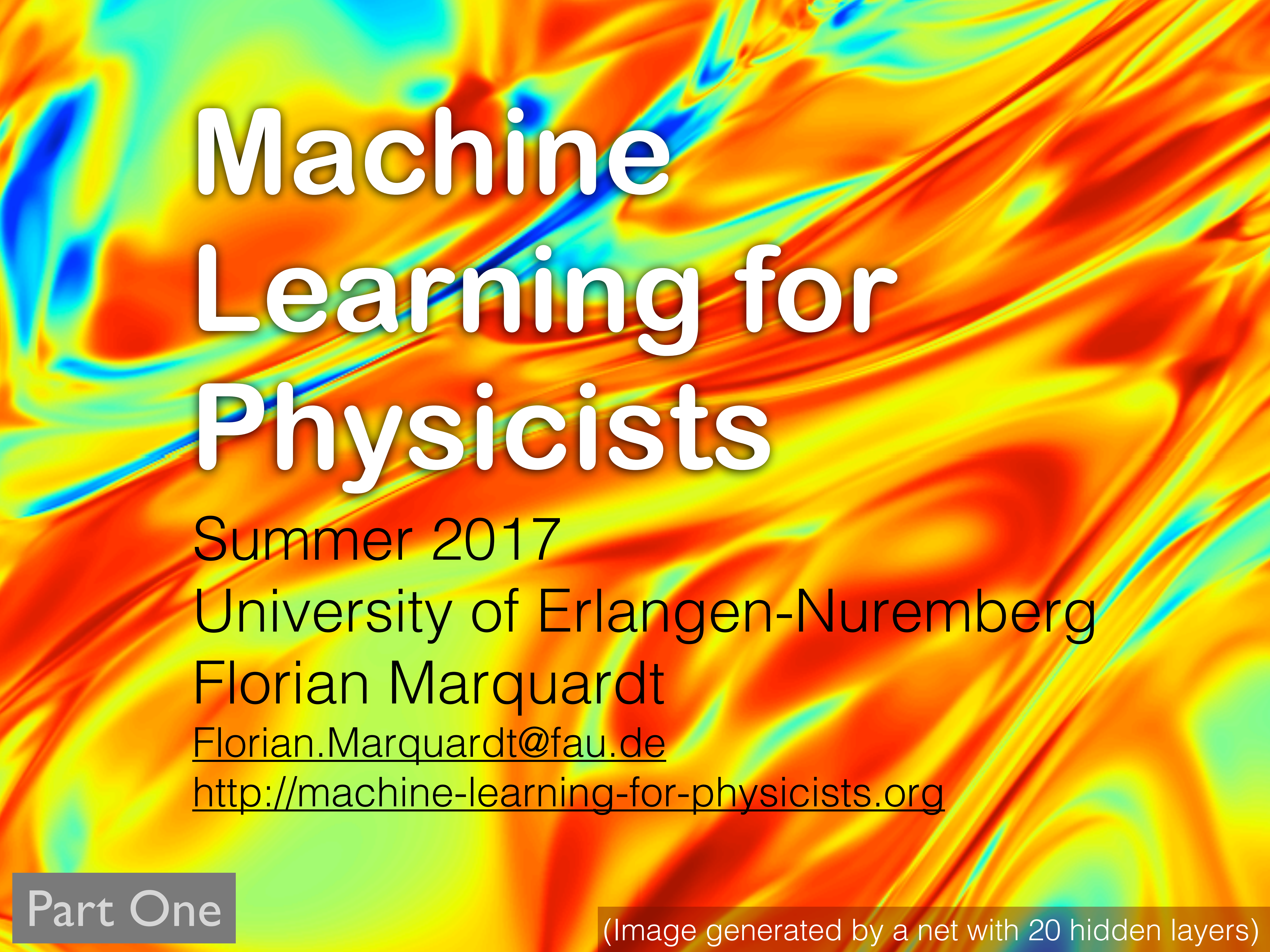
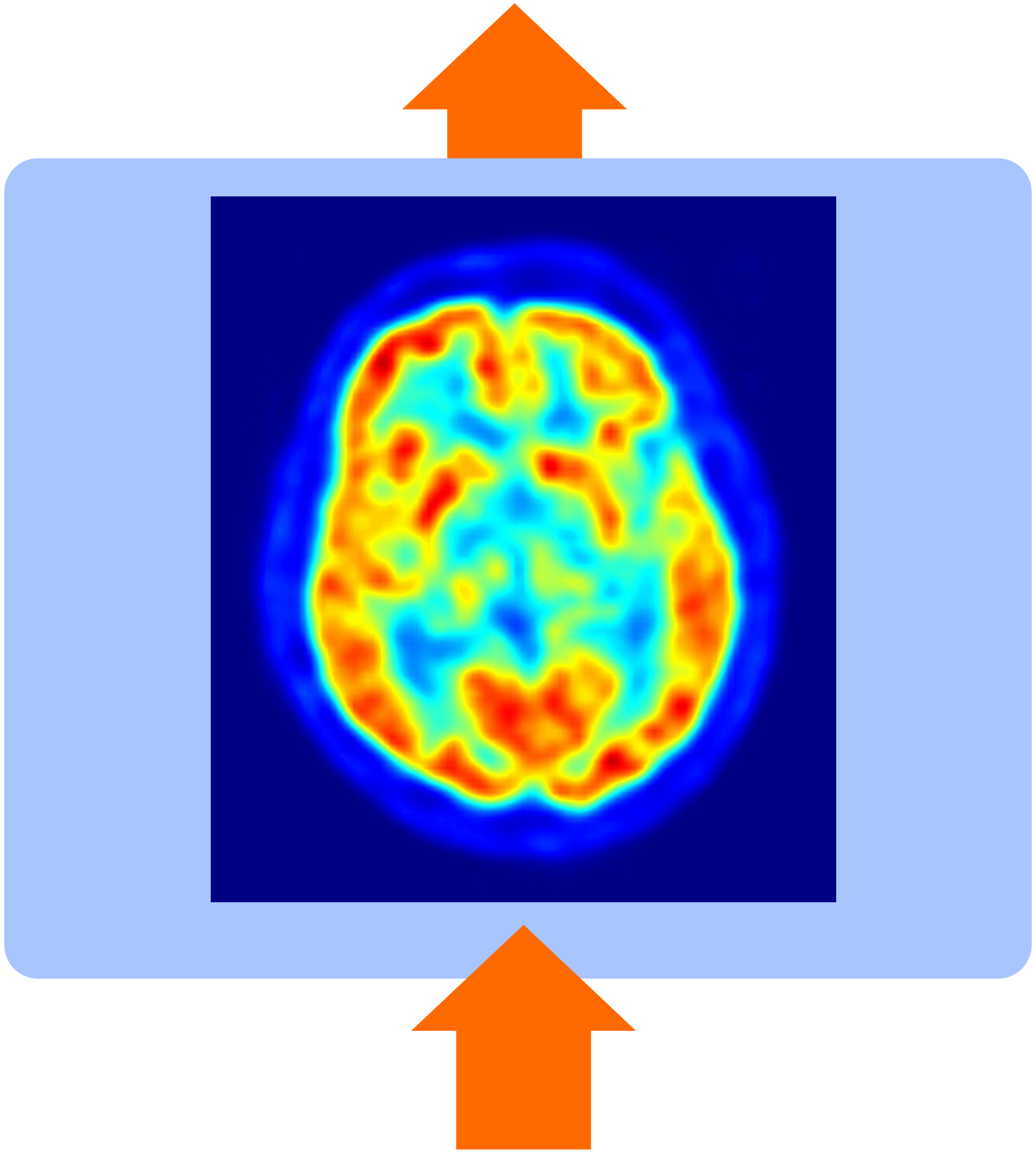


Machine Learning for Physicists

Summer 2017
University of Erlangen-Nuremberg
Florian Marquardt
Florian.Marquardt@fau.de
http://machine-learning-for-physicists.org

Part One

(Image generated by a net with 20 hidden layers)

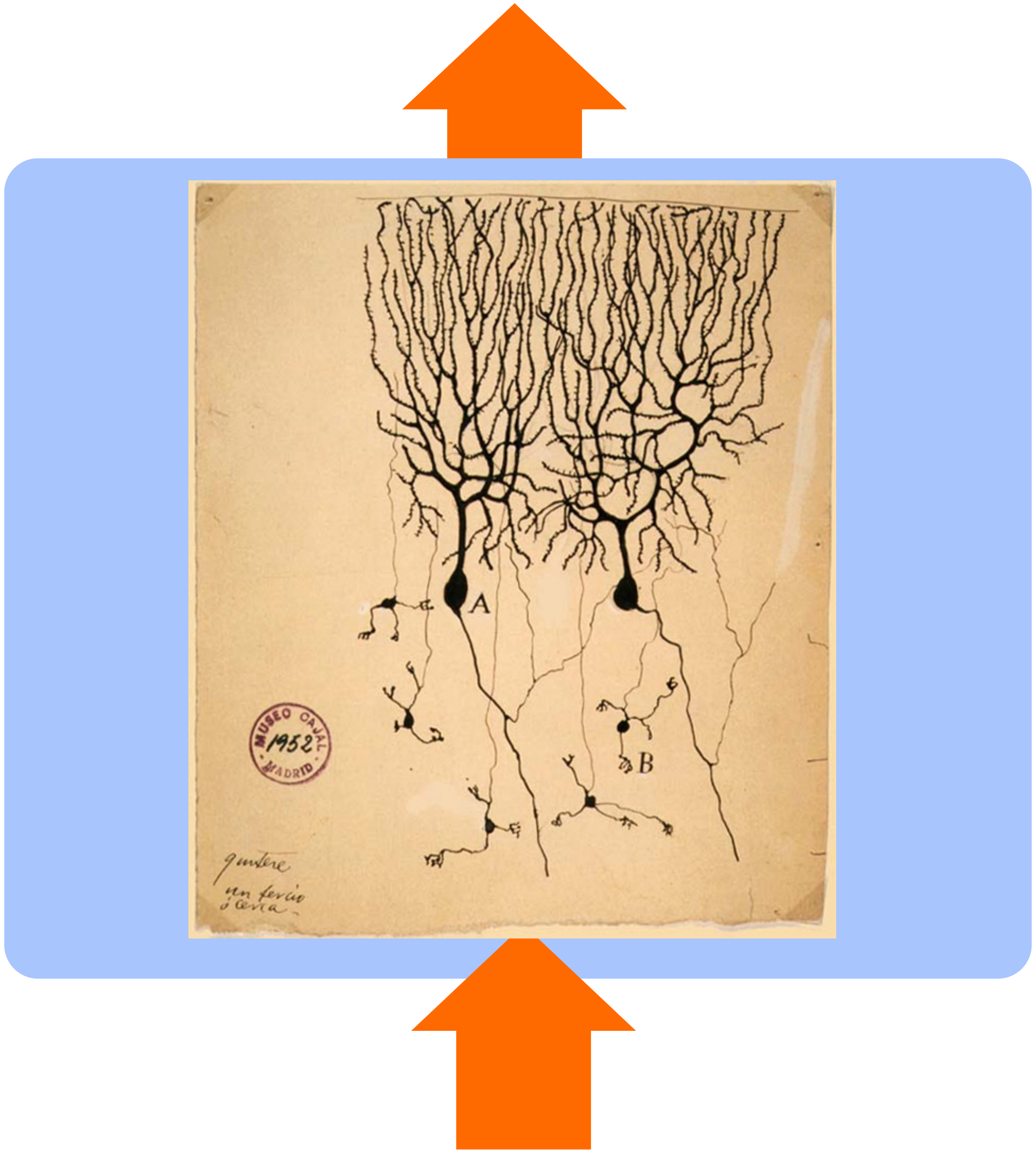


OUTPUT

INPUT

(Picture: Wikimedia Commons)

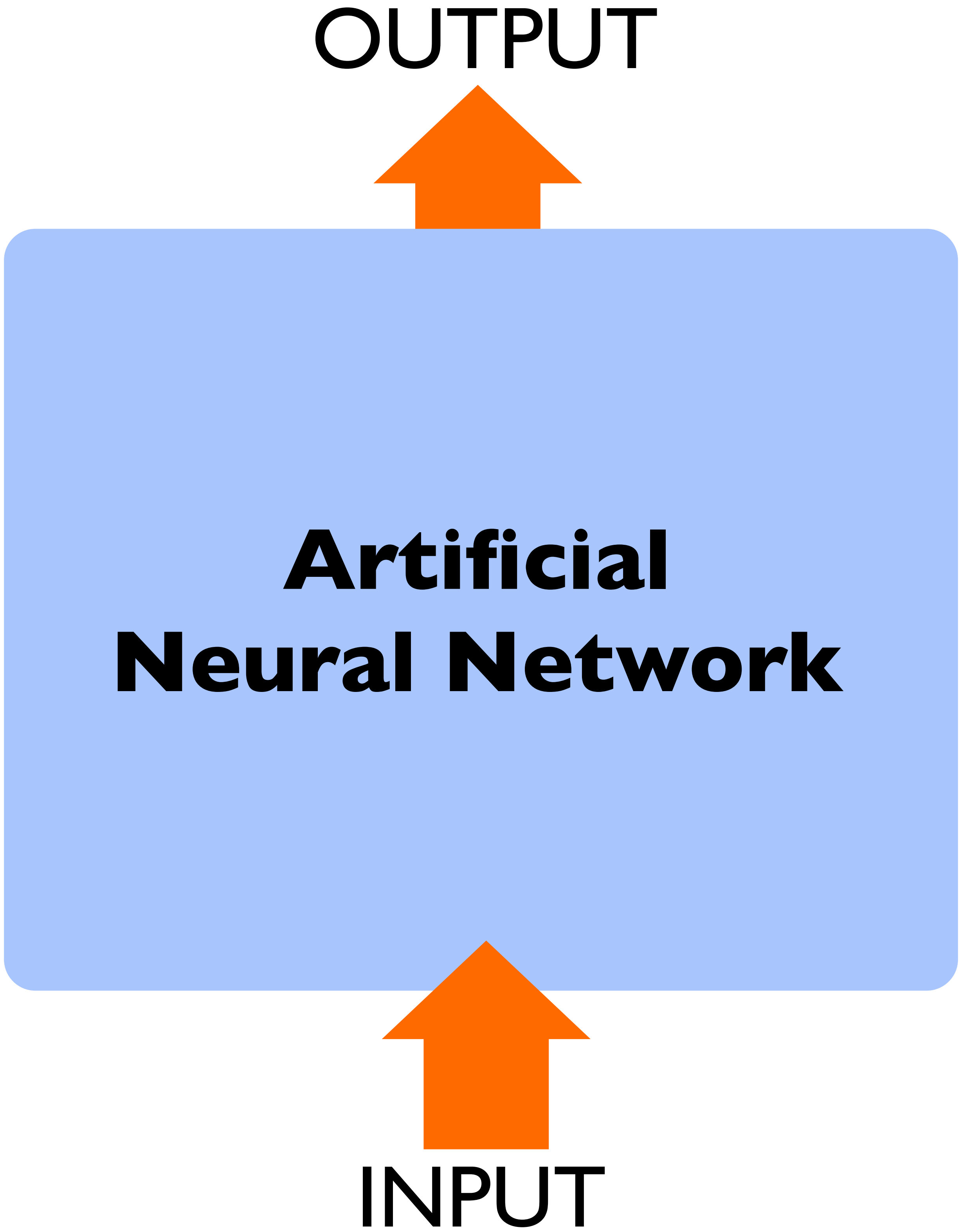
OUTPUT



INPUT

(drawing by Ramon y Cajal, ~1900)

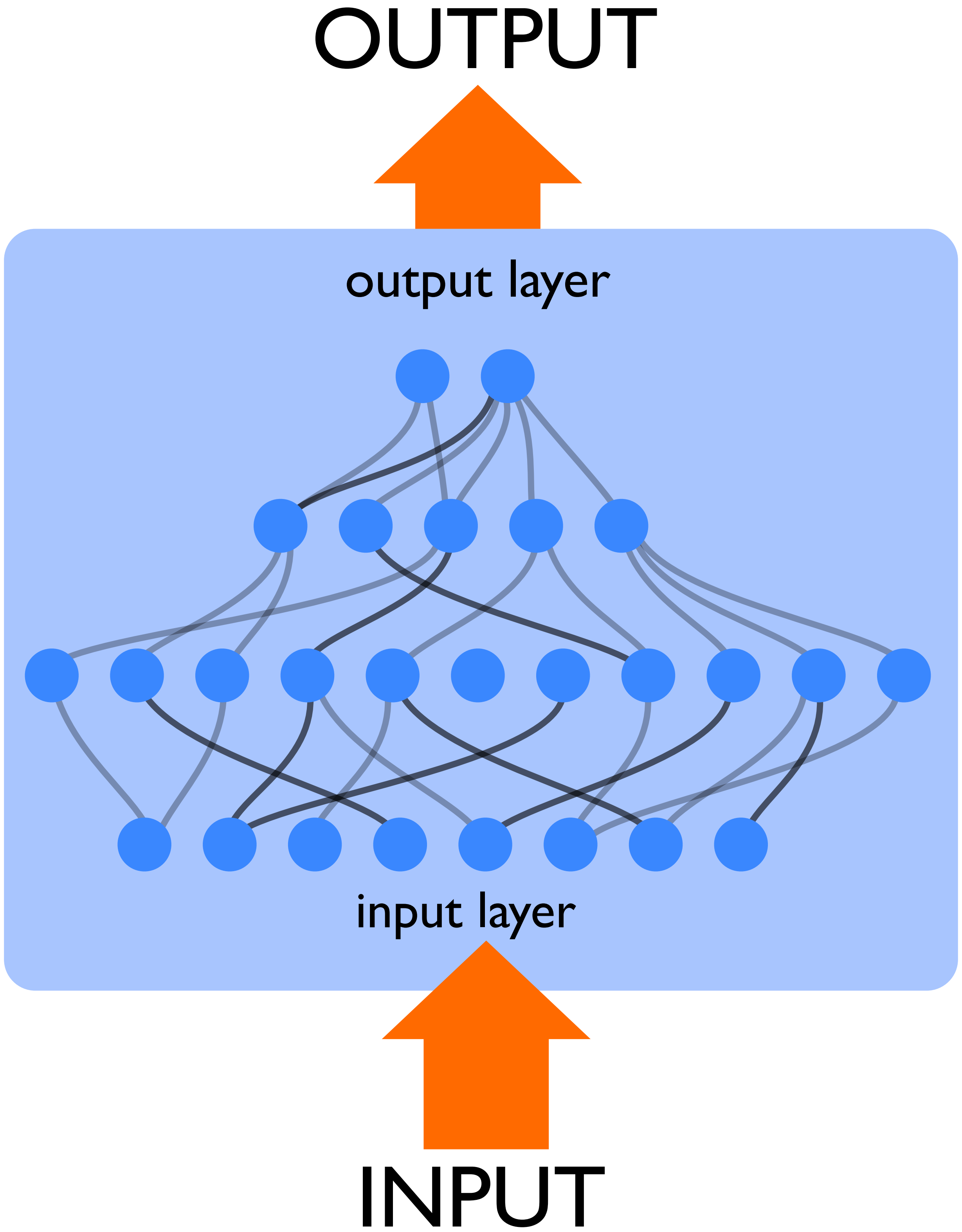
(Picture: Wikimedia Commons)



OUTPUT

Artificial Neural Network

INPUT



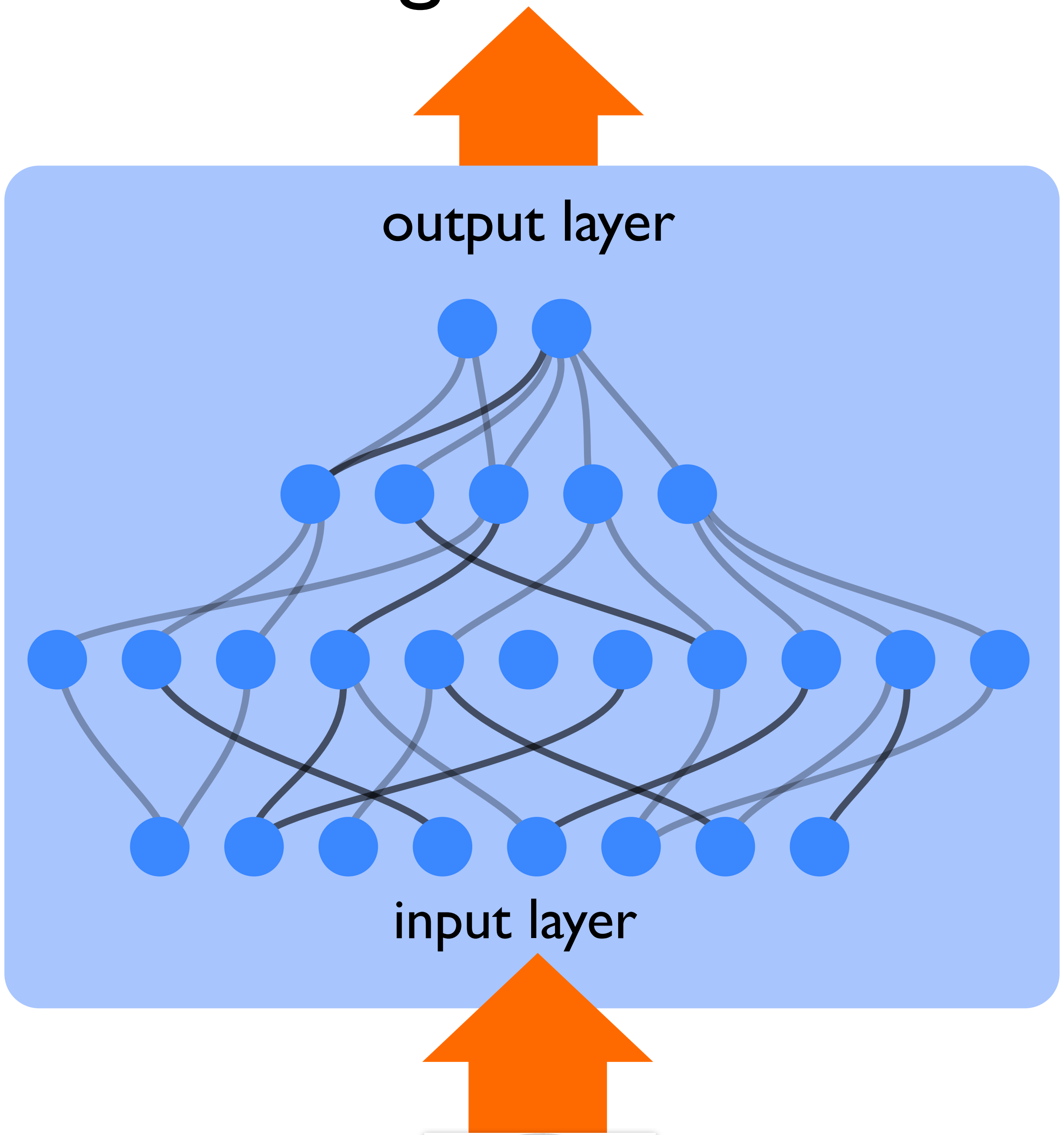
OUTPUT

output layer

input layer

INPUT

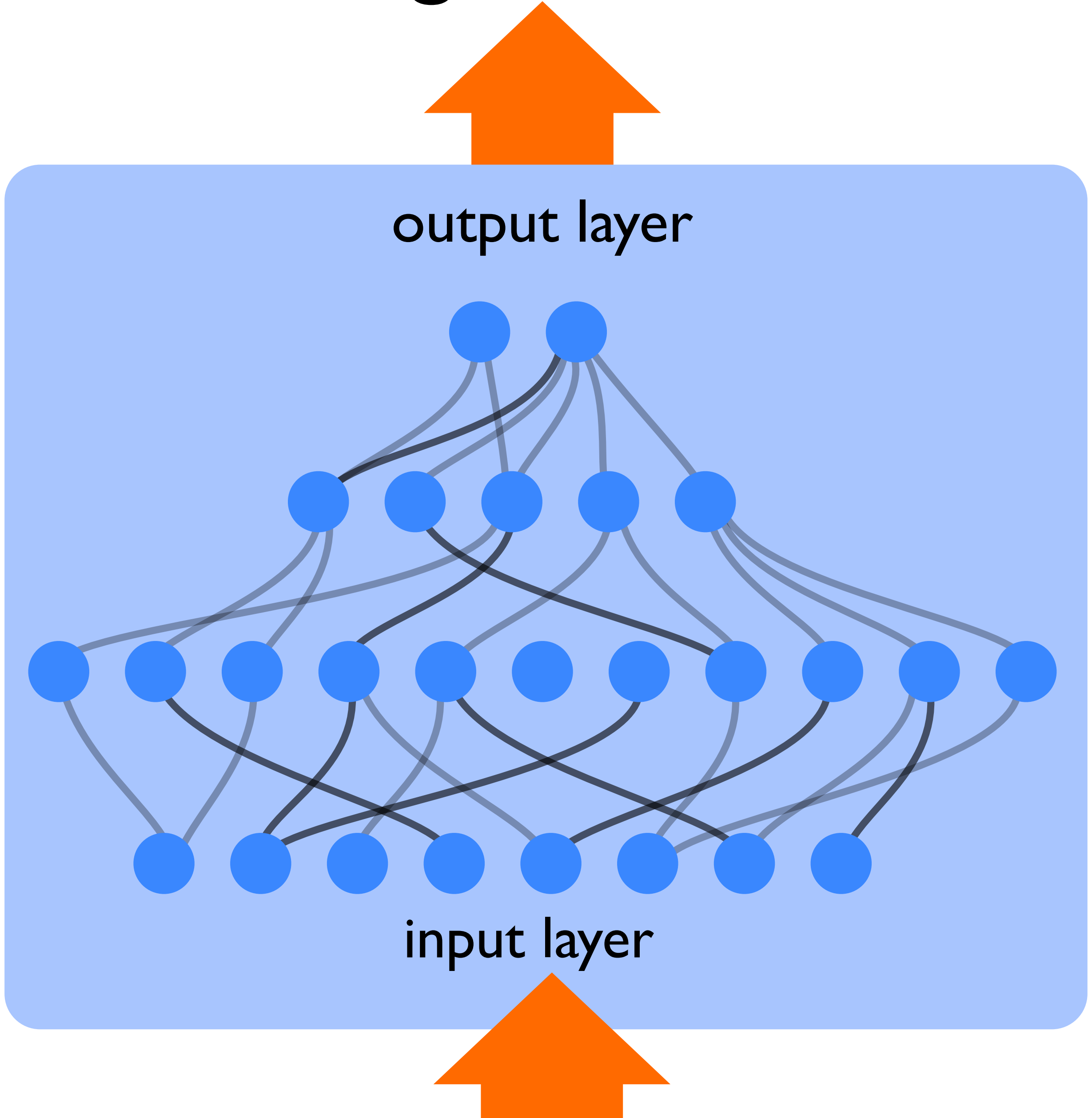
“light bulb”



(this particular picture has never been seen before!)

(Picture: Wikimedia Commons)

“light bulb”



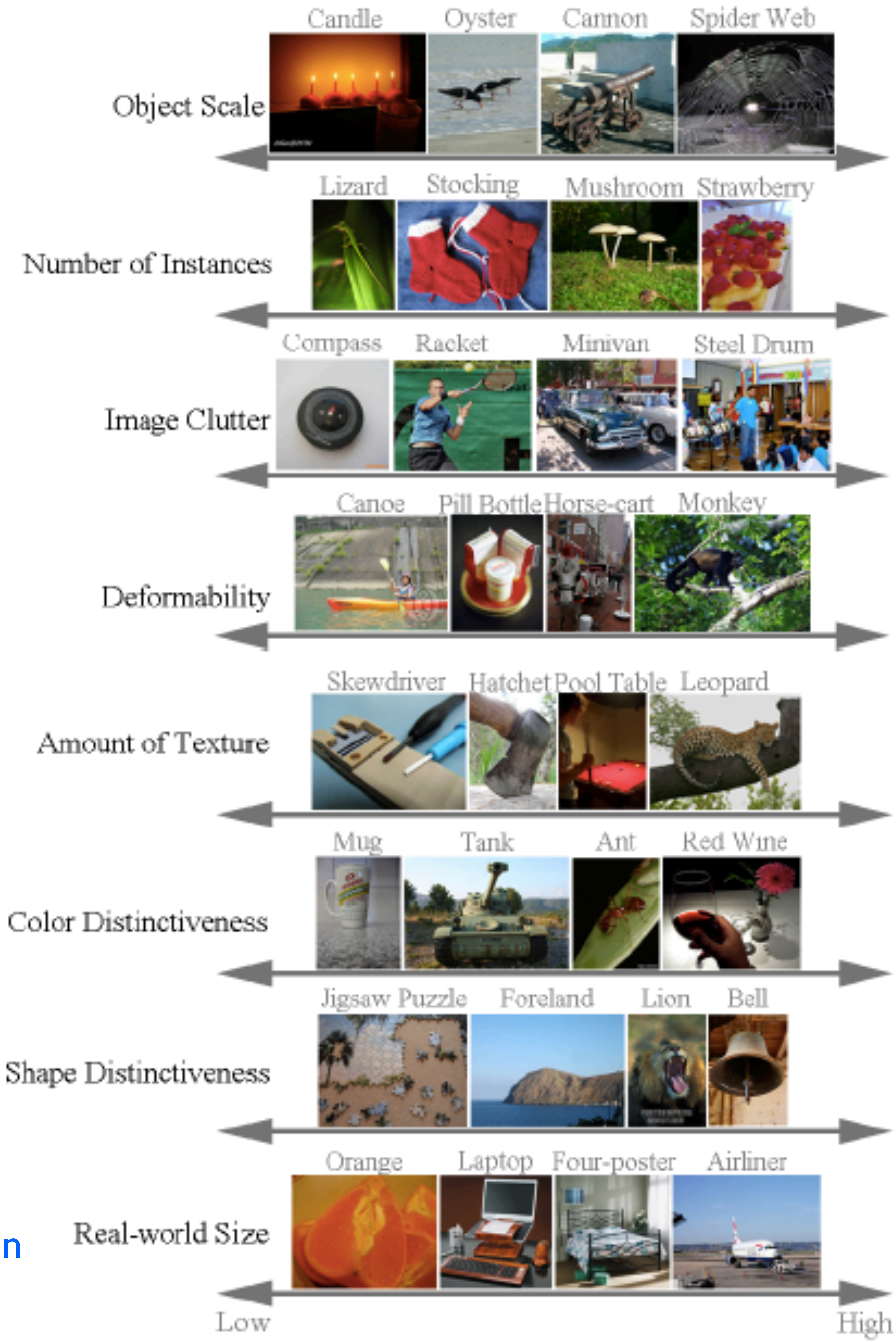
(training images)

(Picture: Wikimedia Commons)

ImageNet competition

1.2 million training pictures (annotated by humans)
1000 object classes

2012: A deep neural network beats competition clearly (16% error rate; since then rapid decrease of error rate, down to about 7%)



Picture: "ImageNet Large Scale Visual Recognition Challenge", Russakovsky et al. 2014

Example applications of (deep) neural networks

(see links on website)

e.g. http://machinelearningmastery.com/inspirational-applications-deep-learning/

Recognize images
Describe images in sentences
Colorize images
Translate languages (French to Spanish, etc.)
Answer questions about a brief text
Play video games & board games at superhuman level

(in physics:)
predict properties of materials
classify phases of matter
represent quantum wave functions

Lectures Outline

- Basic structure of artificial neural networks
- Training a network (backpropagation)
- Exploiting translational invariance in image processing (convolutional networks)
- Unsupervised learning of essential features (autoencoders)
- Learning temporal data, e.g. sentences (recurrent networks)
- Learning a probability distribution (Boltzmann machine)
- Learning from rare rewards (reinforcement learning)
- Further tricks and concepts
- Modern applications to physics and science

Learning by doing!

Lectur

- Basic structur neural networks
- Training a ackpropagation)

Python

- Expl ational invariance in image processing
  nal networks)
- pervised learning of essential featu
  utoencoders)
- Learning temporal data, e.g. sentences (
  networks)
- Learning a probability distribution (Boltzmann machine)
- Learning from rare rewards (reinforcement learning)

Keras package for Python

- Further tricks and concepts
- Modern applications to physics and science

Homework

Homework: (usually) explore via programming

We provide feedback if desired

No regular tutorial sessions

Original site:
http://www.thp2.nat.uni-erlangen.de/index.php/2017_Machine_Learning_for_Physicists,_by_Florian_Marquardt

New site:
http://machine-learning-for-physicists.org

Homework

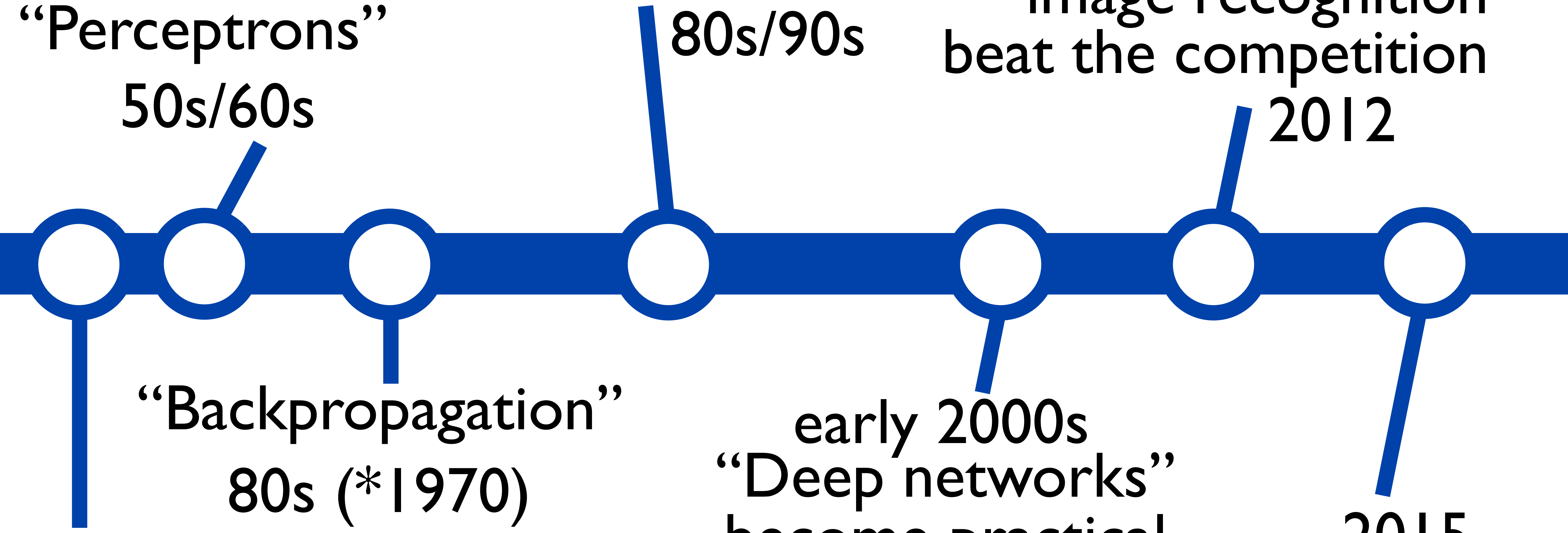
First homework:

1. Install python & keras on your computer (see lecture homepage); questions will be resolved after second lecture THIS IS IMPORTANT!
2. Brainstorm: “Which problems could you address using neural networks?”

Next time: “installation party” after the lecture

Bring your laptop, if available (or ask questions)

Very brief history of artificial neural networks



1956 Dartmouth
Workshop on
Artificial Intelligence

“Perceptrons”
50s/60s

“Backpropagation”
80s (*1970)

“Recurrent networks”
“Convolutional networks”
80s/90s

early 2000s
“Deep networks”
become practical

Deep nets for
image recognition
beat the competition
2012

2015
A deep net
reaches expert level in “Go”

Lots of tutorials/info on the web...

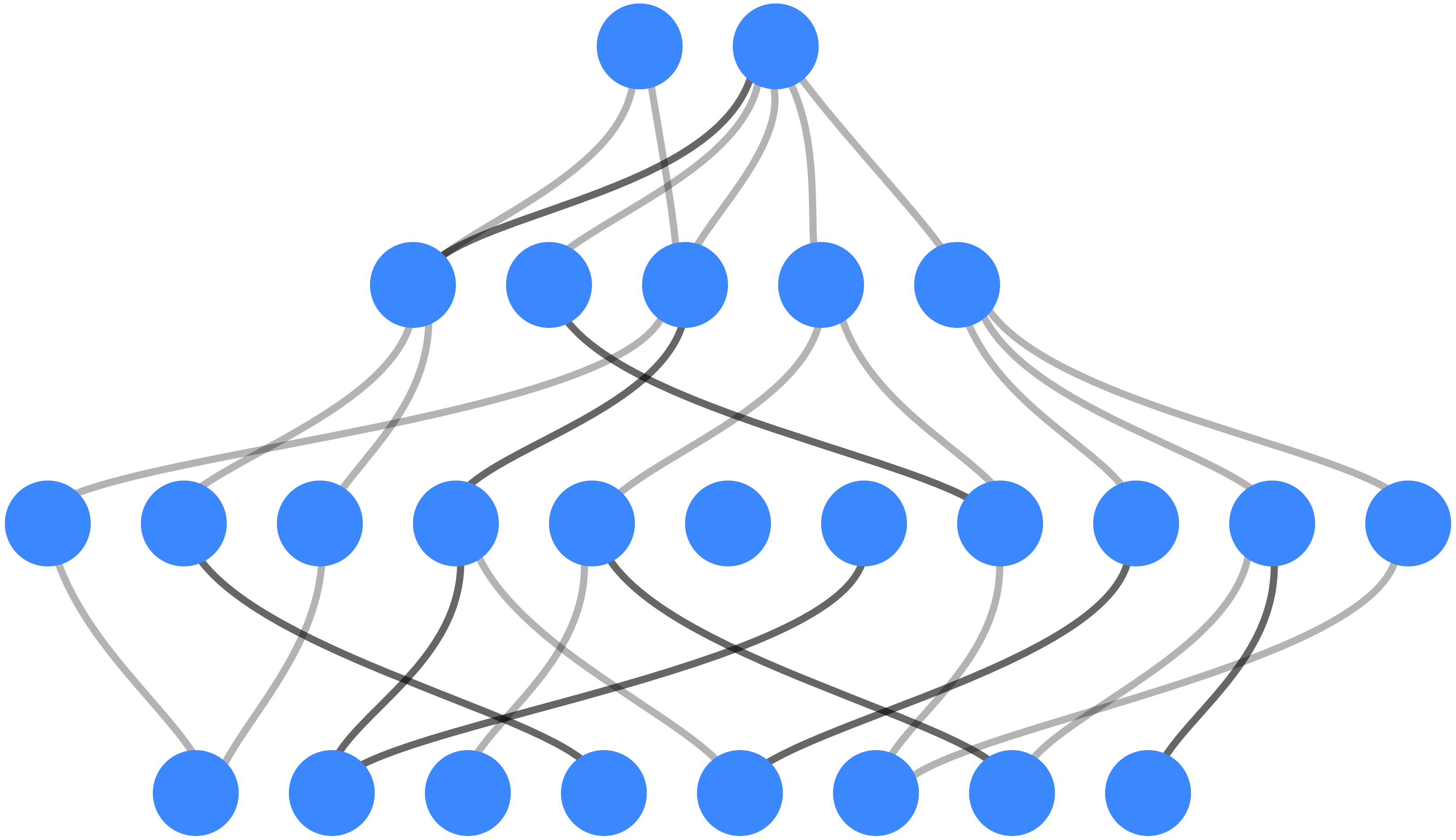
recommend:
online book by Nielsen ("**Neural Networks and Deep Learning**") at https://neuralnetworksanddeeplearning.com

much more detailed book:
"**Deep Learning**" by Goodfellow, Bengio, Courville; MIT press; see also http://www.deeplearningbook.org

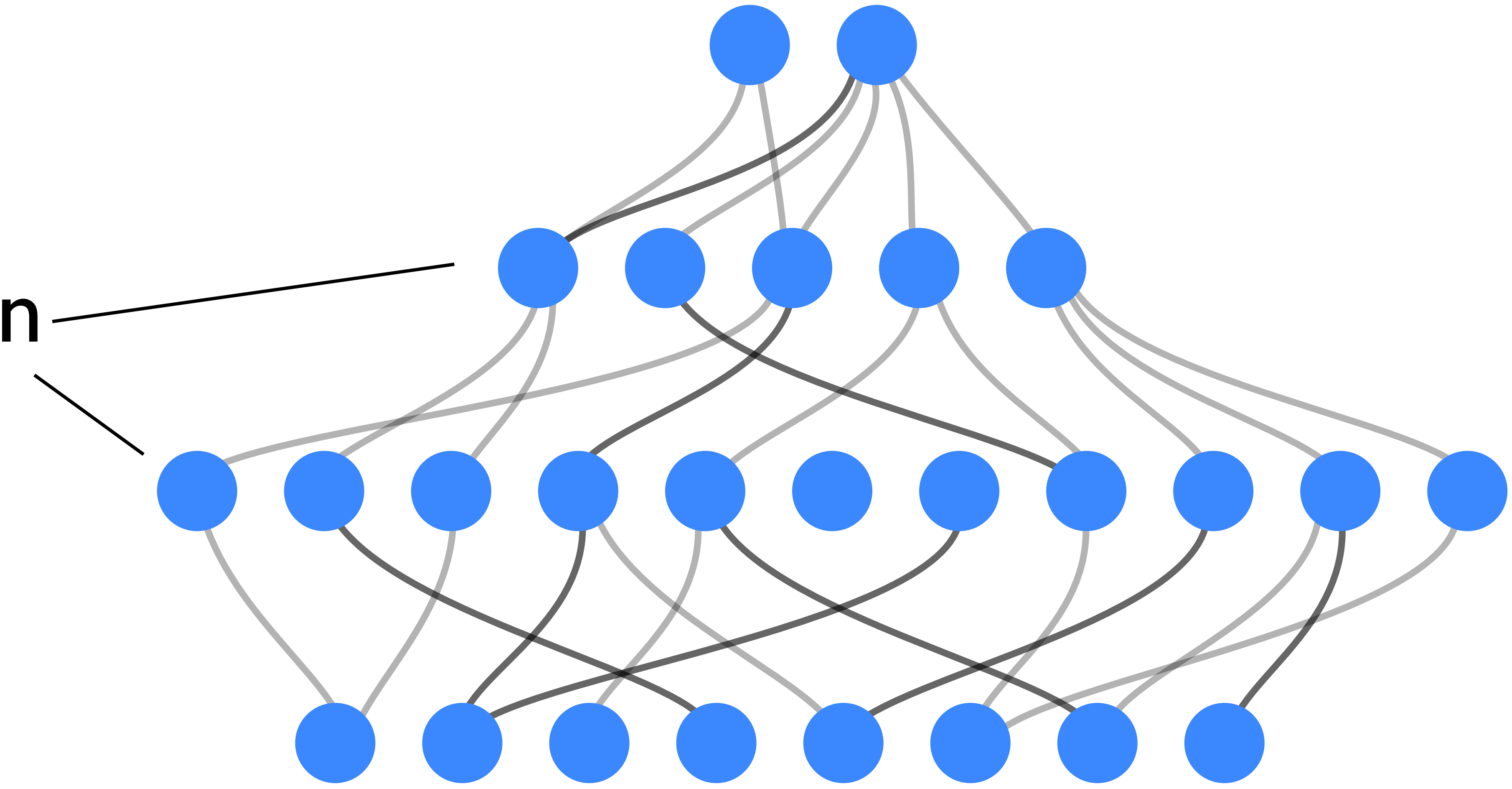
Software –
here: python & keras (builds on theano)

A neural network

= a nonlinear function (of many variables) that depends on many parameters



A neural network

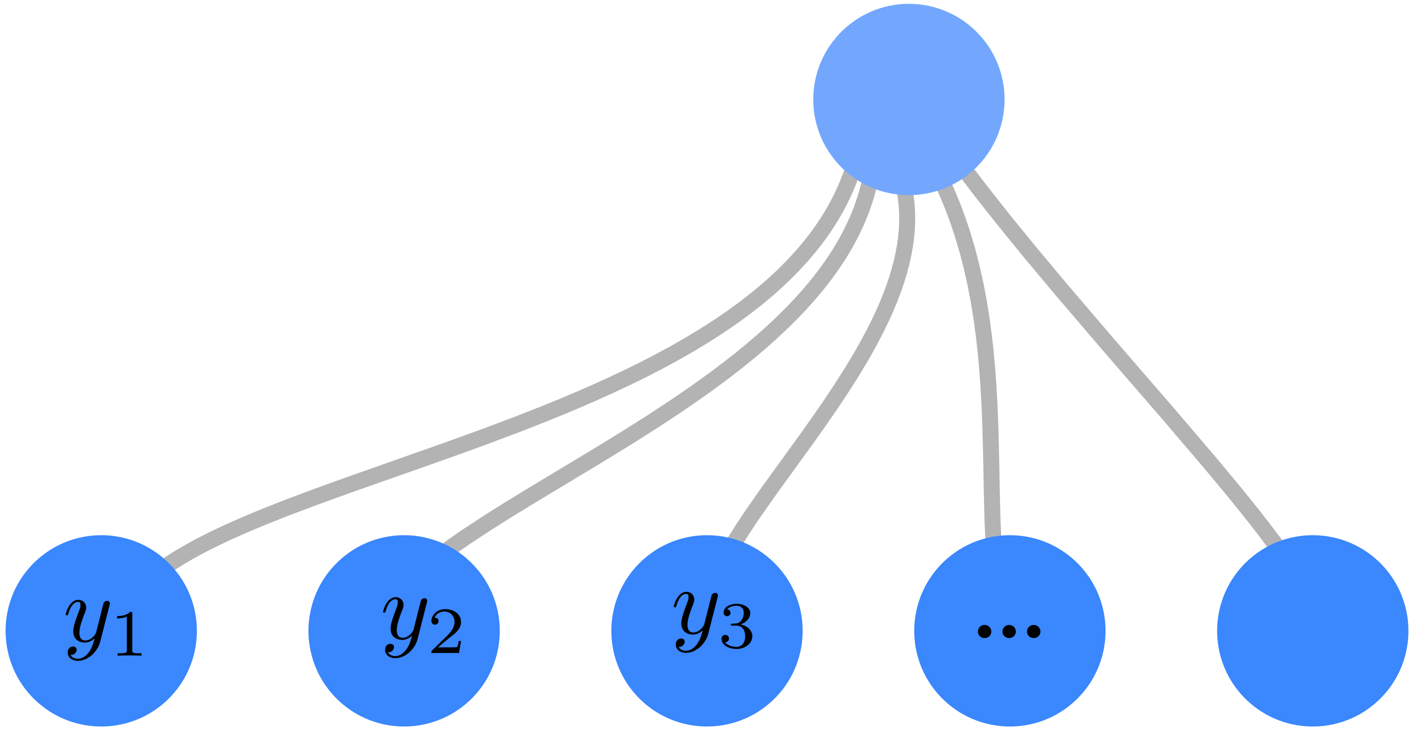


output layer

hidden layers

input layer

output of a neuron =
nonlinear function of
weighted sum of inputs



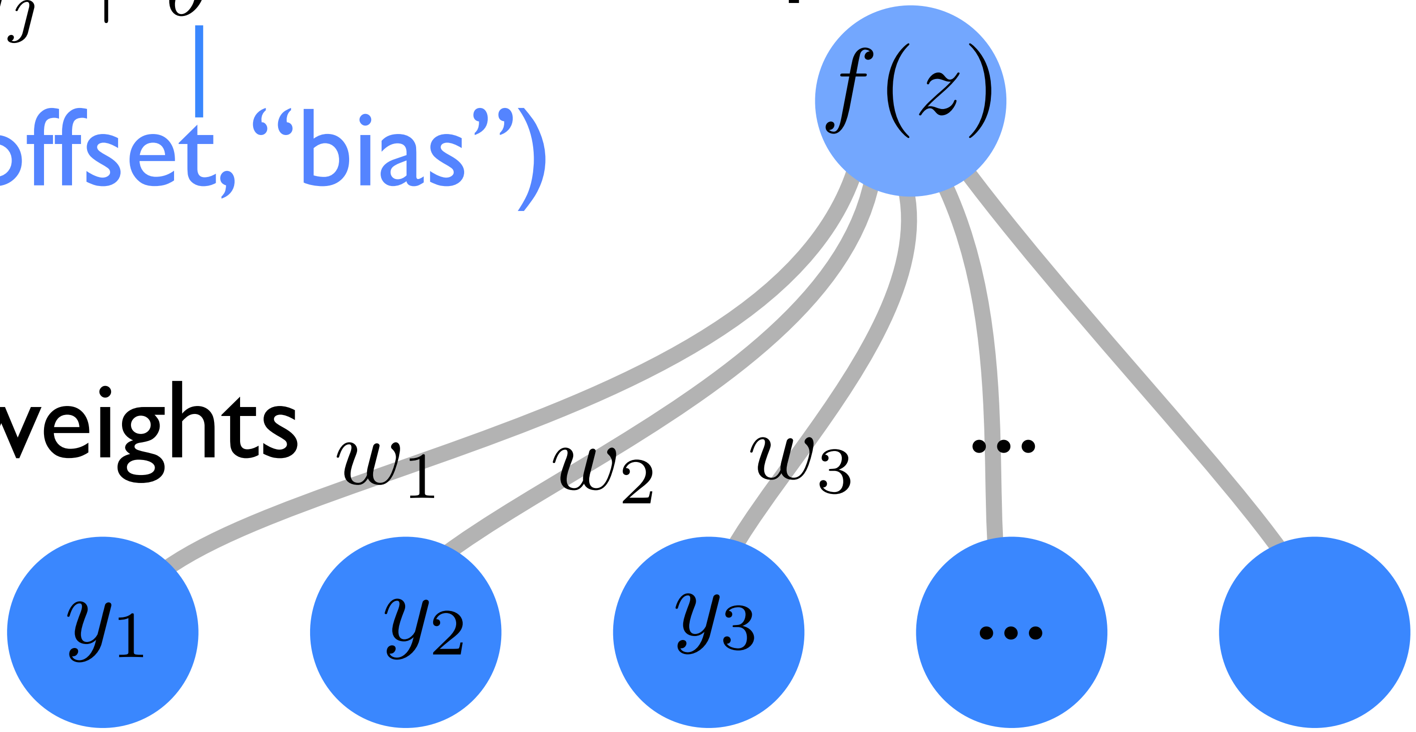
input values

output of a neuron =
nonlinear function of
weighted sum of inputs

weighted sum

$$z = \sum_j w_j y_j + b$$

(offset, "bias")



output value

$f(z)$

weights w_1 w_2 w_3 ...

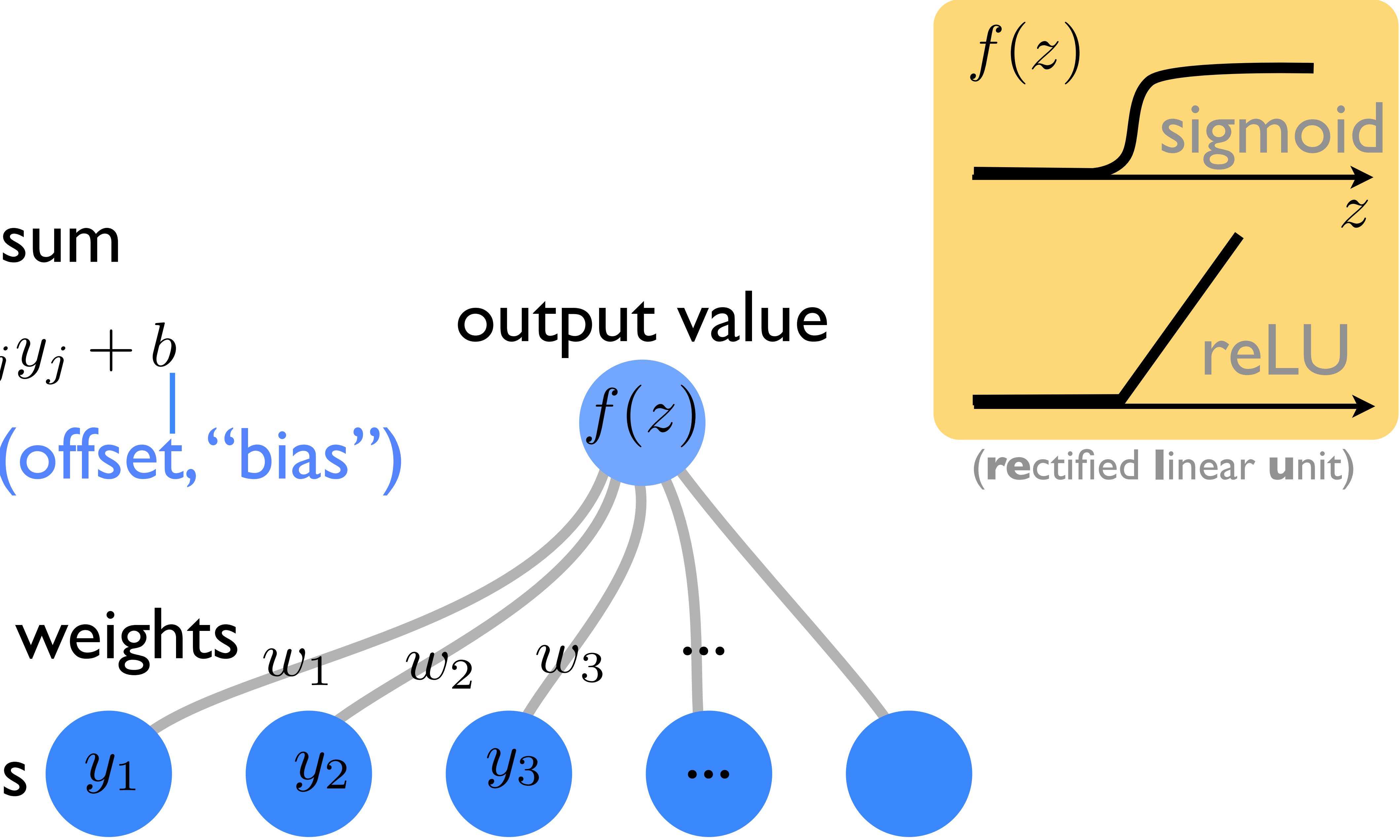
input values y_1 y_2 y_3 ...

output of a neuron = nonlinear function of weighted sum of inputs

weighted sum

$$z = \sum_j w_j y_j + b$$

(offset, “bias”)



output value

$f(z)$

weights

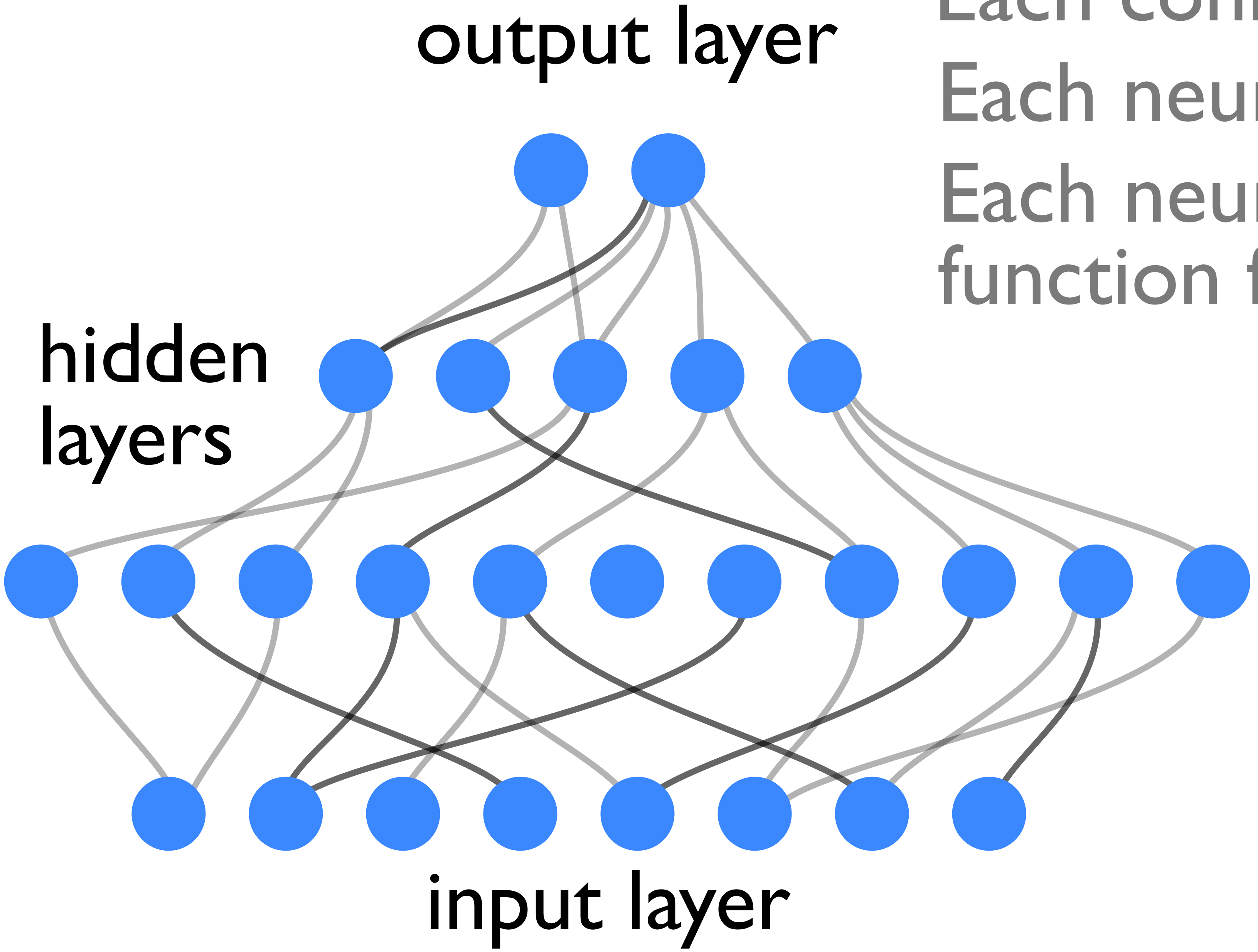
w_1 w_2 w_3 ...

input values

y_1 y_2 y_3 ...

(**re**ctified **l**inear **u**nit)

A neural network

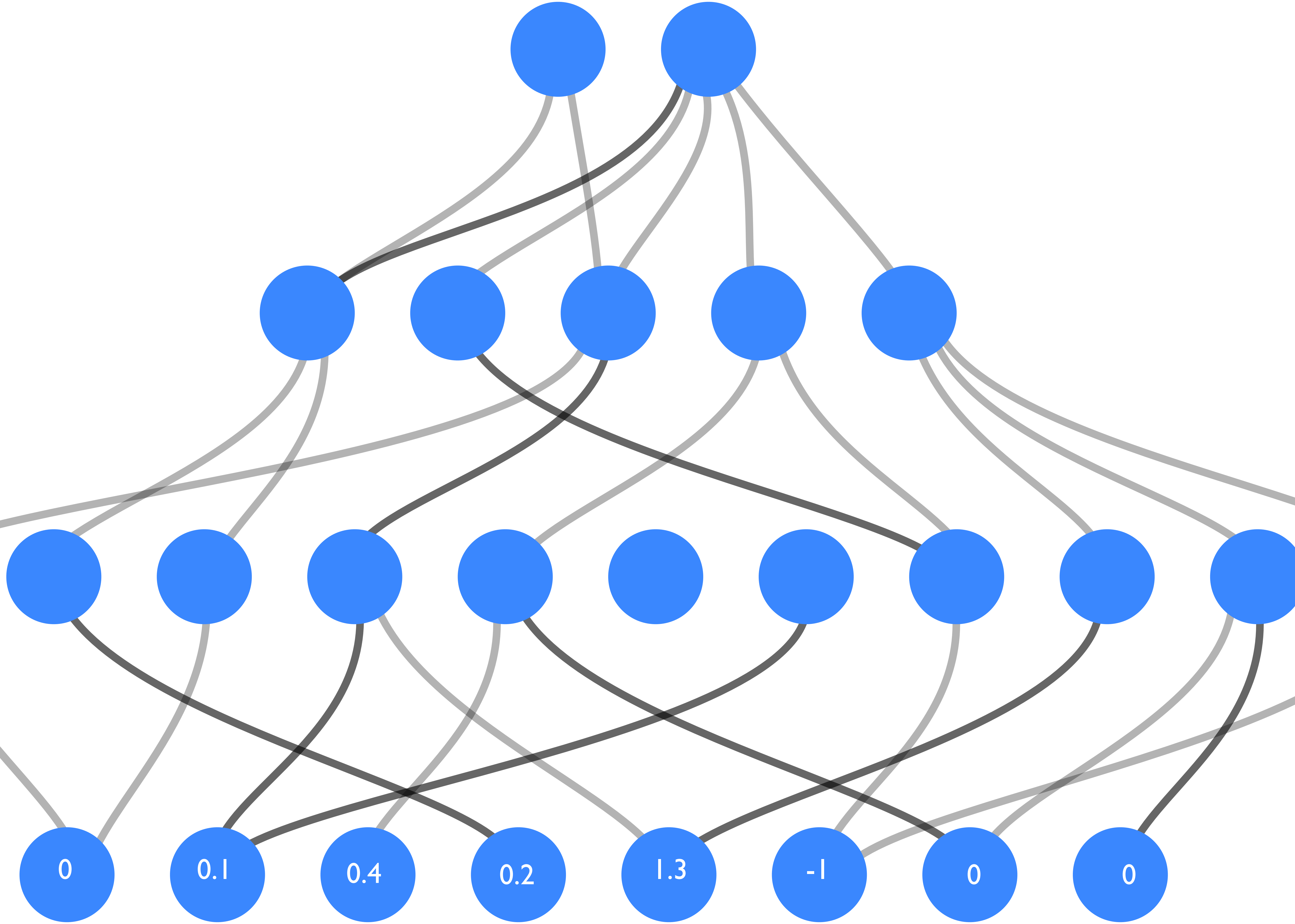


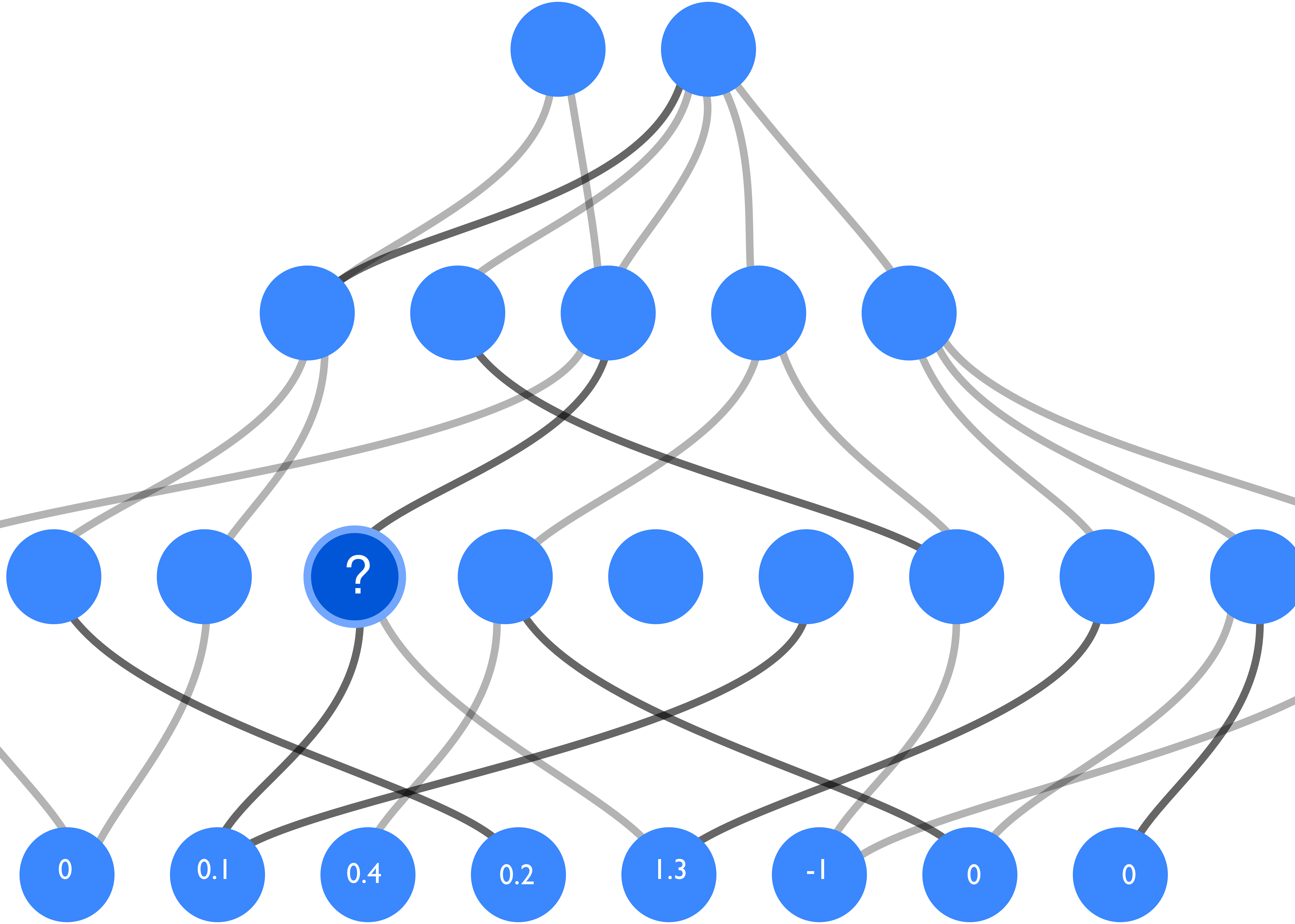
Each connection has a weight w

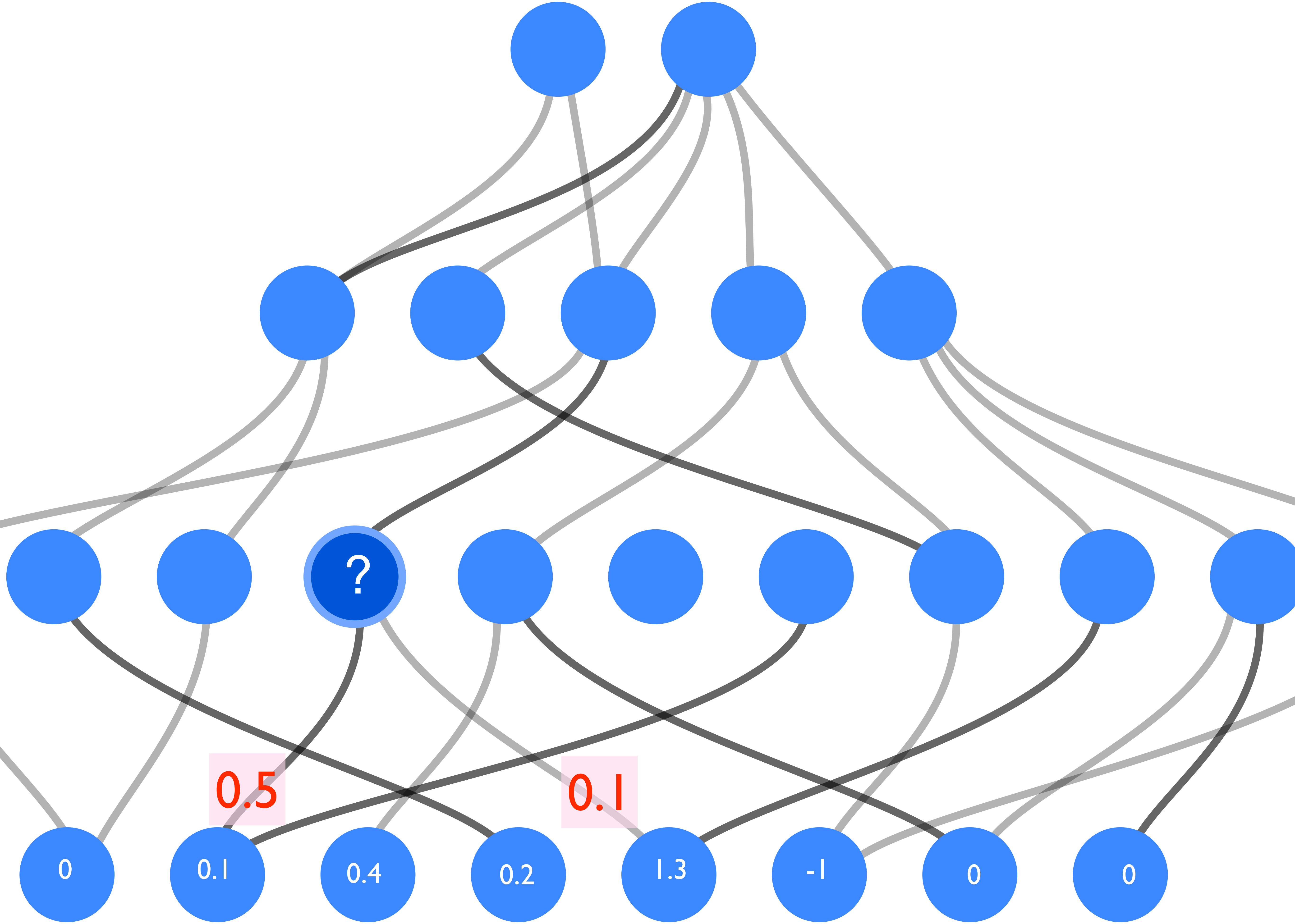
Each neuron has an offset b

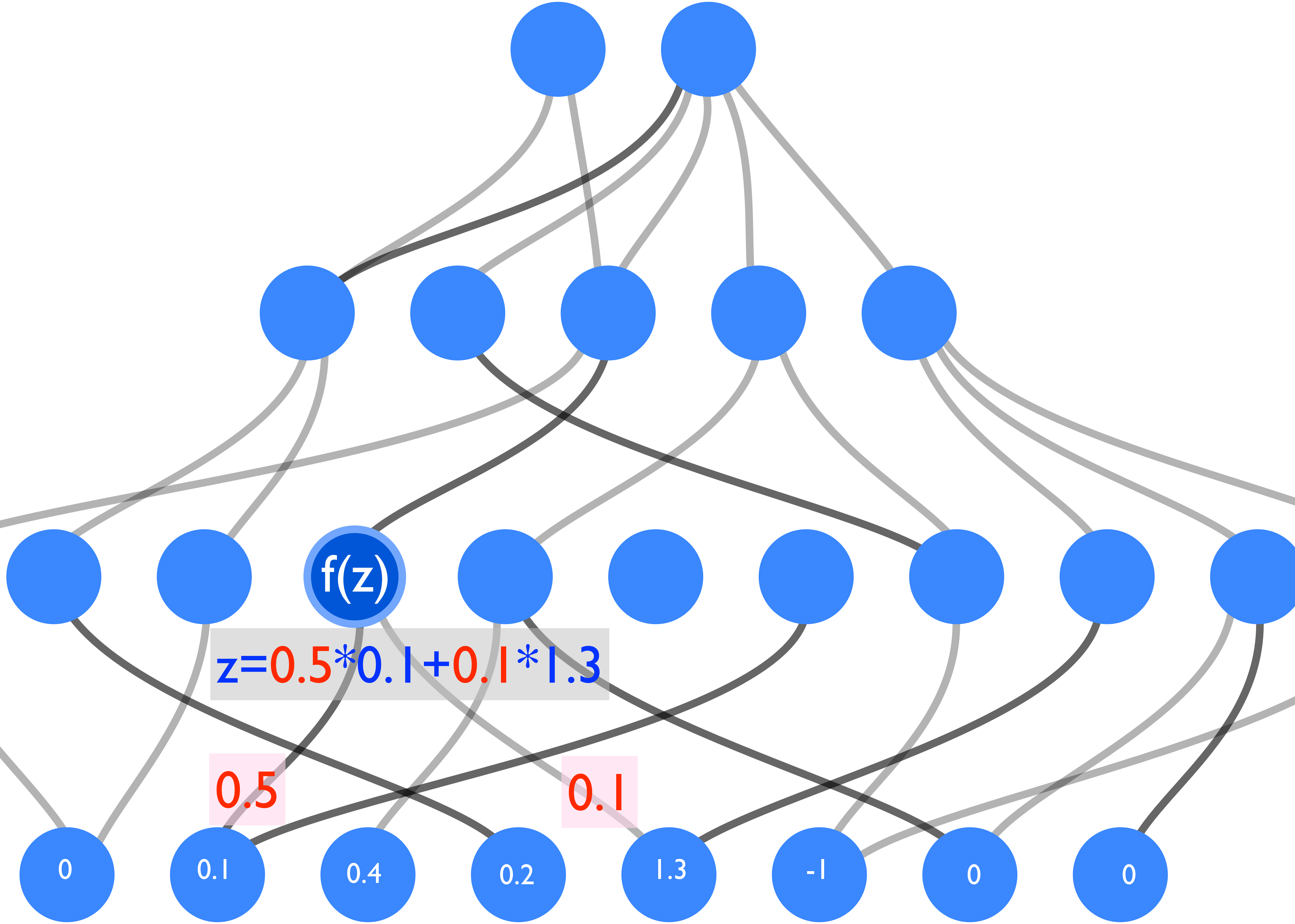
Each neuron has a nonlinear function f (fixed)

The values of input layer neurons are fed into the network from the outside


0
0.1
0.4
0.2
1.3
-1
0
0

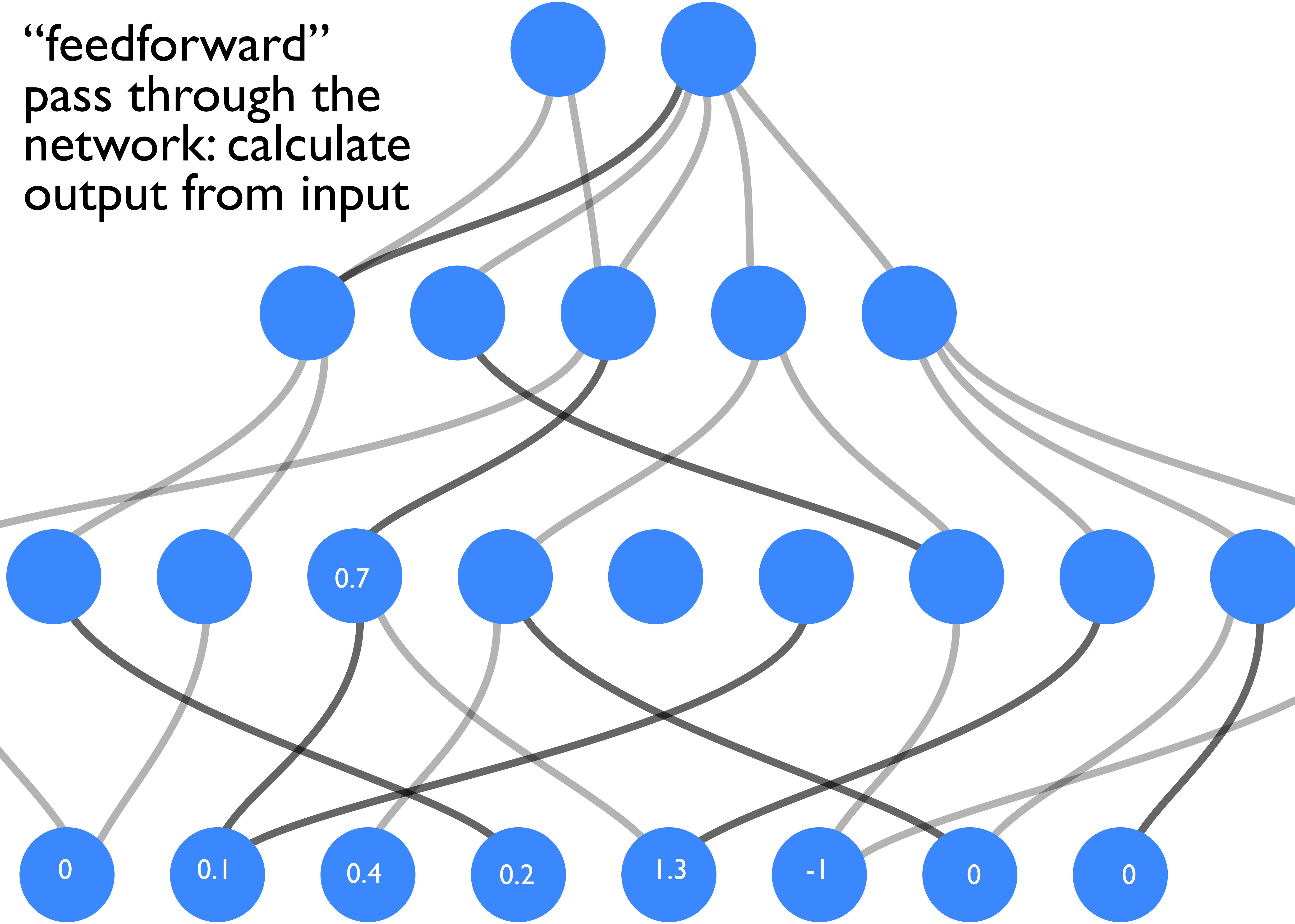

?
0
0.1
0.4
0.2
1.3
-1
0
0


?
0.5
0.1
0
0.1
0.4
0.2
1.3
-1
0
0

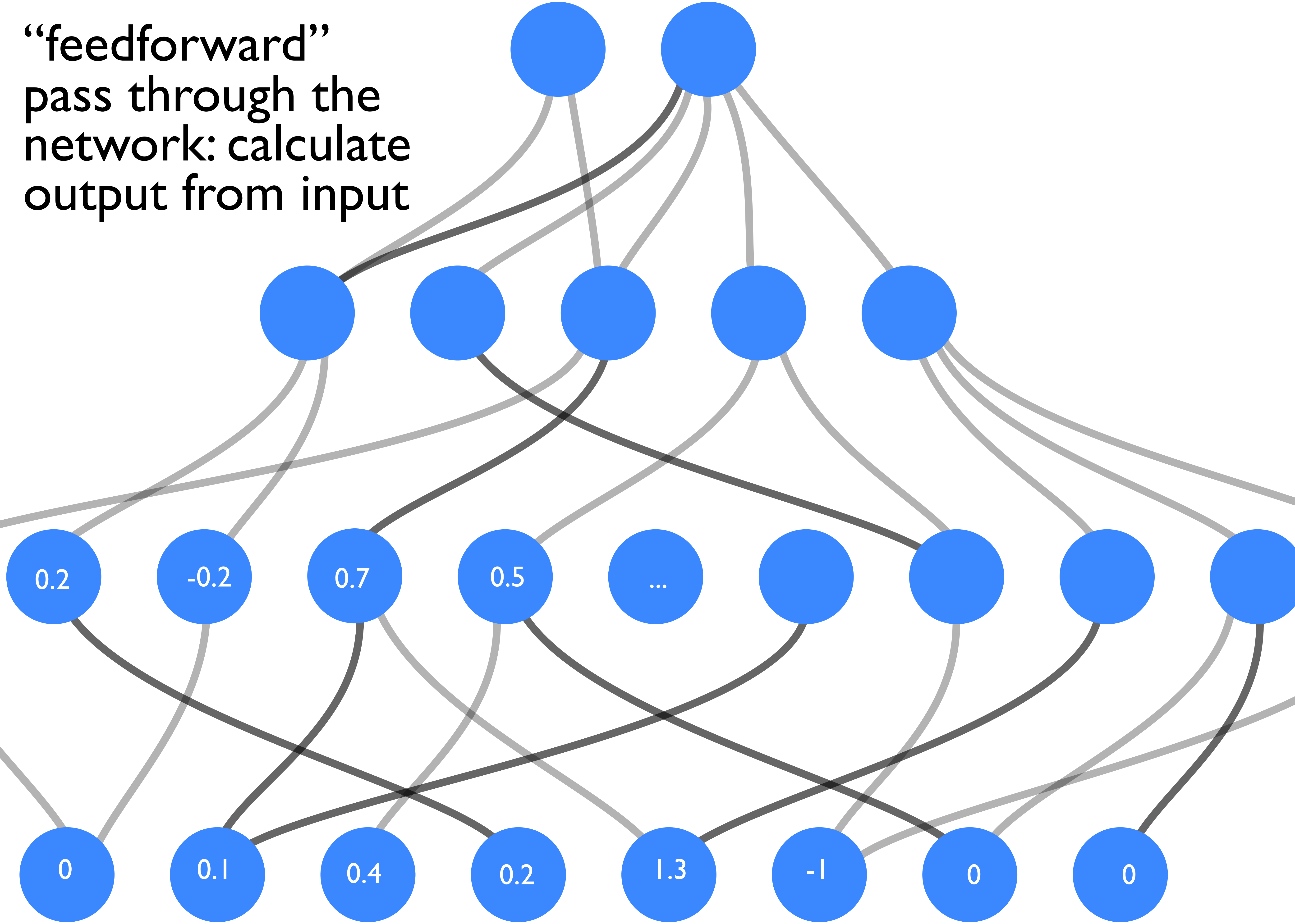


$z=0.5*0.1+0.1*1.3$

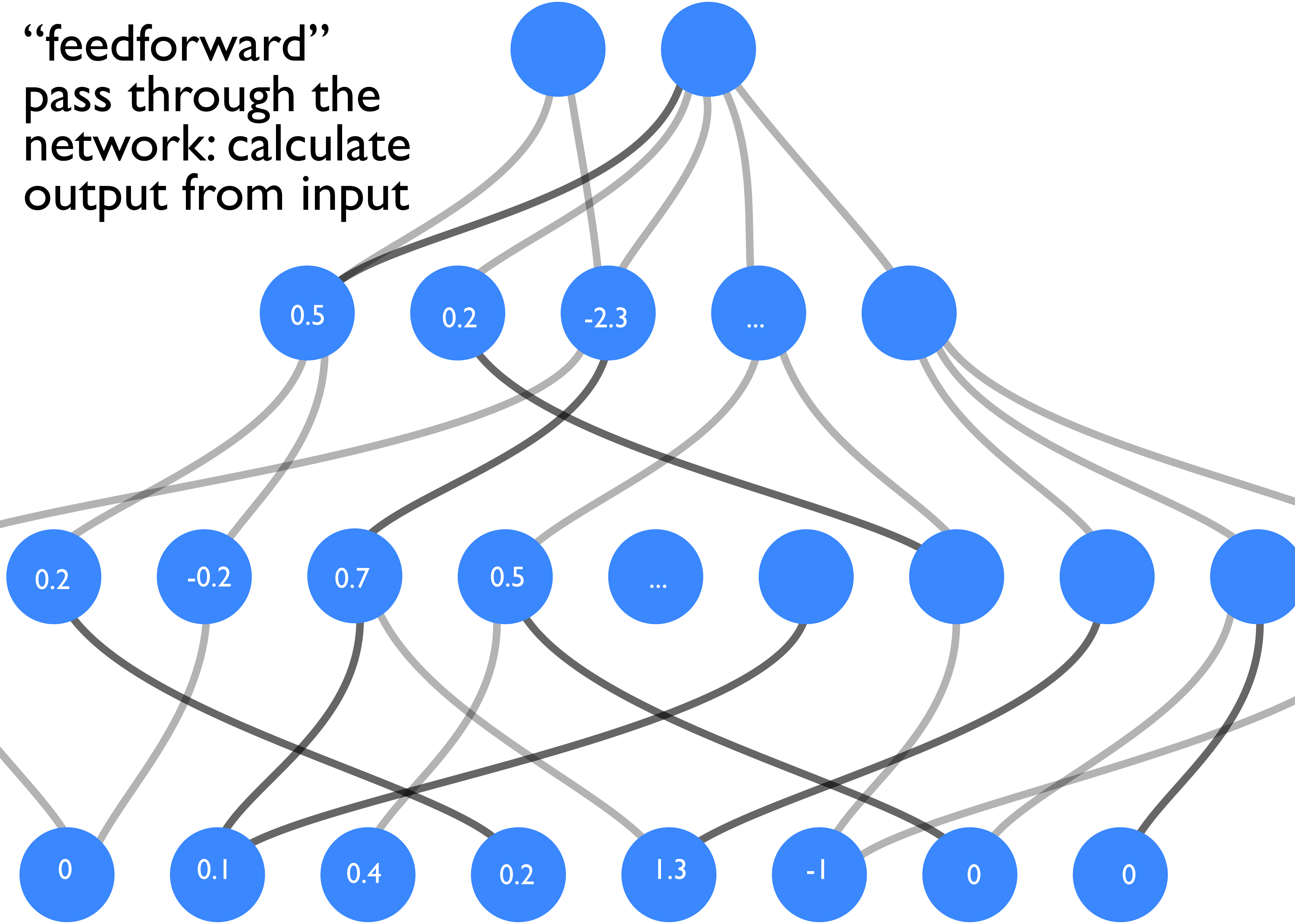
“feedforward” pass through the network: calculate output from input



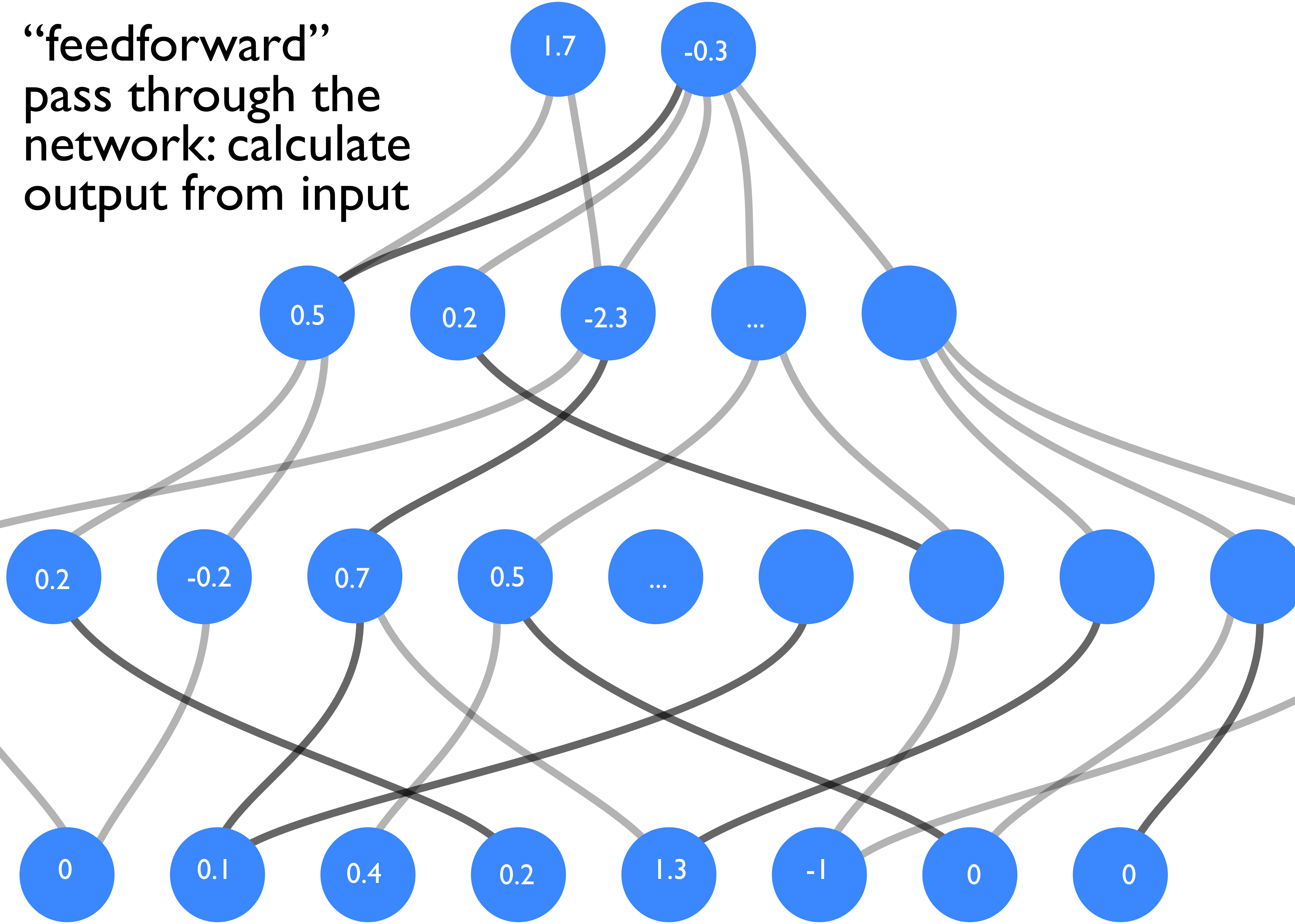
"feedforward" pass through the network: calculate output from input



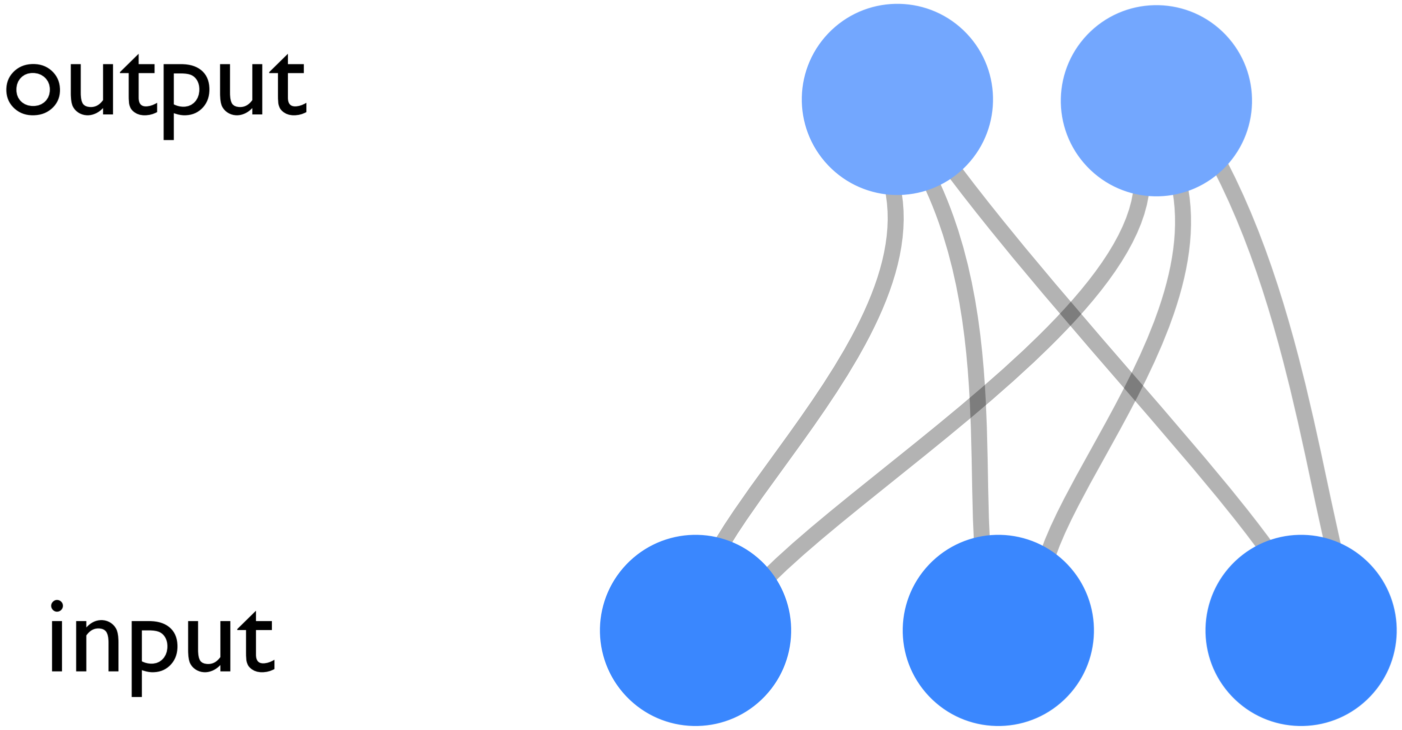
"feedforward" pass through the network: calculate output from input



“feedforward” pass through the network: calculate output from input



One layer



j=output neuron
k=input neuron

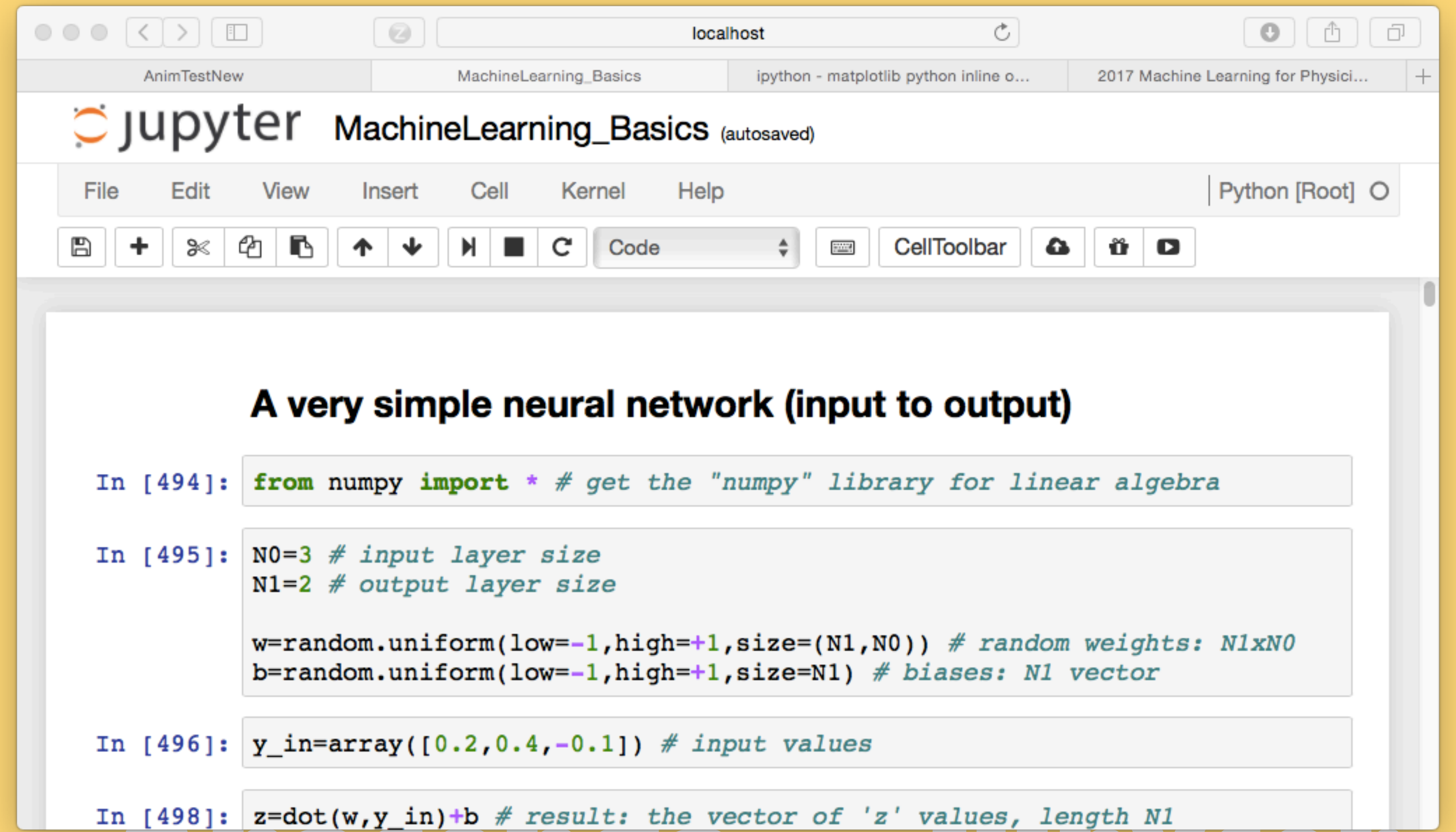
$$z_j = \sum_k w_{jk} y_k^{\text{in}} + b_j$$

in matrix/vector notation:

$$z = w y^{\text{in}} + b$$

elementwise nonlinear function:

$$y_j^{\text{out}} = f(z_j)$$



A very simple neural network (input to output)

```
In [494]: from numpy import * # get the "numpy" library for linear algebra
```

```
In [495]: N0=3 # input layer size
          N1=2 # output layer size

          w=random.uniform(low=-1,high=+1,size=(N1,N0)) # random weights: N1xN0
          b=random.uniform(low=-1,high=+1,size=N1) # biases: N1 vector
```

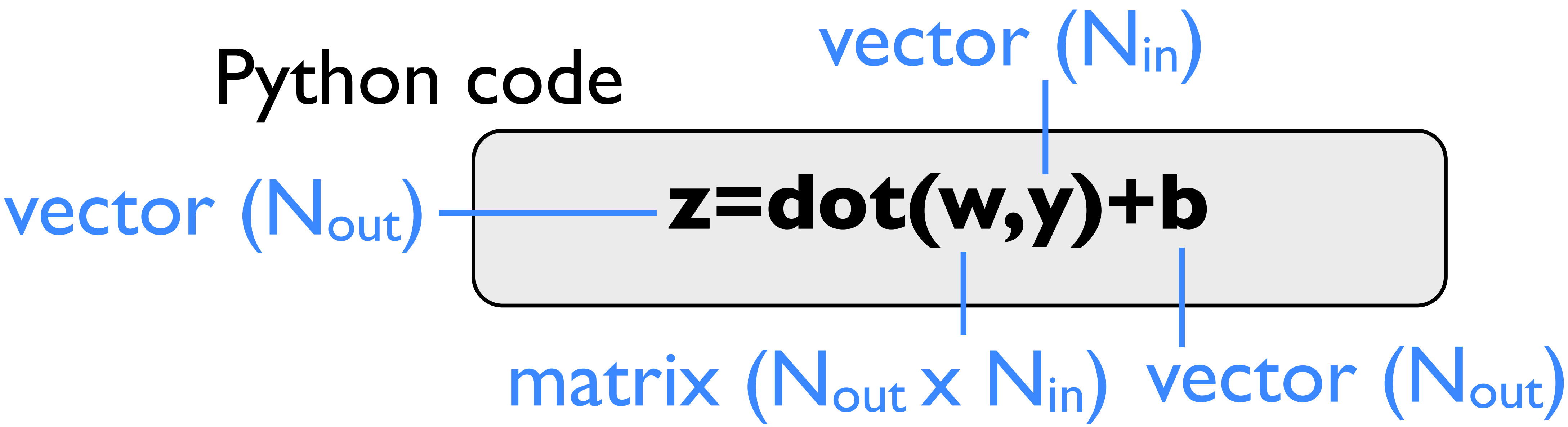
```
In [496]: y_in=array([0.2,0.4,-0.1]) # input values
```

```
In [498]: z=dot(w,y_in)+b # result: the vector of 'z' values, length N1
```

Anaconda Jupyter

Python

A few lines of “python” !



Python code

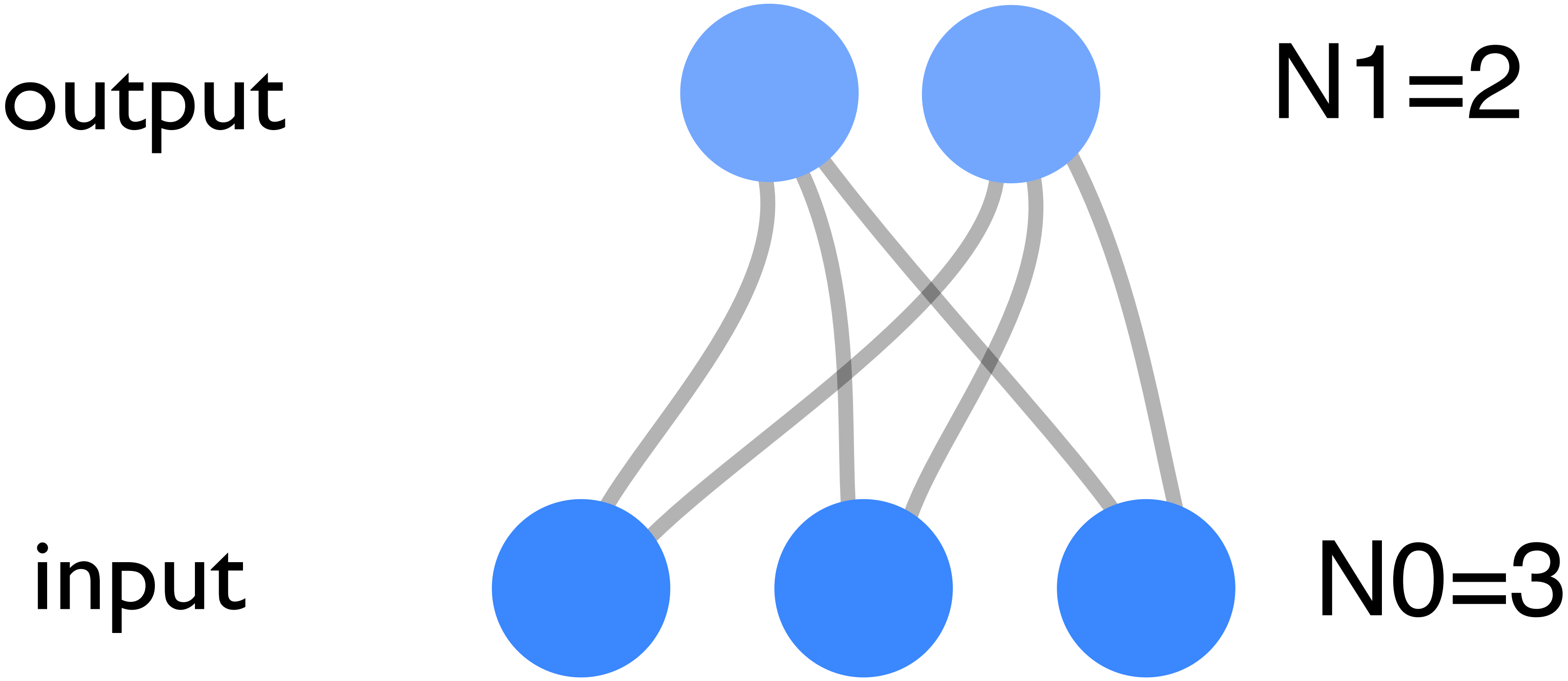
```
z=dot(w,y)+b
```

$$z_j = \sum_k w_{jk} y_k^{\mathrm{in}} + b_j$$

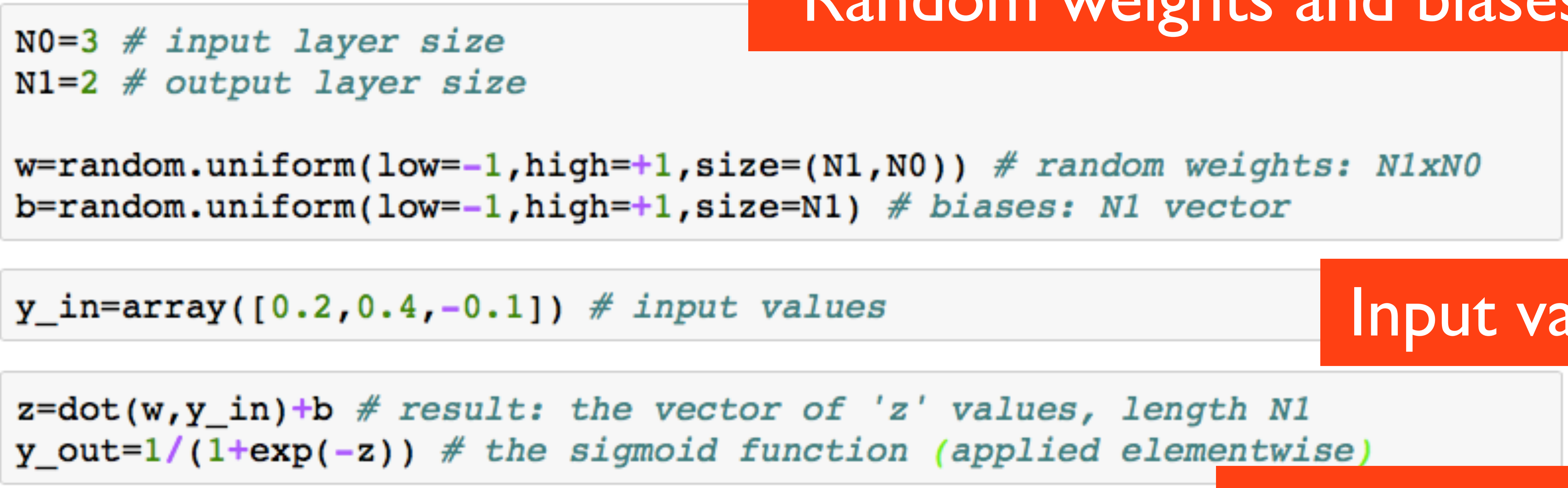
in matrix/vector notation:

$$z = w y^{\mathrm{in}} + b$$

A few lines of "python" !



Random weights and biases



```
N0=3 # input layer size
N1=2 # output layer size

w=random.uniform(low=-1,high=+1,size=(N1,N0)) # random weights: N1xN0
b=random.uniform(low=-1,high=+1,size=N1) # biases: N1 vector
```

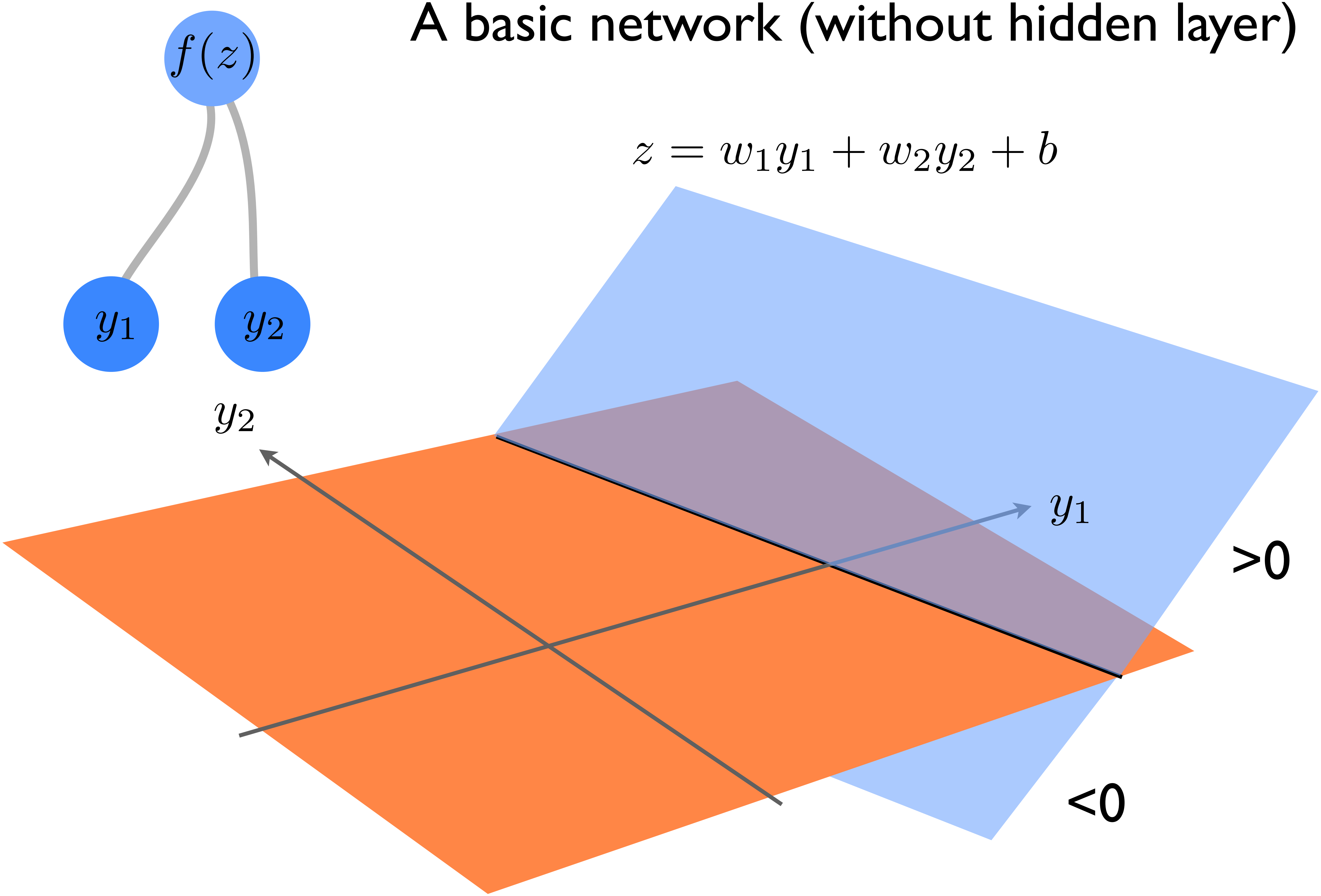
```
y_in=array([0.2,0.4,-0.1]) # input values
```

```
z=dot(w,y_in)+b # result: the vector of 'z' values, length N1
y_out=1/(1+exp(-z)) # the sigmoid function (applied elementwise)
```

Input values

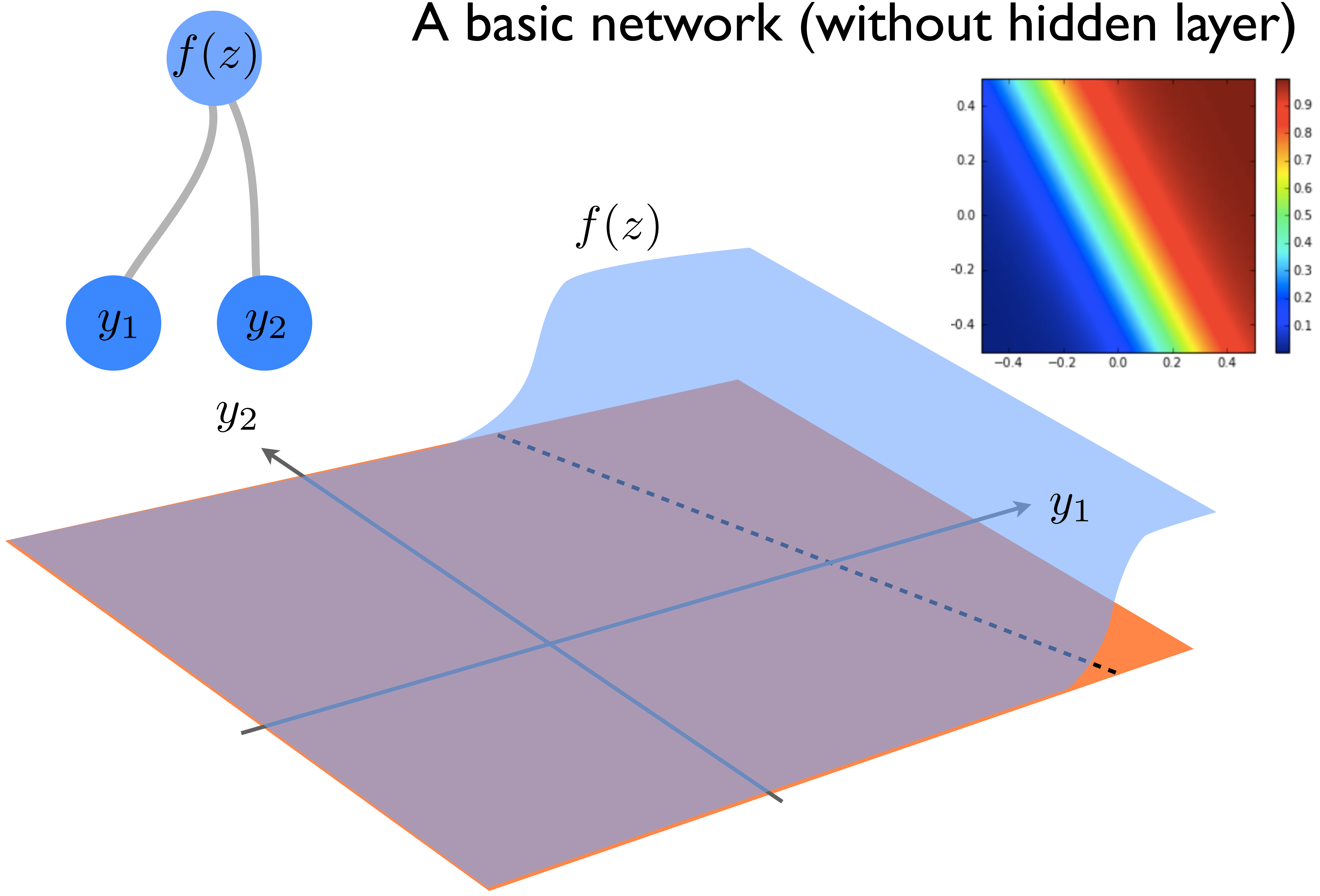
Apply network!

A basic network (without hidden layer)

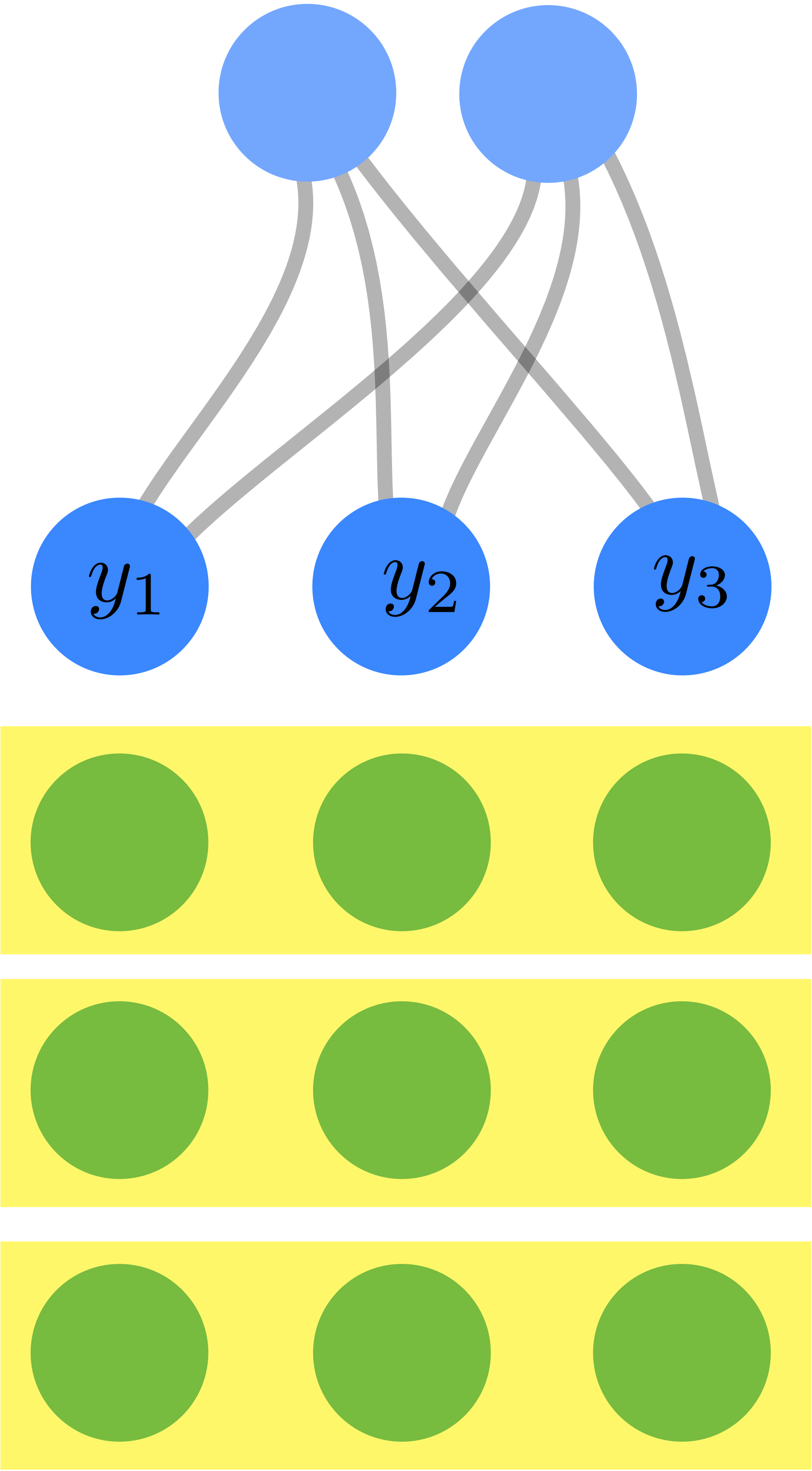


$$z = w_1 y_1 + w_2 y_2 + b$$

A basic network (without hidden layer)



Processing batches: Many samples in parallel



sample 1

sample 2

sample 3

...

Avoid loops! (slow)

Processing batches: Many samples in parallel

one sample:

vector (N_{in}) **y**

many samples:

matrix ($N_{samples}$ x N_{in}) **y**

Apply matrix/vector operations to operate on all samples simultaneously! Avoid loops! (slow)

vector (N_2)

Note: Python interprets **M=A+b**

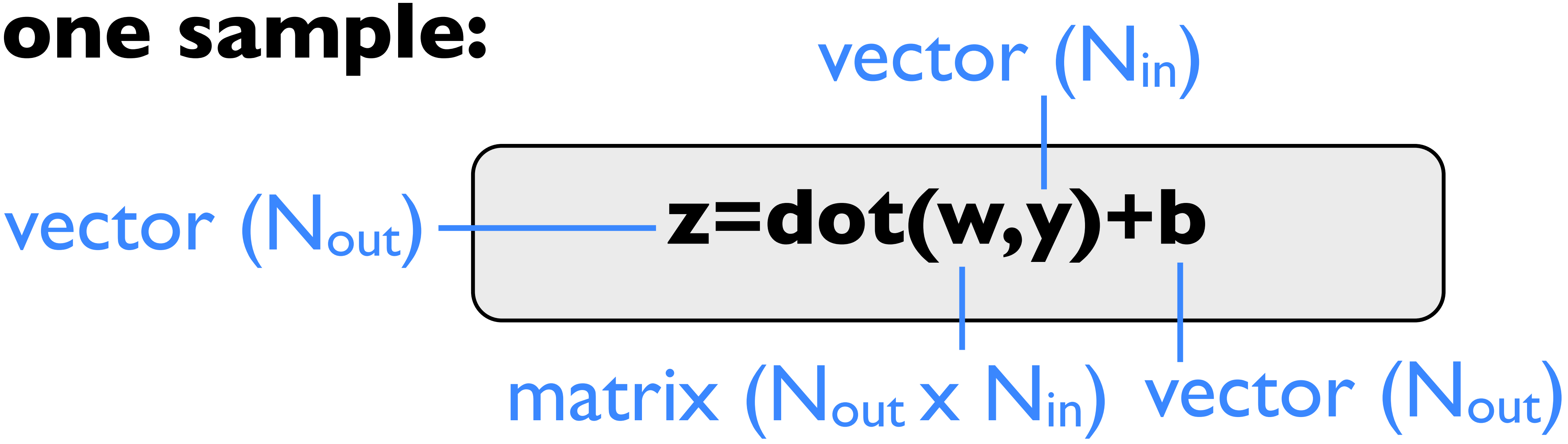
matrix (N_1 x N_2)

as: $$M_{ij} = A_{ij} + b_j$$

First index of **b** is 'expanded' to size indicated by **A**

Processing batches: Many samples in parallel

one sample:



vector (N_{in})

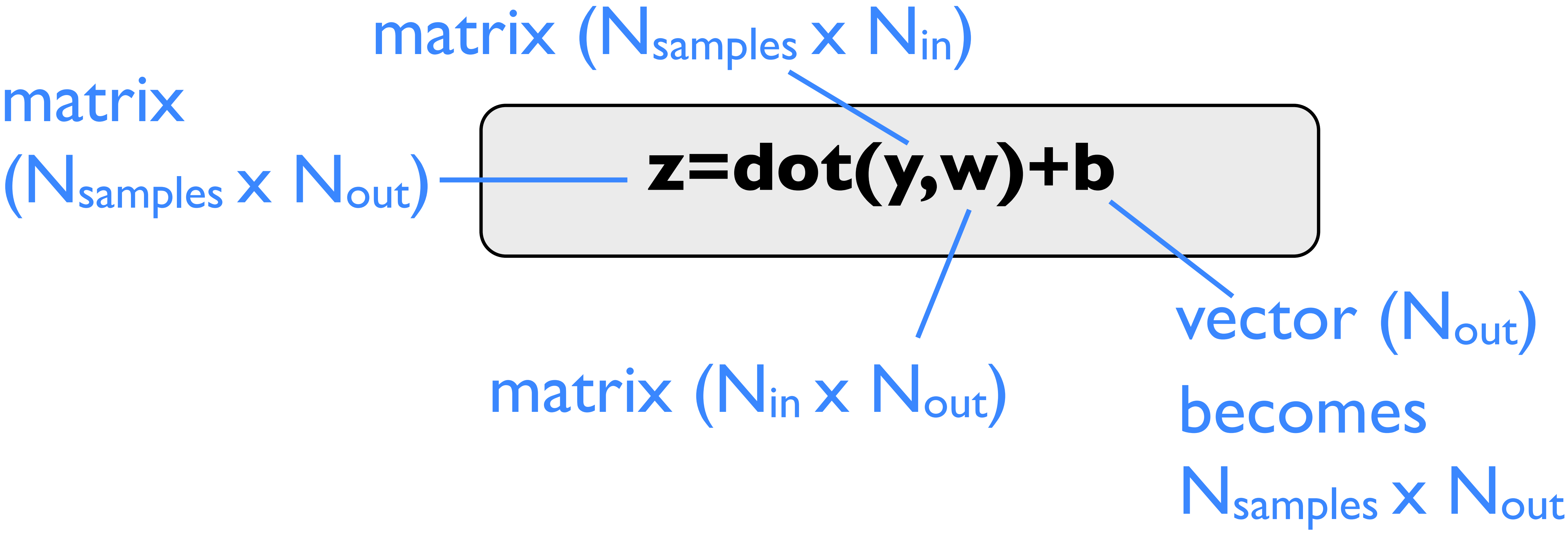
vector (N_{out})

z=dot(w,y)+b

matrix (N_{out} x N_{in})

vector (N_{out})

many samples:



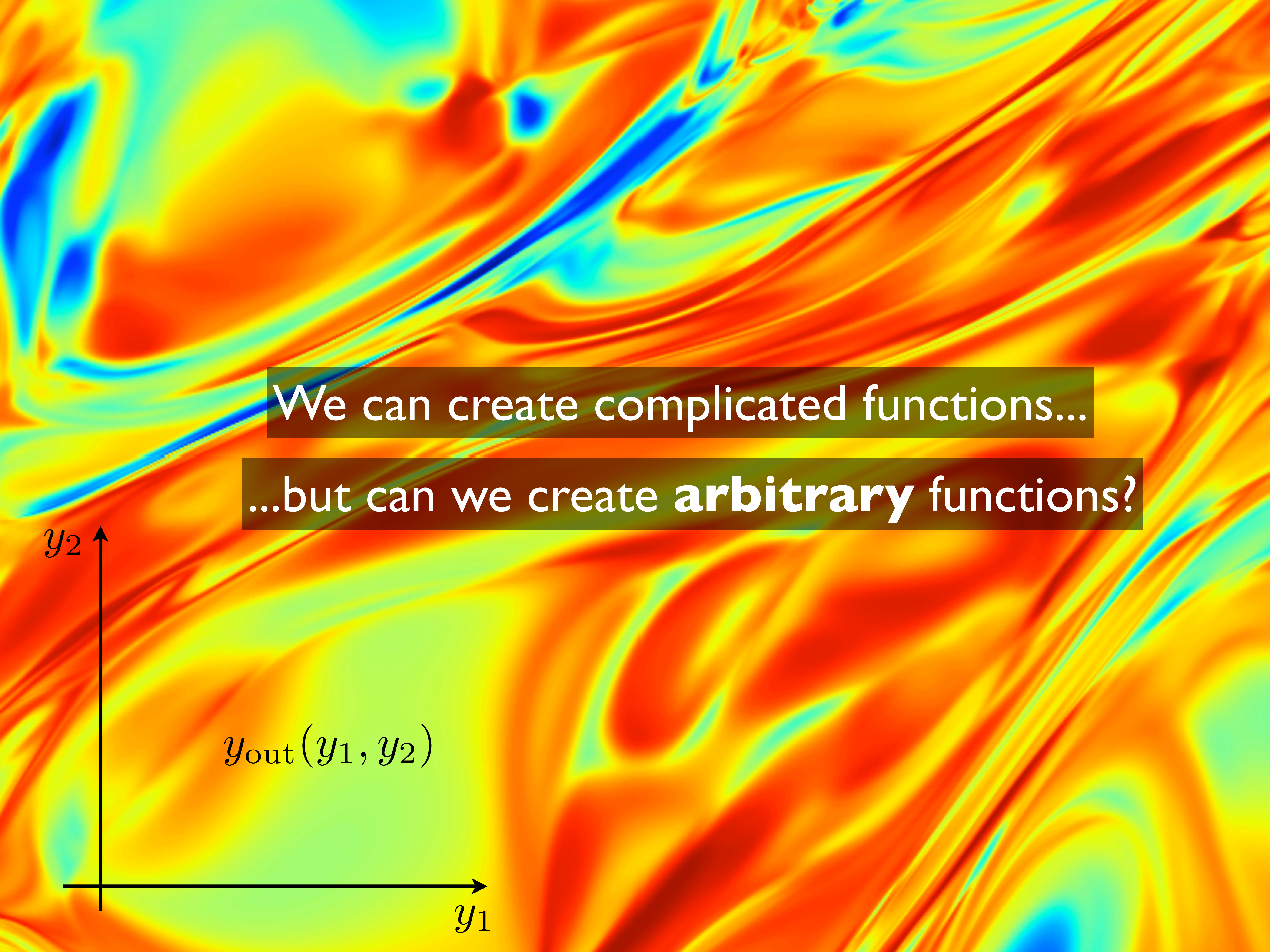
matrix ($N_{samples}$ x N_{in})

matrix ($N_{samples}$ x N_{out})

z=dot(y,w)+b

matrix (N_{in} x N_{out})

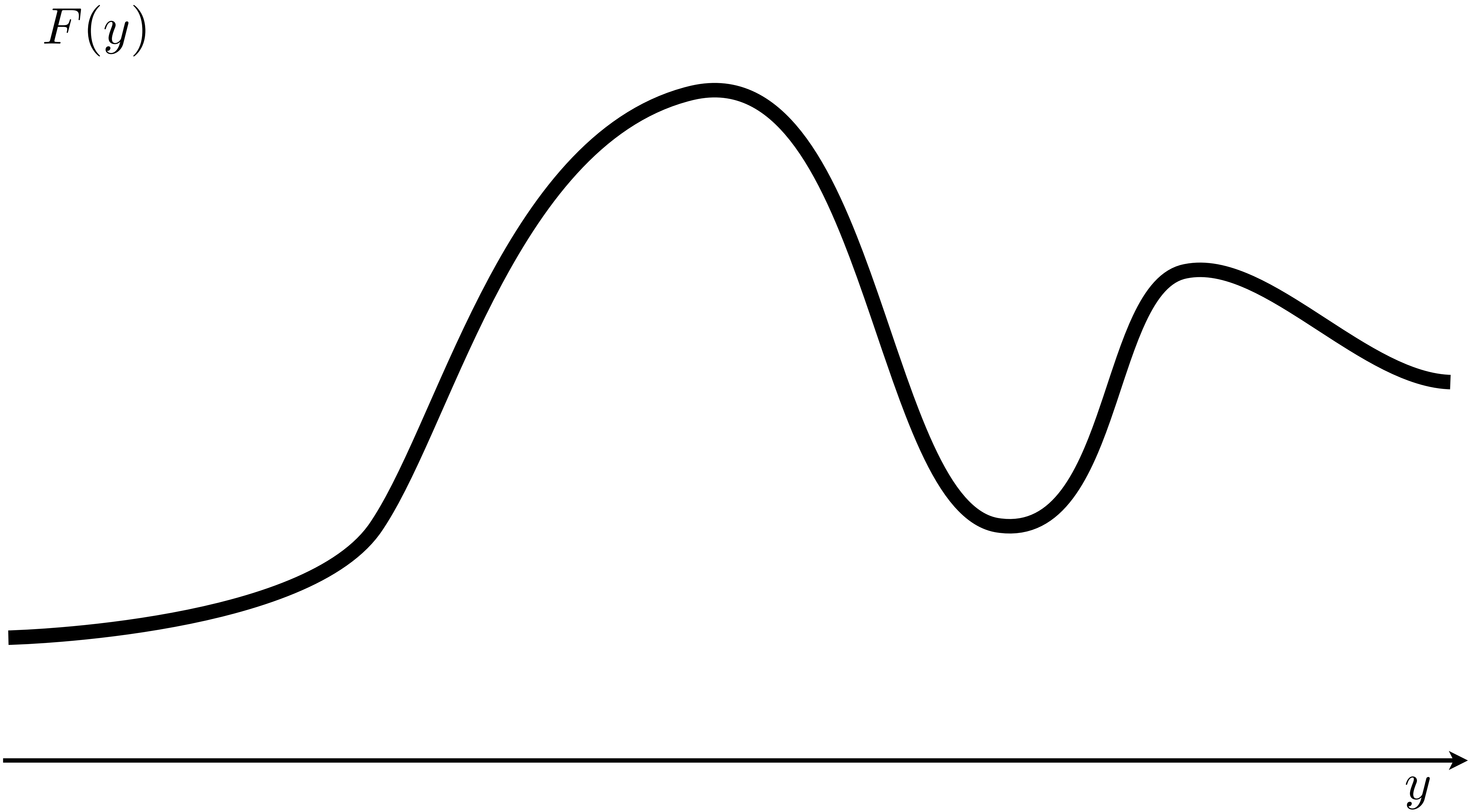
vector (N_{out}) becomes $N_{samples}$ x N_{out}



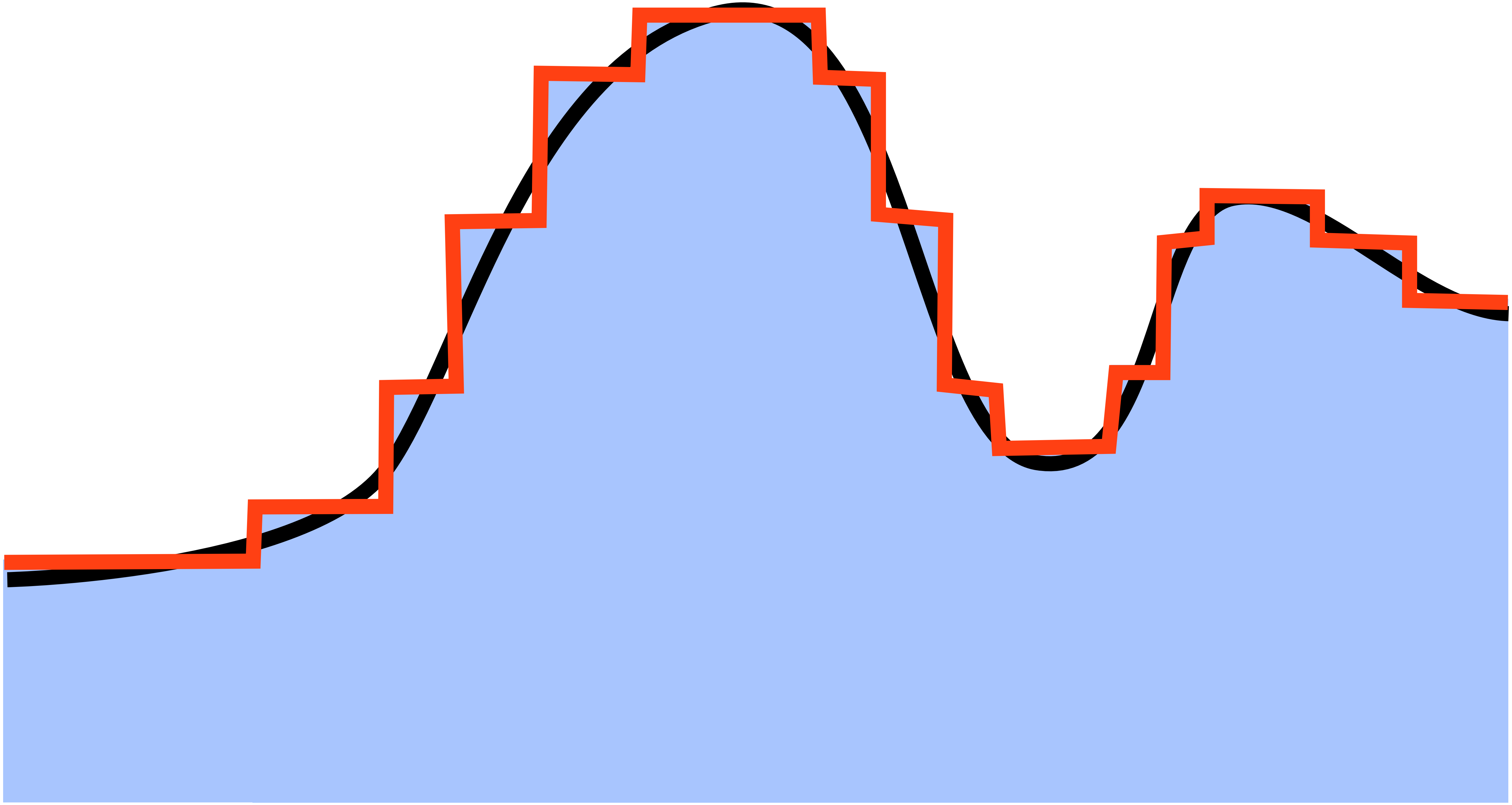
We can create complicated functions...

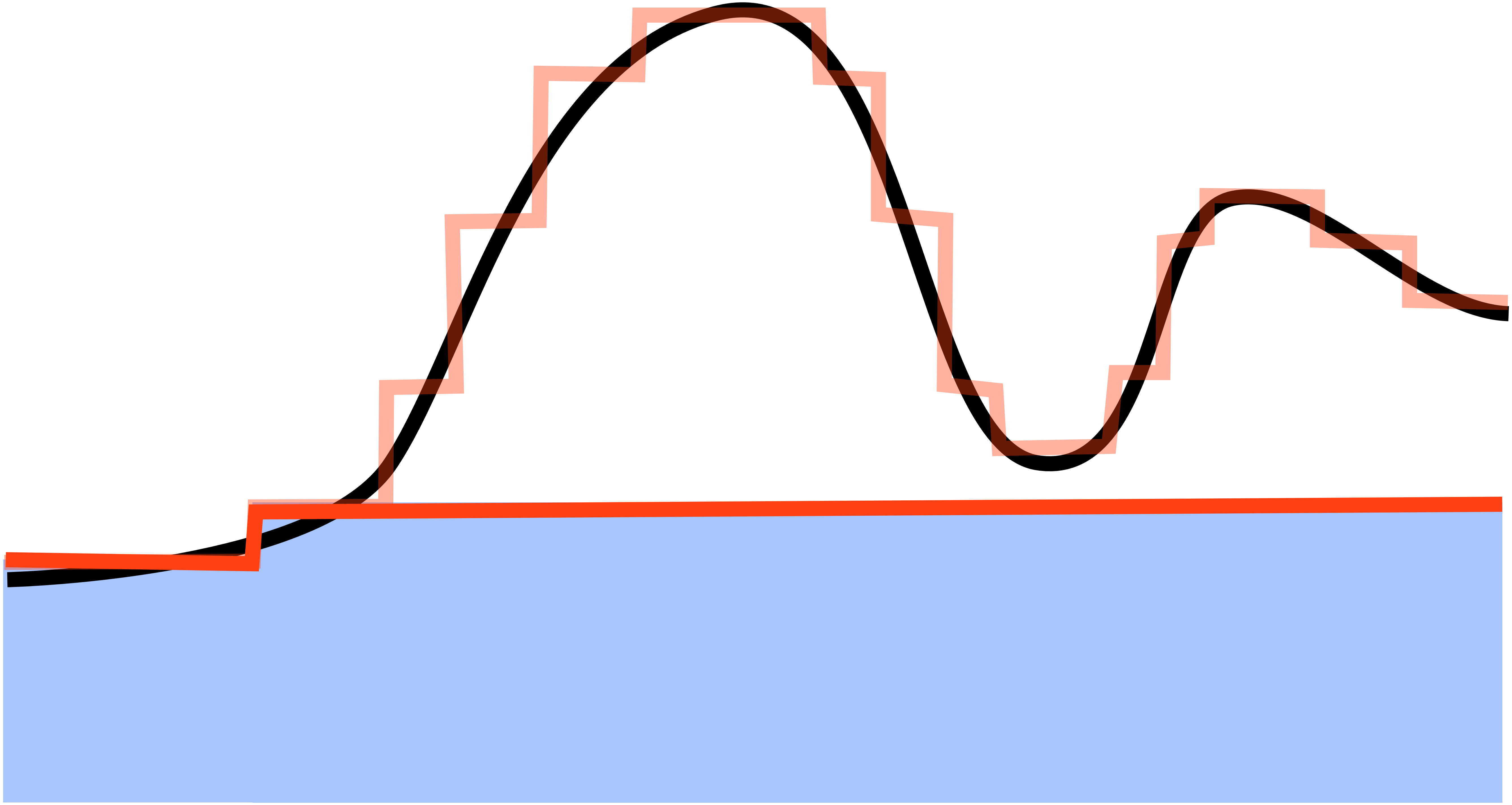
...but can we create **arbitrary** functions?

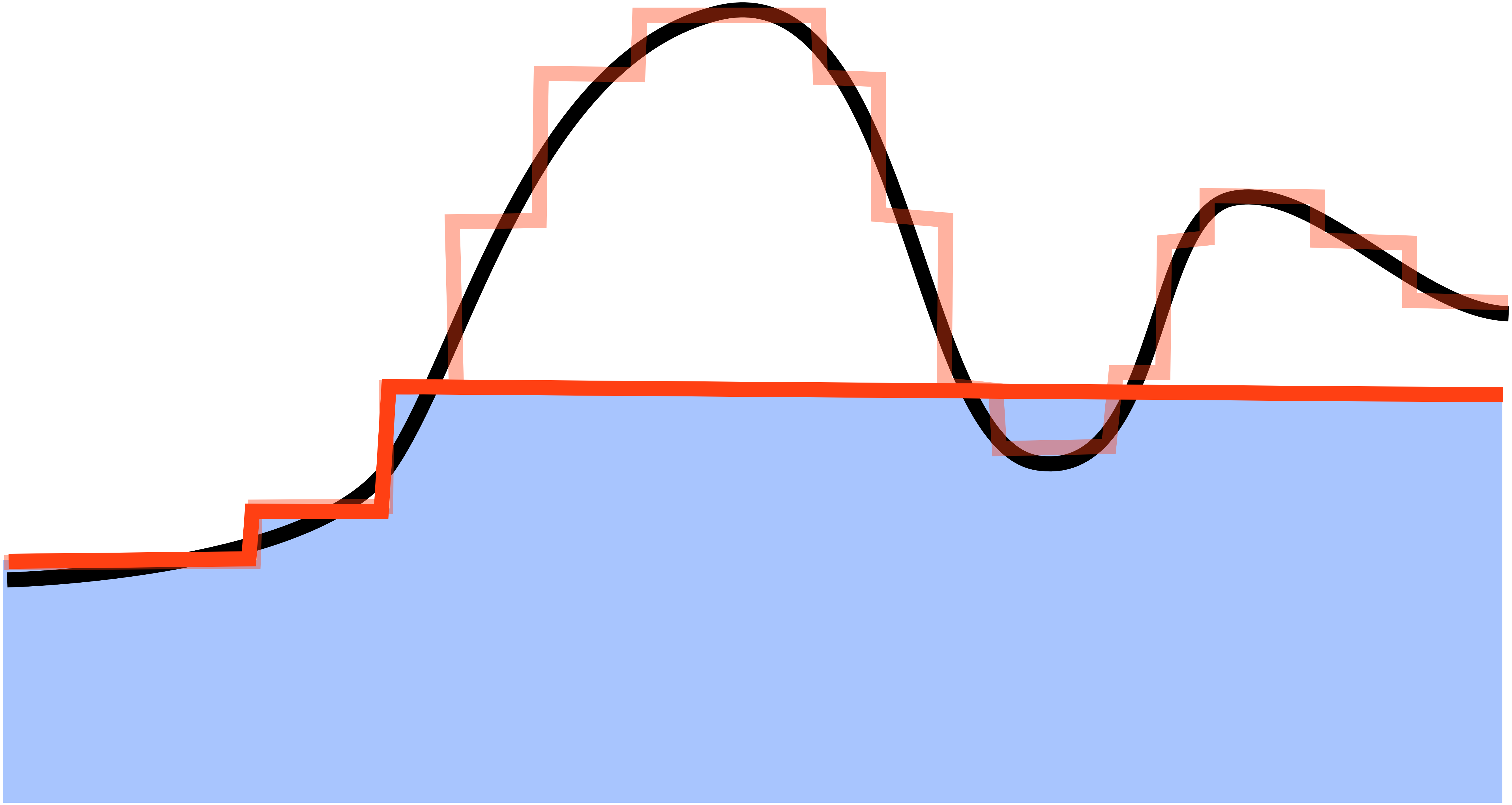
Approximating an arbitrary nonlinear function



Approximating an arbitrary nonlinear function

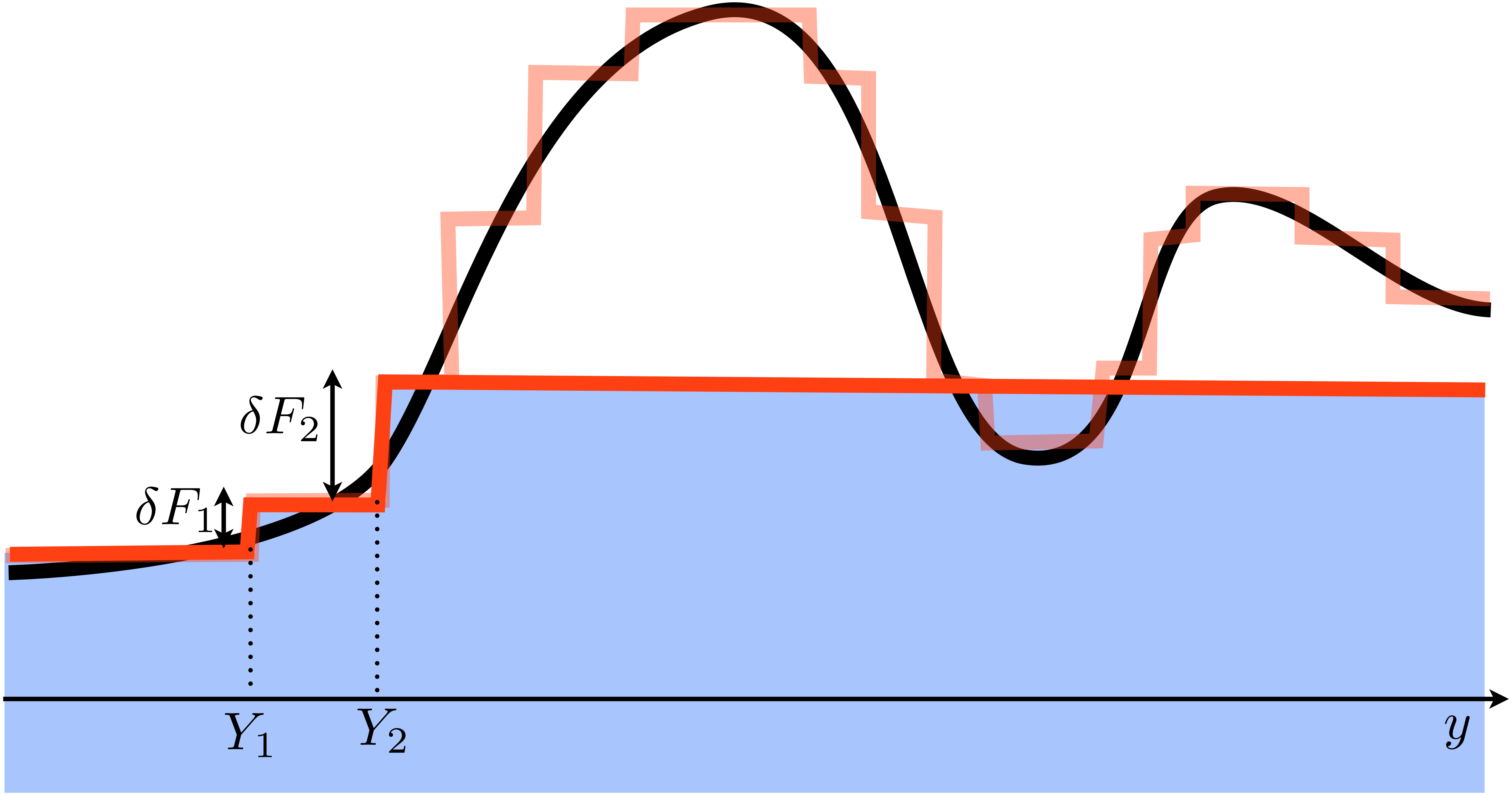


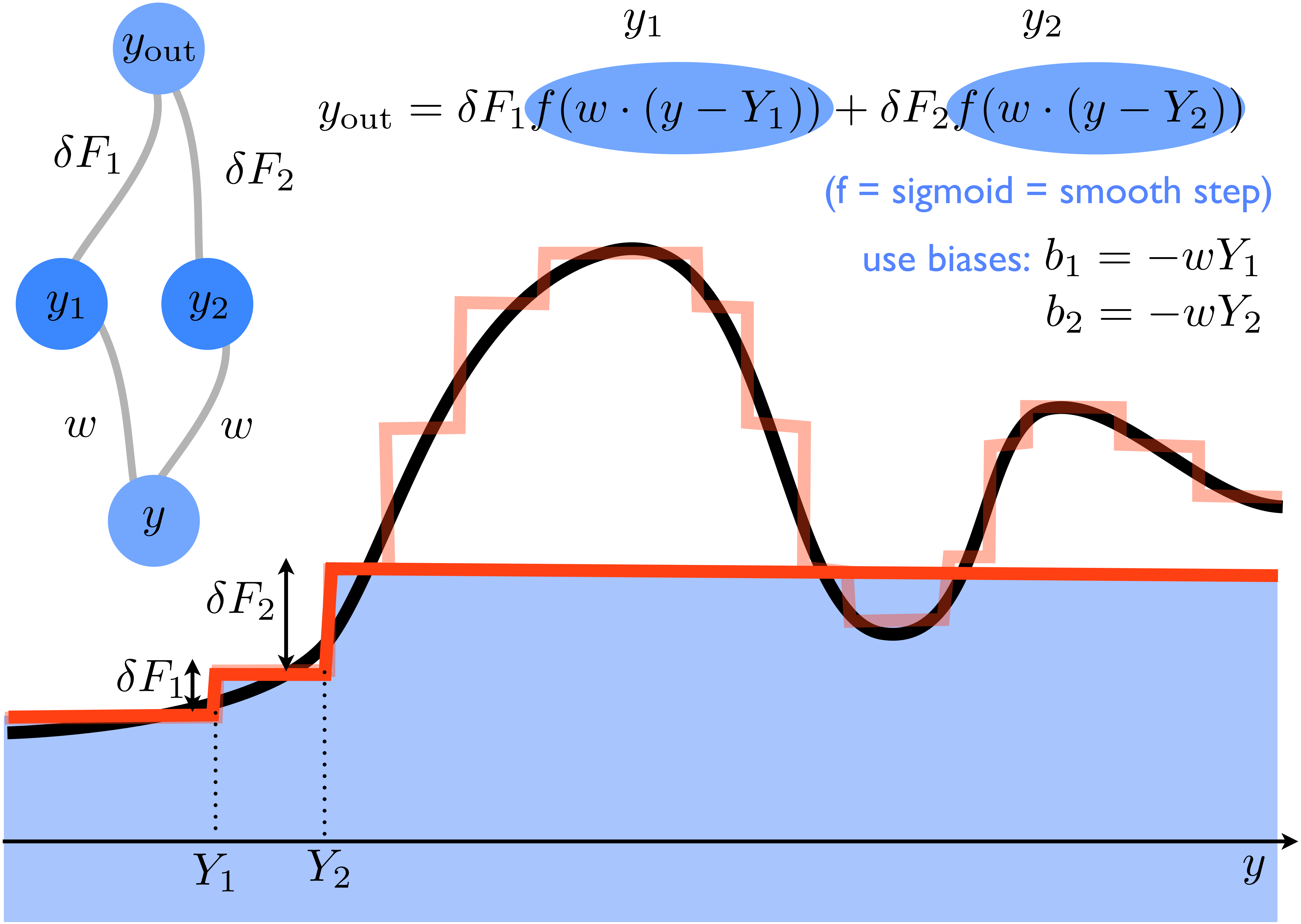




$$y_{\mathrm{out}} = \delta F_1 f(w \cdot (y - Y_1)) + \delta F_2 f(w \cdot (y - Y_2))$$

(f = sigmoid = smooth step)



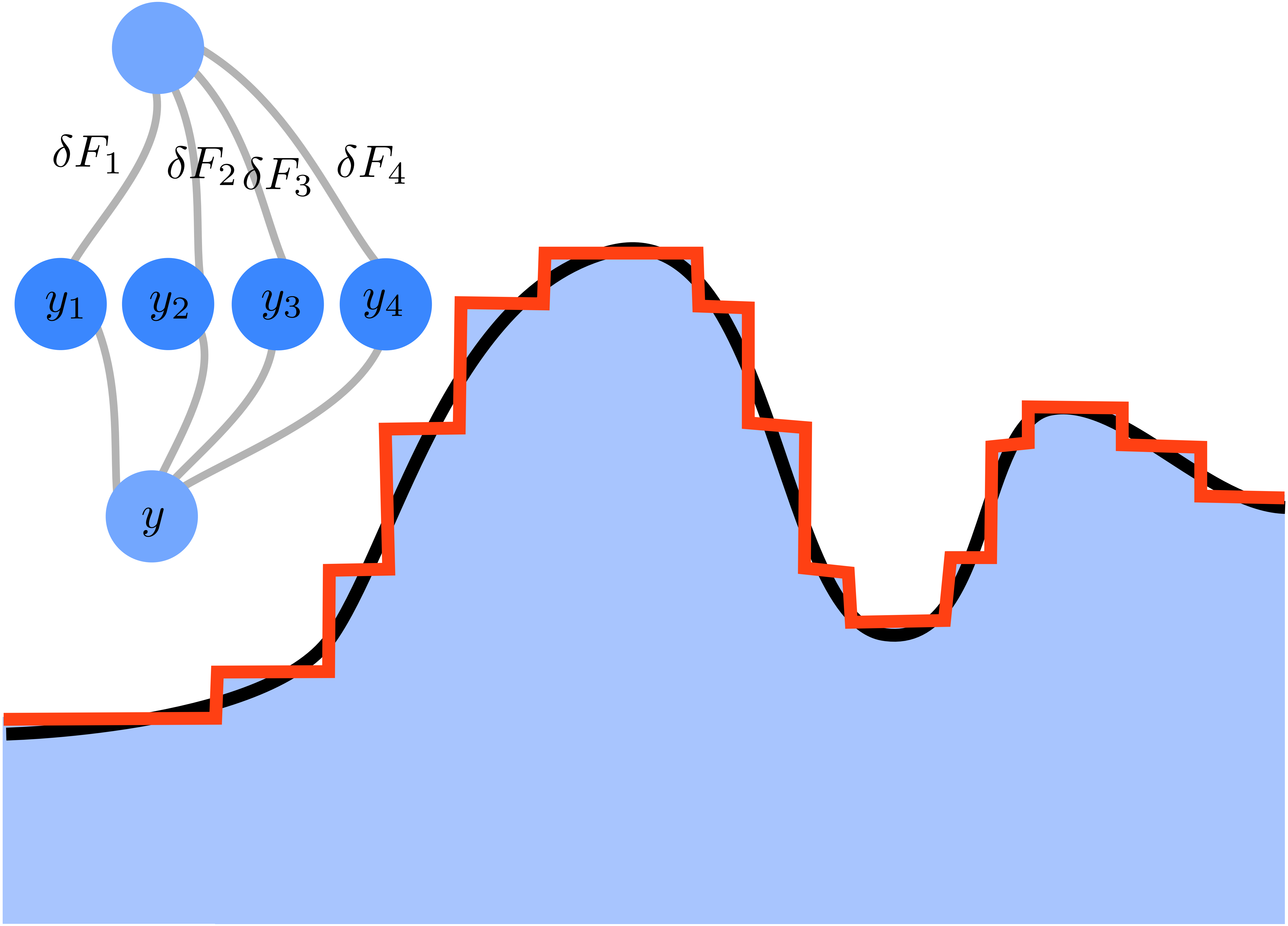


$$y_{\text{out}} = \delta F_1 \overset{y_1}{f(w\cdot(y-Y_1))} + \delta F_2 \overset{y_2}{f(w\cdot(y-Y_2))}$$

(f = sigmoid = smooth step)

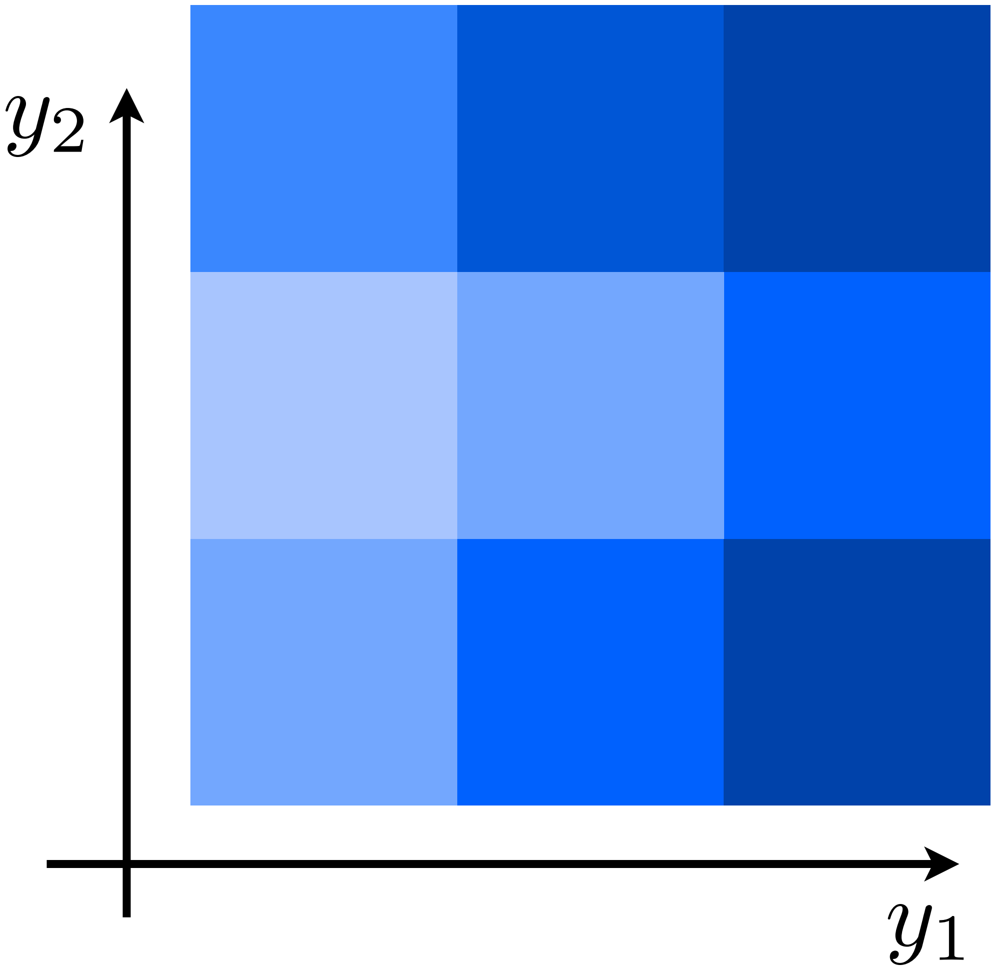
use biases: $b_1 = -wY_1$

$b_2 = -wY_2$



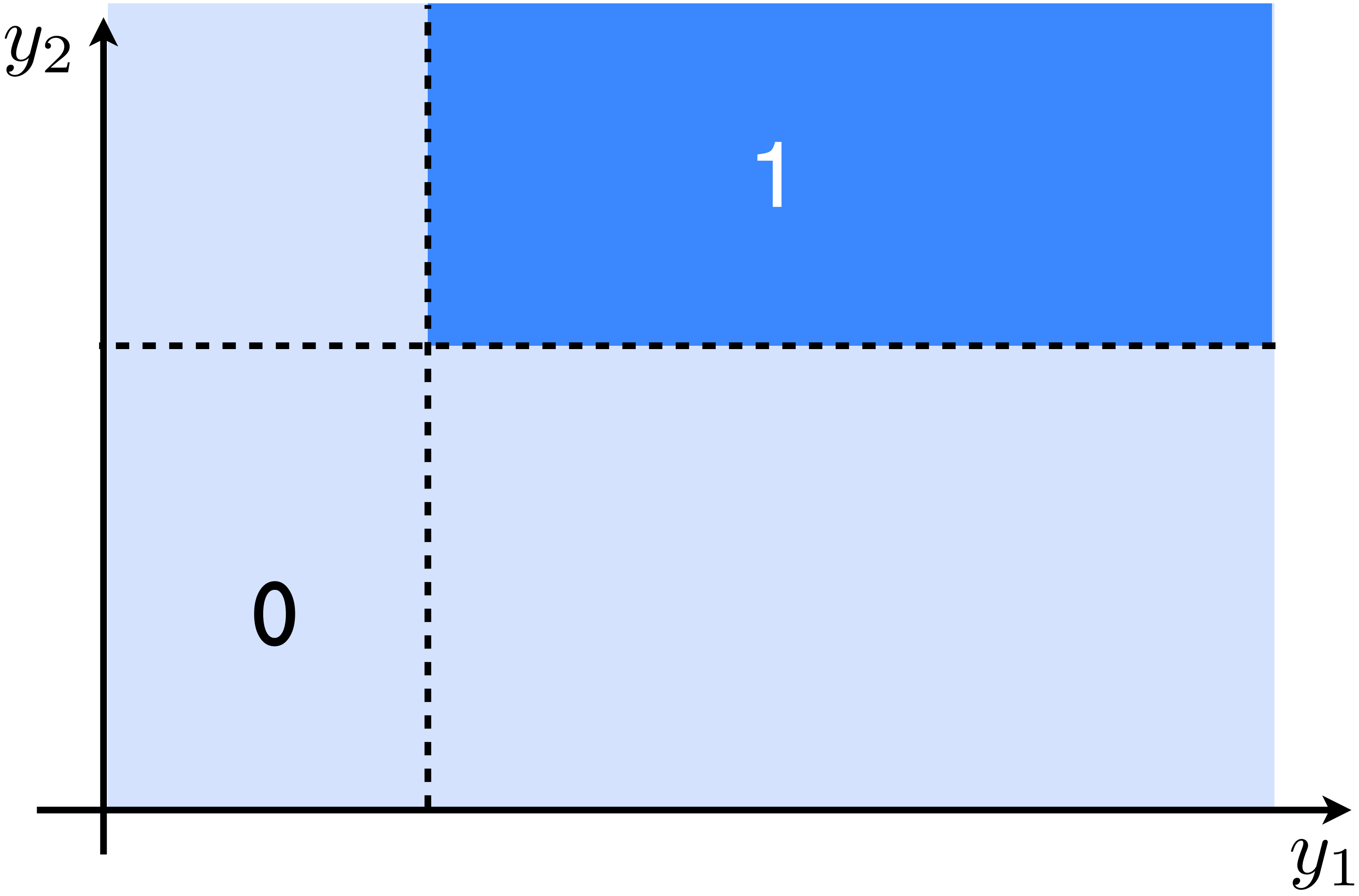
δF_1
δF_2
δF_3
δF_4
y_1
y_2
y_3
y_4
y

Approximating an arbitrary 2D nonlin. function

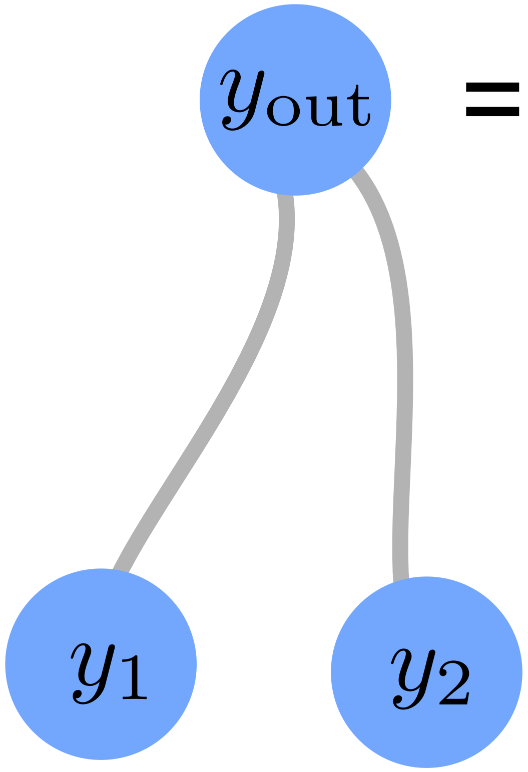


Approximating an arbitrary 2D nonlin. function

First step: create quarter-space "step function"



Trick: "AND" operation in a neural network



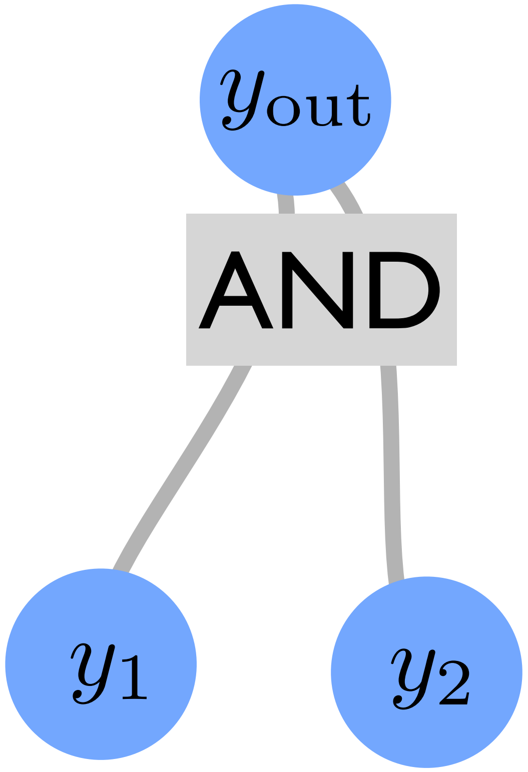
= $f(w \cdot (y_1 + y_2 - 1.5))$

$0 \leq y_j \leq 1$

for large w :

y_1 \ y_2	0	1
0	0	0
1	0	1

Trick: "AND" operation in a neural network



$0 \leq y_j \leq 1$

for large w :

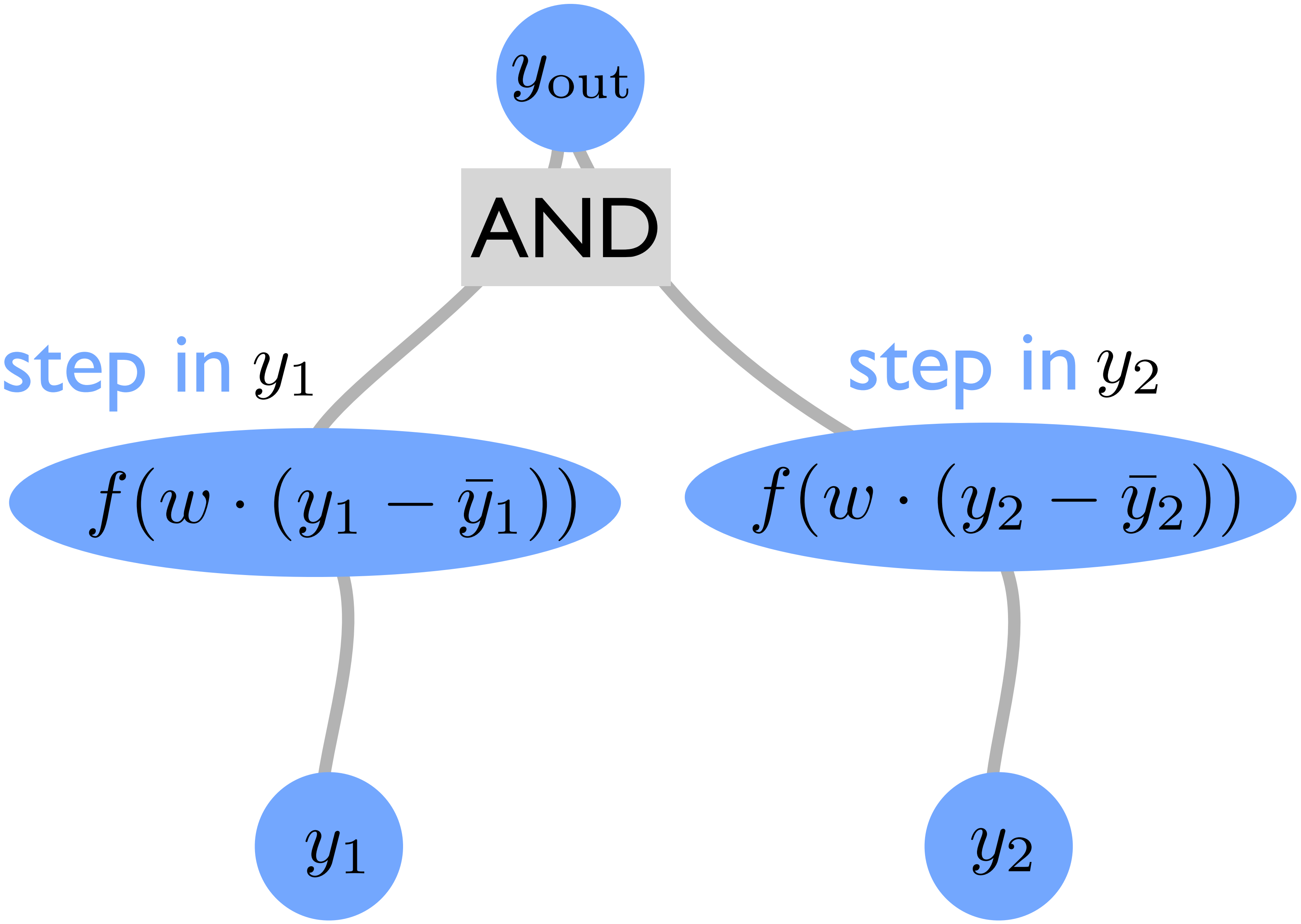
y_2 y_1	0	1
0	0	0
1	0	1

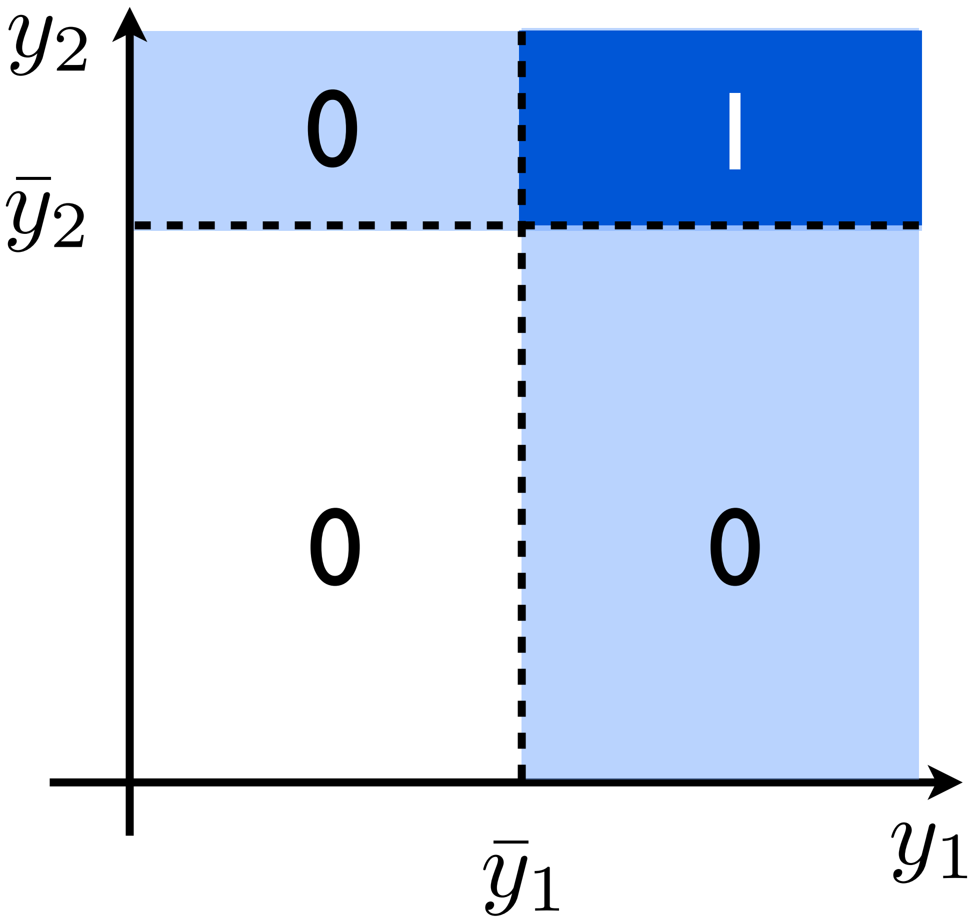
Homework

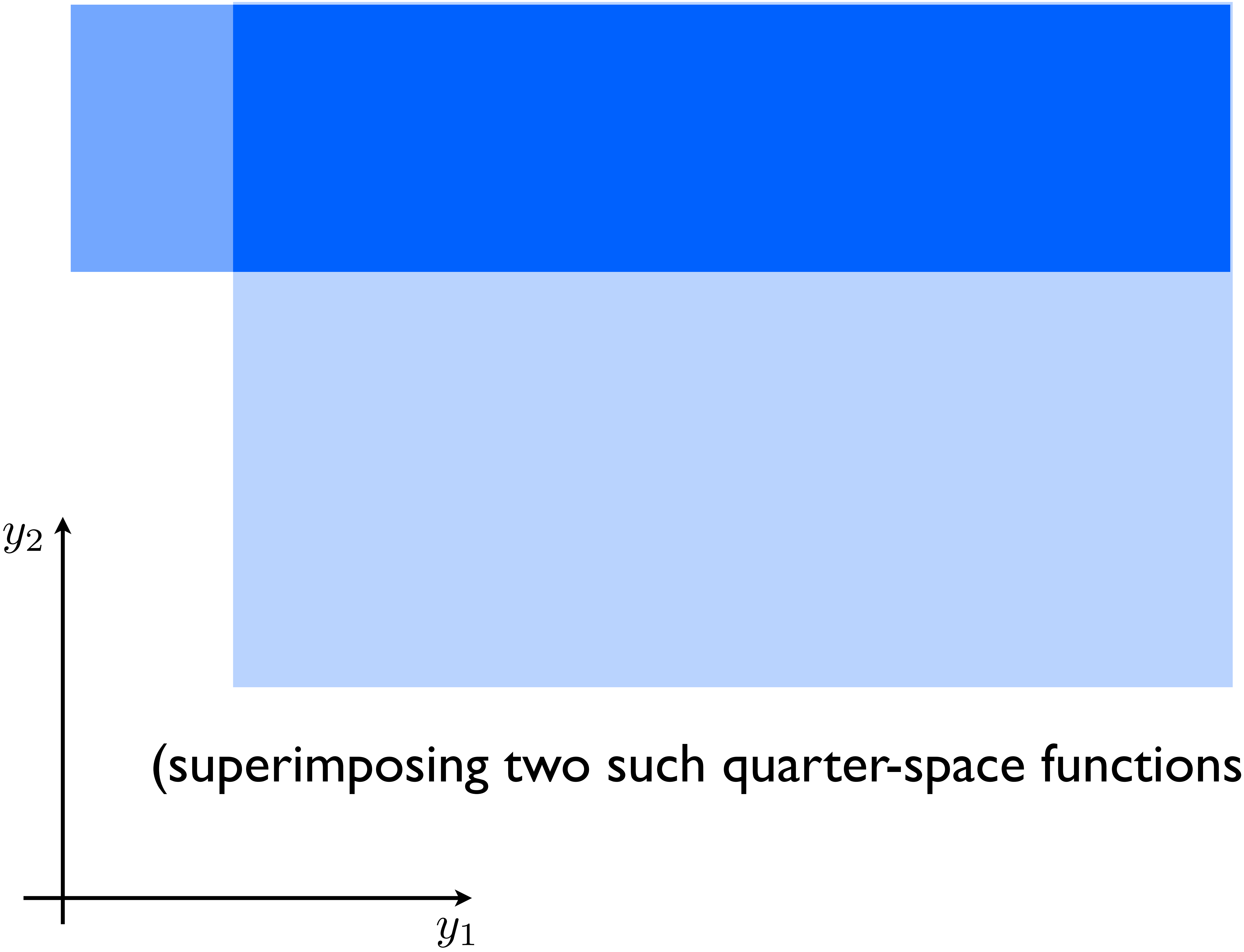
Figure out how to implement the following operations using a neural network:

OR
XOR (gives 1 only if inputs are different, i.e. for 10 and 01)

Approximating an arbitrary 2D nonlin. function

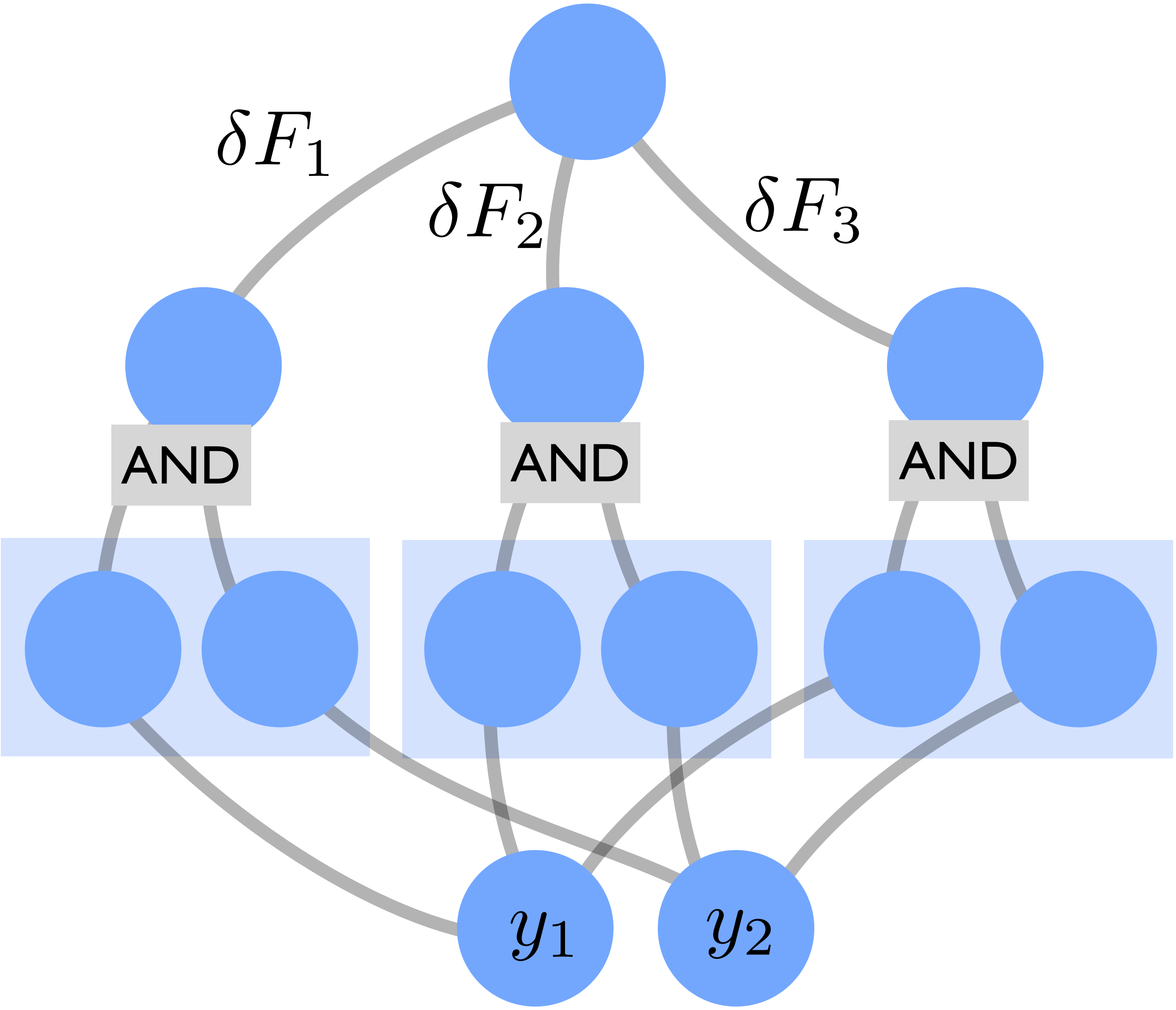


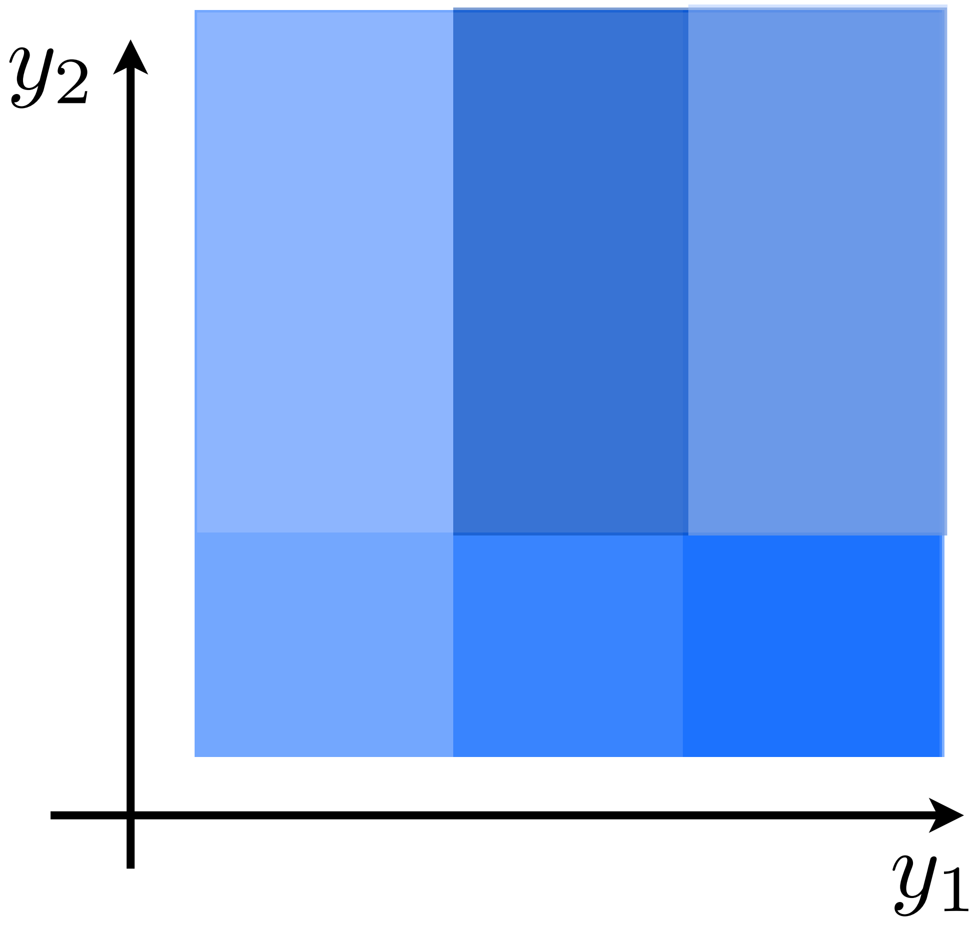




(superimposing two such quarter-space functions)

Approximating an arbitrary 2D nonlin. function

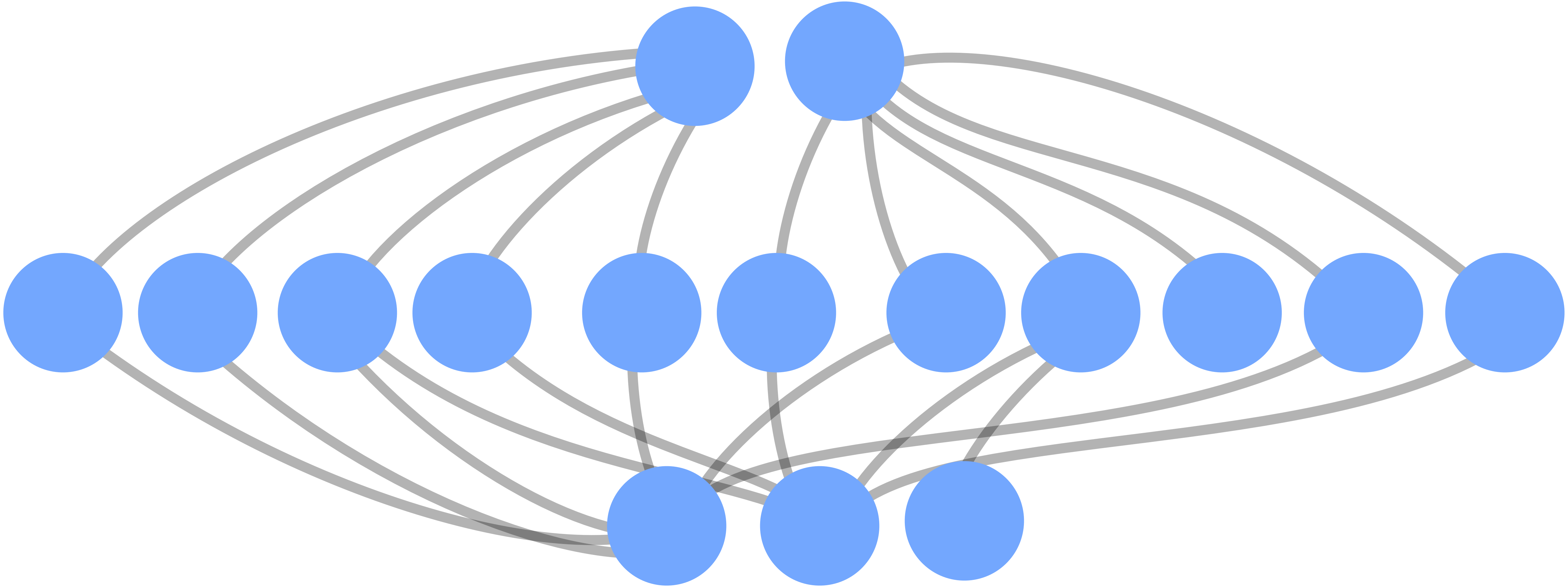




Universality of neural networks

Any arbitrary (smooth) function (with vector input and vector output) can be approximated as well as desired by a neural network with a single (!) hidden layer.

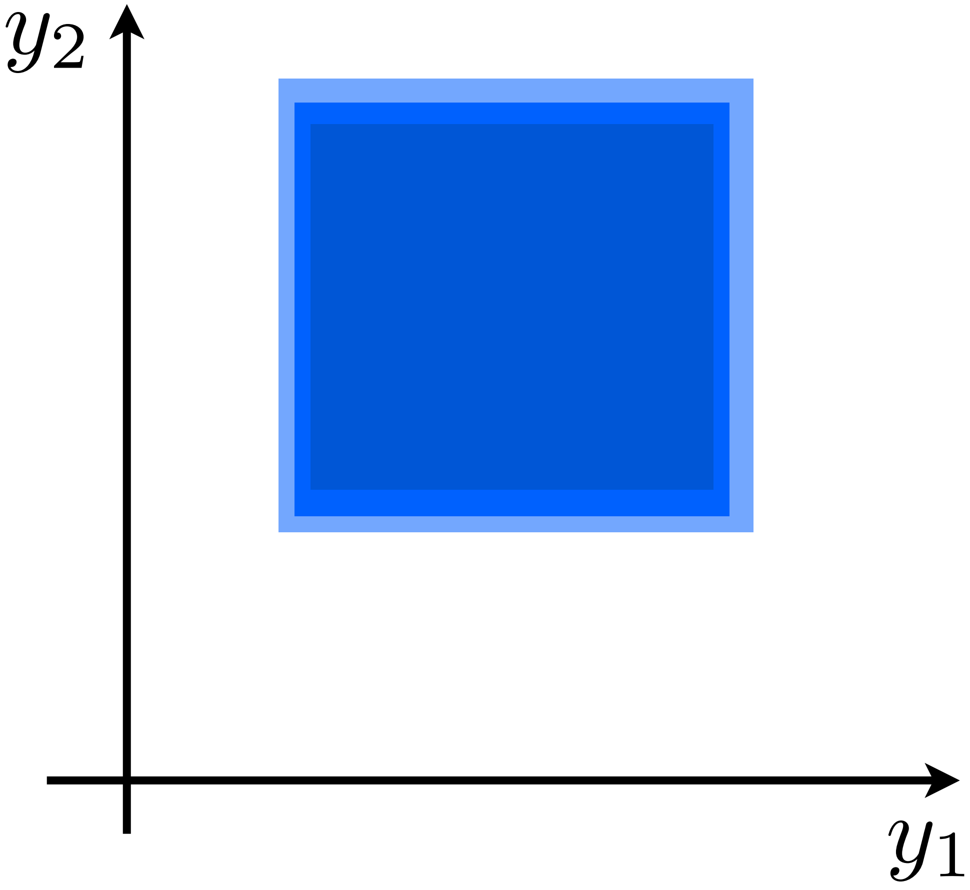
(as long as we allow for sufficiently many neurons)



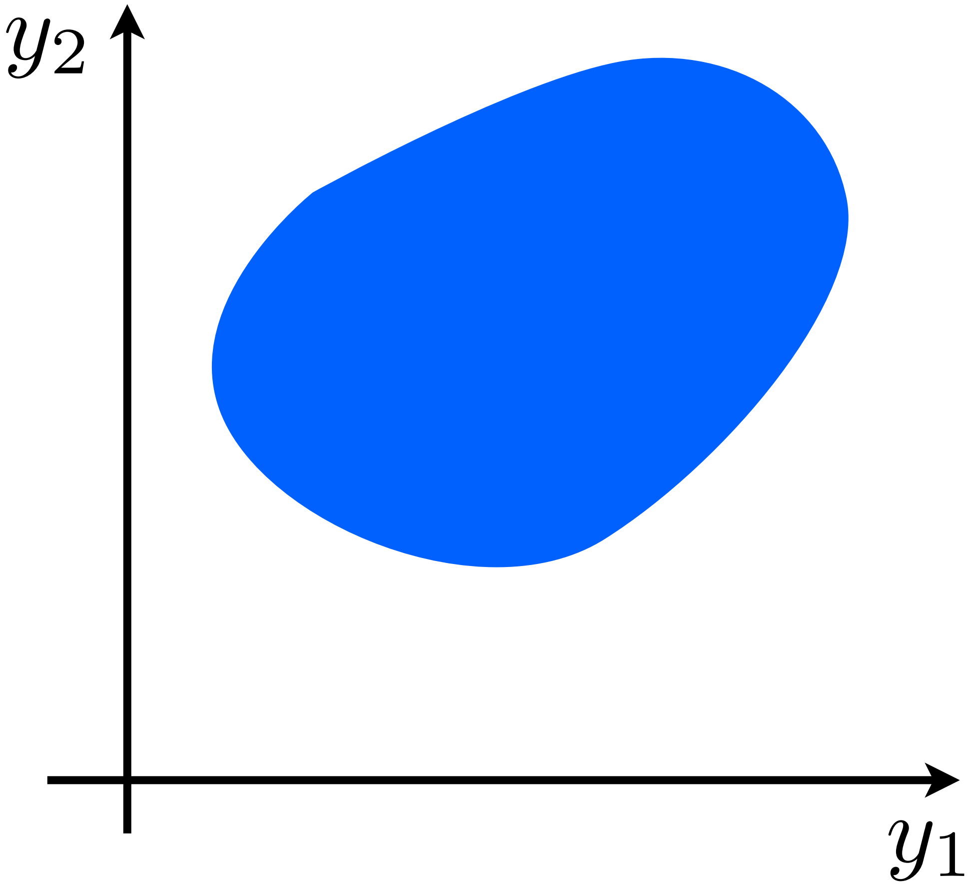
"Approximation by superpositions of a sigmoidal function", by George Cybenko (1989)

Homework

Figure out how to implement a 2D function that produces a (smoothened) square



Bonus version: how to get an arbitrary convex shape (approximately)?



Implement them on the computer and play around...

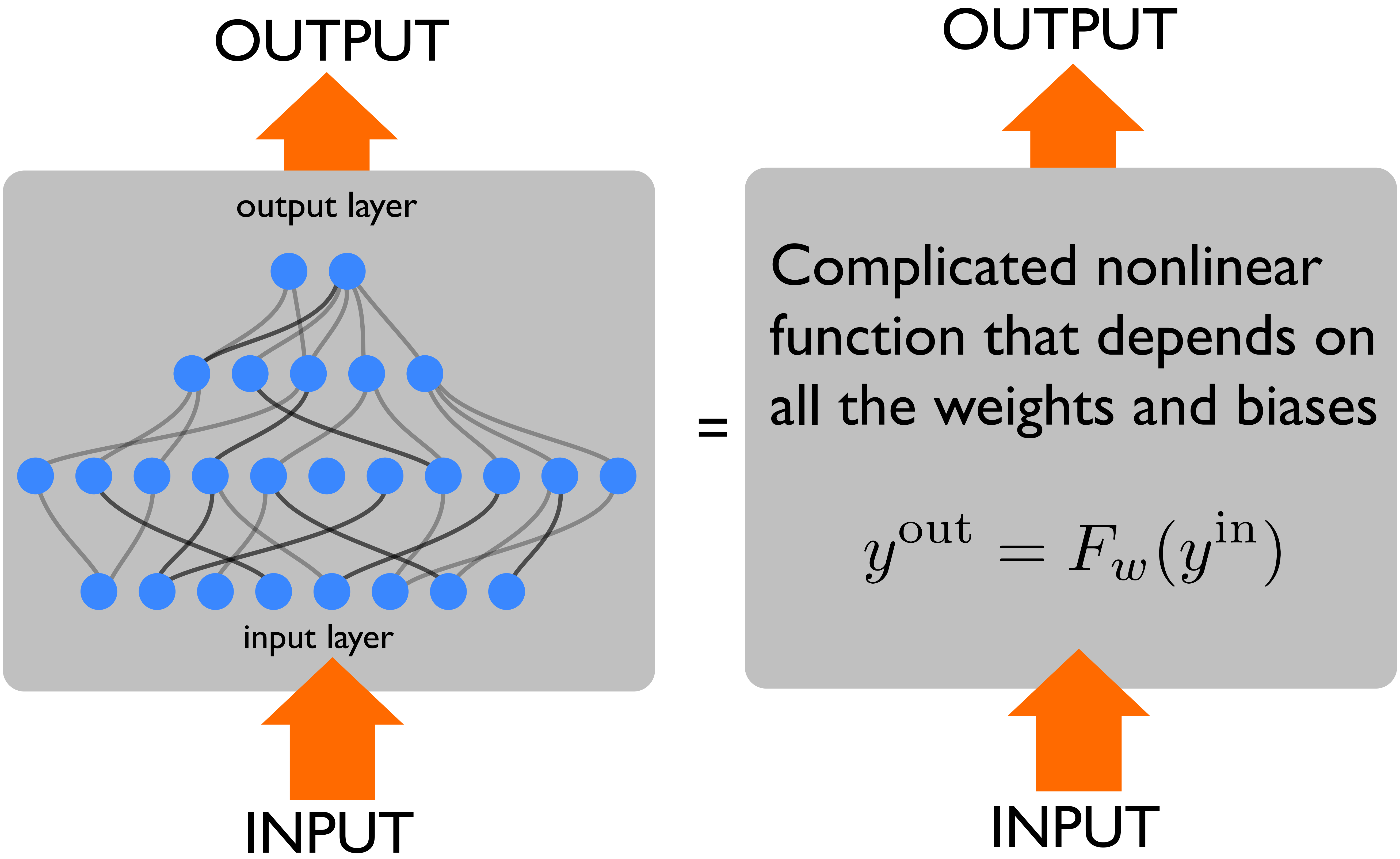
Homework

Extra * bonus version:

We have indicated how to approximate arbitrary functions in 2D using 2 hidden layers (with our AND construction, and summing up in the end)

Can you do it with a **single** hidden layer?

A neural network



=

OUTPUT

Complicated nonlinear function that depends on all the weights and biases

$$y^{\mathrm{out}} = F_w(y^{\mathrm{in}})$$

INPUT

Note: When we write "w" as subscript of F, we mean all the weights and also biases

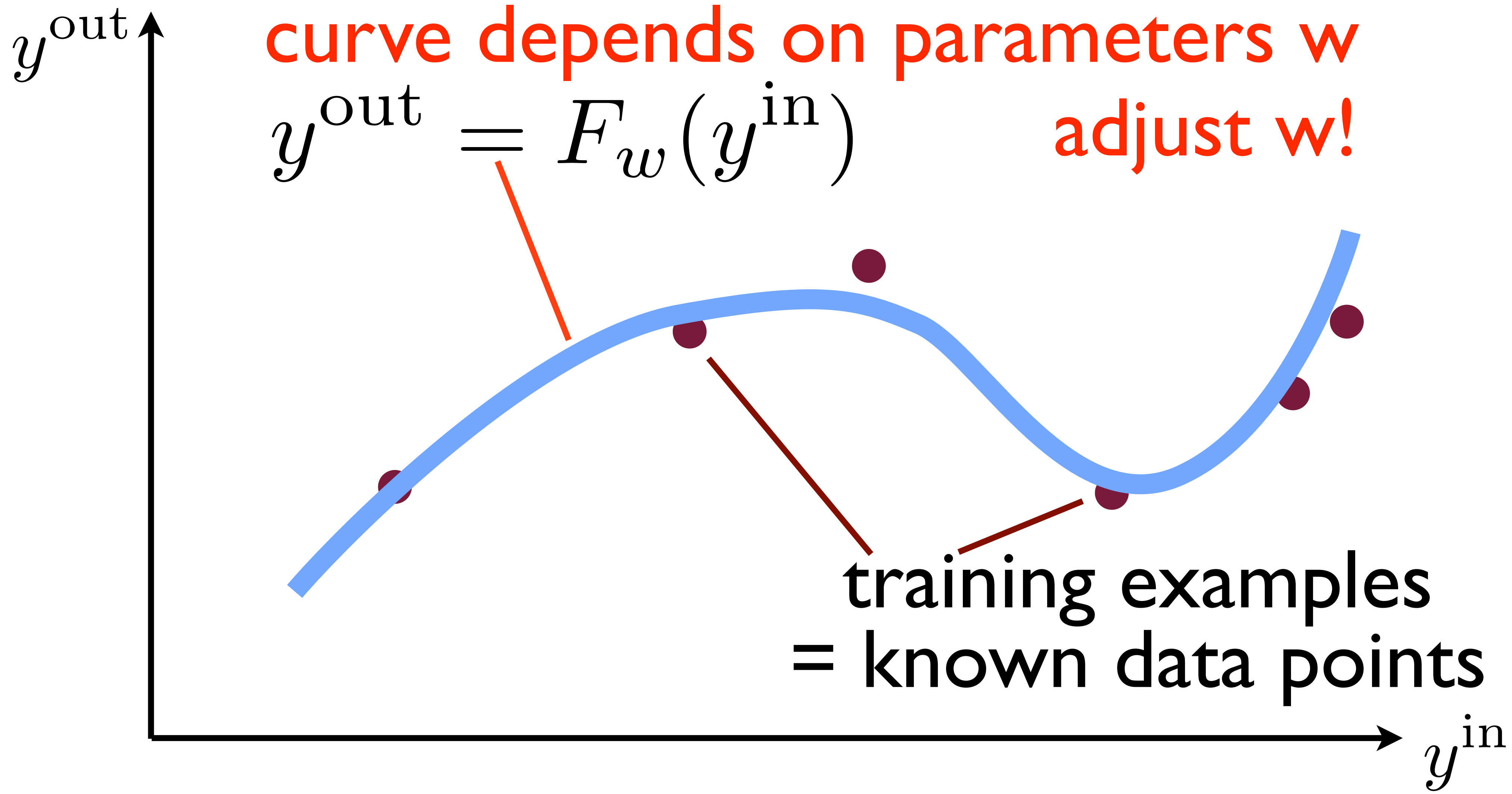
Note: When we write "y^{out}", we mean the whole vector of output values

How to choose the weights (and biases) ?

By "training" with thousands of examples!

This is essentially nonlinear curve fitting!

Example for one output neuron and one input neuron



curve depends on parameters w

$$y^{\mathrm{out}} = F_w(y^{\mathrm{in}})$$

adjust w!

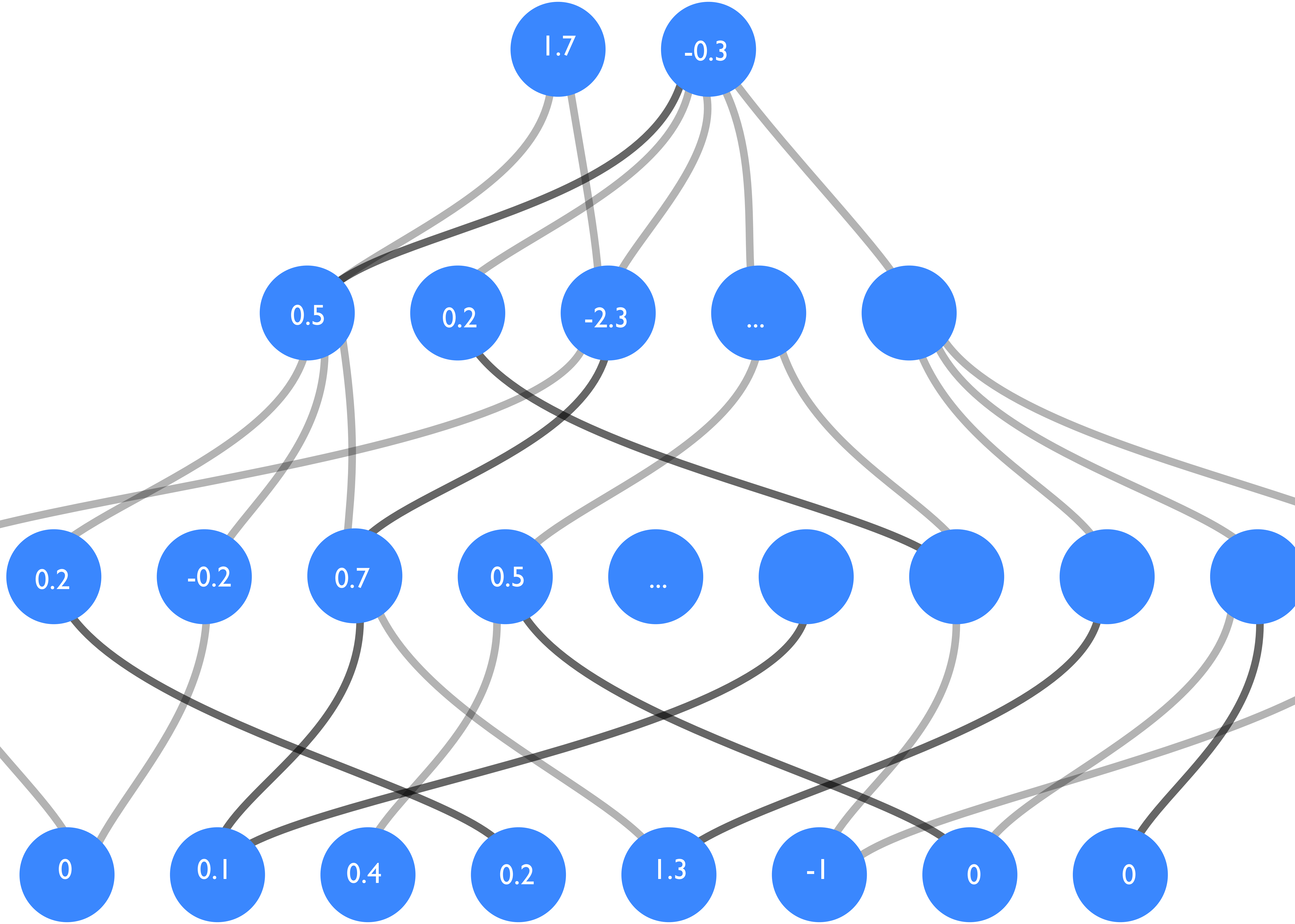
training examples
= known data points

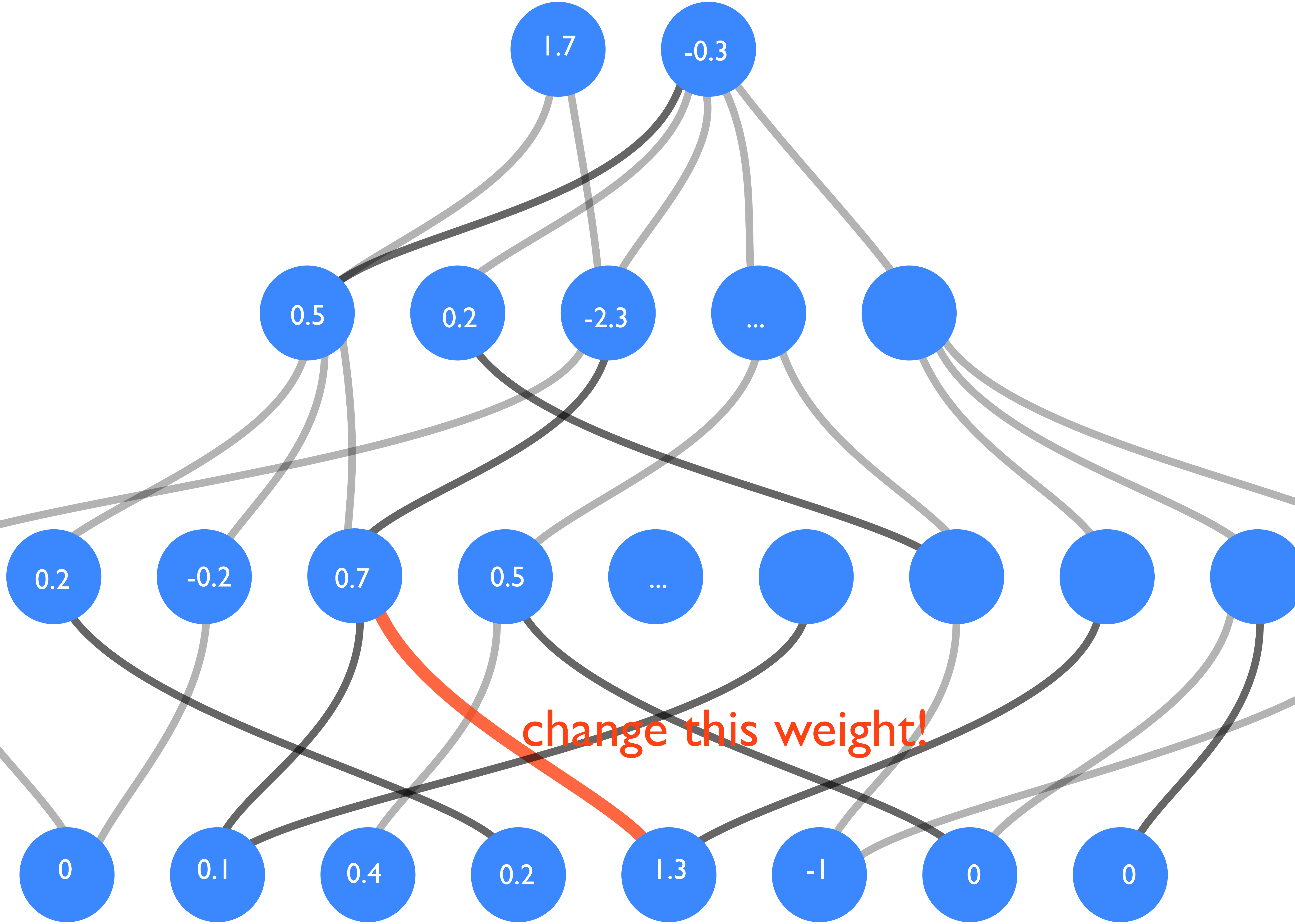
Challenge:

Curve fitting with a million parameters!

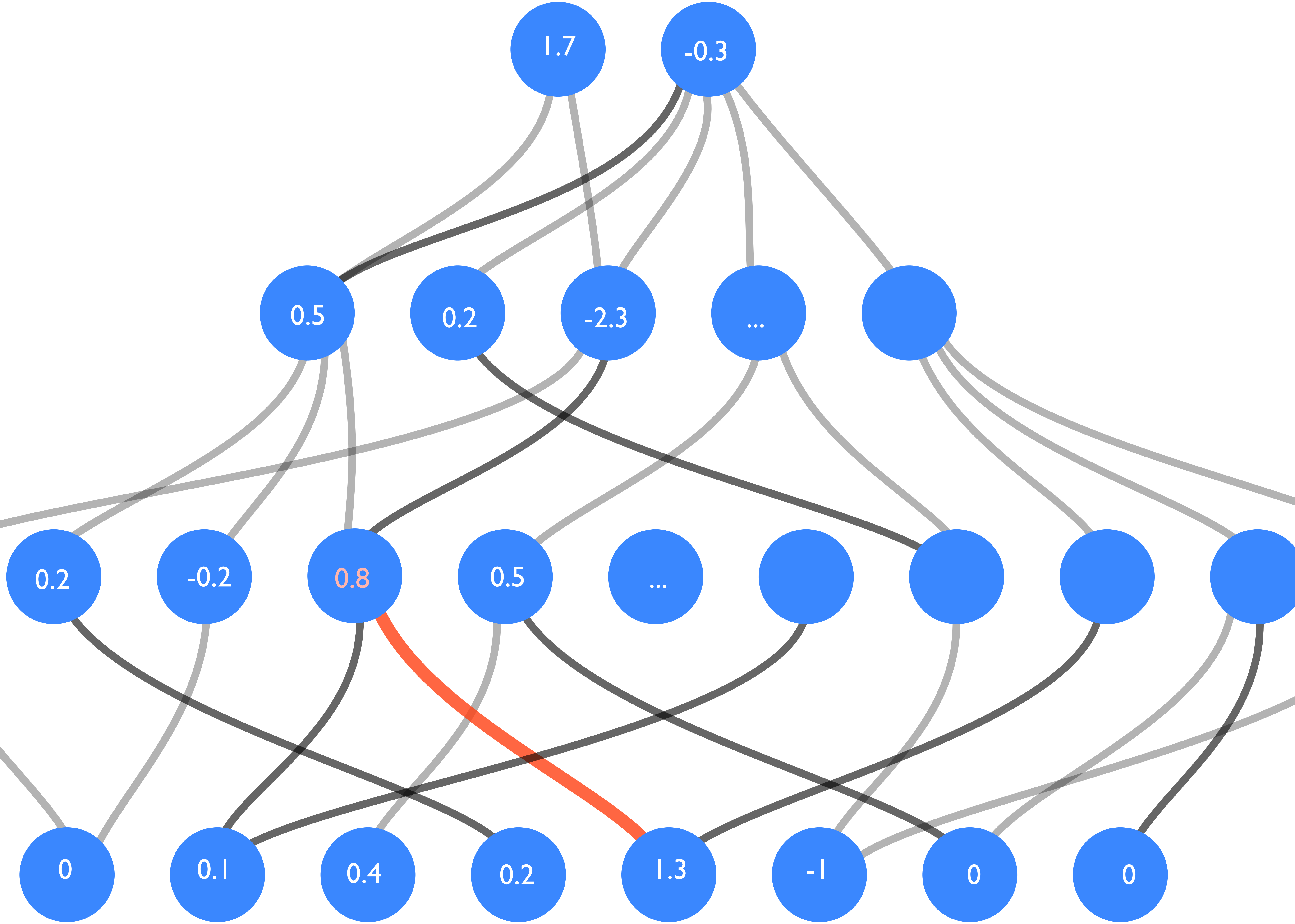
maybe 1000s of input neurons (dimension of y^{in})
many 1000s of hidden layer neurons
millions of weights

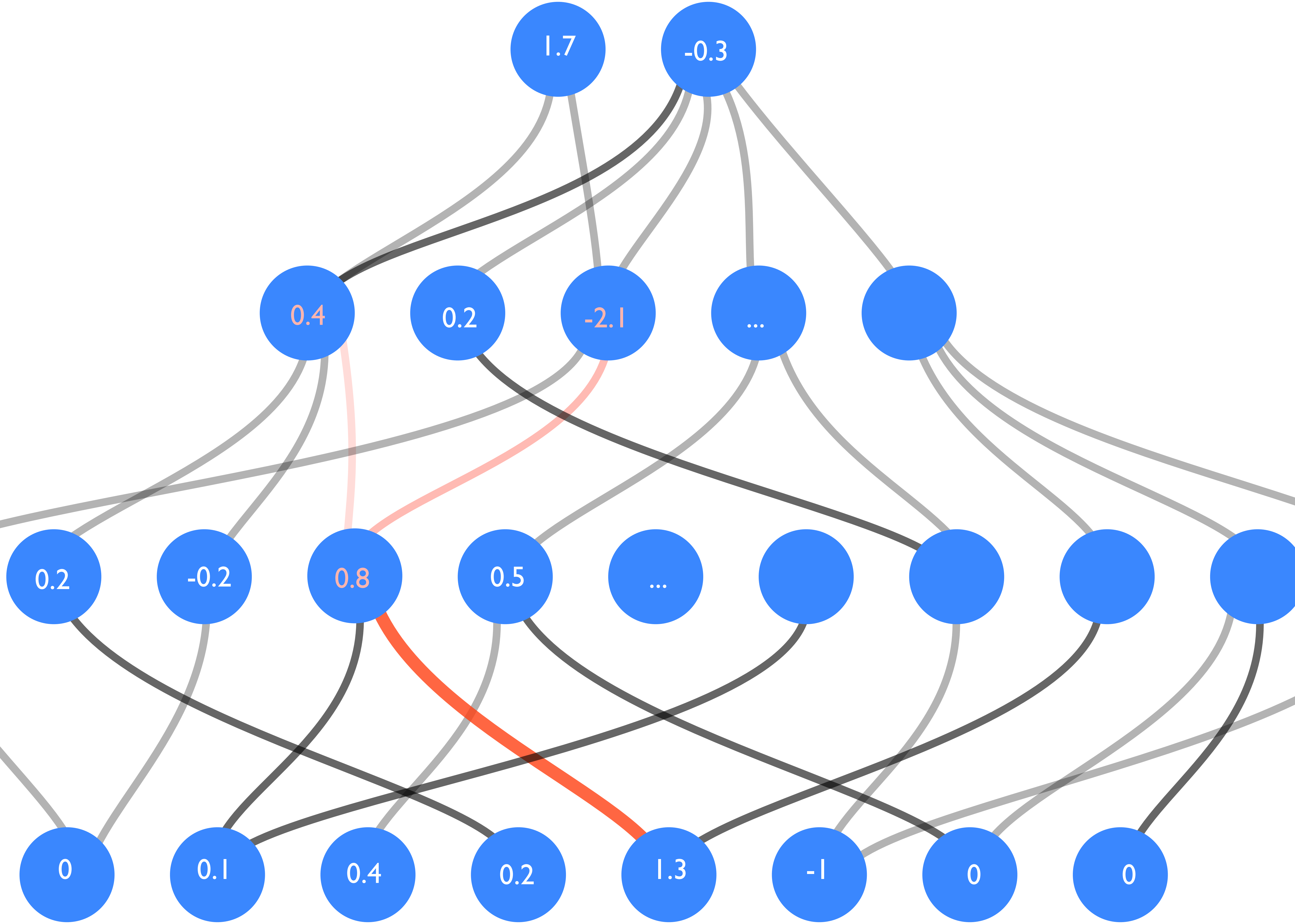
need at least tens of thousands (or more) examples

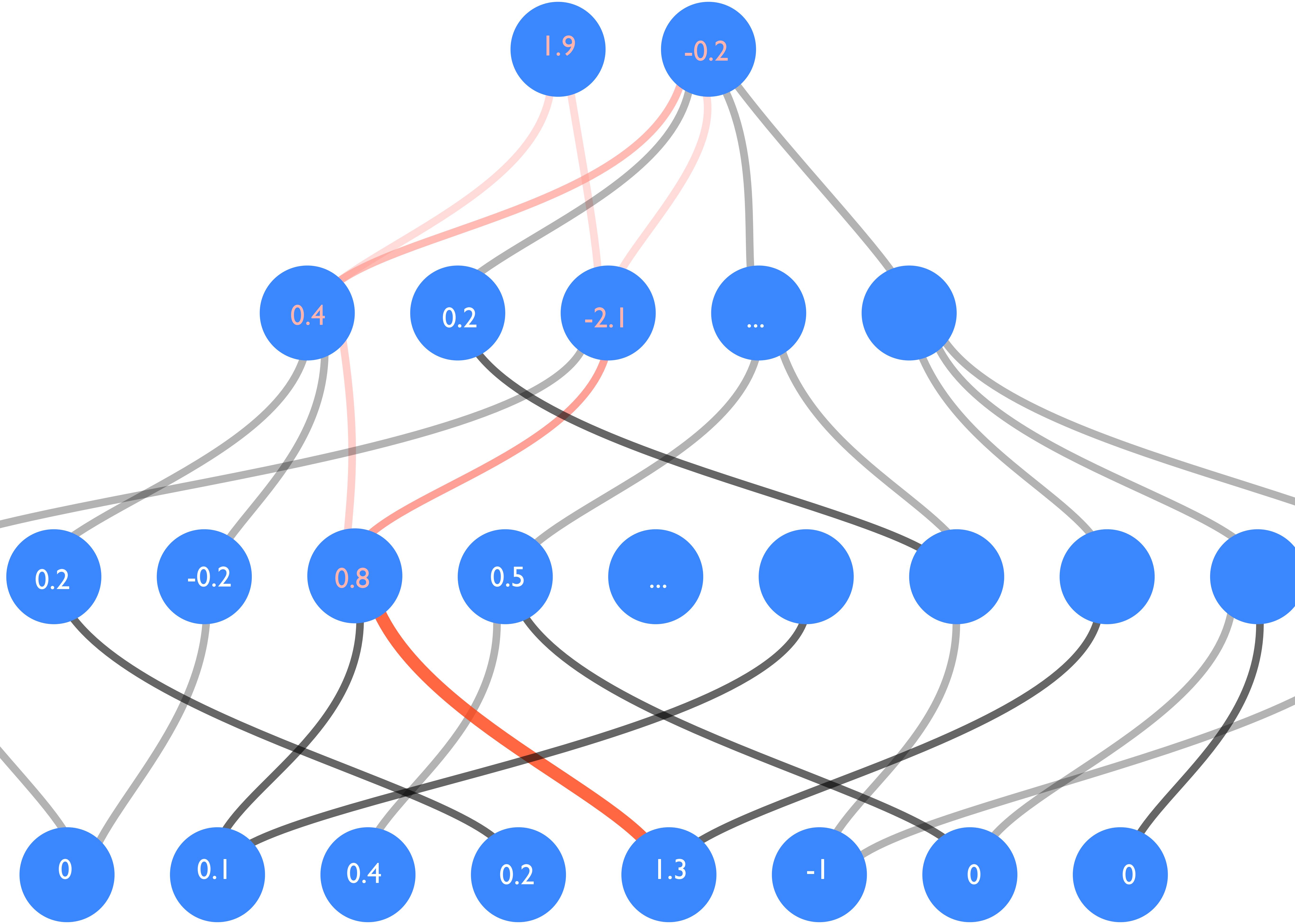

1.7
-0.3
0.5
0.2
-2.3
...
0.2
-0.2
0.7
0.5
...
0
0.1
0.4
0.2
1.3
-1
0
0

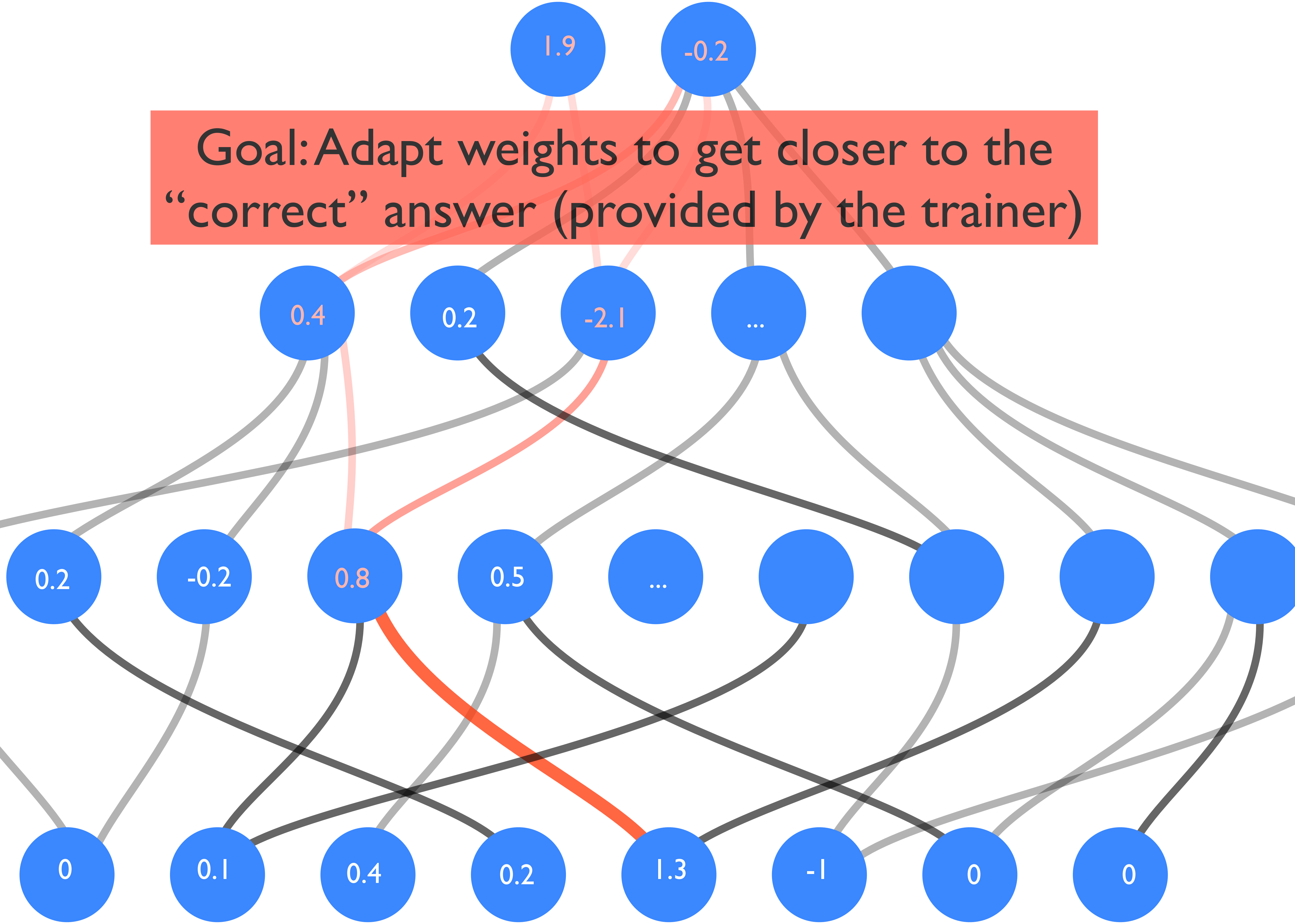


1.7
-0.3
0.5
0.2
-2.3
...
0.2
-0.2
0.7
0.5
...
change this weight!
0
0.1
0.4
0.2
1.3
-1
0
0

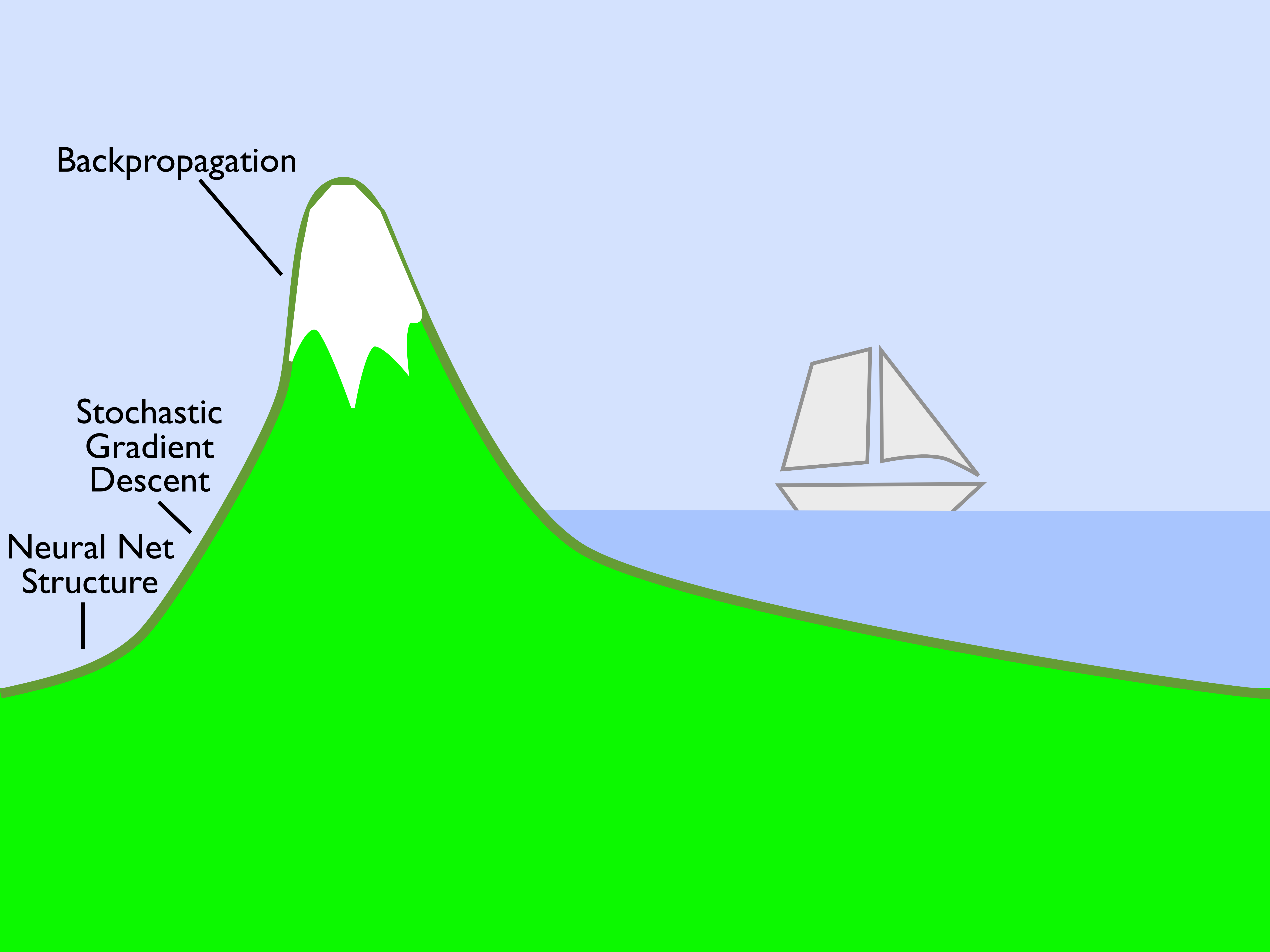

1.7
-0.3
0.5
0.2
-2.3
...
0.2
-0.2
0.8
0.5
...
0
0.1
0.4
0.2
1.3
-1
0
0


1.7
-0.3
0.4
0.2
-2.1
...
0.2
-0.2
0.8
0.5
...
0
0.1
0.4
0.2
1.3
-1
0
0

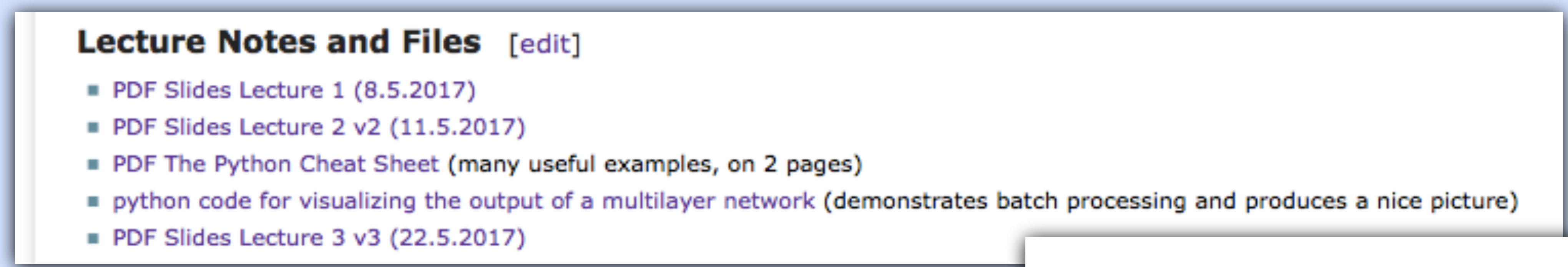

1.9
-0.2
0.4
0.2
-2.1
...
0.2
-0.2
0.8
0.5
...
0
0.1
0.4
0.2
1.3
-1
0
0



Goal: Adapt weights to get closer to the “correct” answer (provided by the trainer)

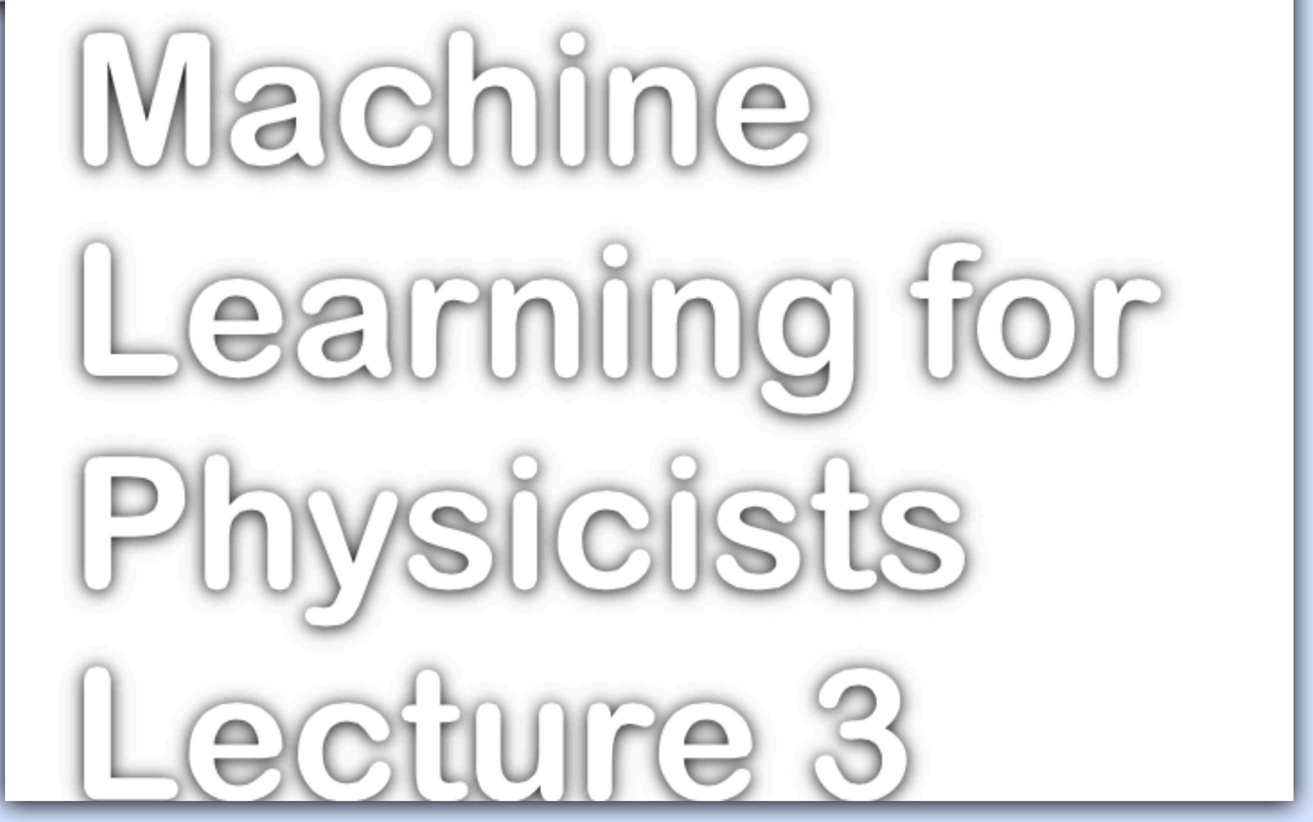

Backpropagation
Stochastic Gradient Descent
Neural Net Structure

http://www.thp2.nat.uni-erlangen.de/index.php/2017_Machine_Learning_for_Physicists,_by_Florian_Marquardt

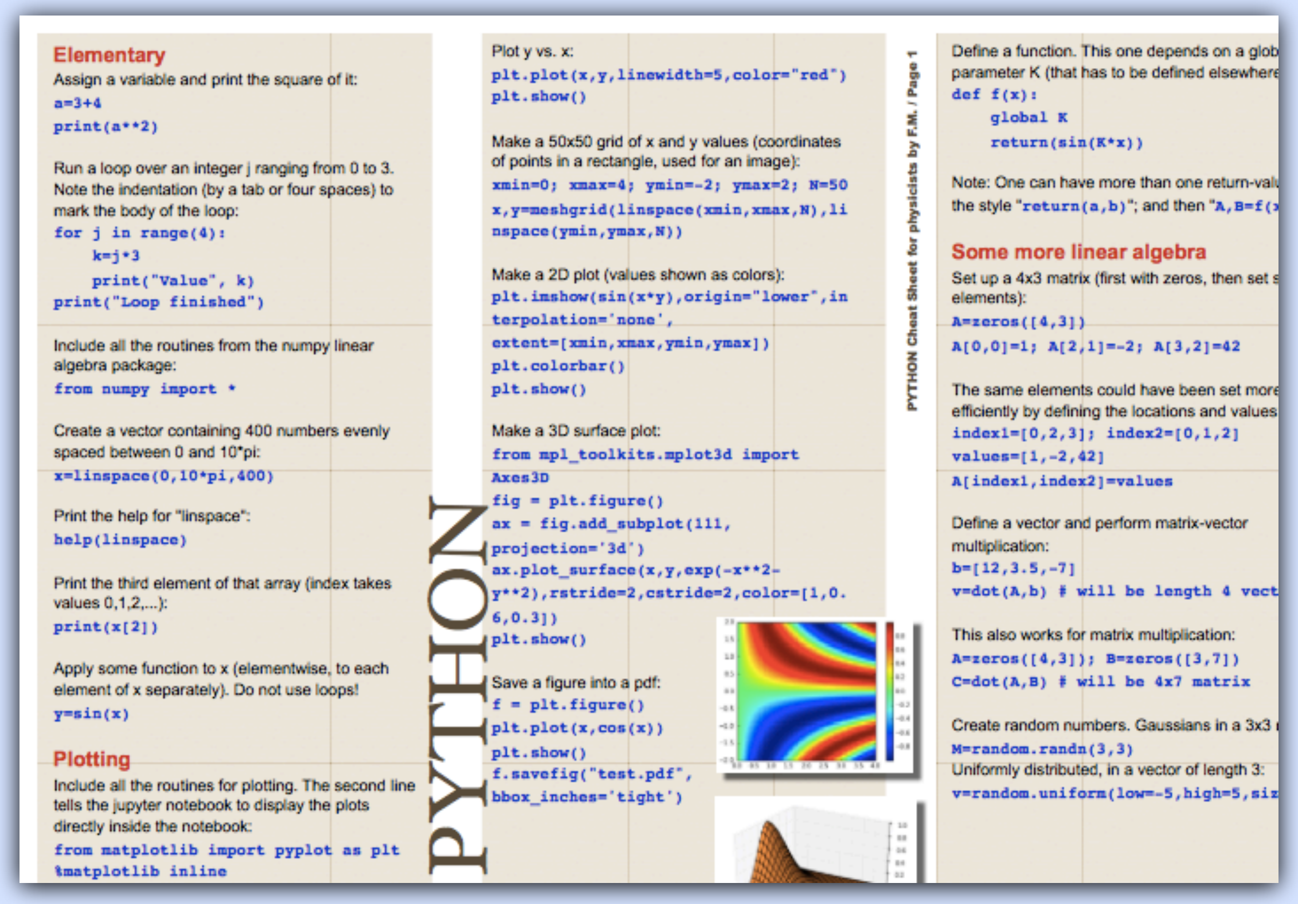


Lecture Notes and Files [edit]

- PDF Slides Lecture 1 (8.5.2017)
- PDF Slides Lecture 2 v2 (11.5.2017)
- PDF The Python Cheat Sheet (many useful examples, on 2 pages)
- python code for visualizing the output of a multilayer network (demonstrates batch processing and produces a nice picture)
- PDF Slides Lecture 3 v3 (22.5.2017)



Machine Learning for Physicists Lecture 3



Elementary

Assign a variable and print the square of it:

```
a=3+4
print(a**2)
```

Run a loop over an integer j ranging from 0 to 3. Note the indentation (by a tab or four spaces) to mark the body of the loop:

```
for j in range(4):
    k=j*3
    print("Value", k)
print("Loop finished")
```

Include all the routines from the numpy linear algebra package:

```
from numpy import *
```

Create a vector containing 400 numbers evenly spaced between 0 and 10*pi:

```
x=linspace(0,10*pi,400)
```

Print the help for "linspace":

```
help(linspace)
```

Print the third element of that array (index takes values 0,1,2,...):

```
print(x[2])
```

Apply some function to x (elementwise, to each element of x separately). Do not use loops!

```
y=sin(x)
```

Plotting

Include all the routines for plotting. The second line tells the jupyter notebook to display the plots directly inside the notebook:

```
from matplotlib import pyplot as plt
%matplotlib inline
```

Plot y vs. x:

```
plt.plot(x,y,linewidth=5,color="red")
plt.show()
```

Make a 50x50 grid of x and y values (coordinates of points in a rectangle, used for an image):

```
xmin=0; xmax=4; ymin=-2; ymax=2; N=50
x,y=meshgrid(linspace(xmin,xmax,N),li
nspace(ymin,ymax,N))
```

Make a 2D plot (values shown as colors):

```
plt.imshow(sin(x*y),origin="lower",in
terpolation='none',
extent=[xmin,xmax,ymin,ymax])
plt.colorbar()
plt.show()
```

Make a 3D surface plot:

```
from mpl_toolkits.mplot3d import
Axes3D
fig = plt.figure()
ax = fig.add_subplot(111,
projection='3d')
ax.plot_surface(x,y,exp(-x**2-
y**2),rstride=2,cstride=2,color=[1,0.
6,0.3])
plt.show()
```

Save a figure into a pdf:

```
f = plt.figure()
plt.plot(x,cos(x))
plt.show()
f.savefig("test.pdf",
bbox_inches='tight')
```

Define a function. This one depends on a glob
parameter K (that has to be defined elsewhere

```
def f(x):
    global K
    return(sin(K*x))
```

Note: One can have more than one return-valu
the style "`return(a,b)`"; and then "`A,B=f(x`

Some more linear algebra

Set up a 4x3 matrix (first with zeros, then set s
elements):

```
A=zeros([4,3])
A[0,0]=1; A[2,1]=-2; A[3,2]=42
```

The same elements could have been set more
efficiently by defining the locations and values

```
index1=[0,2,3]; index2=[0,1,2]
values=[1,-2,42]
A[index1,index2]=values
```

Define a vector and perform matrix-vector multiplication:

```
b=[12,3.5,-7]
v=dot(A,b) # will be length 4 vect
```

This also works for matrix multiplication:

```
A=zeros([4,3]); B=zeros([3,7])
C=dot(A,B) # will be 4x7 matrix
```

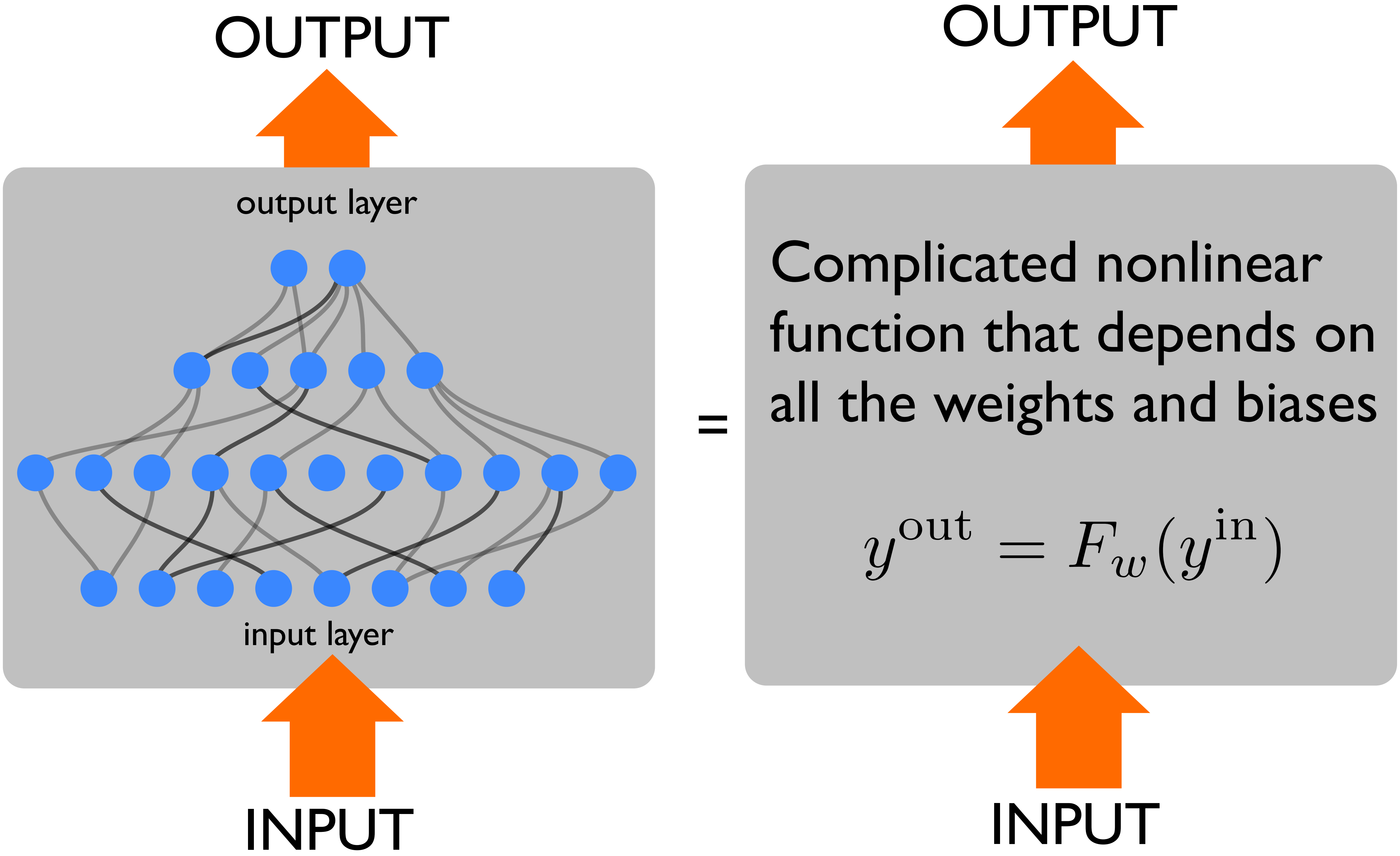
Create random numbers. Gaussians in a 3x3

```
M=random.randn(3,3)
```

Uniformly distributed, in a vector of length 3:

```
v=random.uniform(low=-5,high=5,siz
```

A neural network



Complicated nonlinear function that depends on all the weights and biases

$$y^{\mathrm{out}} = F_w(y^{\mathrm{in}})$$

Note: When we write “w” as subscript of F, we mean all the weights and also biases

Note: When we write “y^{out}”, we mean the whole vector of output values

We have:

$$y^{\text{out}} = F_w(y^{\text{in}})$$

neural network
(w here also stands for the biases)

We would like:

$$y^{\text{out}} \approx F(y^{\text{in}})$$

desired “target” function

Cost function measures deviation:

$$C(w) = \frac{1}{2}\langle \| \, F_w(y^{\text{in}}) - F(y^{\text{in}}) \, \|^2 \rangle$$

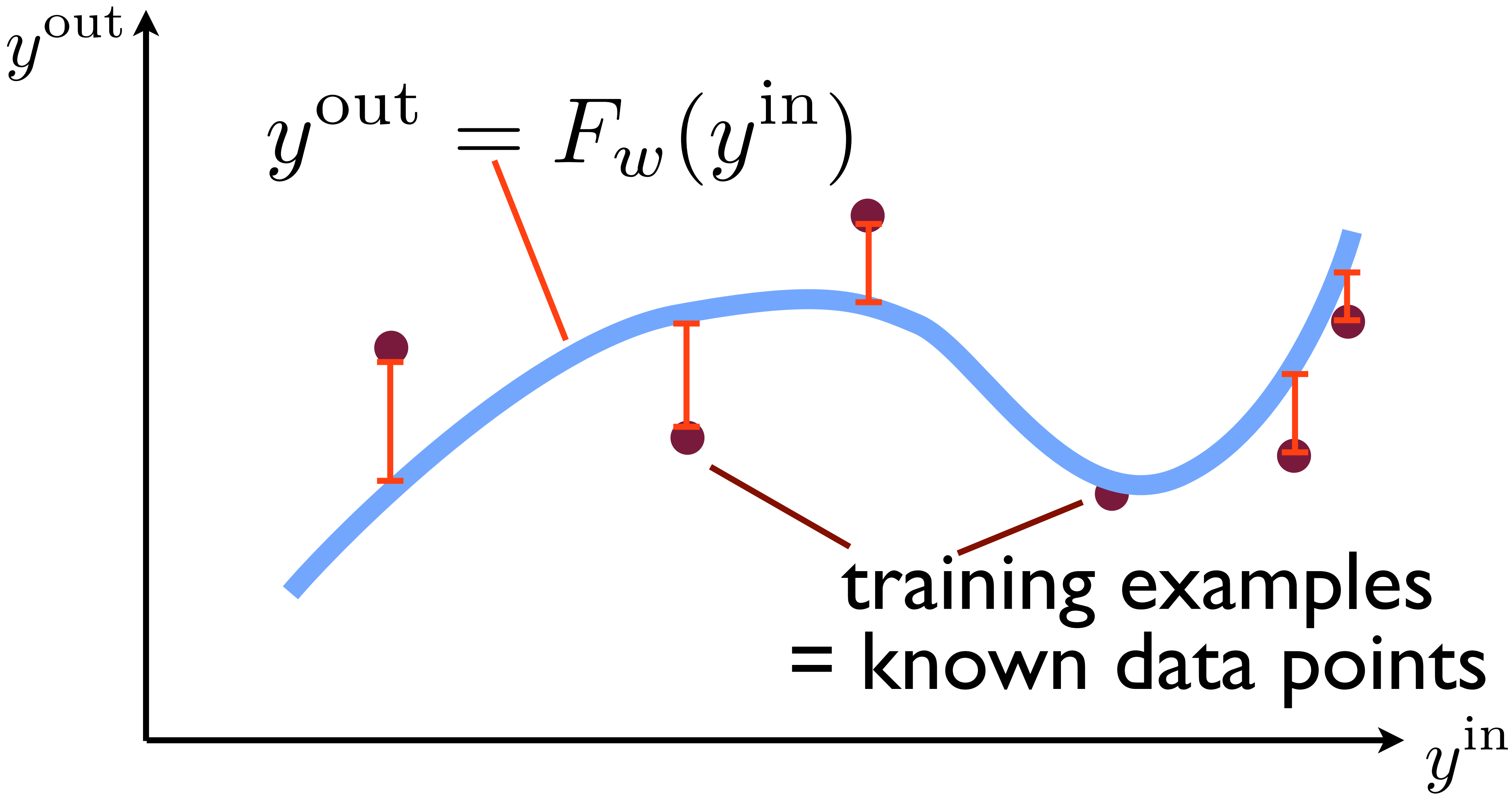
vector norm

average over
all samples

Approximate version, for N samples:

$$C(w) \approx \frac{1}{2}\frac{1}{N}\sum_{s=1}^{N} \| F_w(y^{(s)}) - F(y^{(s)}) \|^2$$

s=index of sample

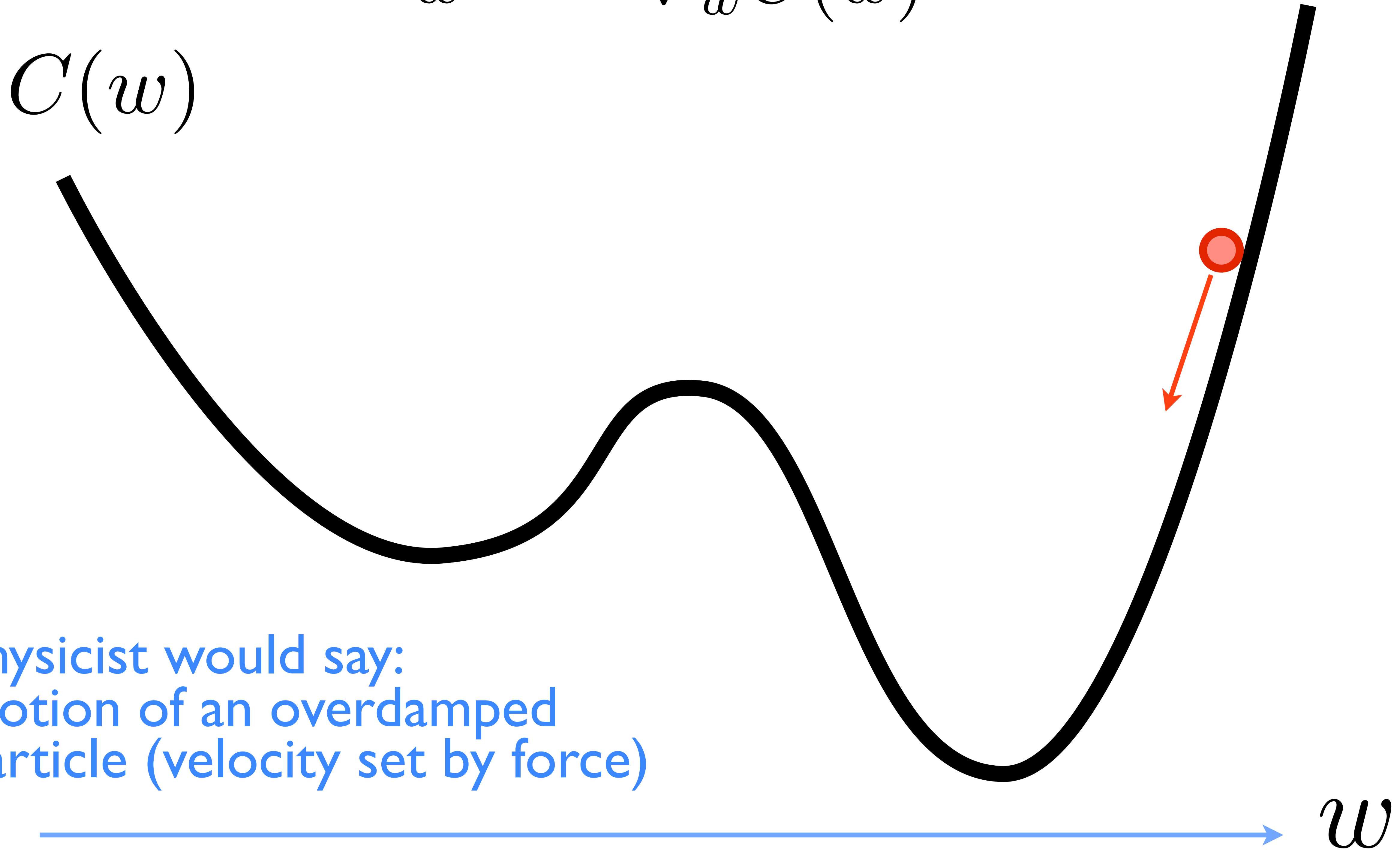


training examples
= known data points

Minimizing C for this case: "least-squares fitting"!

Method: "Sliding down the hill" ("gradient descent")

$$\dot{w} \sim -\nabla_w C(w)$$



physicist would say:
motion of an overdamped
particle (velocity set by force)

Stochastic Gradient Descent

Problem: Evaluating C would mean averaging over ALL training samples

Solution: Only average over a few samples, get approximate C

Discrete steps: for each step evaluate a few samples and update weights according to:

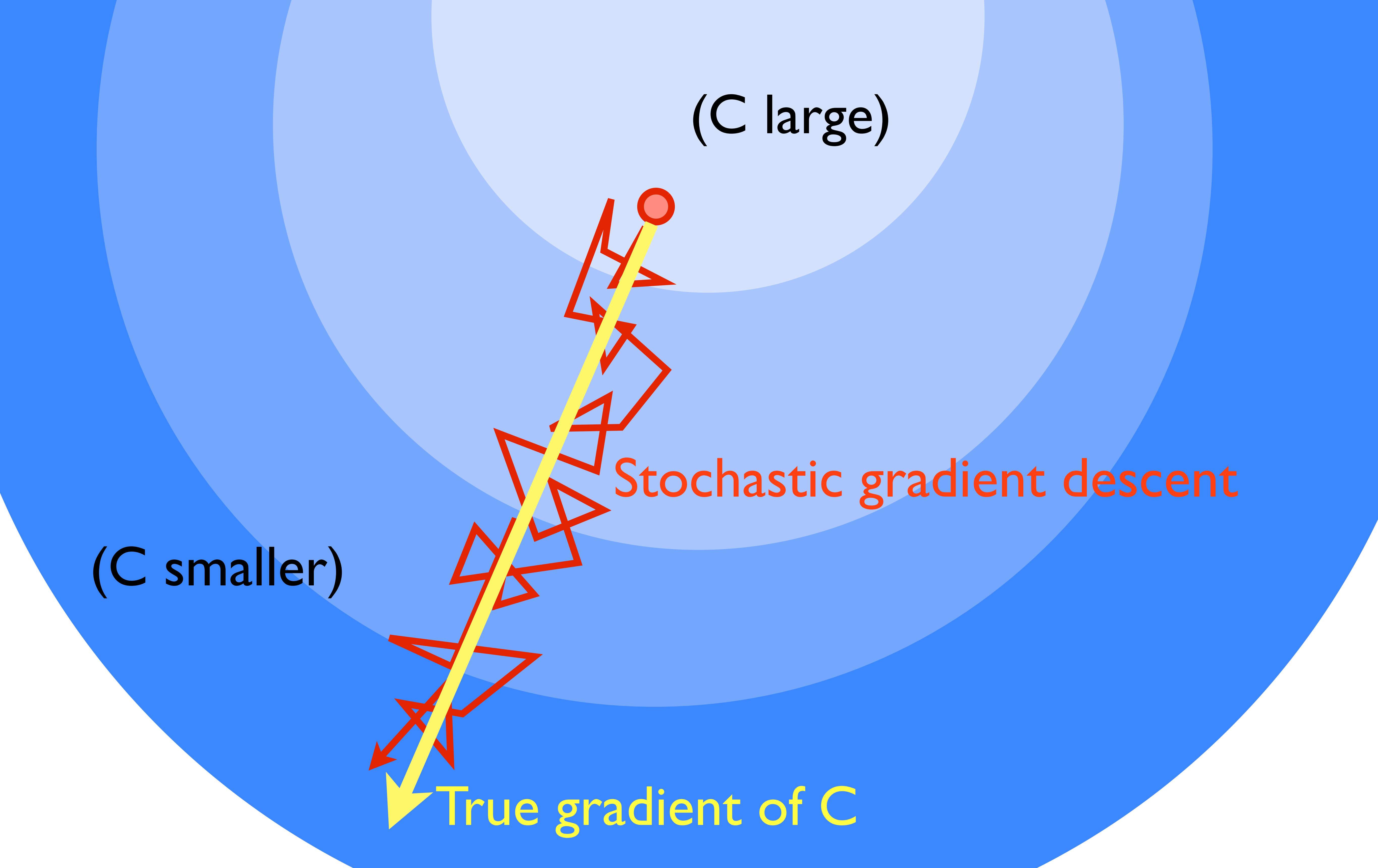
approximate version of C

$$w_j \mapsto w_j - \eta \frac{\partial \tilde{C}(w)}{\partial w_j}$$

(take different samples in each step!)

stepsize parameter

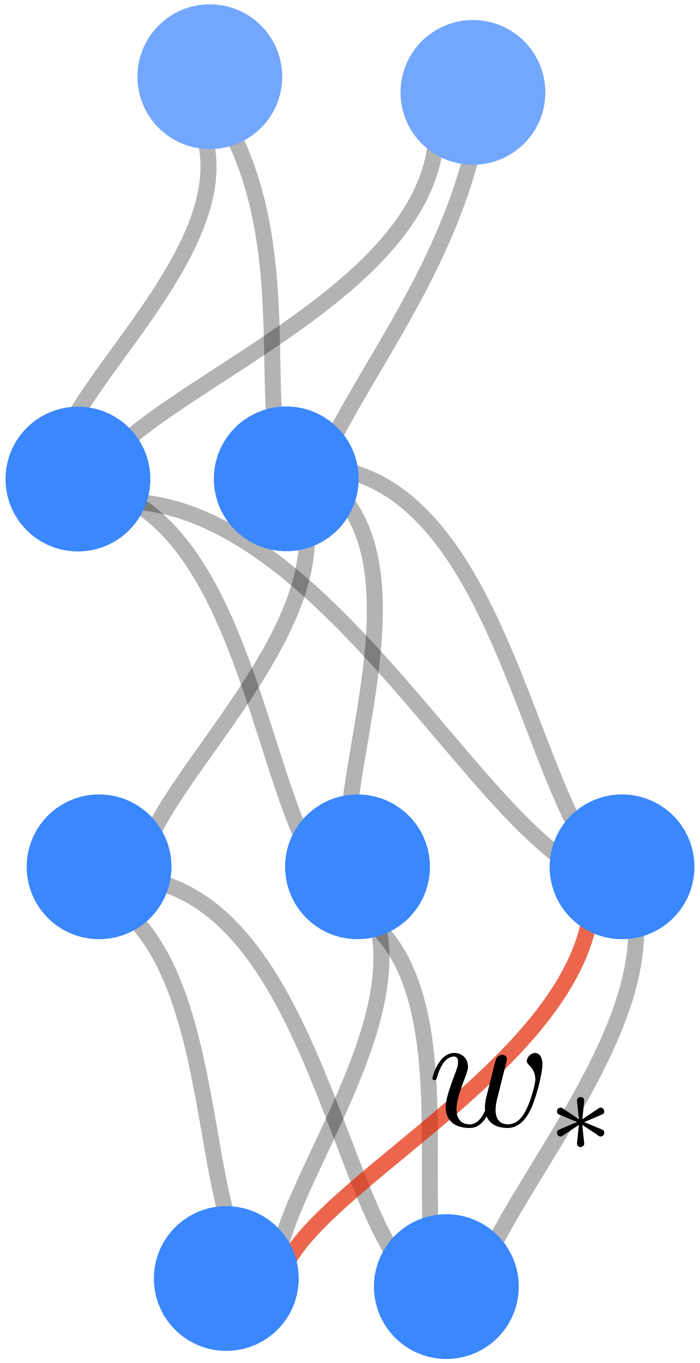
(Note: just as before, the biases b are included here, think of them as extra parameters w)



For sufficiently small steps: sum over many steps approximates true gradient (because it is an additional average)

$$\frac{\partial C(w)}{\partial w_*} = ?$$

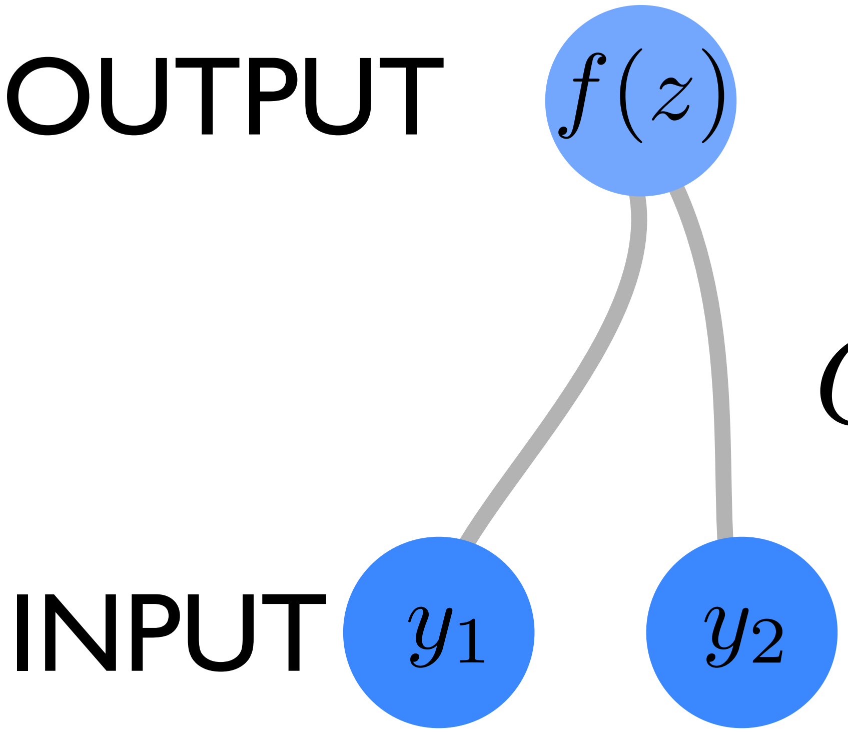
some weight (or bias), somewhere in the net



It’s time to use the chain rule!

(image: Wikimedia)

Small network: Calculate derivative of cost function “by hand”



$$z = w_1 y_1 + w_2 y_2 + b$$

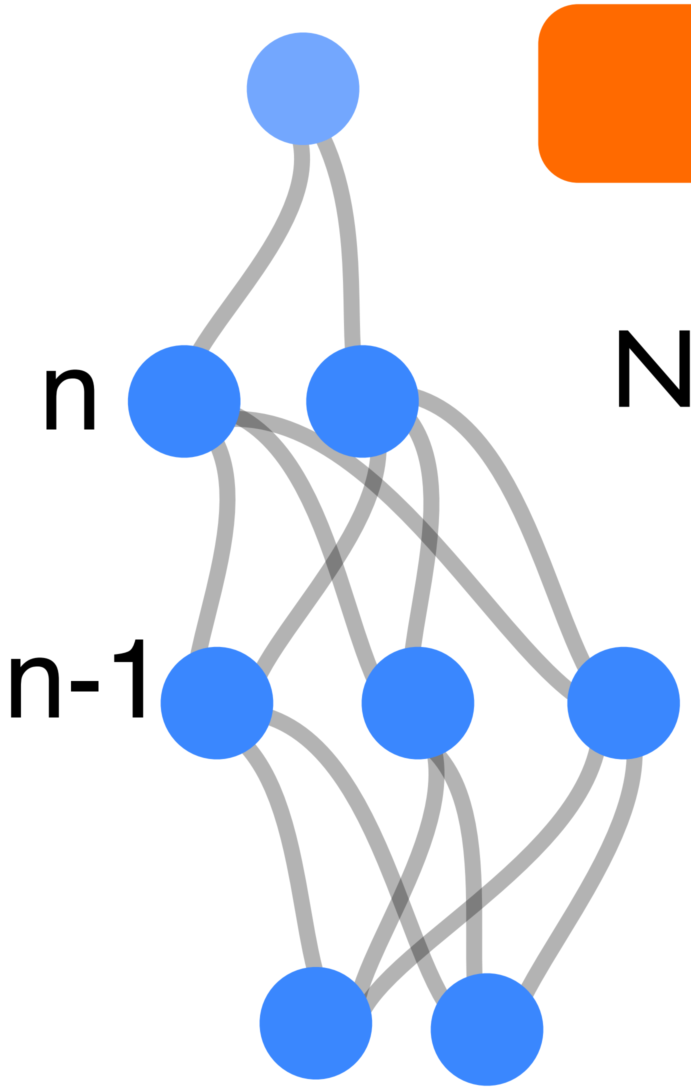
$$C(w) = \frac{1}{2} \left\langle (f(z) - F(y_1, y_2))^2 \right\rangle$$

cost network output desired output

$$\frac{\partial C}{\partial w_1} = \left\langle (f(z) - F) f'(z) \frac{\partial z}{\partial w_1} \right\rangle$$

$$\frac{\partial z}{\partial w_1} = y_1$$

Backpropagation



Now for the full network!

Need to keep track of indices carefully:

$y_j^{(n)}$	Value of neuron j in layer n
$z_j^{(n)}$	Input value for "y=f(z)"
$w_{jk}^{n,n-1}$	Weight (neuron k in layer n-1 feeding into neuron j in layer n)

Backpropagation

We have:

$$C(w) = \langle \underline{C(w, y^{\mathrm{in}})} \rangle$$

cost value for one particular input

We get:

$$\frac{\partial C(w, y^{\mathrm{in}})}{\partial w_*} = \sum_j (y_j^{(n)} - F_j(y^{\mathrm{in}})) \frac{\partial y_j^{(n)}}{\partial w_*}$$

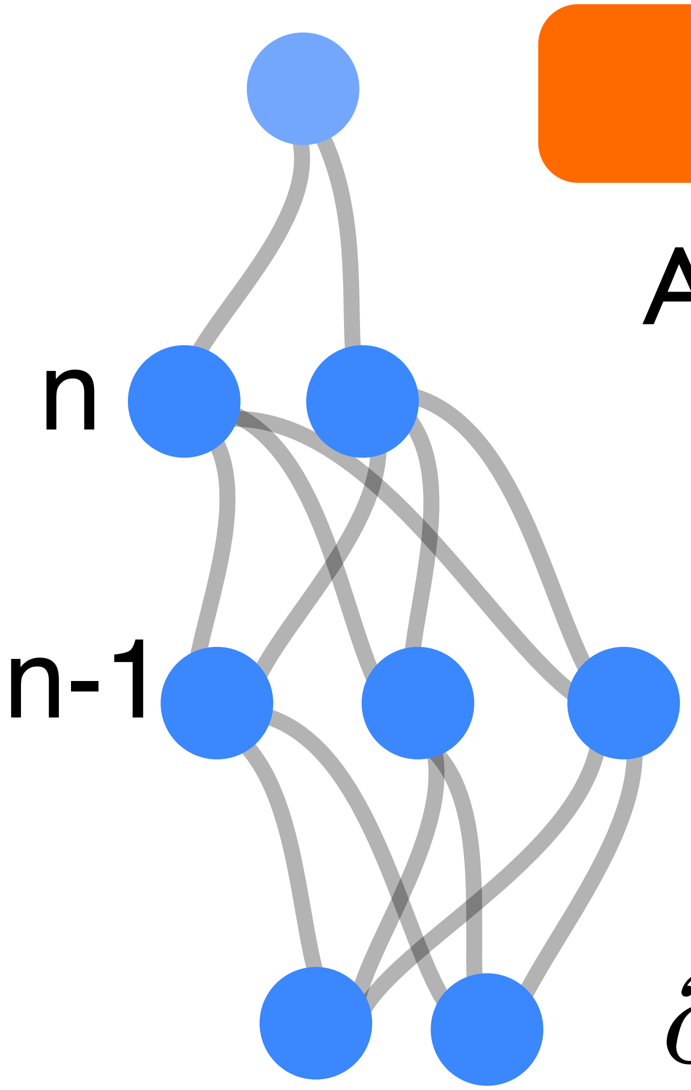
$$= \sum_j (y_j^{(n)} - F_j(y^{\mathrm{in}})) f'(z_j^{(n)}) \frac{\partial z_j^{(n)}}{\partial w_*}$$

some weight (or bias), somewhere in the net

=?

(we used:)

$$y_j^{(n)} = f(z_j^{(n)})$$



Backpropagation

Apply chain rule repeatedly

We want: Change of neuron j in layer n due to change of some arbitrary weight w_* :

$$\frac{\partial z_j^{(n)}}{\partial w_*} = \sum_k \frac{\partial z_j^{(n)}}{\partial y_k^{(n-1)}} \frac{\partial y_k^{(n-1)}}{\partial w_*}$$
$$= \sum_k w_{jk}^{n,n-1} f'(z_k^{(n-1)}) \frac{\partial z_k^{(n-1)}}{\partial w_*}$$

And now: the same again (recursion)

Backpropagation

$$\frac{\partial z_j^{(n)}}{\partial w_*} = \sum_k w_{jk}^{n,n-1} f'(z_k^{(n-1)}) \frac{\partial z_k^{(n-1)}}{\partial w_*}$$

Important insight: Each pair of layers [n,n-1] contributes multiplication with the following **matrix**:

$$M_{jk}^{(n,n-1)} = w_{jk}^{(n,n-1)} f'(z_k^{(n-1)})$$

Backpropagation

Repeated matrix multiplication, going down the net:

$$\frac{\partial z_j^{(n)}}{\partial w_*} = \sum_{k,l,\ldots,u,v} M_{jk}^{n,n-1} M_{kl}^{n-1,n-2} \ldots M_{uv}^{\tilde{n}+1,\tilde{n}} \frac{\partial z_v^{(\tilde{n})}}{\partial w_*}$$

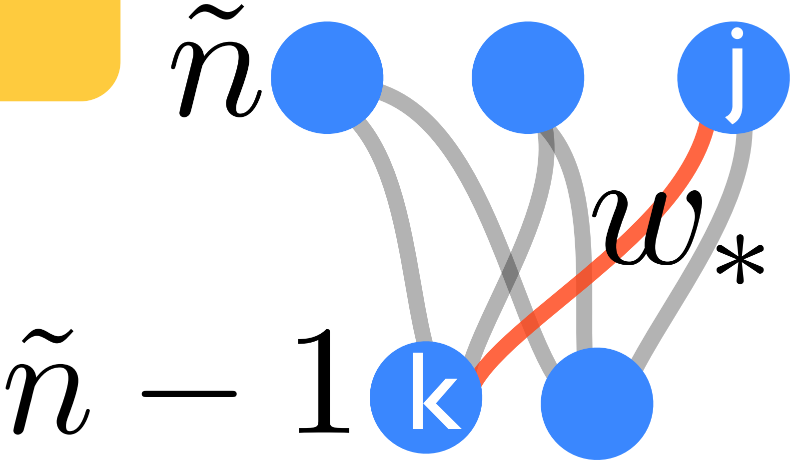
Backpropagation

What happens when we finally encounter the weight with respect to which we wanted to calculate the derivative of the cost function?

If w_* was really a weight:

w_*

$$\frac{\partial z_j^{(\tilde{n})}}{\partial w_{jk}^{\tilde{n},\tilde{n}-1}} = y_k^{(\tilde{n}-1)}$$



...if it was a bias:

w_*

$$\frac{\partial z_j^{(\tilde{n})}}{\partial b_j^{\tilde{n}}} = 1$$

Backpropagation

We have:

$$C(w) = \langle \underbrace{C(w, y^{\text{in}})}_{} \rangle$$

cost value for one particular input

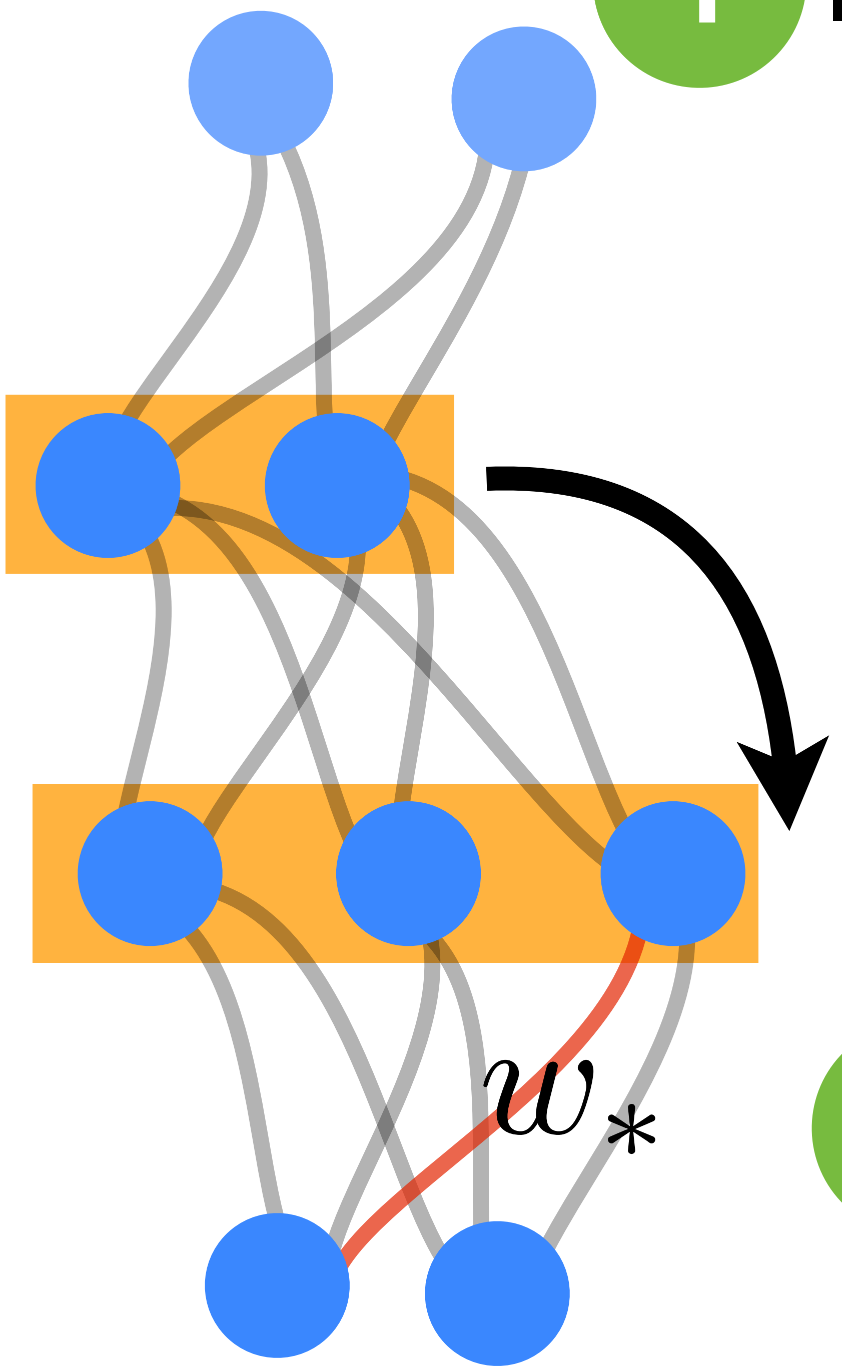
In total, we get:

$$\frac{\partial C(w, y^{\text{in}})}{\partial w_*} = \sum_j (y_j^{(n)} - F_j(y^{\text{in}})) \frac{\partial y_j^{(n)}}{\partial w_*}$$

$$= \sum_j (y_j^{(n)} - F_j(y^{\text{in}})) f'(z_j^{(n)}) \frac{\partial z_j^{(n)}}{\partial w_*}$$

How to evaluate this: construct vector for output layer n, and then multiply with matrices from the right (as shown above)

Backpropagation Summary



1 Initialize vector from output layer:

$$\Delta_j = (y_j^n - F_j(y^{\text{in}}))f'(z_j^n)$$

2 For each layer: store outcomes (cost derivatives) for all weights and biases w_* in that layer

$$\frac{\partial C(w, y^{\text{in}})}{\partial w_*} = \Delta_j \frac{\partial z_j^{(n)}}{\partial w_*}$$

(see above)

(j is the index where this particular weight appears)

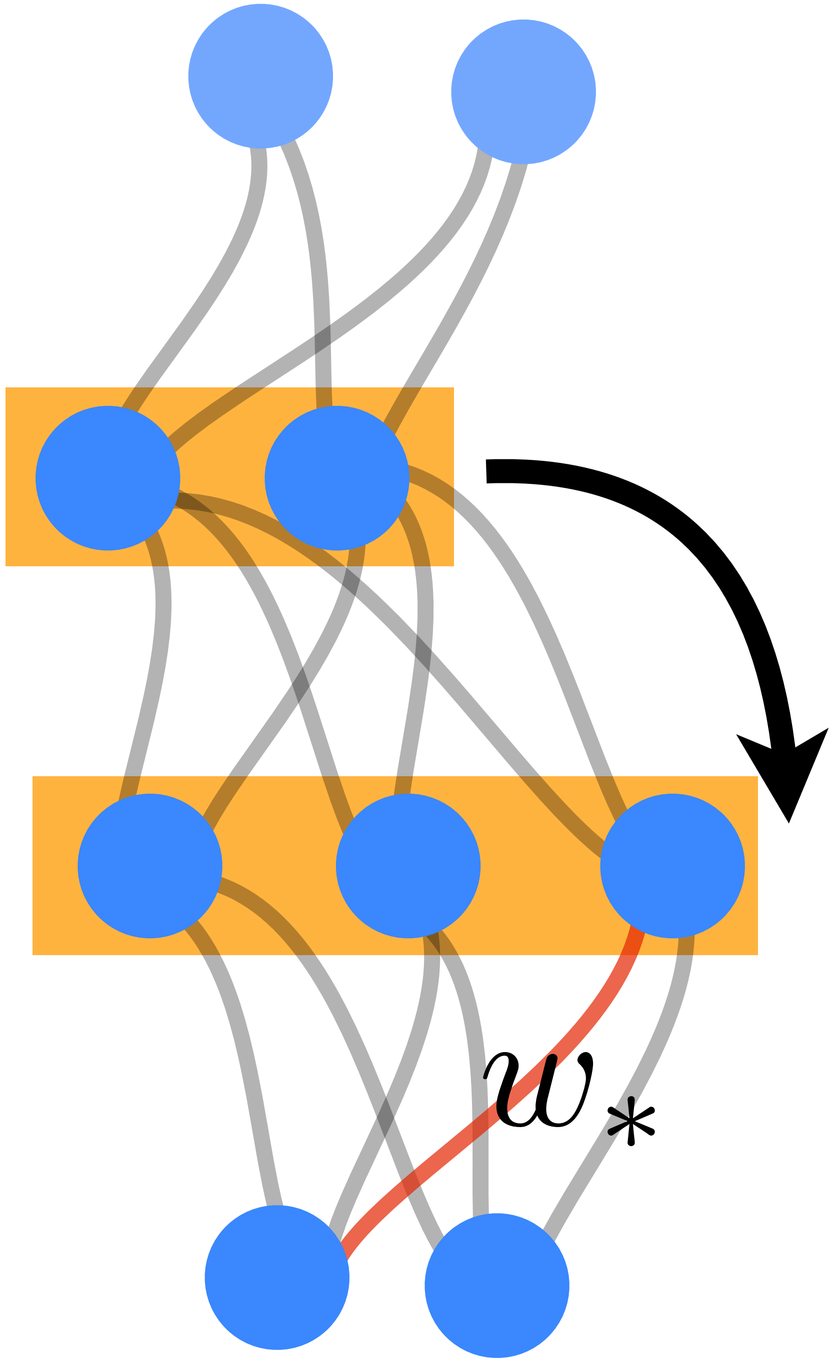
3 Multiply vector by matrix

$$\Delta_k^{\text{new}} = \sum_j \Delta_j M_{jk}^{n,n-1}$$

(see above for M)

(& return to step 2)

Backpropagation



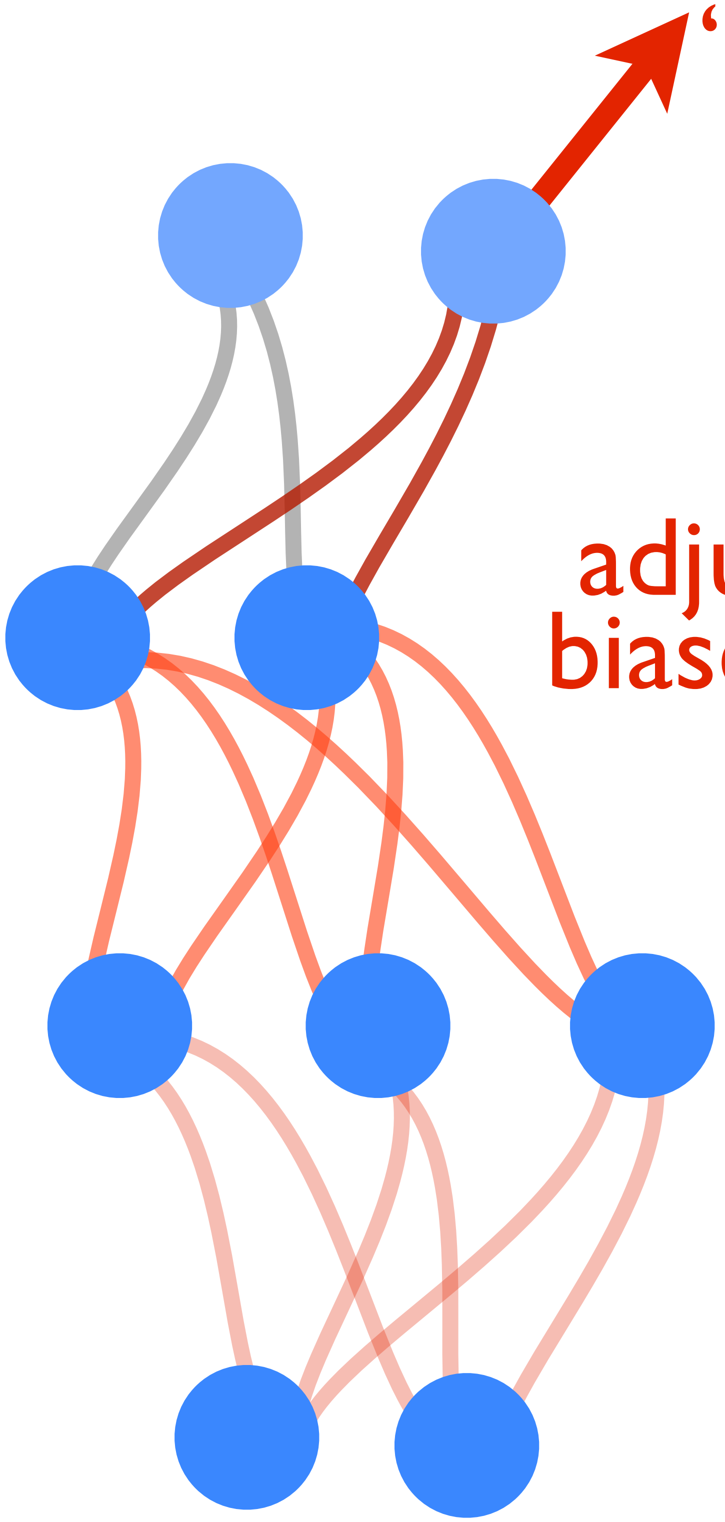
Very efficient: One single backpropagation pass through the network yields ALL the derivatives of C with respect to all the weights and biases!

No more effort than forward propagation!

Huge ("million-fold") advantage over naive approach of calculating numerically derivatives for all weights individually!

Backpropagation

Physical
intuitive
picture:



“force” tries to pull into direction of correct outcome

adjusts all weights (& biases) in layers below

Backpropagation

In each layer:

$$\frac{\partial C(w, y^{\mathrm{in}})}{\partial w_*} = \Delta_j \frac{\partial z_j^{(n)}}{\partial w_*}$$

Weight:

$$\frac{\partial z_j^{(n)}}{\partial w_{jk}^{n,n-1}} = y_k^{(n-1)}$$

Bias:

$$\frac{\partial z_j^{(n)}}{\partial b_j^n} = 1$$

— Averaging over samples: —

$$\frac{\partial C(w)}{\partial w_{jk}^{n,n-1}} = \left\langle \Delta_j y_k^{(n-1)} \right\rangle$$

$$\frac{\partial C(w)}{\partial b_j^n} = \langle \Delta_j \rangle$$

Implementation

We are doing batch processing of many samples!

Variable	Dimensions
`y[layer]`	batchsize x neurons[layer]
`Delta`	batchsize x neurons[layer]
`Weights[layer]`	neurons[lower layer] x neurons[layer]
`Biases[layer]`	neurons[layer]

$$\frac{\partial C(w)}{\partial w_{jk}^{n,n-1}} = \left\langle \Delta_j y_k^{(n-1)} \right\rangle \qquad \frac{\partial C(w)}{\partial b_j^n} = \langle \Delta_j \rangle$$

averaging: sum over batch index!

```
dWeights[layer]=dot(transpose(y[lower layer]),Delta)/batchsize
```

neurons[lower layer] x batchsize

```
dBiases[layer]=Delta.sum(0)/batchsize
```

(summation over index 0=batch index)

Implementation

We are doing batch processing of many samples!

Variable	Dimensions
`y[layer]`	batchsize x neurons[layer]
`Delta`	batchsize x neurons[layer]
`Weights[layer]`	neurons[lower layer] x neurons[layer]
`Biases[layer]`	neurons[layer]

$$\Delta_k^{\text{new}} = \sum_j \Delta_j M_{jk}^{n,n-1}$$

with: $$M_{jk}^{(n,n-1)} = w_{jk}^{(n,n-1)} f'(z_k^{(n-1)})$$

Take step from 'layer' down to 'lower layer':

```
Delta=dot(Delta,transpose(Weights))*df_layer[lower layer]
```

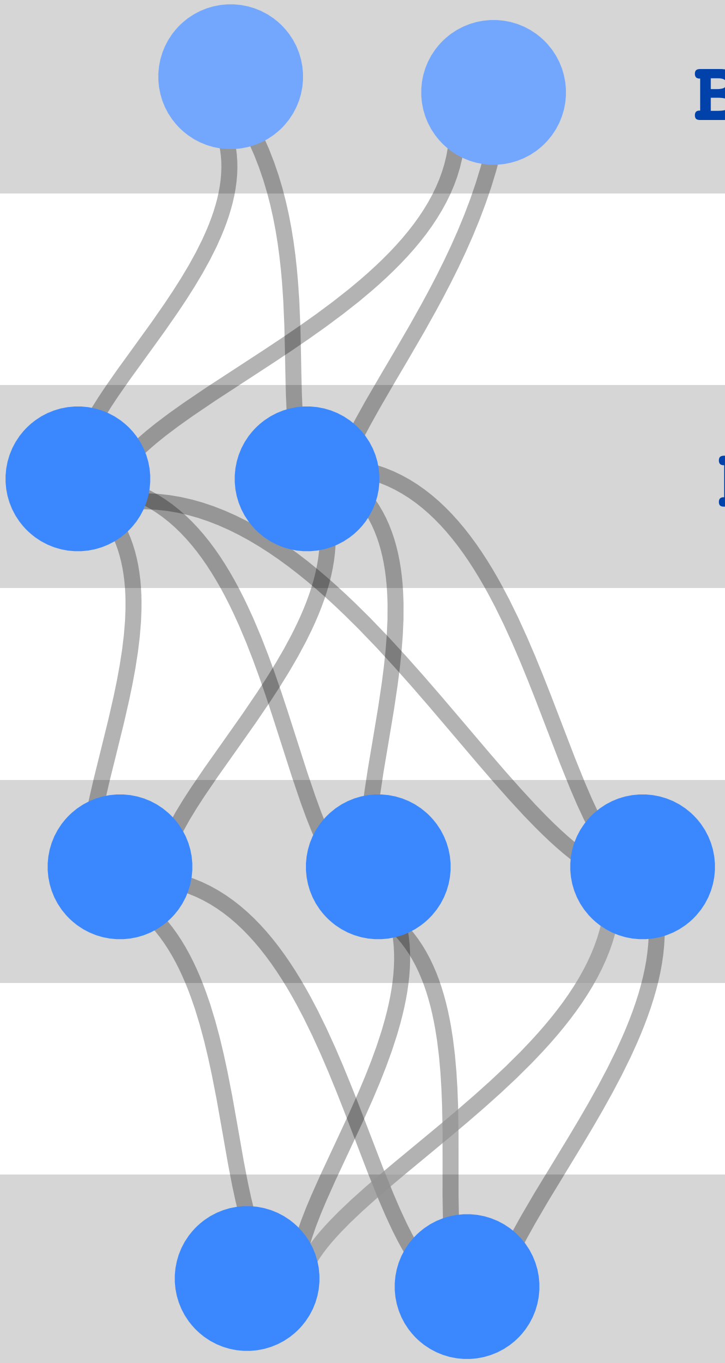
batchsize x neurons[lower layer]

f'(z) in lower layer

(first dimension will be expanded)

Implementation

here: NumLayers=3 (count all, except input)



`Biases[2]` **`y_layer[3]`** **`df_layer[2]`**

`Weights[2]` (stores f'(z))

`Biases[1]` **`y_layer[2]`** **`df_layer[1]`**

`Weights[1]`

`Biases[0]` **`y_layer[1]`** **`df_layer[0]`**

`Weights[0]` (here: a 2x3 matrix)

`y_layer[0]`

Implementation

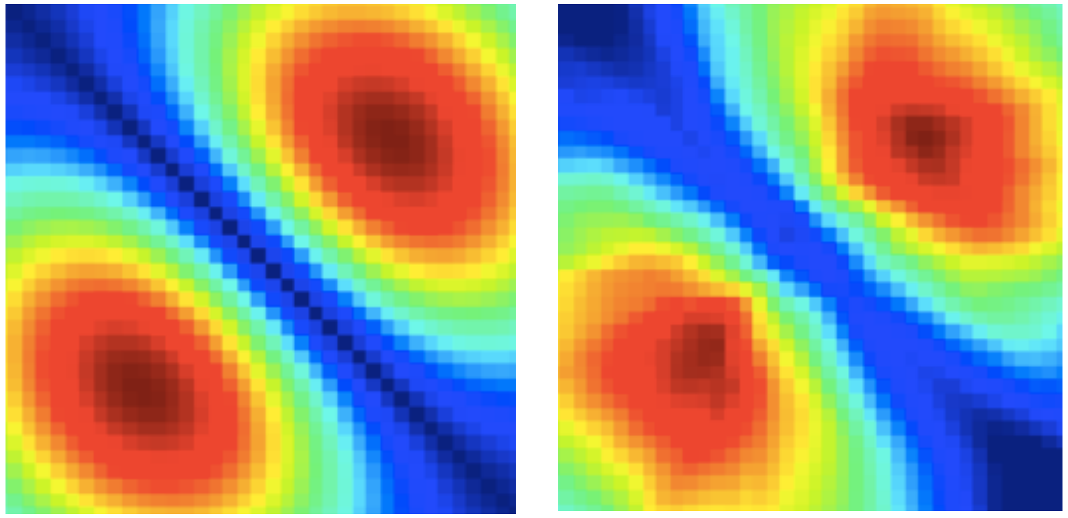
Now: The full algorithm, with forward propagation and backpropagation!

(will store neuron values and f'(z) values during forward propagation, to be used later during backpropagation)

```python
def net_f_df(z): # calculate f(z) and f'(z)
    val=1/(1+exp(-z))
    return(val,exp(-z)*(val**2)) # return both f and f'

def forward_step(y,w,b): # calculate values in next layer
    z=dot(y,w)+b # w=weights, b=bias vector for next layer
    return(net_f_df(z)) # apply nonlinearity


def apply_net(y_in): # one forward pass through the network
    global Weights, Biases, NumLayers
    global y_layer, df_layer # store y-values and df/dz
    y=y_in # start with input values
    y_layer[0]=y
    for j in range(NumLayers): # loop through all layers
        # j=0 corresponds to the first layer above input
        y,df=forward_step(y,Weights[j],Biases[j])
        df_layer[j]=df # store f'(z)
        y_layer[j+1]=y # store f(z)
    return(y)
```

```
def backward_step(delta,w,df):
    # delta at layer N, of batchsize x layersize(N))
    # w [layersize(N-1) x layersize(N) matrix]
    # df = df/dz at layer N-1, of batchsize x layersize(N-1)
    return( dot(delta,transpose(w))*df )

def backprop(y_target): # one backward pass
    # the result will be the 'dw_layer' matrices with
    # the derivatives of the cost function with respect to
    # the corresponding weight (similar for biases)
    global y_layer, df_layer, Weights, Biases, NumLayers
    global dw_layer, db_layer # dCost/dw and dCost/db
    #(w,b=weights,biases)
    global batchsize

    delta=(y_layer[-1]-y_target)*df_layer[-1]
    dw_layer[-1]=dot(transpose(y_layer[-2]),delta)/batchsize
    db_layer[-1]=delta.sum(0)/batchsize
    for j in range(NumLayers-1):
        delta=backward_step(delta,Weights[-1-j],...
         ...df_layer[-2-j])
        dw_layer[-2-j]=dot(transpose(y_layer[-3-j]),delta)...
         .../batchsize
        db_layer[-2-j]=delta.sum(0)/batchsize
```

Homework



For the example case (learning a 2D function; see code on the website)

Try out the effects of:
- Value of the stepsize eta
- Layout of the network (number of neurons and number of layers)
- Initialization of the weights

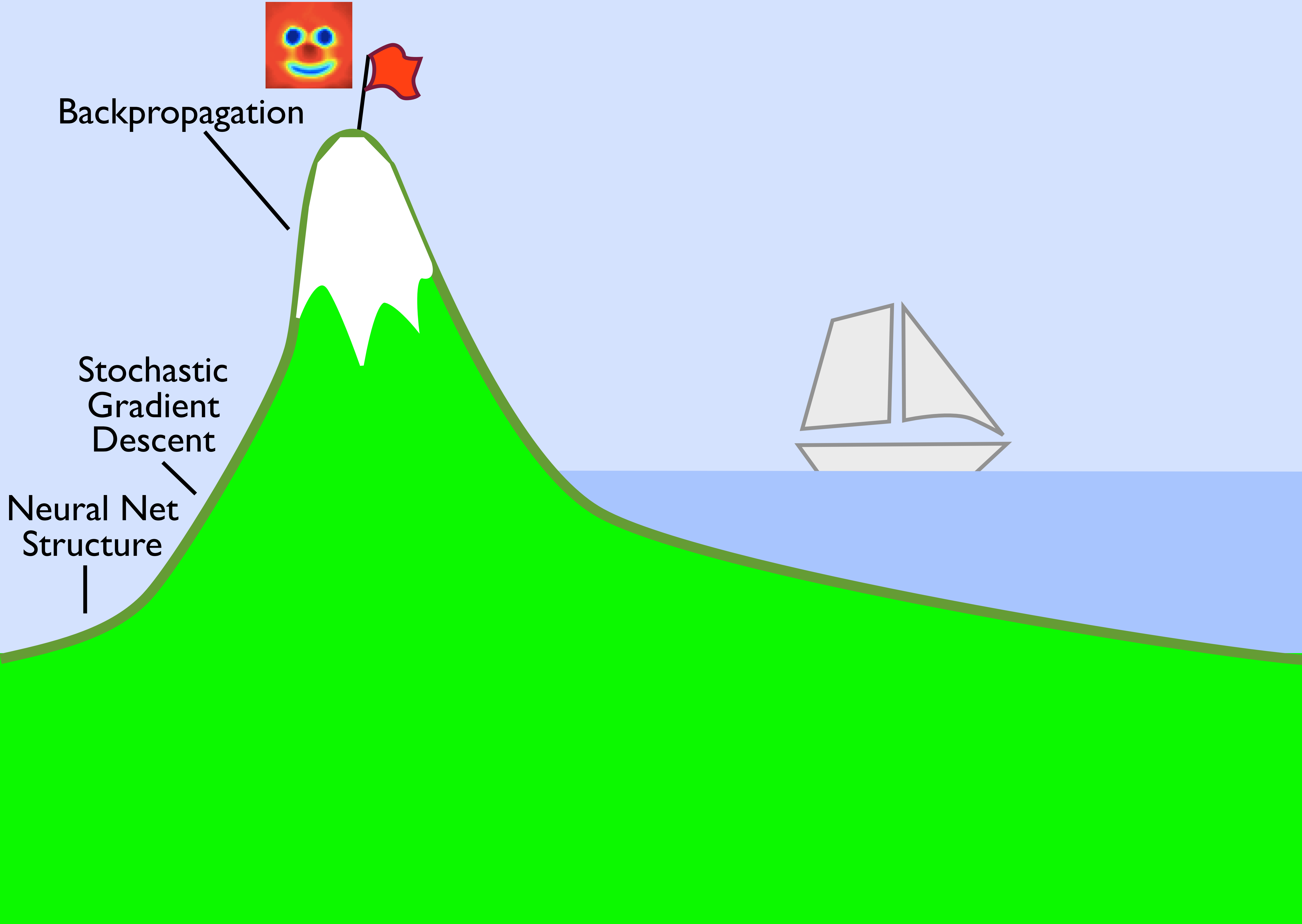
How do these things affect the speed of learning and the final quality (final value of the cost function)?

Try them out also for other test functions (other than in the example)

Homework

Change the output layer f(z) to a LINEAR function, i.e. $f(z)=z$! Implement the required changes to the backpropagation code.

Apply this to the example case (learning a 2D function; see code on the website).



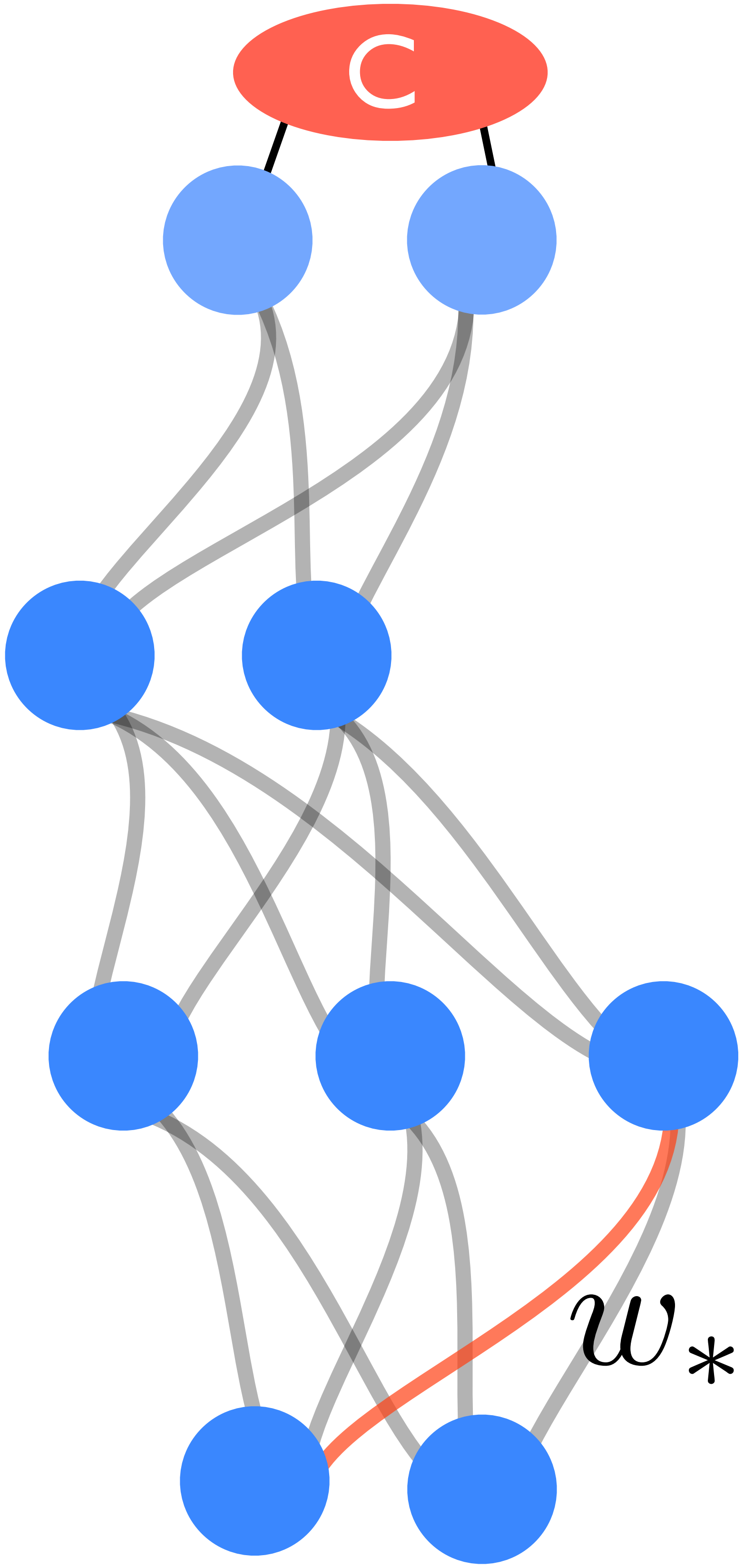
Backpropagation

Stochastic
Gradient
Descent

Neural Net
Structure

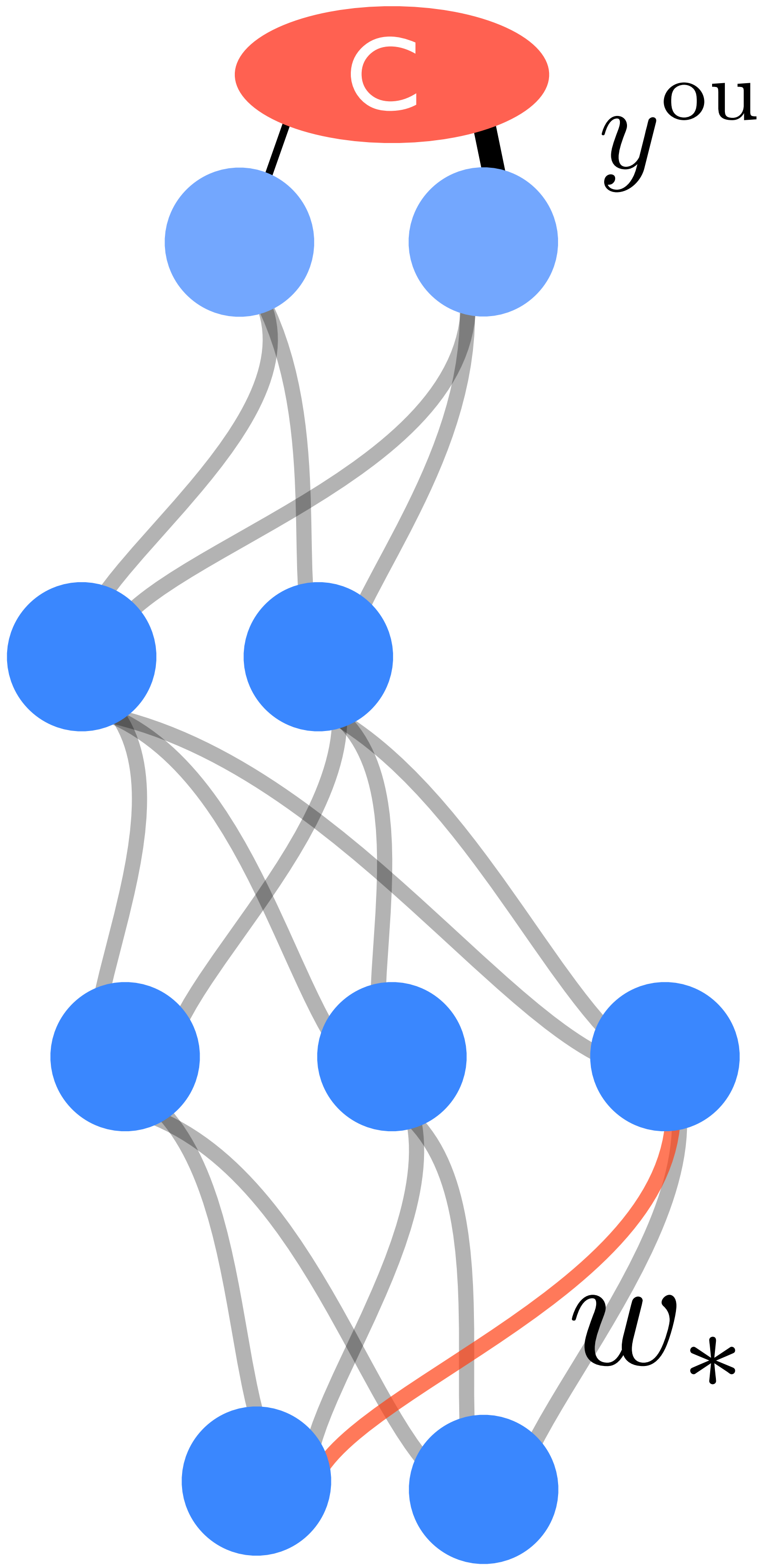
Backpropagation: the principle

$$\frac{\partial C}{\partial w_*} = ?$$



Backpropagation: the principle

$$\frac{\partial C}{\partial w_*} = ?$$

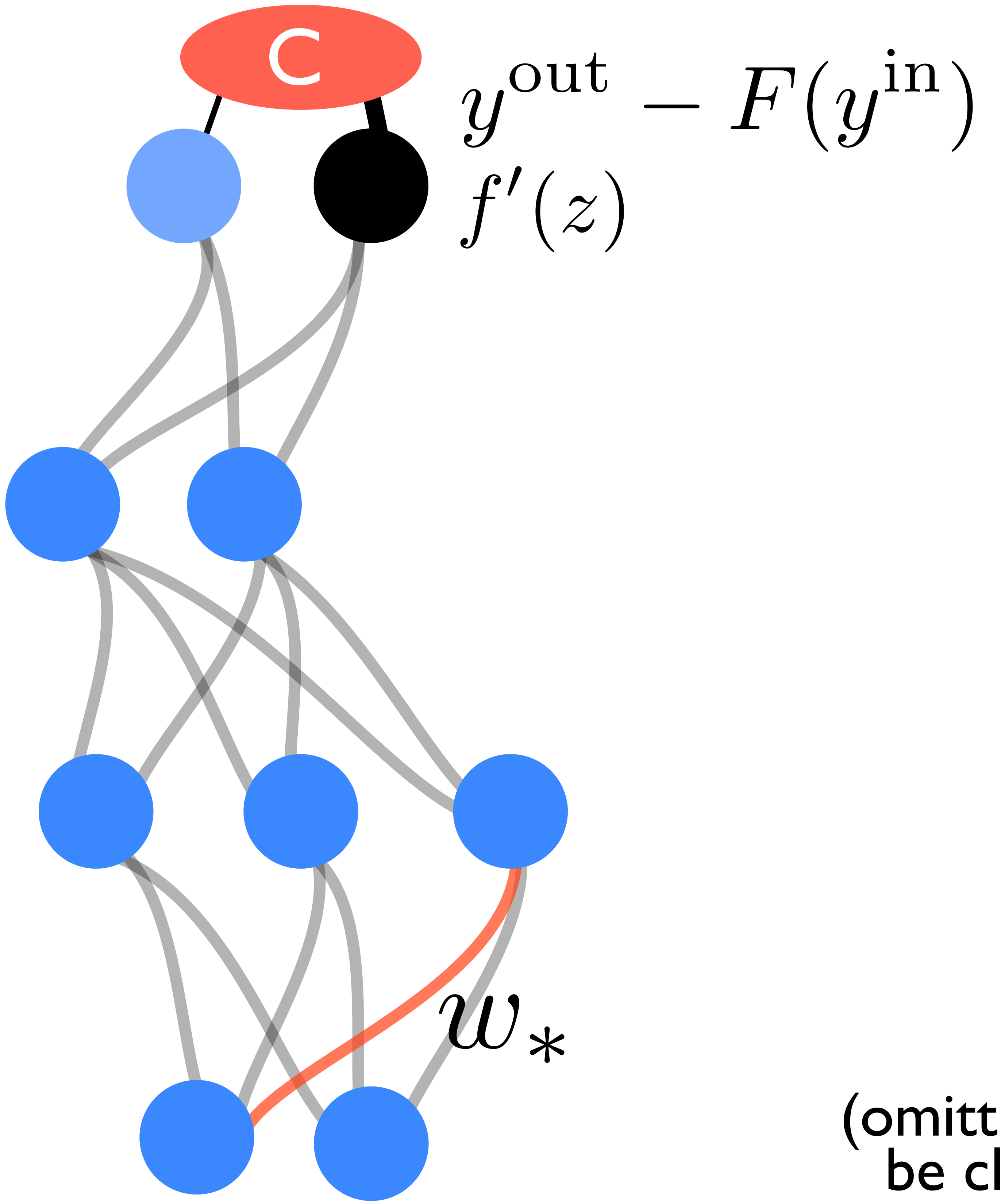


$$y^{\text{out}} - F(y^{\text{in}})$$

(omitting indices, should
be clear from figure)

Backpropagation: the principle

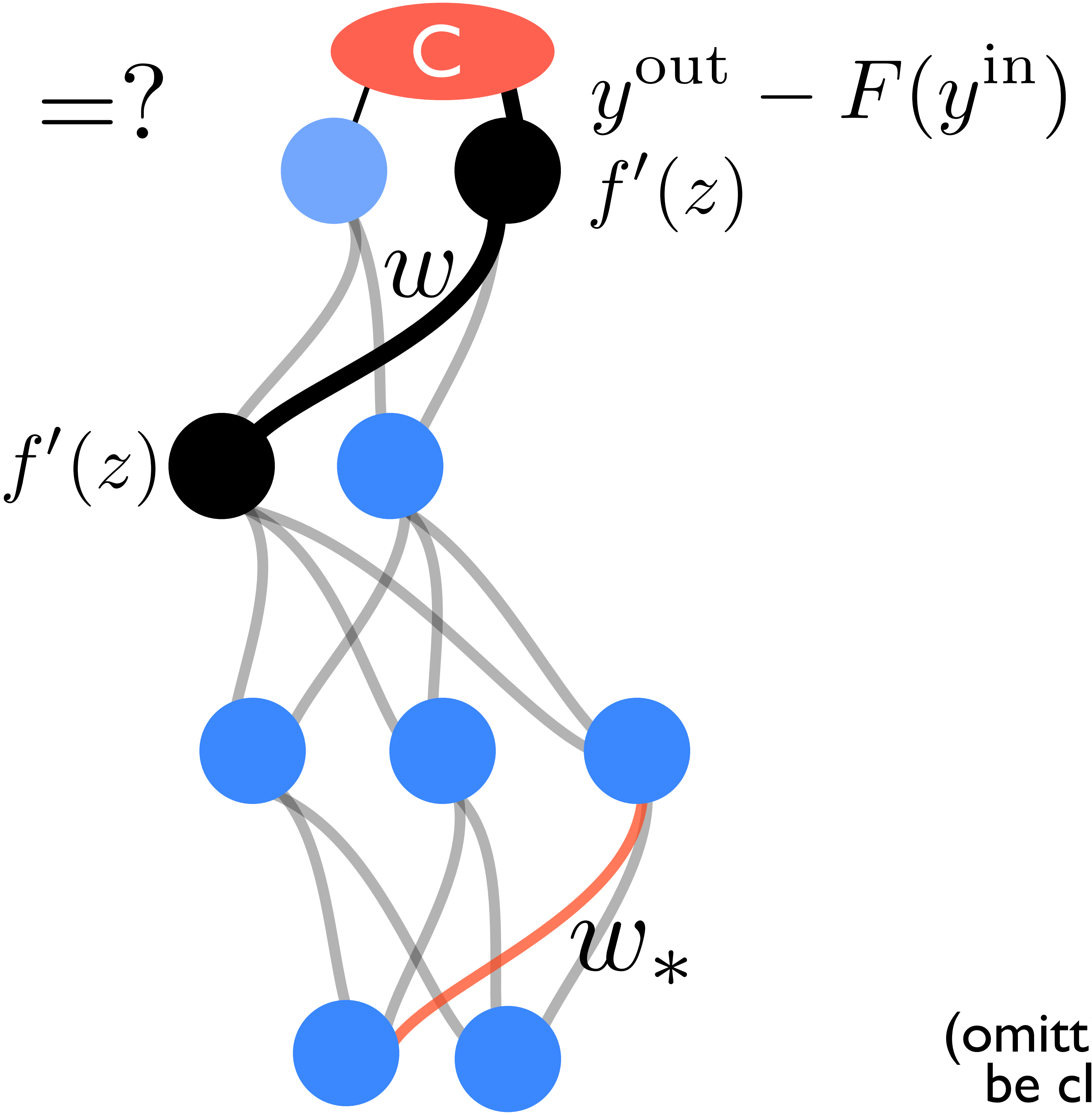
$$\frac{\partial C}{\partial w_*} = ?$$



(omitting indices, should be clear from figure)

Backpropagation: the principle

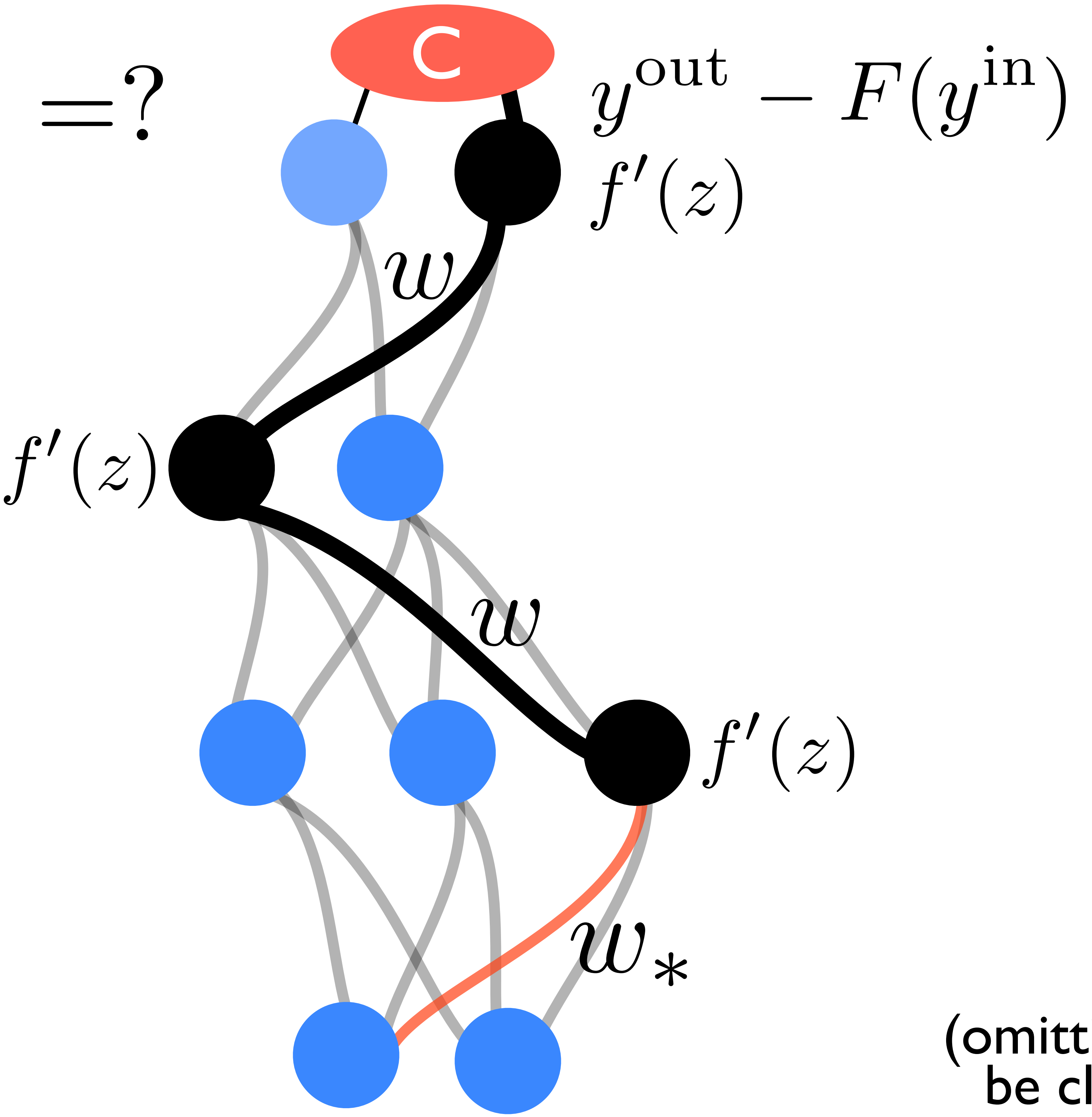
$$\frac{\partial C}{\partial w_*} = ?$$



(omitting indices, should be clear from figure)

Backpropagation: the principle

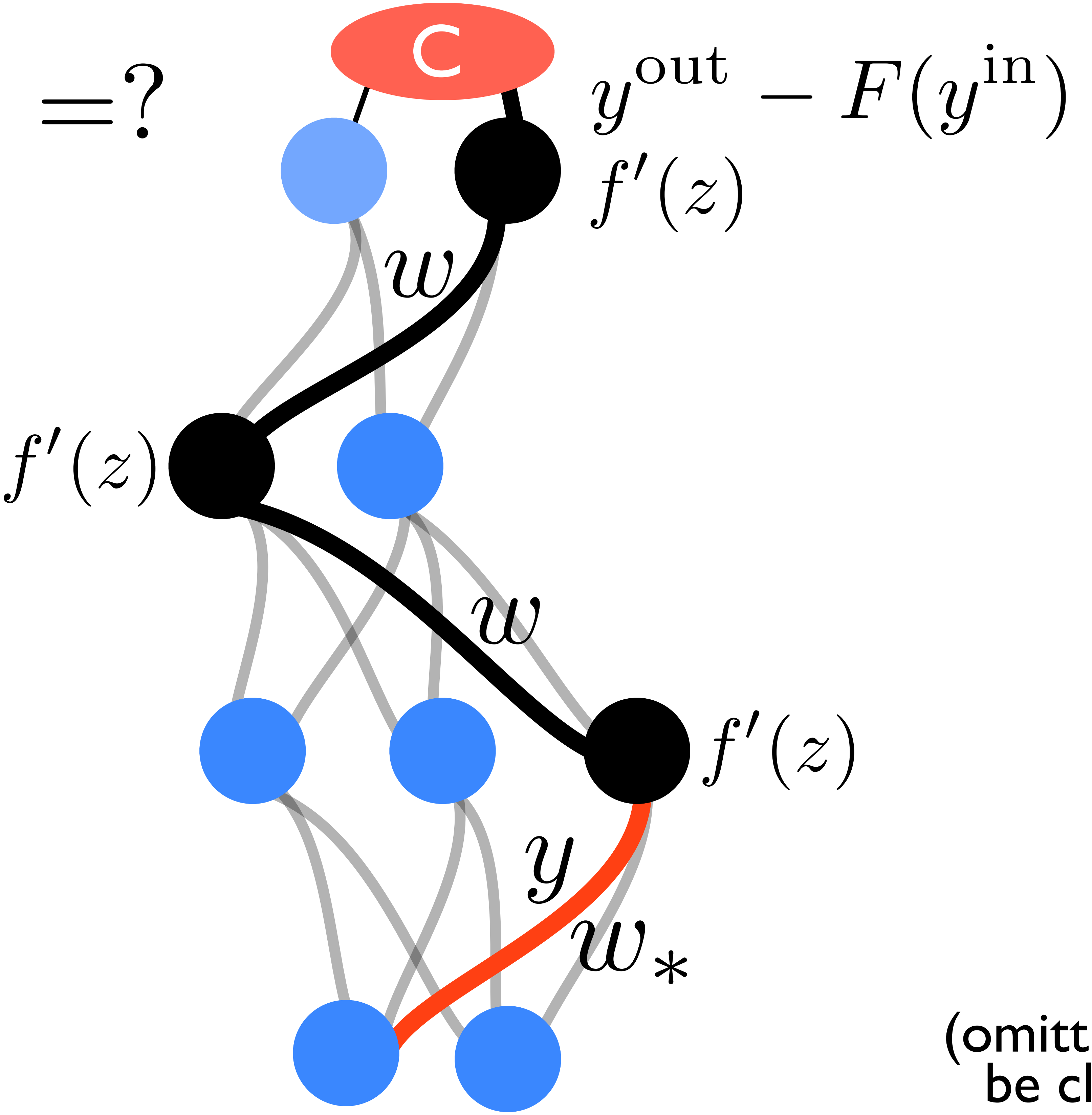
$$\frac{\partial C}{\partial w_*} = ?$$



(omitting indices, should be clear from figure)

Backpropagation: the principle

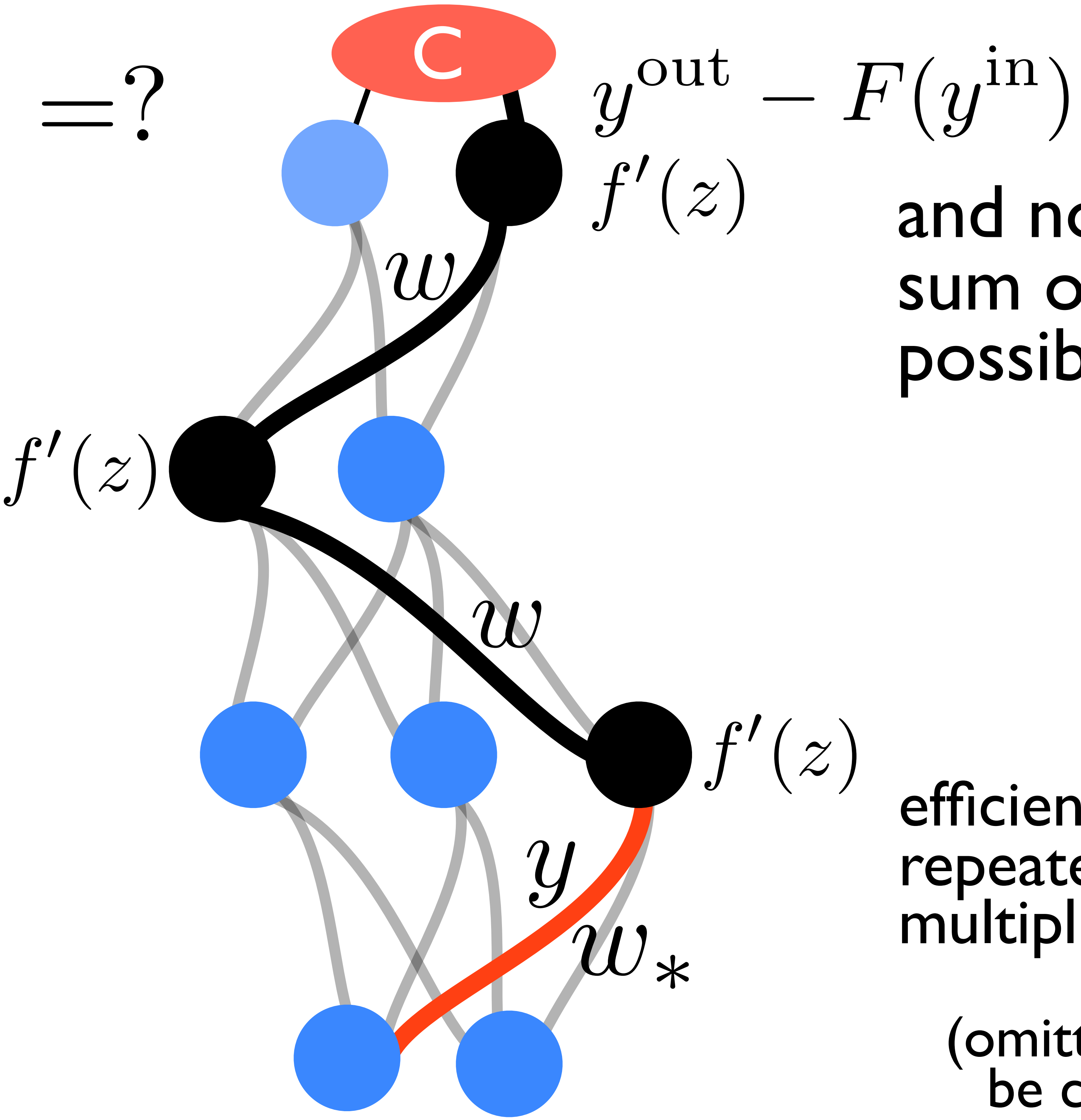
$$\frac{\partial C}{\partial w_*} = ?$$



(omitting indices, should be clear from figure)

Backpropagation: the principle

$$\frac{\partial C}{\partial w_*} = ?$$

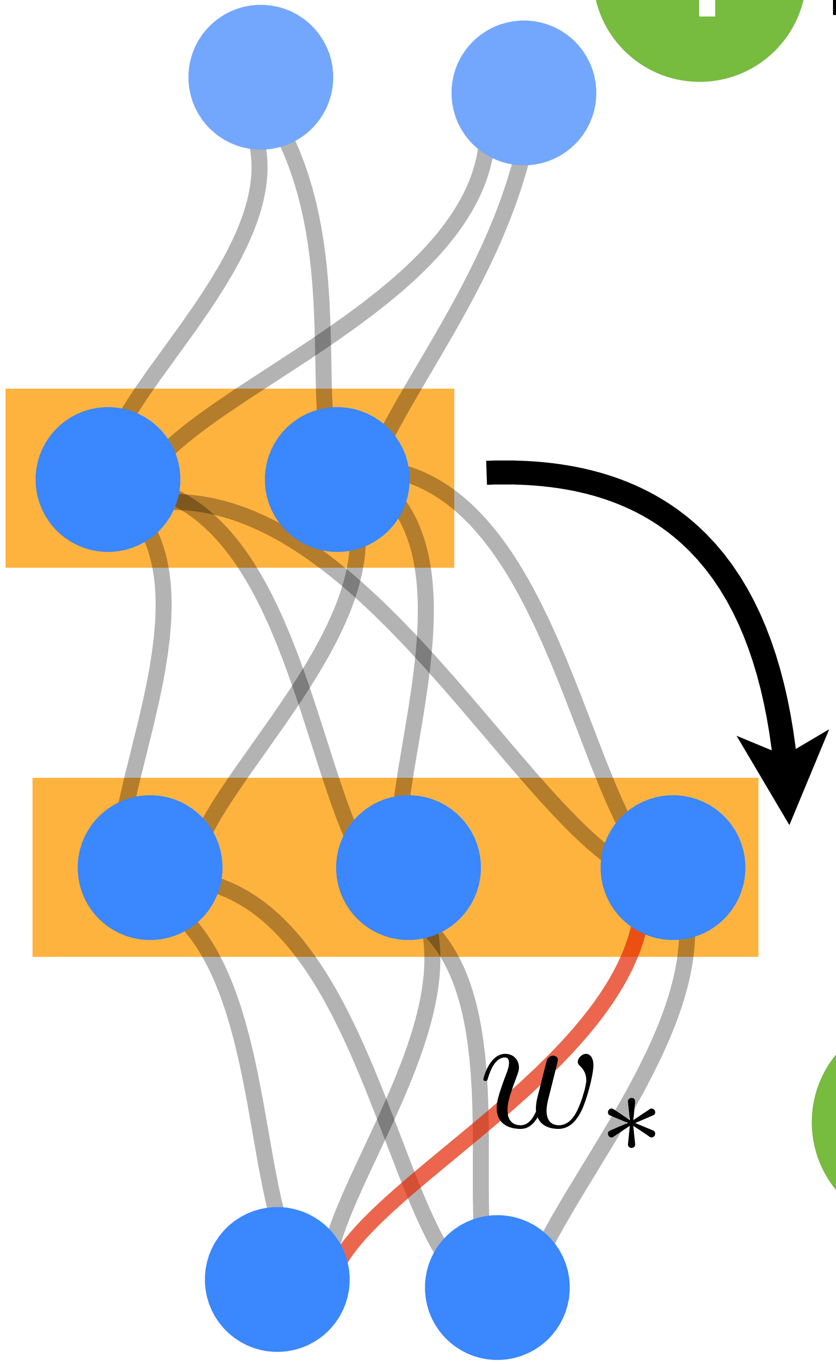


and now:
sum over ALL
possible paths!

efficient implementation:
repeated matrix/vector
multiplication

(omitting indices, should
be clear from figure)

Backpropagation Summary



① Initialize vector from output layer:

$$\Delta_j = (y_j^n - F_j(y^{\text{in}}))f'(z_j^n)$$

② For each layer: store outcomes (cost derivatives) for all weights and biases w_* in that layer

$$\frac{\partial C(w, y^{\text{in}})}{\partial w_*} = \Delta_j \frac{\partial z_j^{(n)}}{\partial w_*}$$

(see above)

(j is the index where this particular weight appears)

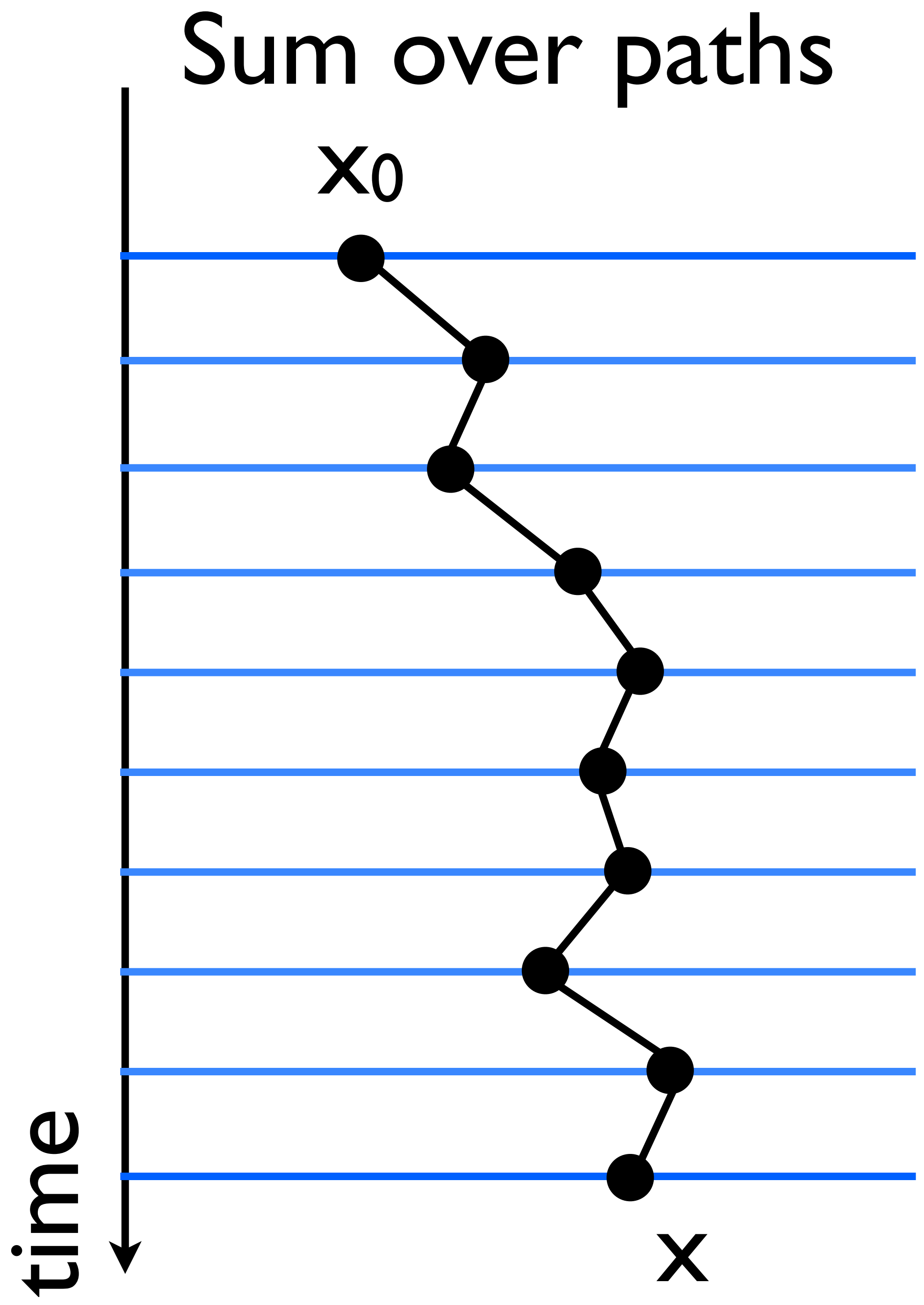
③ Multiply vector by matrix

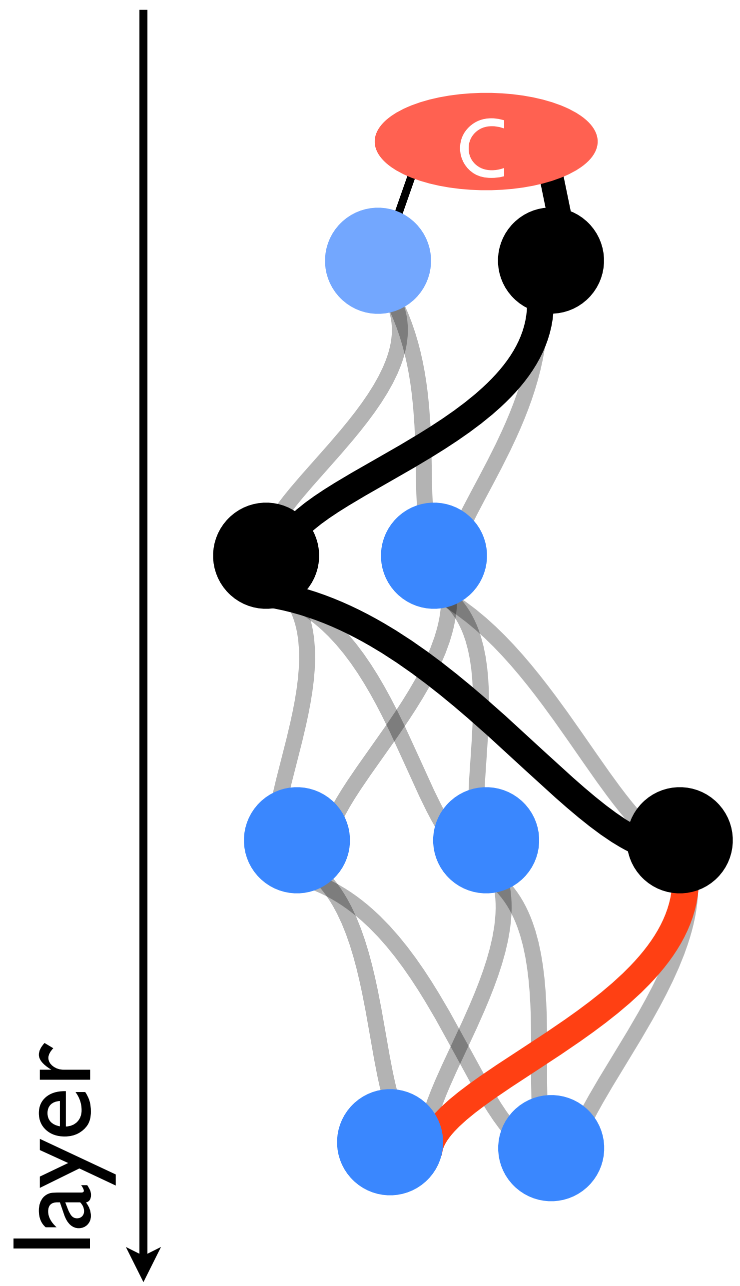
$$\Delta_k^{\text{new}} = \sum_j \Delta_j M_{jk}^{n,n-1}$$

(see above for M)

(& return to step 2)

similar to Feynman sum over paths (path integral)





...or matrix product:

$$\Psi(t) = \hat{U}(t)\Psi(0) = \hat{U}_1\hat{U}_2\hat{U}_3\ldots\Psi(0)$$

Backpropagation: the code

```
def net_f_df(z): # calculate f(z) and f'(z)
    val=1/(1+exp(-z))
    return(val,exp(-z)*(val**2)) # return both f and f'
```

```
def forward_step(y,w,b): # calculate values in next layer
z=dot(y,w)+b # w=weights, b=bias vector for next layer
    return(net_f_df(z)) # apply nonlinearity
```

```
def apply_net(y_in): # one forward pass through the network
    global Weights, Biases, NumLayers
    global y_layer, df_layer # store y-values and df/dz
    y=y_in # start with input values
    y_layer[0]=y
    for j in range(NumLayers): # loop through all layers
        # j=0 corresponds to the first layer above input
        y,df=forward_step(y,Weights[j],Biases[j])
        df_layer[j]=df # store f'(z)
        y_layer[j+1]=y # store f(z)
    return(y)
```

only 30 lines of code!

```
def backward_step(delta,w,df):
    # delta at layer N, of batchsize x layersize(N))
    # w [layersize(N-1) x layersize(N) matrix]
    # df = df/dz at layer N-1, of batchsize x layersize(N-1)
    return( dot(delta,transpose(w))*df )
```

```
def backprop(y_target): # one backward pass
    global y_layer, df_layer, Weights, Biases, NumLayers
    global dw_layer, db_layer # dCost/dw and dCost/db
  #(w,b=weights,biases)
    global batchsize

    delta=(y_layer[-1]-y_target)*df_layer[-1]
    dw_layer[-1]=dot(transpose(y_layer[-2]),delta)/batchsize
    db_layer[-1]=delta.sum(0)/batchsize
    for j in range(NumLayers-1):
        delta=backward_step(delta,Weights[-1-j],df_layer[-2-j])
        dw_layer[-2-j]=dot(transpose(y_layer[-3-j]),delta)/batchsize
        db_layer[-2-j]=delta.sum(0)/batchsize
```

Neural networks: the ingredients

General purpose algorithm: feedforward & backpropagation (implement once, use often)

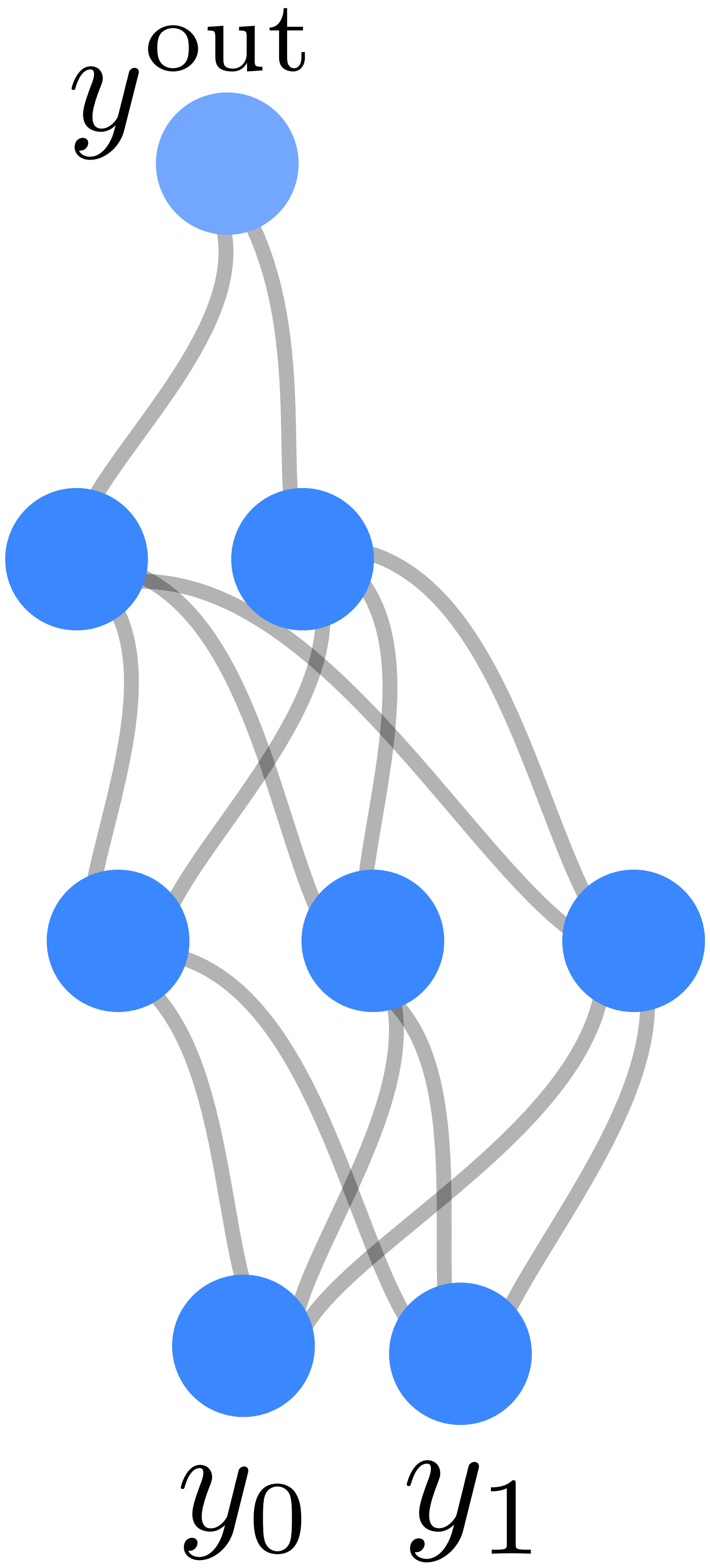
Problem-specific:
Choose network layout (number of layers, number of neurons in each layer, type of nonlinear functions, maybe specialized structures of the weights) **“Hyperparameters”**

Generate training (& validation & test) samples: load from big databases (that have to be compiled from the internet or by hand!) or produce by software

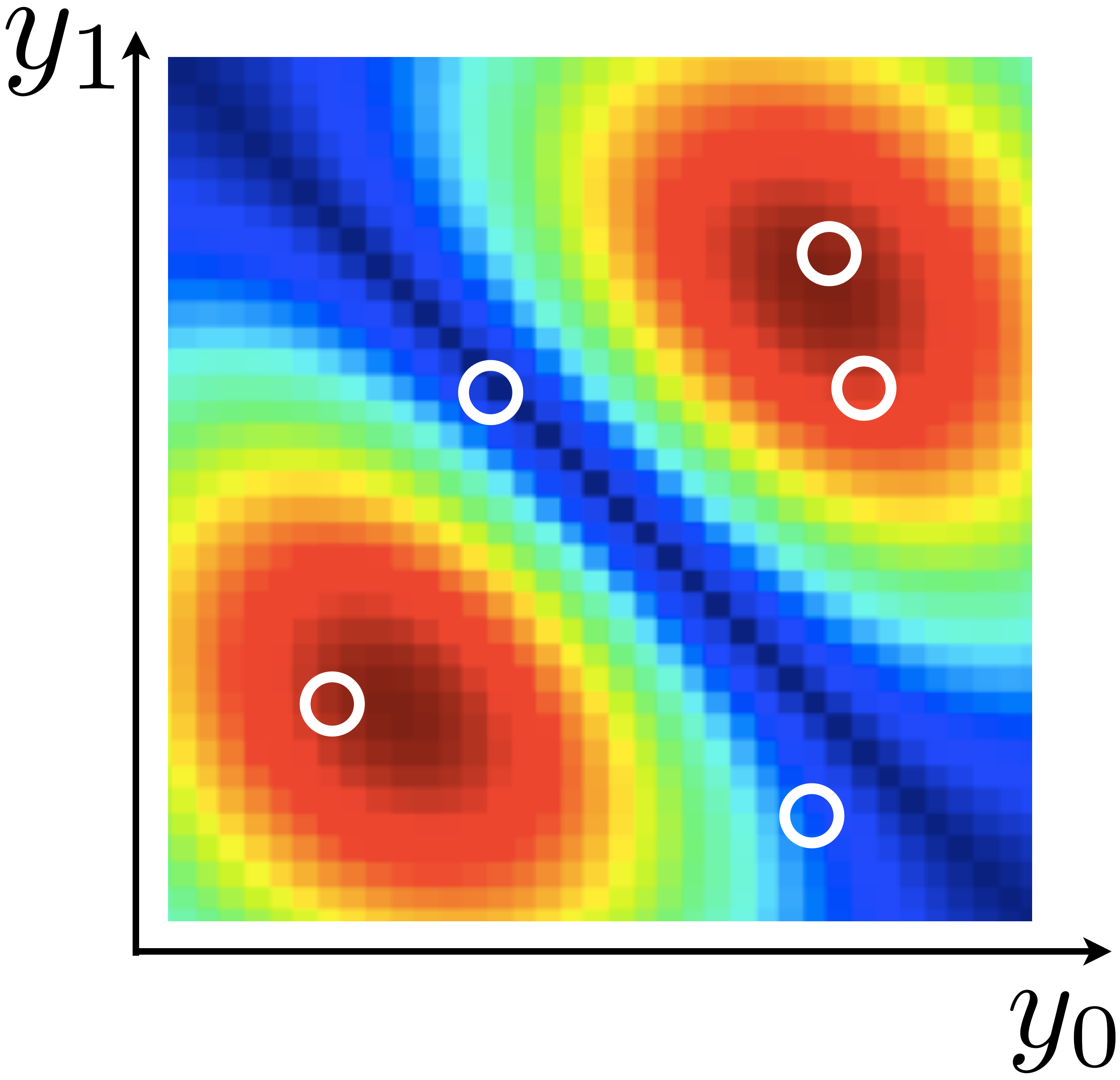
Monitor/optimize training progress (possibly choose learning rate and batch size or other parameters, maybe try out many combinations) **“Hyperparameters”**

Example: Learning a 2D function

see notebook (on website): **MultiLayerBackProp**



y^{out} (values as color)



Evaluate at sample points

Example: Learning a 2D function

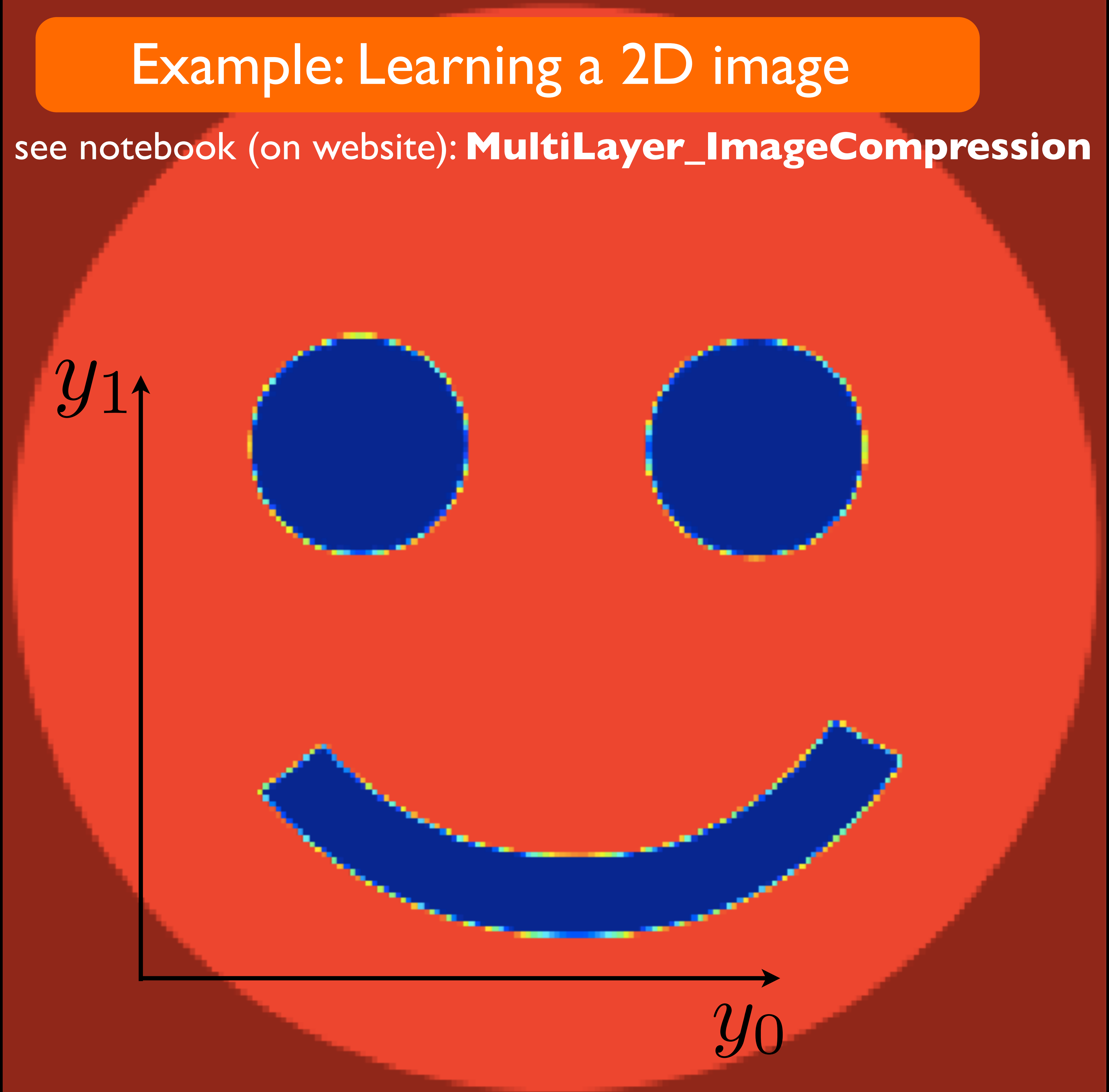
see notebook (on website): **MultiLayerBackProp**

```python
# pick batchsize random positions in the 2D square
def make_batch():
    global batchsize

    inputs=random.uniform(low=-0.5,high=+0.5,size=[batchsize,2])
    targets=zeros([batchsize,1]) # must have right dimensions
    targets[:,0]=myFunc(inputs[:,0],inputs[:,1])
    return(inputs,targets)
```

```python
eta=.1
batchsize=1000
batches=2000
costs=zeros(batches)

for k in range(batches):
    y_in,y_target=make_batch()
    costs[k]=train_net(y_in,y_target,eta)
```

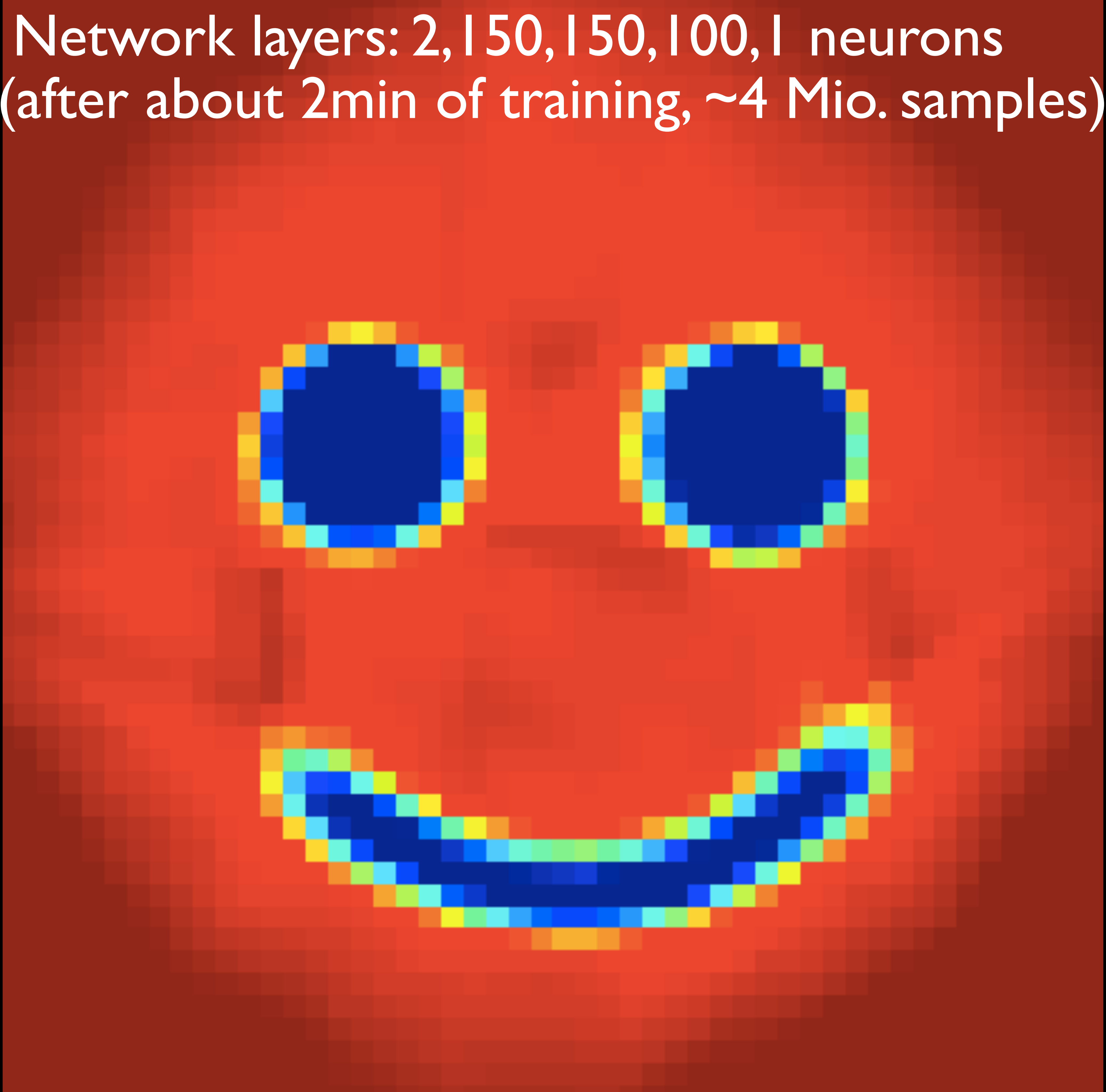
Example: Learning a 2D image

see notebook (on website): **MultiLayer_ImageCompression**



Network layers: 2,150,150,100,1 neurons

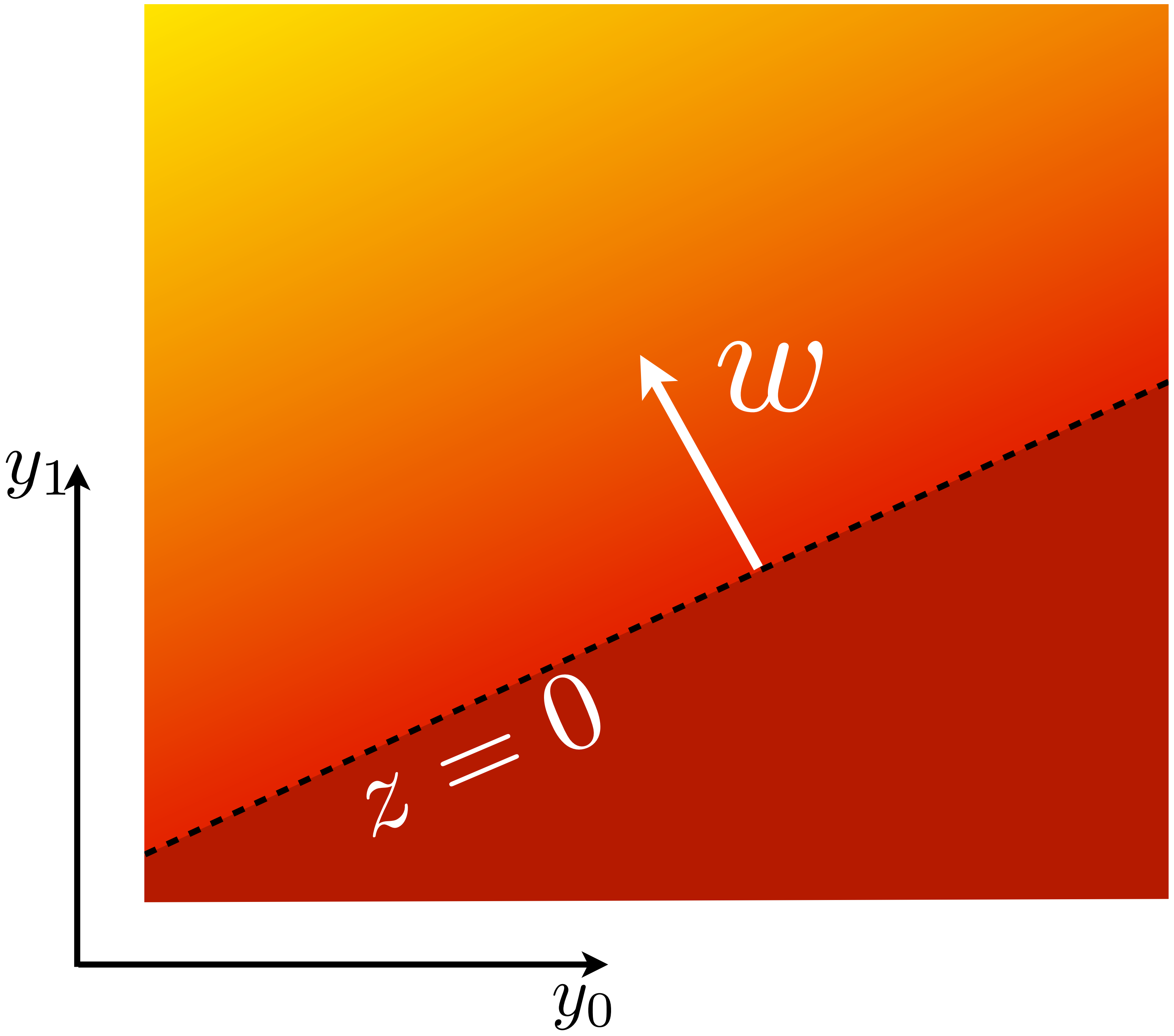
(after about 2min of training, ~4 Mio. samples)

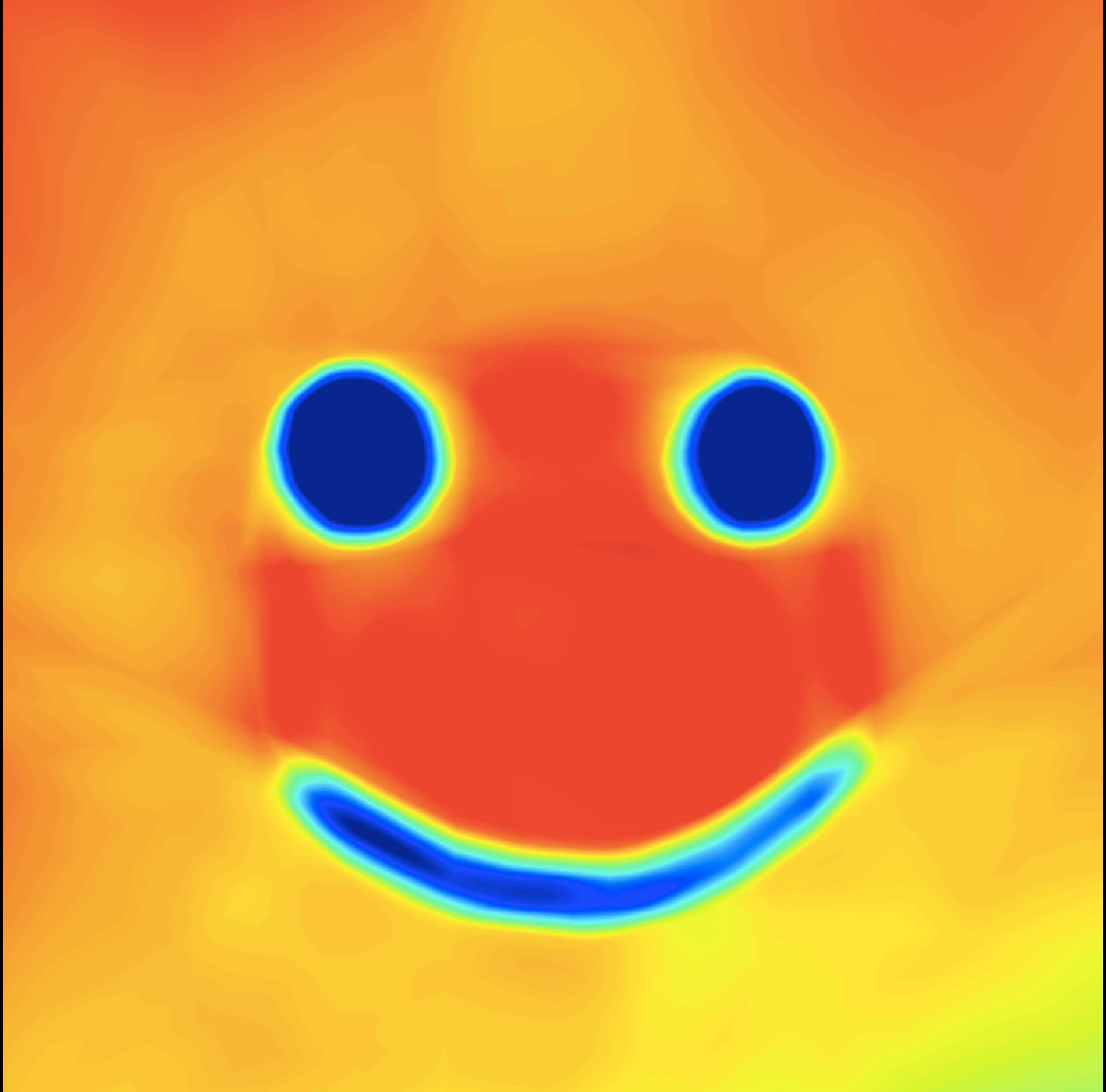


Reminder: ReLU (rectified linear unit)

$$f(z) = \begin{matrix} z \text{ for } z > 0 \\ 0 \text{ for } z \leq 0 \end{matrix}$$

$$z = wy + b$$





à la Franz Marc?

Try to understand how the network operates!

Switching on only a single neuron of the last hidden layer

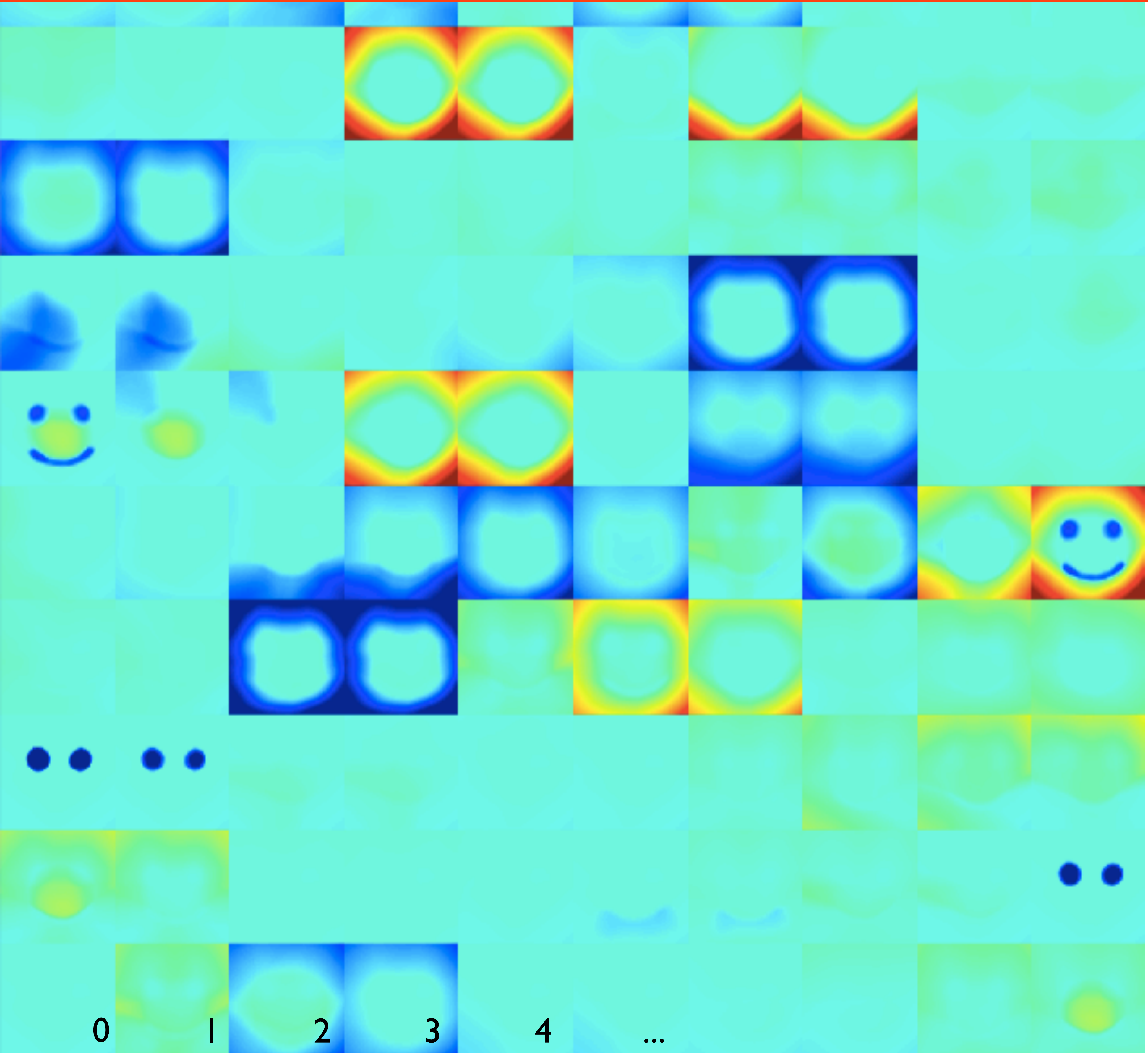
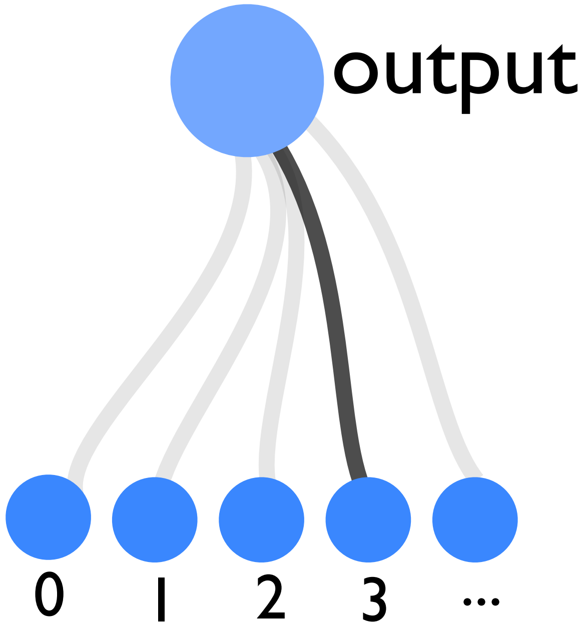
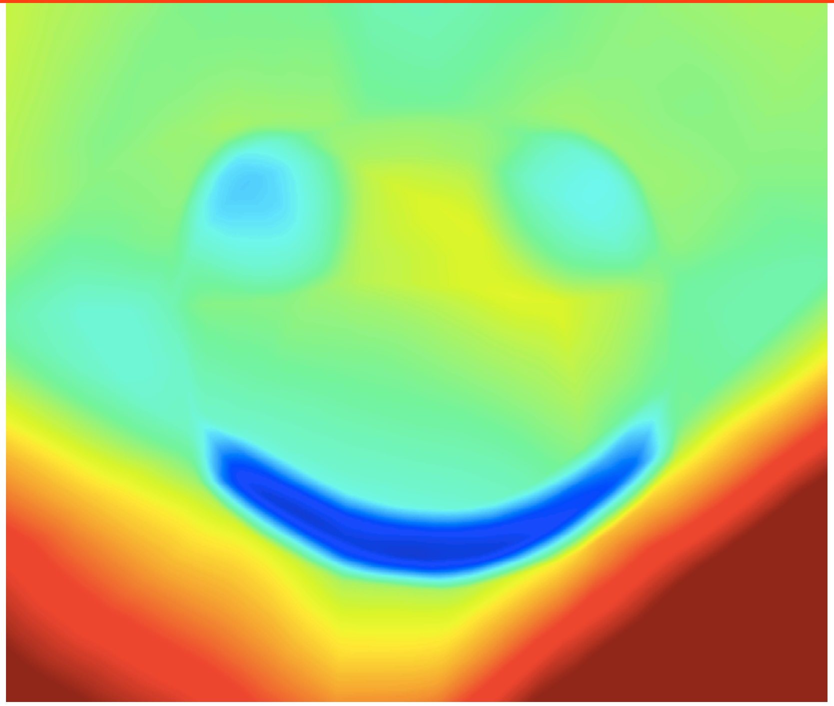


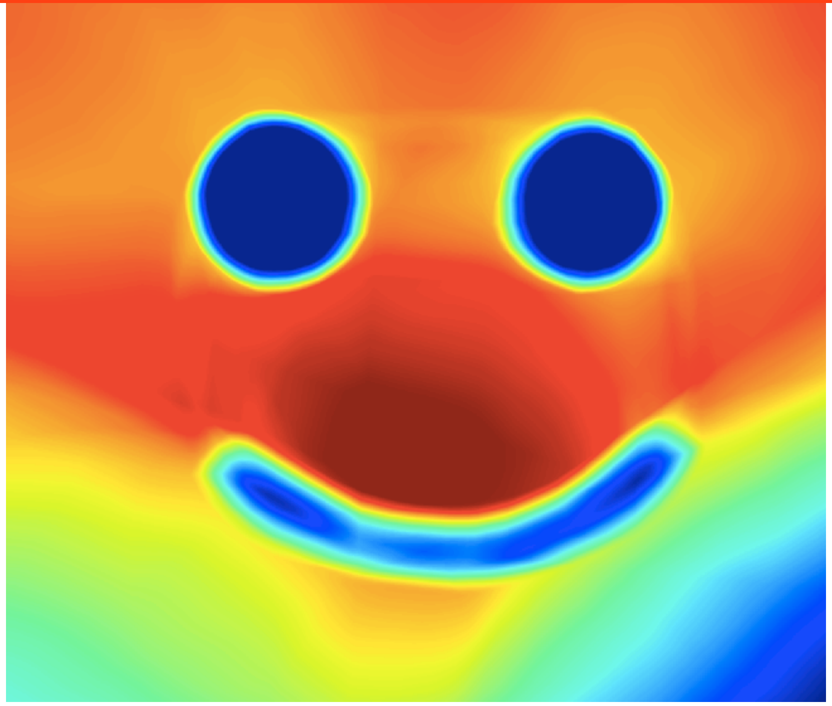
Image shows results of switching on individually each of 100 neurons



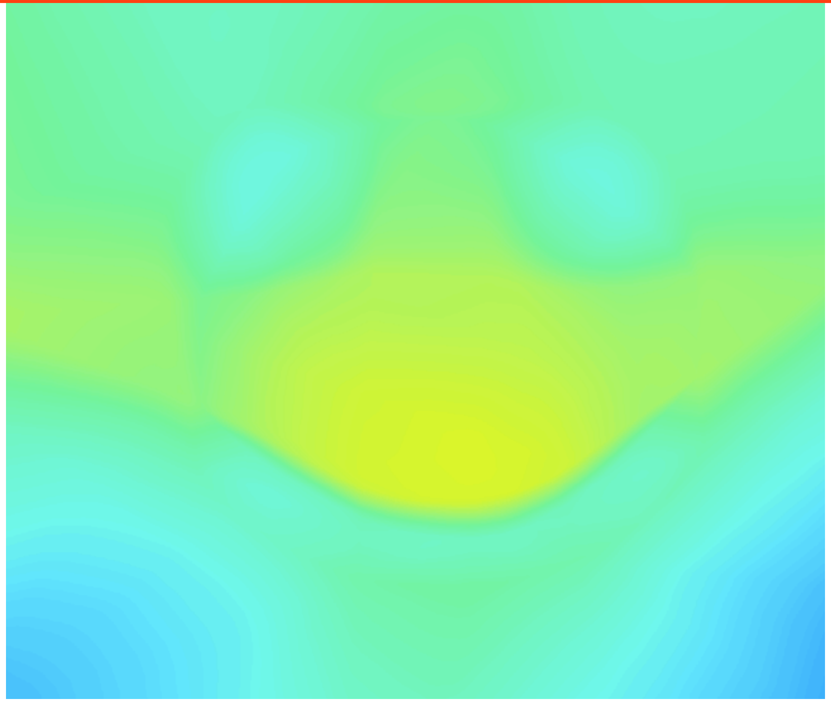
Weights from last hidden layer to output



deleted first 50 weights



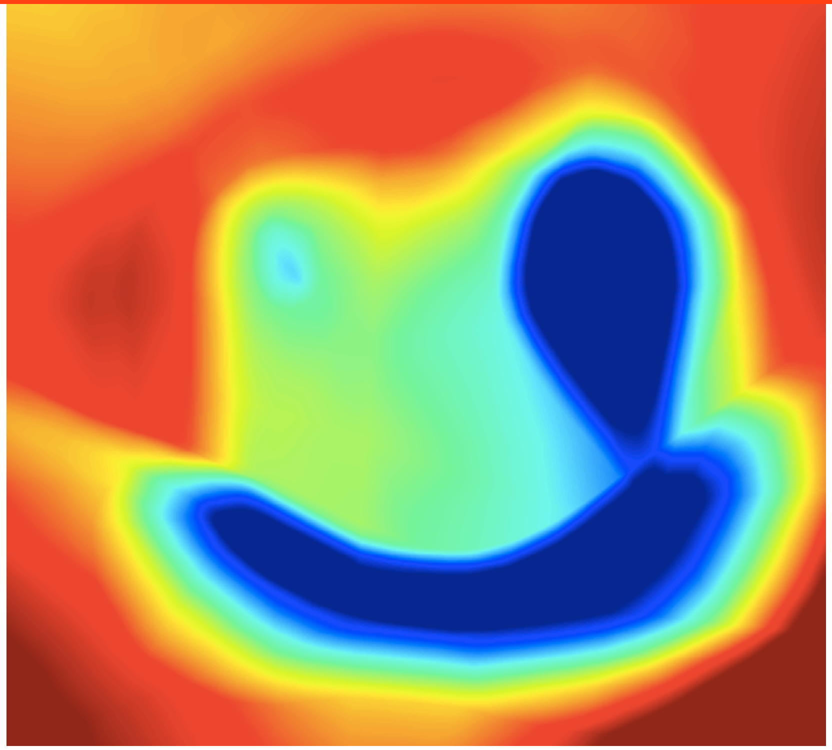
deleted last 50 weights



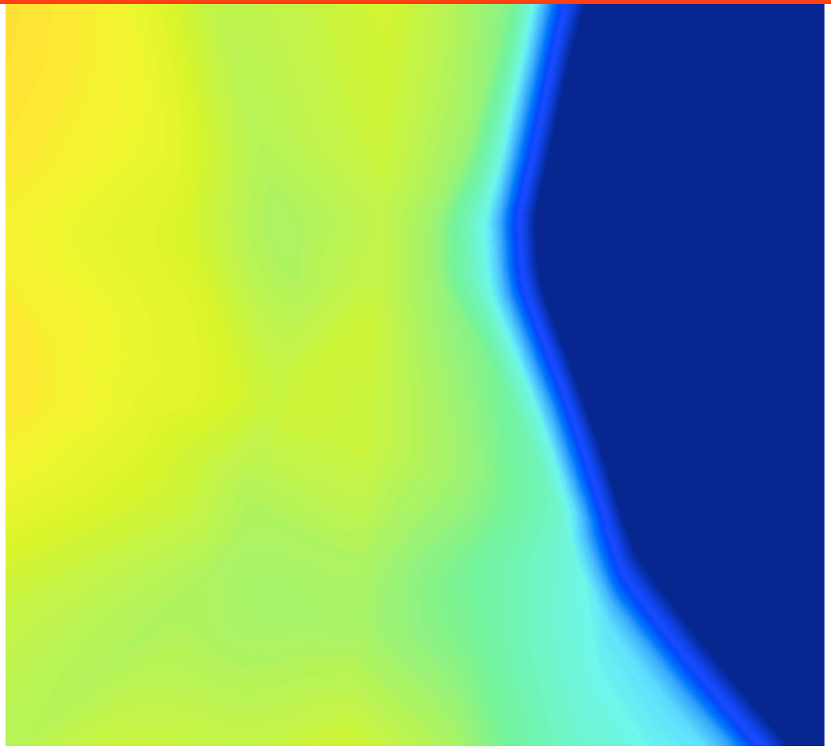
kept only 10 out of 100

Weights from 2nd hidden layer to last hidden layer

deleted first 75

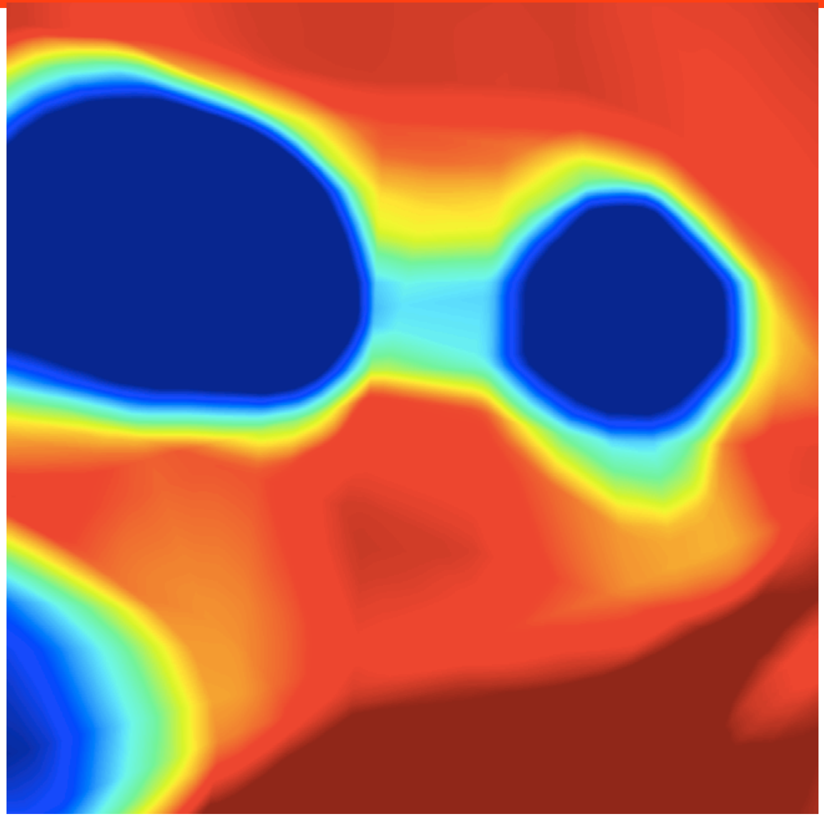


deleted last 75

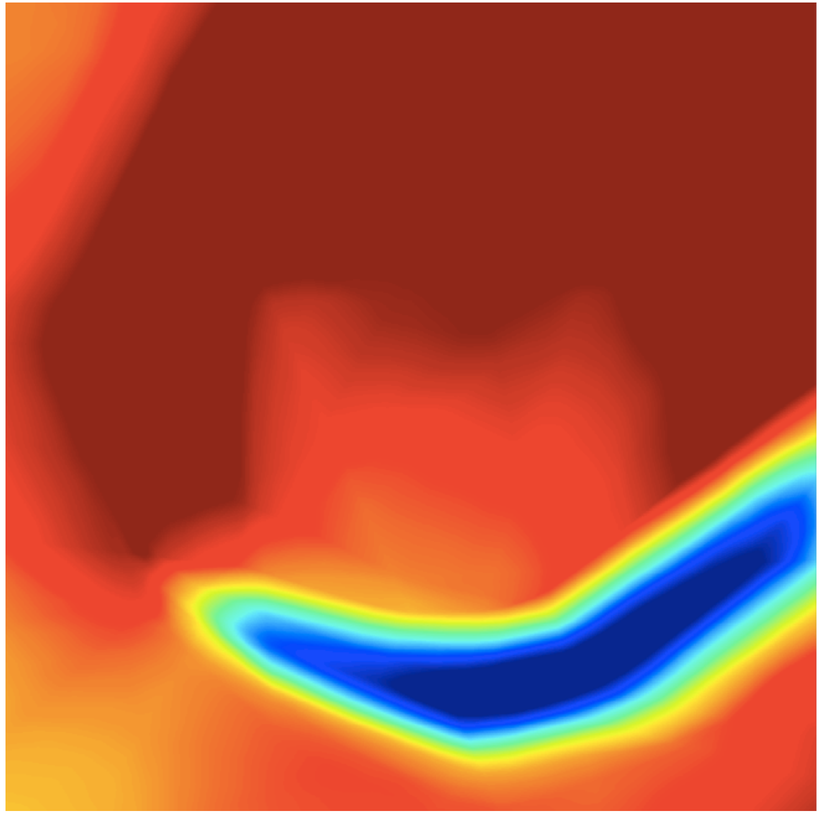


kept only 10 out of 150

Weights from 1st hidden layer to 2nd hidden layer



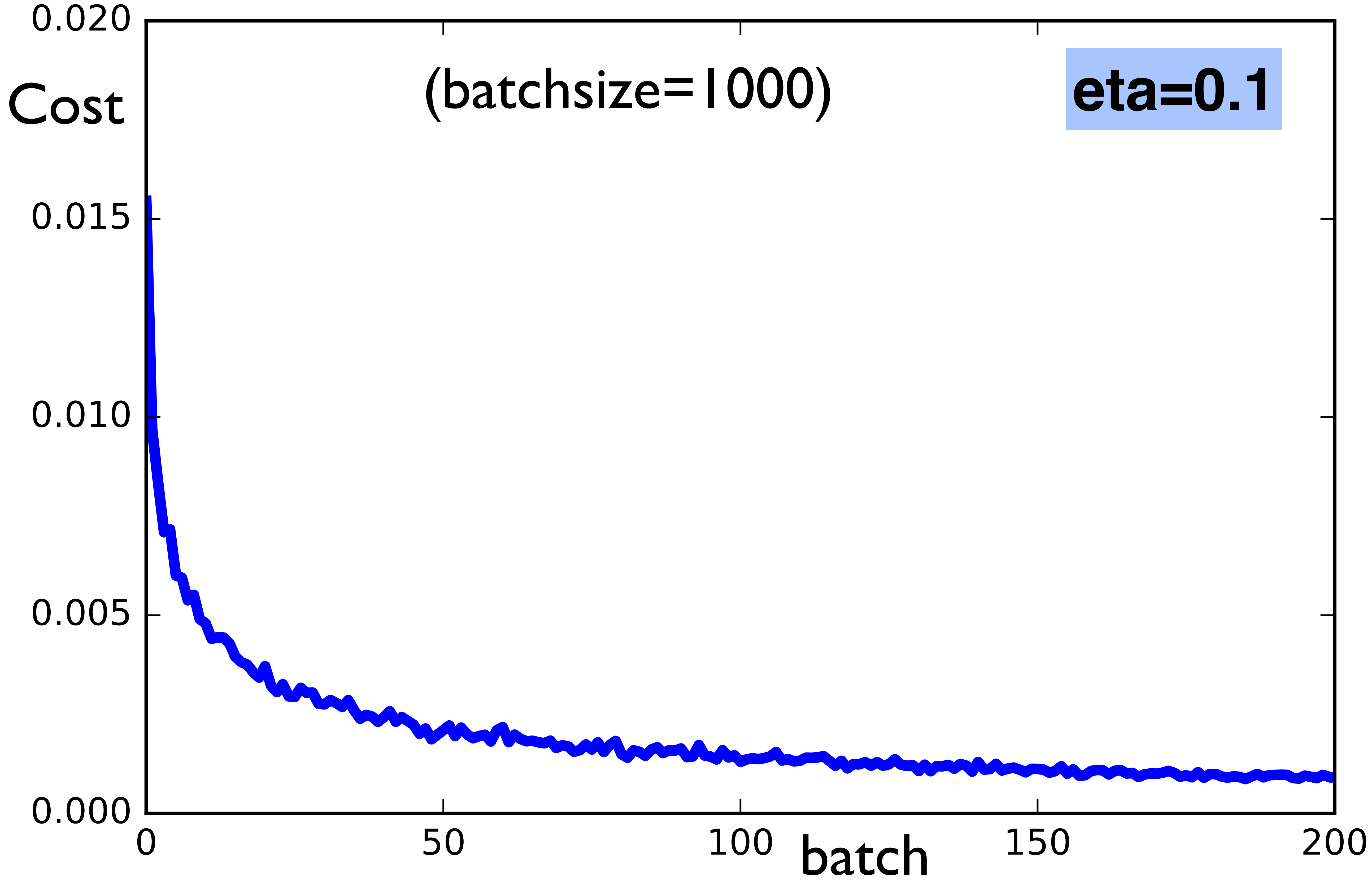
deleted first 75



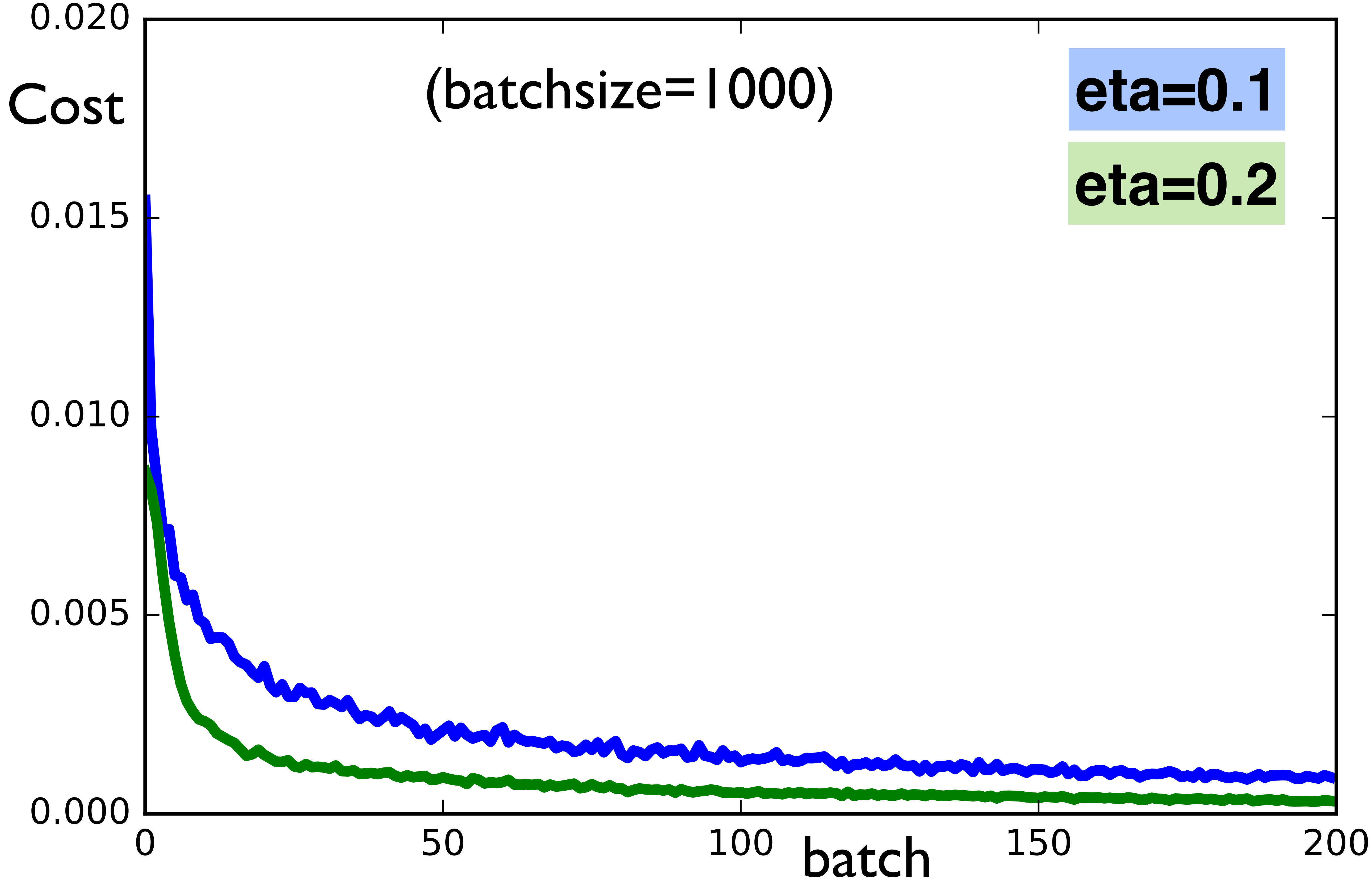
deleted last 75

kept only 10 out of 150

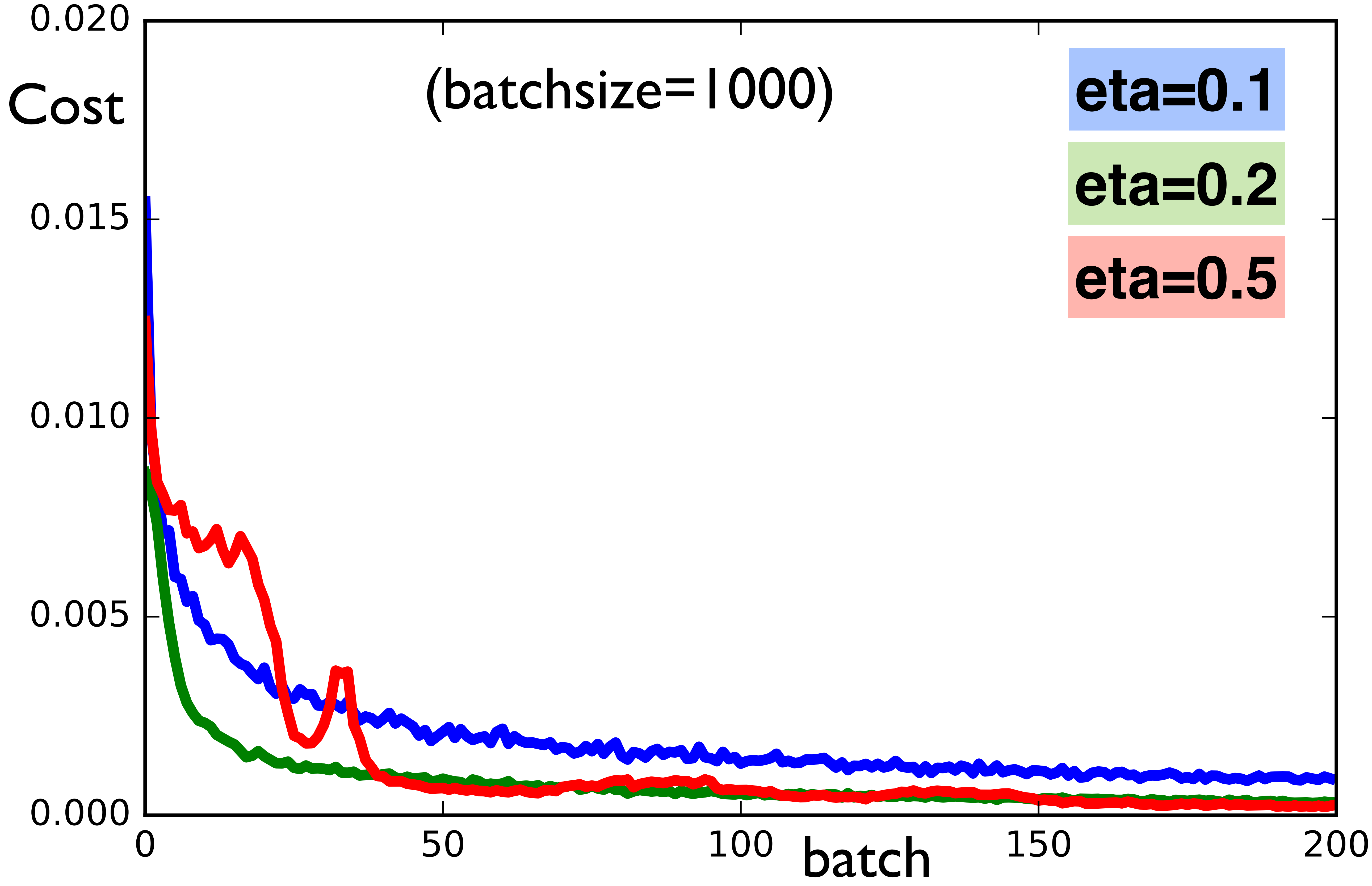
Influence of learning rate (stepsize)



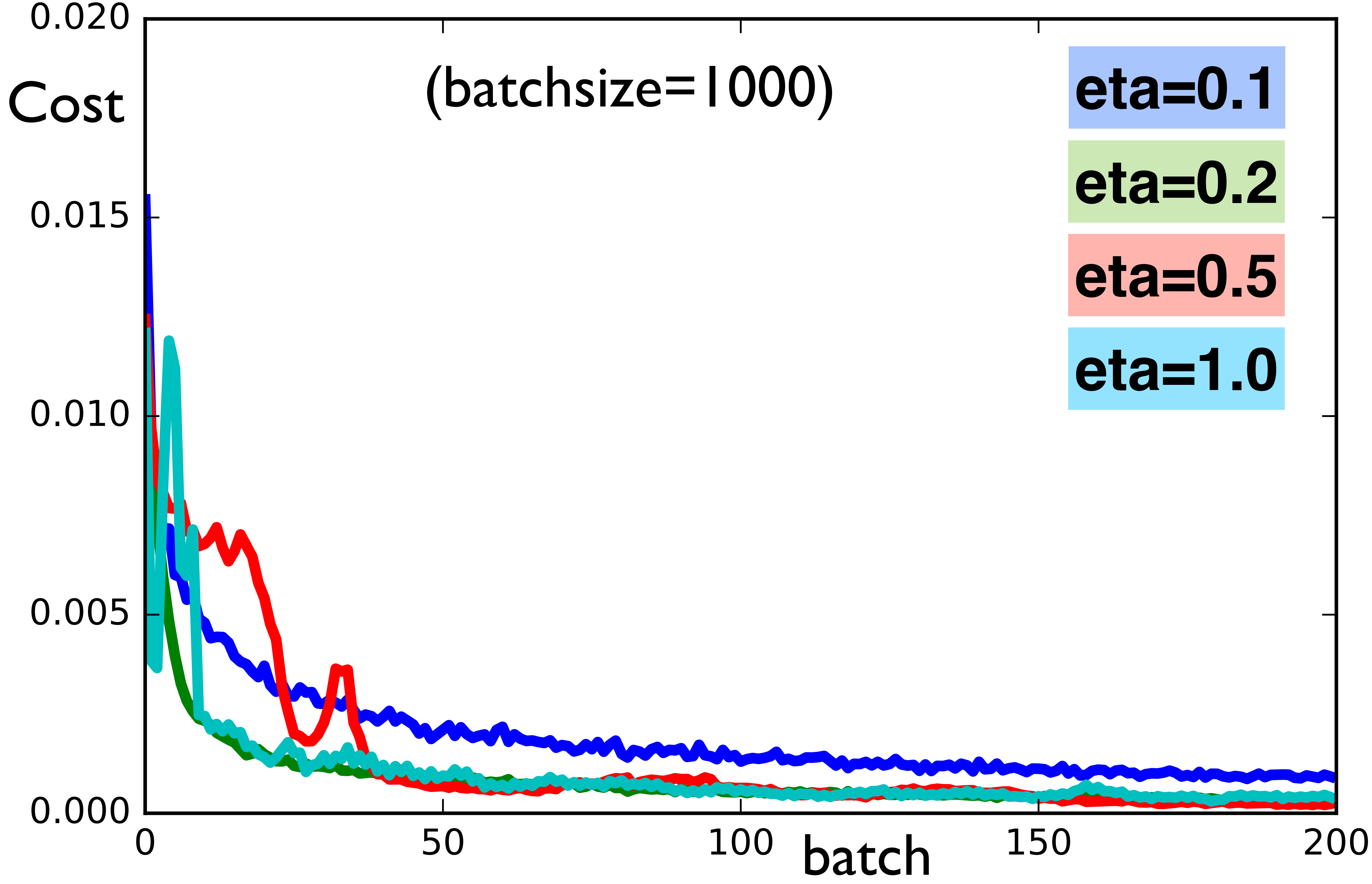
Influence of learning rate (stepsize)



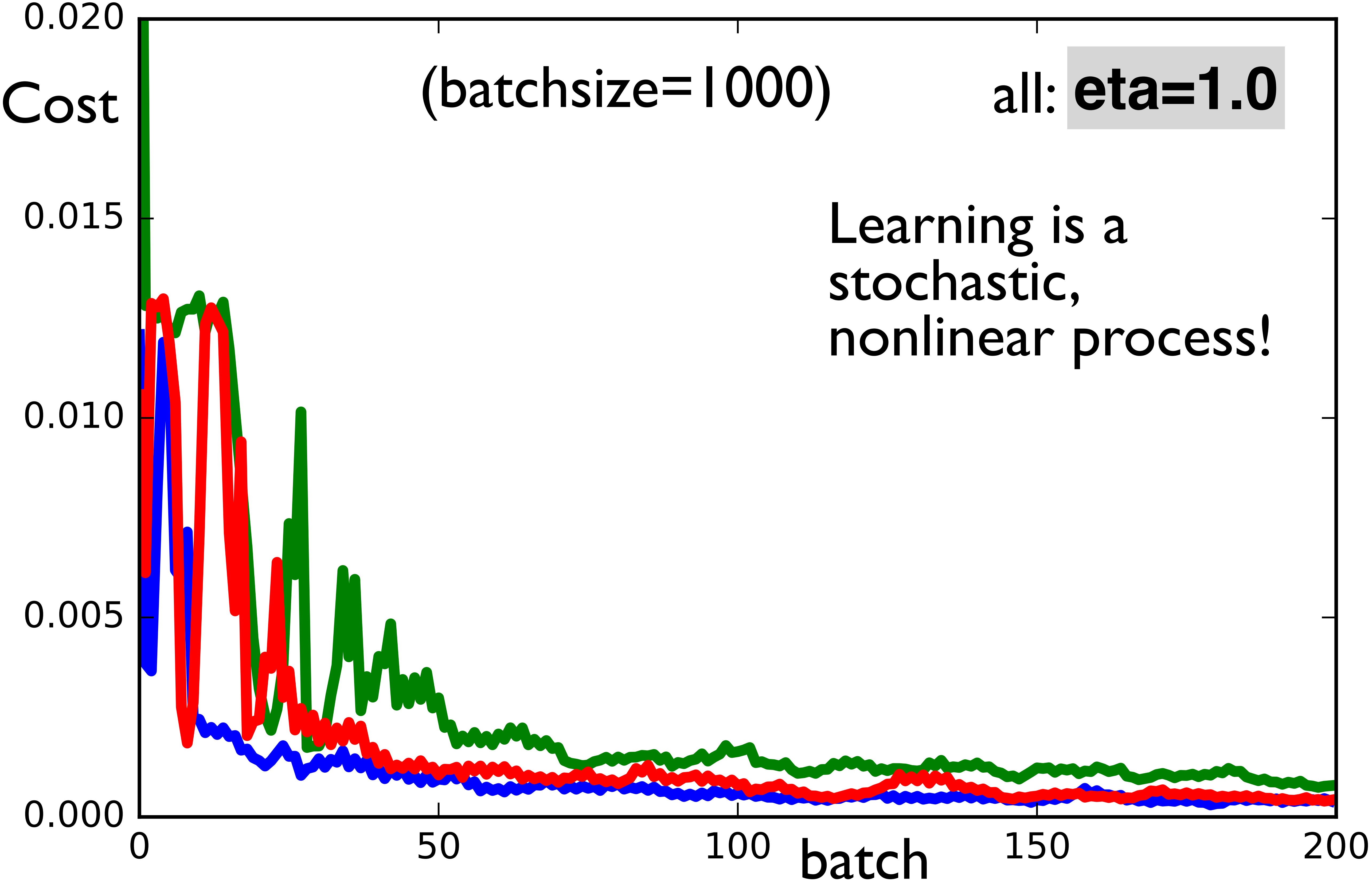
Influence of learning rate (stepsize)



Influence of learning rate (stepsize)



Randomness (initial weights, learning samples)

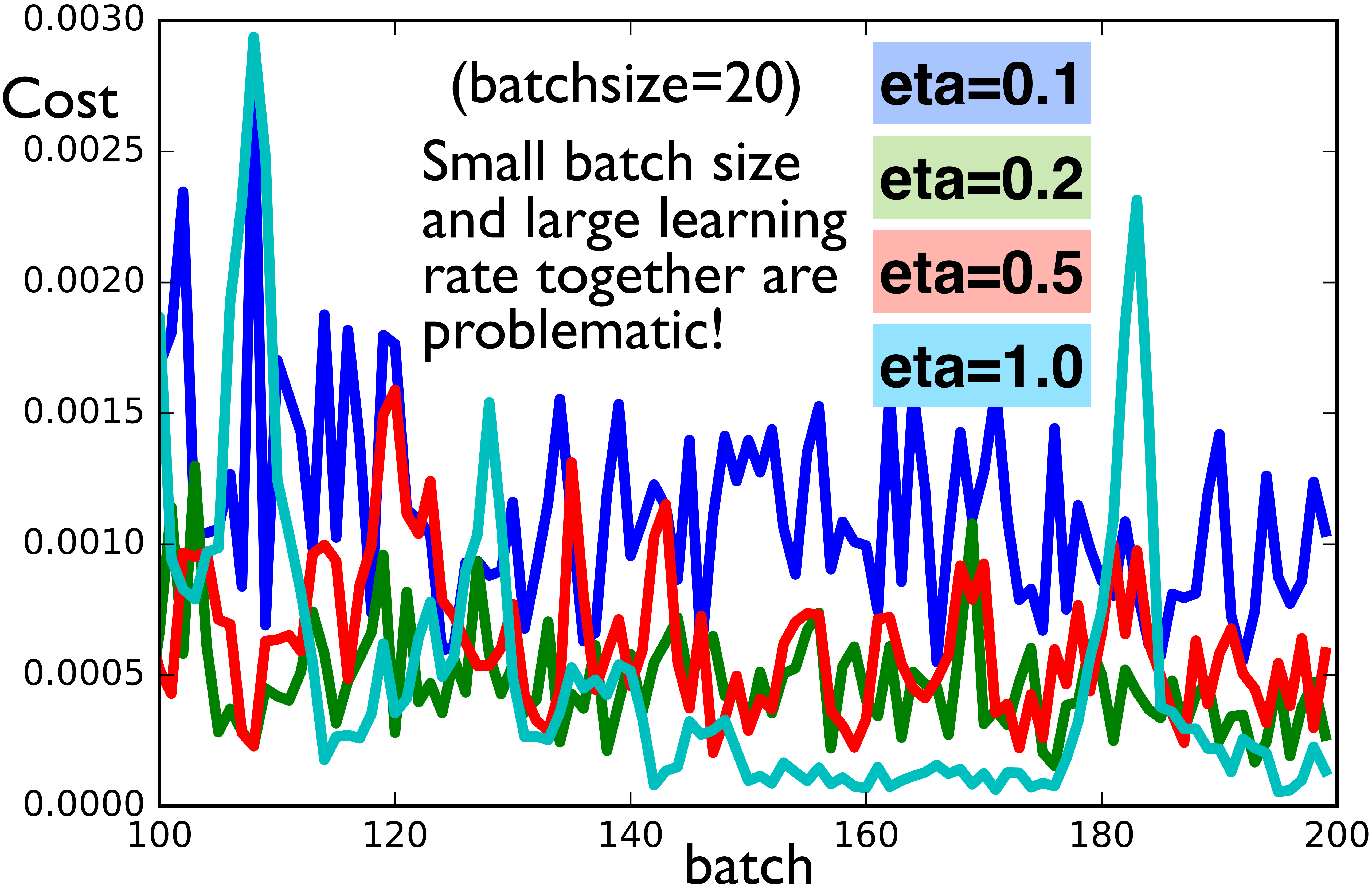


(batchsize=1000)

all: **eta=1.0**

Learning is a
stochastic,
nonlinear process!

Influence of batch size / learning rate



(batchsize=20)

Small batch size and large learning rate together are problematic!

Influence of batch size / learning rate

always >0

$$C(w-\eta\nabla_w C)\approx C(w)-\eta(\nabla_w C)(\nabla_w C)+\ldots$$

new weights

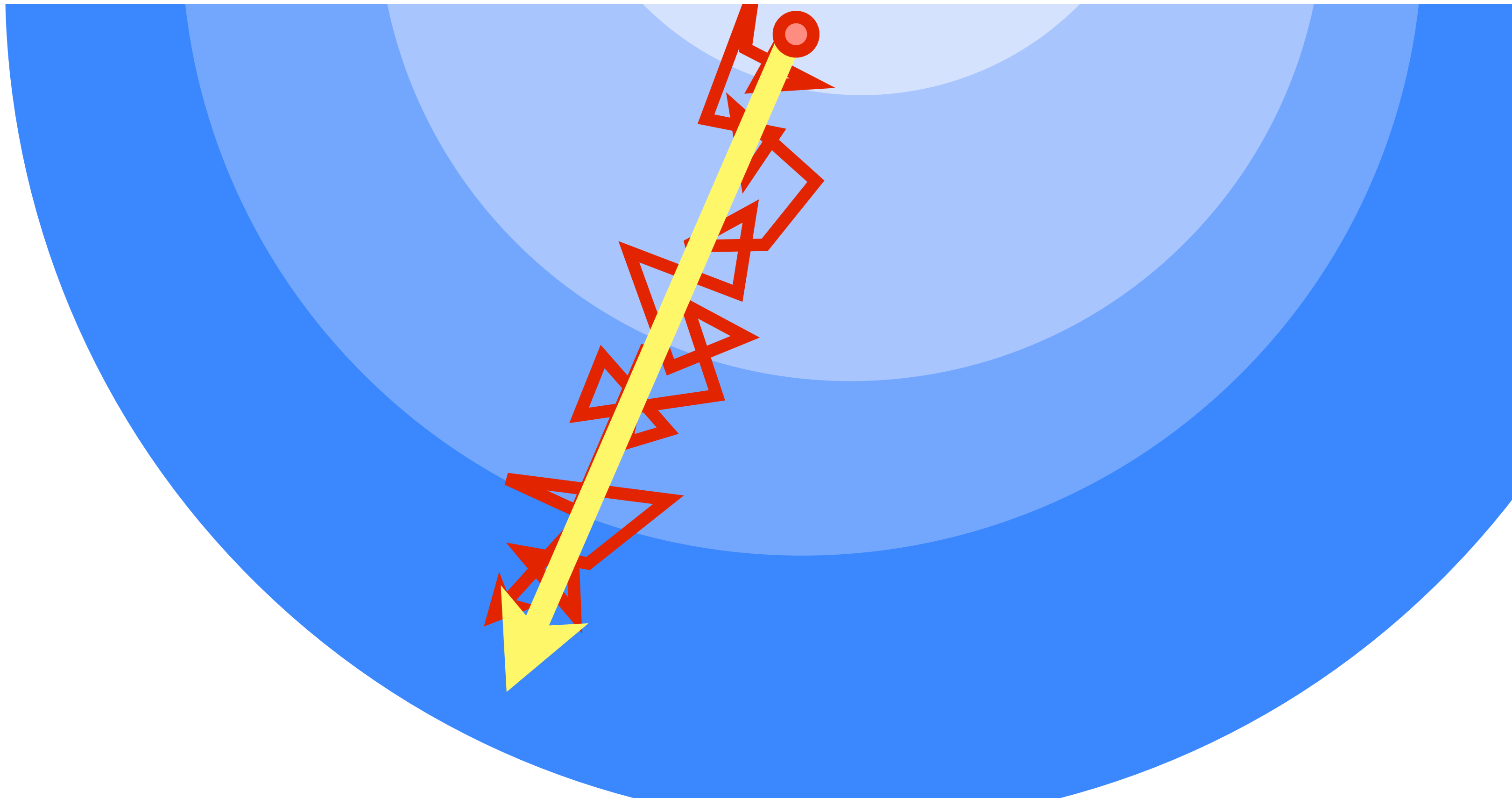
decrease in C!

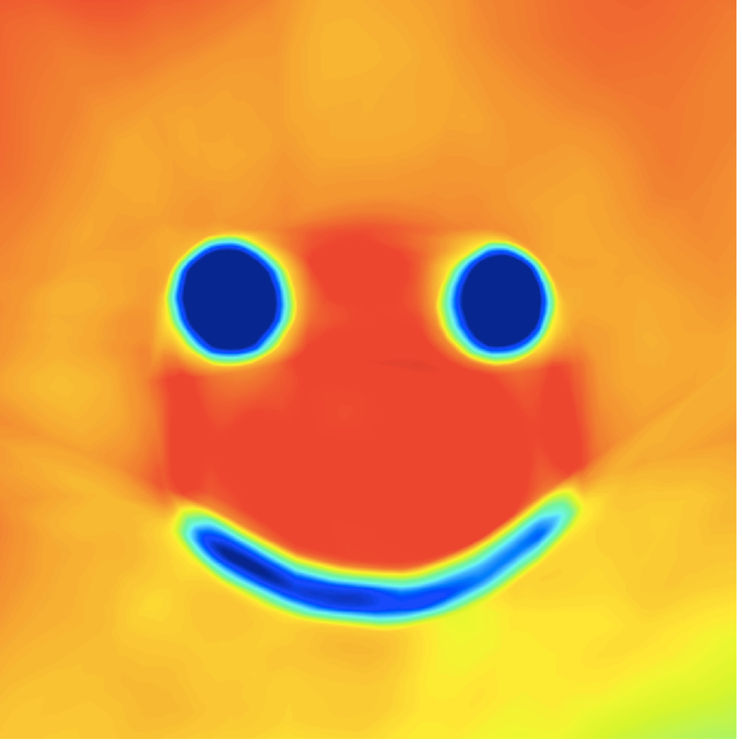
higher order in η

Potential problems:

- step too large: need higher-order terms [will not be a problem near minimum of C]
- approx. of C bad [small batch size: approx. C fluctuates]

Sufficiently small learning rate: multiple training steps (batches) add up, and their average is like a larger batch





Programming a general multilayer neural network & backpropagation was not so hard (once you know it!)

Could now go on to image recognition etc. with the same program!

But: want more flexibility and added features!

For example:
- Arbitrary nonlinear functions for each layer
- Adaptive learning rate
- More advanced layer structures (such as convolutional networks)
- etc.

Keras

- Convenient neural network package for python
- Set up and training of a network in a few lines
- Based on underlying neural network / symbolic differentiation package [which also provides run-time compilation to CPU and GPU]: either ‘theano’ or ‘tensorflow’ [User does not care]

Keras

From the website **keras.io**

“Keras is a high-level neural networks API, written in Python and capable of running on top of either TensorFlow or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.”

```
from keras import *
from keras.models import Sequential
from keras.layers import Dense
```

Defining a network

layers with 2,150,150,100,1 neurons

```
net=Sequential()
net.add(Dense(150, input_shape=(2,), activation='relu'))
net.add(Dense(150, activation='relu'))
net.add(Dense(100, activation='relu'))
net.add(Dense(1, activation='relu'))
```

‘Compiling’ the network

```
net.compile(loss='mean_squared_error',
             optimizer=optimizers.SGD(lr=0.1),
             metrics=['accuracy'])
```

```
from keras import *
from keras.models import Sequential
from keras.layers import Dense
```

Defining a network

“Sequential”: the usual neural network, with several layers

```
net=Sequential()
net.add(Dense(150, input_shape=(2,), activation='relu'))
net
net
```

“Dense”:“fully connected layer” (all weights there)

input_shape: number of input neurons

‘relu’: rectified linear unit

‘Compiling’ the network

SGD=stoch. gradient descent

‘loss’=cost

```
net.compile(loss='mean_squared_error',
                optimizer=optimizers.SGD(lr=0.1),
                                   curacy'])
```

lr=learning rate=stepsize

Training the network

```
batchsize=20
batches=200
costs=zeros(batches)

for k in range(batches):
    y_in,y_target=make_batch()
    costs[k]=net.train_on_batch(y_in,y_target)[0]
```

y_in   array dimensions ‘batchsize’ x 2

y_target   array dimensions ‘batchsize’ x 1

(just like before, for our own python code)

Predicting with the network

```
y_out=net.predict_on_batch(y_in)
```

y_in array dimensions ‘batchsize’ x 2
y_out array dimensions ‘batchsize’ x 1
(just like before, for our own python code)

Homework

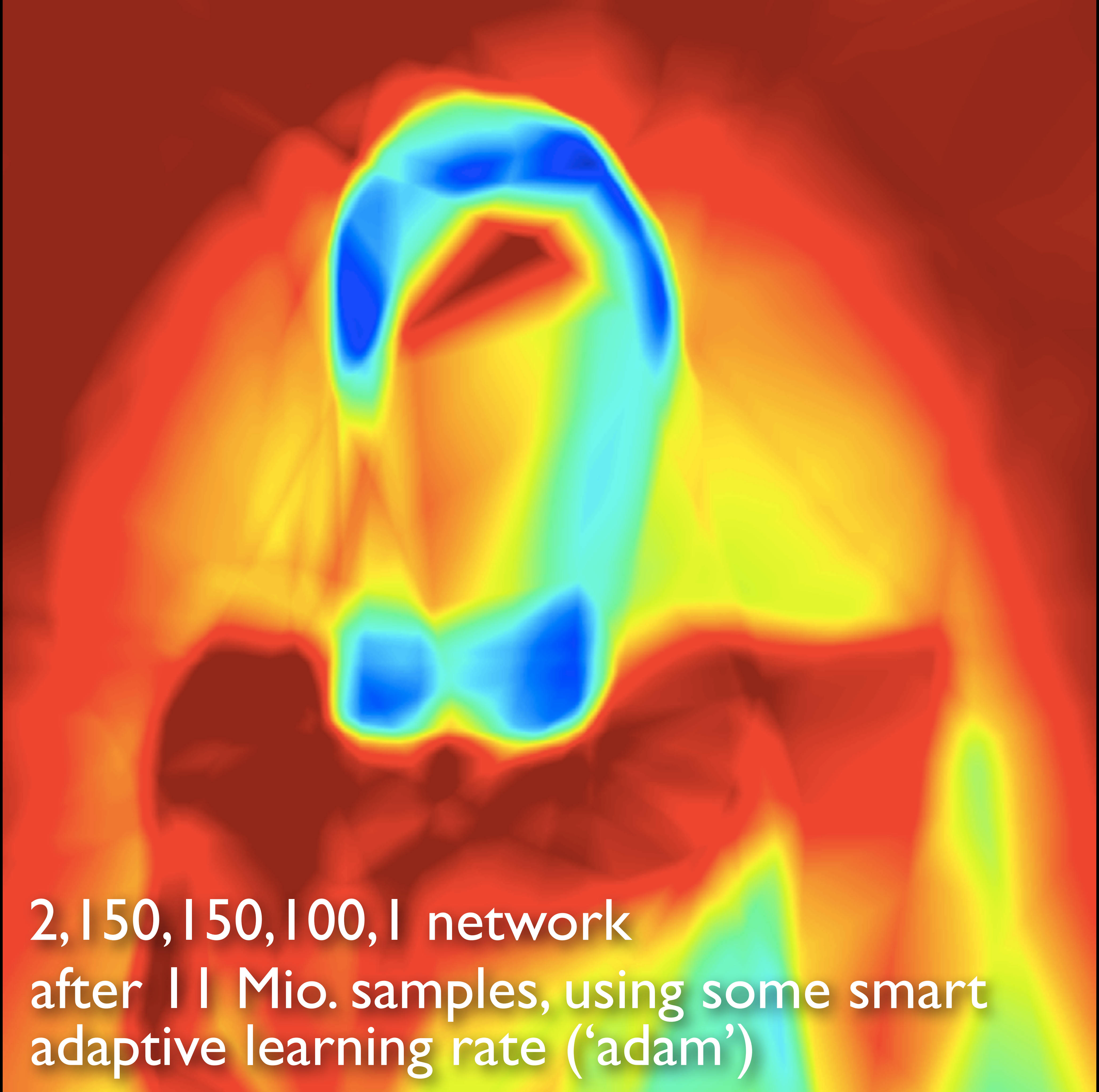
Explore how well the network can reproduce various features of target images, and how that depends on the network layout!

Aspects to consider (& I do not claim to know all the answers!):

How good are other nonlinear functions? [e.g. sigmoids or your own favorite f(z)]

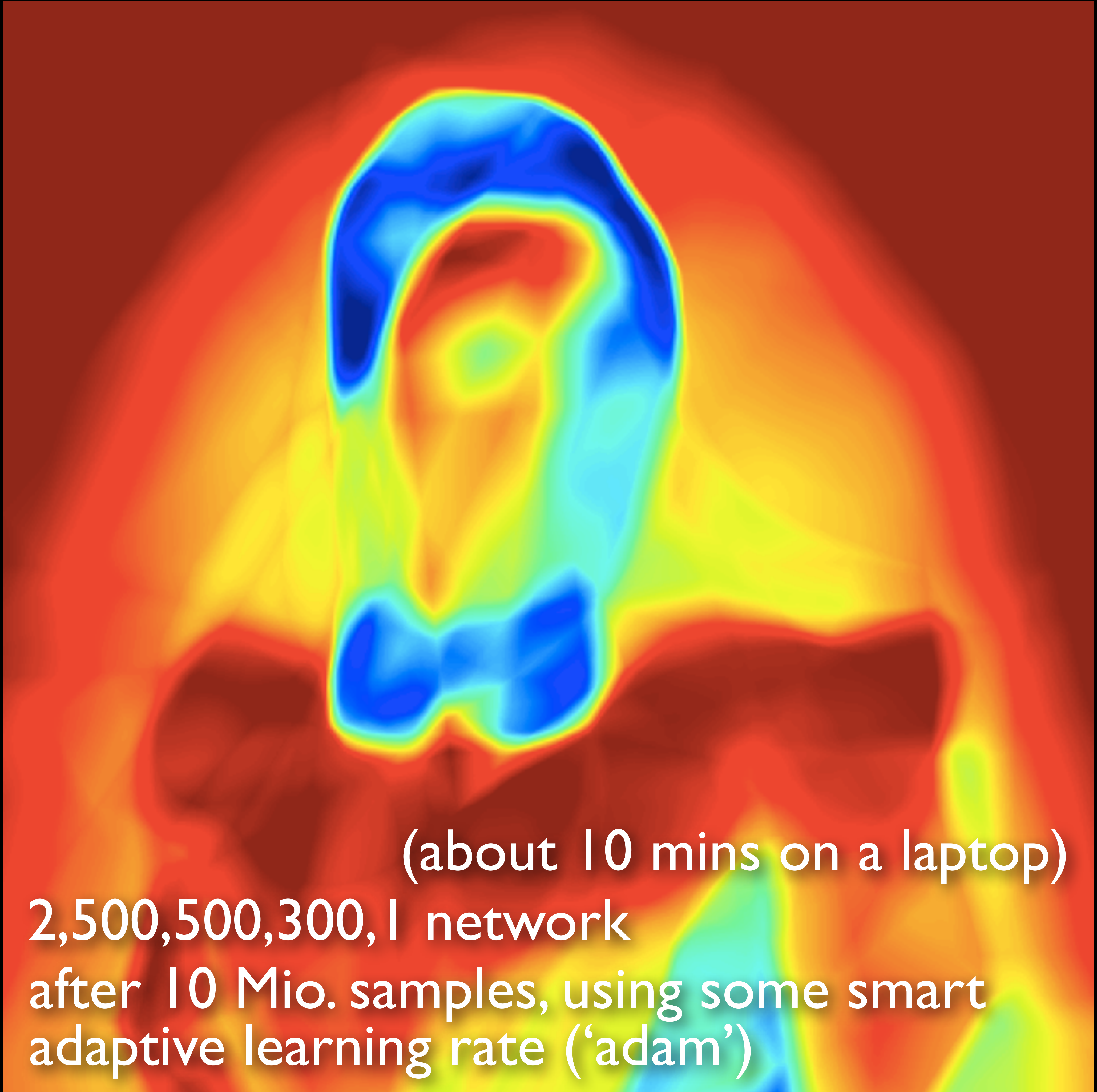
Given a fixed total number of weights, is it better to go deep (many layers) or shallow?

Bonus: After training, try to 'prune' the network, i.e. delete neurons whose deletion does not increase the cost function too much!



2,150,150,100,1 network

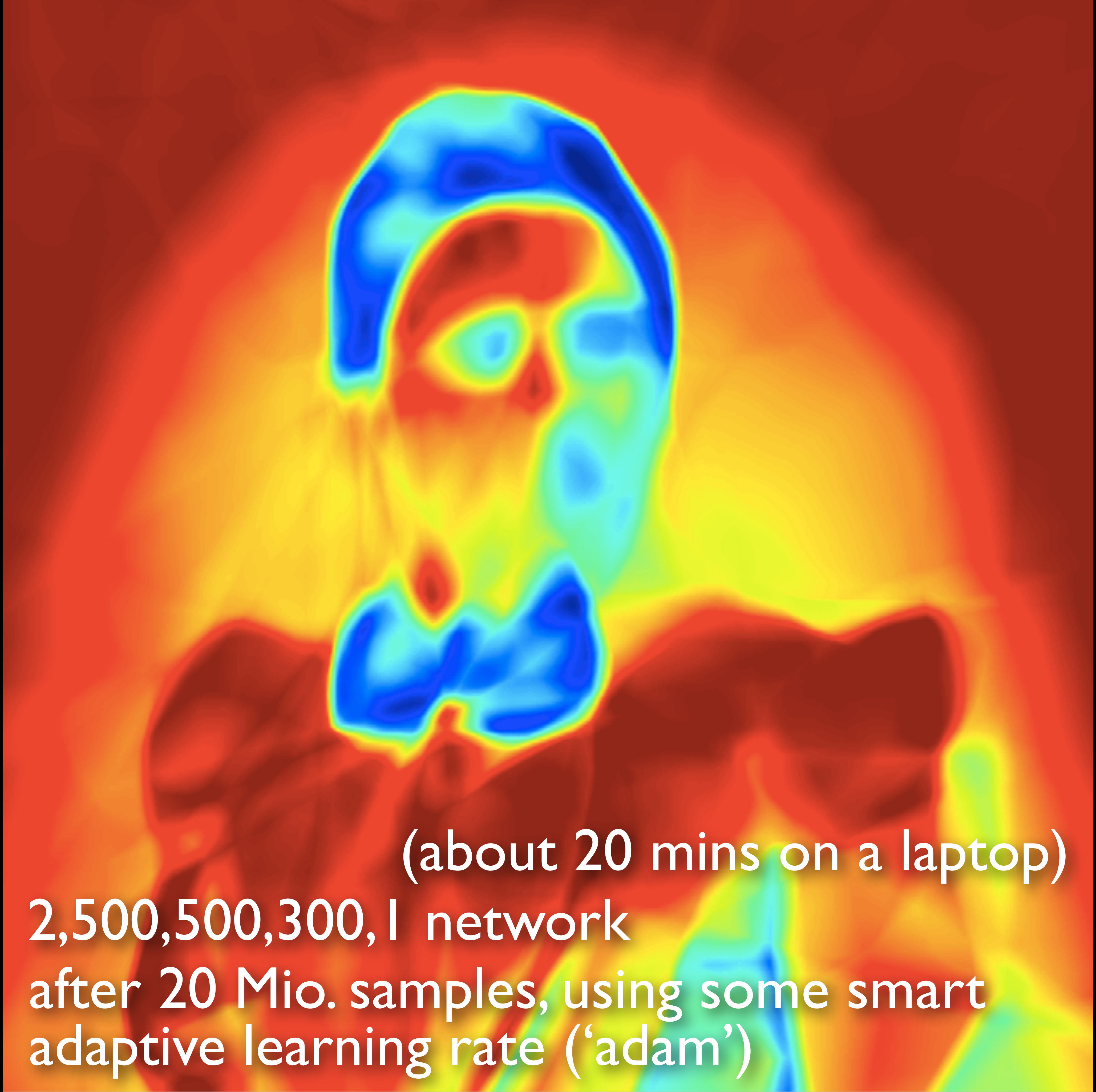
after 11 Mio. samples, using some smart adaptive learning rate (‘adam’)



(about 10 mins on a laptop)

2,500,500,300,1 network

after 10 Mio. samples, using some smart adaptive learning rate (‘adam’)



(about 20 mins on a laptop)

2,500,500,300,1 network

after 20 Mio. samples, using some smart adaptive learning rate ('adam')

Emmy Noether (1882-1935)
Erlangen, Göttingen, Bryn Mawr/USA

→ "Emmy Noether" !

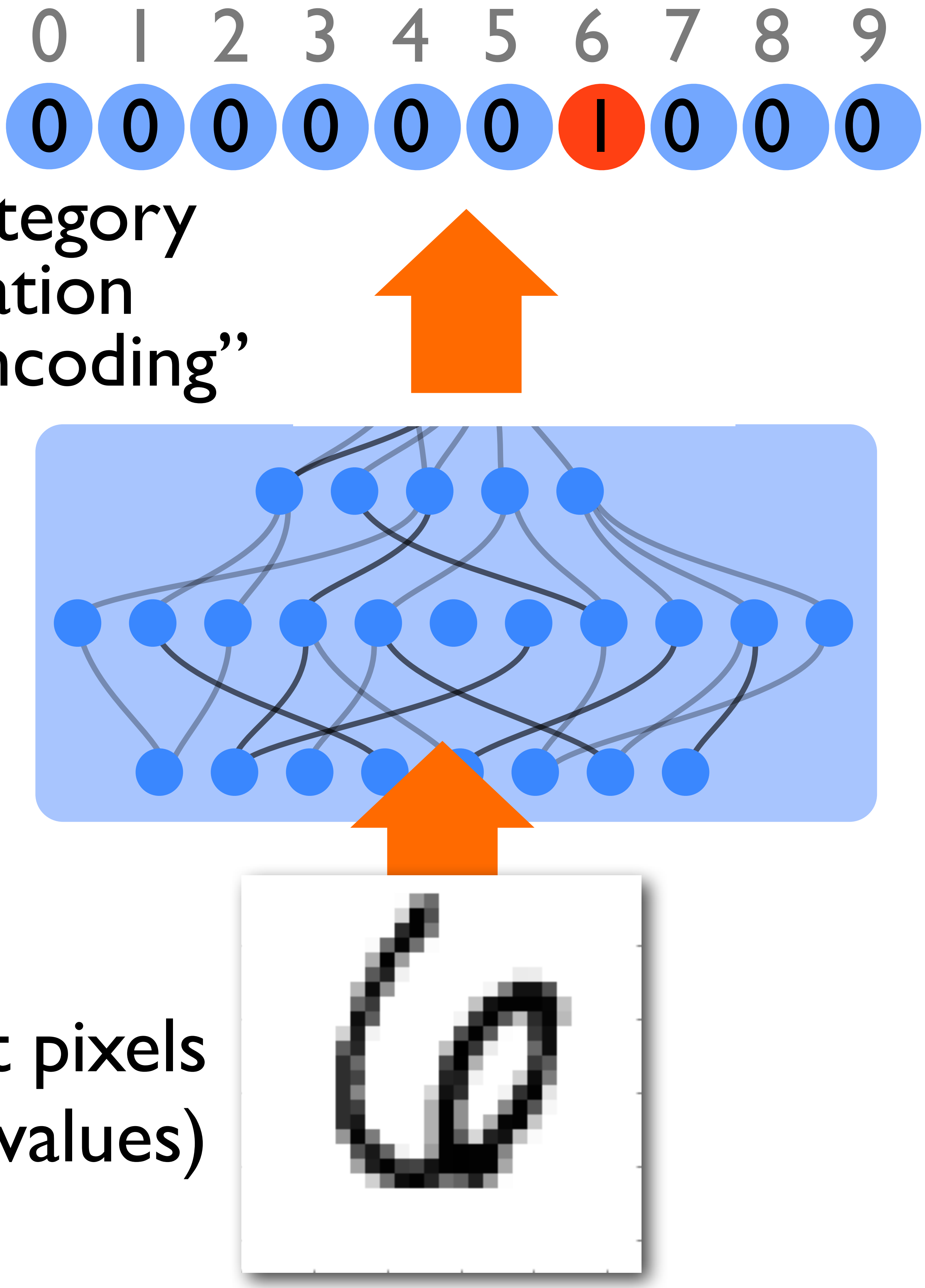
Handwriting recognition

“MNIST” data set (for postal code recognition)

http://yann.lecun.com/exdb/mnist/

Handwriting recognition

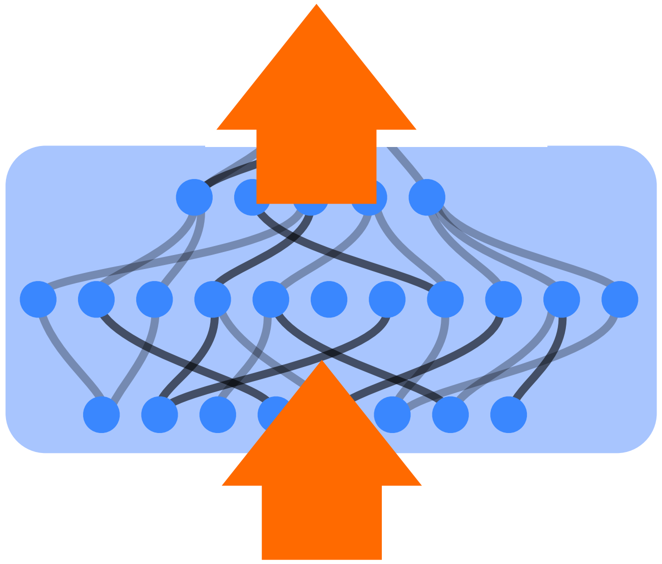
Will learn:
- distinguish categories
- “softmax” nonlinearity for probability distributions
- “categorical cross-entropy” cost function
- training/validation/test data
- “overfitting” and some solutions



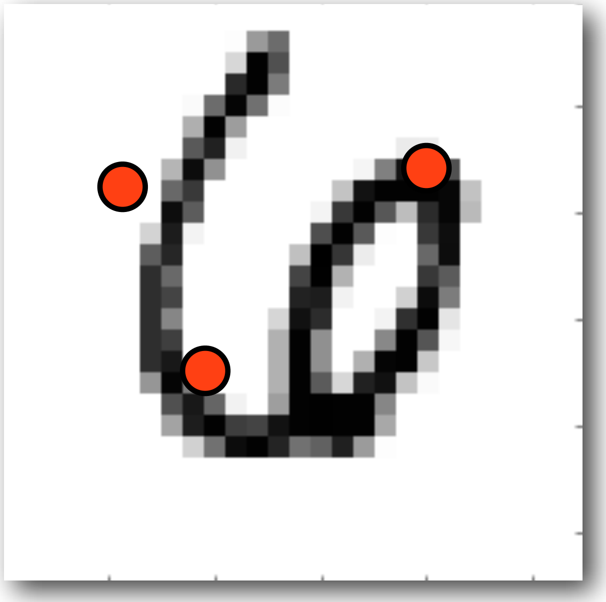
output: category classification
"one-hot encoding"

28x28 input pixels
(=784 gray values)

value

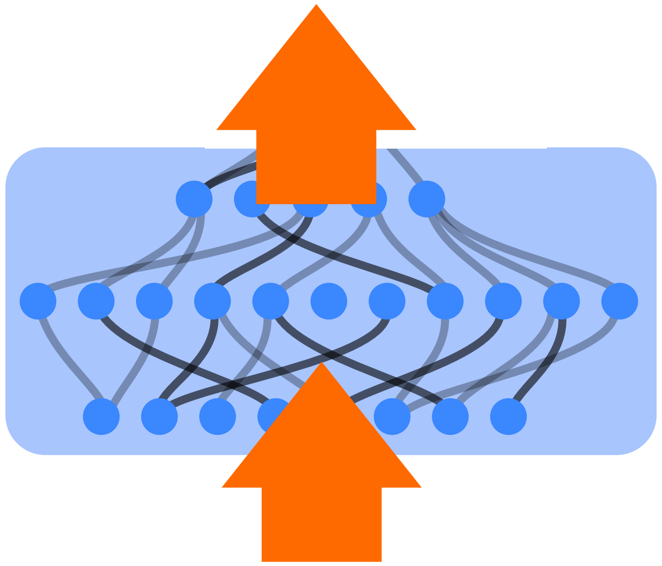


x,y
(coordinates)



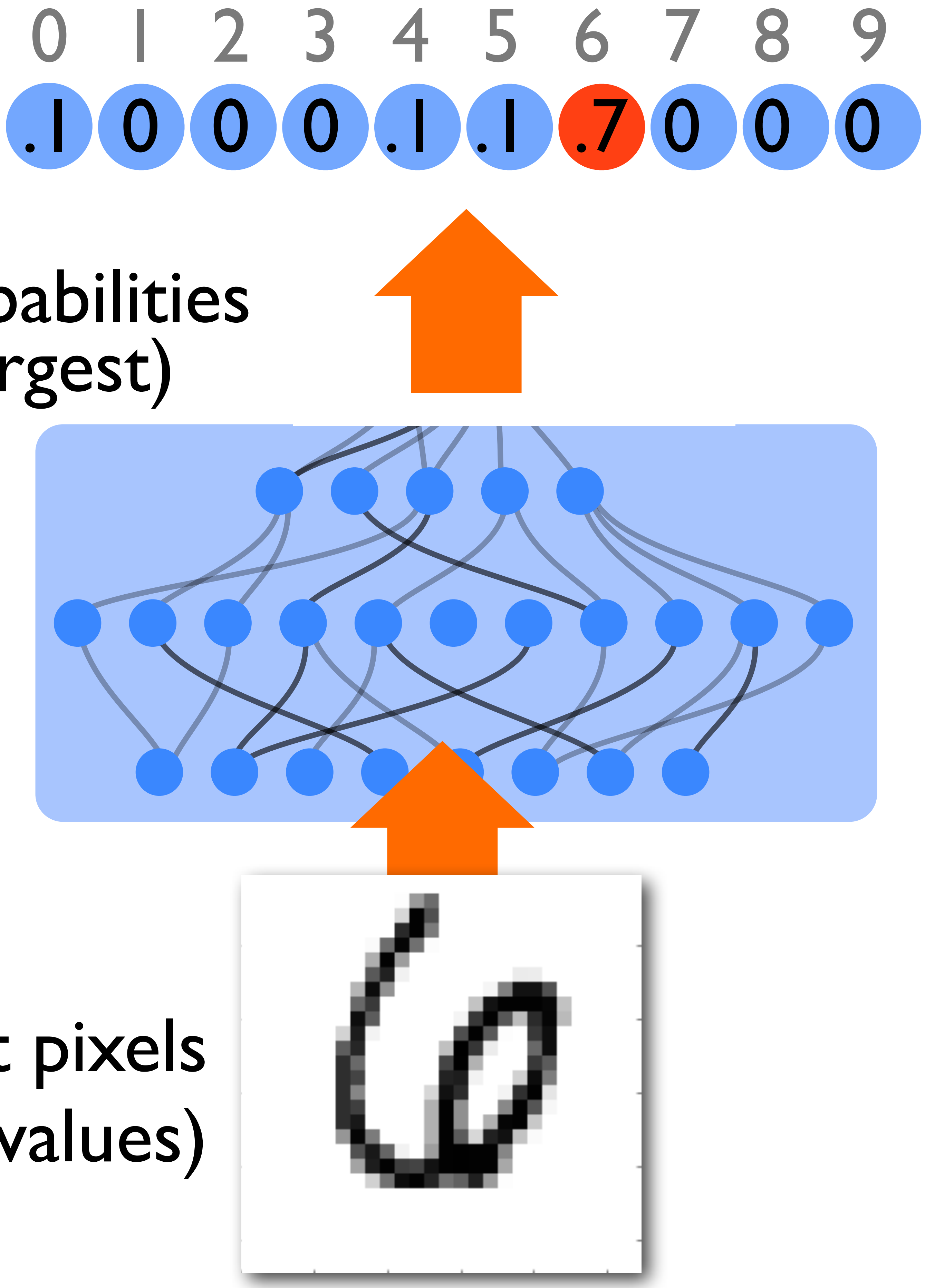
network learns to represent **one** specific image

category



val1,val2,val3,val4,...
(all pixels)

network learns to classify **a whole class of images**

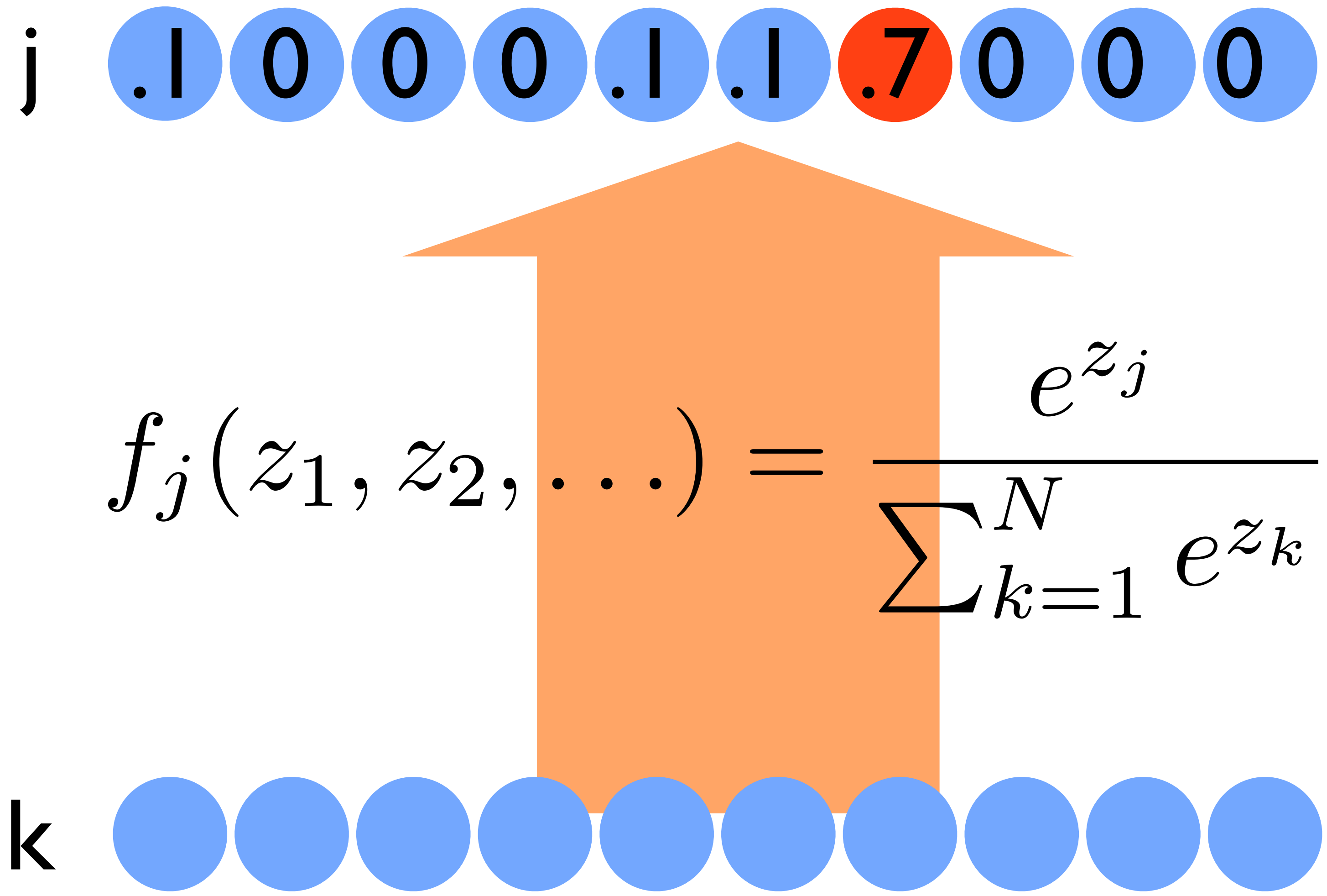


output: probabilities
(select largest)

28x28 input pixels
(=784 gray values)

“Softmax” activation function

Generate normalized probability distribution, from arbitrary vector of input values



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

(multi-variable 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'))
```

Entropy

For any probability distribution:

$$S = -\sum_j p_j \ln p_j$$

(non-negative, additive for factorizable distributions)

Categorical cross-entropy cost function

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

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

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)

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

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

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

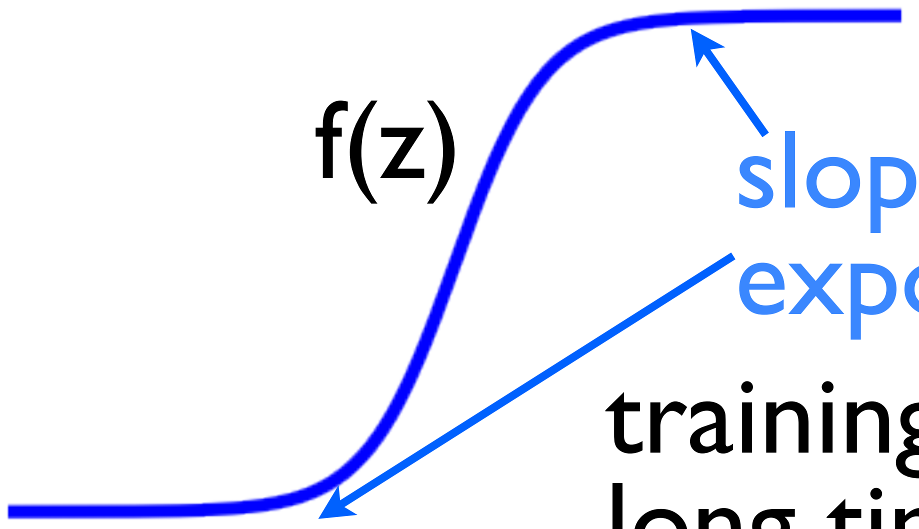
derivative of input values

Compare situation for quadratic cost function

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

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

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



slope becomes exponentially small!

training may get ‘stuck’ for a long time!

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")

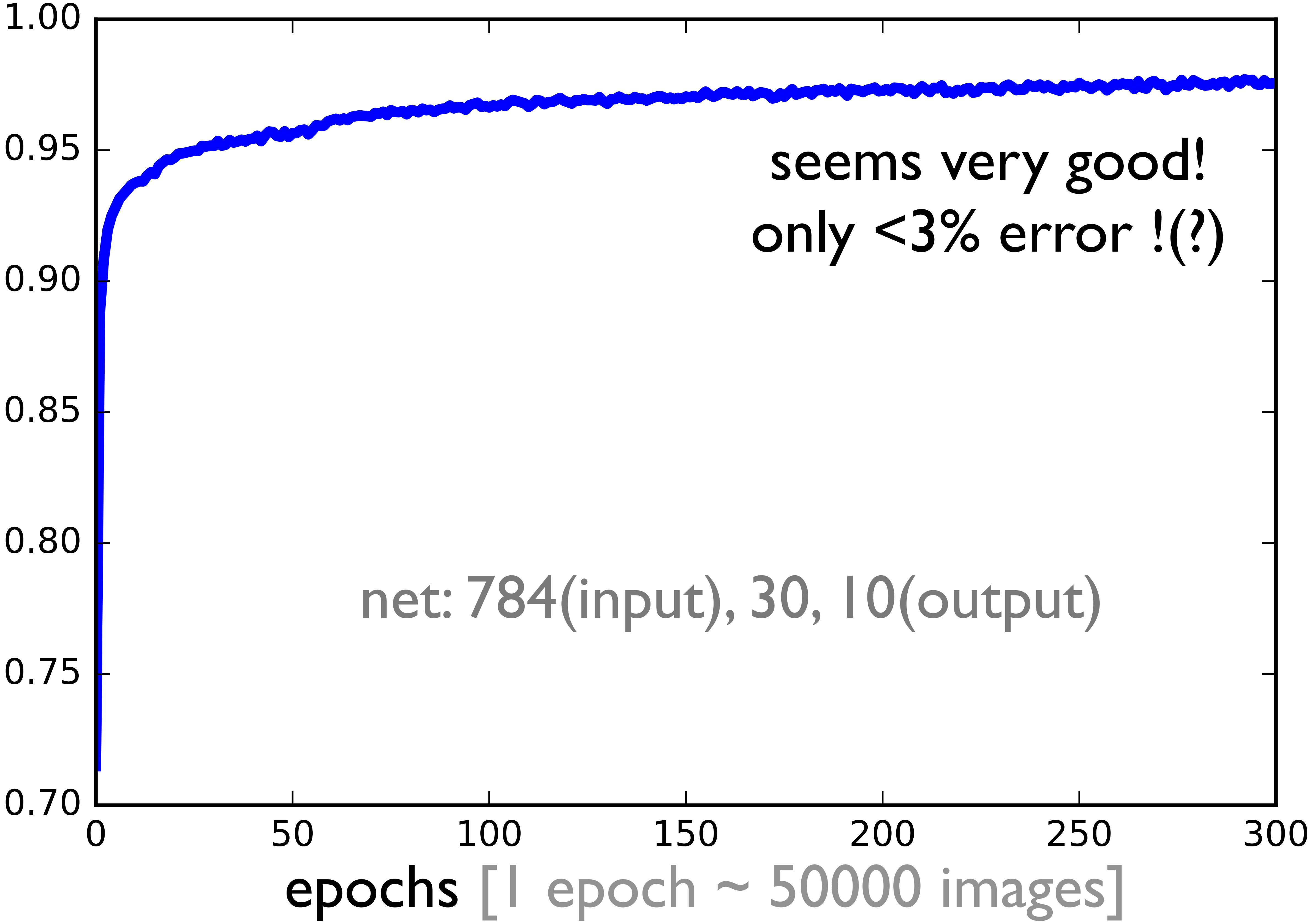
in keras:

```
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

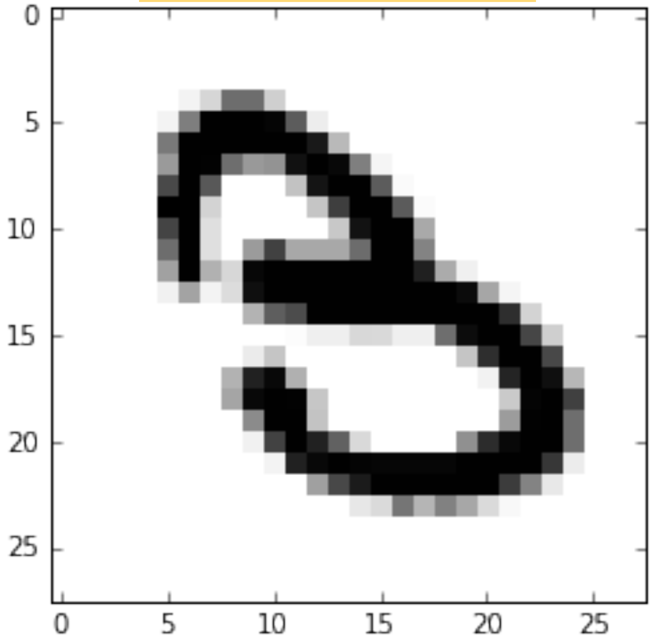


seems very good!
only <3% error !(?)

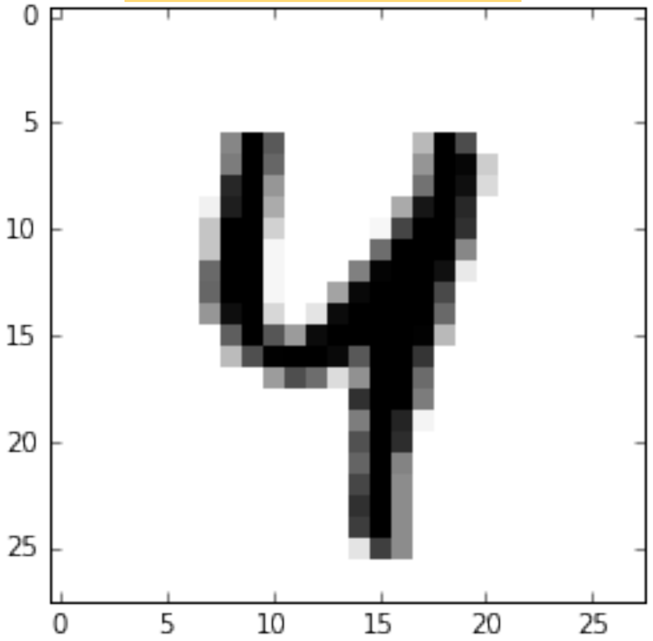
net: 784(input), 30, 10(output)

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

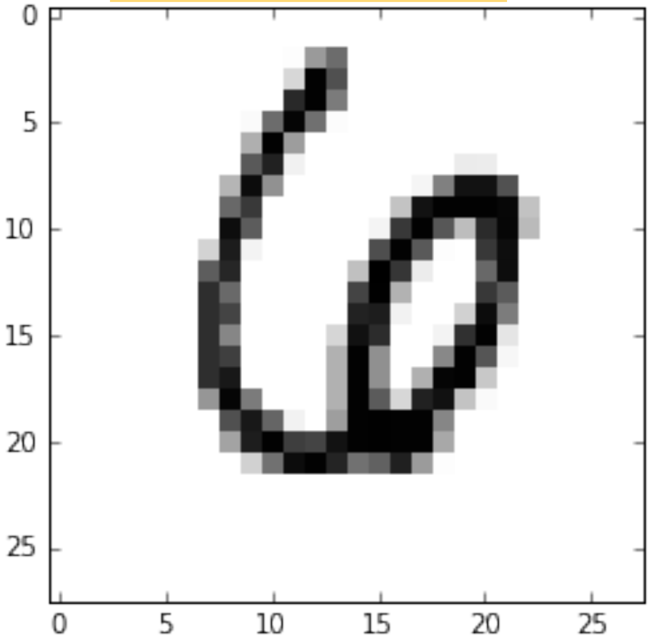
“8” (3 !)



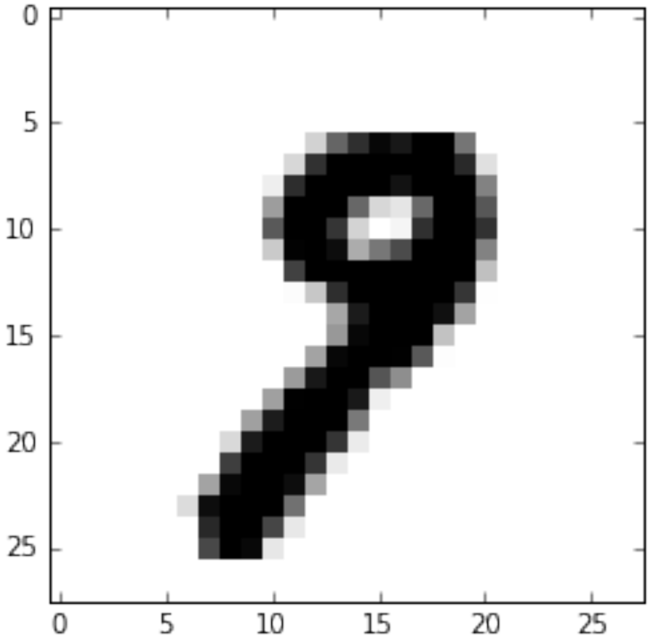
“7” (4 !)



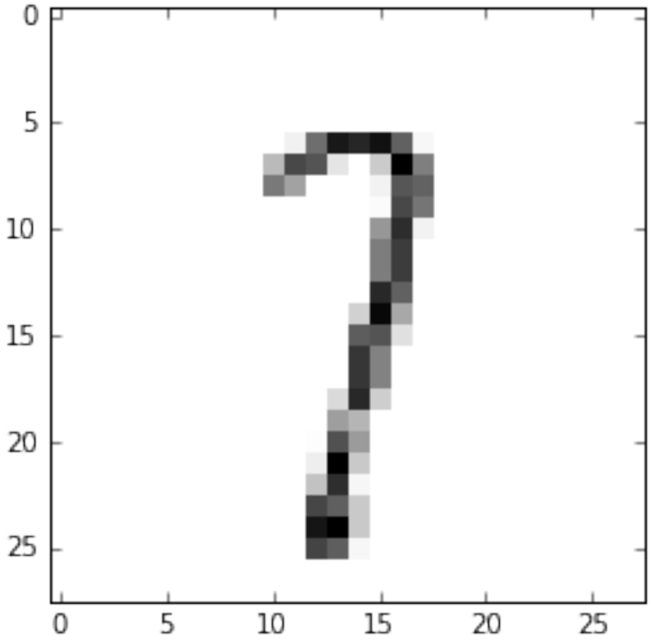
“5” (6 !)



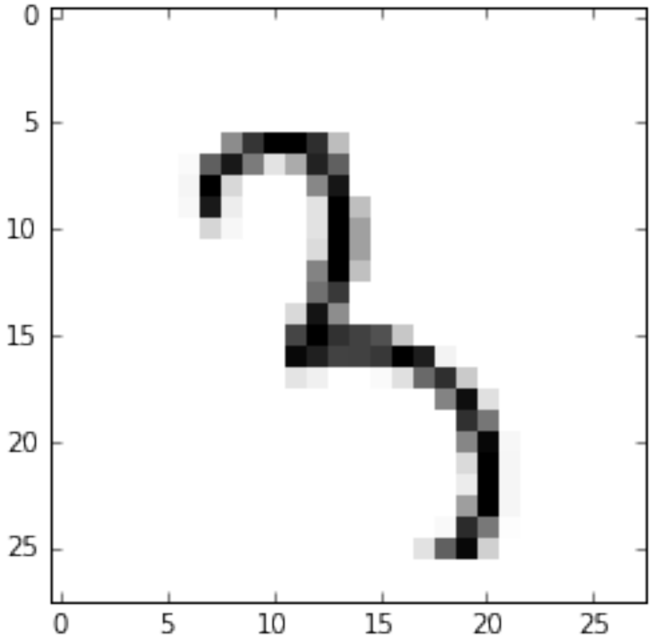
“7” (9 !)



“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

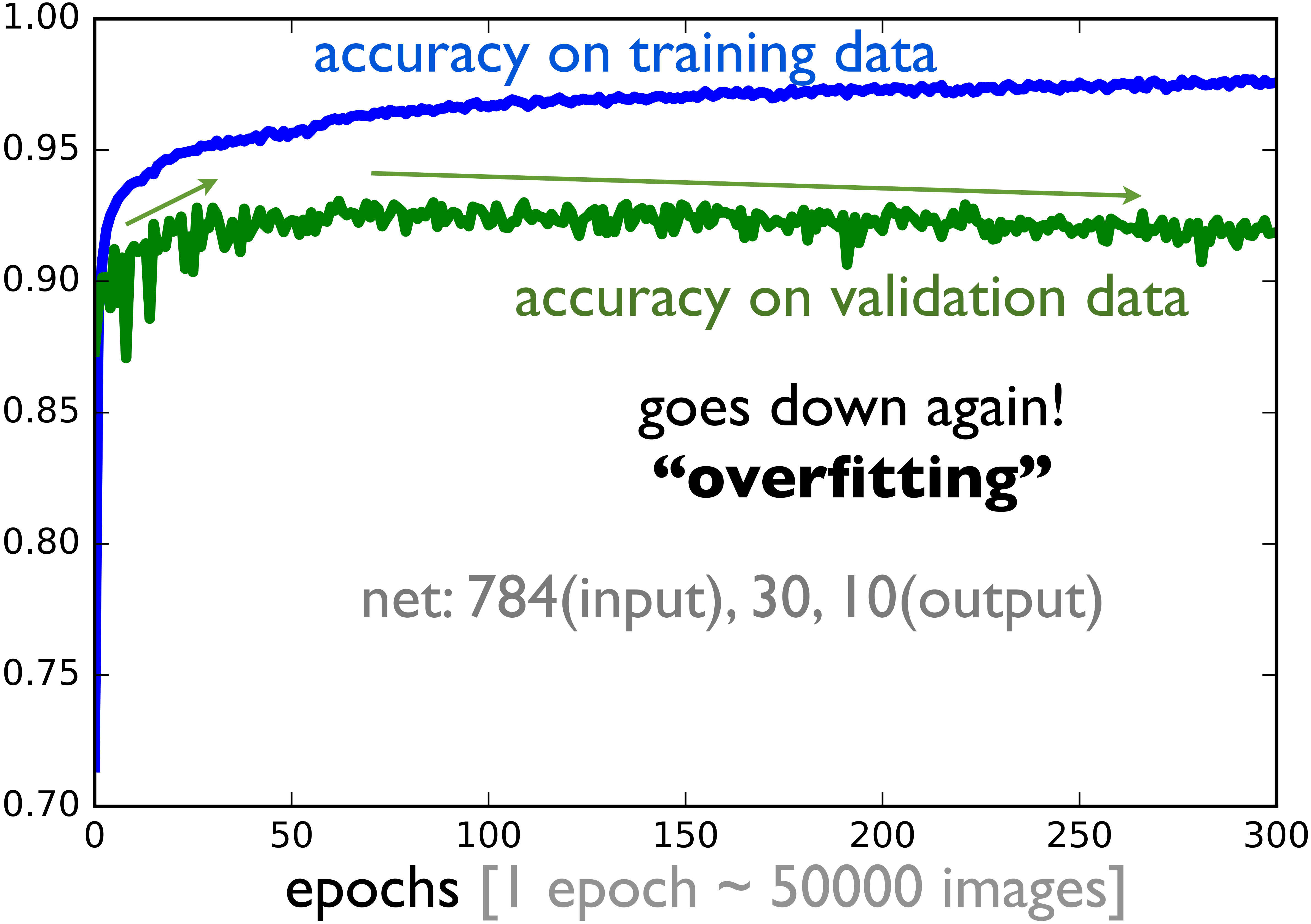
10000 images

Test set

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

(numbers for our MNIST example)

Accuracy during training



goes down again!

“overfitting”

net: 784(input), 30, 10(output)

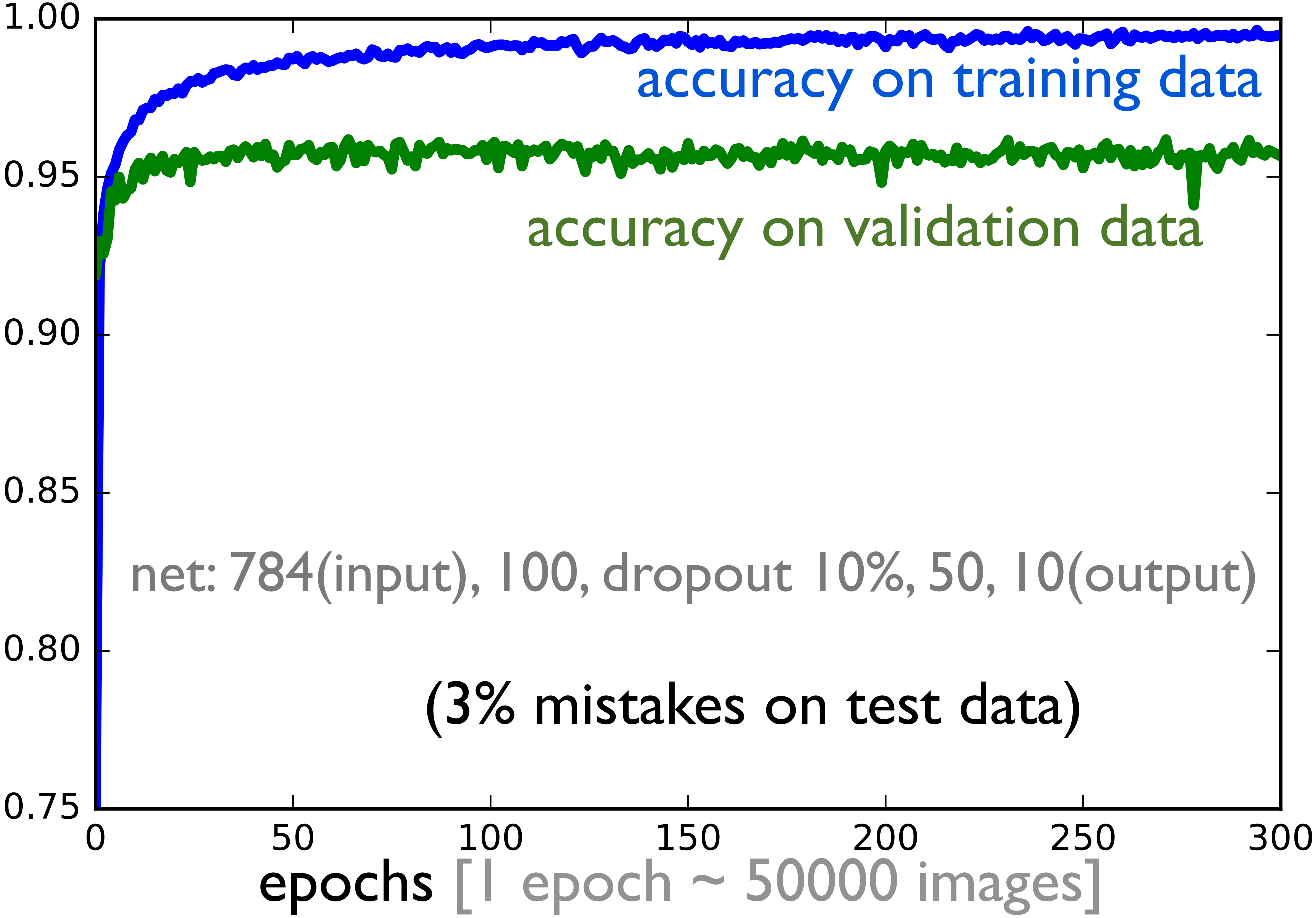
“Overfitting”

- 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

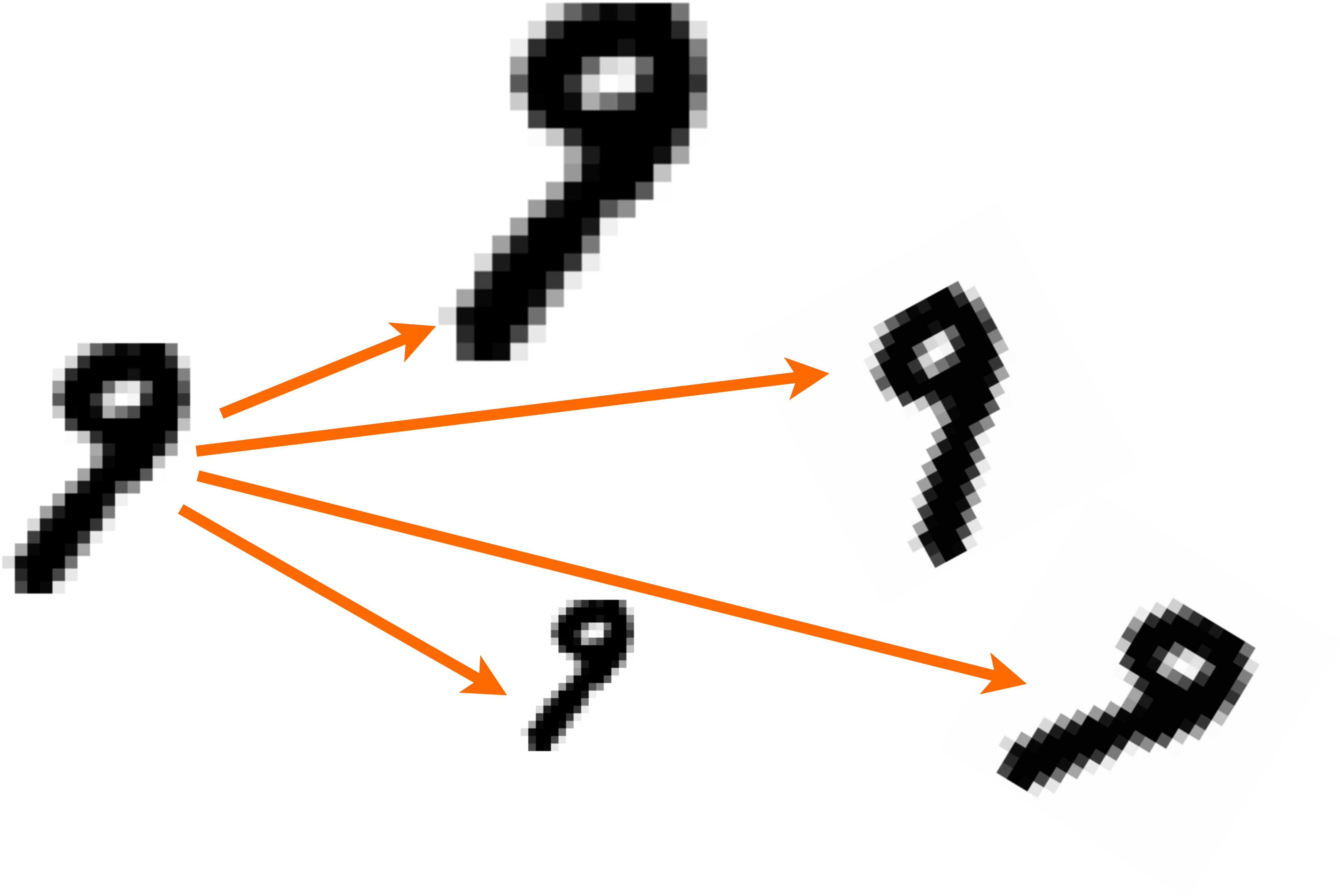
Accuracy during training



net: 784(input), 100, dropout 10%, 50, 10(output)

(3% mistakes on test data)

Generating new training images by transformations



Comparison of machine learning methods on MNIST

>60 entries on http://yann.lecun.com/exdb/mnist/

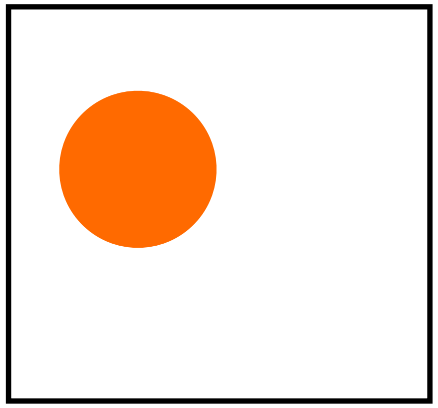
	Error rate
Linear classifier (1 layer NN)	12%
2 layer (300 hidden)	4.7%
2 layer (800 hidden)	1.6%
2 layer (300 hidden), with image preprocessing (deskewing)	1.6%
2 layer (800 hidden), distorted images	0.7%
6 layers, distorted images 784/2500/2000/1500/1000/500/10	0.35%
conv. net “LeNet-1”	1.1%
committee of 35 conv. nets, with distorted images	0.23%

Homework

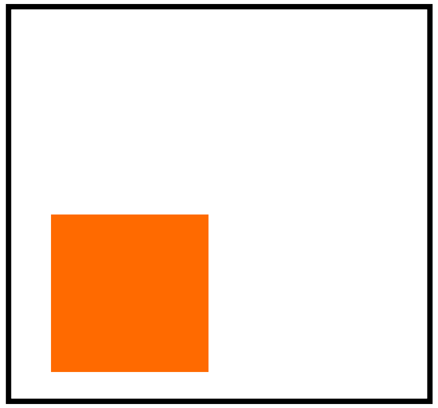
Explore how well the network can do if you add noise to the images (or you occlude parts of them!)

Note: Either use the existing net, or train it explicitly on such noisy/occluded images!

Apply image recognition to some algorithmically generated images



“circle”



“square”

Convolutional Networks

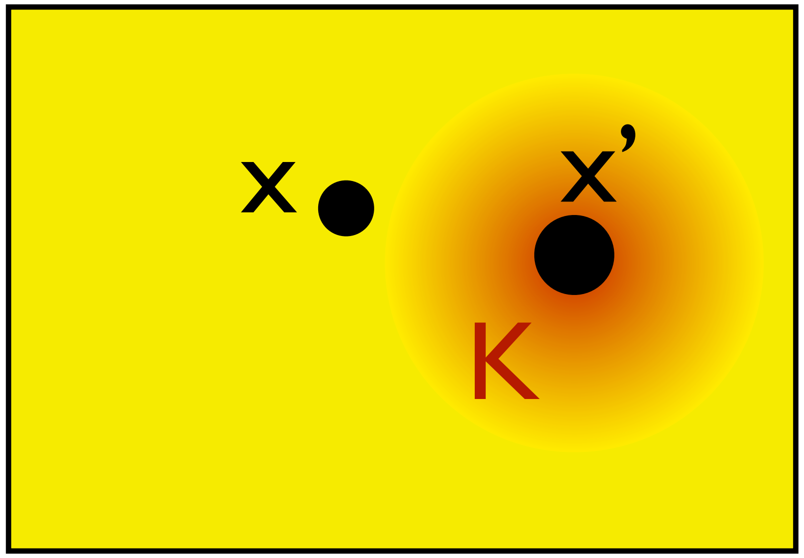
Exploit translational invariance!

different image, same meaning!

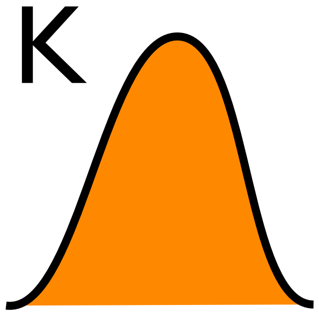
Convolutions

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

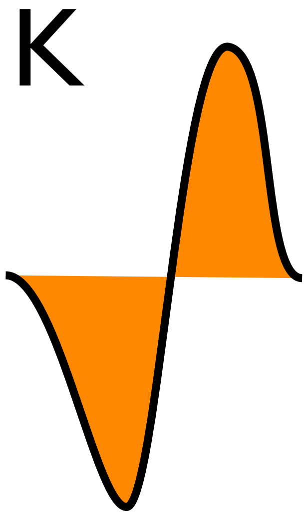
“kernel”



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

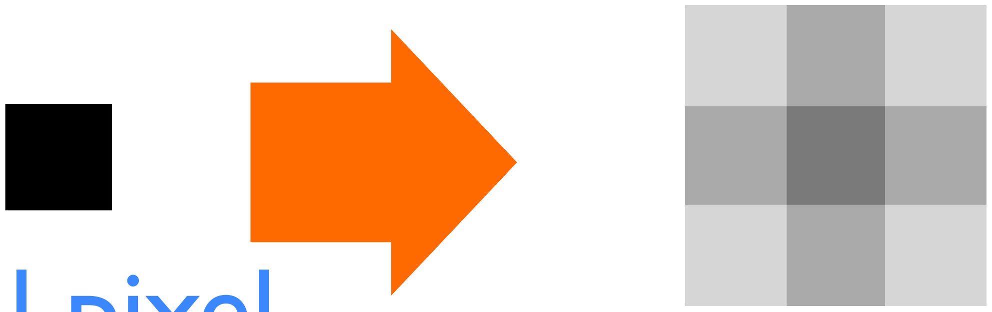


smoothing



(approx.) derivative

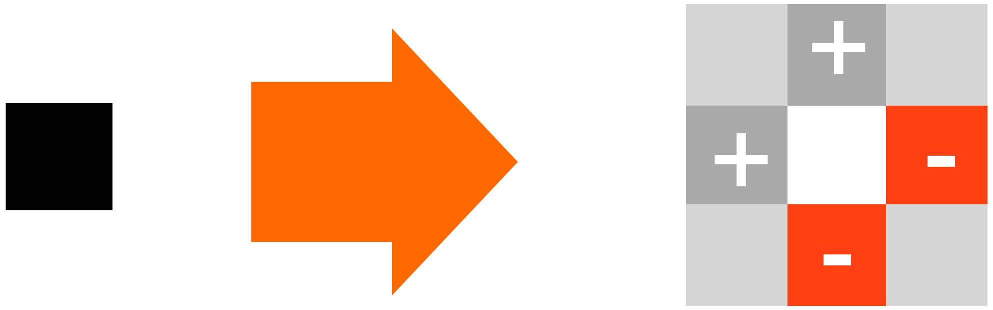
Image filtering: how to blur...



original pixel

resulting pattern

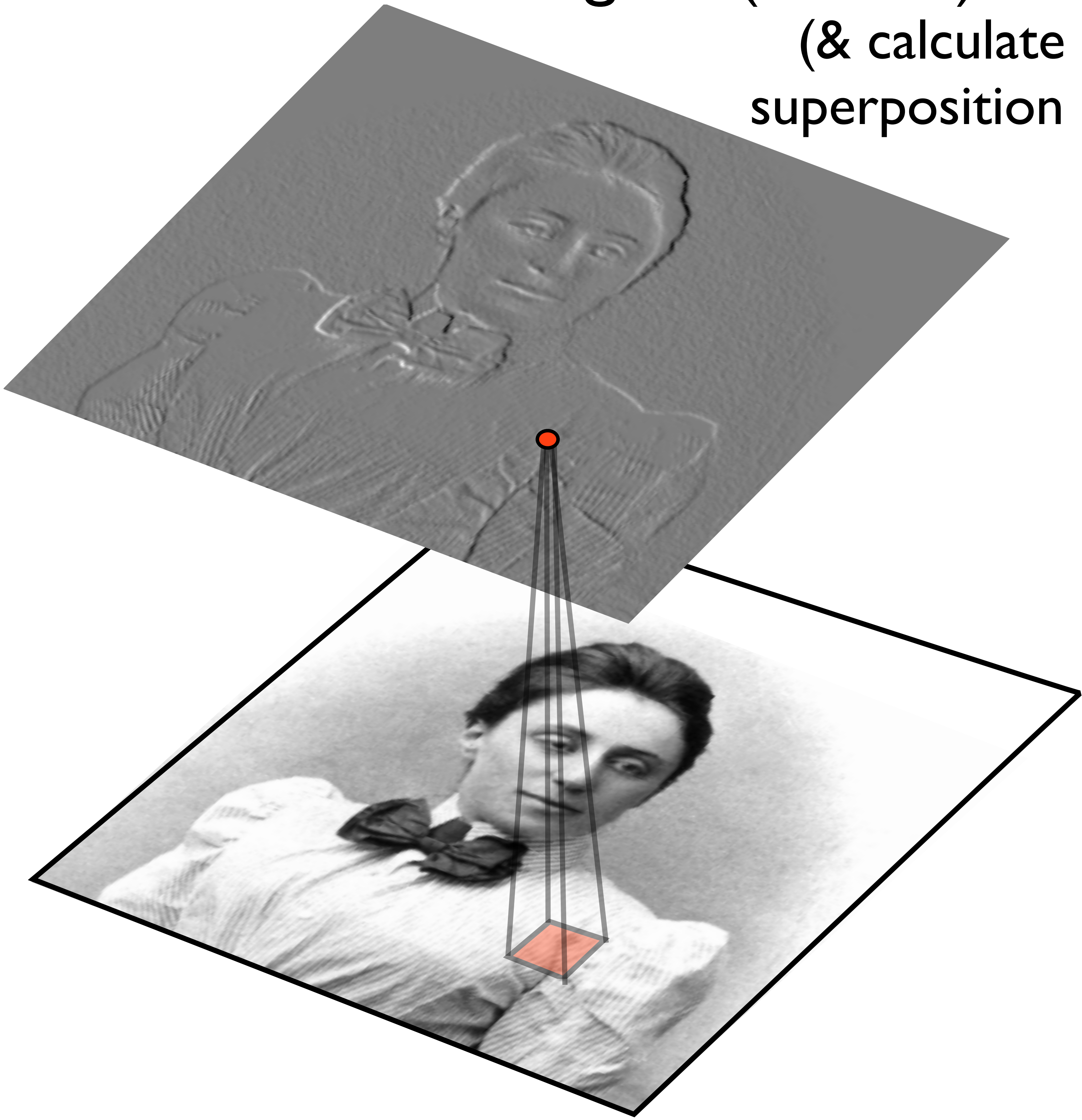
Image filtering: how to obtain contours...



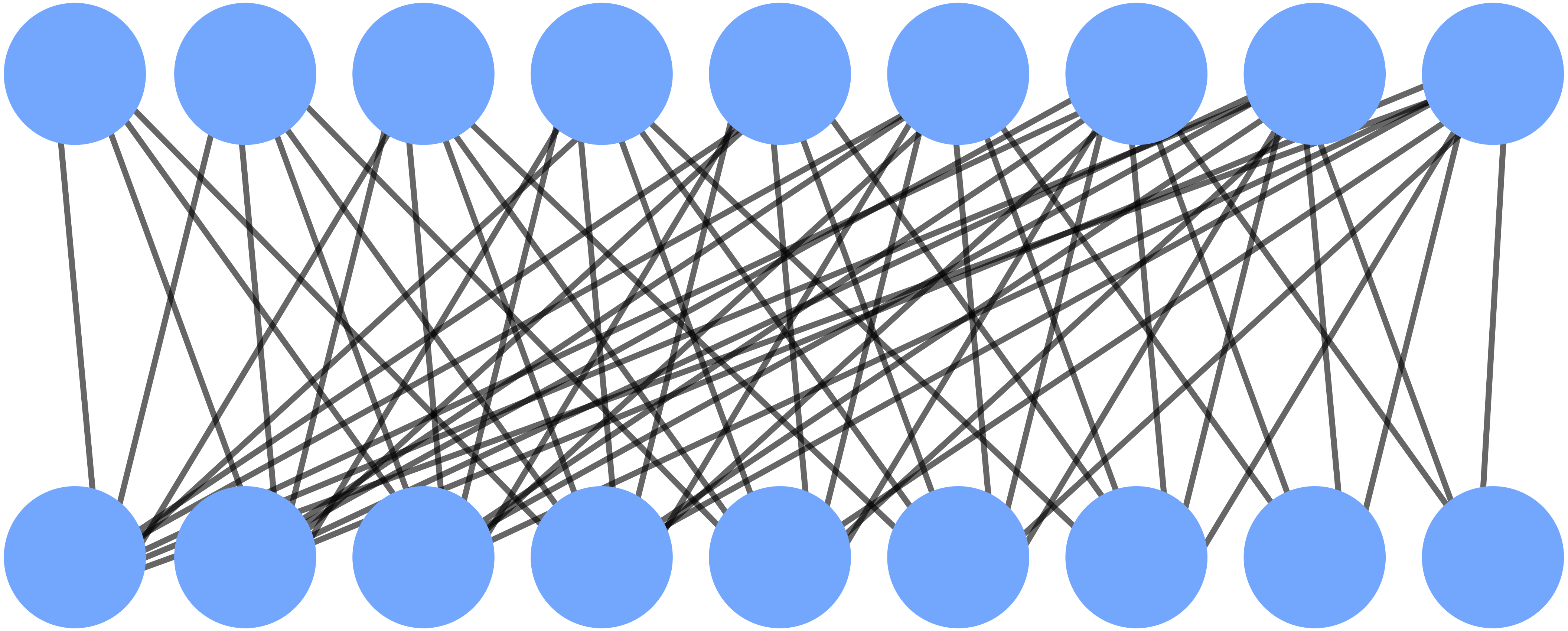
Alternative view:

Scan kernel over original (source) image

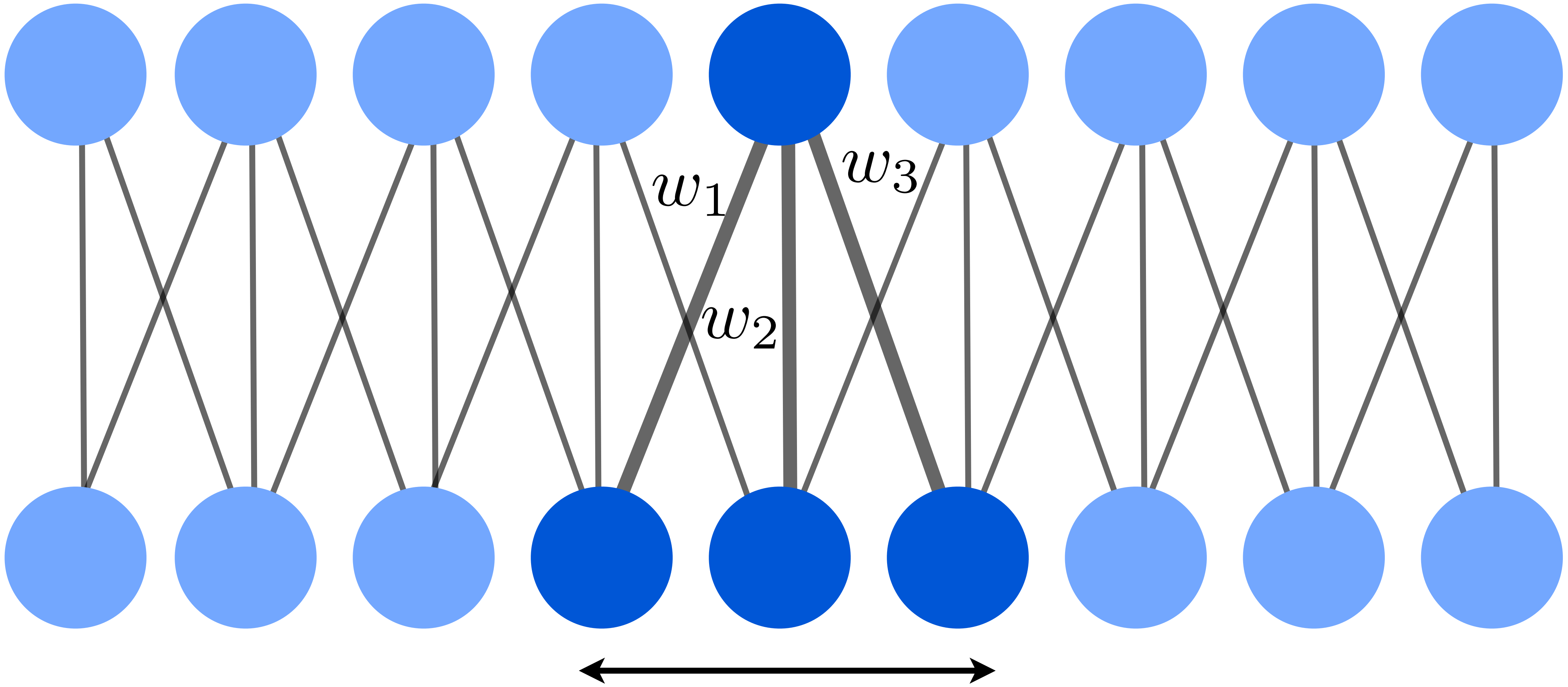
(& calculate linear, weighted superposition of original pixel values)



“Fully connected (dense) layer”



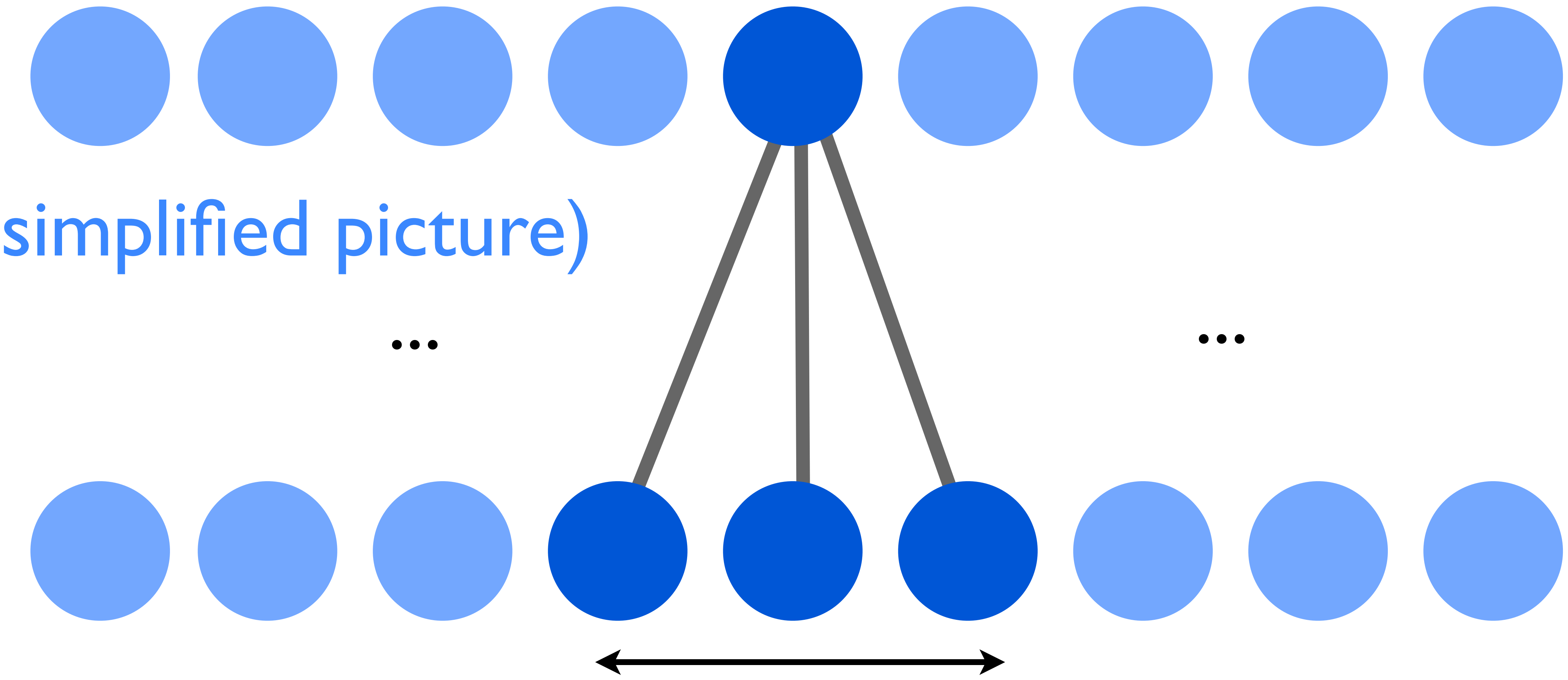
“Convolutional layer”



filter (kernel) size

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

“Convolutional layer”

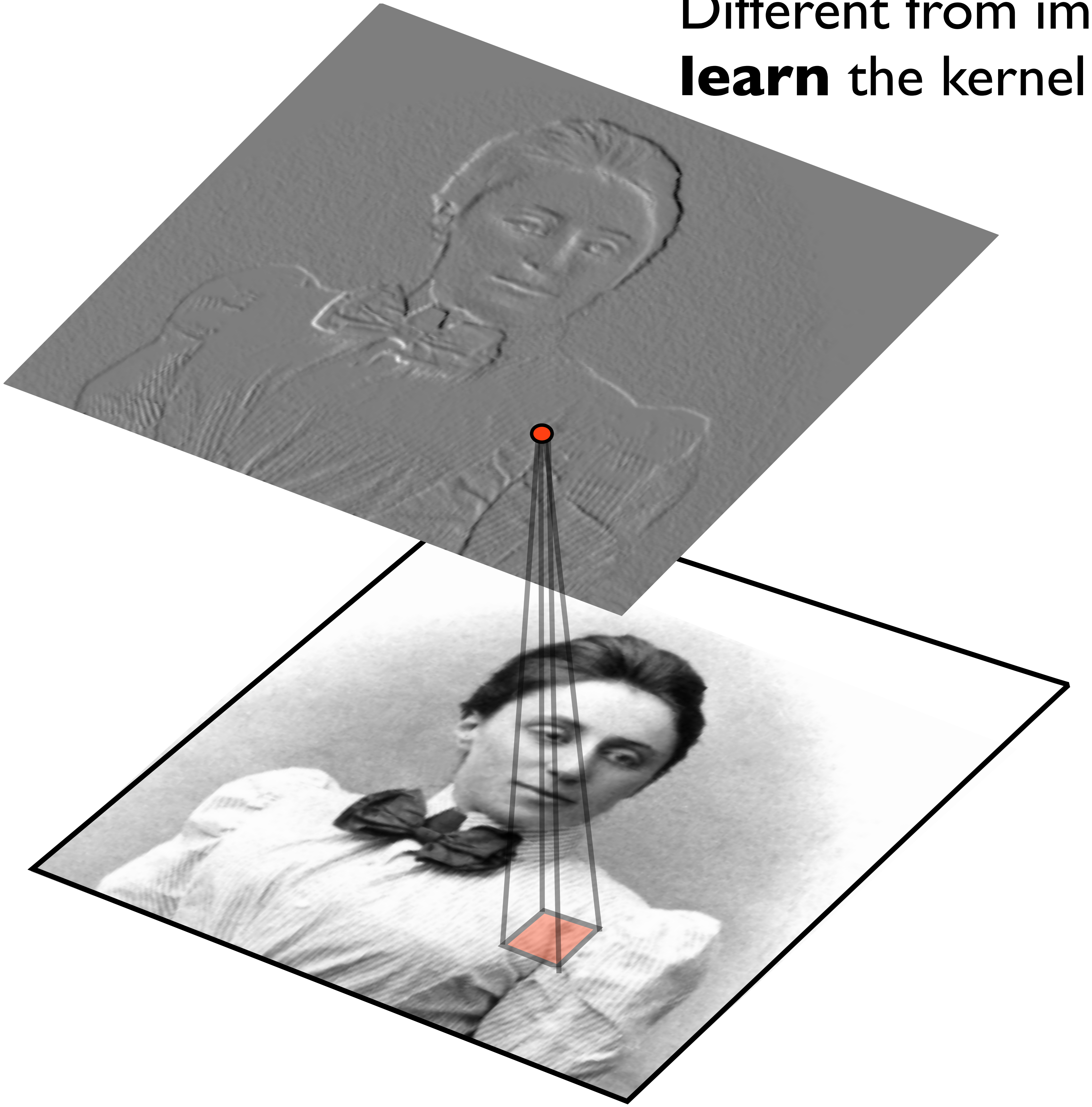


filter (kernel) size

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!

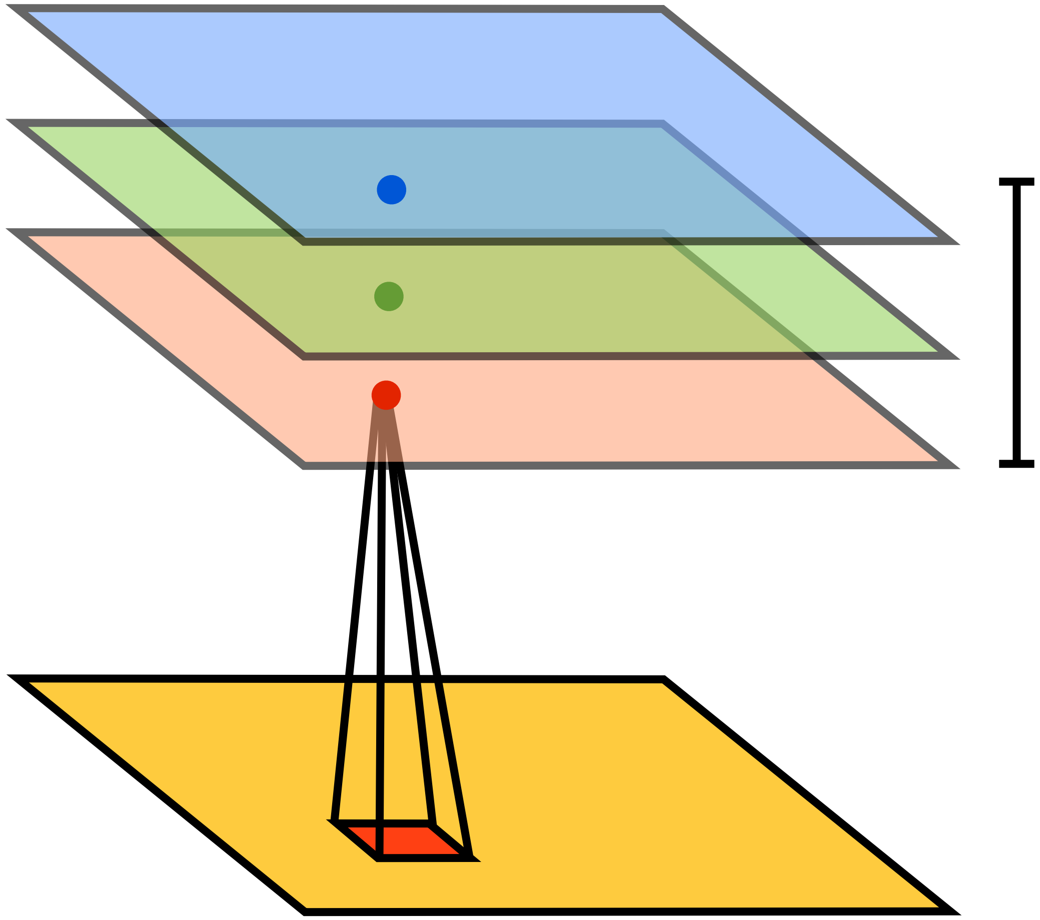
fully connected: N^2 (N=size of layer/image)
convolutional: M (M=size of kernel)

independent of the size of the image!

lower memory consumption, improved speed

Several filters (kernels)

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



several 'channels'

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'))
```

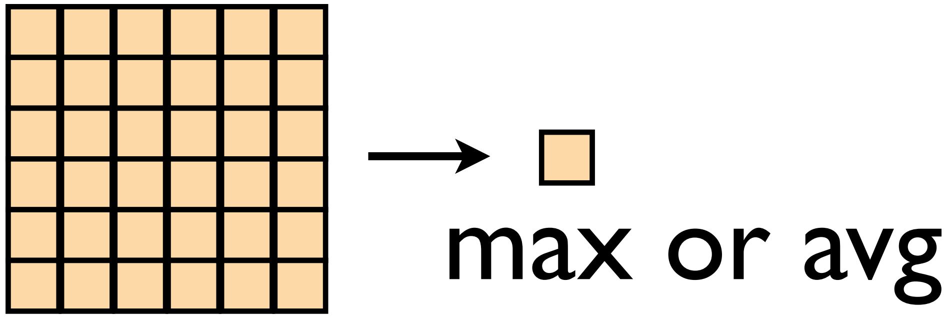
next layer will be NxNx20 (20 channels!)

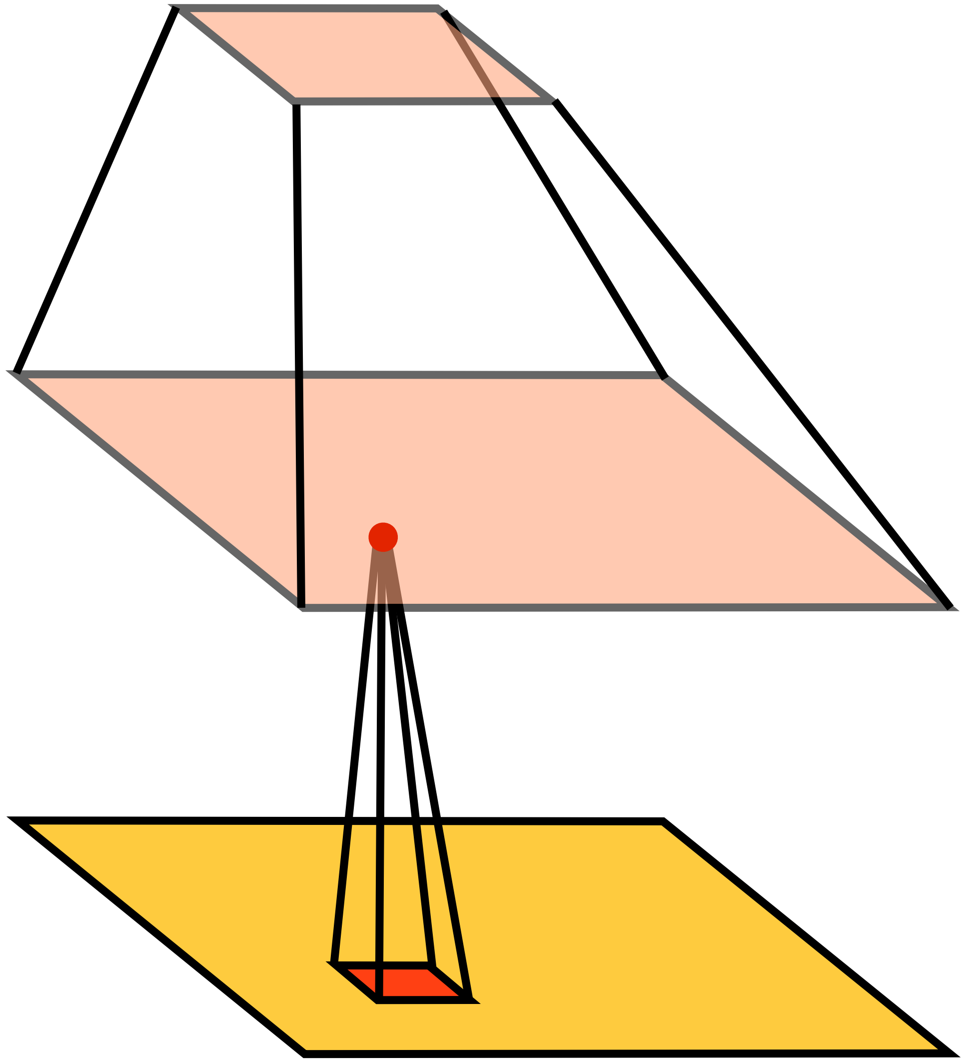
kernel size (region)

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

Reducing the resolution

“max pooling”
“average pooling”

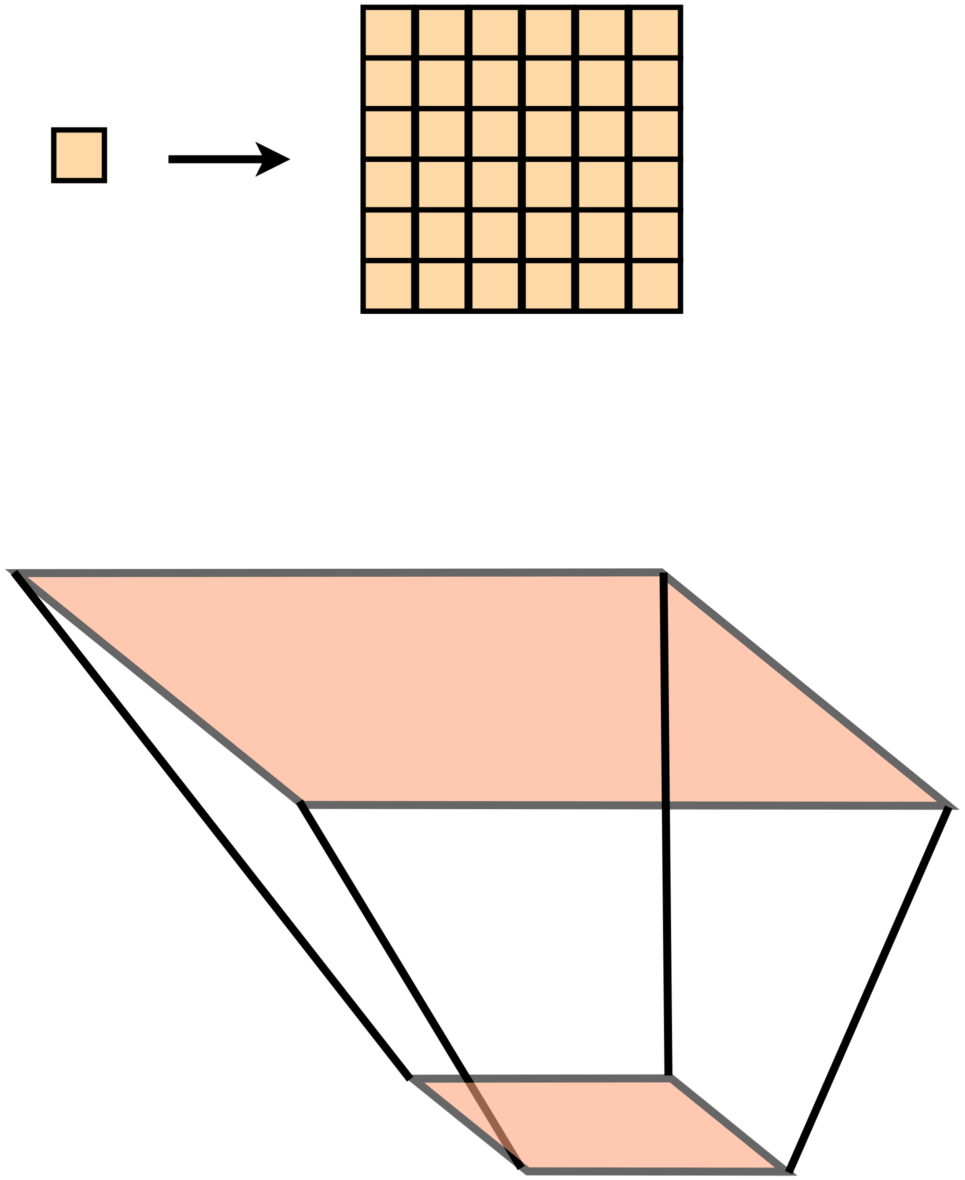




in keras:

```
net.add(AveragePooling2D(pool_size=8))
```

Enlarging the image size (again)

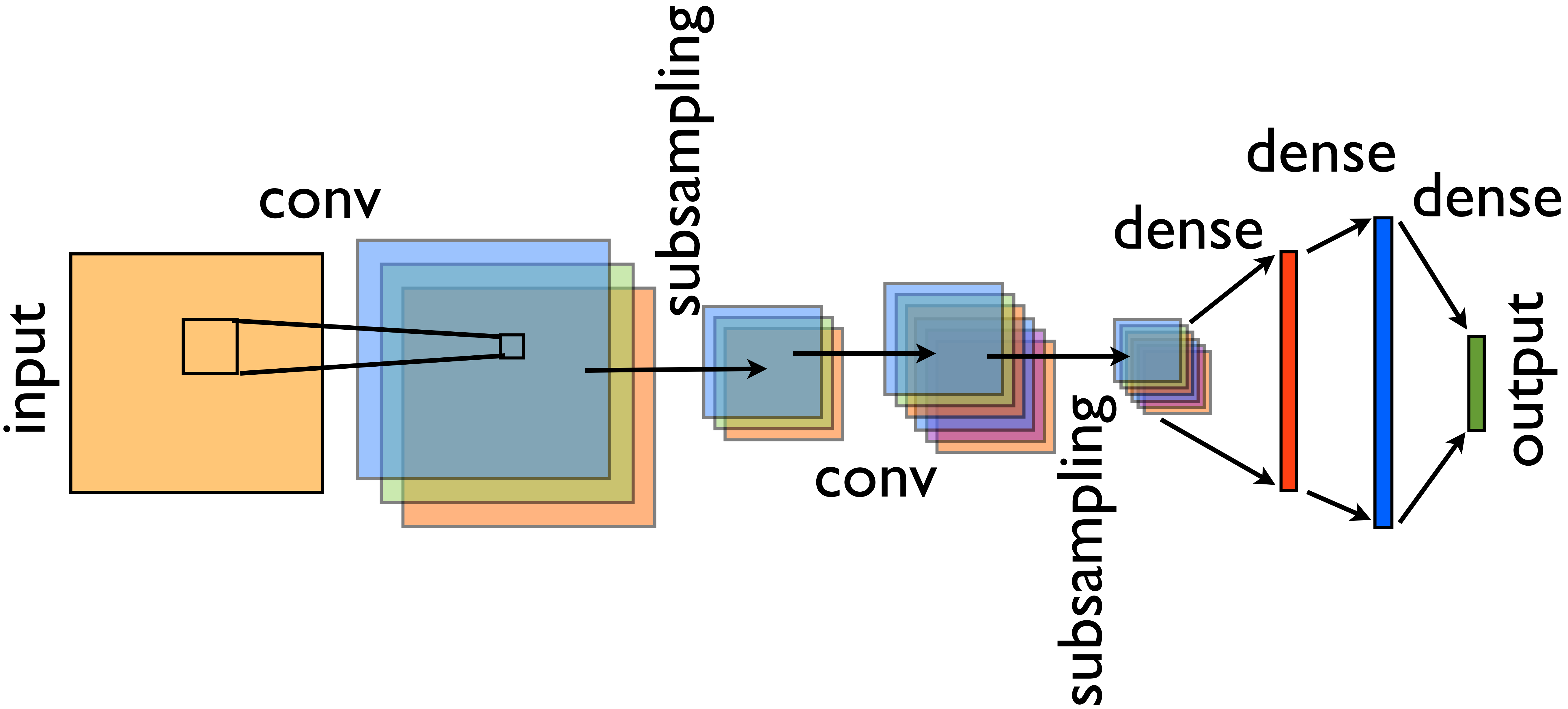


in keras:

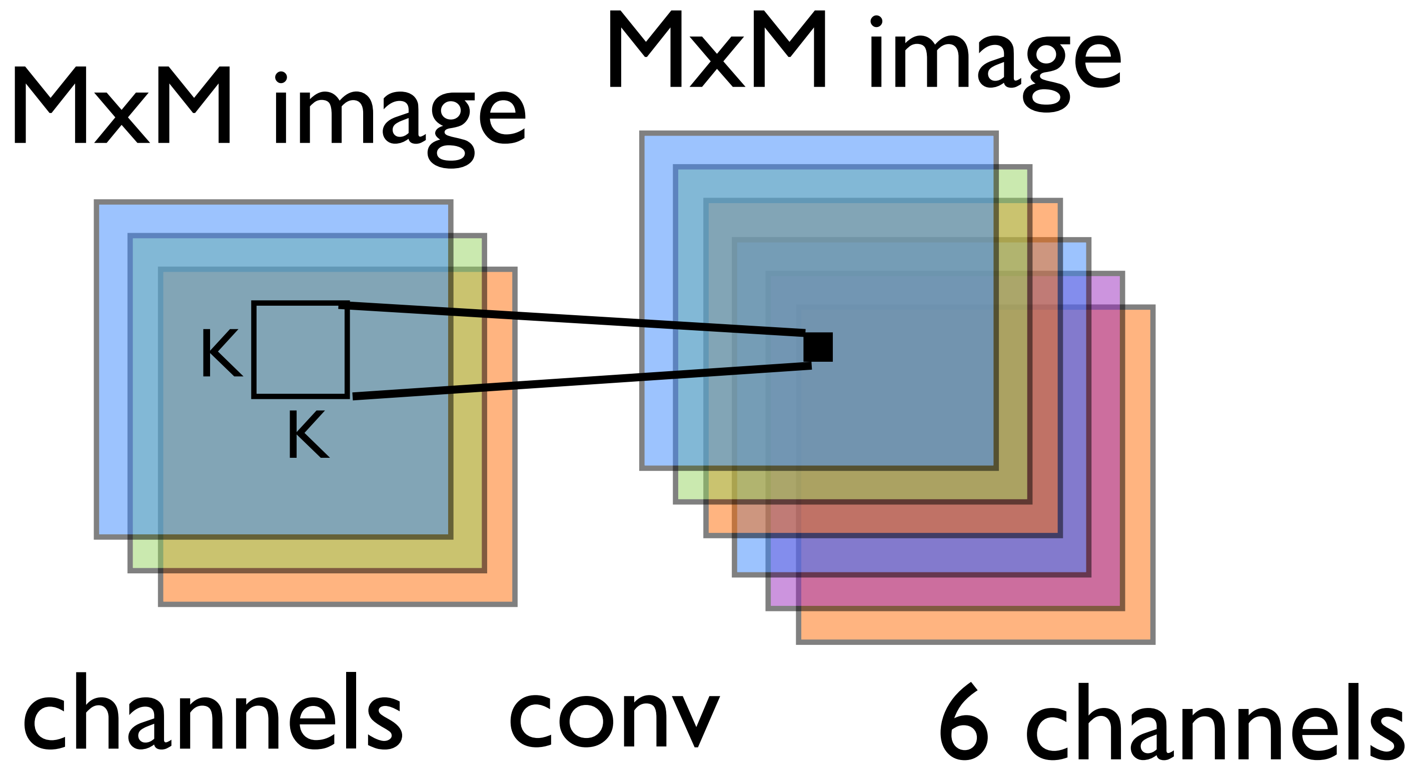
```
net.add(UpSampling2D(size=8))
```

(simply repeats values)

A fully developed convolutional net



"Channels"



MxM image

MxM image

3 channels

conv

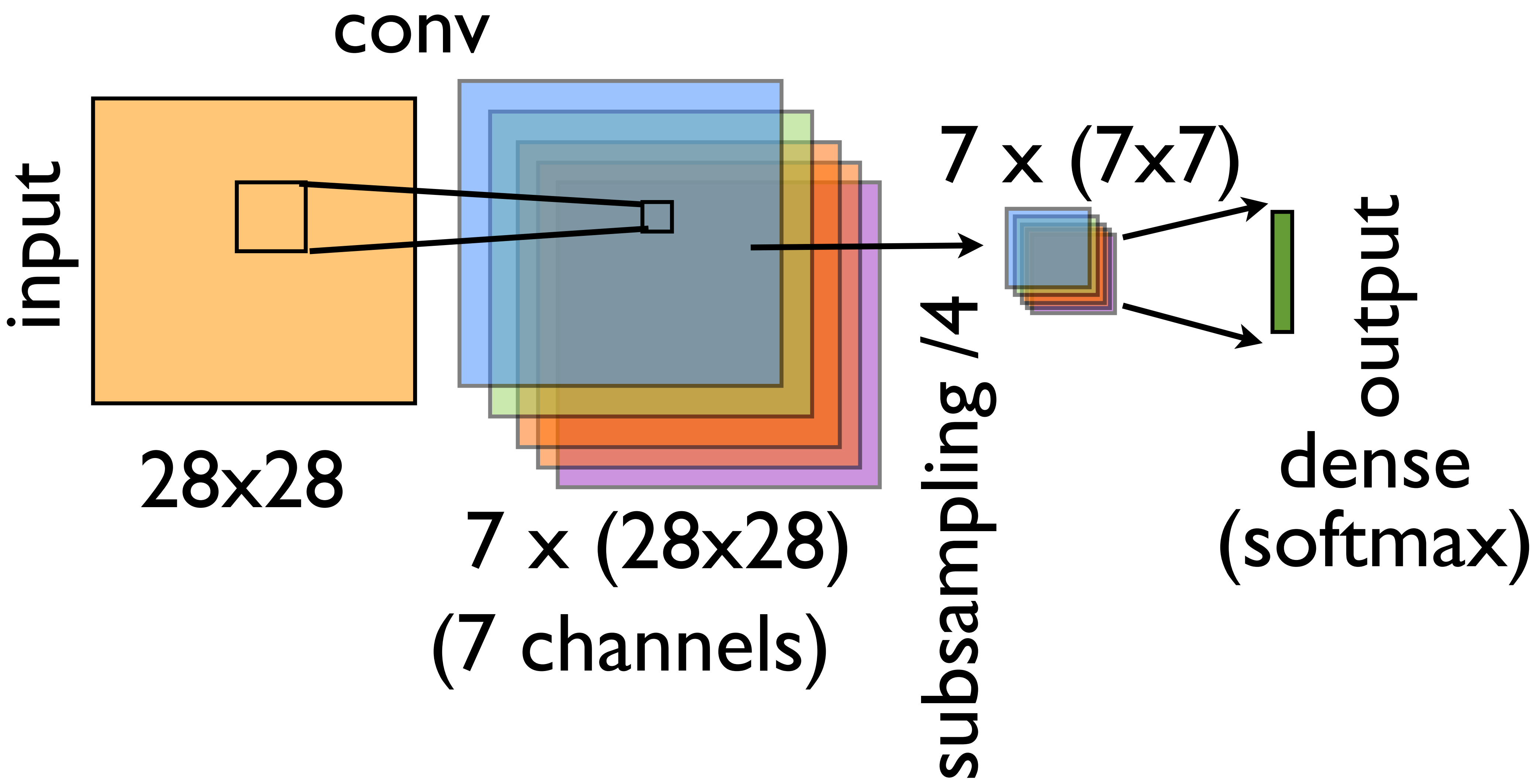
6 channels

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!)

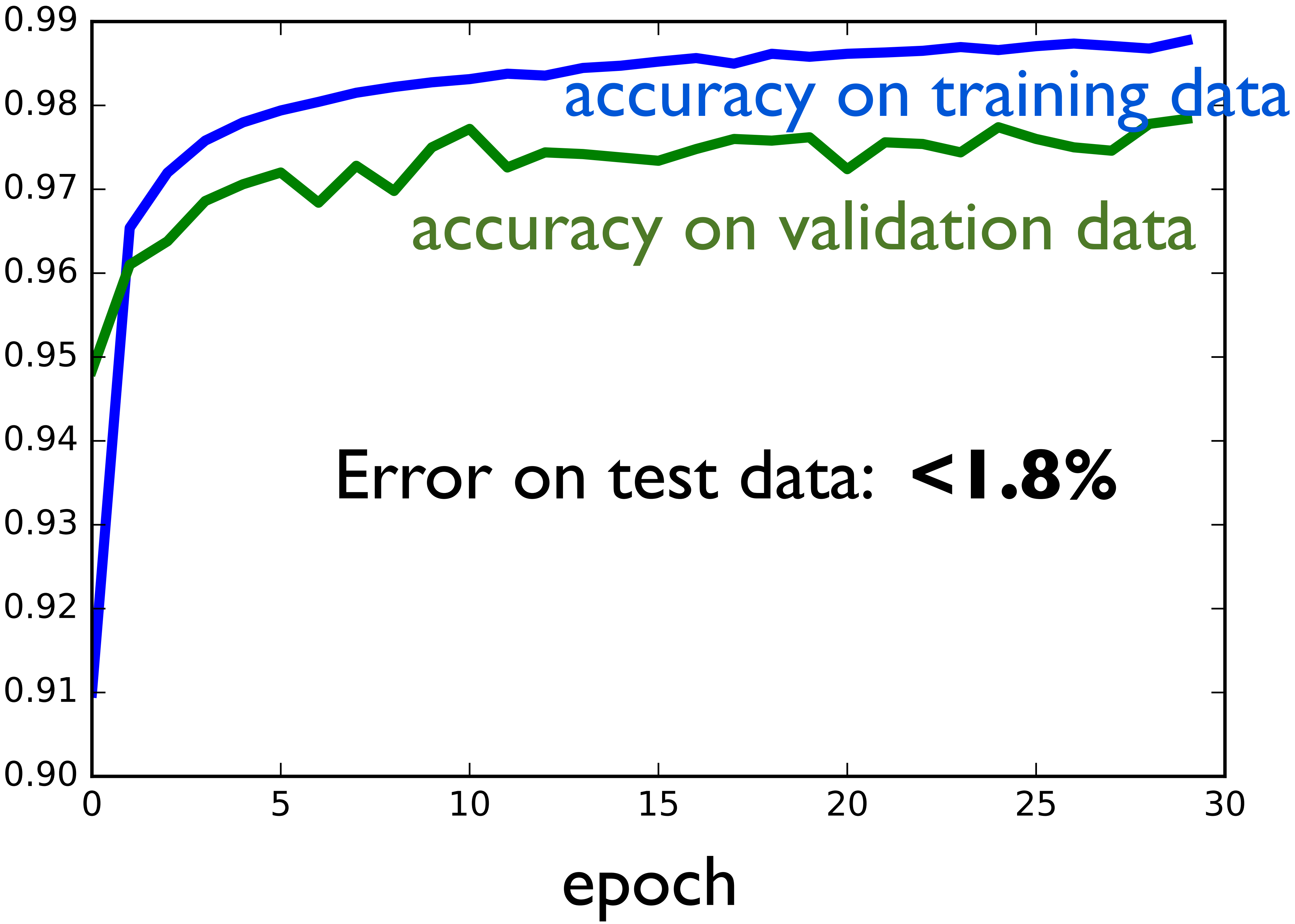
Note: keras automatically takes care of all of this, need only specify number of channels

Handwritten digits recognition with a convolutional net

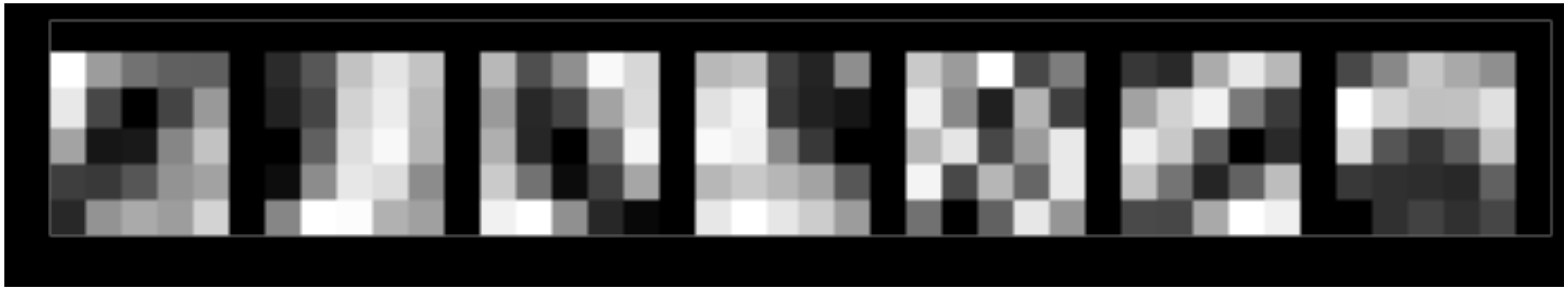


```
# initialize the convolutional network
def init_net_conv_simple():
    global net, M
    net = Sequential()
    net.add(Conv2D(input_shape=(M,M,1), filters=7,
kernel_size=[5,5],activation='relu',padding='same'))
    net.add(AveragePooling2D(pool_size=4))
    net.add(Flatten()) ← needed for transition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

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
epoch
accuracy on training data
accuracy on validation data
Error on test data: **<1.8%**

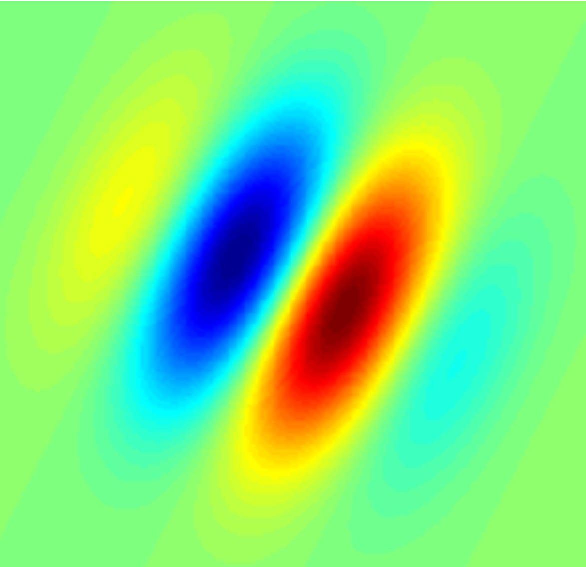
The convolutional filters



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

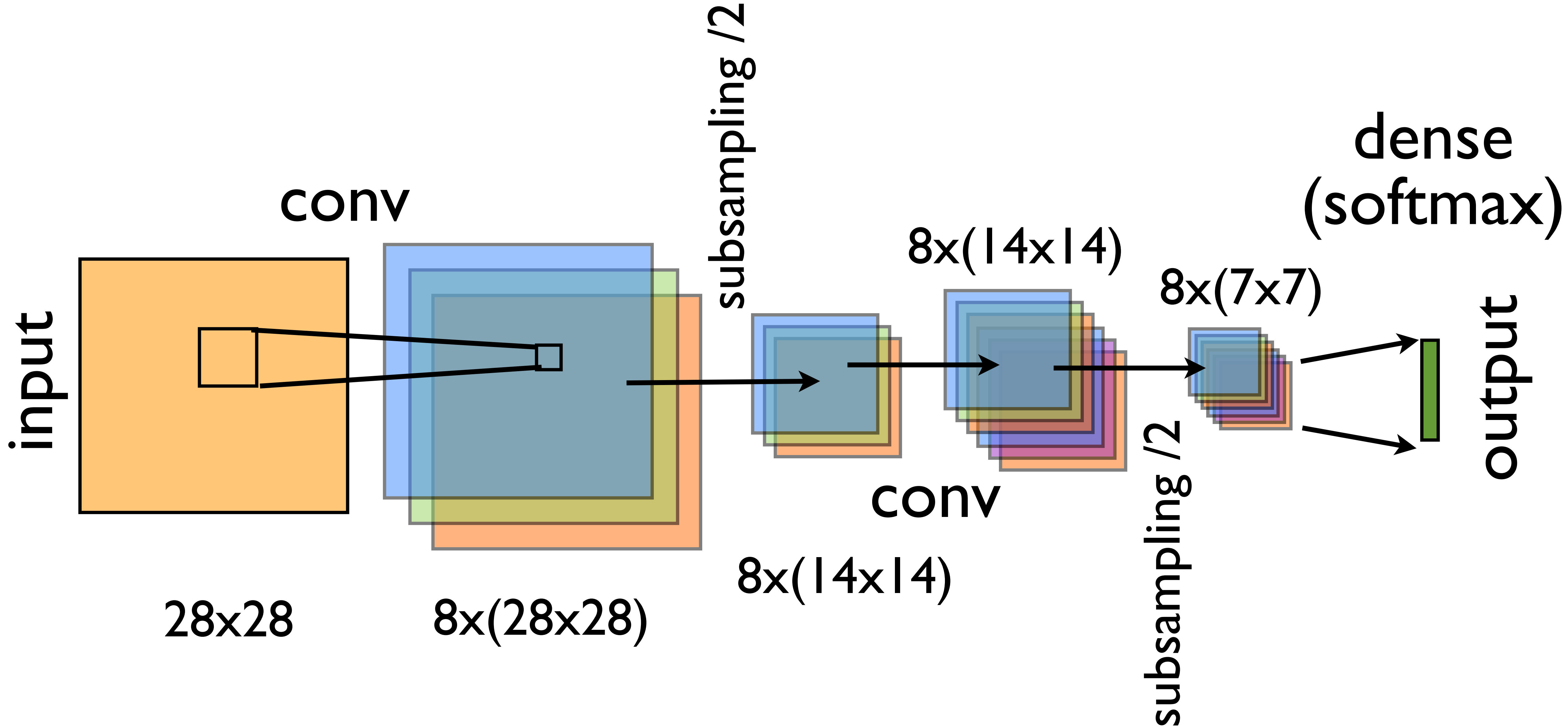
encodes orientation and spatial frequency

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

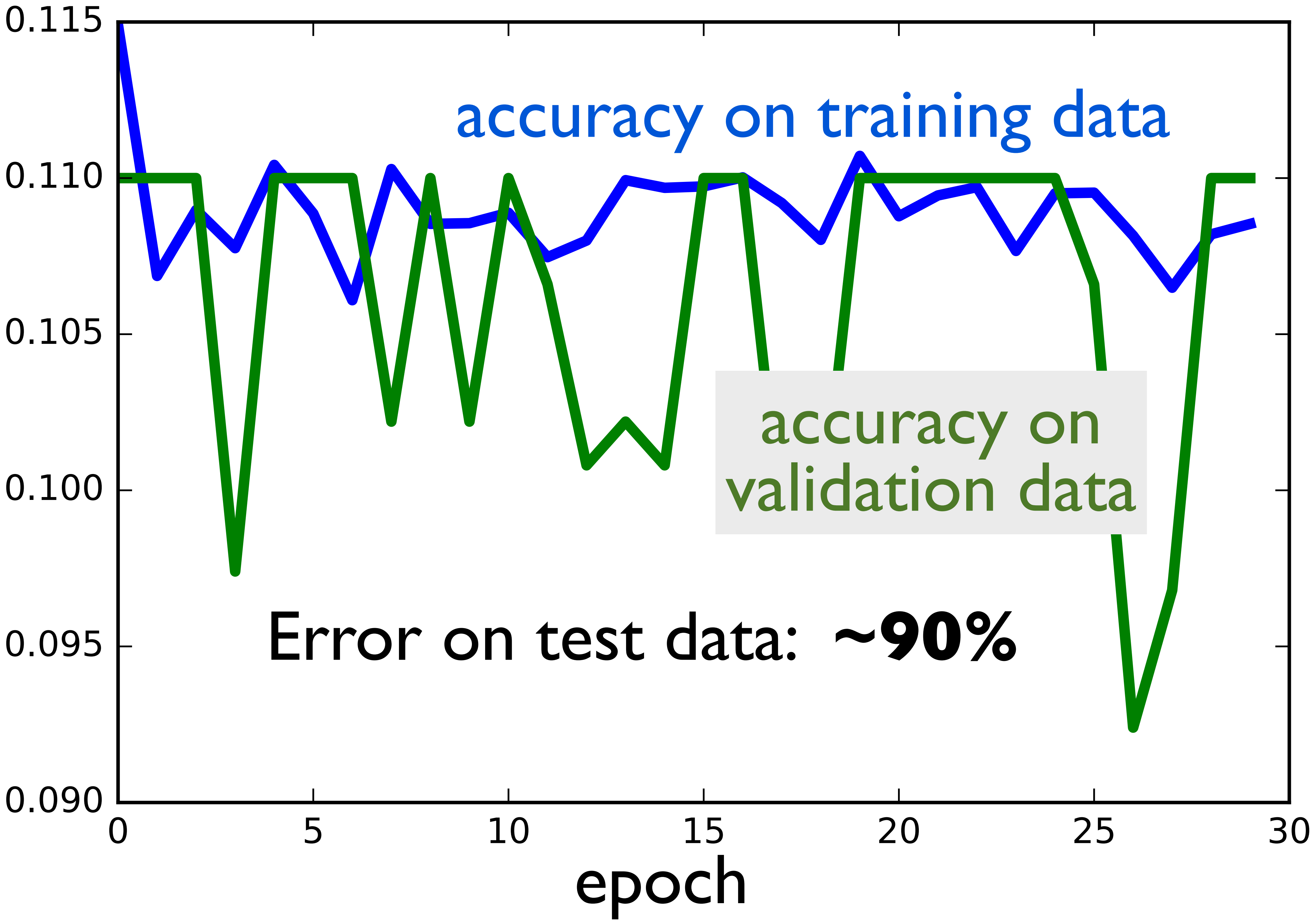
believed to be good approximation to first stage of image processing in visual cortex

Handwritten digits recognition with a convolutional net

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

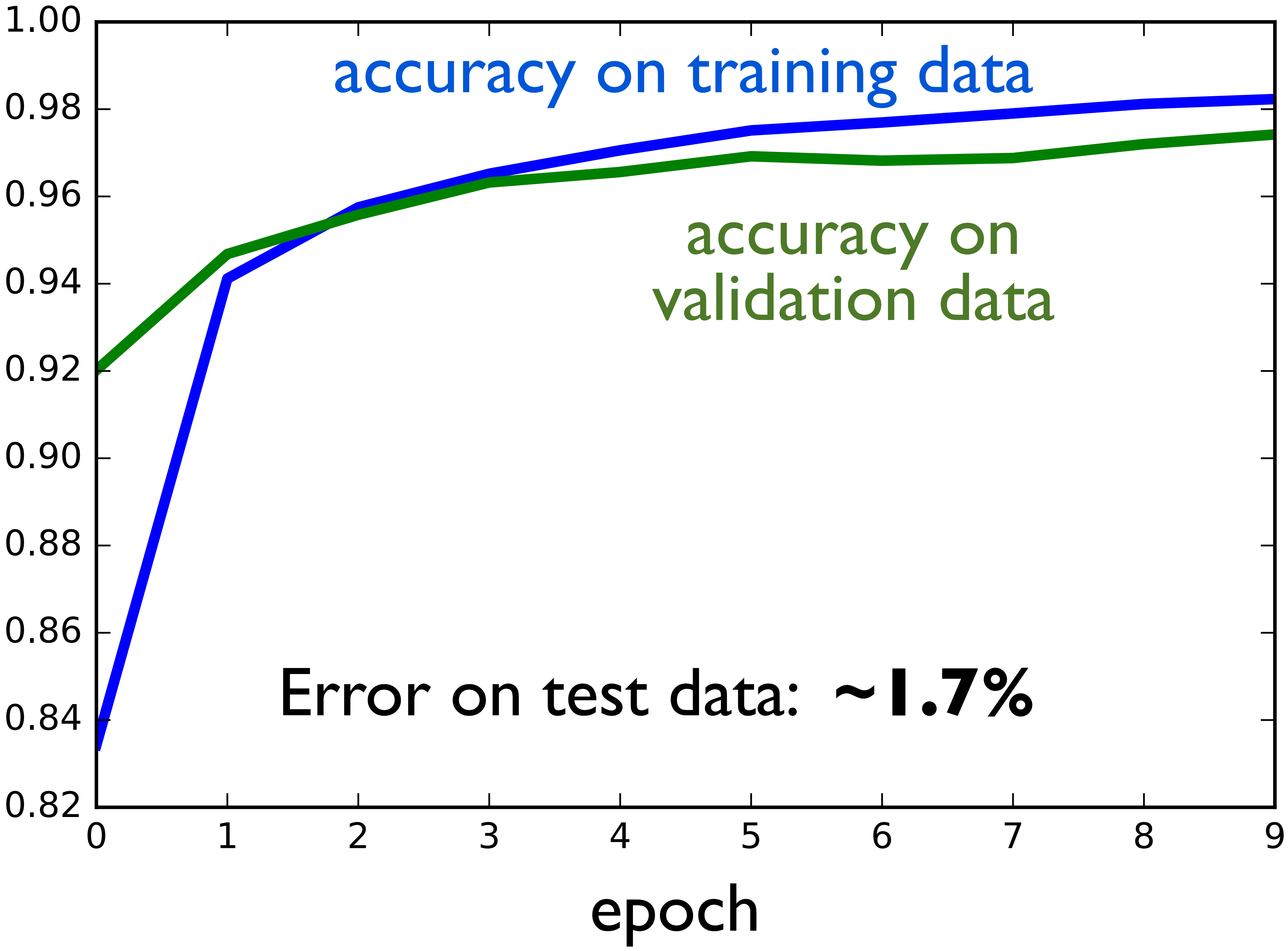


Does not learn at all! Gets 90% wrong!

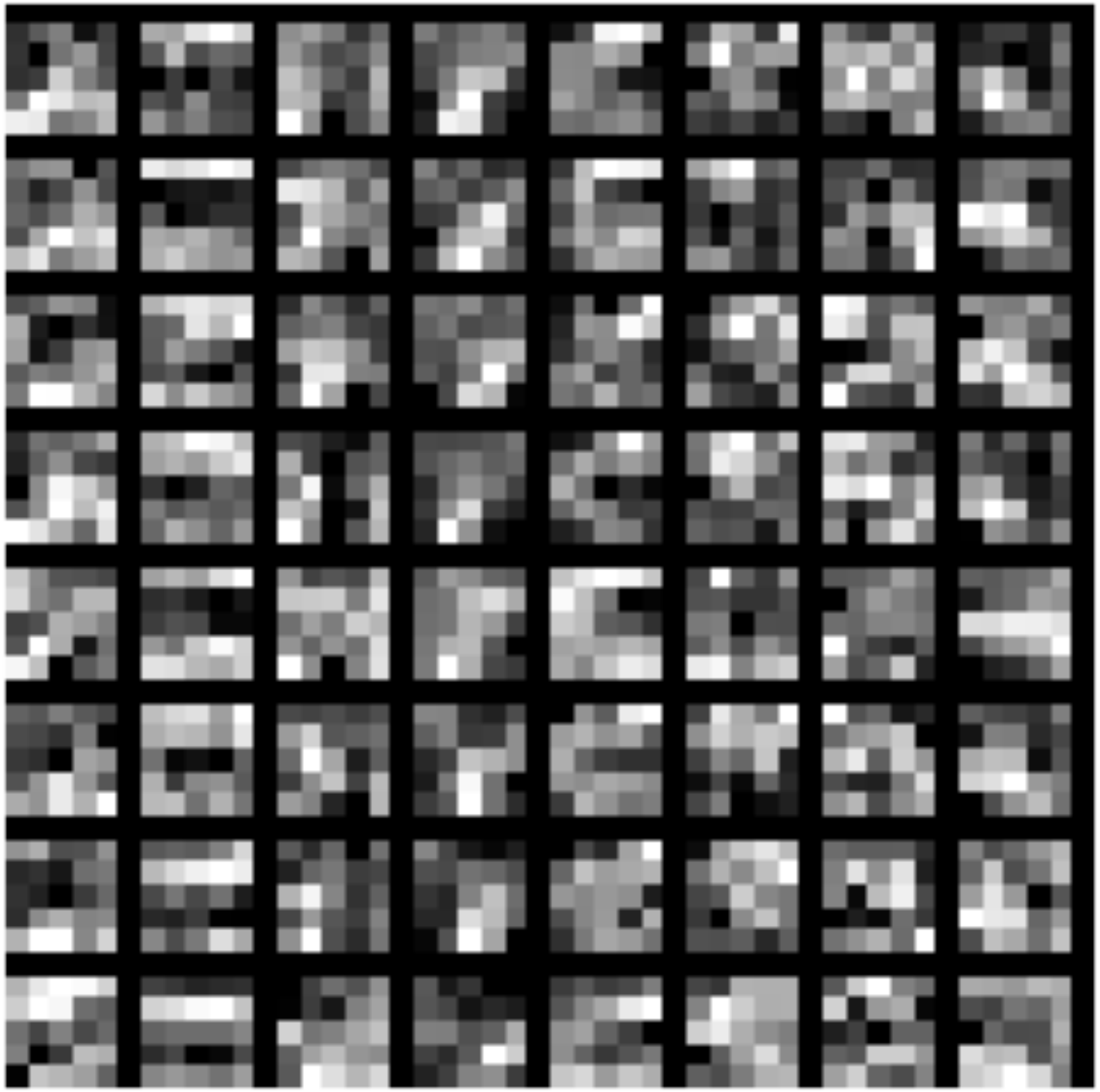


Error on test data: **~90%**

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



Error on test data: **~1.7%**



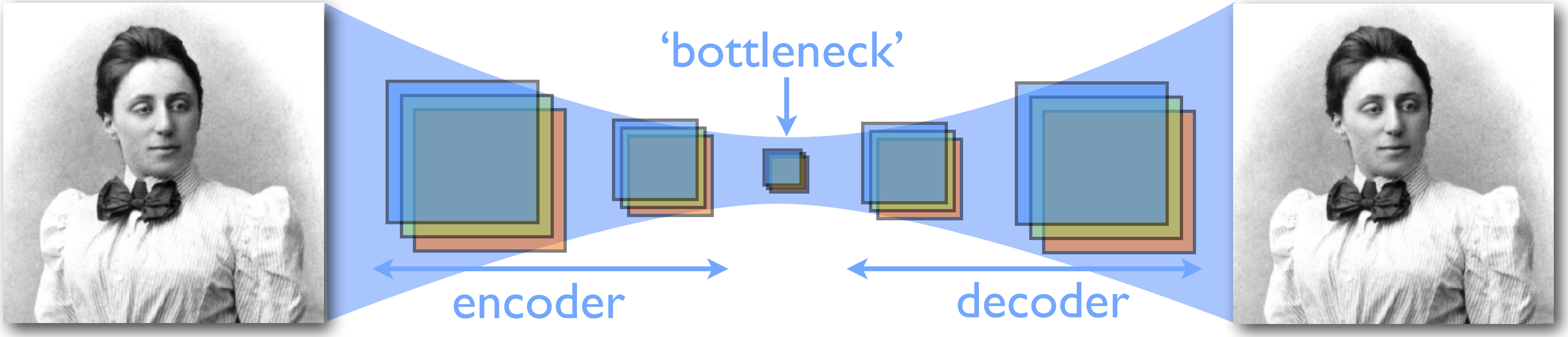
Homework

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!

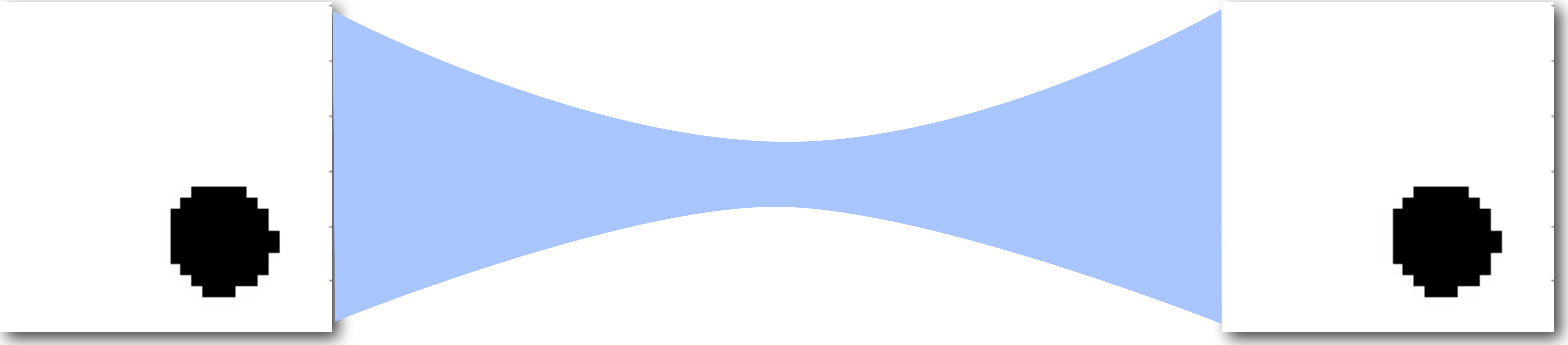
Autoencoder



- 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)

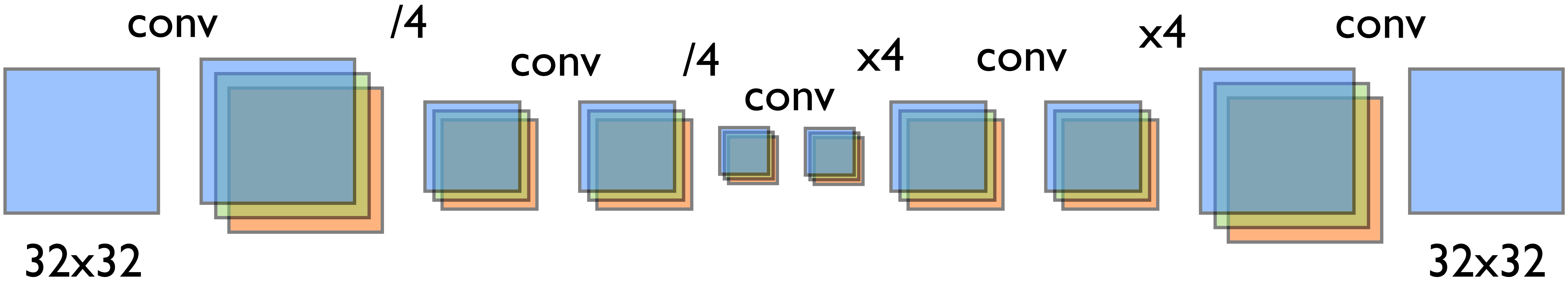
Still: need a lot of training examples

Here: generate those examples algorithmically



for example: randomly placed circle

Our convolutional autoencoder network

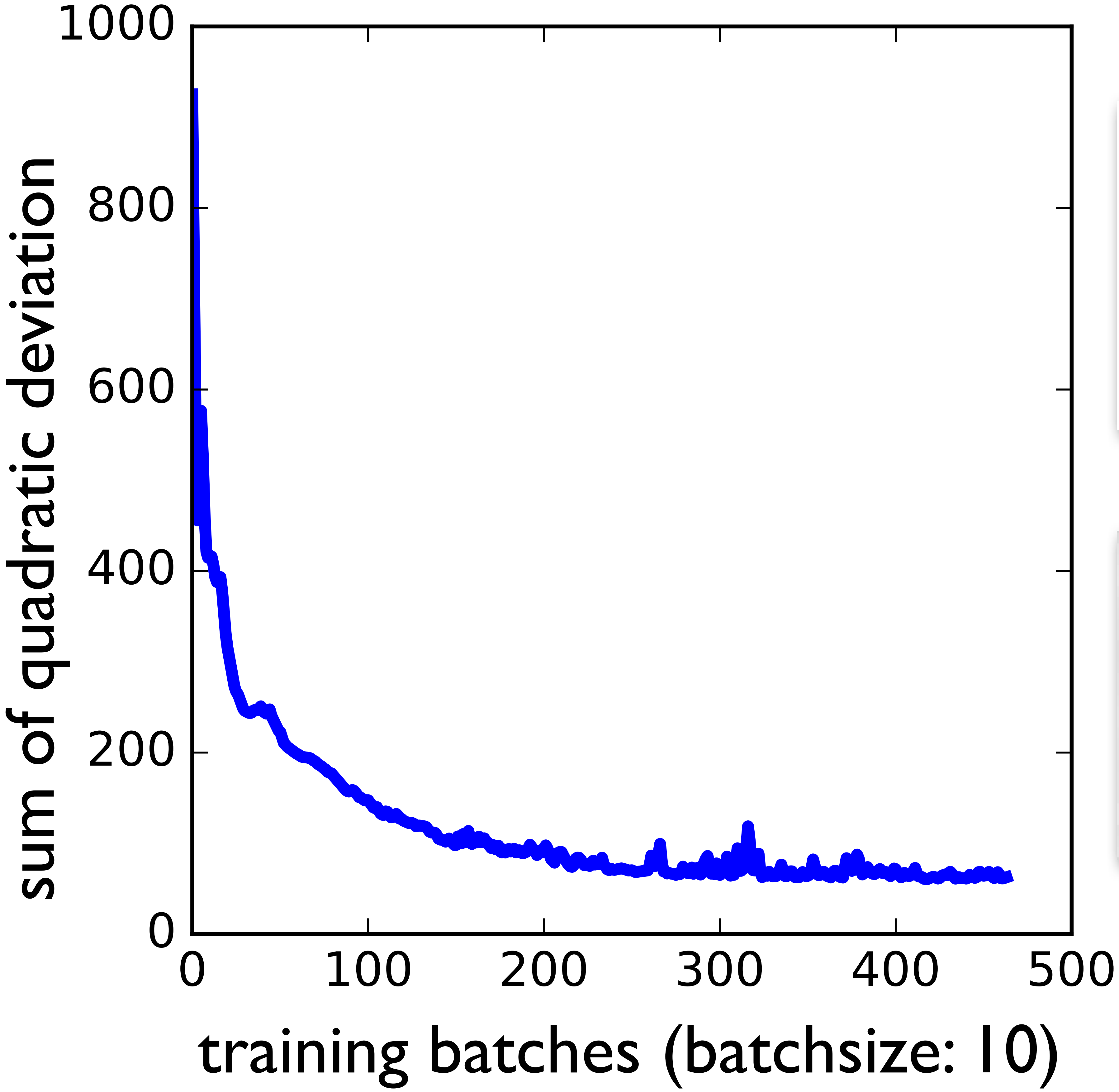


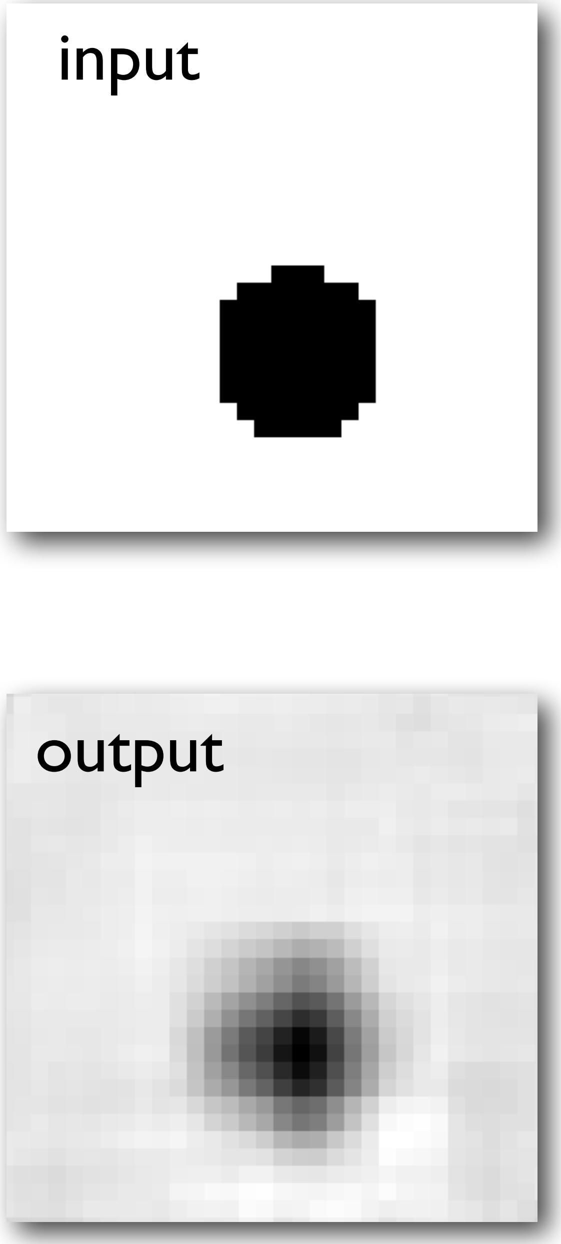
conv /4 conv /4 conv x4 conv x4 conv

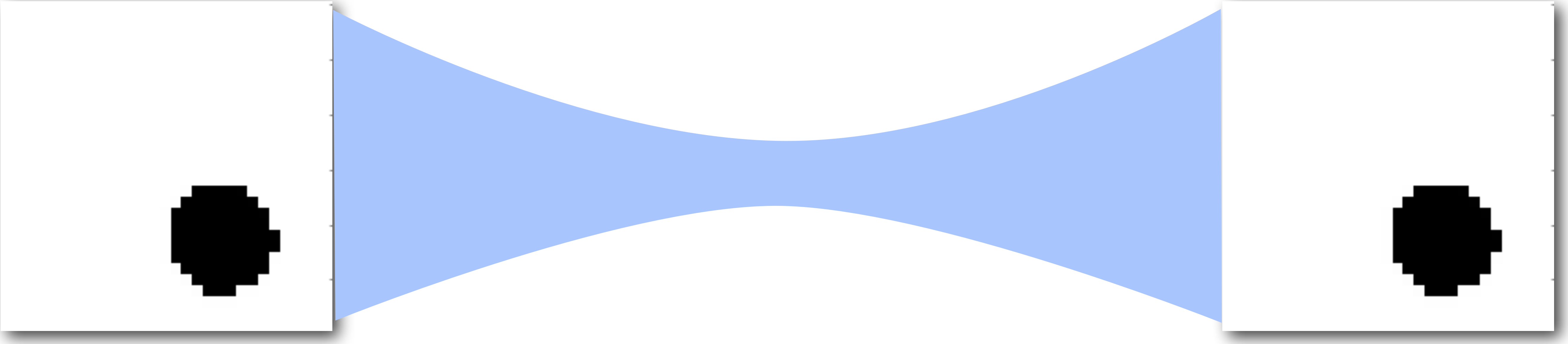
32x32 32x32

(20 channels in all intermediate steps)

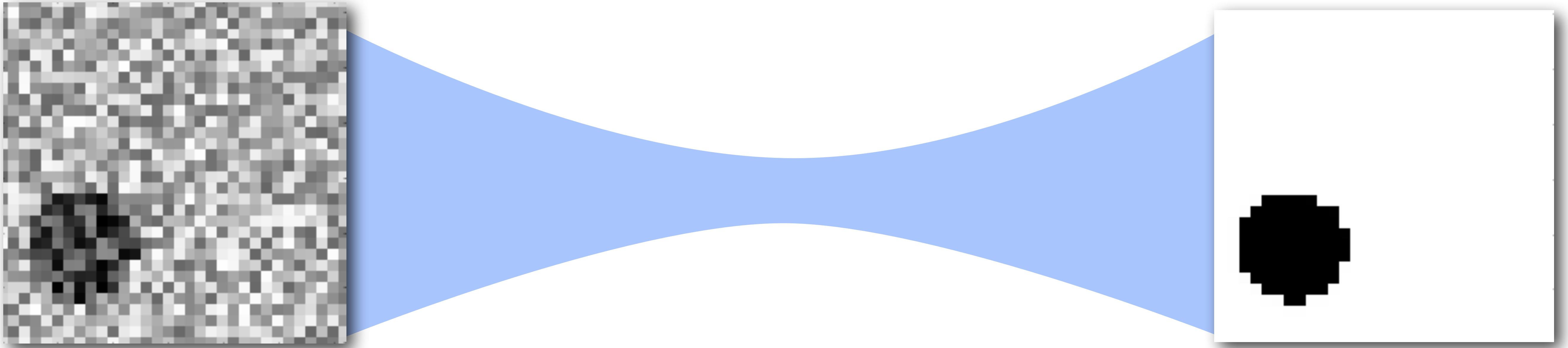
cost function for a single test image





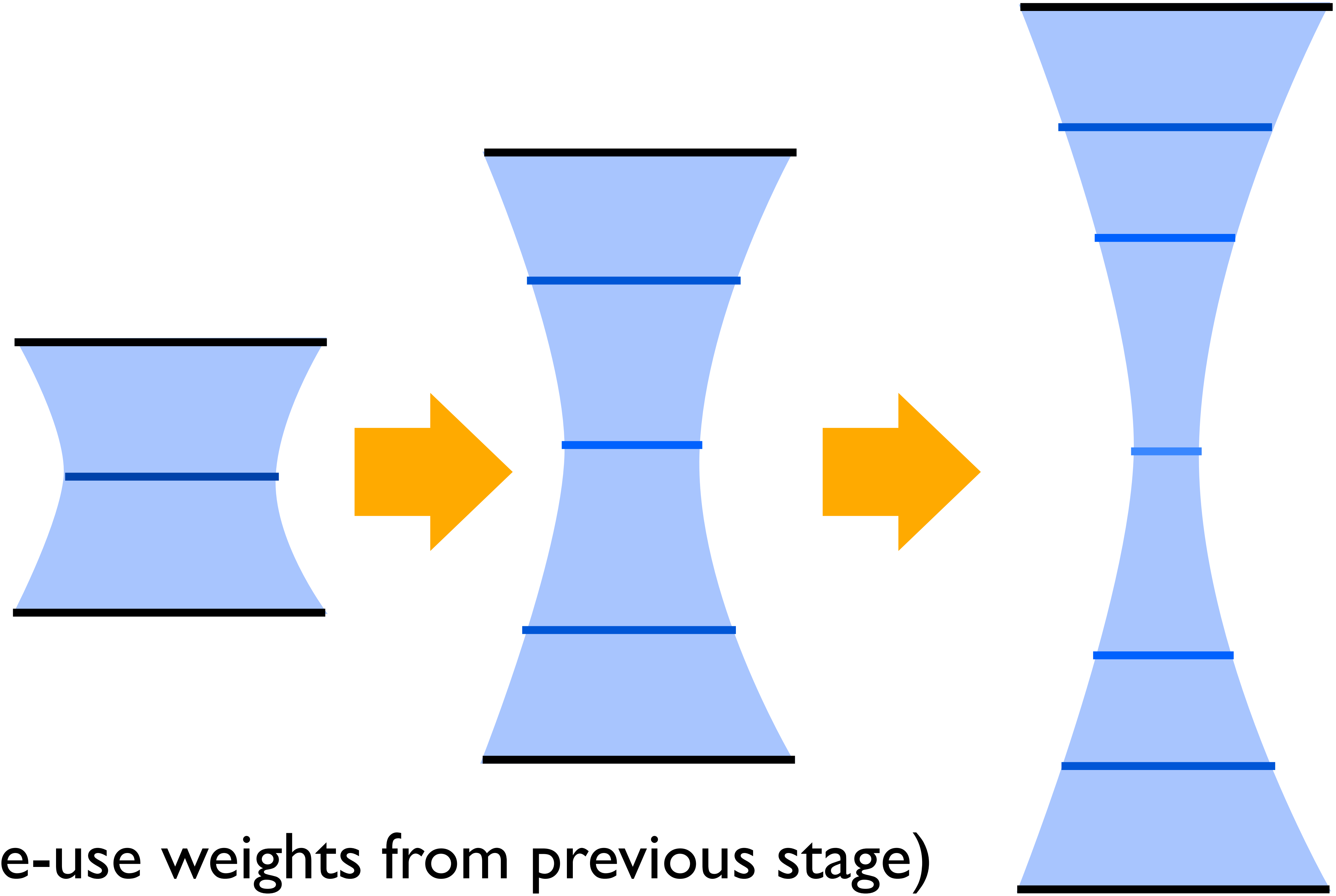


Can make it even more challenging: produce a cleaned-up version of a noisy input image!



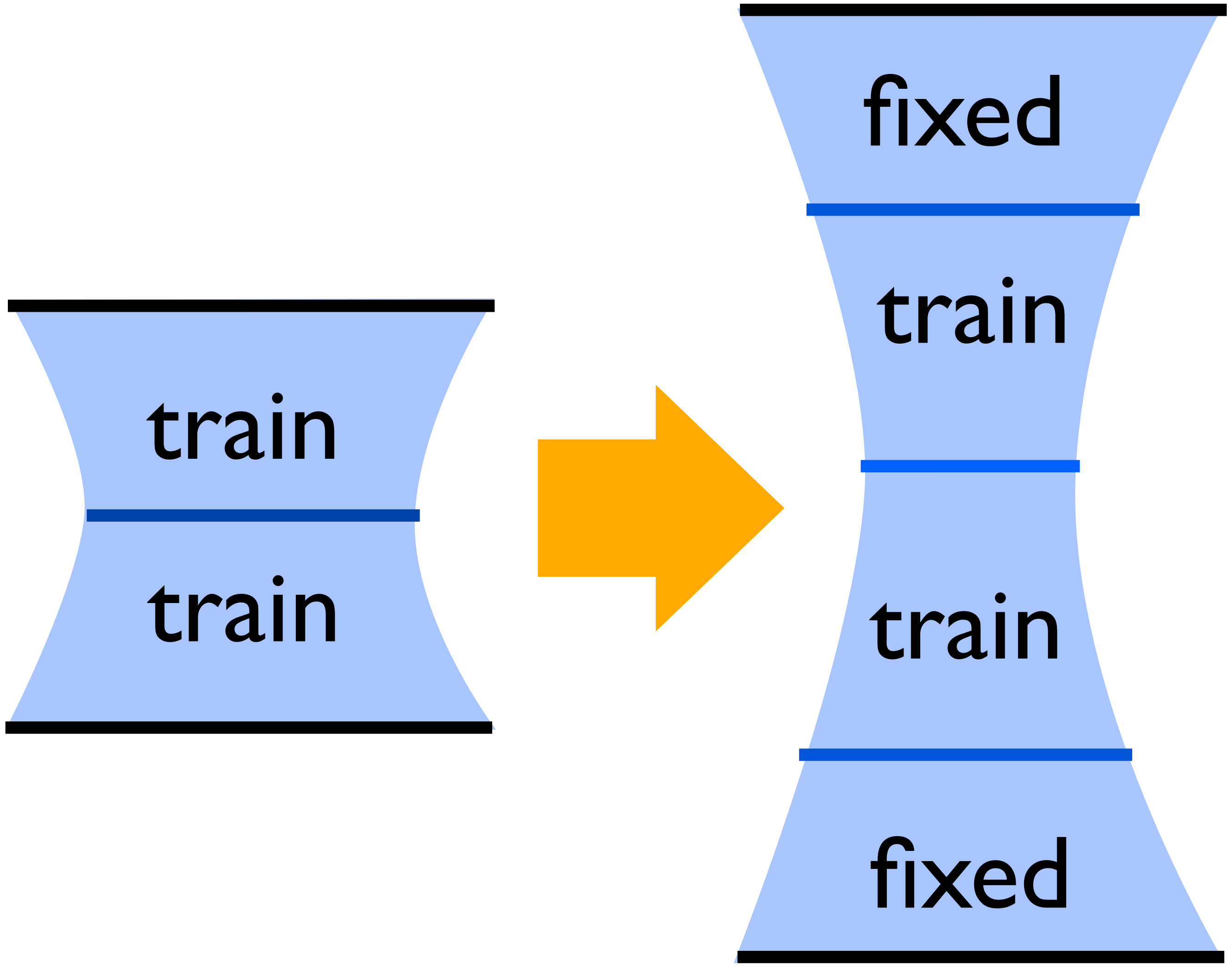
“denoising autoencoder”

Stacking autoencoders



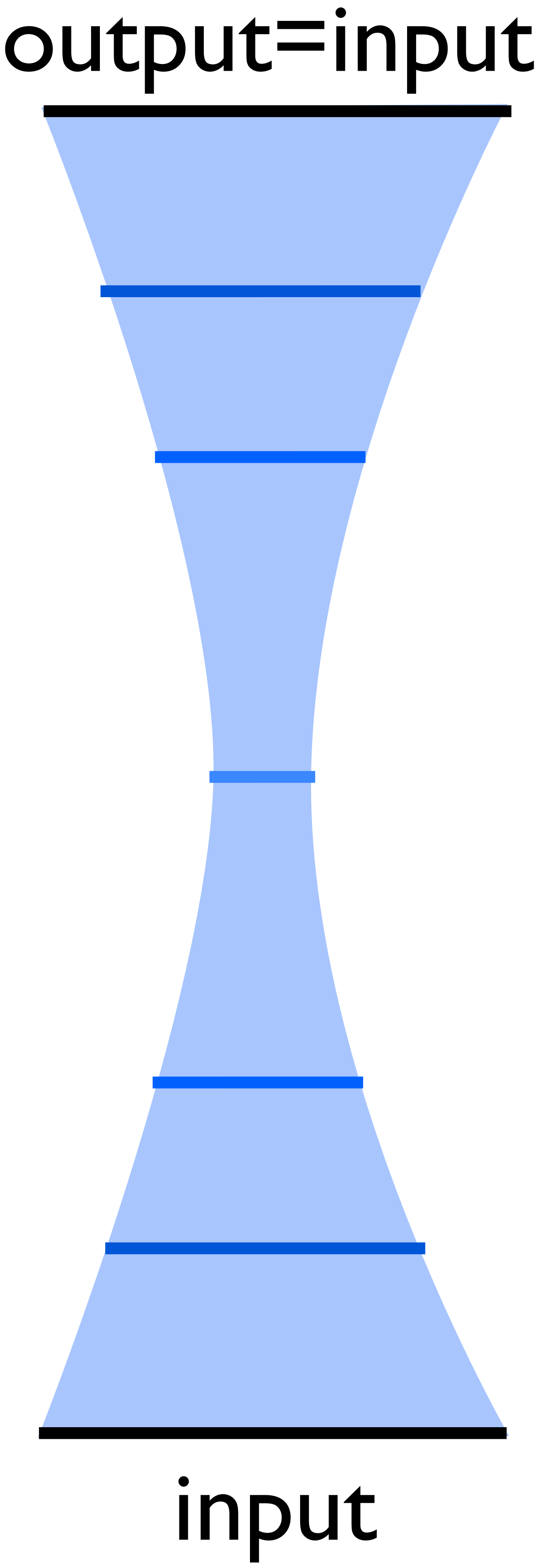
(re-use weights from previous stage)

“greedy layer-wise training”

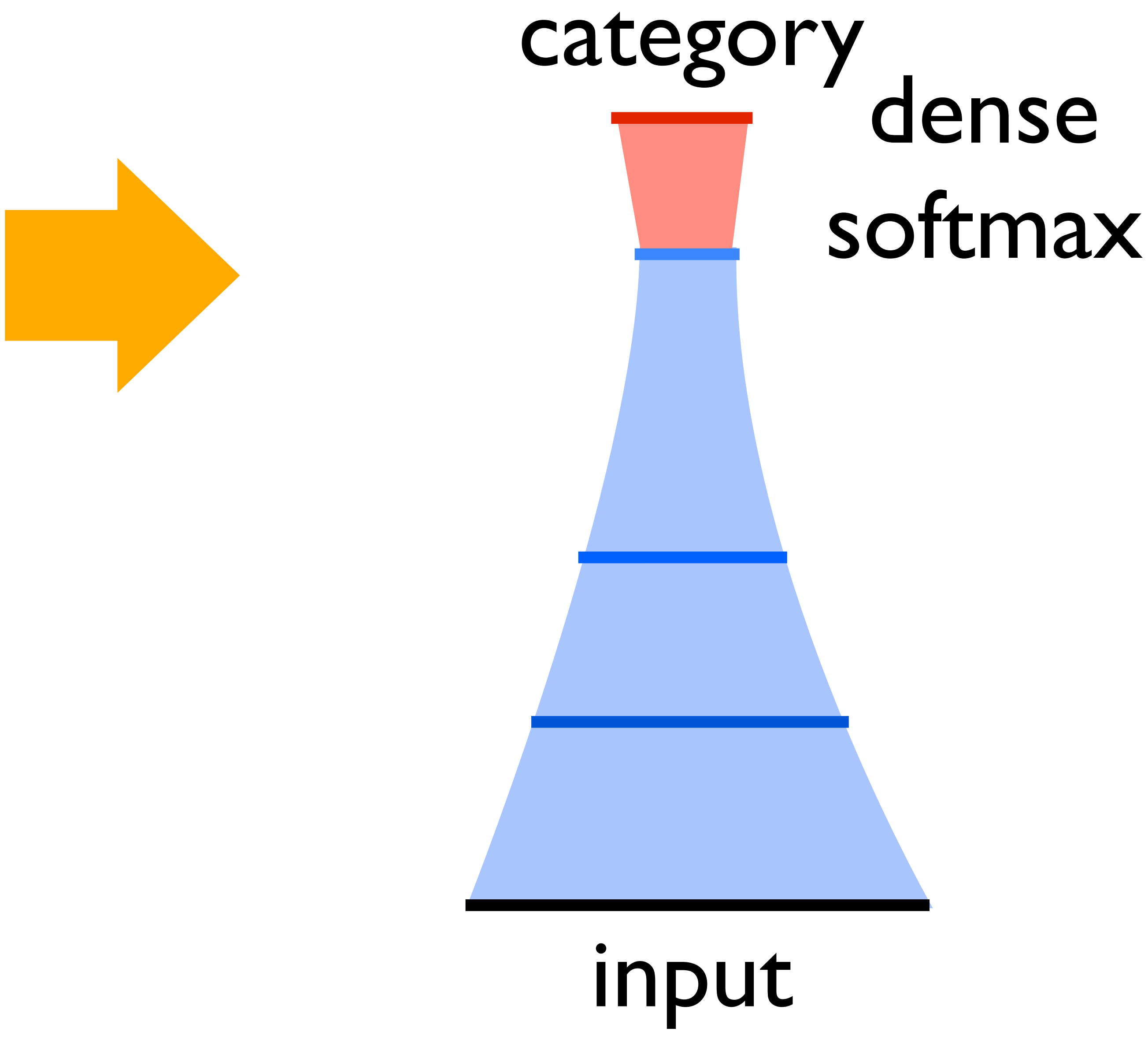


(and so on, for more and more layers)

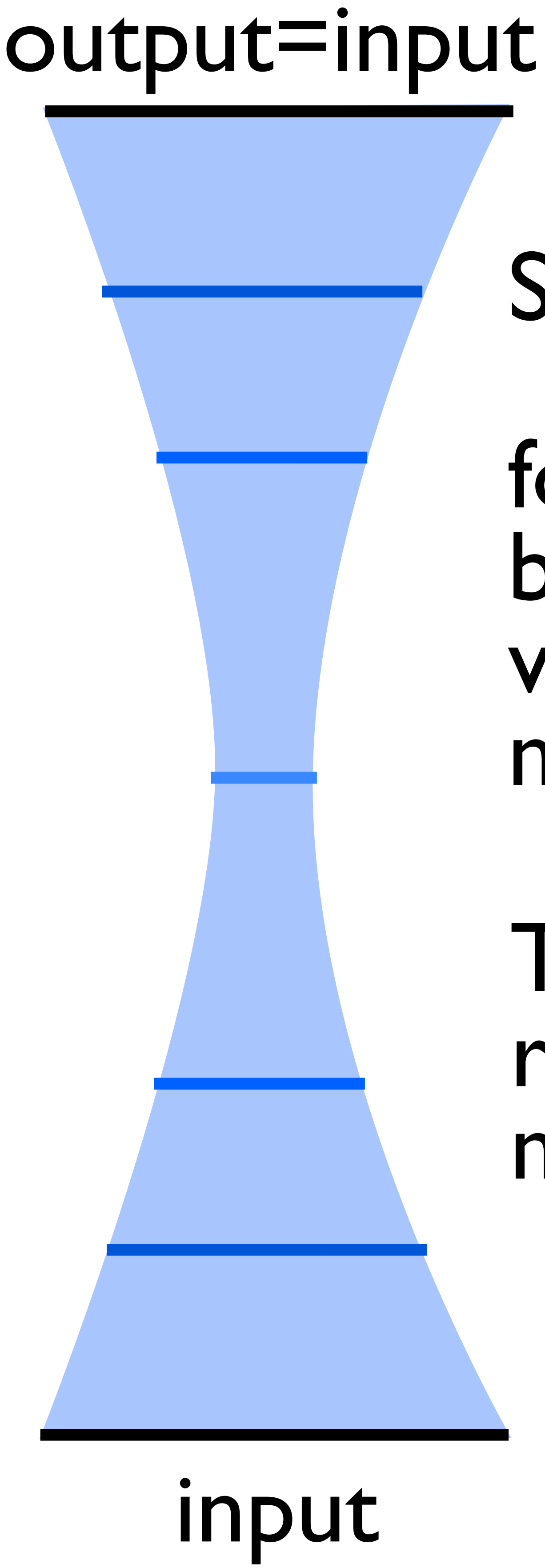
afterwards can ‘fine-tune’ weights by training all of them together, in the large multi-layer network



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?

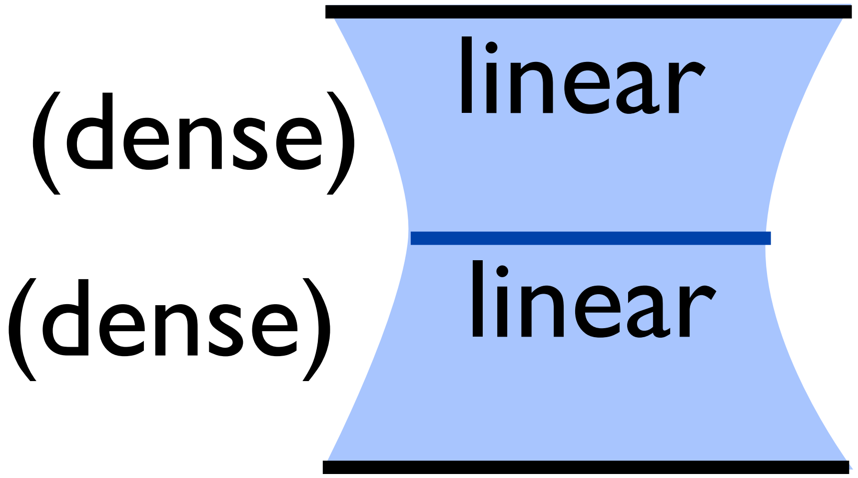
- Autoencoders are useful for pretraining, but nowadays one can train deep networks (with many layers) from scratch

- 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?

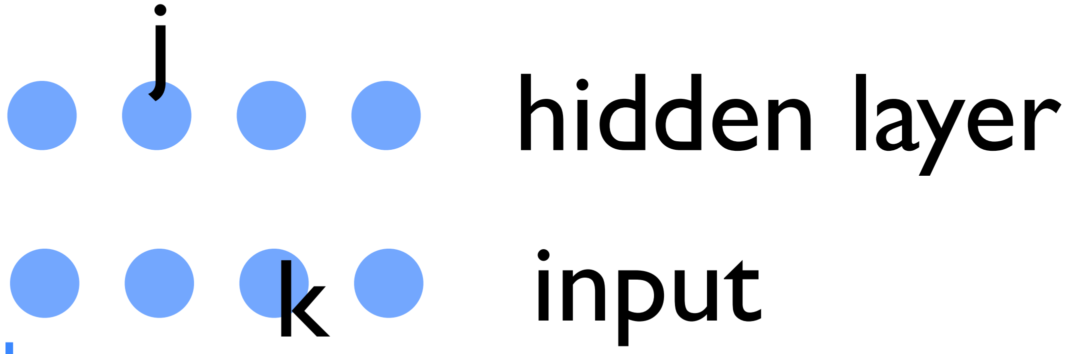


no f(z)!

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

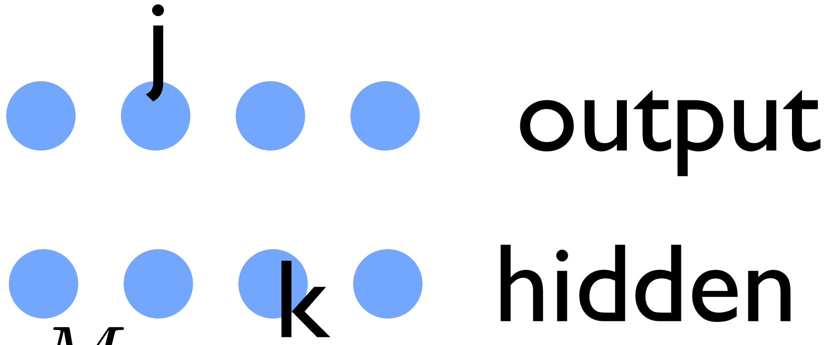
for the input-hidden weights



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 $\hat{P} = \sum_{j=1}^{M} |v_j\rangle \langle v_j|$

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]

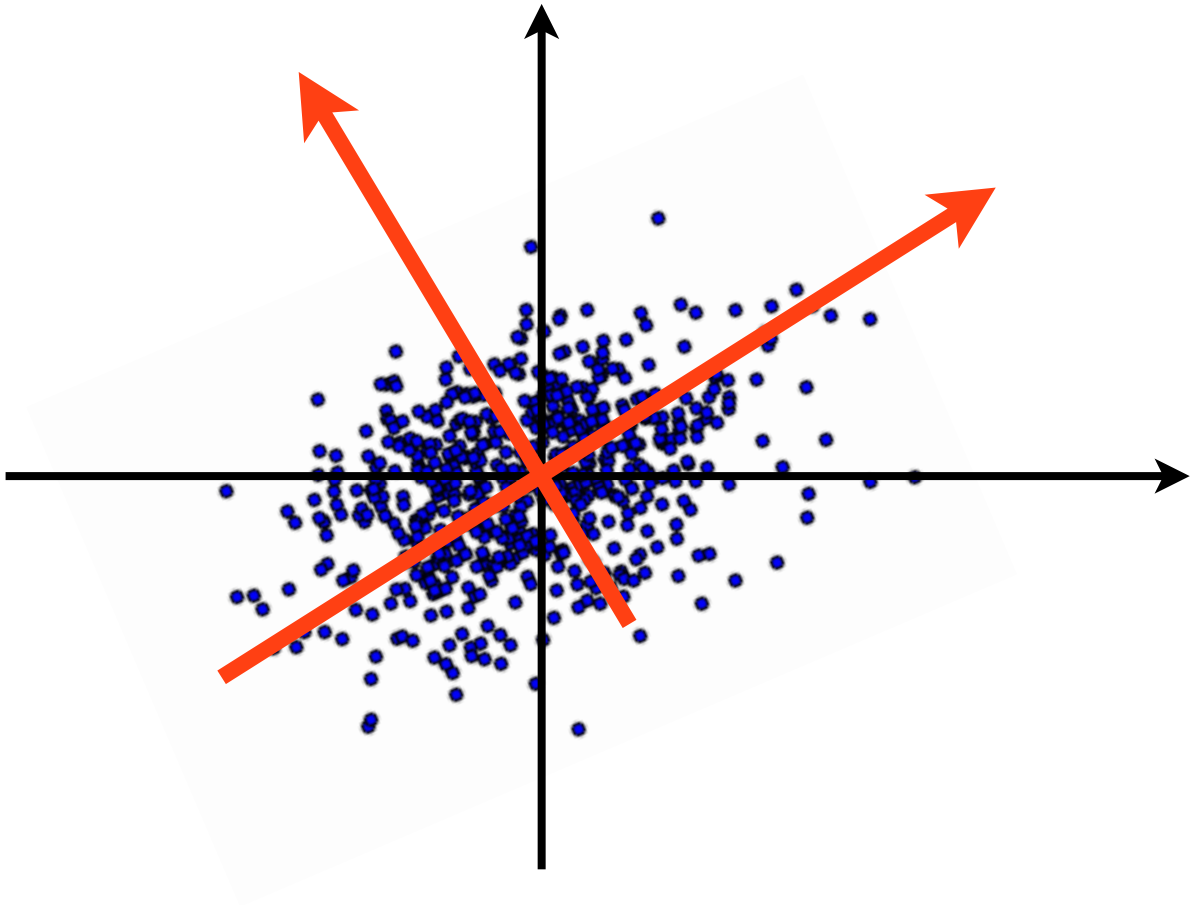
[compare density matrix in quantum physics!]

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)
(50000, 784)
```

the MNIST images

subtract average

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

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

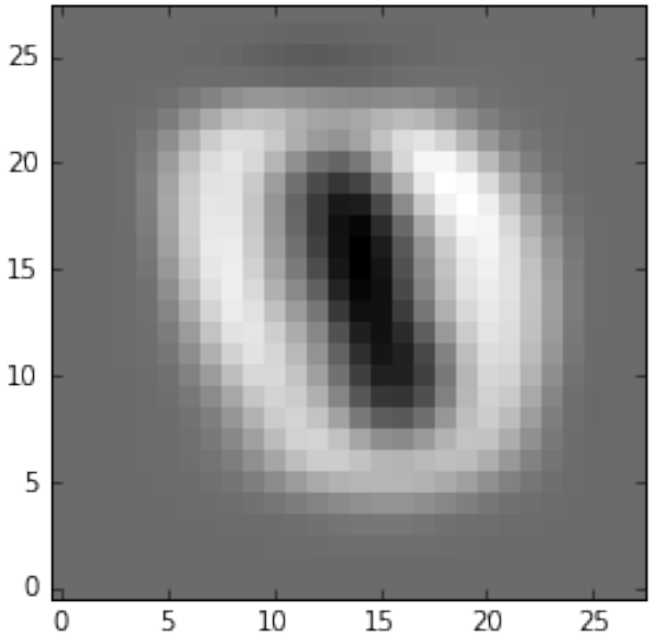
rho will be 784x784 matrix

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

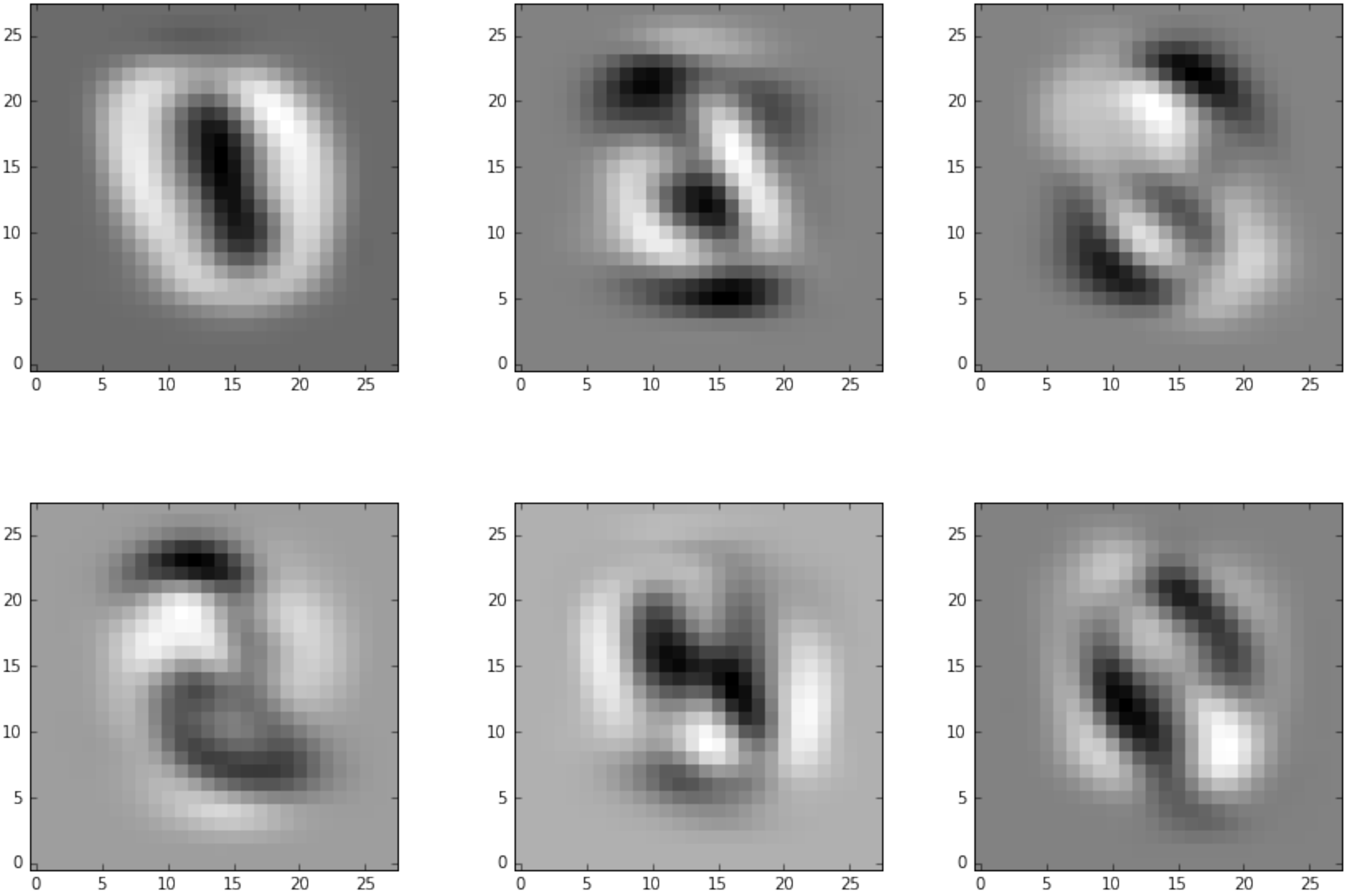
get eigenvalues- and vectors (already sorted, largest first)

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

display the 28x28 image belonging to the largest eigenvector

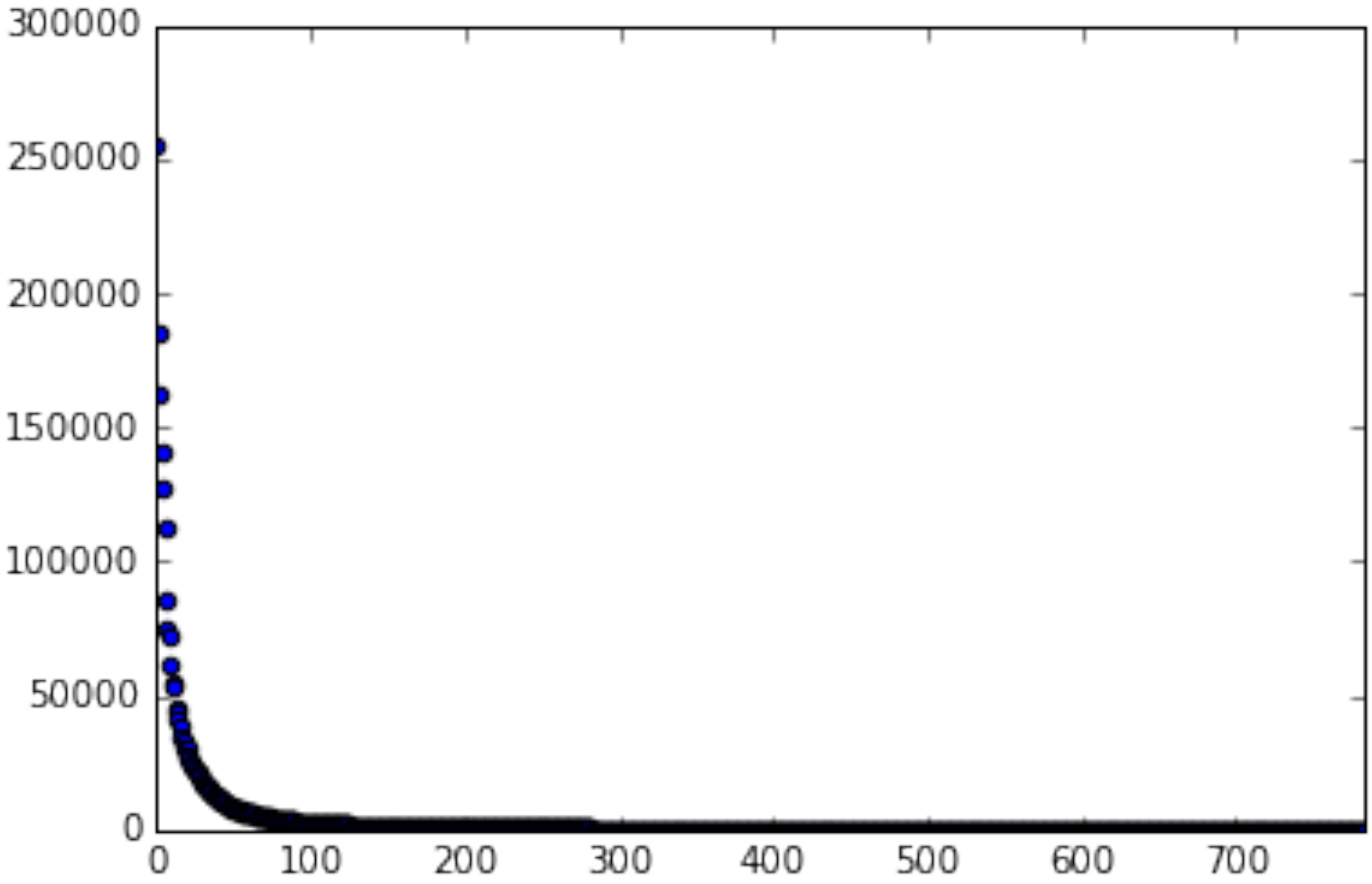


The first 6 PCA components (eigenvectors)



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

Visualizing high-dimensional data

Neuron values in some intermediate layer represent input data in some interesting way, but they are hard to visualize! [there are more than 2 neurons in such a layer, typically]

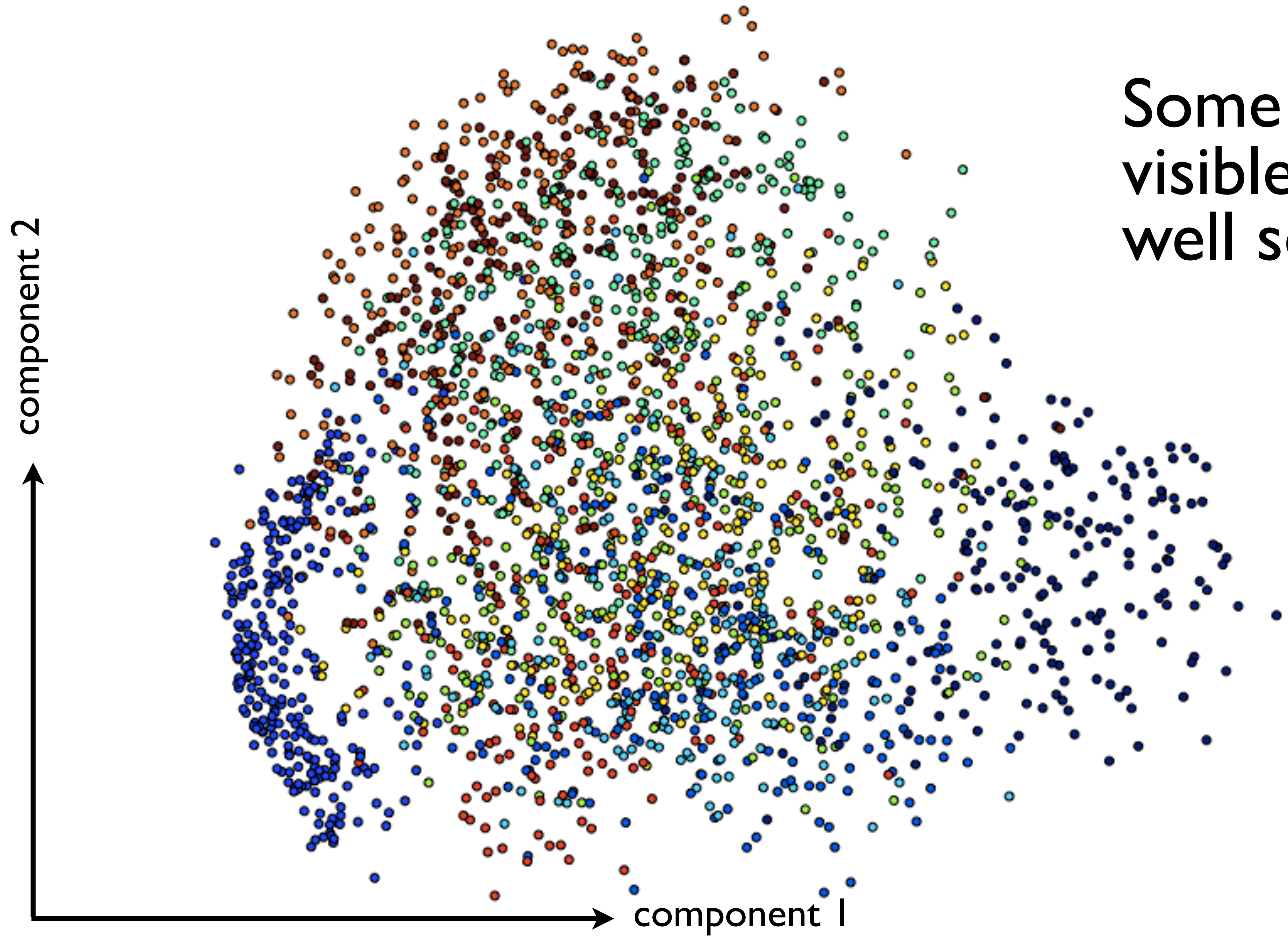
Need some method to project down to 2 dimensions, keeping the distance relation qualitatively similar: "Which inputs are close to each other, which are very different?"

Can also apply this to the input data itself directly, or to some compressed version of it (like PCA components)!

MNIST sample images, reduced to 2D, using PCA

Obtain PCA, then plot components of each image with respect to two eigenvectors with largest eigenvalues (as a point in 2D plane)

Different colors = differently labeled images (diff. digits)

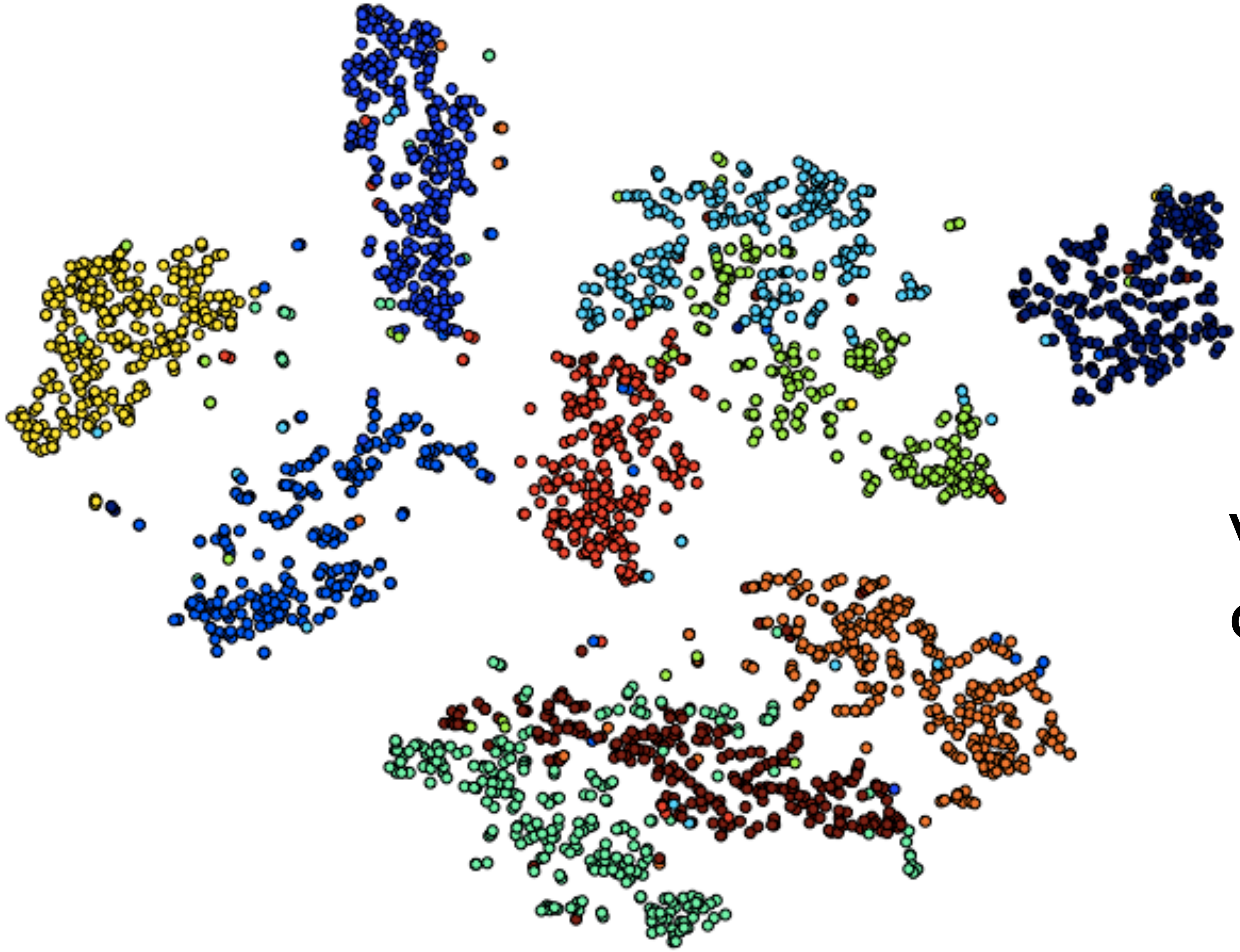


Some trends visible, but not well separated!

MNIST sample images, reduced to 2D, using "t-SNE"

[using python program by Laurens van der Maaten]

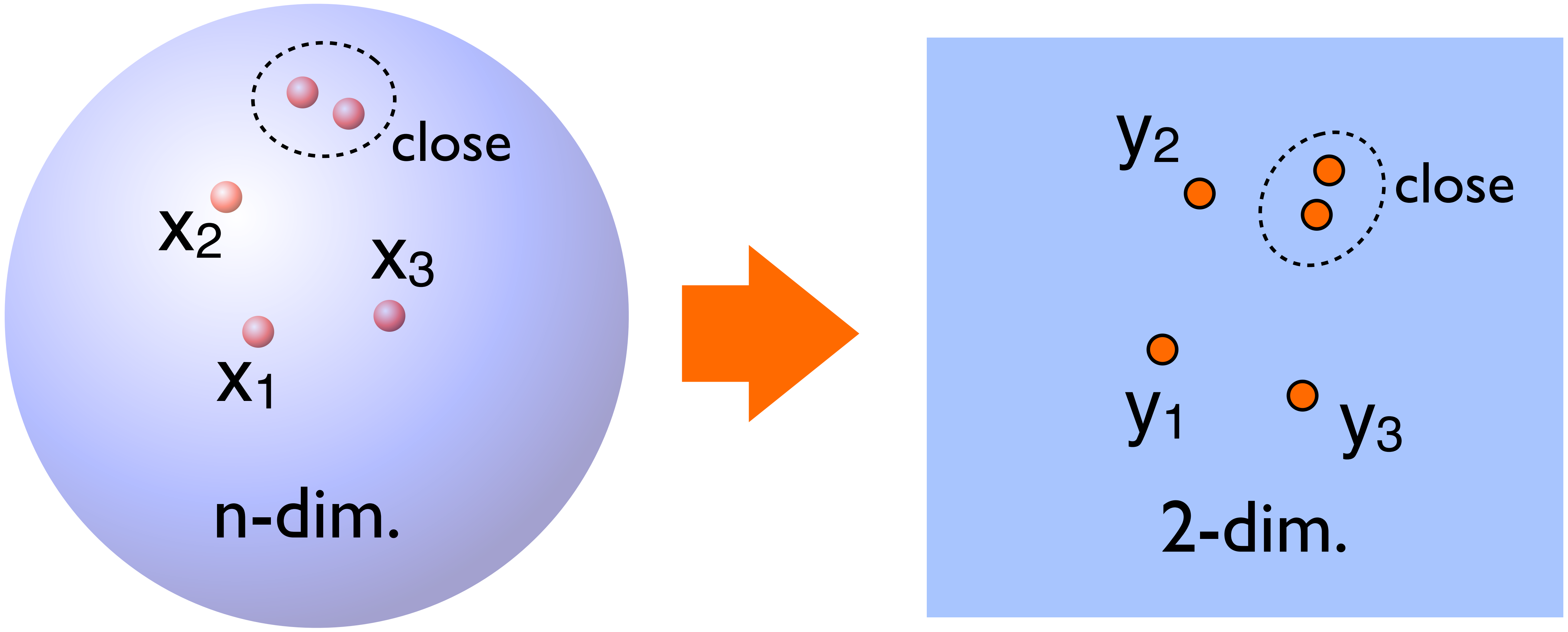
Different colors = differently labeled images (diff. digits)



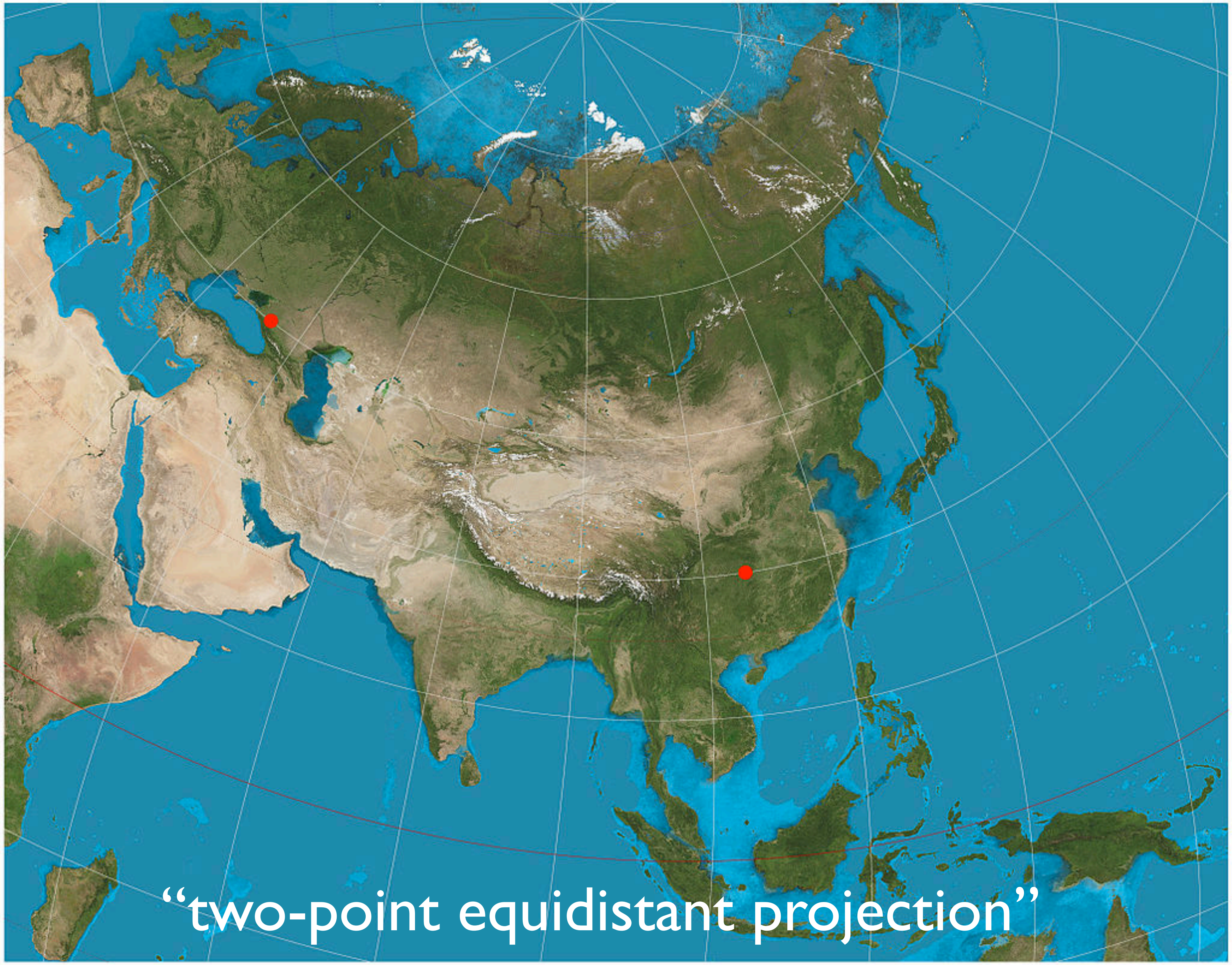
Well-defined clusters!

(starts from 50 PCA components for each image; t-SNE takes about 10min)

Basic idea of dimensionality reduction: reproduce distances in higher-dimensional space inside the lower-dimensional "map", as closely as possible



Usually not perfectly possible: Remember the map-maker's dilemma!



"two-point equidistant projection"

(Wikipedia)

Can define cost-function, that depends on how close the distances of low-dimensional data points “y” are to those of high-dimensional points “x”

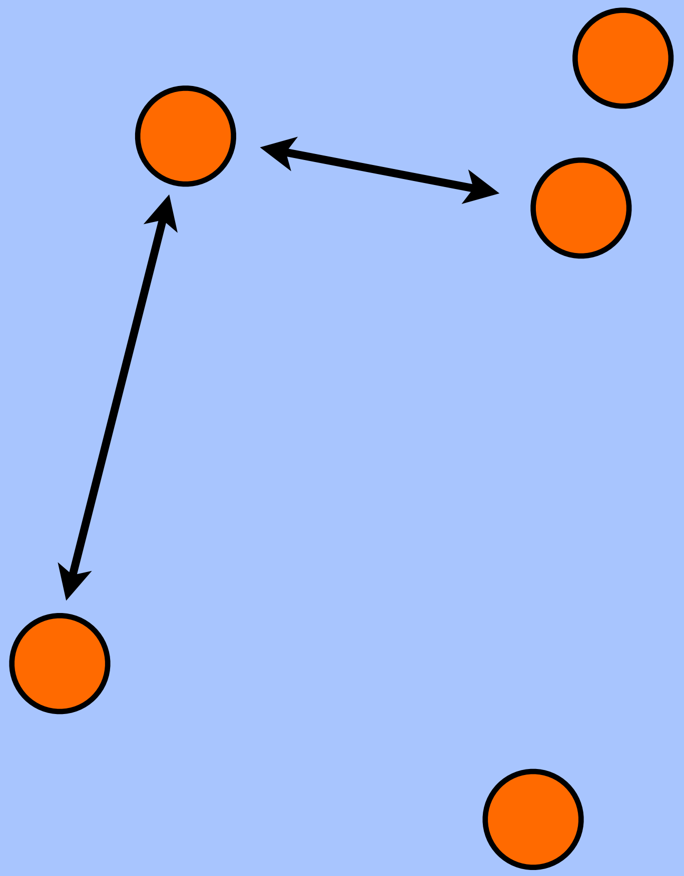
$$C = \sum_{i \neq j} F(|x_i - x_j|, |y_i - y_j|)$$

Then: minimize cost function, using e.g. gradient descent!

Points in low-dim. space repel if they are closer than their counterparts in high-dim. space, and attract otherwise

[Can introduce arbitrary (monotonous) functions of distances]

attractive forces, if high-dim.
distance is smaller than
represented here in low dim.



$$\dot{y}_j = -\frac{\partial C}{\partial y_j}$$

2-dim.

“Stochastic neighbor embedding” (SNE): Define “probability distributions” that depend not only on the distance but also include some normalization

p_{ij} Probability to pick a pair of points (i,j). Defined to be larger if they are close neighbors [in the high-dim. space]

q_{ij} similar for low-dim. space

$$\sum_{i\neq j} q_{ij} = 1 \qquad \sum_{i\neq j} p_{ij} = 1$$

Want q-distribution to be a close approximation of the p-distribution:

$$C = KL(P||Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}.$$

“Kullback-Leibler divergence”, a way of comparing probability distributions

Choices made for t-SNE

[for heuristics behind this see Hinton & v.d.Maaten 2008]

high-dim. space:

(Gaussians dist.)

$$p_{j|i} = \frac{\exp\left(-\|x_i - x_j\|^2/2\sigma_i^2\right)}{\sum_{k\neq i}\exp\left(-\|x_i - x_k\|^2/2\sigma_i^2\right)}$$

$$p_{ij} = \frac{p_{j|i}+p_{i|j}}{2n}$$

low-dim. space:

(Cauchy dist.= "Student-t dist.")

$$q_{ij} = \frac{\left(1+\|y_i - y_j\|^2\right)^{-1}}{\sum_{k\neq l}\left(1+\|y_k - y_l\|^2\right)^{-1}}$$

q is comparatively larger at long distances: allows points in low-dim. space to spread out for intermediate distances (they do not have as much space as high-dim. points! need to give them more room!)

The t-SNE “force”:

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j) \left(1 + \|y_i - y_j\|^2\right)^{-1}$$

spring-like

gravity”-like at larger distances

sign depends on match between low- and high-dim. distributions

An example application from biophysics

“T-SNE visualization of large-scale neural recordings”

George Dimitriadis, Joana Neto, Adam Kampff

Multiple electrodes record voltage time-traces due to nearby spiking neurons: but which spike belongs to which neuron?

Visualizing the evolution during t-SNE optimization.

http://biorxiv.org/content/early/2016/11/14/087395.figures-only

take neurons out of a multi-layer convolutional network that classifies images, and represent using t-SNE

(example by Andrej Karpathy)

[t-SNE applied to a 4096-dim. representation]

http://cs.stanford.edu/people/karpathy/cnnembed/

take neurons out of a multi-layer convolutional network that classifies images, and represent using t-SNE

(example by Andrej Karpathy)

[t-SNE applied to a 4096-dim. representation]

http://cs.stanford.edu/people/karpathy/cnnembed/

take neurons out of a multi-layer convolutional network that classifies images, and represent using t-SNE

(example by Andrej Karpathy)

[t-SNE applied to a 4096-dim. representation]

http://cs.stanford.edu/people/karpathy/cnnembed/

Open access to 1,276,278 e-prints in Physics, Mathematics, Computer Science, Quantitative Biology, Quantitative Finance and Statistics

Subject search and browse:

20 Apr 2017: Applied Physics subject area added to arXiv
10 Mar 2017: New members join arXiv Member Advisory Board
06 Mar 2017: arXiv Scientific Director Search
10 Feb 2017: Attention Submitters: our TeX processing system has been updated
See cumulative "What's New" pages. Read robots beware before attempting any automated d

1.3 million articles

Physics

- Astrophysics (**astro-ph** new, recent, find)
  includes: Astrophysics of Galaxies; Cosmology and Nonga s; Earth and Planetary Astrophysics; High Energy Astrophysical Phenomena; Instrumentation and Methods fo ysics; Solar and Stellar Astrophysics
- Condensed Matter (**cond-mat** new, recent, find)
  includes: Disordered Systems and Neural Networks; Materials Science; Mesoscale and Nanoscale Physics; Other Condensed Matter; Quantum Gases; Soft Condensed Matter; Statistical Mechanics; Strongly Correlated Electrons; Superconductivity
- General Relativity and Quantum Cosmology (**gr-qc** new, recent, find)
- High Energy Physics - Experiment (**hep-ex** new, recent, find)
- High Energy Physics - Lattice (**hep-lat** new, recent, find)
- High Energy Physics - Phenomenology (**hep-ph** new, recent, find)
- High Energy Physics - Theory (**hep-th** new, recent, find)
- Mathematical Physics (**math-ph** new, recent, find)
- Nonlinear Sciences (**nlin** new, recent, find)
  includes: Adaptation and Self-Organizing Systems; Cellular Automata and Lattice Gases; Chaotic Dynamics; Exactly Solvable and Integrable Systems; Pattern Formation and Solitons
- Nuclear Experiment (**nucl-ex** new, recent, find)
- Nuclear Theory (**nucl-th** new, recent, find)
- Physics (**physics** new, recent, find)
  includes: Accelerator Physics; Applied Physics; Atmospheric and Oceanic Physics; Atomic Physics; Atomic and Molecular Clusters; Biological Physics; Chemical Physics; Classical Physics; Computational Physics; Data Analysis, Statistics and Probability; Fluid Dynamics; General Physics; Geophysics; History and Philosophy of Physics; Instrumentation and Detectors; Medical Physics; Optics; Physics Education; Physics and Society; Plasma Physics; Popular Physics; Space Physics
- Quantum Physics (**quant-ph** new, recent, find)

paperscape.org

The whole arXiv preprint server, represented as a 2D map

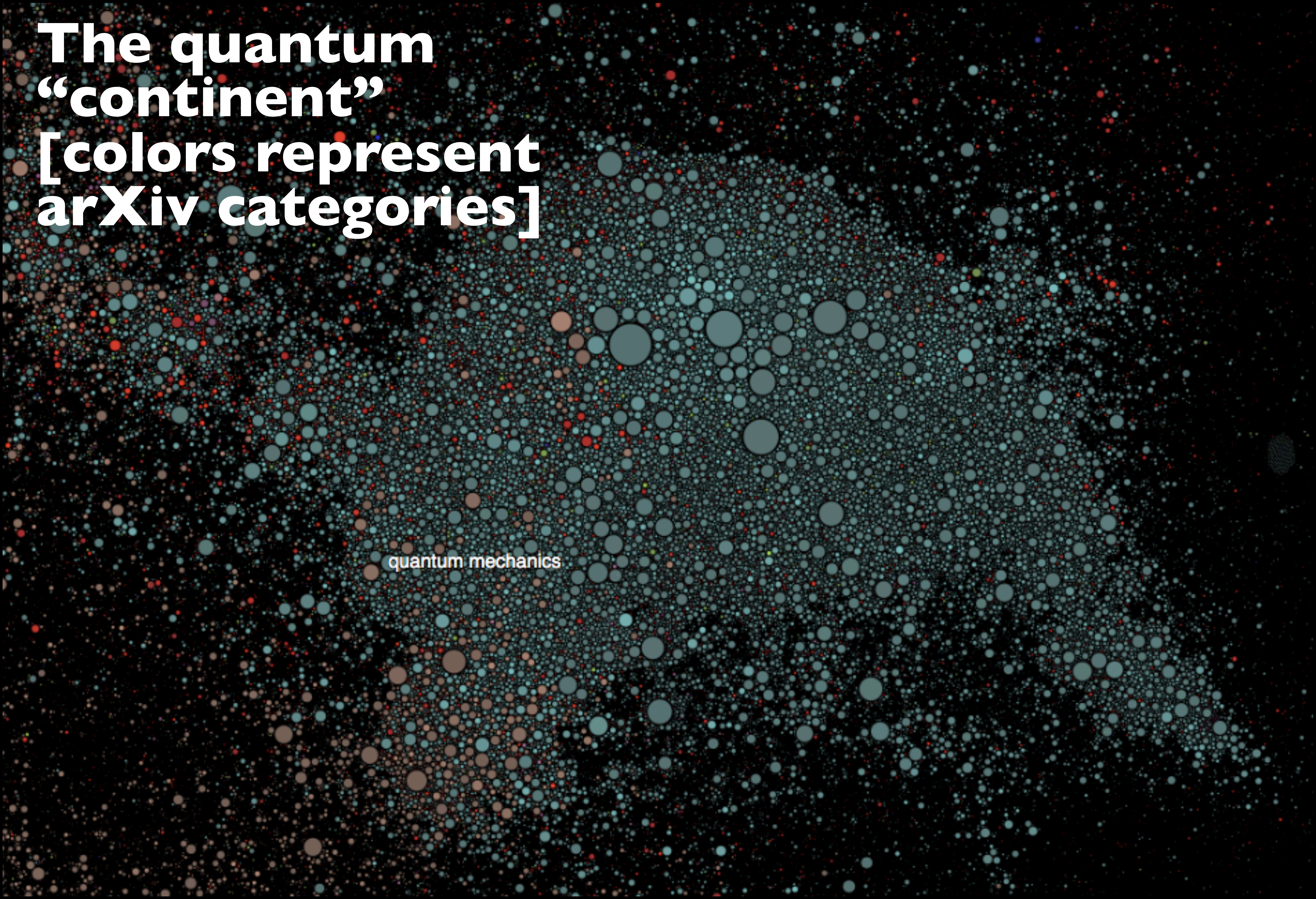
Paperscape uses a simple physical model (similar to t-SNE, but more physical).

Between each two papers there are two forces:

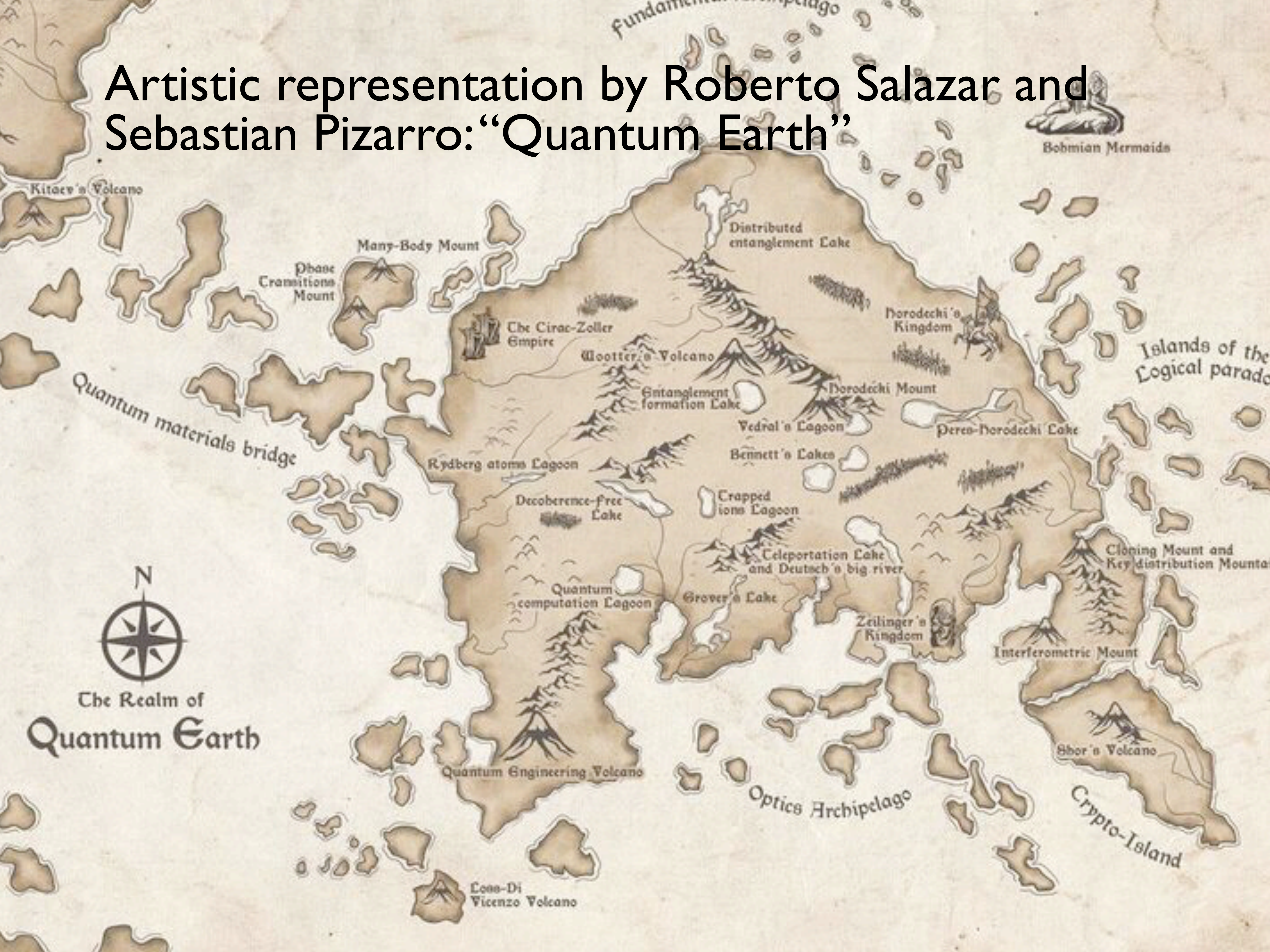
- repulsion (anti-gravity inverse-distance force)
- attraction of any paper to all its references by a linear spring
- also avoid overlap (circle sizes represent number of citations to that paper)

Every morning, after new papers are announced, the map of all 1.3 million papers on the arXiv is re-calculated (takes 3-4 hours)

The quantum "continent" [colors represent arXiv categories]



Artistic representation by Roberto Salazar and Sebastian Pizarro: "Quantum Earth"



Optimized gradient descent algorithms

How to speed up stochastic gradient descent?

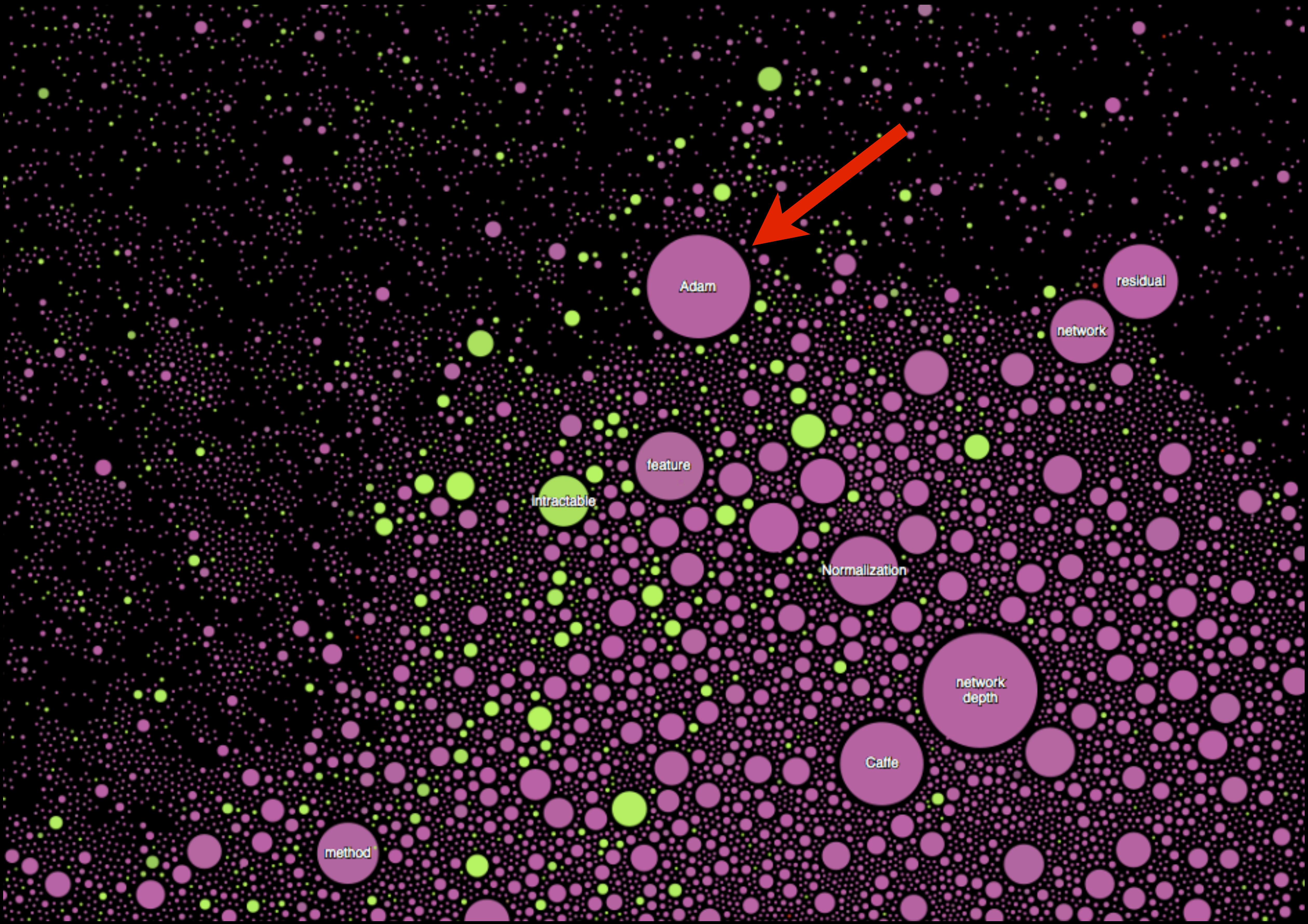
- accelerate ("momentum") towards minimum
- Automatically choose learning rate
- ...even different rates for different weights

Summary: a few gradient update methods

see overview by S. Ruder https://arxiv.org/abs/1609.04747

adagrad	divide by RMS of all previous gradients
RMSprop	divide by RMS of last few previous gradients
adadelta	same, but multiply also by RMS of last few parameter updates
adam	divide running average of last few gradients by RMS of last few gradients (* with corrections during earliest steps)

adam often works best


Adam
residual
network
feature
Intractable
Normalization
network depth
Caffe
method

Please download “Part Two” to continue

http://machine-learning-for-physicists.org