A Shallow Introduction to Deep Learning

Tutorial

Ghent University, LT3 Group
Thu 30 June 2016

mike.kestemont@uantwerp.be
University of Antwerp, Department of Literature
www.mike-kestemont.org

Aim

- Intuitive introduction to Deep Learning
  - Almost no mathematics or calculus
- Lots of hands-on coding in notebooks
- Show how (surprisingly) simple it can be
- Theory in morning; Practice in afternoon

Requirements

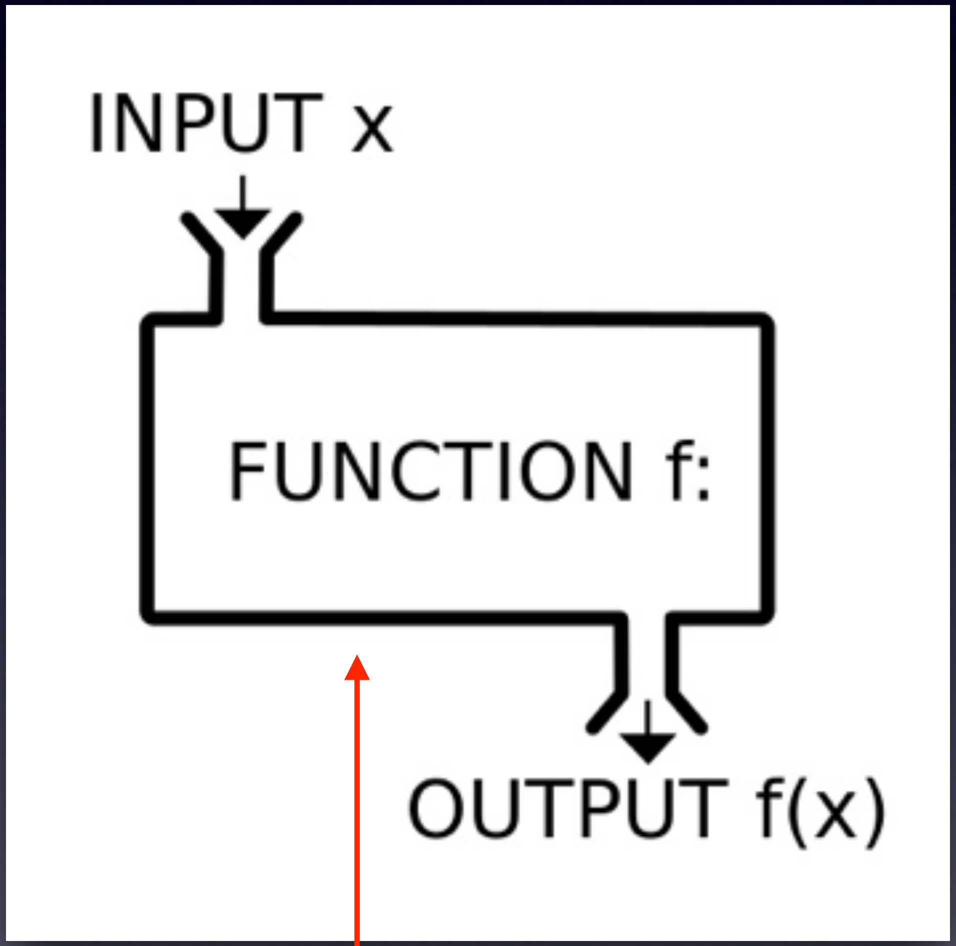
- Python 2.7+ (All code compatible with Python 3); Anaconda?
- External (apart from Jupiter for notebooks)
  - Scikit-learn
  - Numpy
  - Matplotlib
- DL specific (bleeding edge versions from Github)
  - TensorFlow
  - Theano
  - Keras

Disclaimer

- Training as philologist
- Hobby that got out of hand…
- Not good at math and calculus
- Most of what I know through self-study

Model

- System that takes an input to produce a certain output
- Has set of parameters Θ that can be adjusted to produce a different output
- Model = System = Function f
    - f_Θ(input) -> output



Magic Box

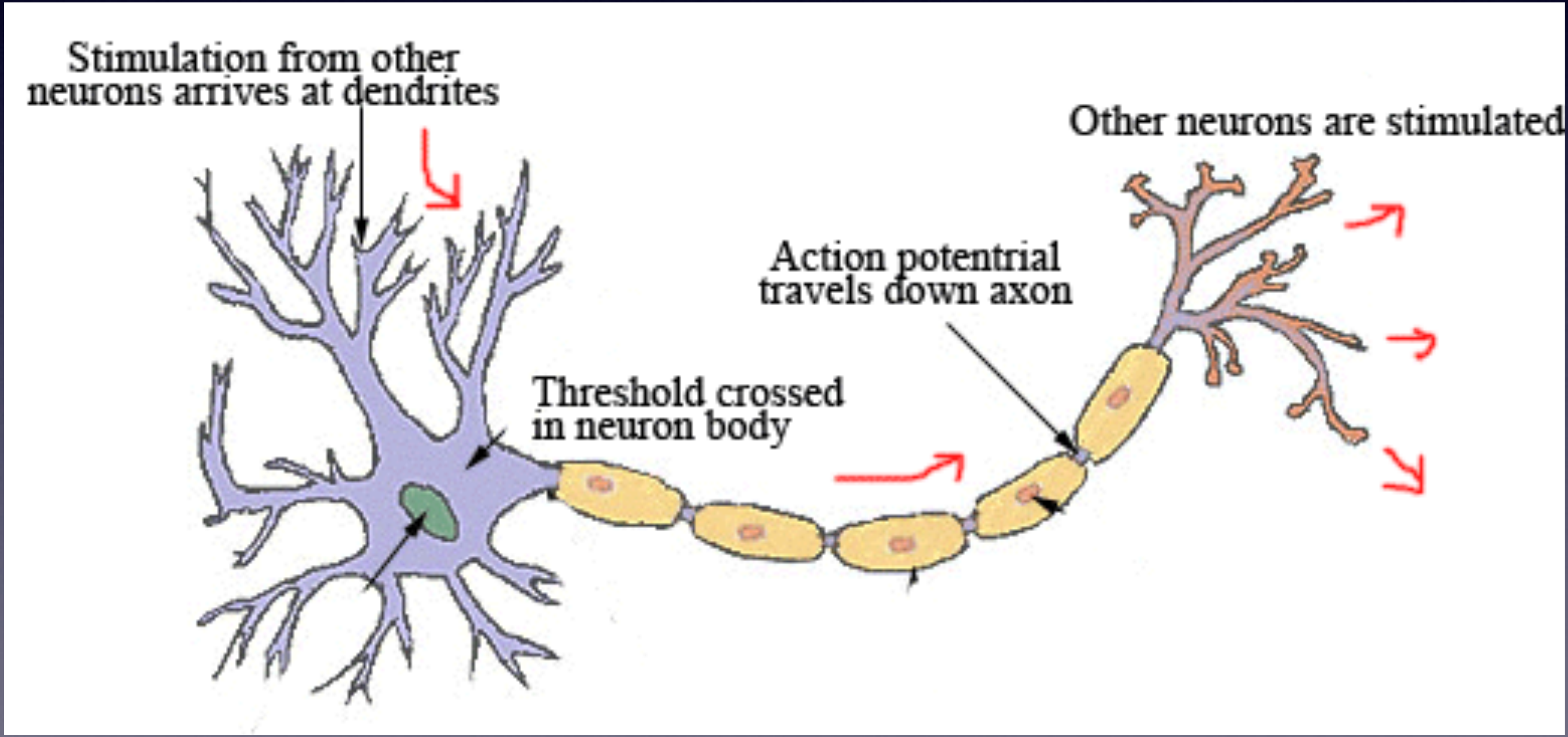
‘Neural’ networks

- Historically **inspired** by working human brain
  - Exaggerated in media…
  - But interesting parallels
- Still today: architectures often described with brain terminology (neurons, activations, …)

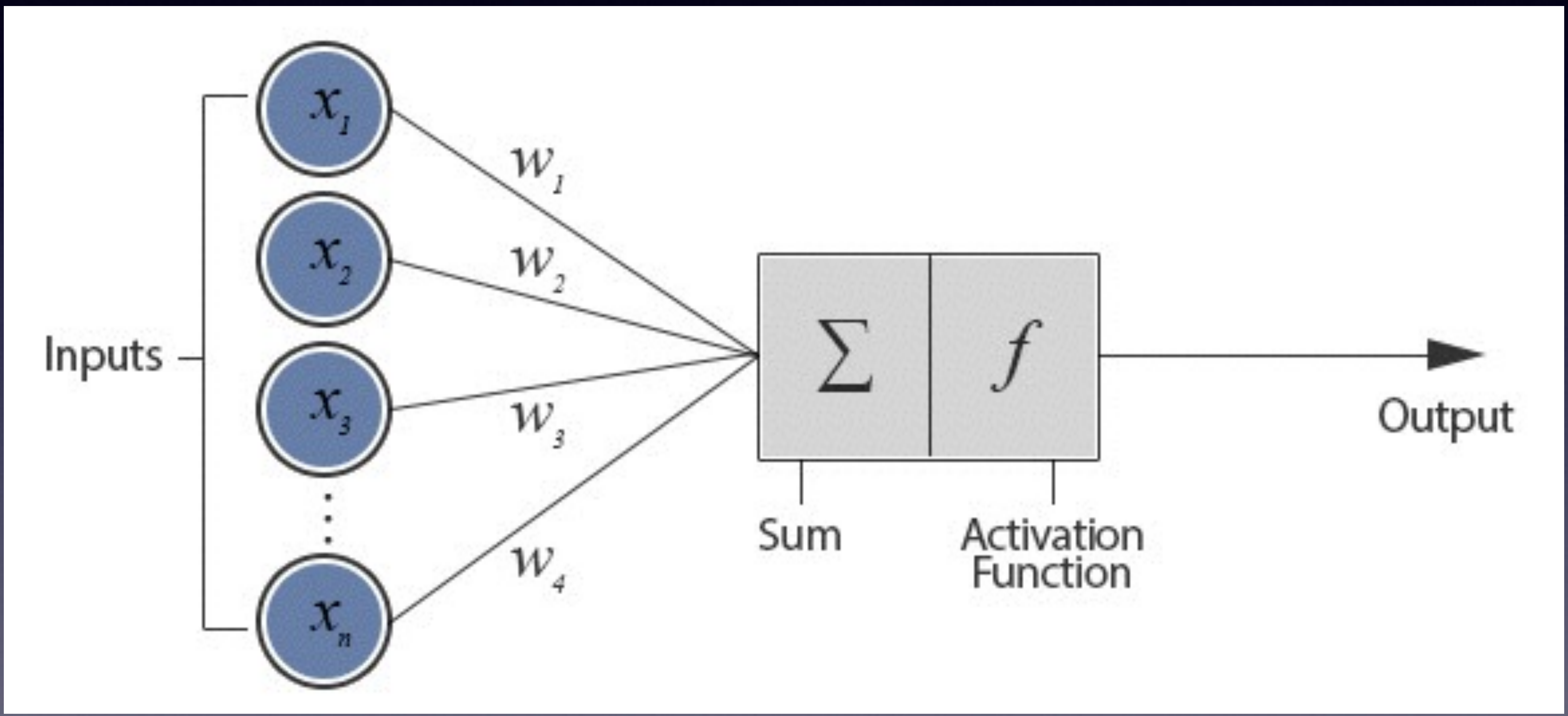
Brain as network of information units (*neurons*) that can share information

Single neuron

Sum of incoming connections determines whether neuron will ‘fire’ (threshold)



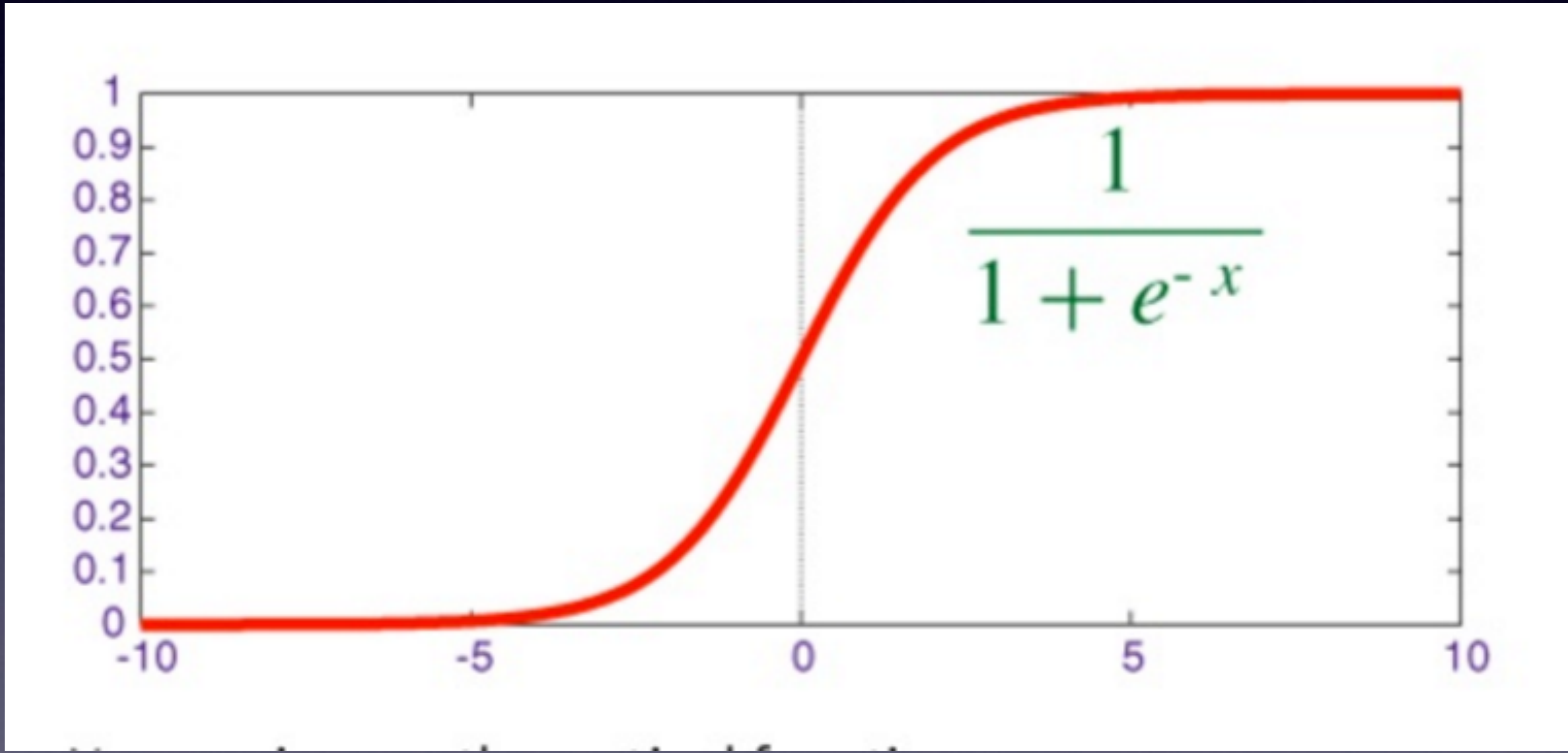
Mathematically



Weights control sensitivity of neuron to information

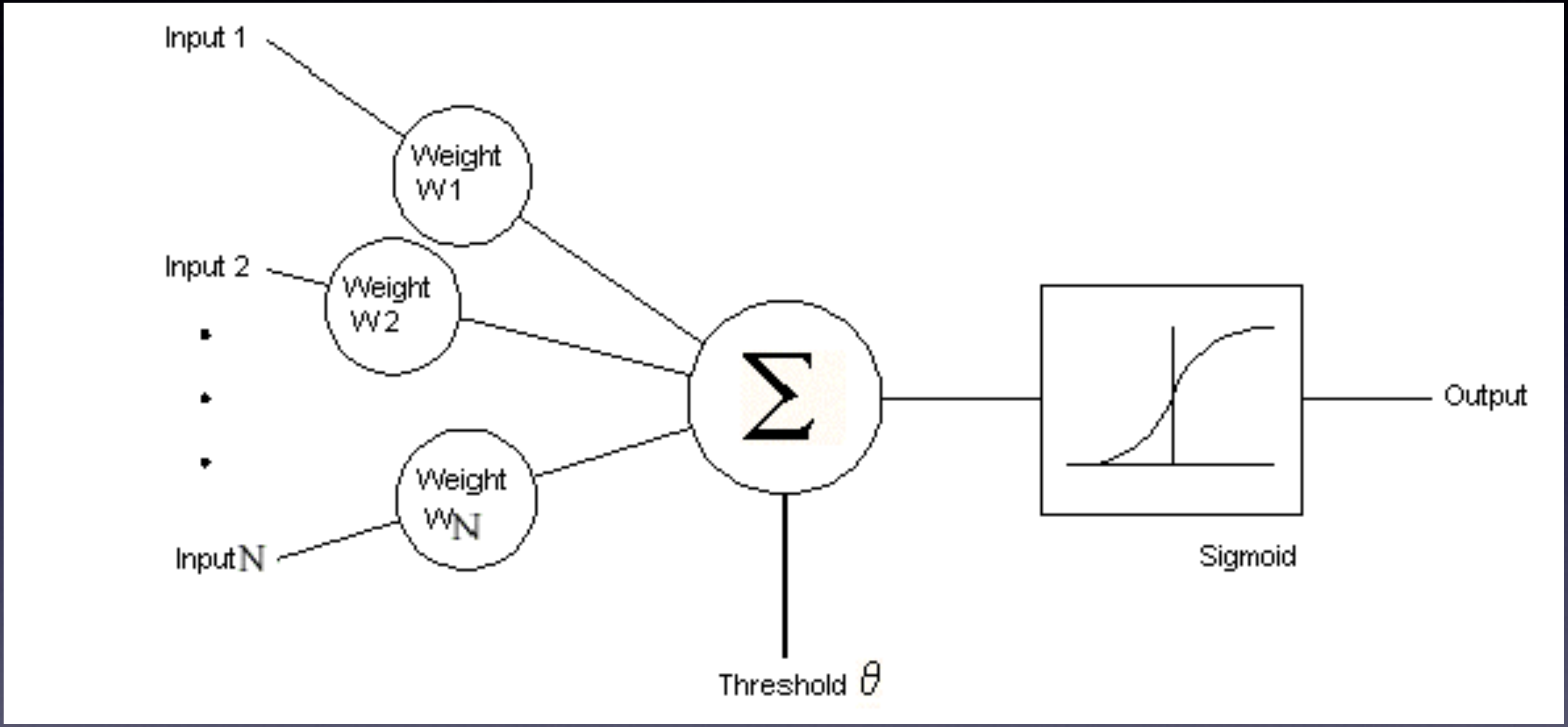
Activation function

Squash info in range [0, 1]



Sigmoid activation (historically dominant)

Perceptron



Already useful for regression (*single output*) in ML
E.g. predict house prices using location, bedrooms, …

Notebook: Perceptron

Training

How do we avoid having to set weights ourselves?

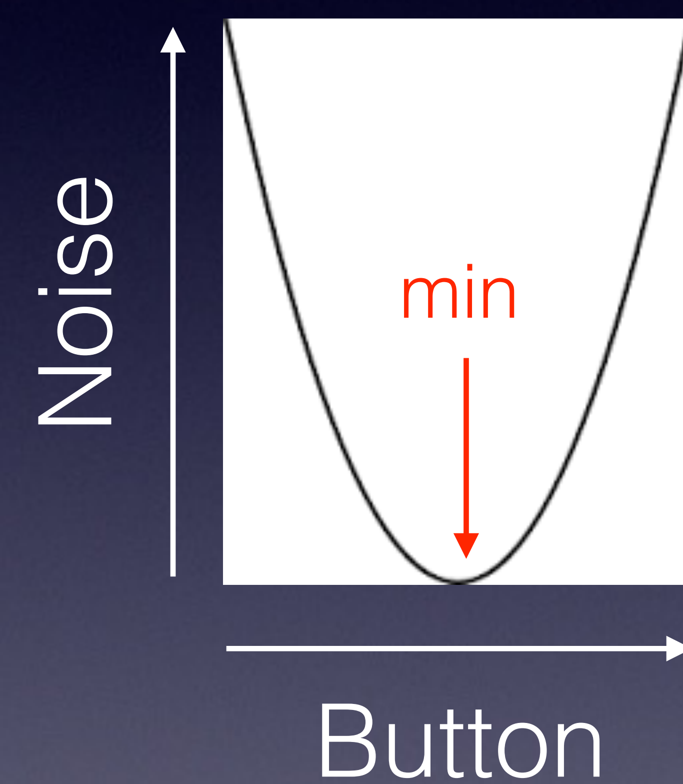
Tuning an Old Radio

Initially lots of noise, reduce by turning knob

Volume

Frequency

Tuning an Old Radio

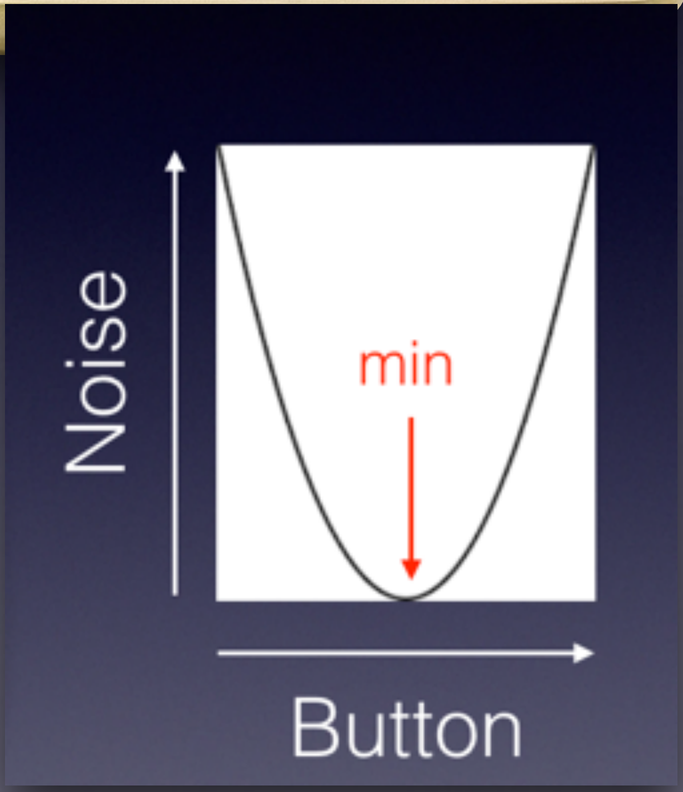


Volume

Frequency

Intuition?

- Left and right
- Movements get slower as you finetune: learning rate
- You don't know how the radio works internally: only knob and a loss estimate
- Naming conventions:
    - radio = system; knob = parameter
    - sound quality = loss function (which we want to minimize)



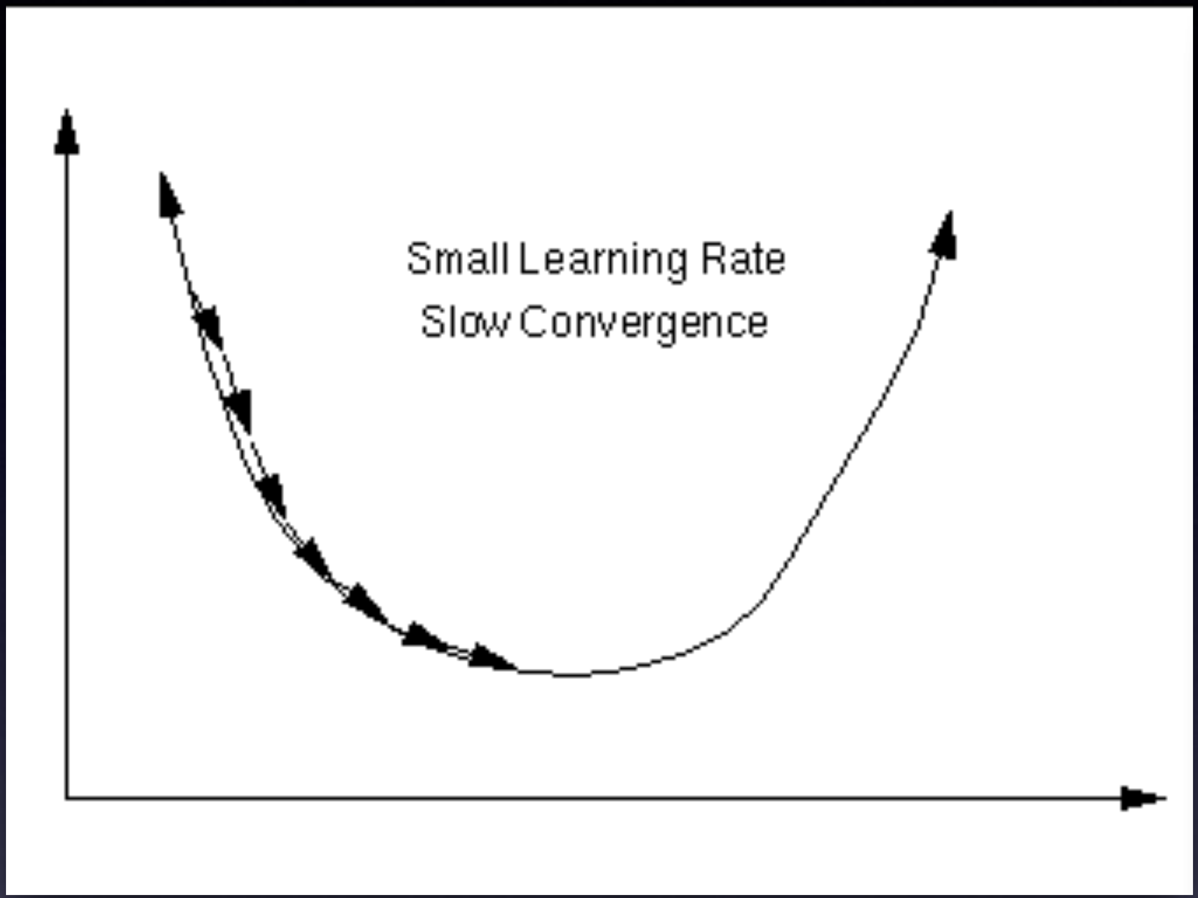
In neural networks?

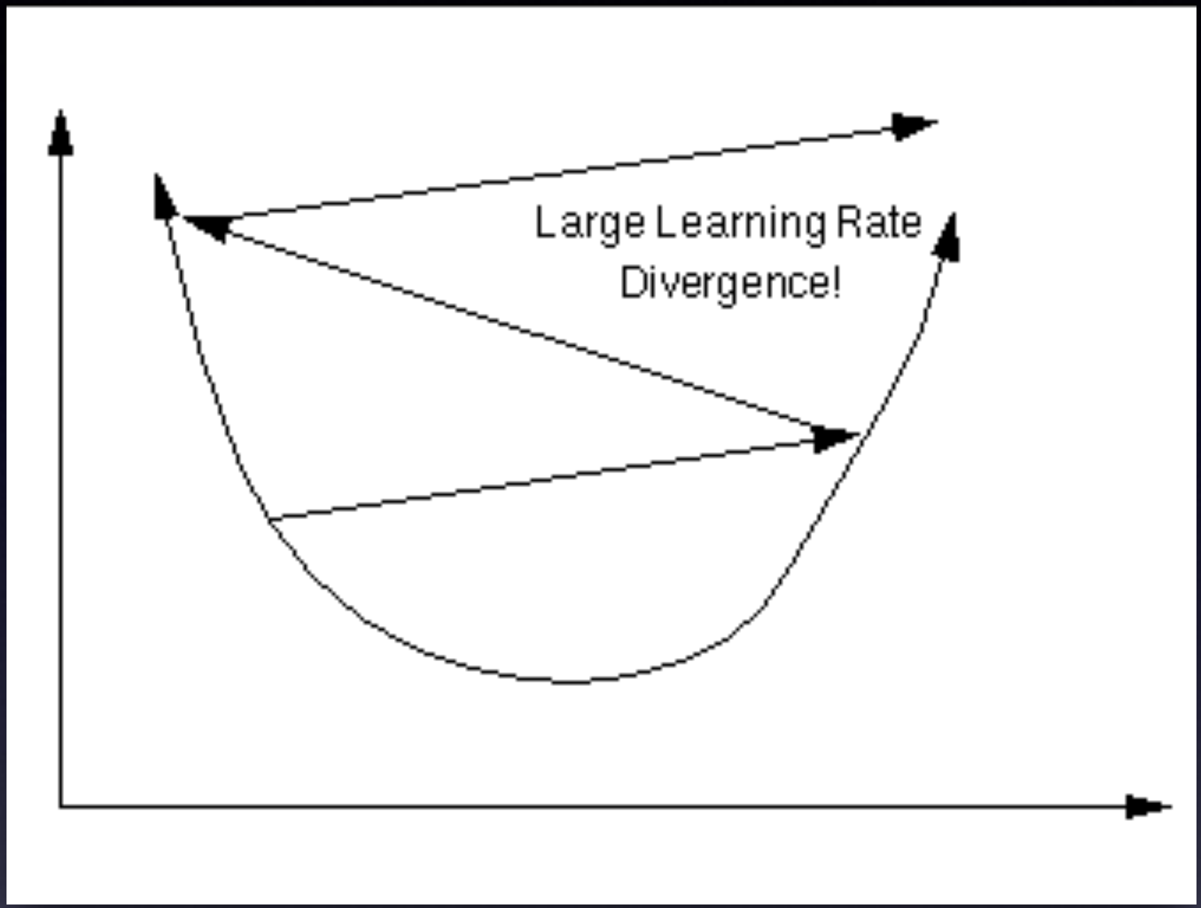
System or function with many more knobs,
but exact same principle: one-by-one adjustments

Training?

- Find the parametrization which minimizes the loss function
- How? Hard, slow and ugly ways:
  - Random search?
  - Try out +/- for each knob and keep best setting
  - …

Learning rate





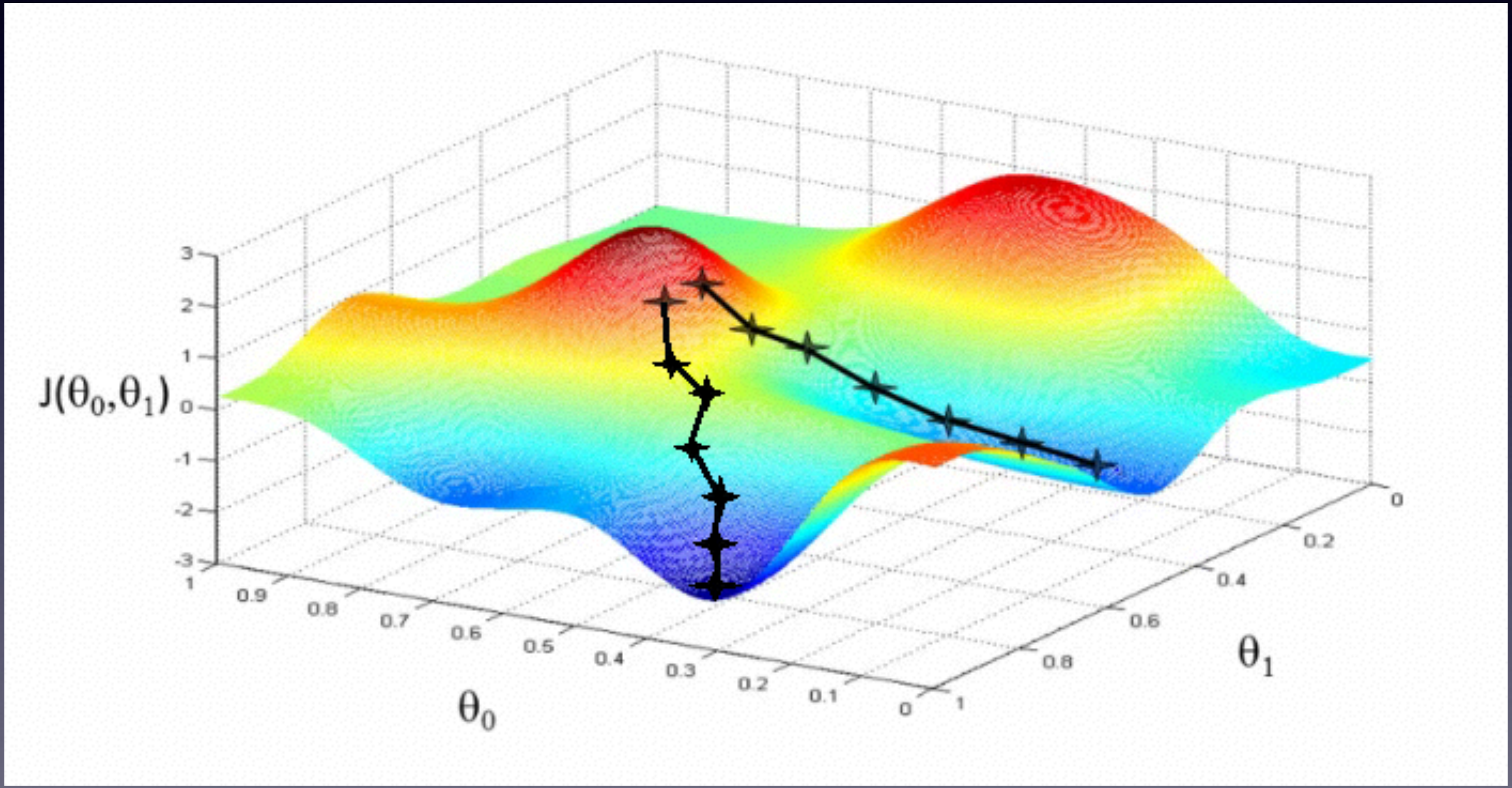
Is real issue in practice (cf. radio):

- too **low** a learning rate: convergence to slow
- too **high** a learning rate: you ‘miss’ the optimum

Notebook: Naive optimization

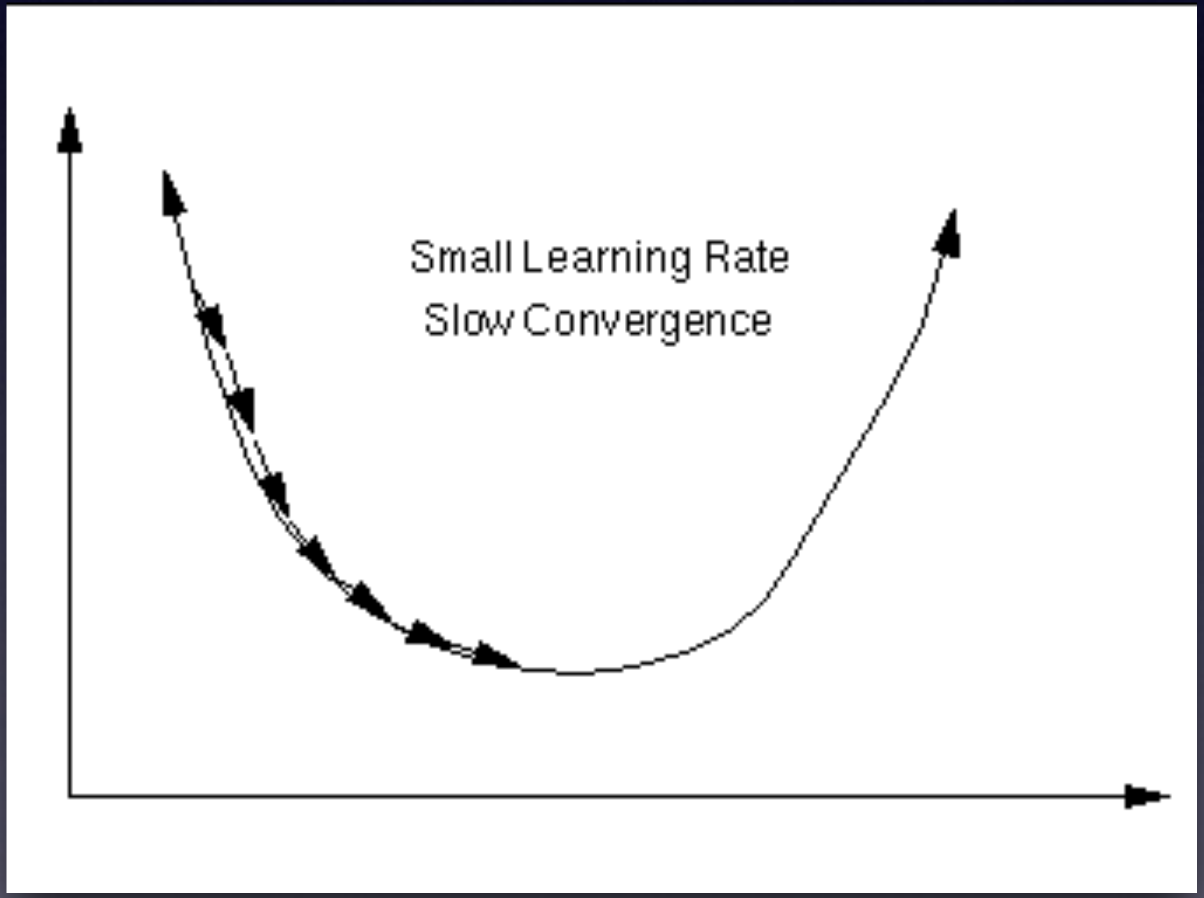
Minimize loss: or ‘objective’ function

Ski down a hill, preferably ASAP



What is wrong with our code?

- Our optimisation is (ugly, but esp.) slow:
  - calculate results both 'minus' and 'plus'
- Our learning steps are the same throughout



Solution

- We don't have to calculate plus and minus…
- Because we can calculate the gradient of each parameters!
- Partial derivative
- 'Gradient ascent'

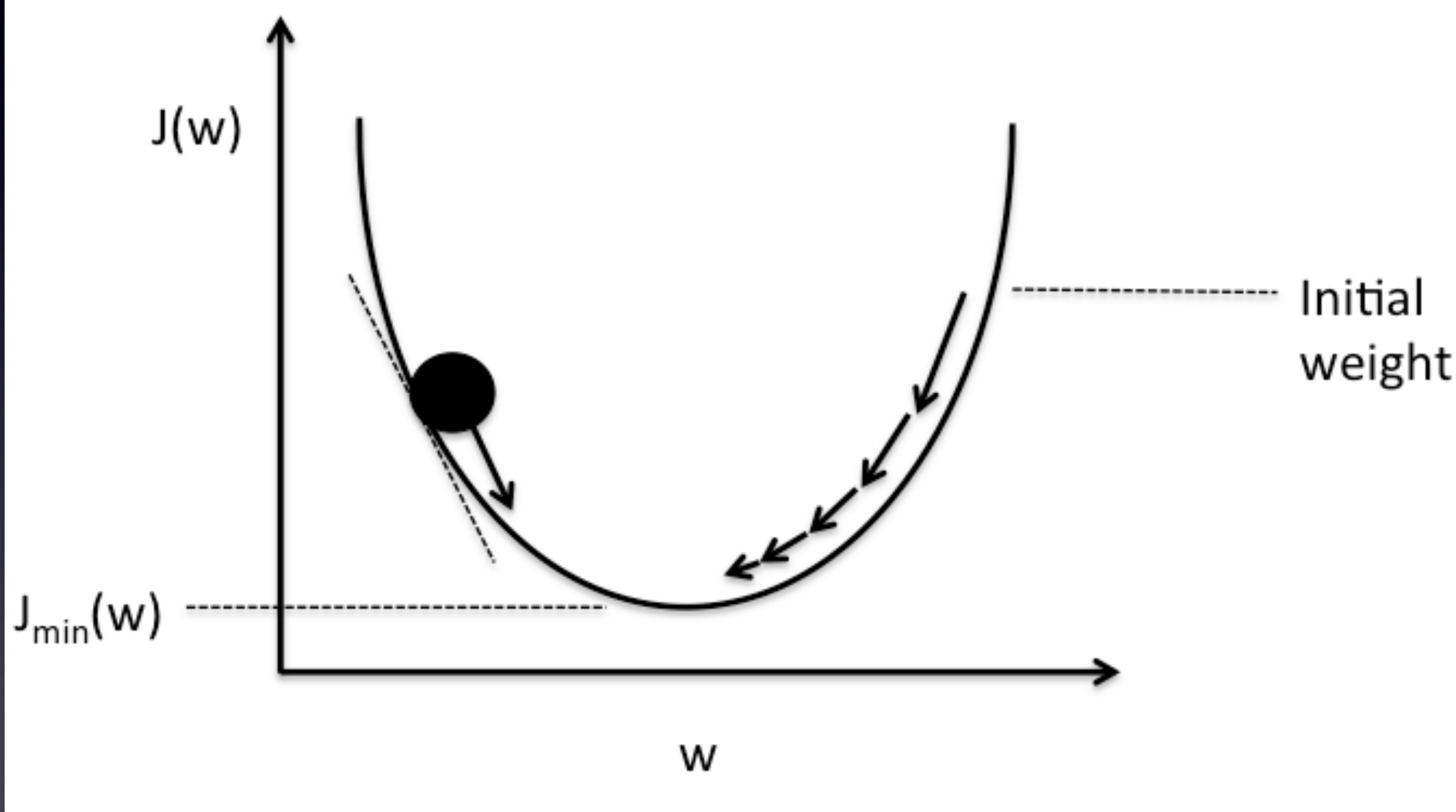
$$\frac{\partial f}{\partial x}$$

'How much does a change in parameter x affect the total model *f()*?'

Optimization?

- Calculate partial derivatives for each parameter
- Update each parameter using rule:
  - param -= learning_rate * gradient
- Negative gradient: parameter grows larger

Solution: Gradient descent



Schematic of gradient descent.

How?

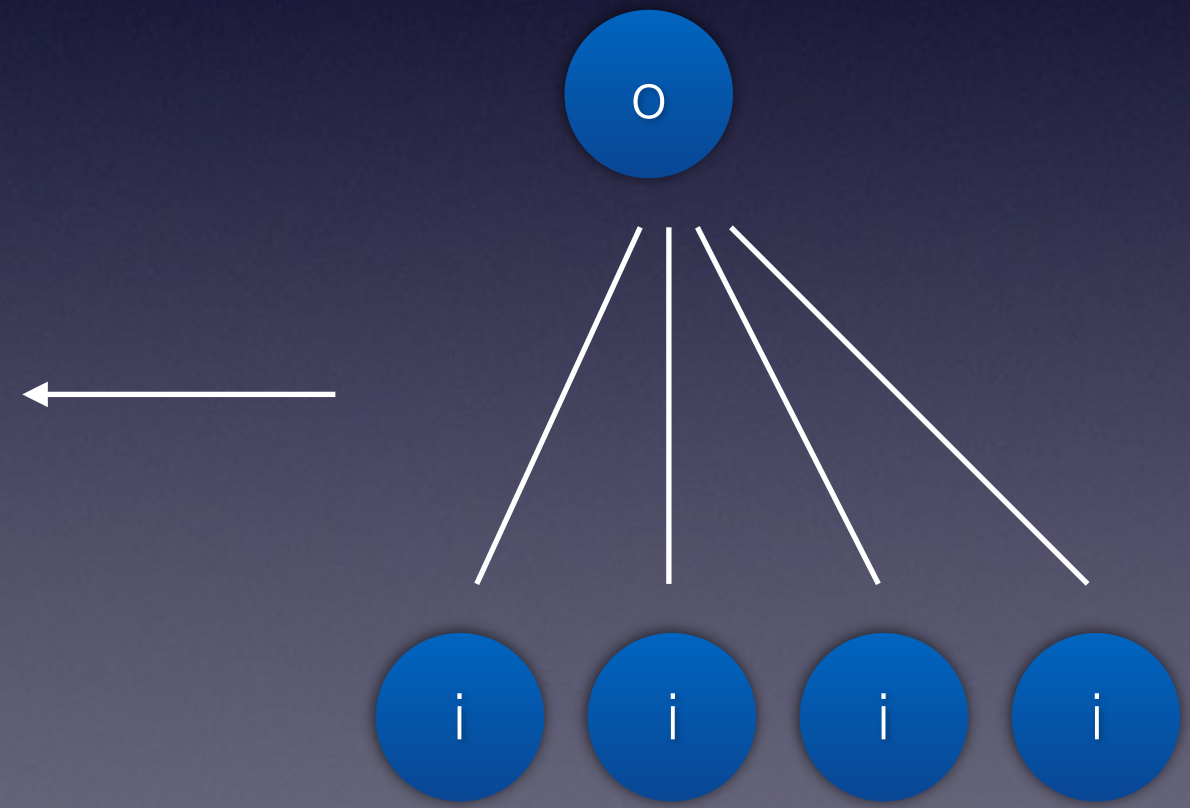
- Gradient descent to be inflexible:
  - manual derivation
  - hard-coding
- Now: libraries for automatic differentiation (you don't to know the math!)
- Python: Theano, TensorFlow, …
- You specify *f()*: library can return the gradients

Notebook: gradient descent in Theano

From simple perceptron…

One class, one weight vector

‘weighted sum’
np.multiply(X, w).sum()



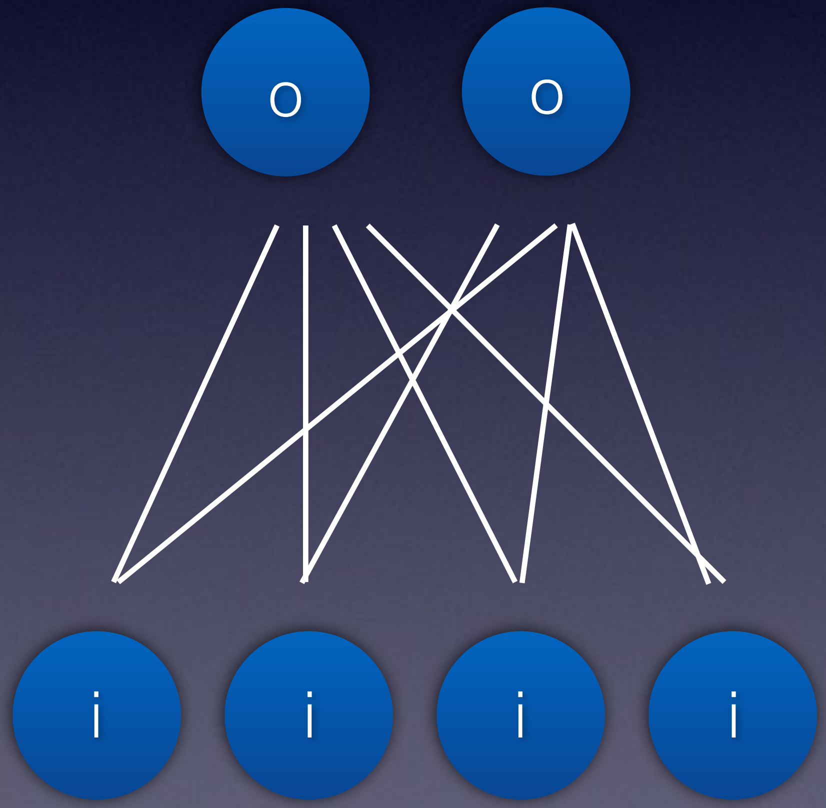
… to classification?

n classes, *n* weight vectors, 1 weight matrix

'weighted sum'
np.multiply(X, w).sum()

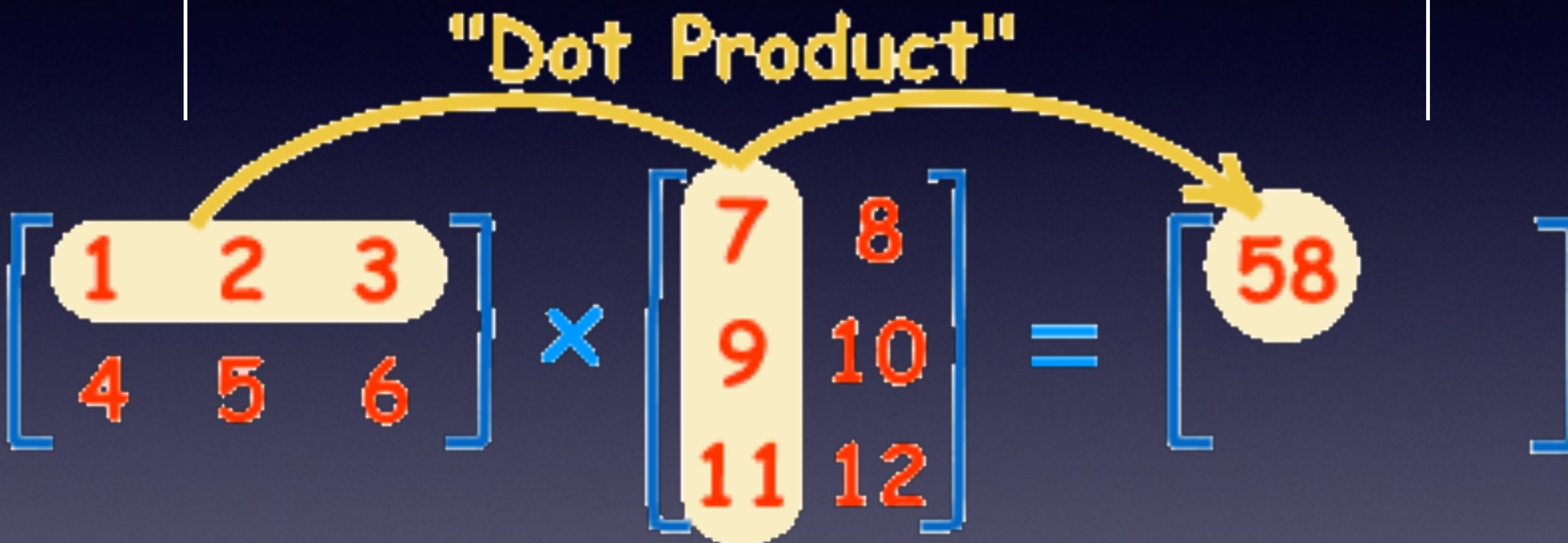
↓

'dot product'
np.dot(X, W)



Feature vectors (X)

output



Weight matrix (W)

The ‘Dense’ Layer

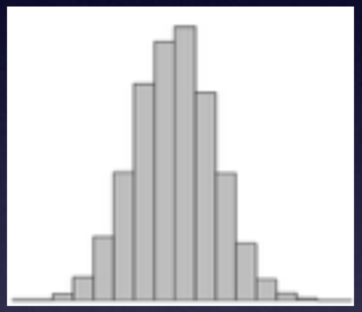
A dot product of an input matrix
with a weight matrix, and
addition of bias vector

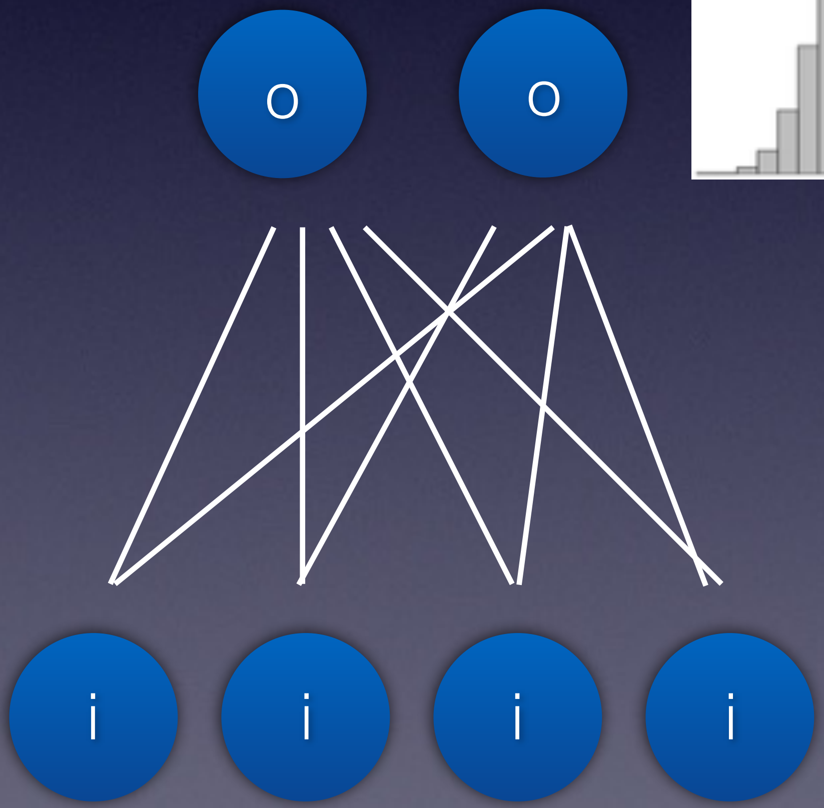
$$output = X \cdot W + bias$$

The **single most fundamental** building block
in deep learning. All fancy stuff goes back to this!

Classification

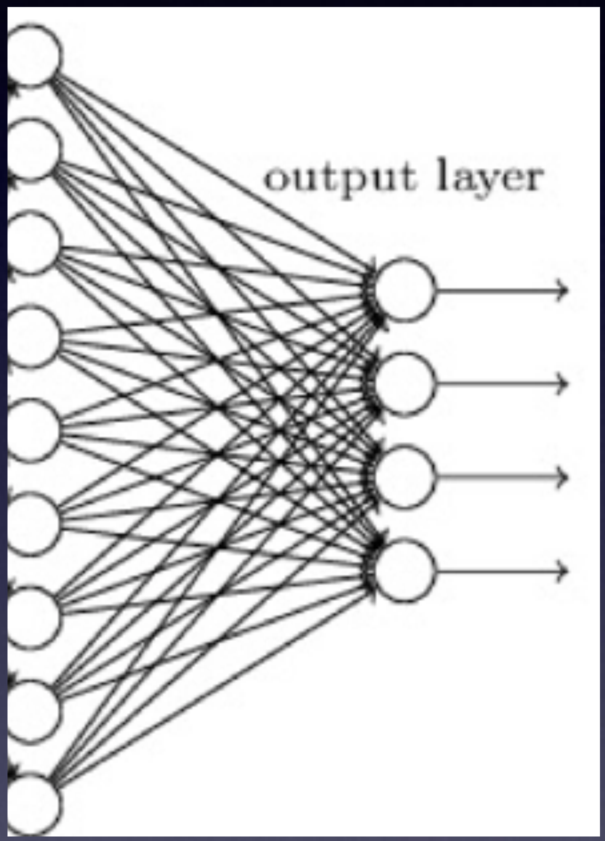
- Use ‘softmax’ to produce probabilities
- Also [0-1] normalization
- Select class with highest probability
- “Logistic Regression”

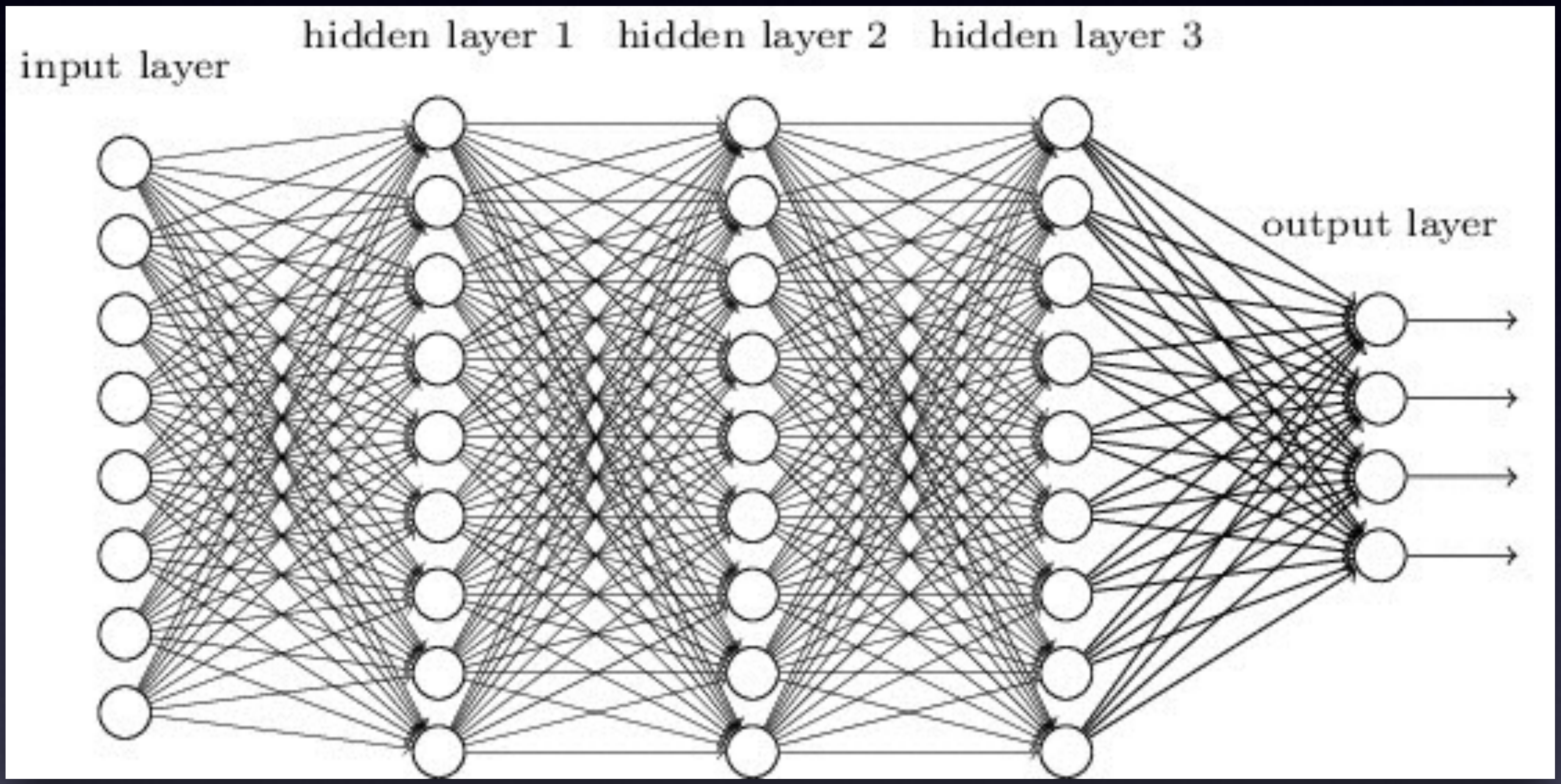




$$P(y = j|\mathbf{x}) = \frac{e^{\mathbf{x}^{\mathsf{T}}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^{\mathsf{T}}\mathbf{w}_k}}$$

“Deep” Learning?

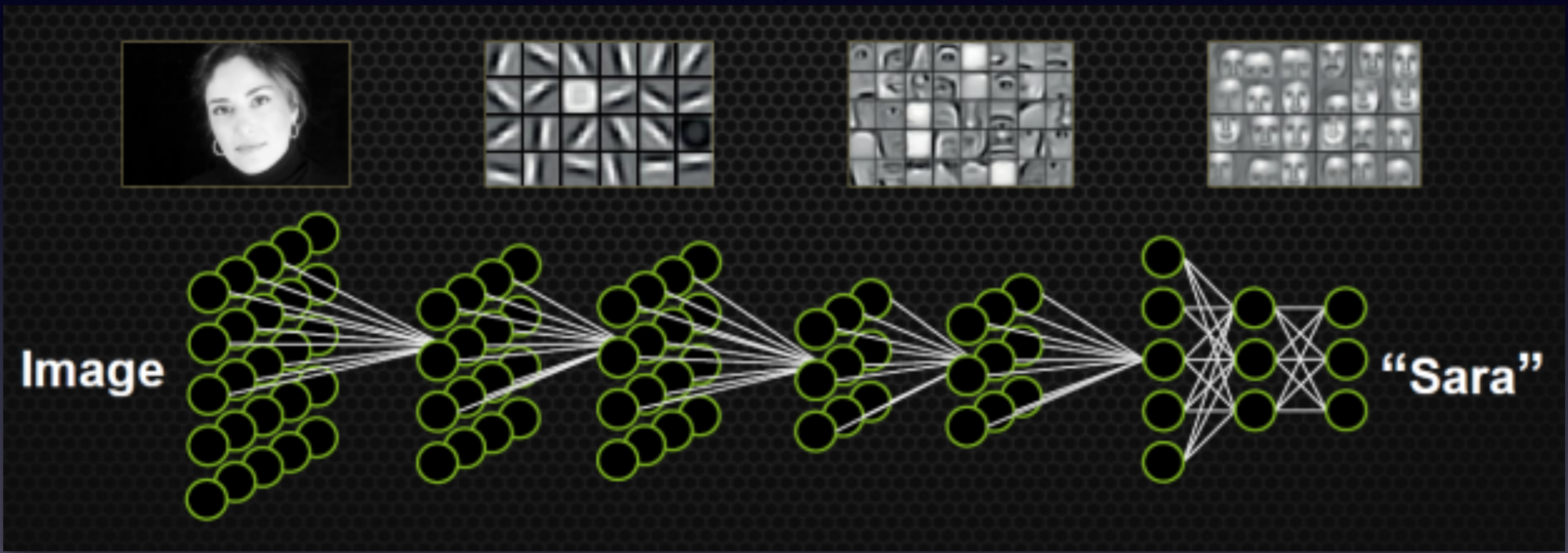




Stack ‘hidden’ layers between input and output layer

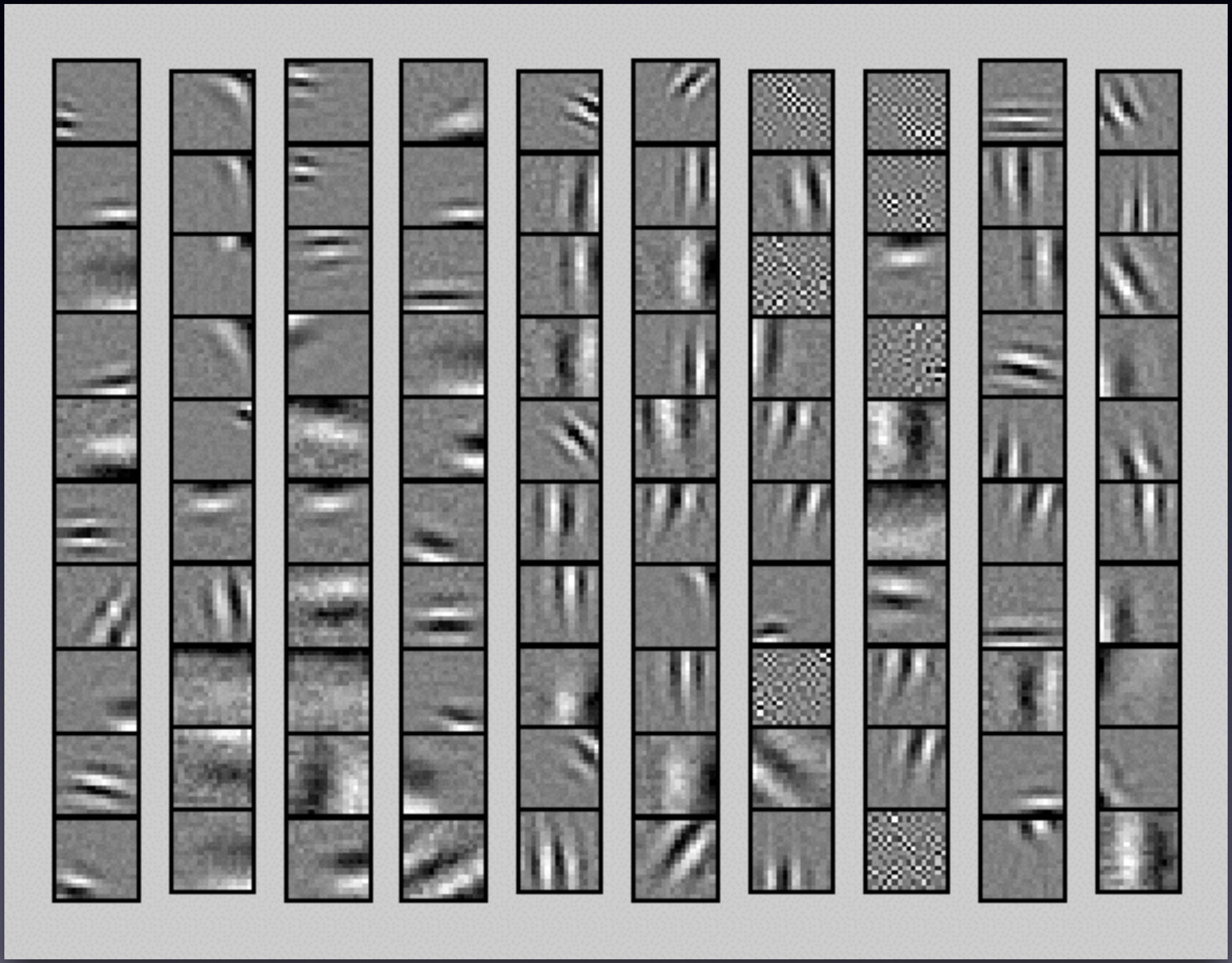
Computer Vision

Importance of layers

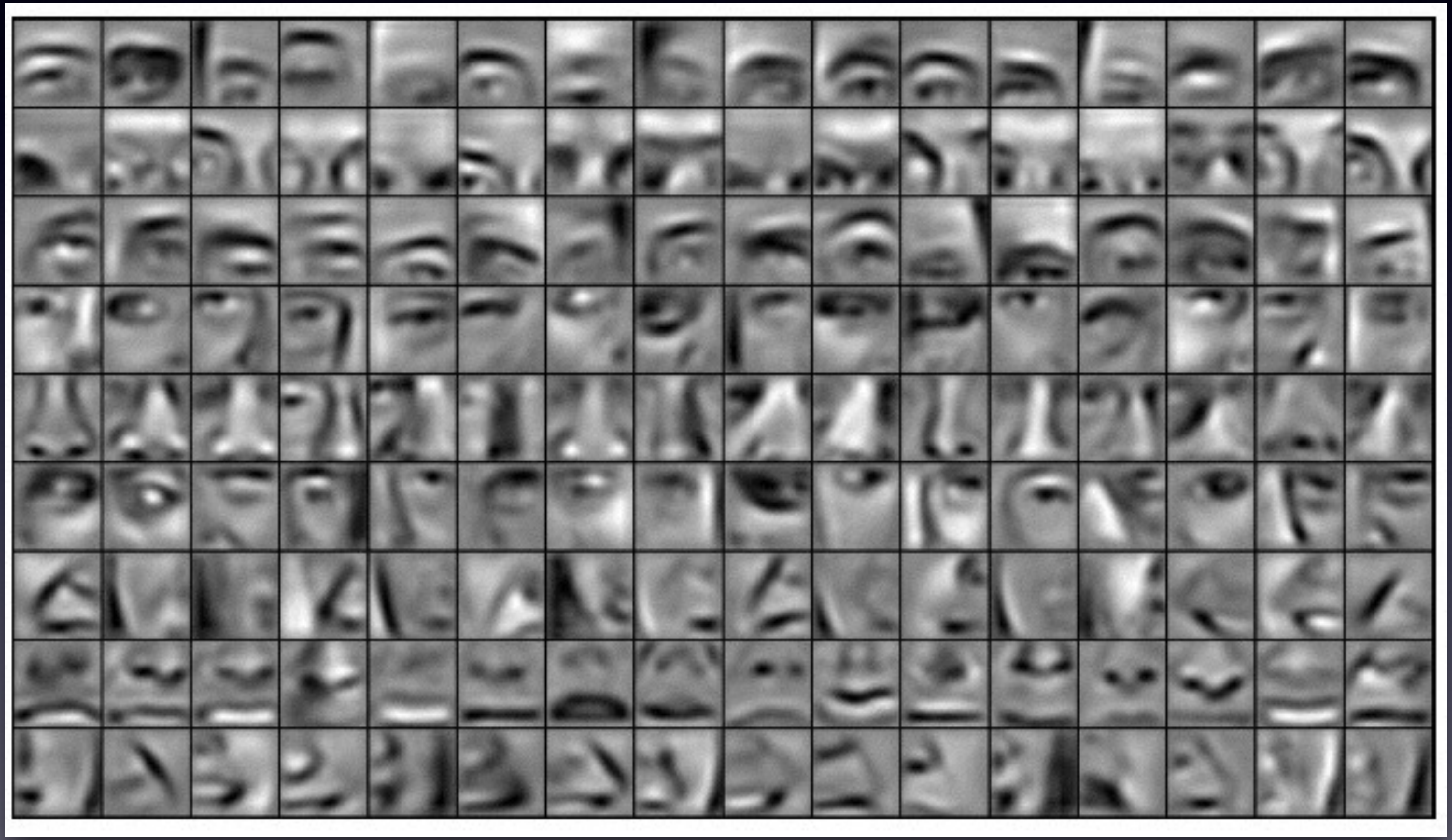


Low-level features

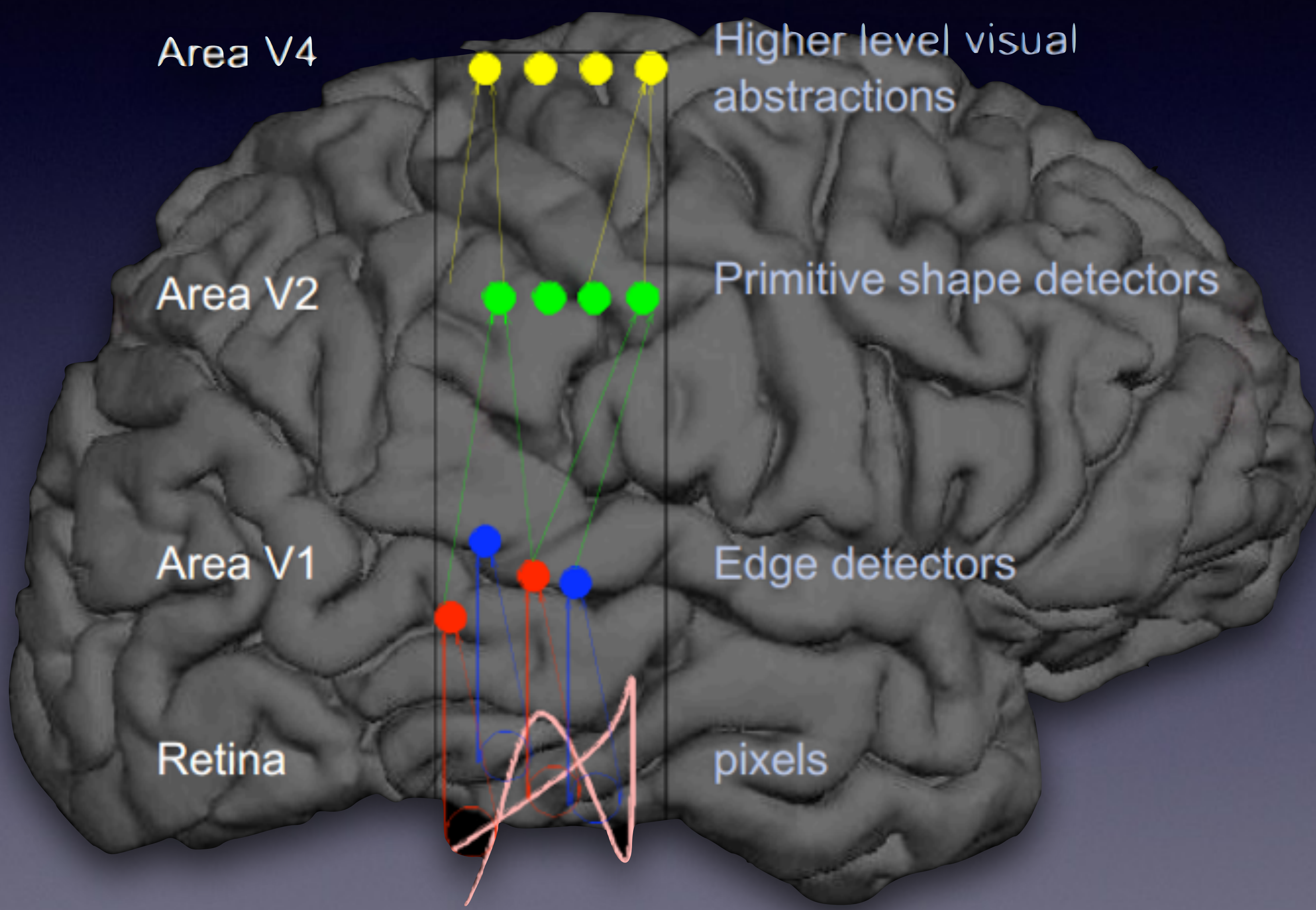
Used to be 'handcrafted'!



Higher-level features



Analogies human brain



e.g. [Cahieu et al. 2014]

Notebook: A Multilayer Perceptron in Keras