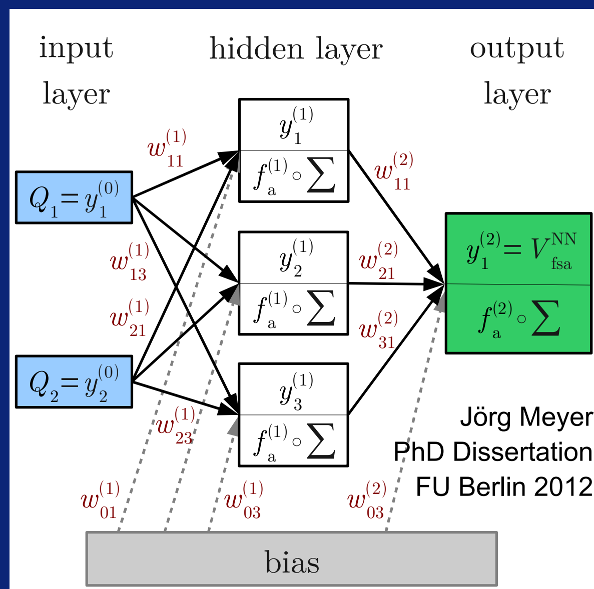
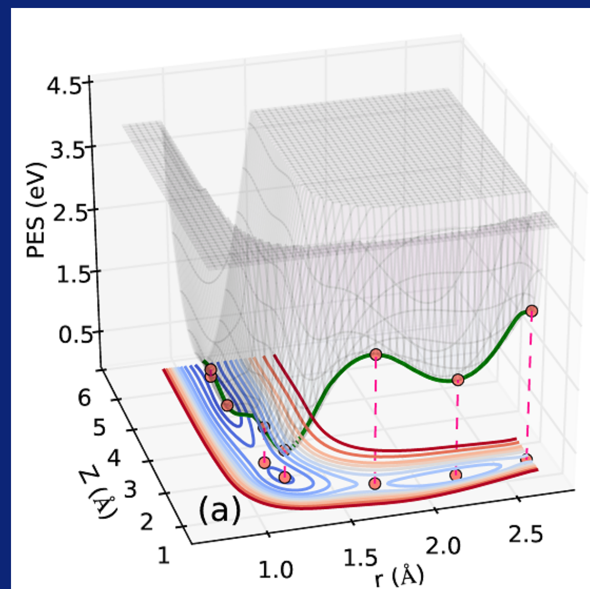
Machine learning in computational chemistry
Foundations and applications



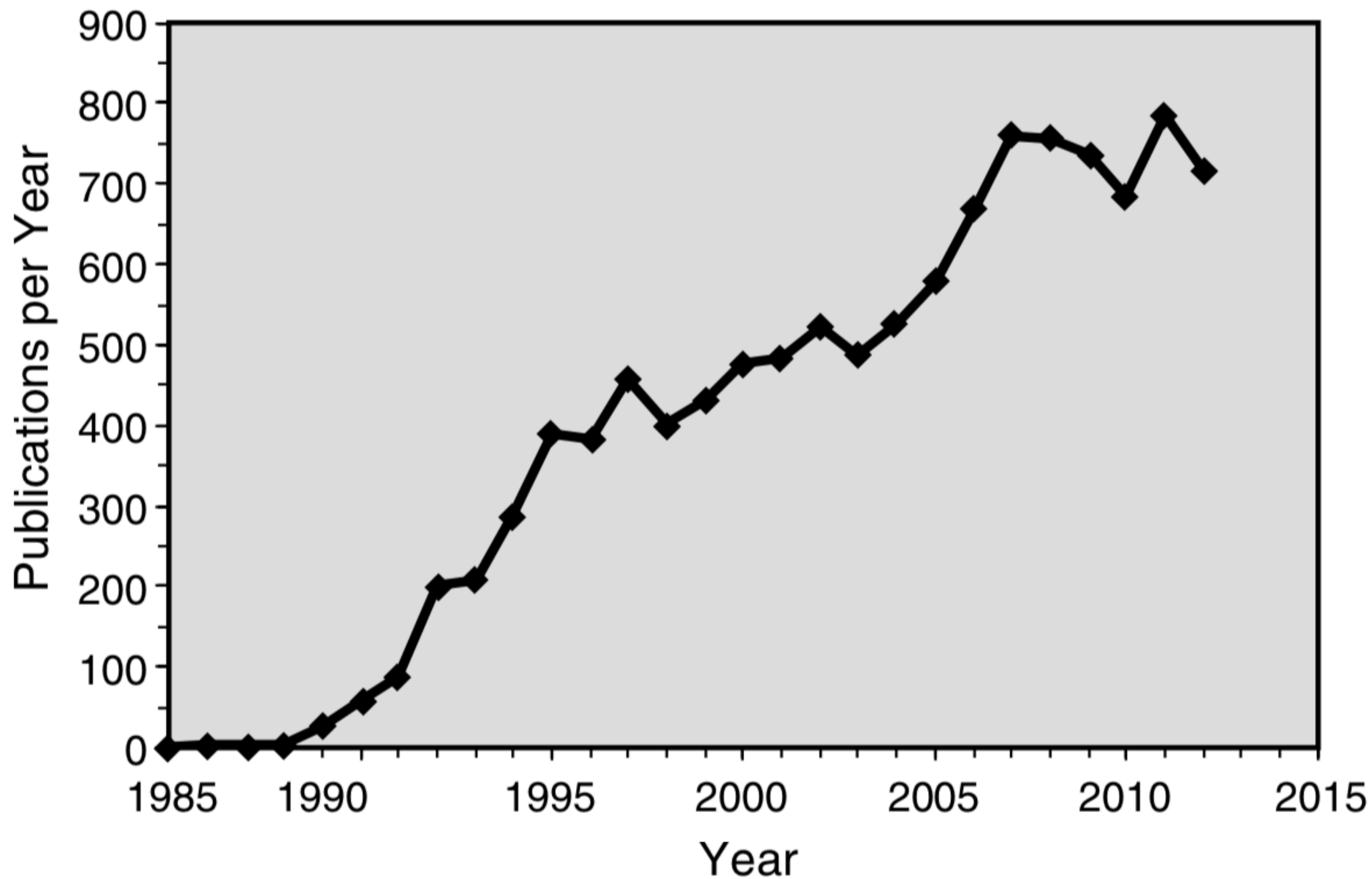


Jörg Meyer, Theoretical Chemistry
j.meyer@chem.leidenuniv.nl

Universiteit Leiden
The Netherlands

Winter School on
Theoretical Chemistry and Spectroscopy
Han-sur-Lesse, December 10 - 14

Neural Networks in Chemistry



J. Behler, *J. Phys.: Condens. Matter*. **26**, 183001 (2014).

Contents

1) **Neural Networks (NNs):**
   Structure & “Learning” (<1h)
   ***Hands-on:* Training a simple NN (45 min)**

2) NNs for potential energy surfaces:
   Coordinate representation (~1h)
   Hands-on: (Re-)Fitting a potential energy surface for O_2@Pd(100) (~2h)

3) Applications in gas-surface dynamics (<1h)

IHPCSS 2018

International High Performance Computing Summer School
http://www.ihpcss.org

MNIST realtime demo

Draw your number here

0123456789

Downsampled drawing: 7

First guess: 7

Second guess: 2

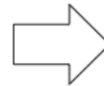
Layer visibility

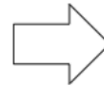
Input layer	Show
Convolution layer 1	Show
Downsampling layer 1	Show
Convolution layer 2	Show
Downsampling layer 2	Show

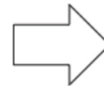
http://scs.ryerson.ca/~aharley/vis/conv/flat.html

Fun with neural networks (1)

Style Photo | Content Photo | Stylized Content





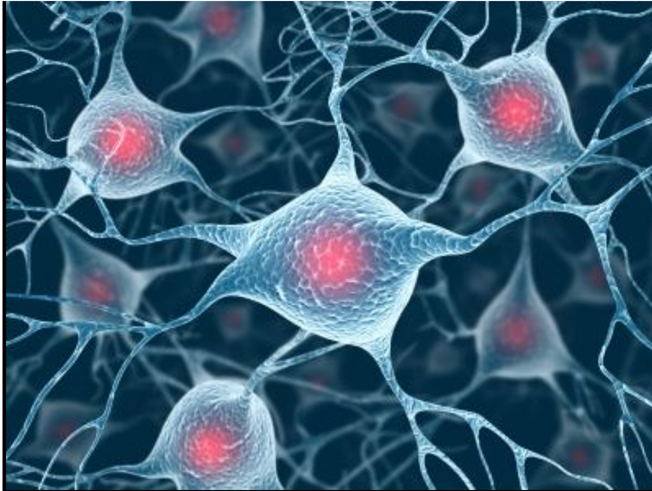


https://github.com/NVIDIA/FastPhotoStyle

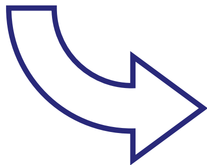
Fun with neural networks (2)

https://deepdreamgenerator.com/feed

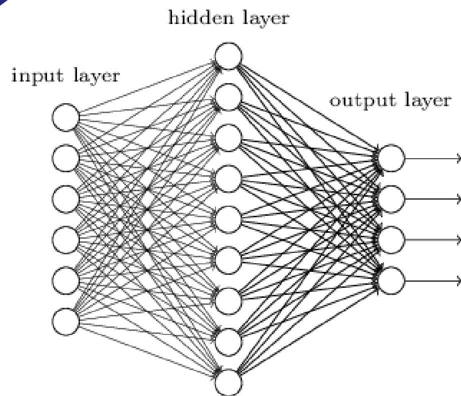
Inspired by biology



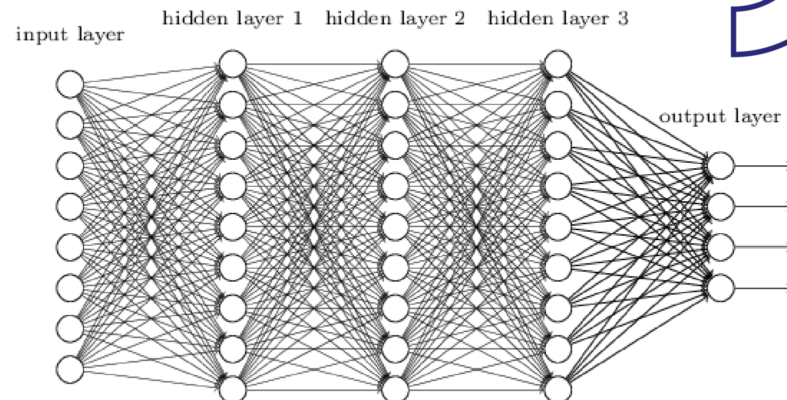
$$M = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

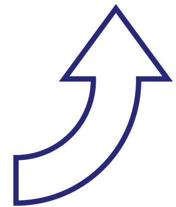


"Non-deep" feedforward neural network

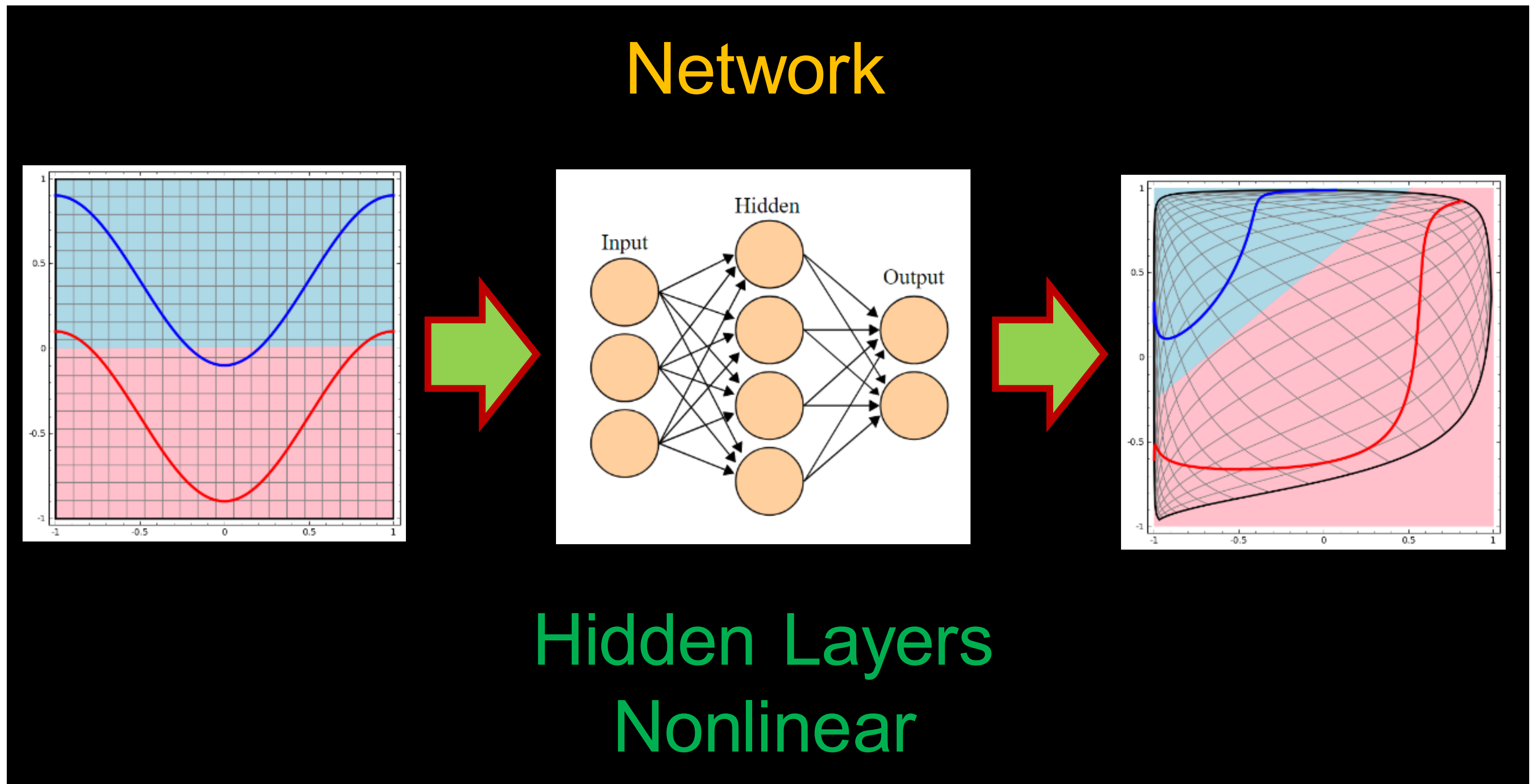


Deep neural network

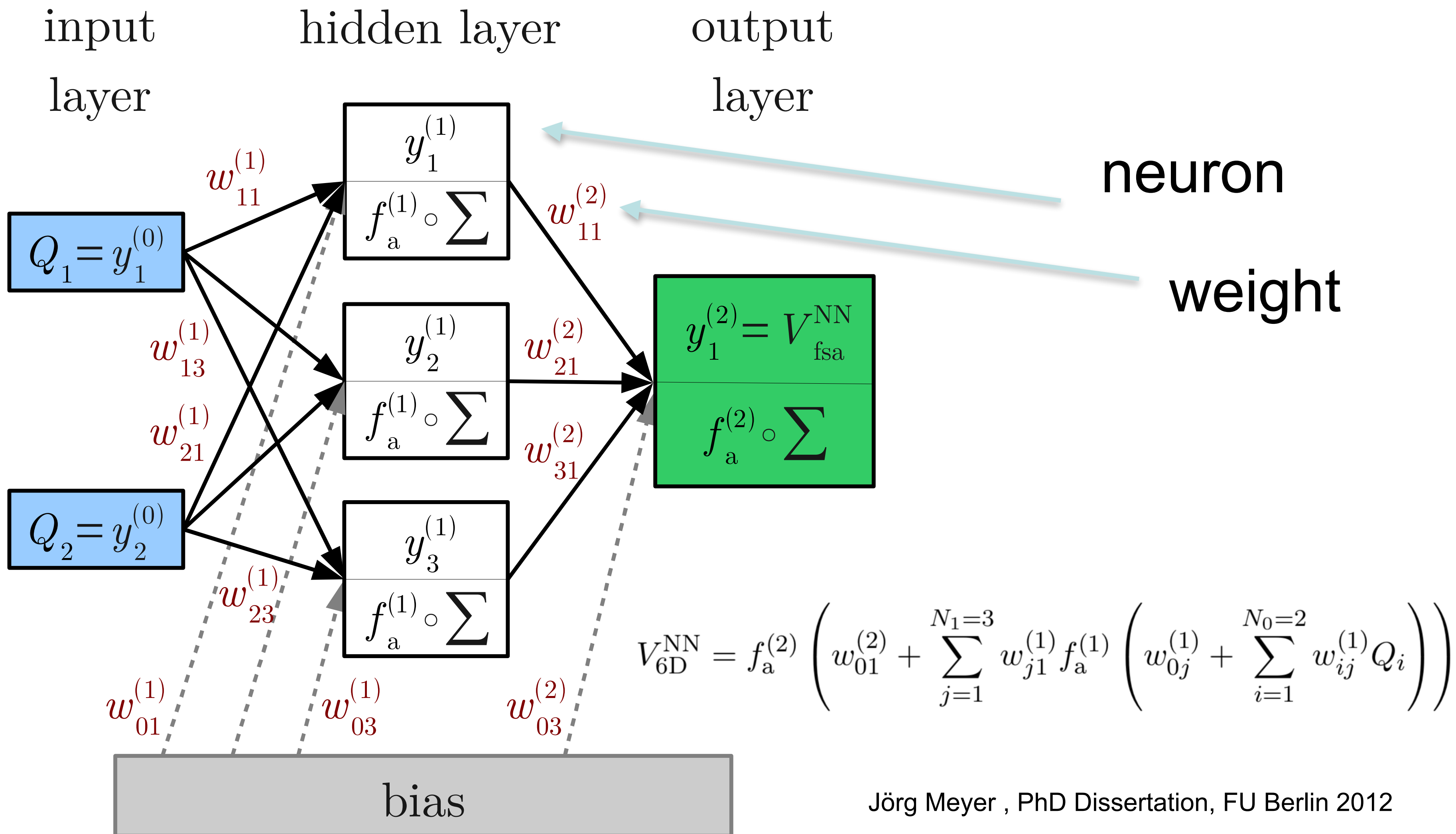




High-dimensional non-linear functions



NN architecture

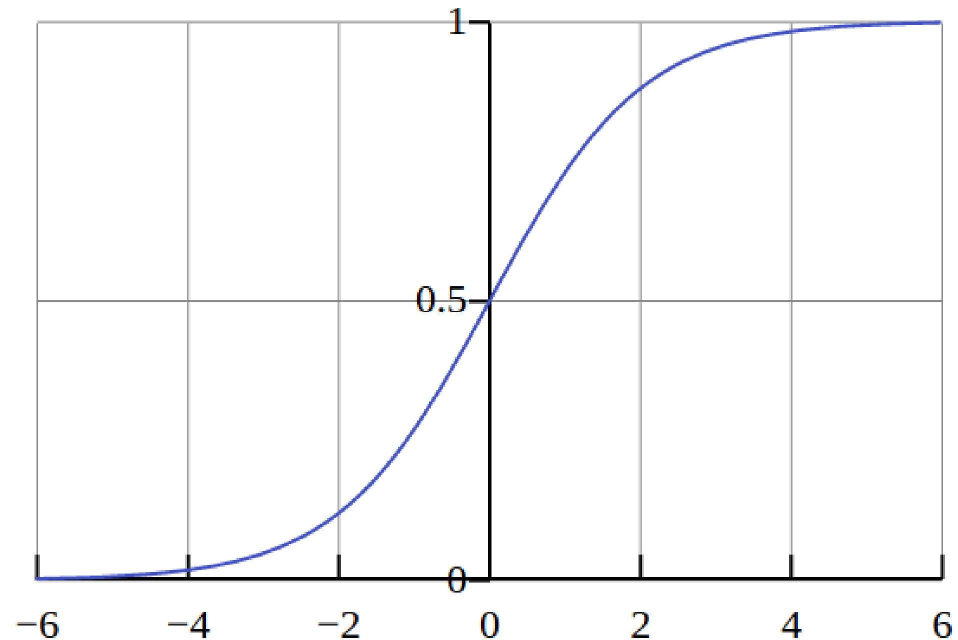


Activation function(s)

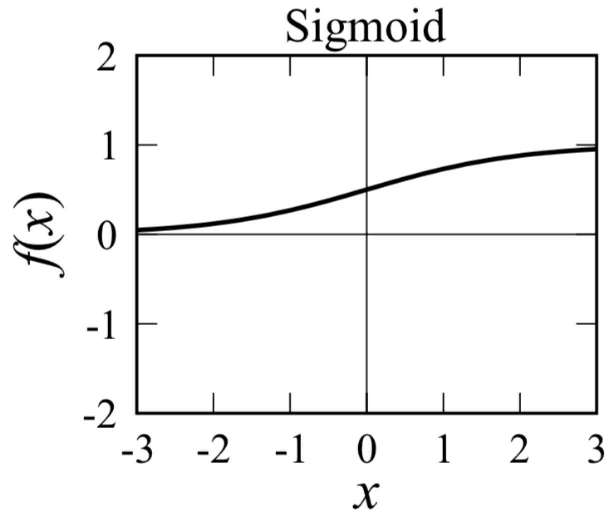
- Neurons apply activation functions at these summed inputs.
- Activation functions are typically non-linear.
- The sigmoid function is very commonly used activation function,

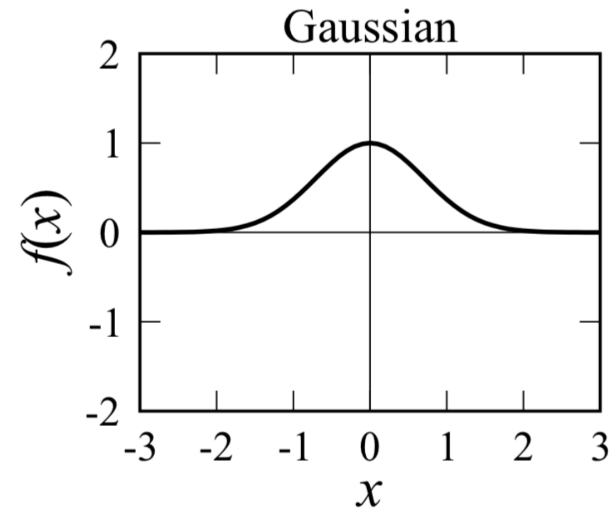
$$S(t) = \frac{1}{1 + e^{-t}}$$

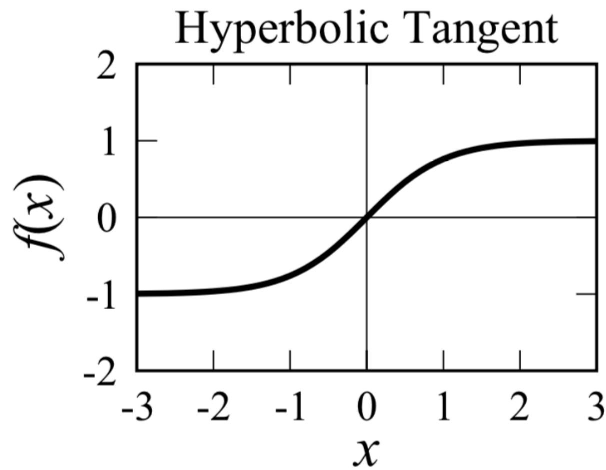
but also hyperbolic tangents.

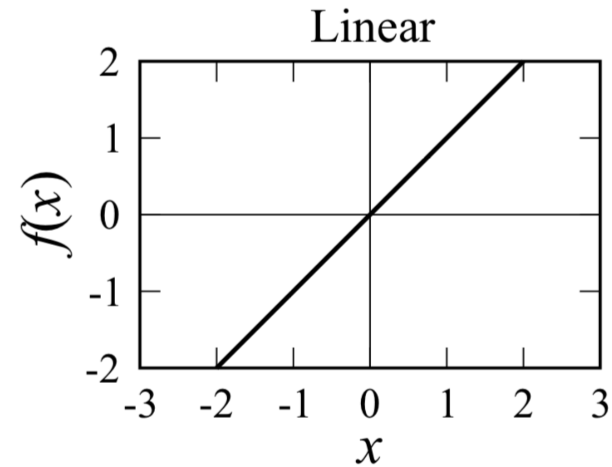


Activation function(s)



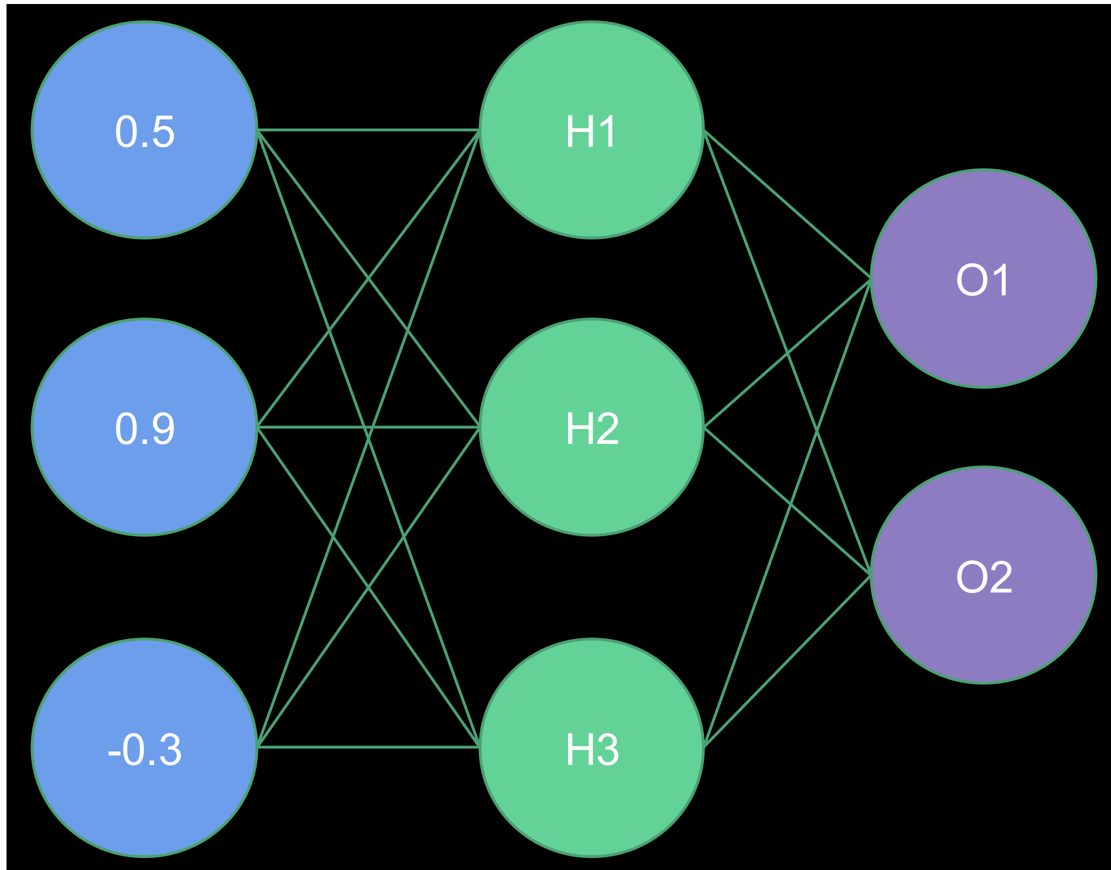






J. Behler, *J. Phys.: Condens. Matter*. **26**, 183001 (2014).

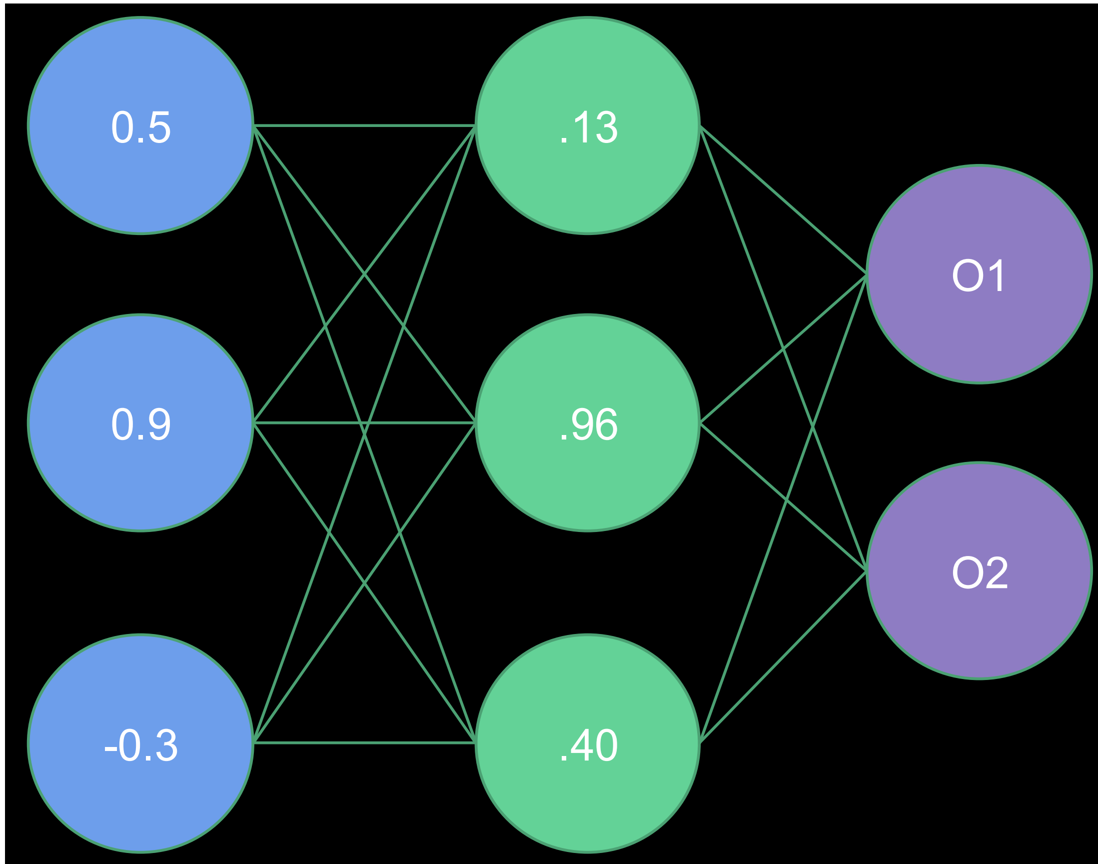
Forward propagation (1)



H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

Forward propagation (2)

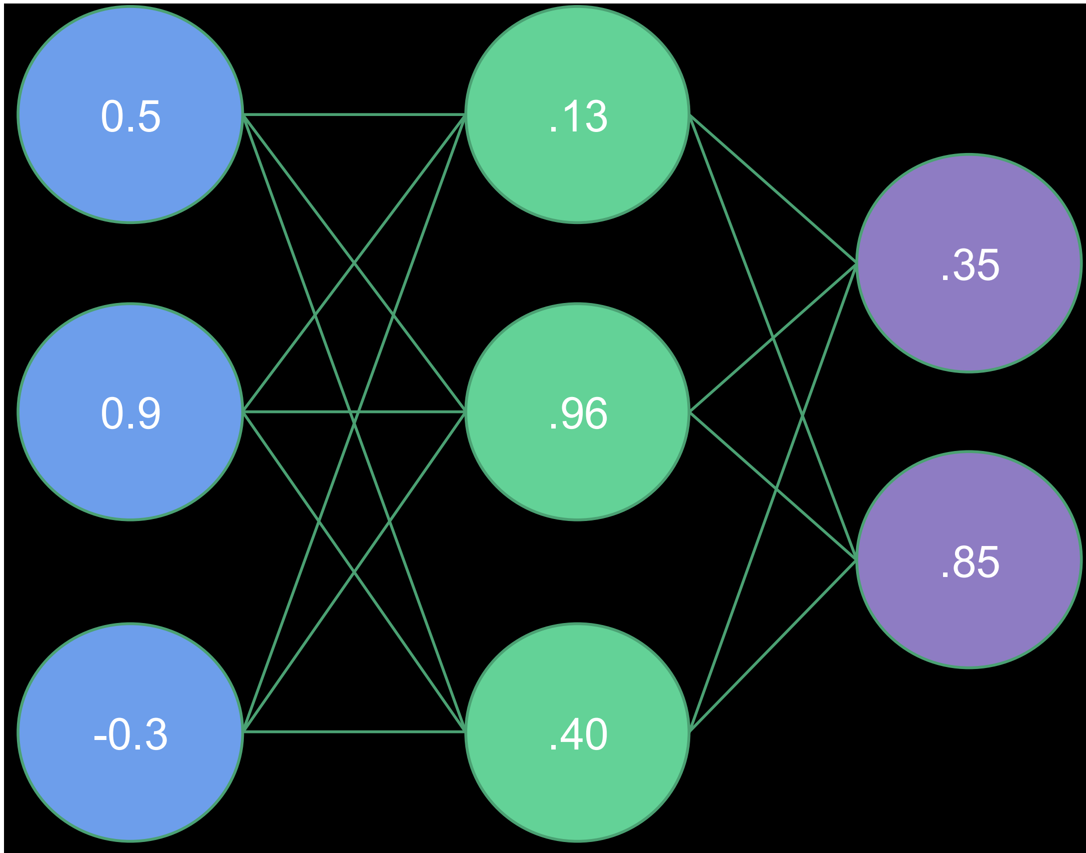


H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

H1 = Sigmoid(0.5 * 1.0 + 0.9 * -2.0 + -0.3 * 2.0) = Sigmoid(-1.9) = .13
H2 = Sigmoid(0.5 * 2.0 + 0.9 * 1.0 + -0.3 * -4.0) = Sigmoid(3.1) = .96
H3 = Sigmoid(0.5 * 1.0 + 0.9 * -1.0 + -0.3 * 0.0) = Sigmoid(-0.4) = .40

Forward propagation (3)



H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

O1 Weights = (-3.0, 1.0, -3.0)
O2 Weights = (0.0, 1.0, 2.0)

O1 = Sigmoid(.13 * -3.0 + .96 * 1.0 + .40 * -3.0) = Sigmoid(-.63) = .35

O2 = Sigmoid(.13 * 0.0 + .96 * 1.0 + .40 * 2.0) = Sigmoid(1.76) = .85

Using matrices

H1 Weights = (1.0, -2.0, 2.0)
H2 Weights = (2.0, 1.0, -4.0)
H3 Weights = (1.0, -1.0, 0.0)

Hidden Layer Weights | Inputs | Hidden Layer Outputs

$$Sig\left(\begin{bmatrix} 1.0 & -2.0 & 2.0 \\ 2.0 & 1.0 & -4.0 \\ 1.0 & -1.0 & 0.0 \end{bmatrix} * \begin{bmatrix} 0.5 \\ 0.9 \\ -0.3 \end{bmatrix}\right) = Sig\left(\begin{bmatrix} -1.9 & 3.1 & -0.4 \end{bmatrix}\right) = \begin{bmatrix} .13 & .96 & 0.4 \end{bmatrix}$$

This can be done very efficiently on GPUs nowadays…

Biases

- It is also very useful to be able to offset our inputs by some constant. You can think of this as centering the activation function, or translating the solution.
  We will call this constant the bias, and it there will often be one value per layer.
- Our math for the previously calculated layer now looks like this with ***b=0.1***:

Hidden Layer Weights, Inputs, Hidden Layer Outputs

$$\mathrm{Sig}\left(\begin{bmatrix} 1.0 & -2.0 & 2.0 \\ 2.0 & 1.0 & -4.0 \\ 1.0 & -1.0 & 0.0 \end{bmatrix} * \begin{bmatrix} 0.5 \\ 0.9 \\ -0.3 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.1 \\ 0.1 \end{bmatrix}\right) = \mathrm{Sig}\left(\begin{bmatrix} -1.8 & 3.2 & -0.3 \end{bmatrix}\right) = \begin{bmatrix} .14 & .96 & 0.4 \end{bmatrix}$$

Training

- So how do we find these magic weights? We want to minimize the error on our training data. Given labeled inputs, select weights that generate the smallest average error on the outputs.
- We know that the output is a function of the weights:

$$E(w_1,w_2,w_3,...)$$

So to figure out which way, and how much, to push any particular weight, say w_3, we want to calculate

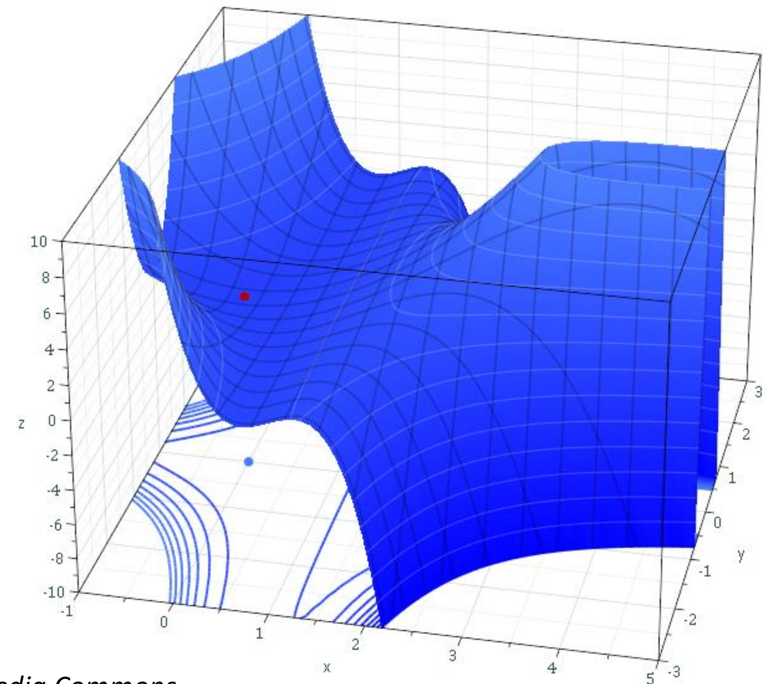
Backpropagation

- If we use the chain rule repeatedly across layers we can work our way backwards from the output error through the weights, adjusting them as we go.
  Note that this is where the requirement that activation functions must have nicely behaved derivatives comes from.
- This technique makes the weight inter-dependencies much more tractable. An elegant perspective on this can be found from Chris Olah at
  http://colah.github.io/posts/2015-08-Backprop
- With basic calculus you can readily work through the details. You can find an excellent explanation from the renowned “3Blue1Brown” at
  https://www.youtube.com/watch?v=Ilg3gGewQ5U

Solvers (1)

- Backpropagation leaves us with potentially many millions of non-linear equations for real-world networks to solve.

- Fortunately, this isn't a new problem created by deep learning, so we have options from the world of numerical methods.



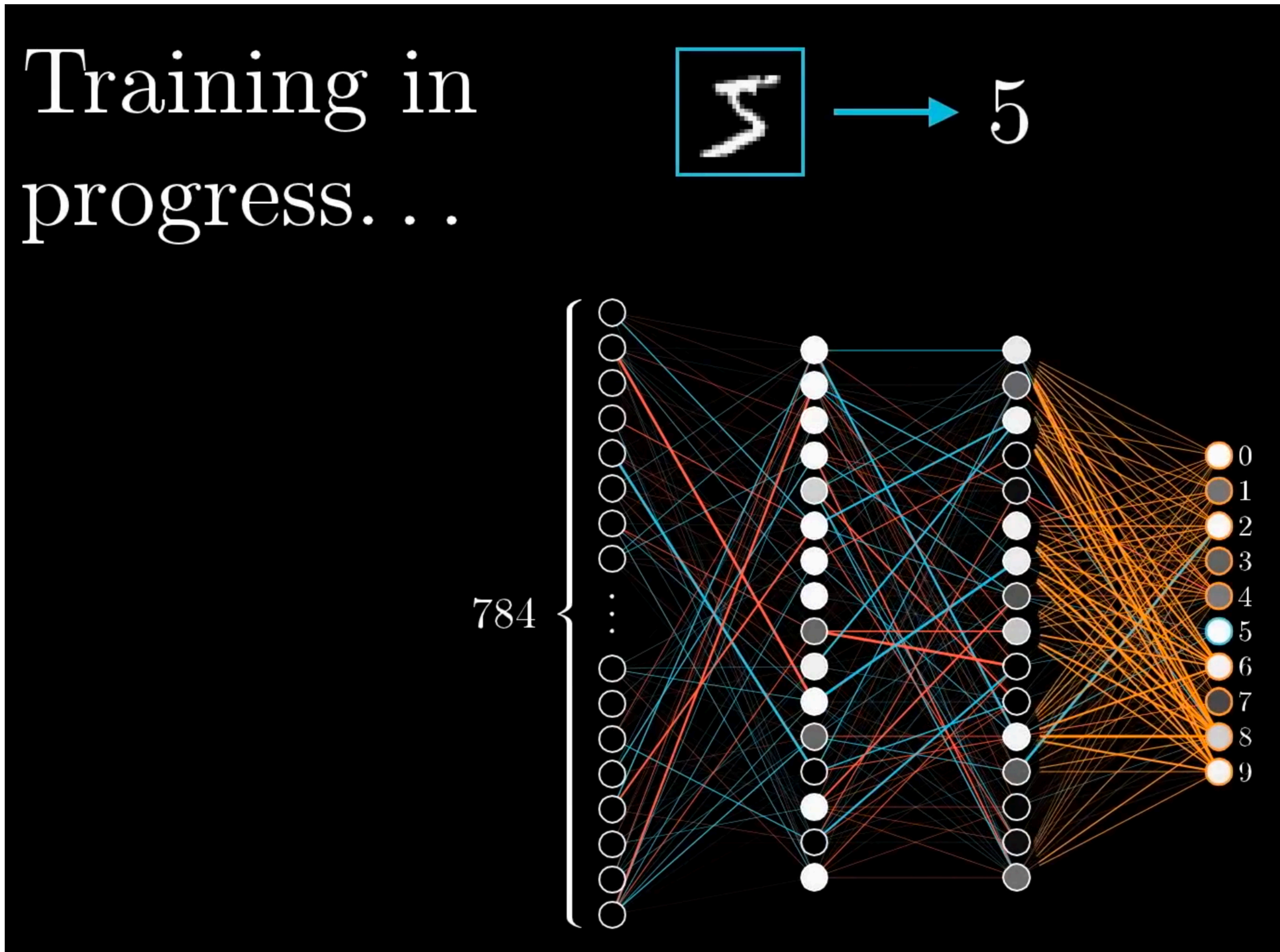
Wikipedia Commons

- The standard has been *gradient descent*. Methods, often similar, have arisen that perform better for machine learning applications. In google's TensorFlow package they can be easily changed due to its modular structure.

Solvers (2)

- Most interesting recent methods incorporate ***momentum*** to help get over a local minimum. Momentum and ***step size*** are the two hyperparameters we will encounter later.
- Nevertheless, we don't expect to ever find the actual global minimum.
- We could/should find the error for all the training data before updating the weights (an ***epoch***). However it is usually much more efficient to use a stochastic approach, sampling a random subset of the data, updating the weights, and then repeating with another ***mini-batch***.

Training in progress



https://www.youtube.com/watch?v=IIg3gGewQ5U

Implementations

Package	Applications	Language	Strengths
TensorFlow	Neural Nets	Python, C++	Very popular.
Caffe	Neural Nets	Python, C++	Many research projects and publications. 2.0 more TF-like.
Spark MLLIB	Classification, Regression, Clustering, etc.	Python, Scala, Java, R	Very scalable. Widely used in serious applications.
Scikit-Learn	Classification, Regression, Clustering	Python	
cuDNN	Neural Nets	C++, GPU-based	Used in many other frameworks: TF, Caffe, etc.
Theano	Neural Nets	Python	Lower level numerical routines. NumPy-esque.
Torch	Neural Nets	Lua (PyTorch=Python)	Dynamic graphs (variable length input/output) good for RNN.
Keras	Neural Nets	Python (on top of TF, Theano)	Higher level approach.
Digits	Neural Nets	"Caffe", GPU-based	Used with other frameworks (only Caffe at moment).

chemistry specific (for PES construction):
e.g. AMP https://bitbucket.org/andrewpeterson/amp