Deep Learning
-an introduction-

Luca Bortolussi

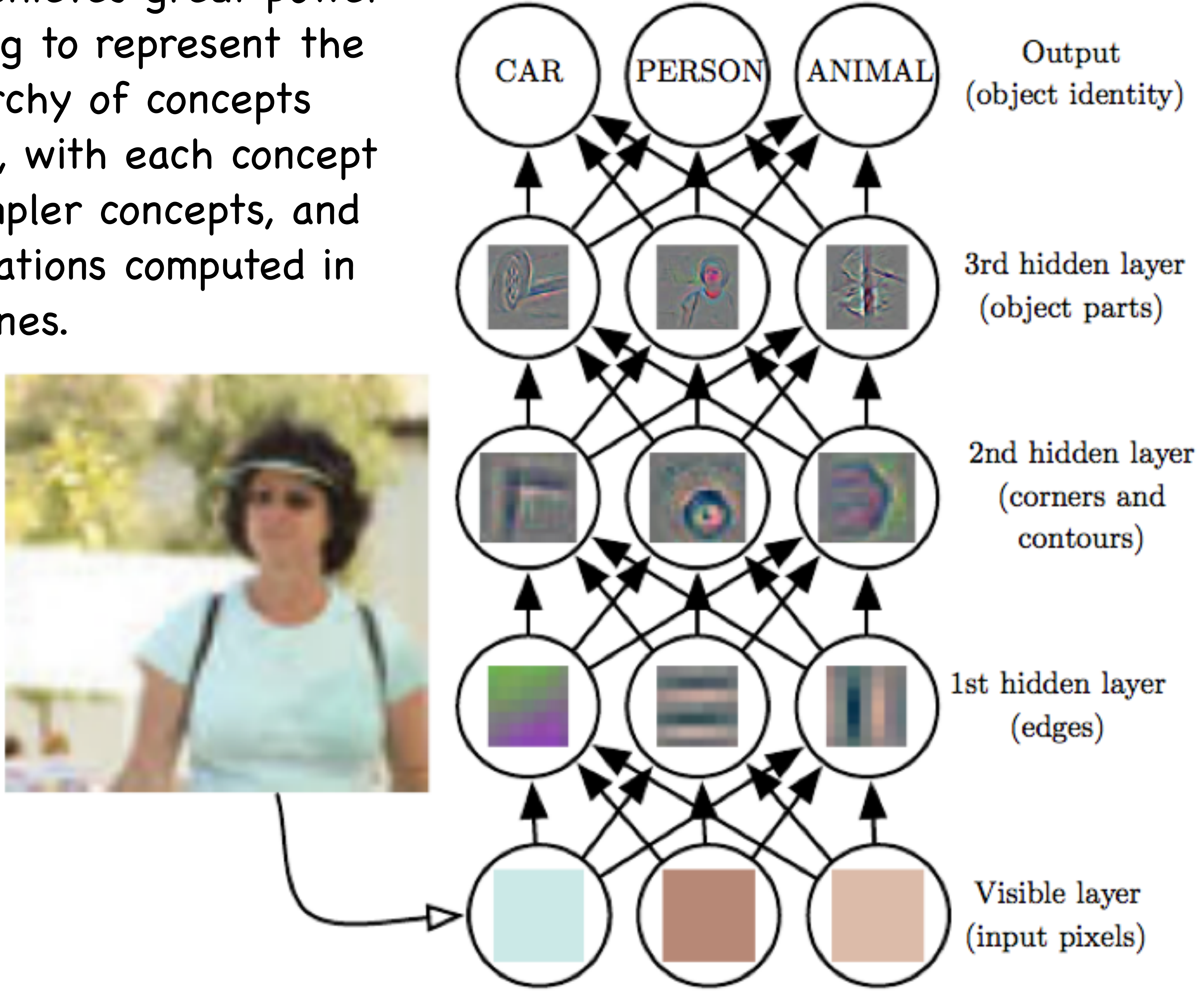
DMG, University of Trieste, IT
Modelling and Simulation, Saarland University, DE

DSSC, Summer Semester 2018

What is deep learning?

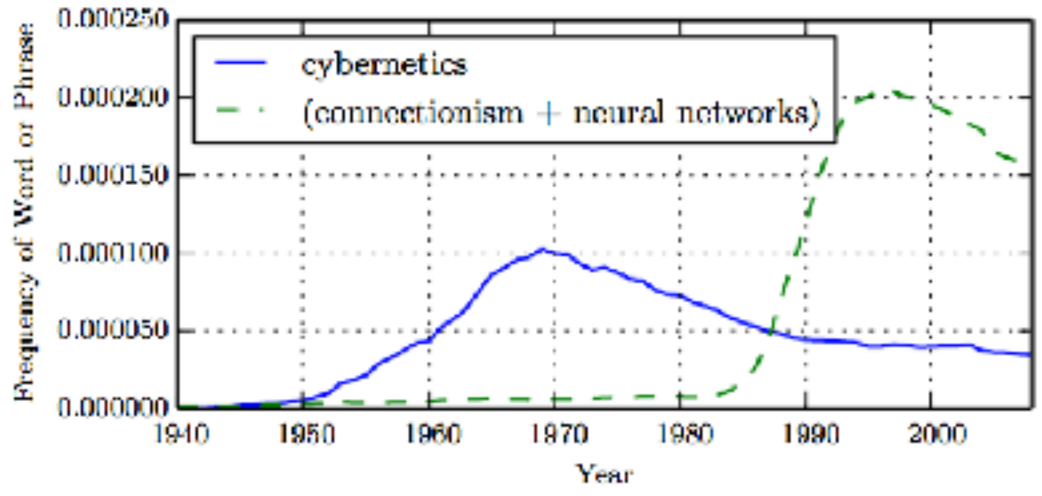
Deep learning is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts [representation learning], with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones.

The mainstream tool in deep learning are (Artificial) **Neural Networks**, also called **Multi-Layer Perceptrons**.

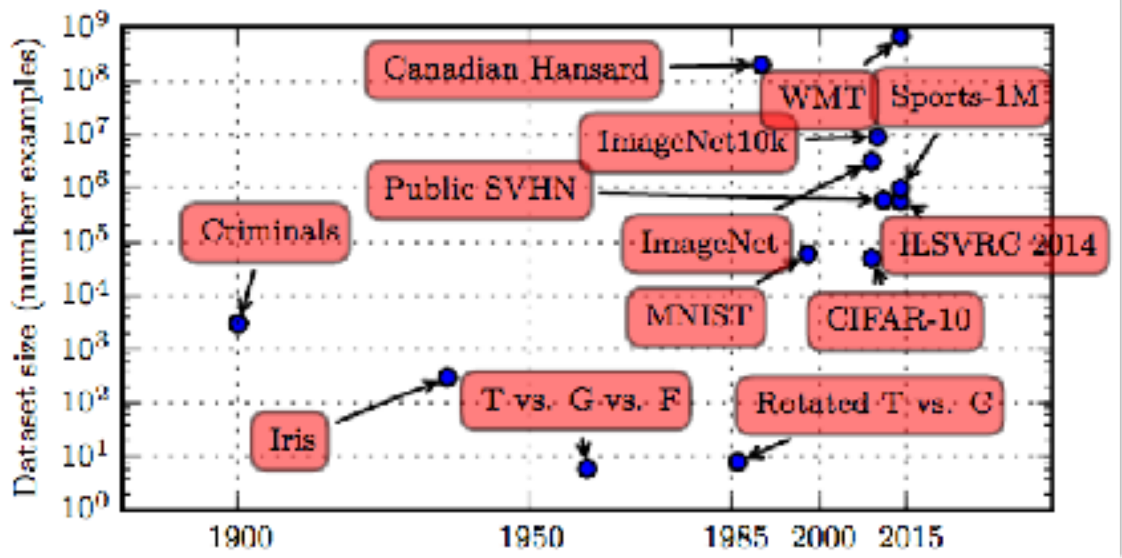

CAR
PERSON
ANIMAL
Output (object identity)
3rd hidden layer (object parts)
2nd hidden layer (corners and contours)
1st hidden layer (edges)
Visible layer (input pixels)

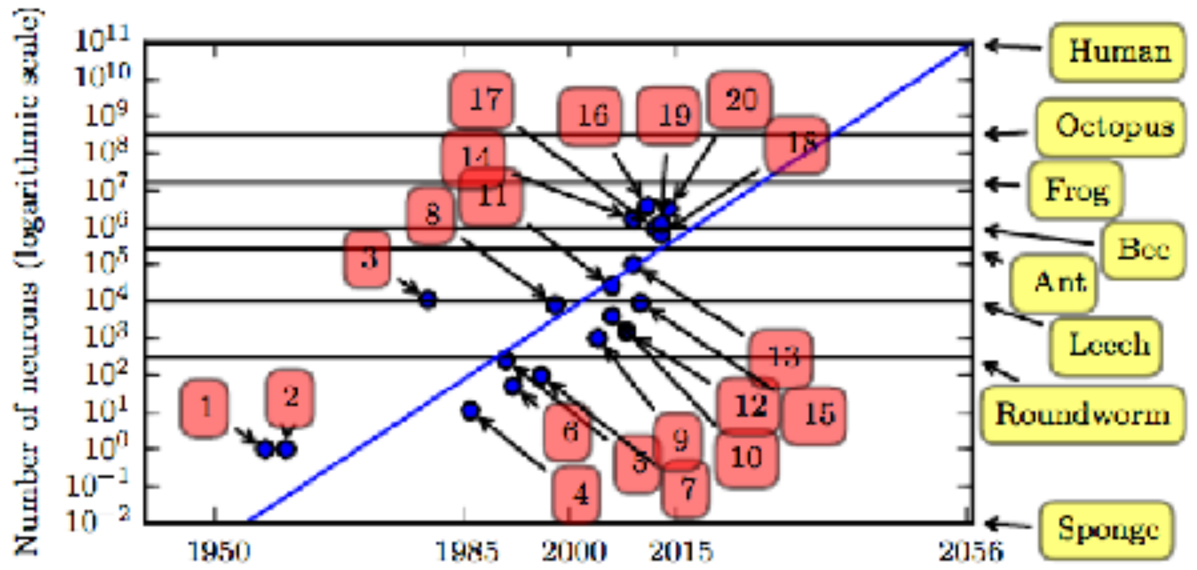
Why deep learning?

Most of the concepts and ideas of deep learning have a long tradition. This is the third wave of Neural Networks. Why now?



1. Availability of **very large datasets**
2. Availability of large computational power, in particular **GPU clusters**, which allows us to train larger and deeper models.
3. Some improvements in the science and technology of NN (ReLU, improved SGD, improved regularisation).

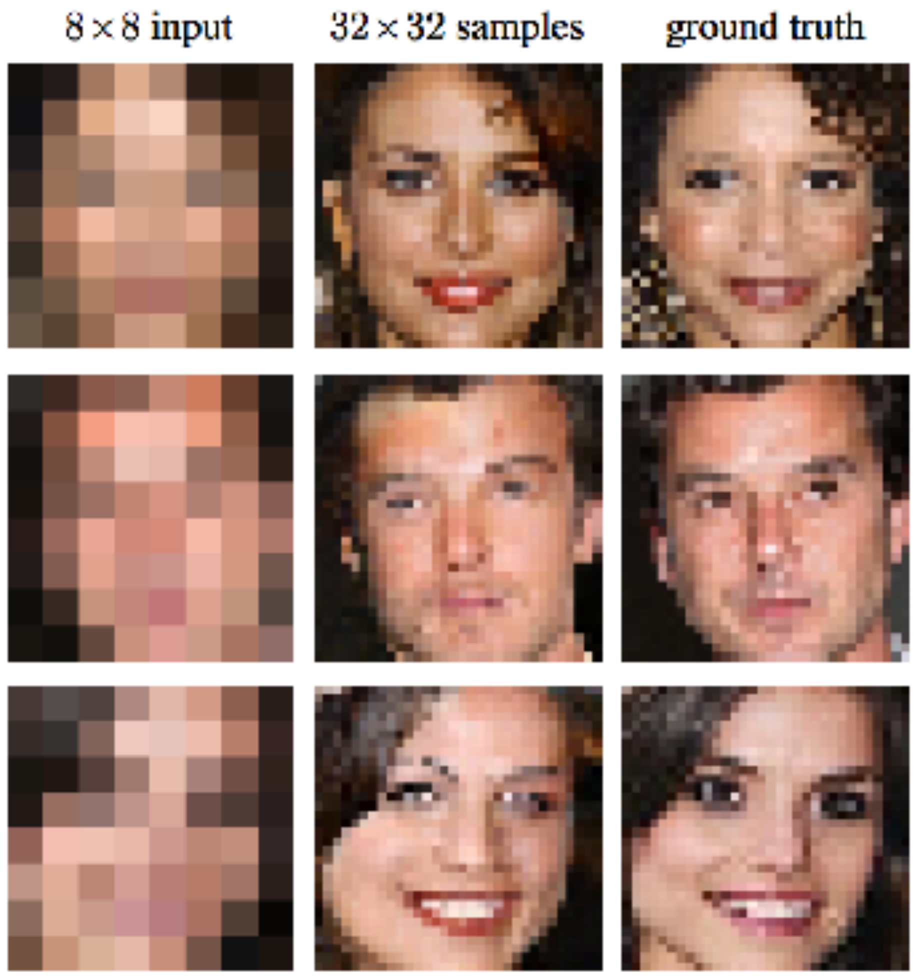


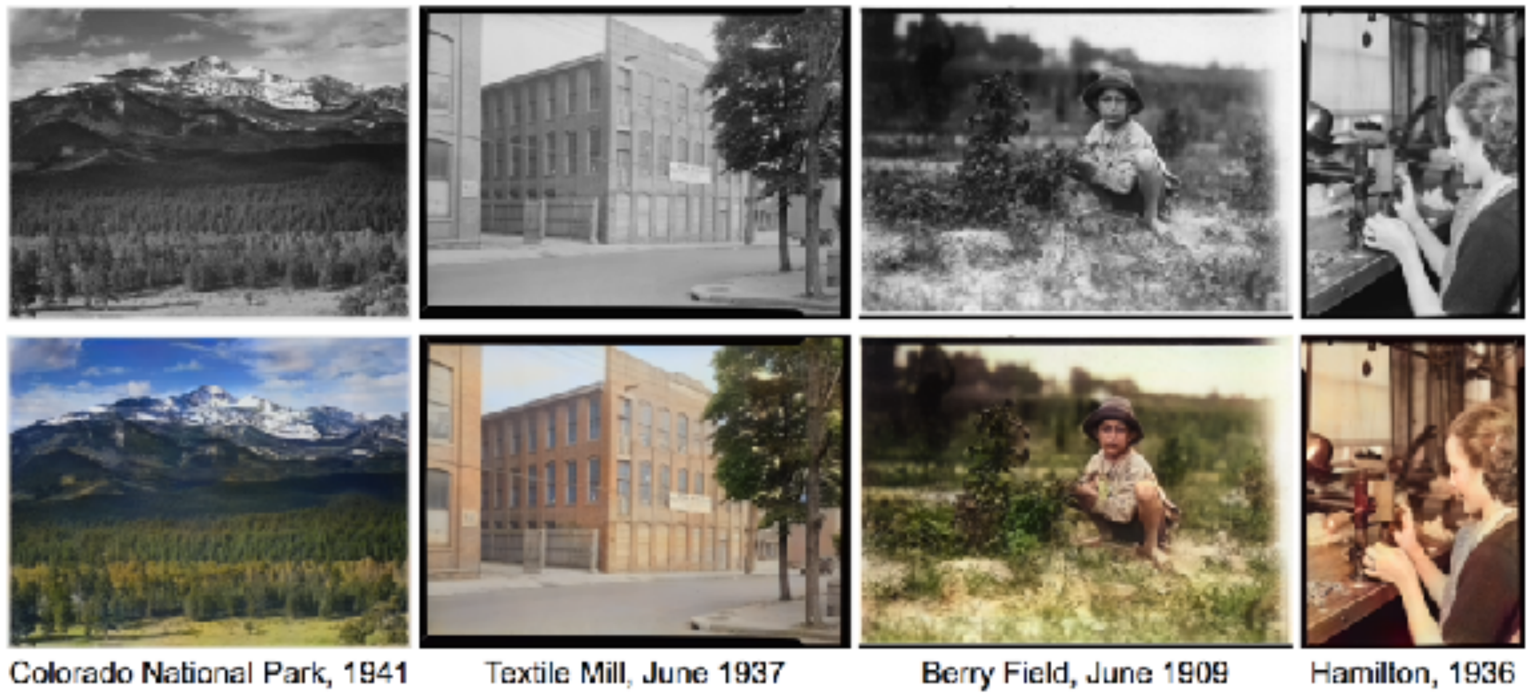


As of 2016, a rough rule of thumb is that a supervised deep learning algorithm will generally achieve acceptable performance with around 5,000 labeled examples per category, and will match or exceed human performance when trained with a dataset containing at least 10 million labeled examples

Deep Learning: reshaping AI

Deep Learning has many impressive achievements in computer vision

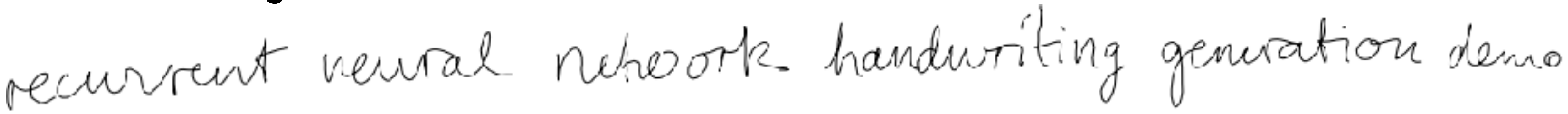




Colouring black and white pictures

Another field in which deep learning is having a profound impact is natural language processing

Face reconstruction from low resolution images

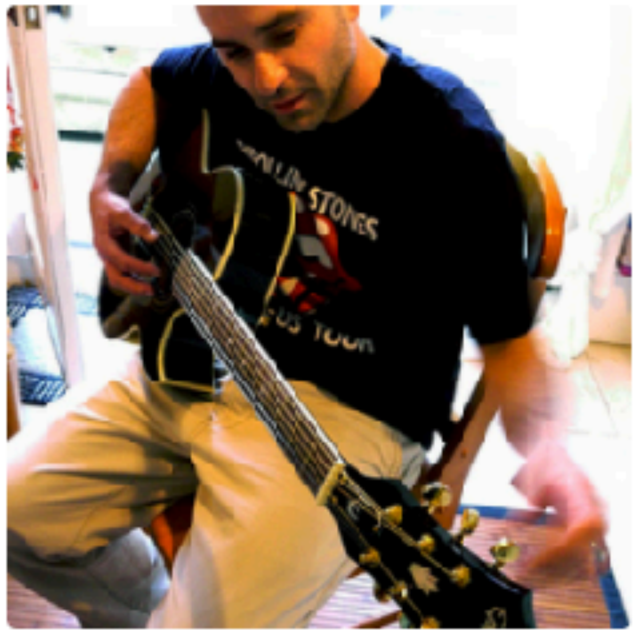


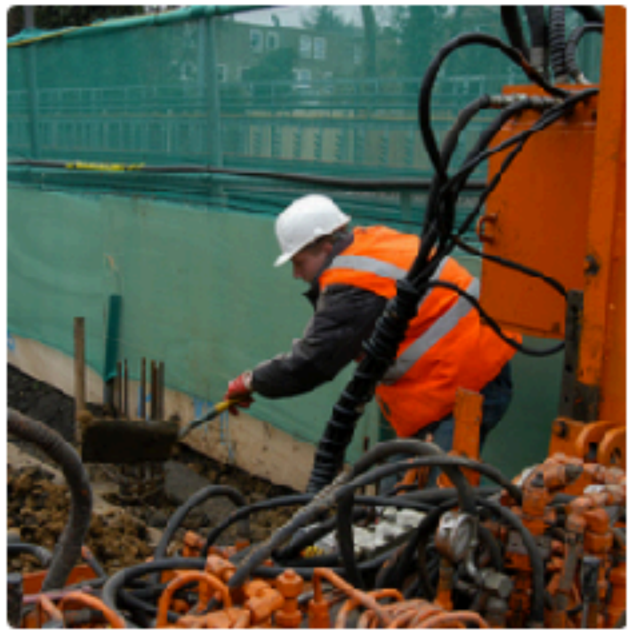
recurrent neural network handwriting generation demo

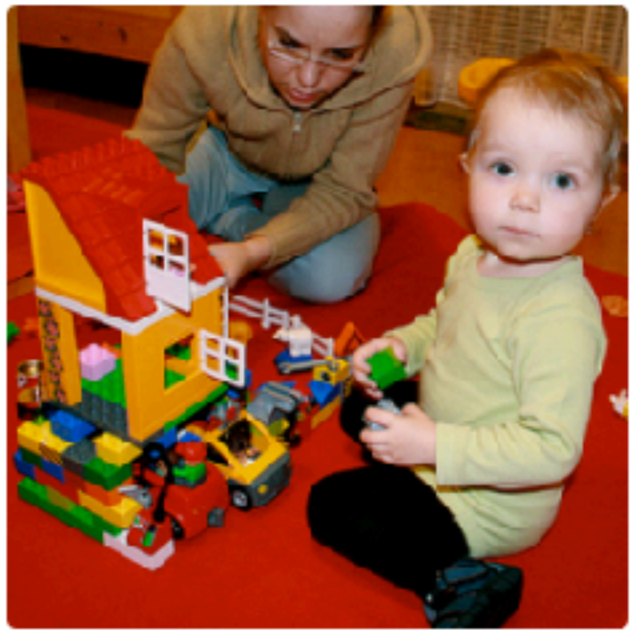
Handwritten text generated by deep learning

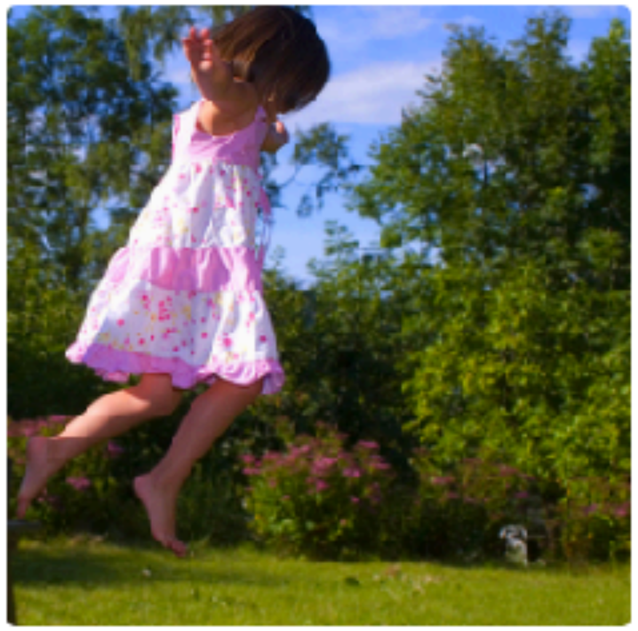
Deep Learning: reshaping AI

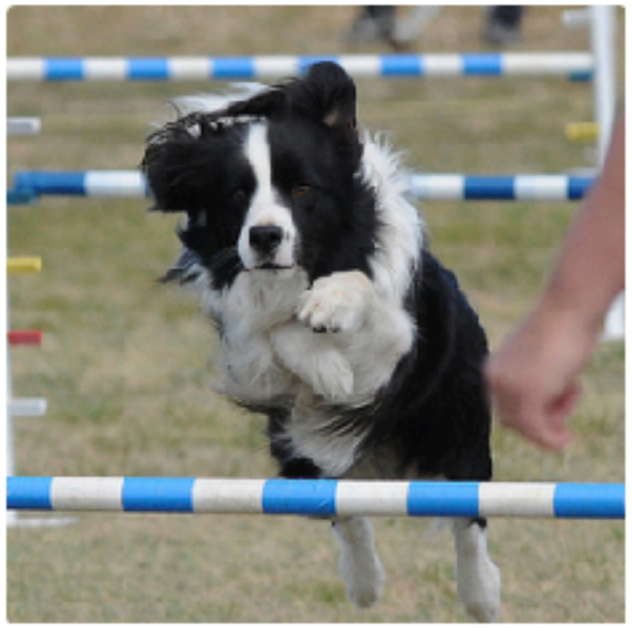
Image recognition and automatic captioning

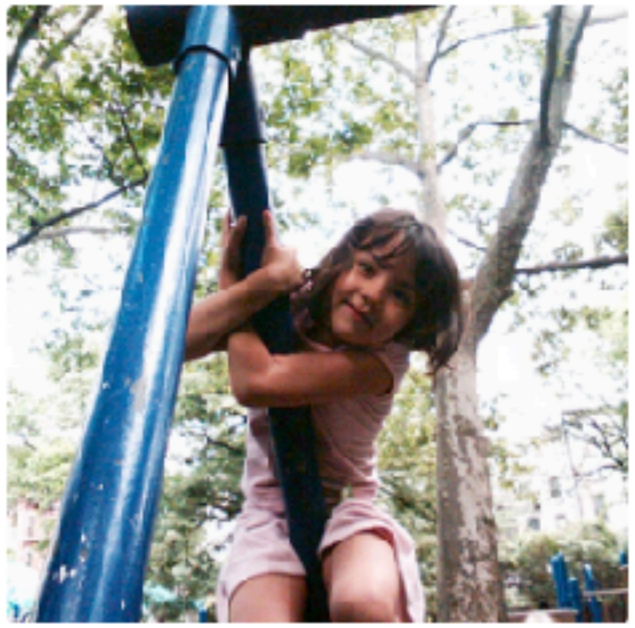

"man in black shirt is playing guitar."


"construction worker in orange safety vest is working on road."


"two young girls are playing with lego toy."

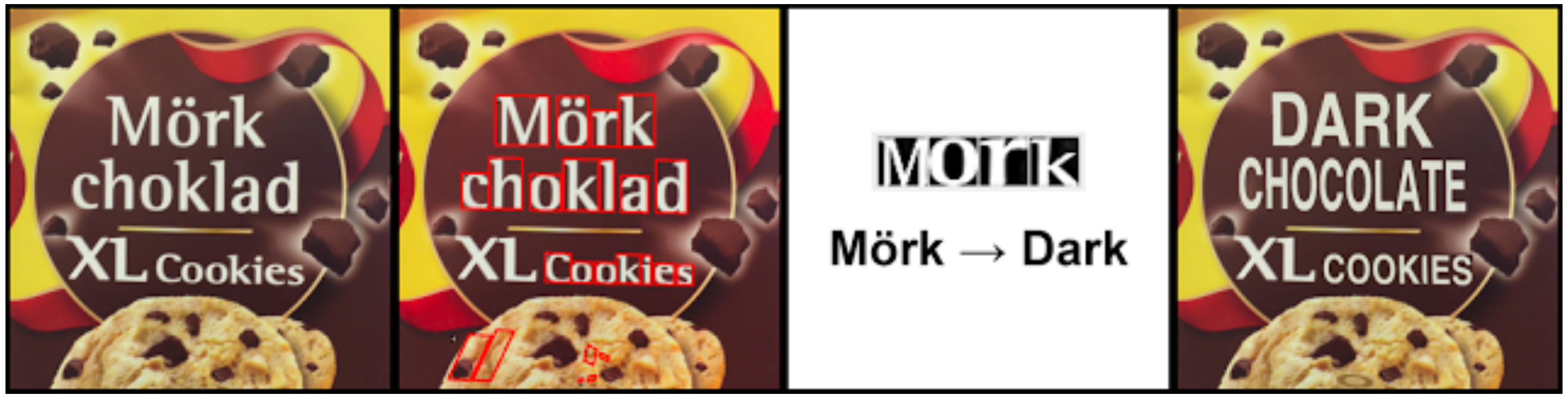

"girl in pink dress is jumping in air."


"black and white dog jumps over bar."


"young girl in pink shirt is swinging on swing."

Deep Learning: reshaping AI

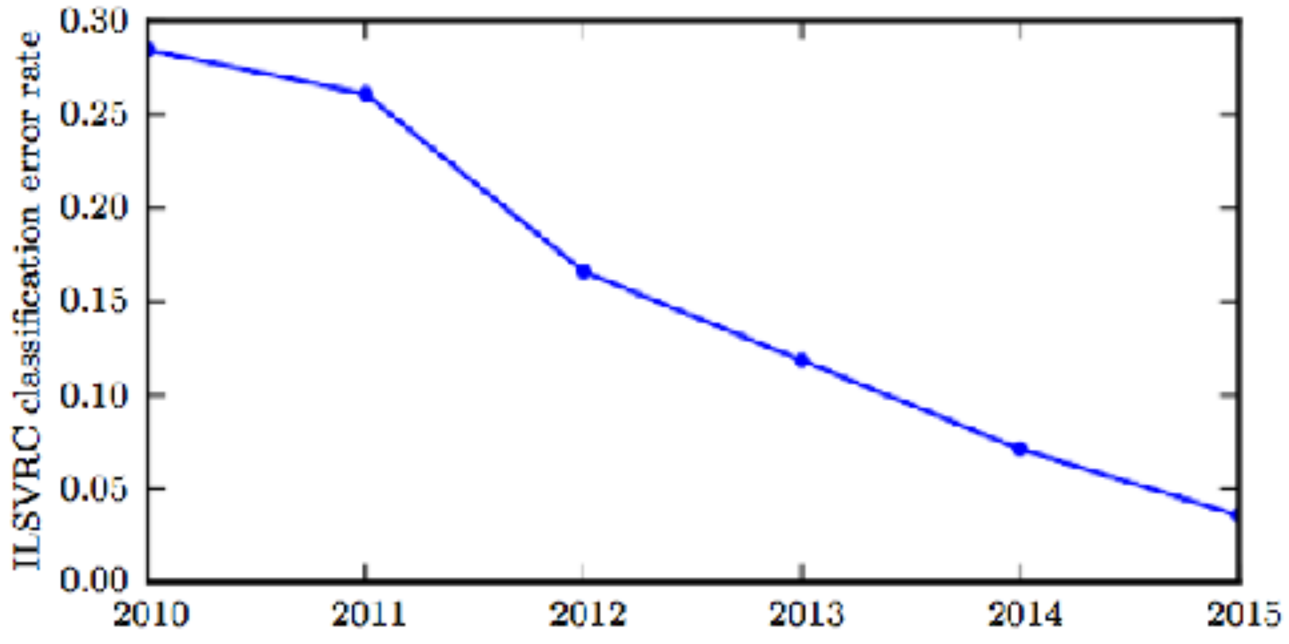
Automatic text recognition and translation in images



Advanced image recognition in automatically controlled systems...

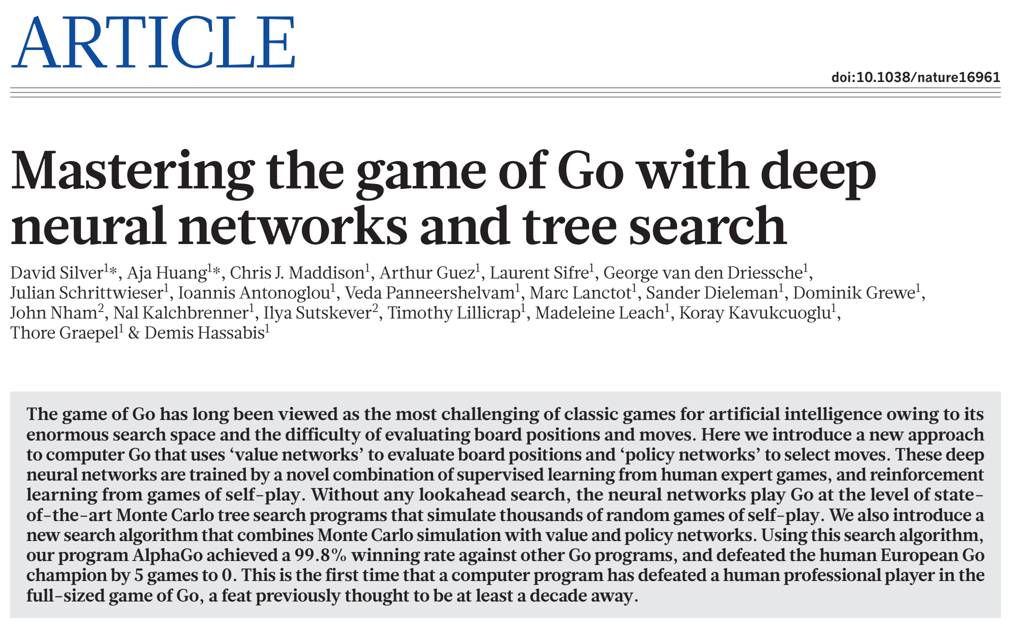
Deep Learning: success stories

ImageNet Large Scale Visual Recognition Challenge (ILSVRC), recognising more than 1000 different kinds of objects (superhuman performance, i.e. below average human error rate, around 4%)



Other success stories:

- Speech recognition
- Pedestrian detection and image segmentation
- Traffic sign classification (superhuman)
- Image captioning and description
- Machine translation
- Neural Turing machines and self-programming
- Reinforcement Learning (AlphaGO)



ARTICLE

doi:10.1038/nature16961

Mastering the game of Go with deep neural networks and tree search

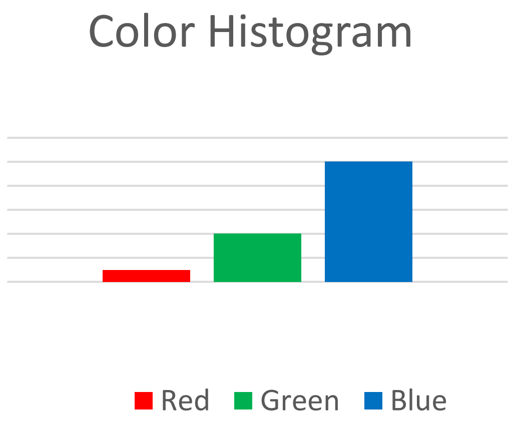
David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. Here we introduce a new approach to computer Go that uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of state-of-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play. We also introduce a new search algorithm that combines Monte Carlo simulation with value and policy networks. Using this search algorithm, our program AlphaGo achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion by 5 games to 0. This is the first time that a computer program has defeated a human professional player in the full-sized game of Go, a feat previously thought to be at least a decade away.

Features and Representation Learning

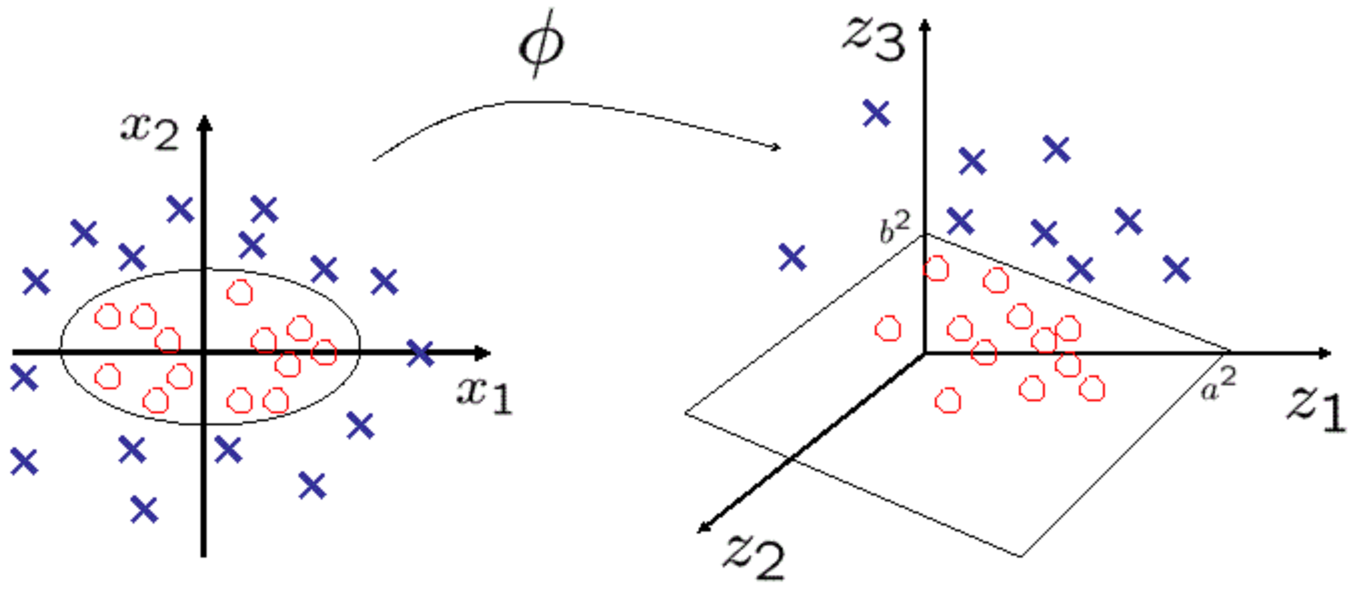
x

$\xrightarrow{\text{Extract features}}$



$\xrightarrow{\text{build hypothesis}}$ $y = w^T\phi(x)$

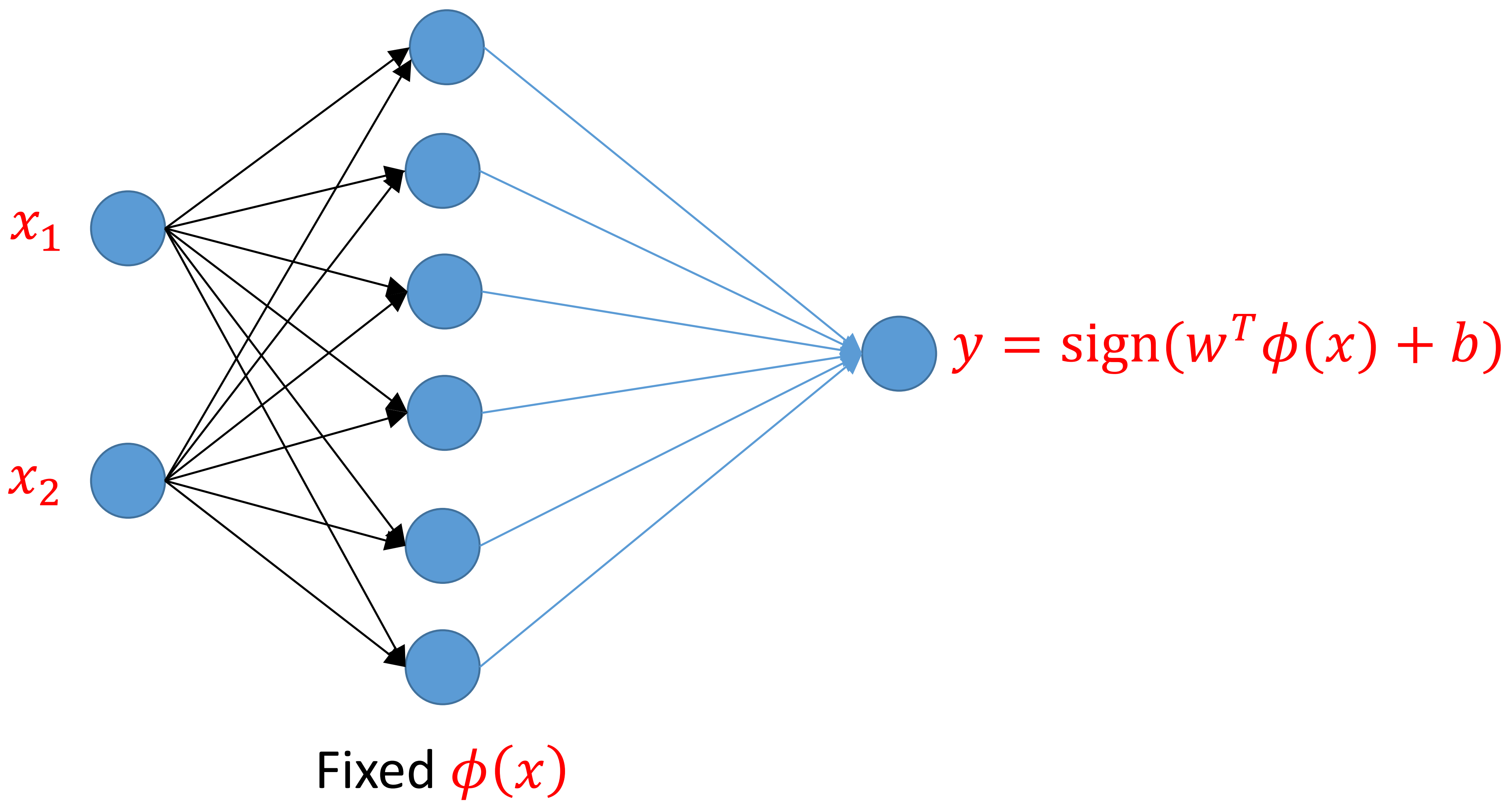
Nonlinear model



$\xrightarrow{\text{build hypothesis}}$ $y = w^T\phi(x)$

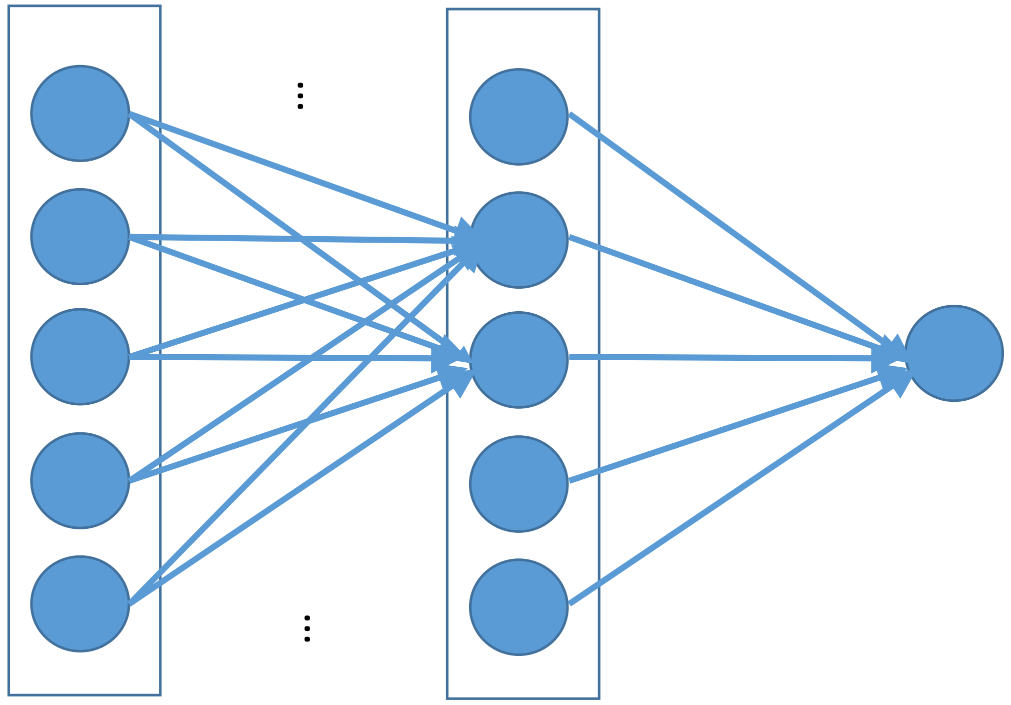
Linear model

Features: (polynomial) basis functions



Features and Representation Learning

- Why don'



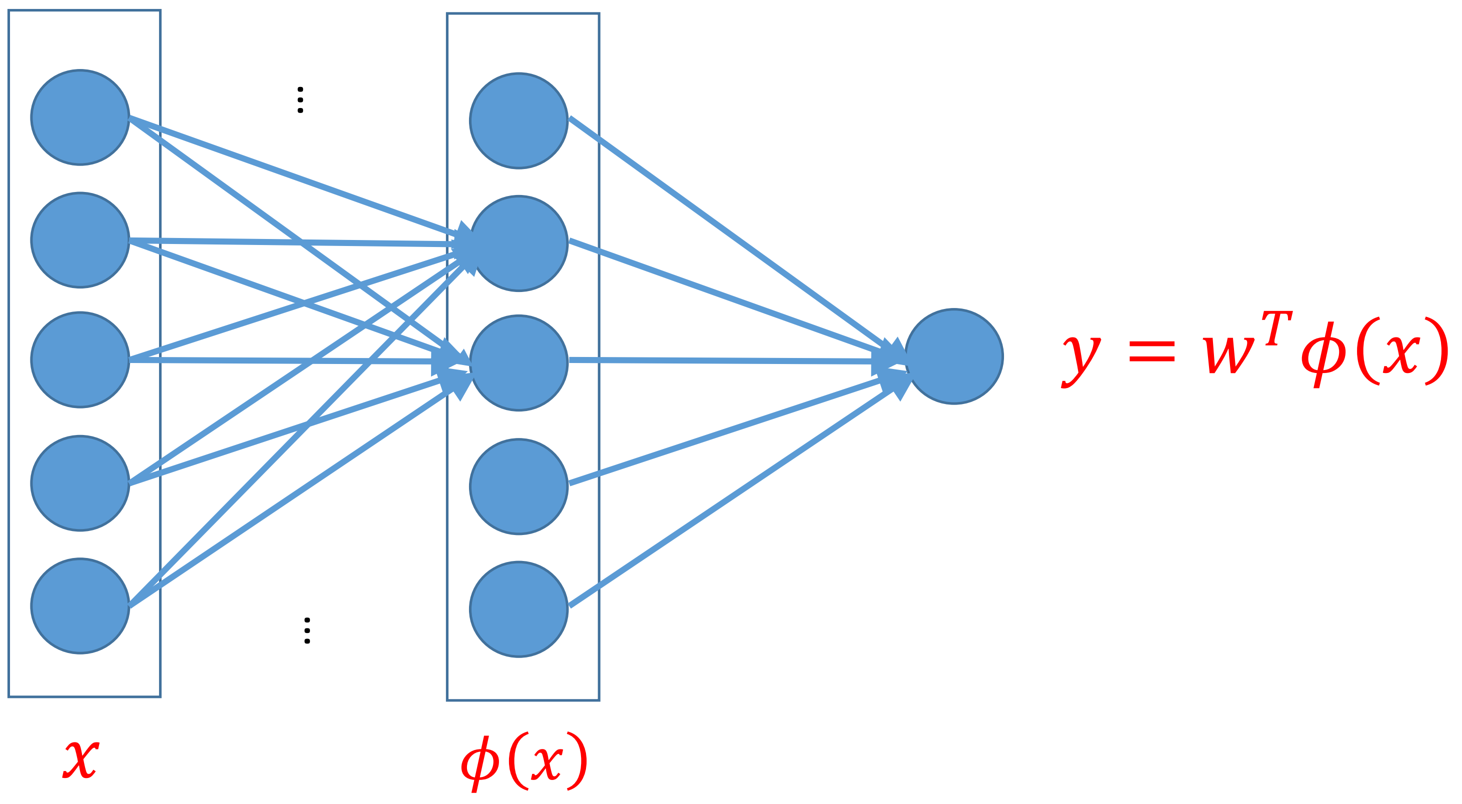
x $\quad$ x $\quad$ $\phi(x)$

$y = w^T \phi(x)$

$= w^T \phi(x)$

Adaptive Basis Functions

- View each dimension of $\phi(x)$ as something to be learned



Linear functions $\phi_i(x) = \theta_i^T x$ don't work: need some nonlinearity

Typically, set $\phi_i(x) = r(\theta_i^T x)$ where $r(\cdot)$ is some nonlinear function

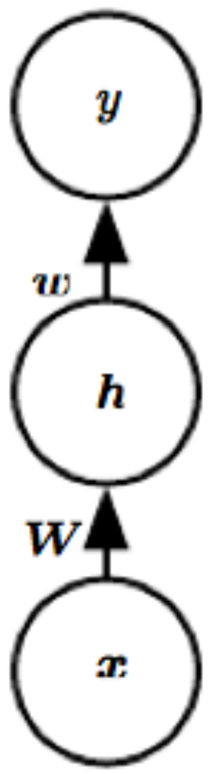
Hence, basis functions 'adapt' to data.

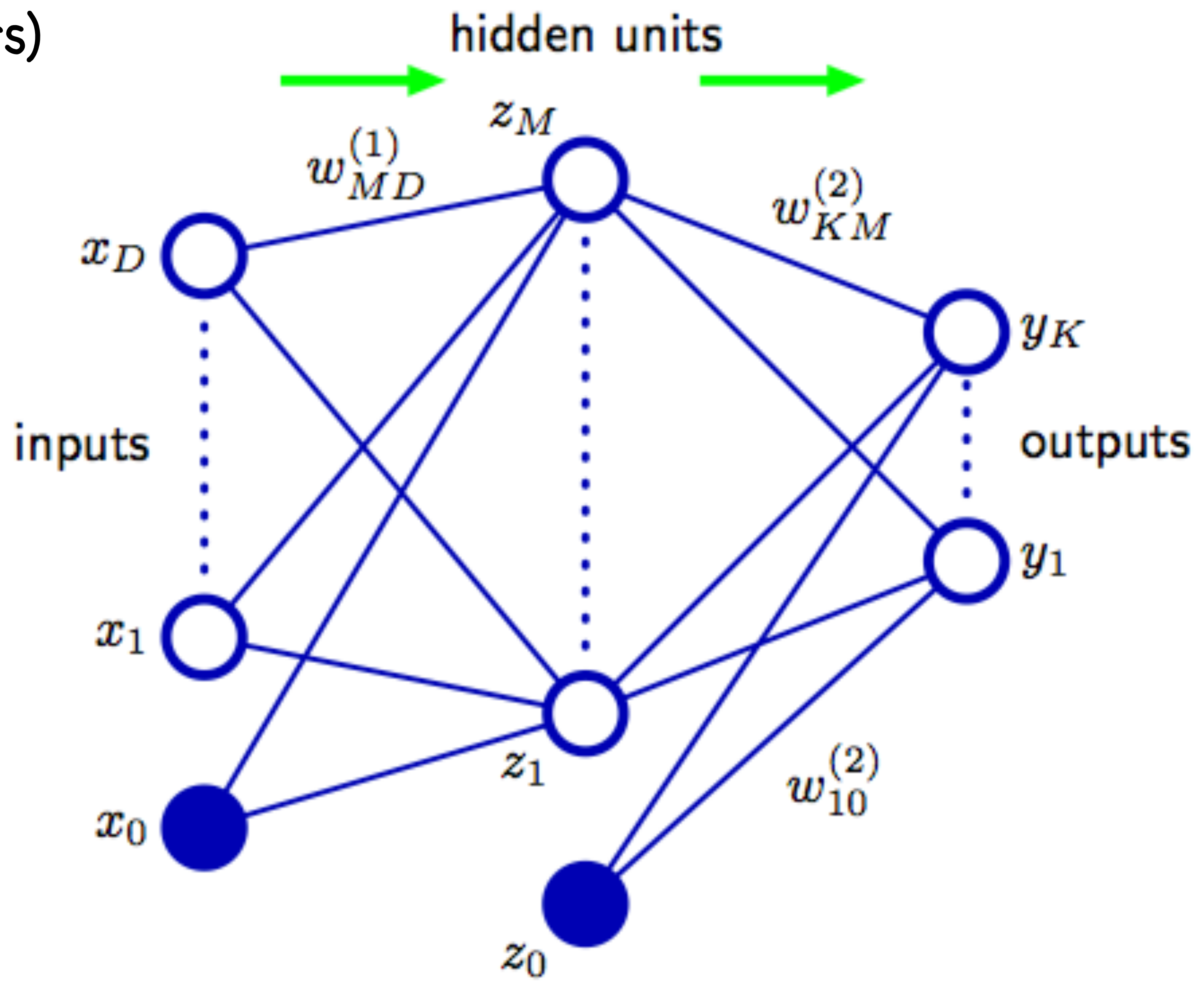
The model above is a (simple) **feedforward neural network**

Feedforward Neural Networks

A FNN is visually described by an acyclic graph. Nodes are of three categories:

- input units/nodes (layer 0)
- output units/nodes (layer n)
- hidden units/nodes (inner layers)





Feedforward Neural Networks

Each unit on an hidden layer takes an affine combination of values of input nodes, and then applies a non-linear activation function h.

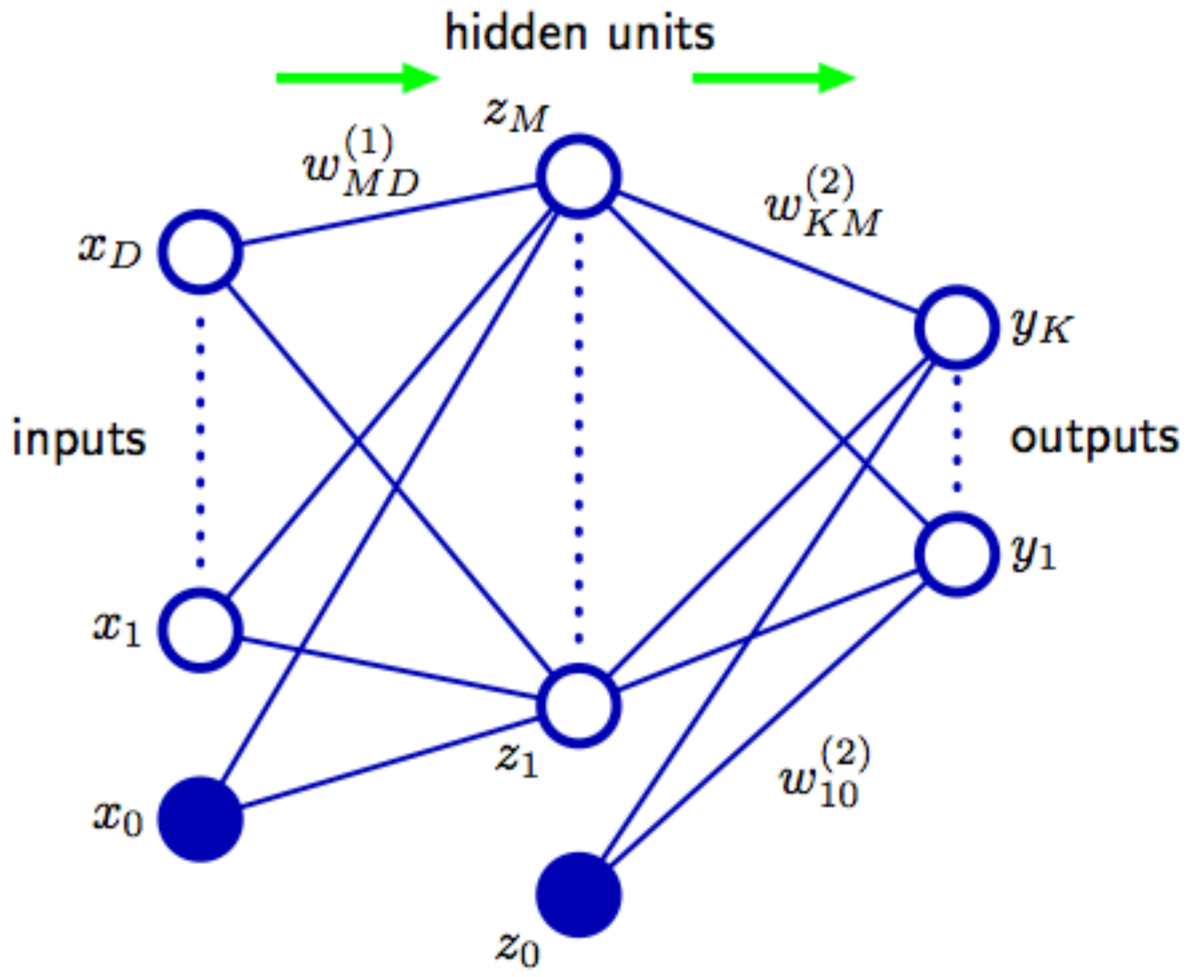
$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \qquad z_j = h(a_j).$$

Output nodes work similarly, but they apply an output activation function σ.

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)}$$

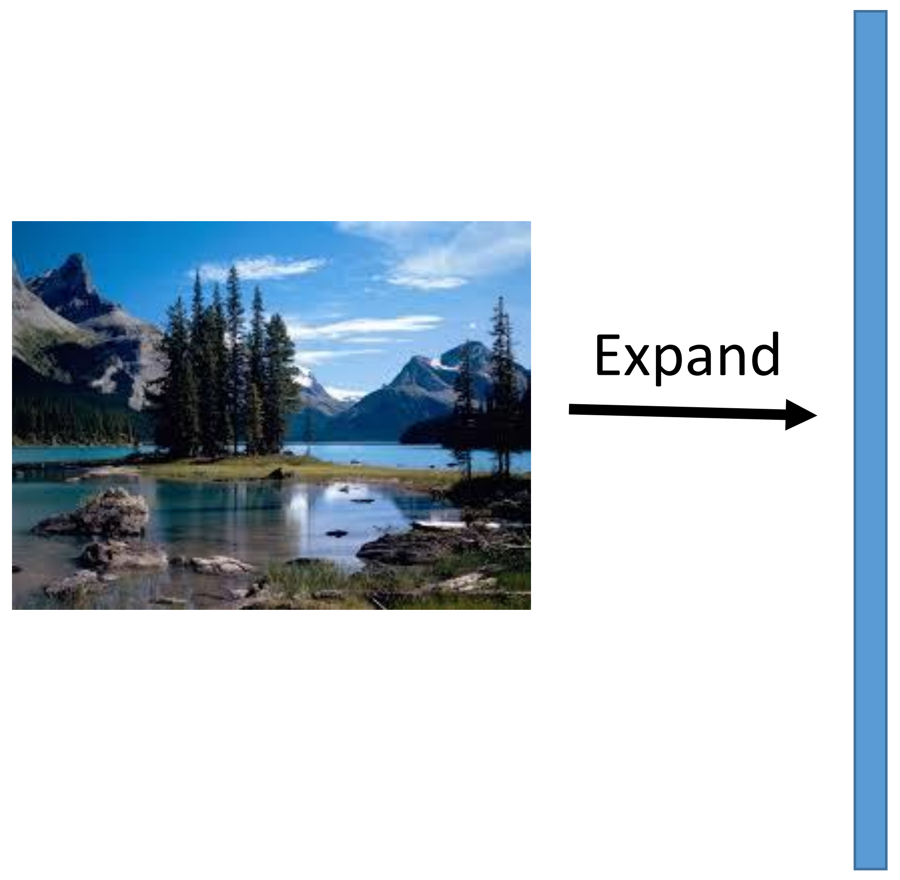
$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

A 1 hidden layer NN with linear output represents a linear combination of parametric non-linear basis functions, but learns also their parameters. This greatly enhances expressivity.



Input Units

- Represented as a vector
- Sometimes require some preprocessing, e.g.,
  - Subtract mean
  - Normalize to [-1,1]



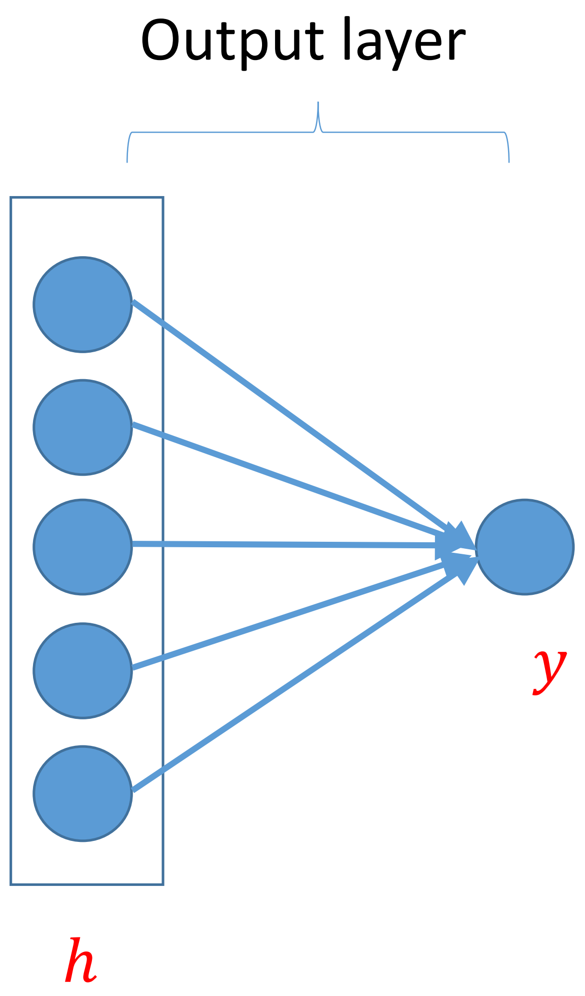
Data normalisation is a crucial step! Don't forget it!

Output units

The design of output units and their activation functions is driven by what we want to learn with our network. Typically, we want to learn a conditional probability distribution $p_{\text{model}}(\mathbf{y}|\mathbf{x})$, optimising the **cross-entropy**:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x})$$

- Regression: $y = w^T h + b$
- Linear units: no nonlinearity

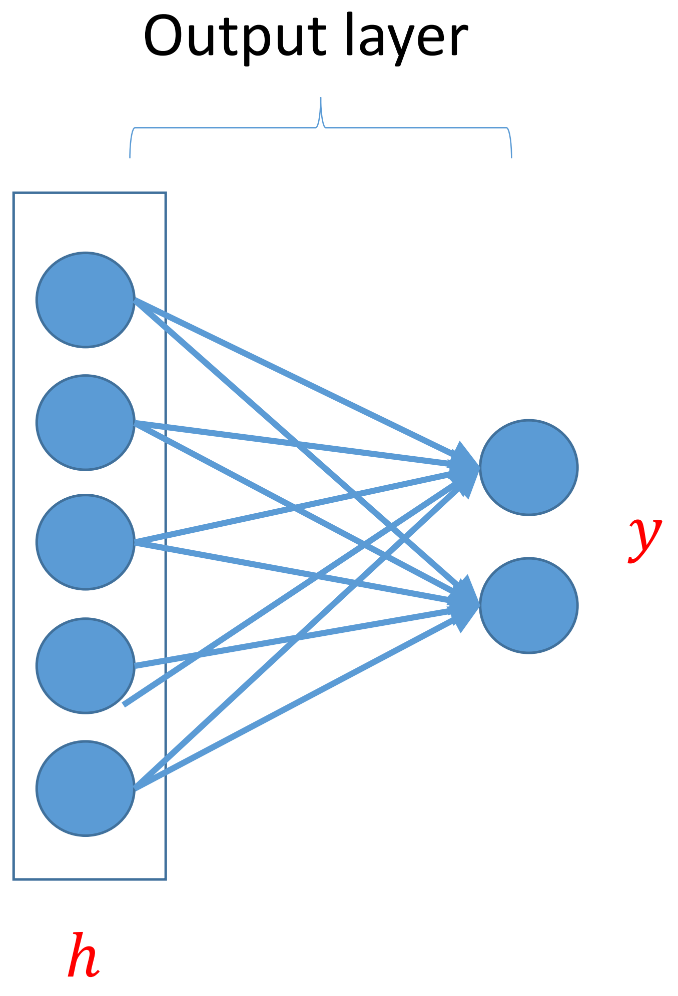


Output units

The design of output units and their activation functions is driven by what we want to learn with our network. Typically, we want to learn a conditional probability distribution $p_{\text{model}}(\mathbf{y}|\mathbf{x})$, optimising the **cross-entropy**:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x})$$

- Multi-dimensional regression: $y = W^T h + b$
- Linear units: no nonlinearity

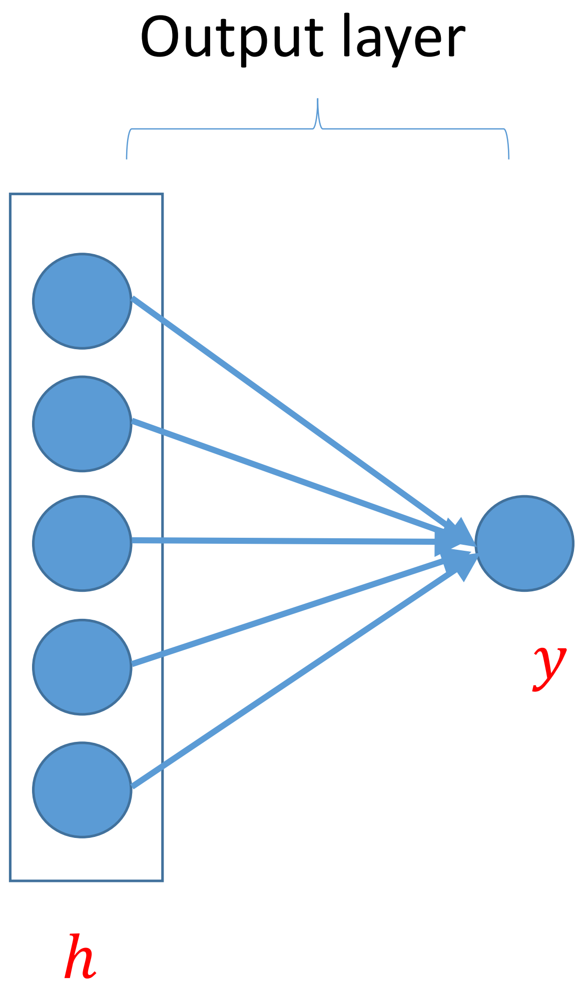


Output units

The design of output units and their activation functions is driven by what we want to learn with our network. Typically, we want to learn a conditional probability distribution $p_{\text{model}}(\mathbf{y}|\mathbf{x})$, optimising the **cross-entropy**:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x})$$

- Binary classification: $y = \sigma(w^T h + b)$
- Corresponds to using logistic regression on h

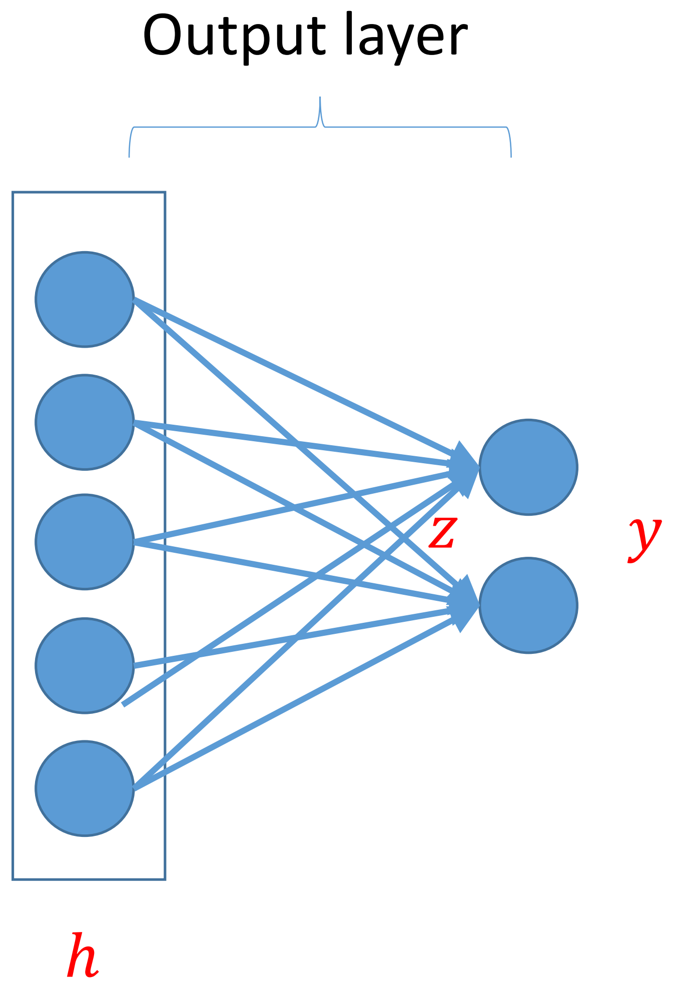


Output units

The design of output units and their activation functions is driven by what we want to learn with our network. Typically, we want to learn a conditional probability distribution $p_{\text{model}}(\mathbf{y}|\mathbf{x})$, optimising the **cross-entropy**:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x})$$

- Multi-class classification:
- $y = \text{softmax}(z)$ where $z = W^T h + b$
- Corresponds to using multi-class logistic regression on h



Output units

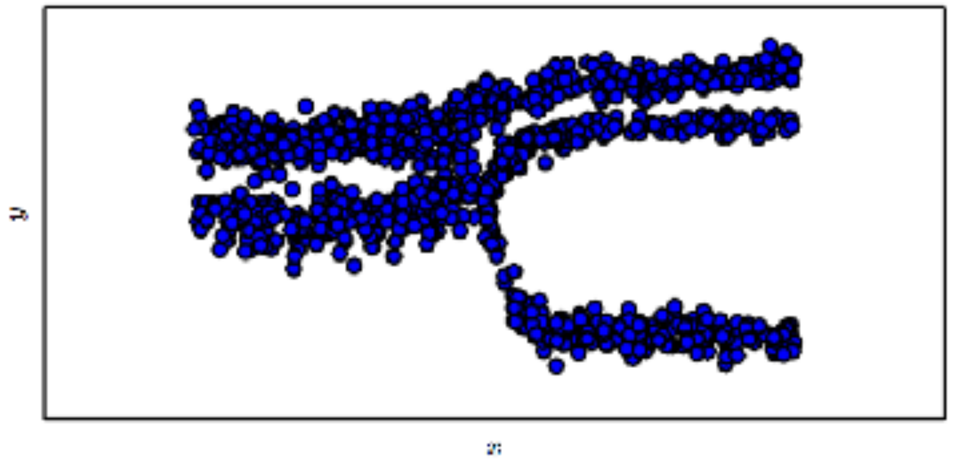
The design of output units and their activation functions is driven by what we want to learn with our network. Typically, we want to learn a conditional probability distribution $p_{\text{model}}(\mathbf{y}|\mathbf{x})$, optimising the **cross-entropy**:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x})$$

More general output units can be obtained by selecting more complex $p_{\text{model}}(\mathbf{y}|\mathbf{x})$.
Example: $p_{\text{model}}(\mathbf{y}|\mathbf{x})$ Gaussian, with heteroschedastic variance

Mixture density Neural Network: $p_{\text{model}}(\mathbf{y}|\mathbf{x})$ is a **mixture of Gaussians**

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \sum_{i=1}^{n} p(\mathrm{c} = i \mid \boldsymbol{x}) \mathcal{N}(\boldsymbol{y}; \boldsymbol{\mu}^{(i)}(\boldsymbol{x}), \boldsymbol{\Sigma}^{(i)}(\boldsymbol{x}))$$



Output units for mixture components are softmax, linear for the mean and the (factor of the) covariance matrix. This is typically assumed diagonal.

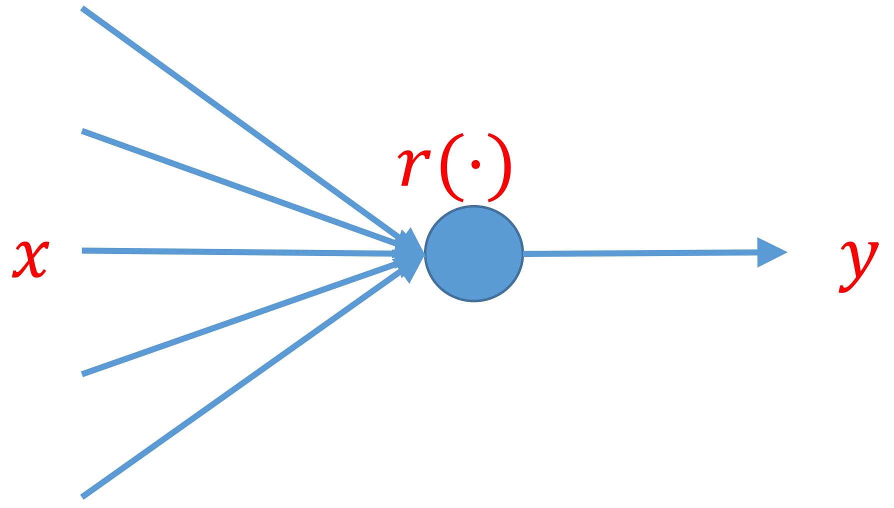
Hidden Units

Hidden units have typically non-linear activation functions (otherwise the NN becomes a linear model).
There are different choices of activation functions, driven by architectural constraints but also easiness in the learning phase.

$$z_k = h\left(\sum_j w_{kj} z_j\right)$$

- $y = r(w^T x + b)$

- Typical activation function r
  - Threshold $\mathrm{t}(z) = \mathbb{I}[z \geq 0]$
  - Sigmoid $\sigma(z) = 1/(1 + \exp(-z))$
  - Tanh $\tanh(z) = 2\sigma(2z) - 1$



Hidden Units

Hidden units have typically non-linear activation functions (otherwise the NN becomes a linear model).
There are different choices of activation functions, driven by architectural constraints but also easiness in the learning phase.

$$z_k = h\left(\sum_j w_{kj} z_j\right)$$

- Problem: saturation

Too small gradient

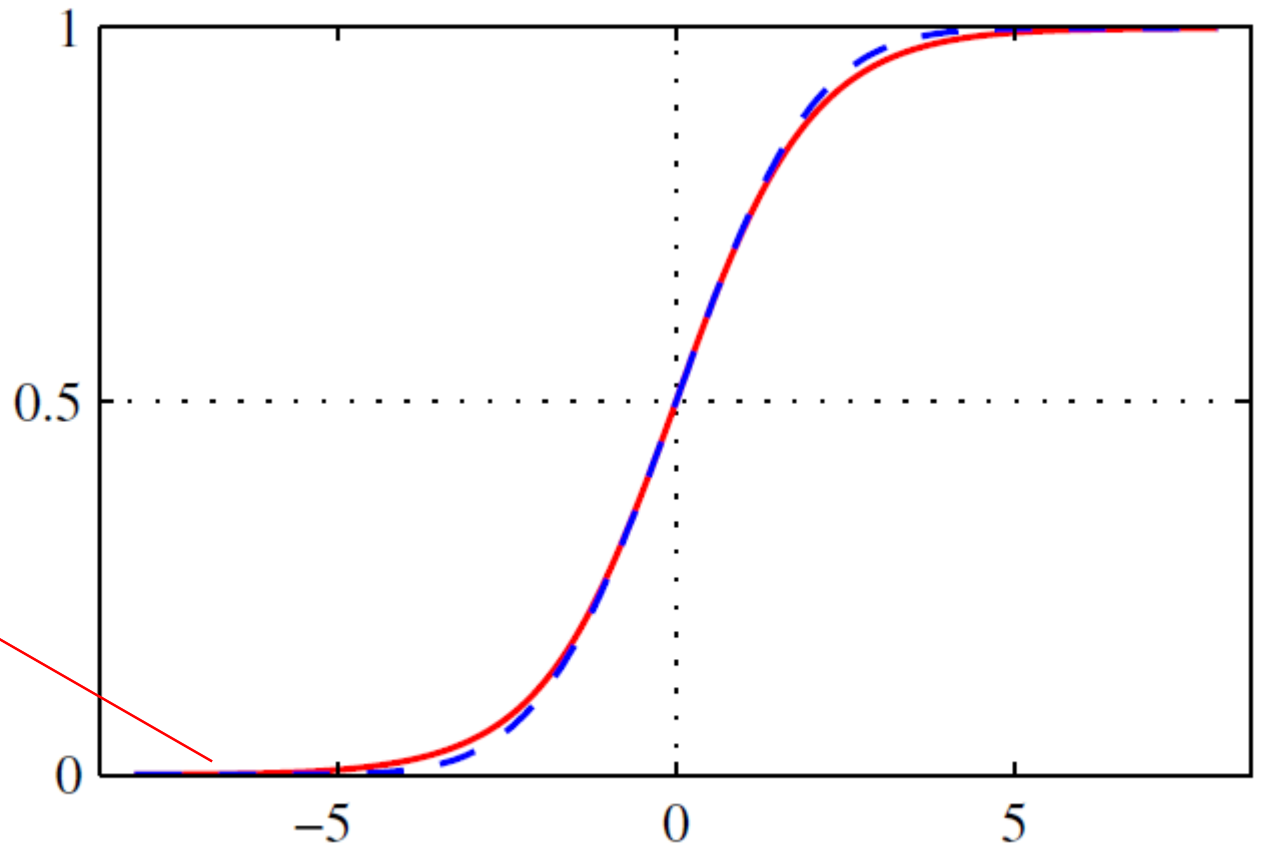


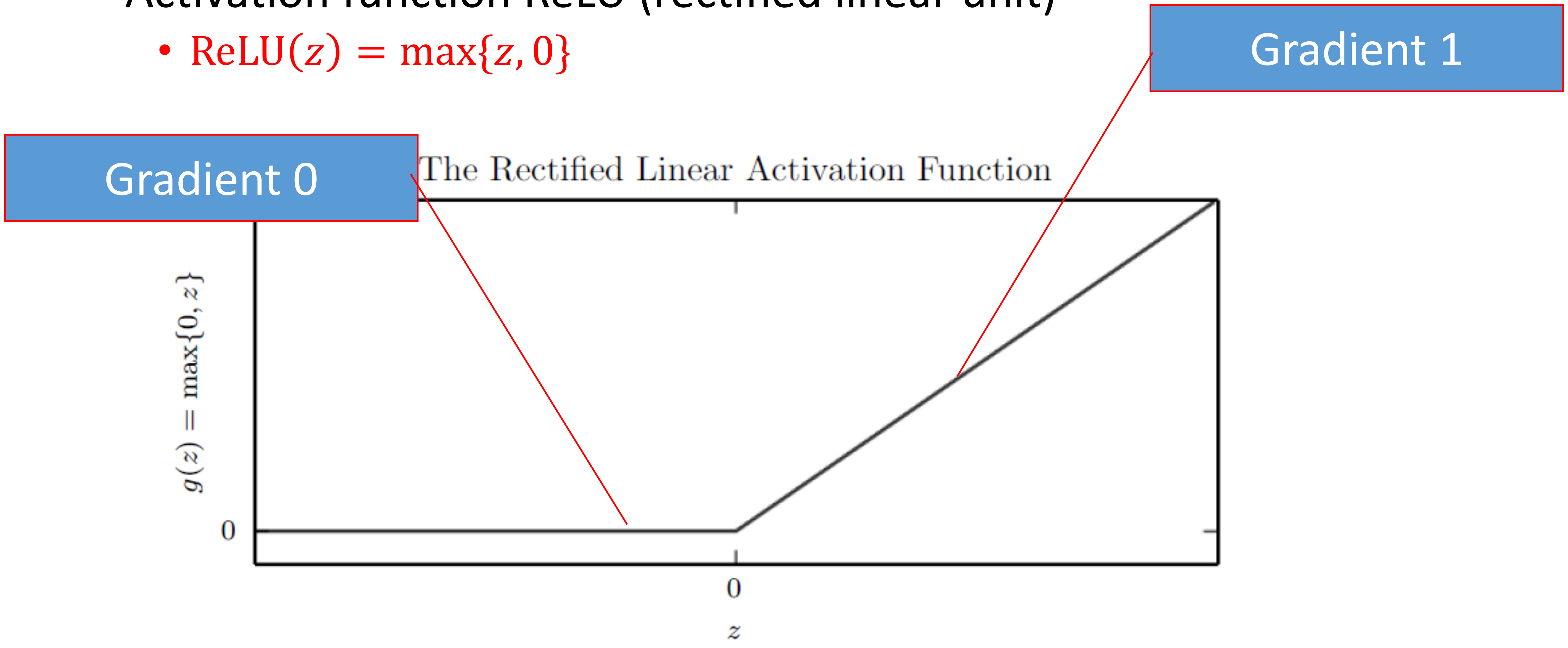
Figure borrowed from *Pattern Recognition and Machine Learning*, Bishop

Hidden Units

Hidden units have typically non-linear activation functions (otherwise the NN becomes a linear model).
There are different choices of activation functions, driven by architectural constraints but also easiness in the learning phase.

$$z_k = h\left(\sum_j w_{kj} z_j\right)$$

- Activation function ReLU (rectified linear unit)
  - $\mathrm{ReLU}(z) = \max\{z, 0\}$

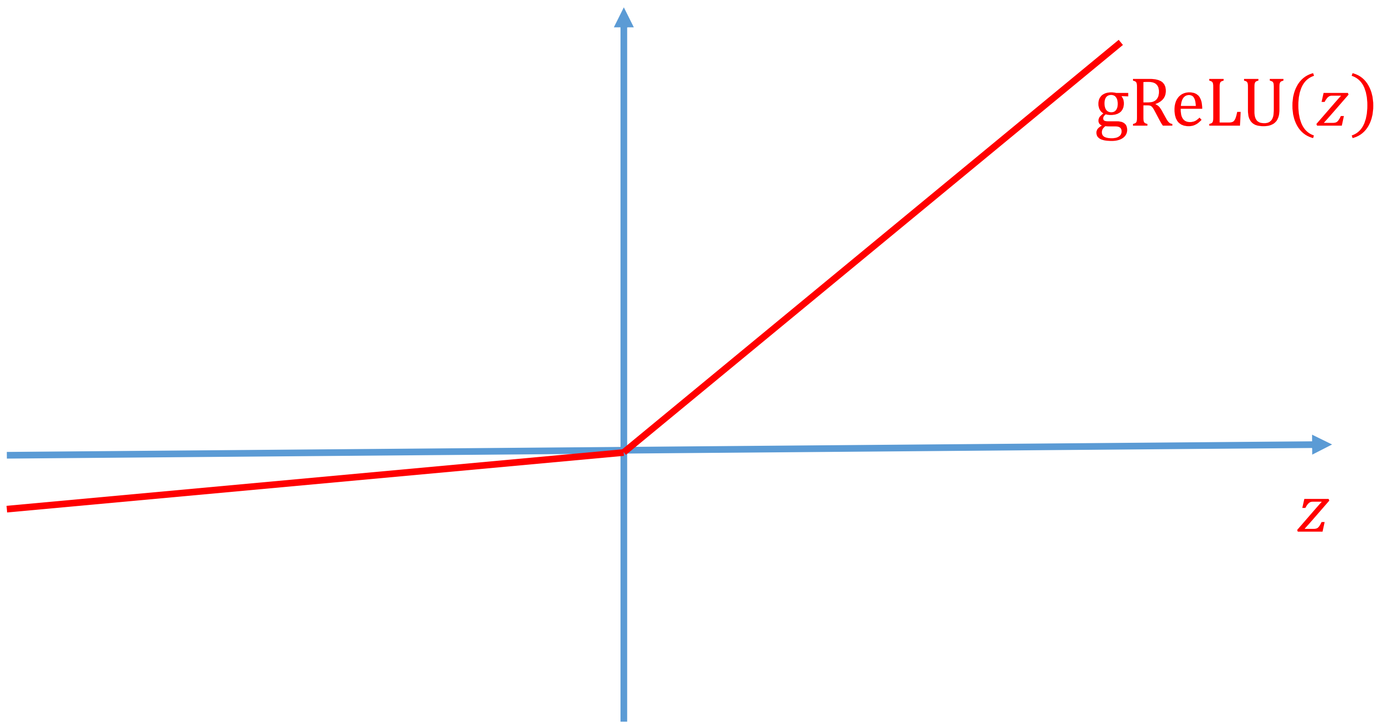


Hidden Units

Hidden units have typically non-linear activation functions (otherwise the NN becomes a linear model).
There are different choices of activation functions, driven by architectural constraints but also easiness in the learning phase.

$$z_k = h\left(\sum_j w_{kj} z_j\right)$$

- Generalizations of ReLU $\mathrm{gReLU}(z) = \max\{z, 0\} + \alpha \min\{z, 0\}$
  - $\text{Leaky-ReLU}(z) = \max\{z, 0\} + 0.01 \min\{z, 0\}$
  - $\text{Parametric-ReLU}(z)$: α learnable



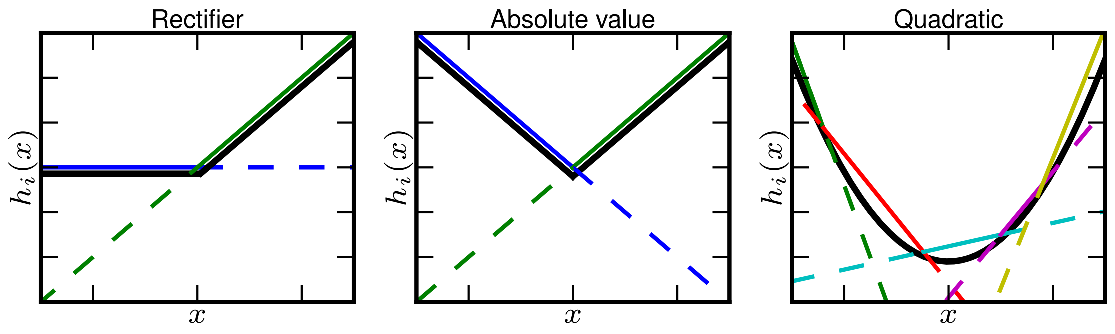
Hidden Units

Hidden units have typically non-linear activation functions (otherwise the NN becomes a linear model).
There are different choices of activation functions, driven by architectural constraints but also easiness in the learning phase.

$$z_k = h\left(\sum_j w_{kj} z_j\right)$$

Maxout Unit: computes k affine transformations of the input, and than takes the max:

$$h_i(x) = \max_{j \in [1,k]} z_{ij}$$



Can have multidimensional output.

Other types of hidden units include: cosine, Gaussian Radial Basis Functions, softplus or smoothed rectifier $g(a) = \zeta(a) = \log(1+e^a)$

Architecture Design

The architecture design, a part from output and hidden units, requires to choose the number of units, and how they are arranged: **width** and **depth** of the network.

MLP with one hidden layer and sigmoid activation functions (and others) are **universal approximators**, meaning that the set of functions that can be represented by such a MLP (with n hidden nodes, n unbounded) is **dense in the set of measurable functions**.

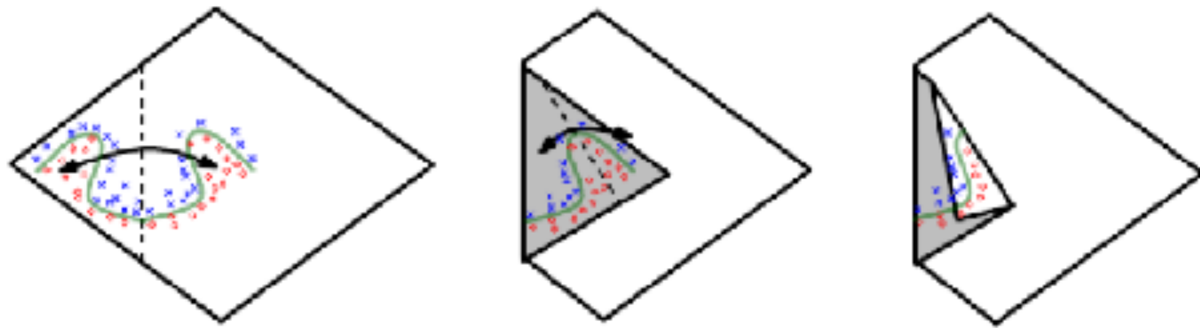
However: we need to fix the number of nodes. One may need exponentially many of them (there are bounds but very loose).

And there is a **No free lunch theorem**: there is no universal machine learning algorithm.

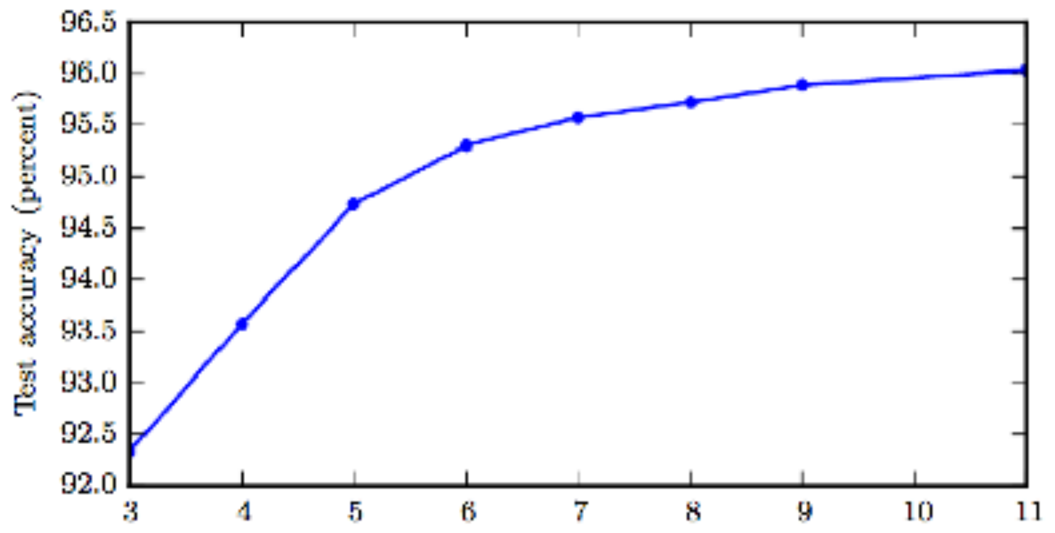
Other things can also go wrong: **SGD may fail, overfitting**.

Architecture Design

Advantage of depth > 1: MLP with depth d and ReLU can learn piecewise linear functions with a number of regions exponential in d. Hence depth can reduce parameters considerably.



Statistical argument for deep networks: a deep MLP expresses the belief that our model must be composed by the composition of many simple functions.



Example: accuracy vs depth for multi digit transcription.

Skip connections: connect directly layer j+1 with layer j-1, i.e. the input to layer j+1 becomes $\mathbf{z}_{j+1} = h_j(\mathbf{z}_j) + \mathbf{z}_j$
(this reduces vanishing gradient problem)

Computing Gradients: Backpropogation

MLP are typically trained by **Stochastic Gradient Descent** (with multistart):

$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)})$ (updates each training point in sequence)

The error function is usually given by cross-entropy: $J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x})$.

Warning: the learning problem is **highly non-convex**. Many local minima, also due to the presence of many **symmetries in the weight space**.

Computing Gradients: Backpropogation

The error function is usually given by **cross-entropy**:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}).$$

The gradient of $E(\theta)=J(\theta)$ can be computed efficiently by **backpropagation**

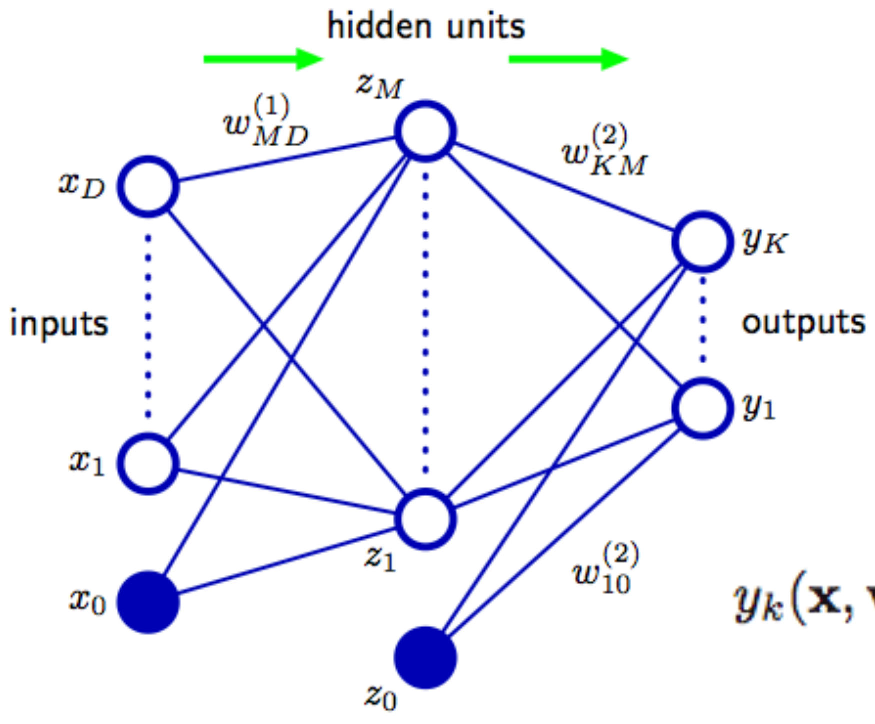
Backpropagation is a dynamic programming algorithm that computes the gradient by going from output nodes back to input nodes in the network.

It has two steps:

1. **Forward** propagate input $\mathbf{x}_n$ computing the value of all hidden and output nodes.

2. **Backward** propagate the gradient from output nodes to input ones.

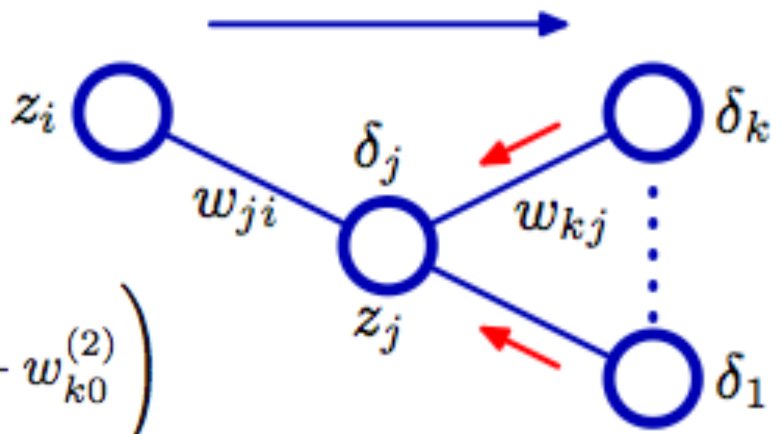
Computing Gradients: Backpropogation



$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \qquad z_j = h(a_j).$$

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)}$$

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$



We have $\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$ and $\frac{\partial a_j}{\partial w_{ji}} = z_i$ Call $\delta_j \equiv \frac{\partial E_n}{\partial a_j}$ Then $\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$

For linear output nodes: $\delta_k = y_k - t_k$ For other nodes: $\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j}$

Now, as $a_k = \sum_j w_{kj} h(a_j)$ it holds that $\delta_j = h'(a_j)\sum_k w_{kj}\delta_k$

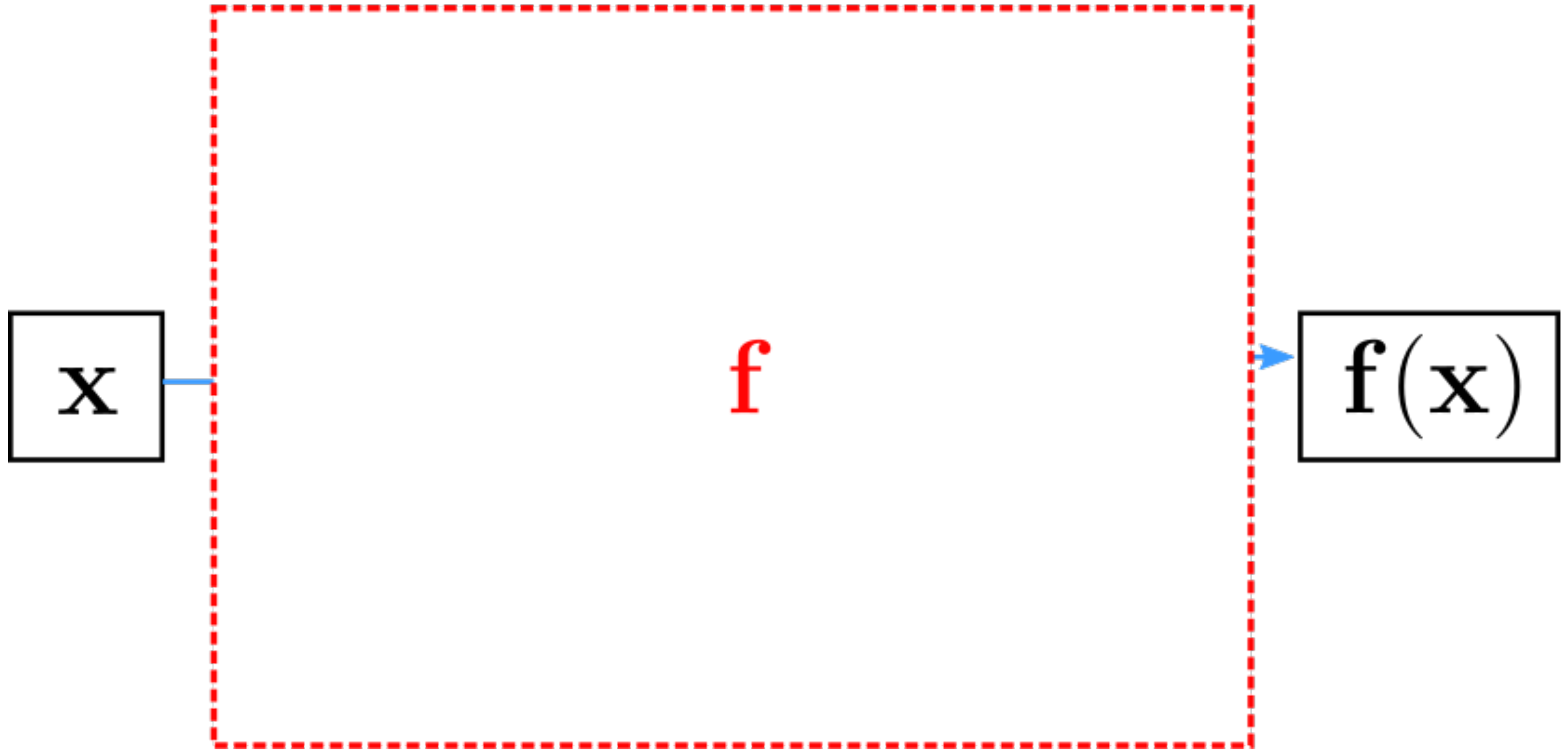
Frameworks for DL

Automatic differentiation: **TensorFlow**, MXnet, CNTK, *Theano*

Dynamic and high level: Torch & **PyTorch** , Chainer, MinPy, DyNet...

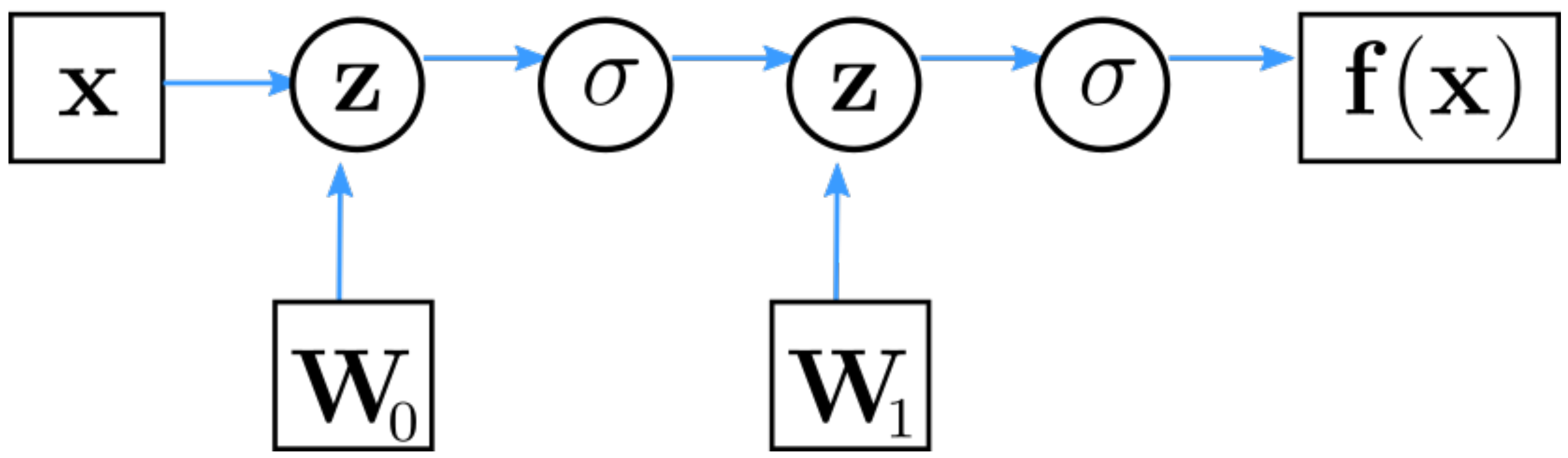
Keras: high level frontend for TensorFlow, mxnet, theano, cntk

Computational Graphs



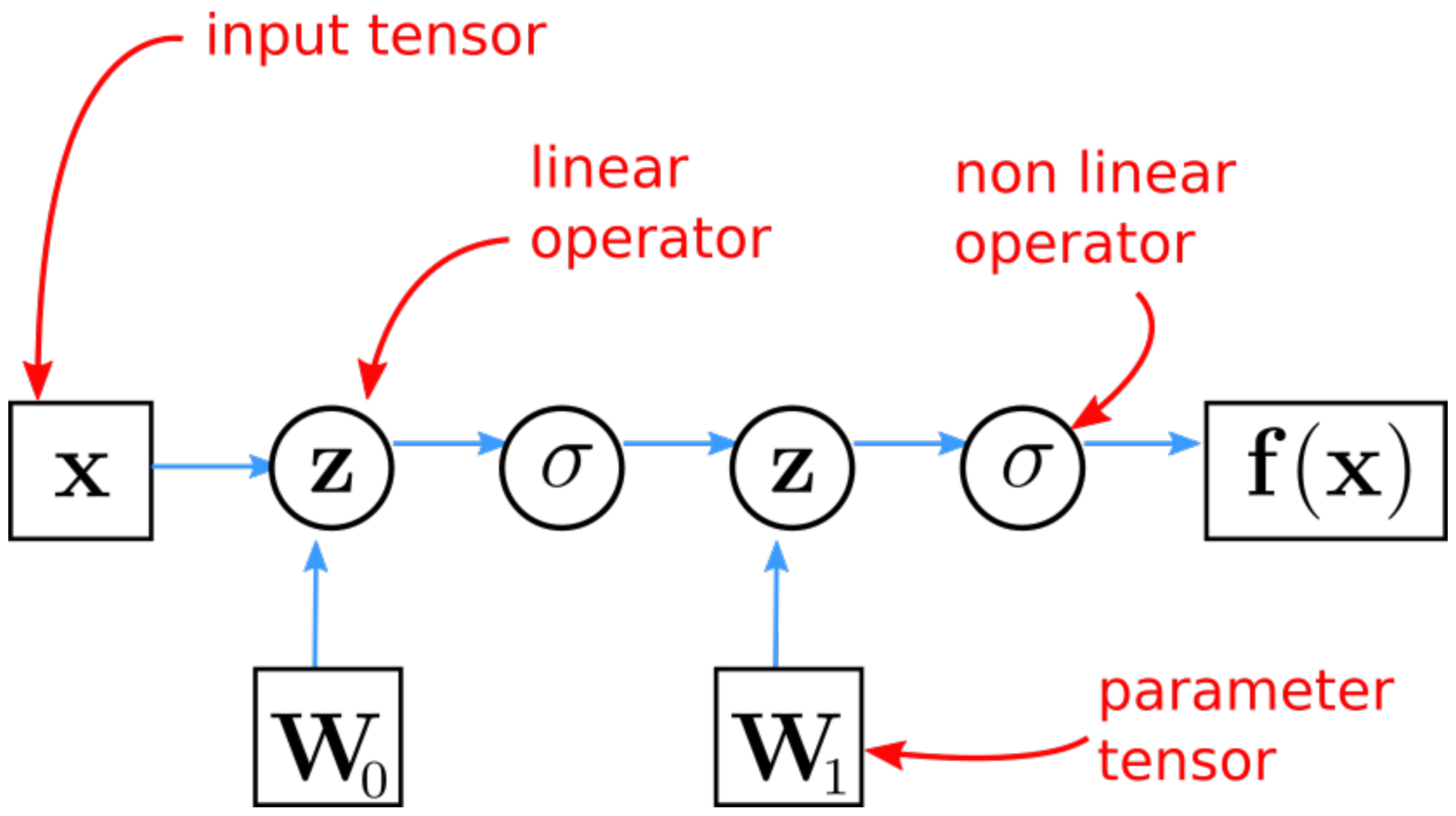
Neural network = parametrized, non-linear function

Computational Graphs



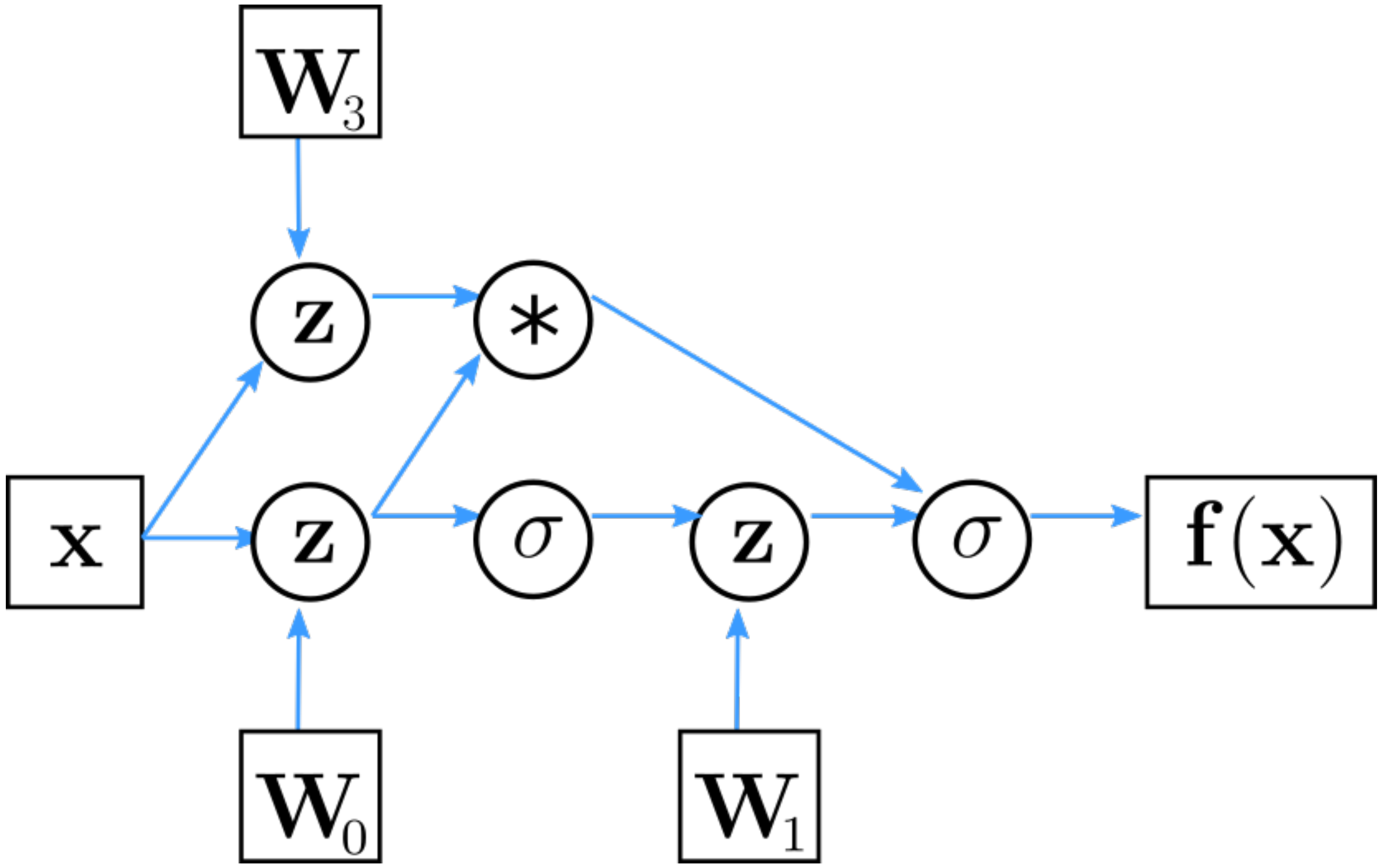
Computation graph: Directed graph of functions, depending on parameters (neuron weights)

Computational Graphs



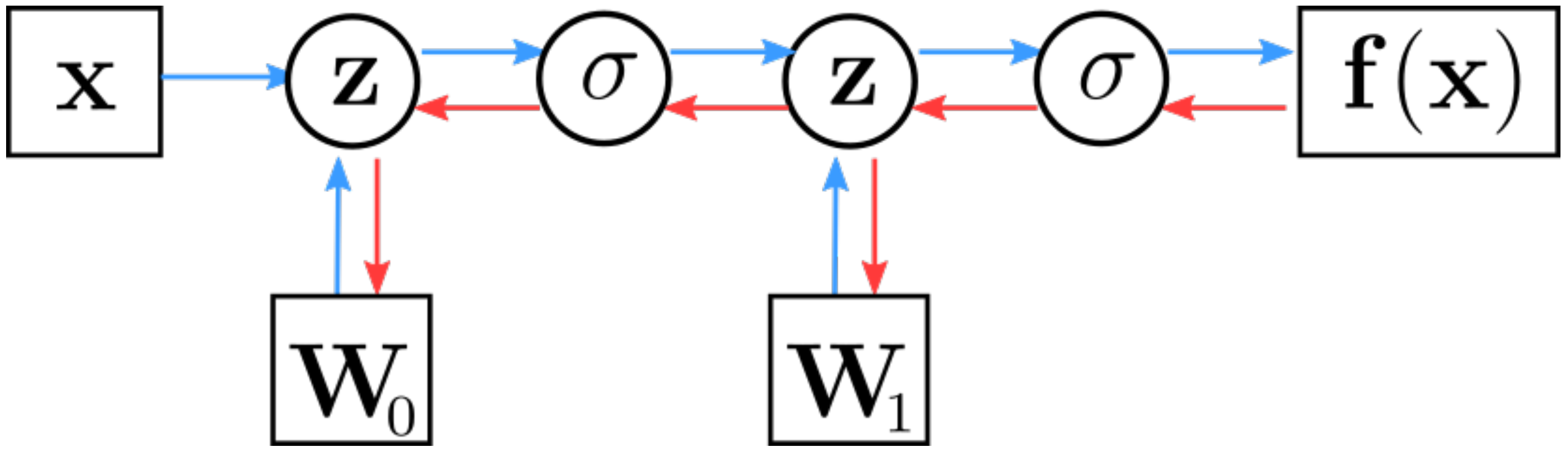
Combination of linear (parametrized) and non-linear functions

Computational Graphs



Not only sequential application of functions

Computational Graphs



Automatic computation of gradients: all modules are **differentiable**!

Tensorflow, *theano*, etc. build a static computation graph

Torch, **pytorch**, etc. rely on dynamic differentiable modules

All frameworks enable parallel computation on **CPU** and **GPU**