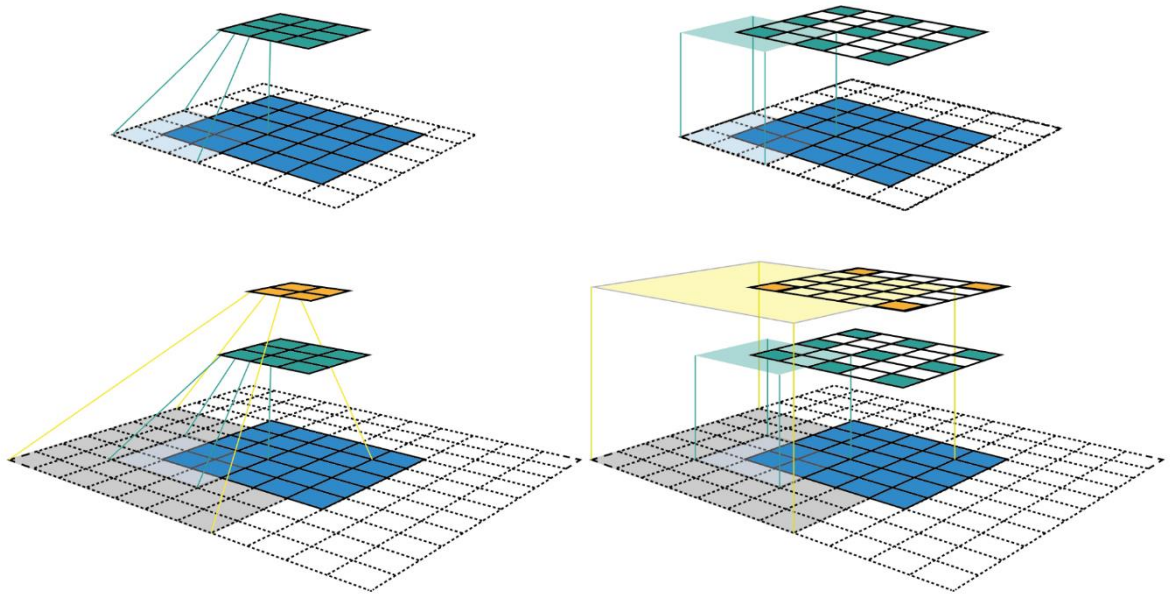
University of Ljubljana, Faculty of Computer and Information Science

Neural networks



Prof Dr Marko Robnik-Šikonja

Intelligent Systems, Edition 2024

Topics overview

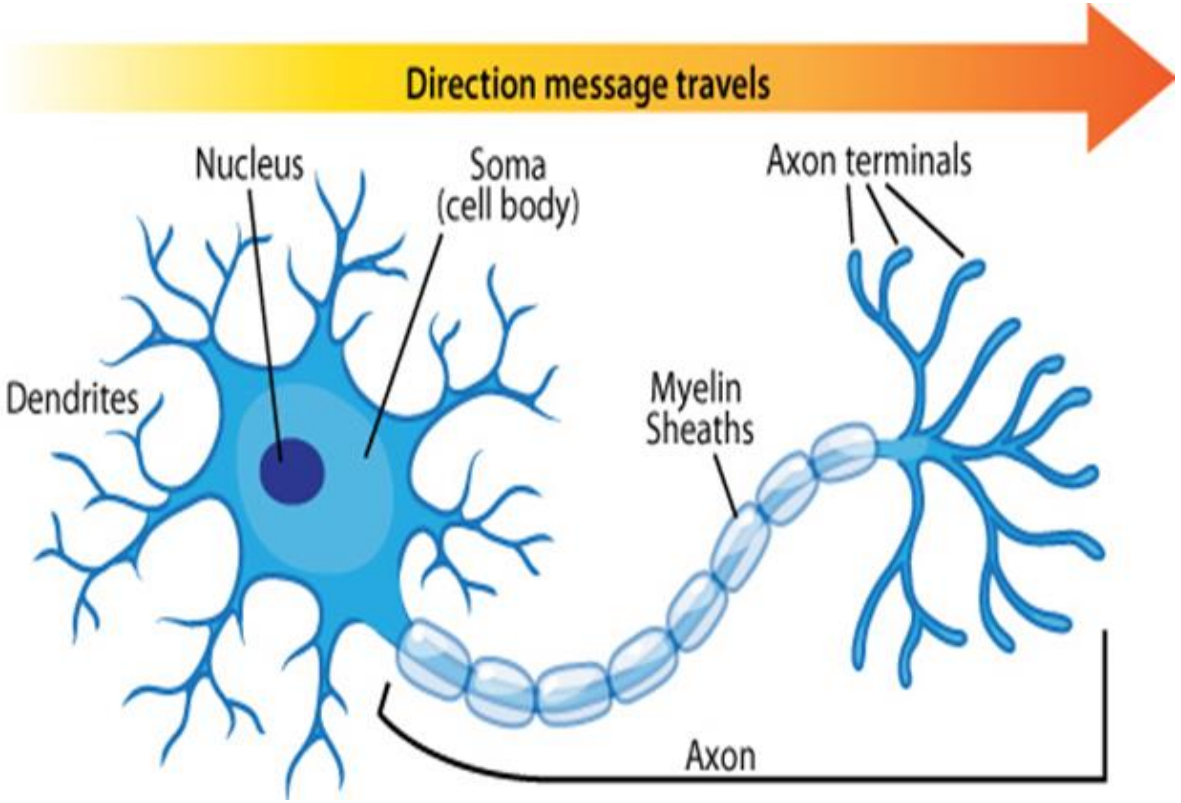
- basics of artificial neural networks
- backpropagation
- deep learning
- convolutional neural networks
- autoencoders
- generative adversarial networks
- robustness

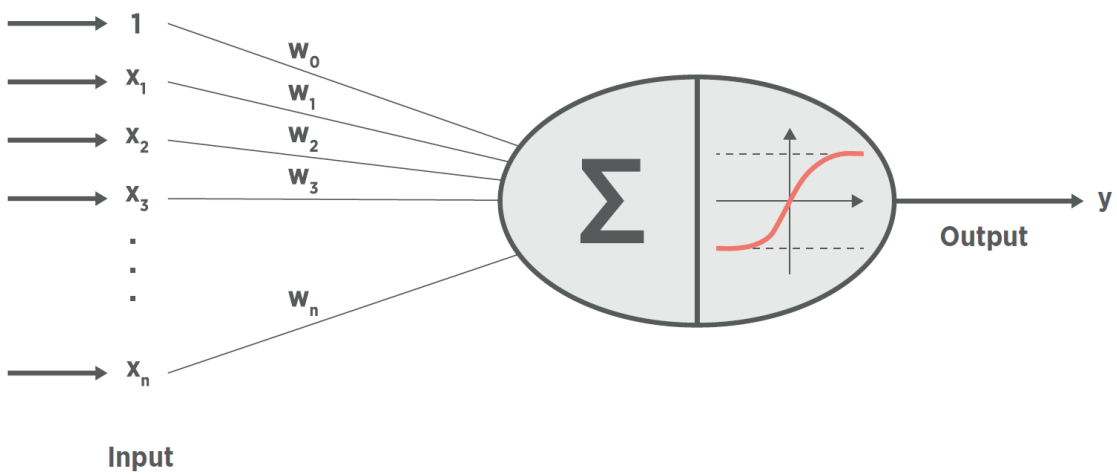
We will mention transformer networks in the natural language processing topic.

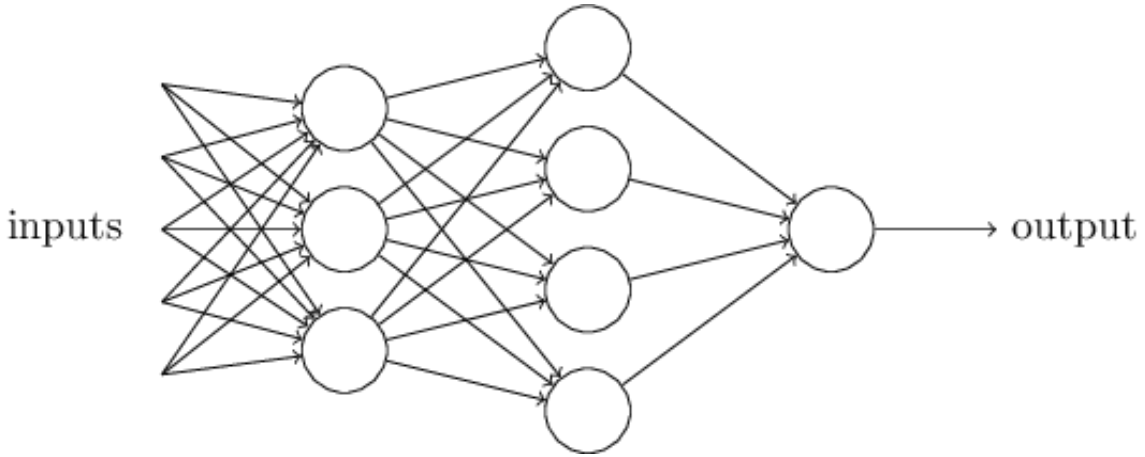
Artificial neural networks

- many approaches, we shall cover the basic ideas
- currently very strong interest, especially in deep neural networks
- http://www.deeplearningbook.org (from 2016, see also other newer literature in the slides 01)

Artificial neural networks: brain analogy



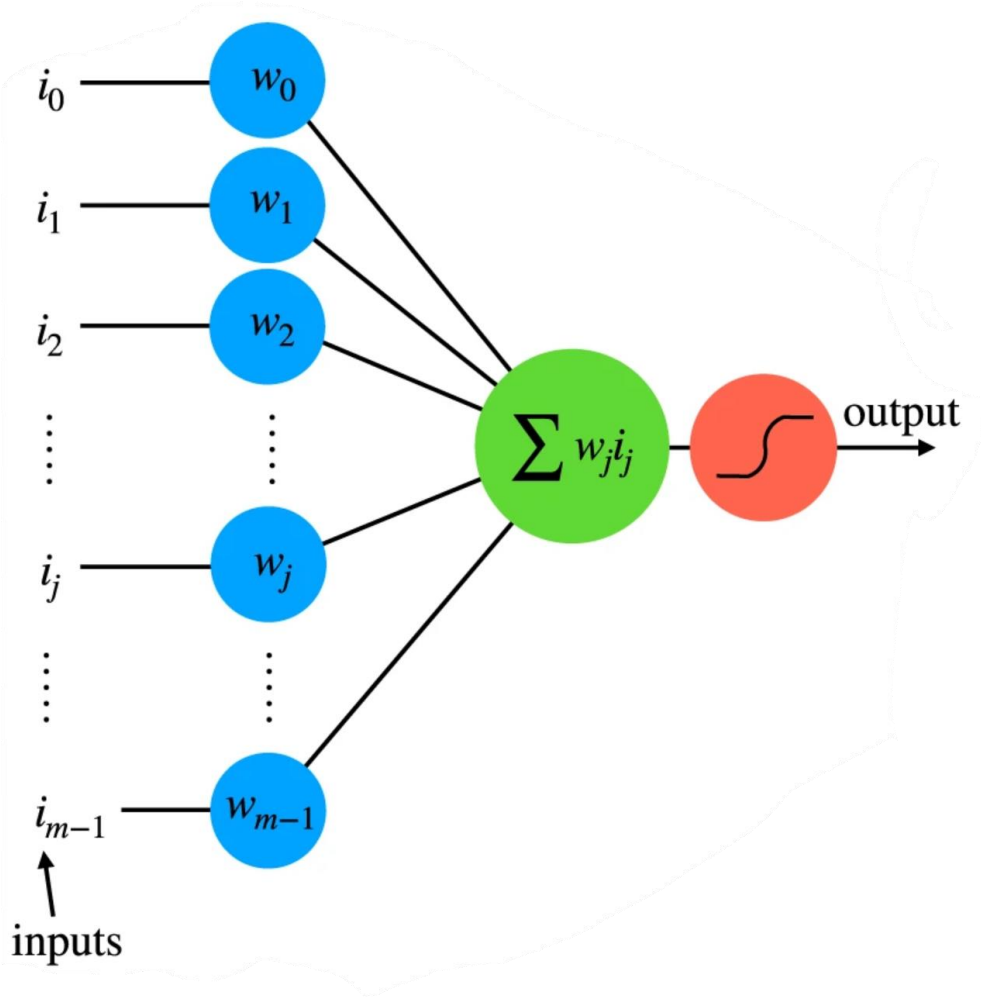




learning: error backpropagation

Neuron

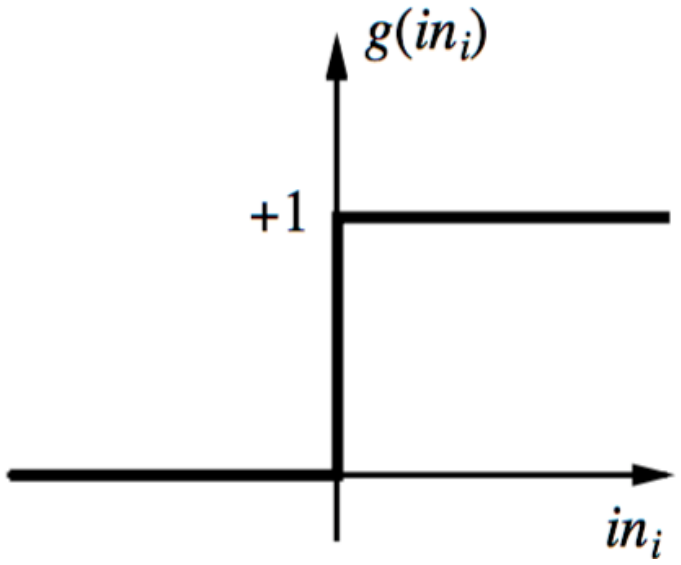
- Computational units, passing messages (information) in the network, typically organized into layers

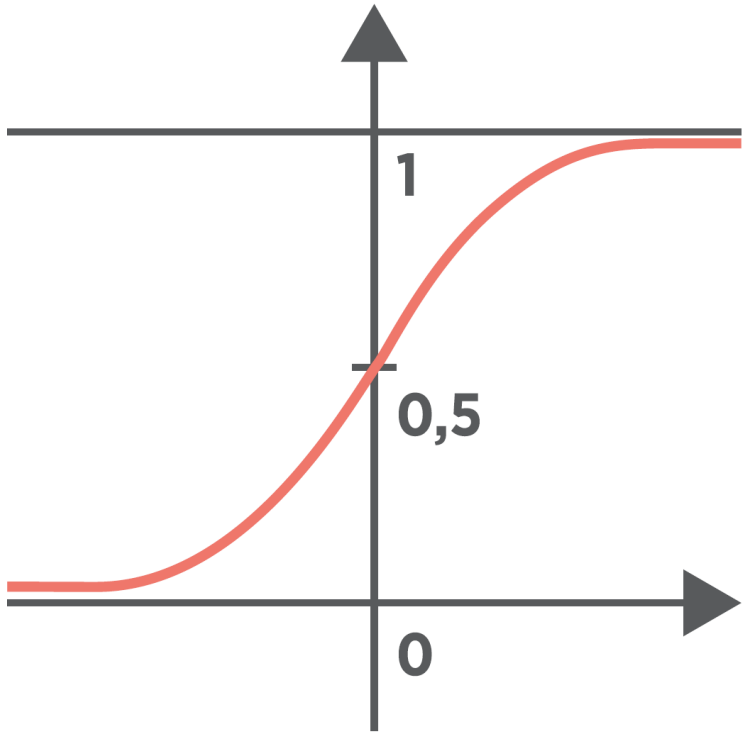


Activation functions

- examples: step function, sigmoid (logistic)

$$f(x) = \frac{1}{1 + e^{-x}}$$





Activation functions

- ReLU (rectified linear unit)

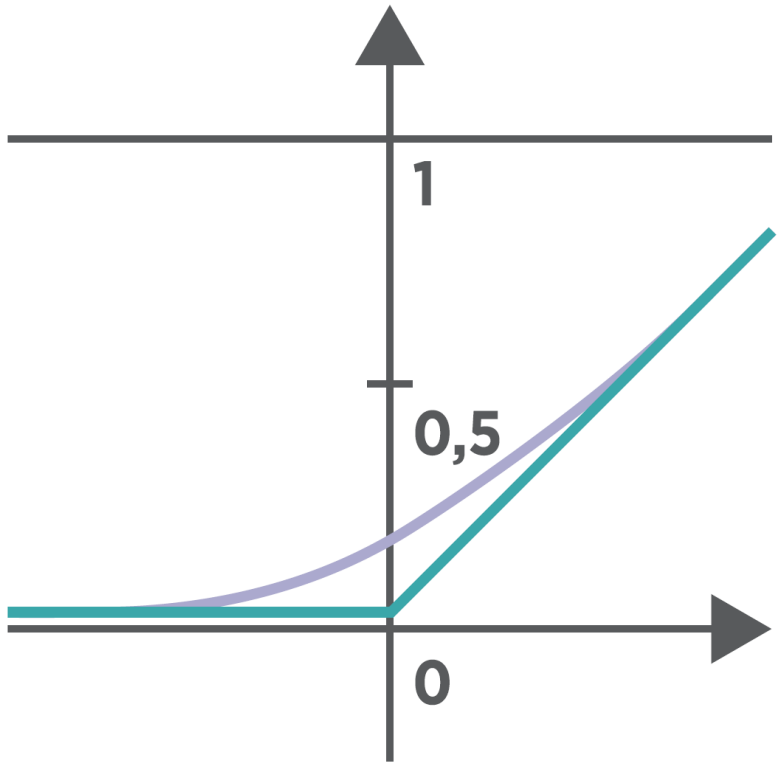
  $f(x) = \max(0, x)$

- softplus / approximation of ReLU with continuous derivation

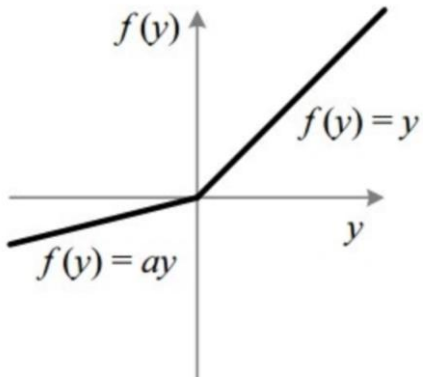
  $f(x) = \ln(1+e^x)$

- ELU (Exponential Linear Unit)

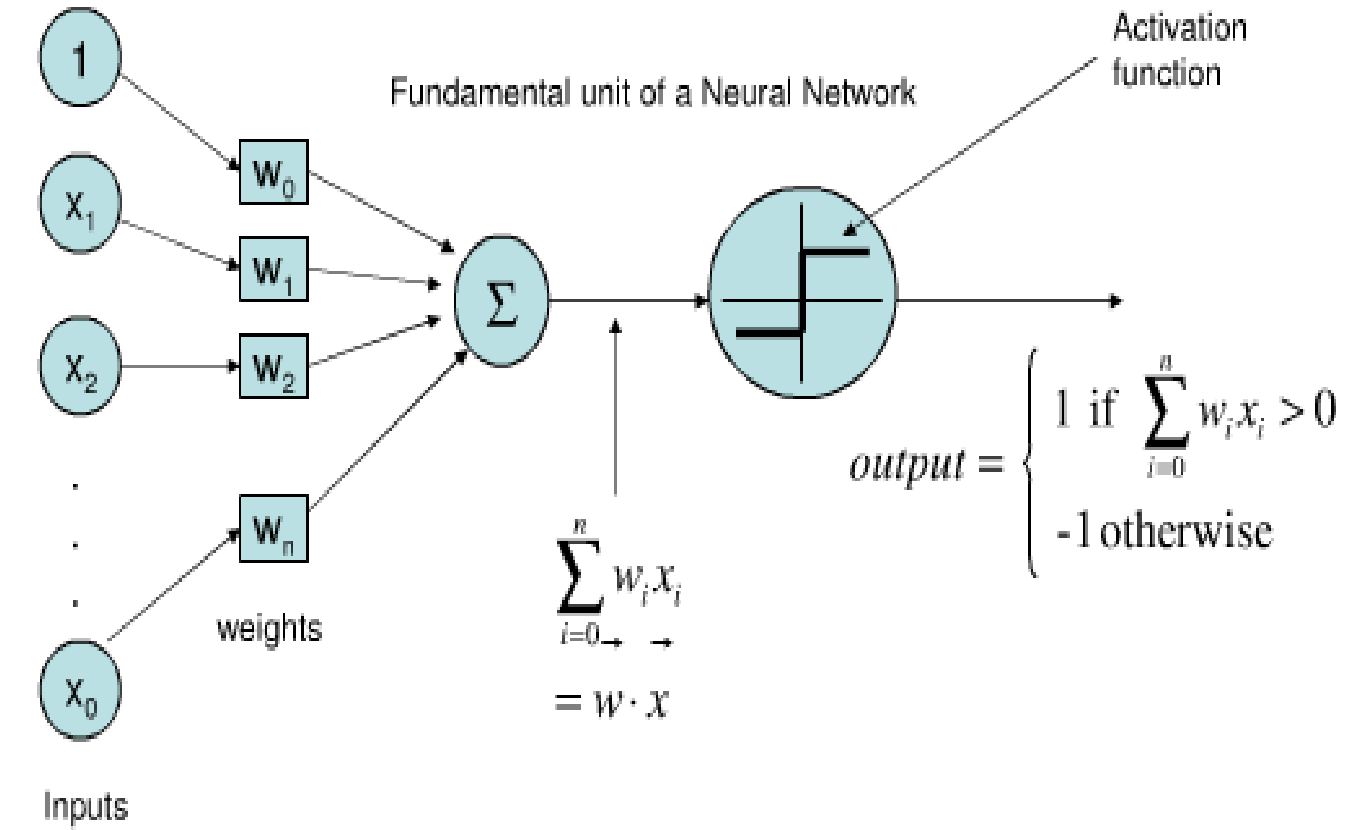
$$ELU(x) = \begin{cases} c \cdot (e^x - 1), & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$



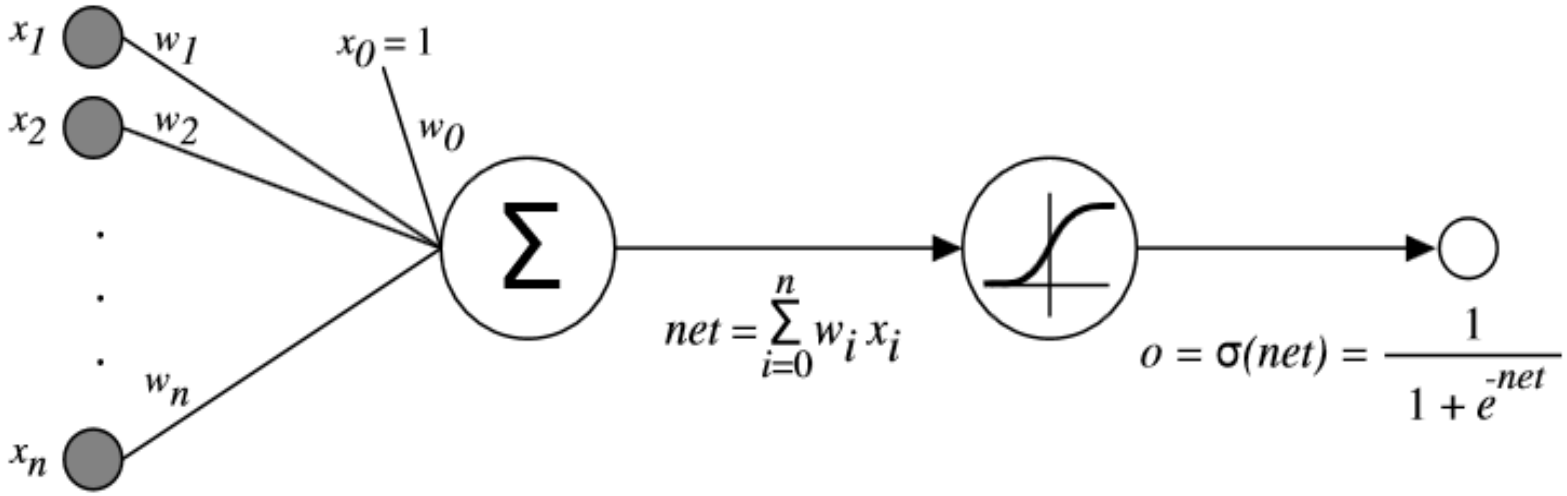
- Leaky ReLU: like ReLU but small slope for negative values instead of a flat slope
- many others



Historically: Perceptron

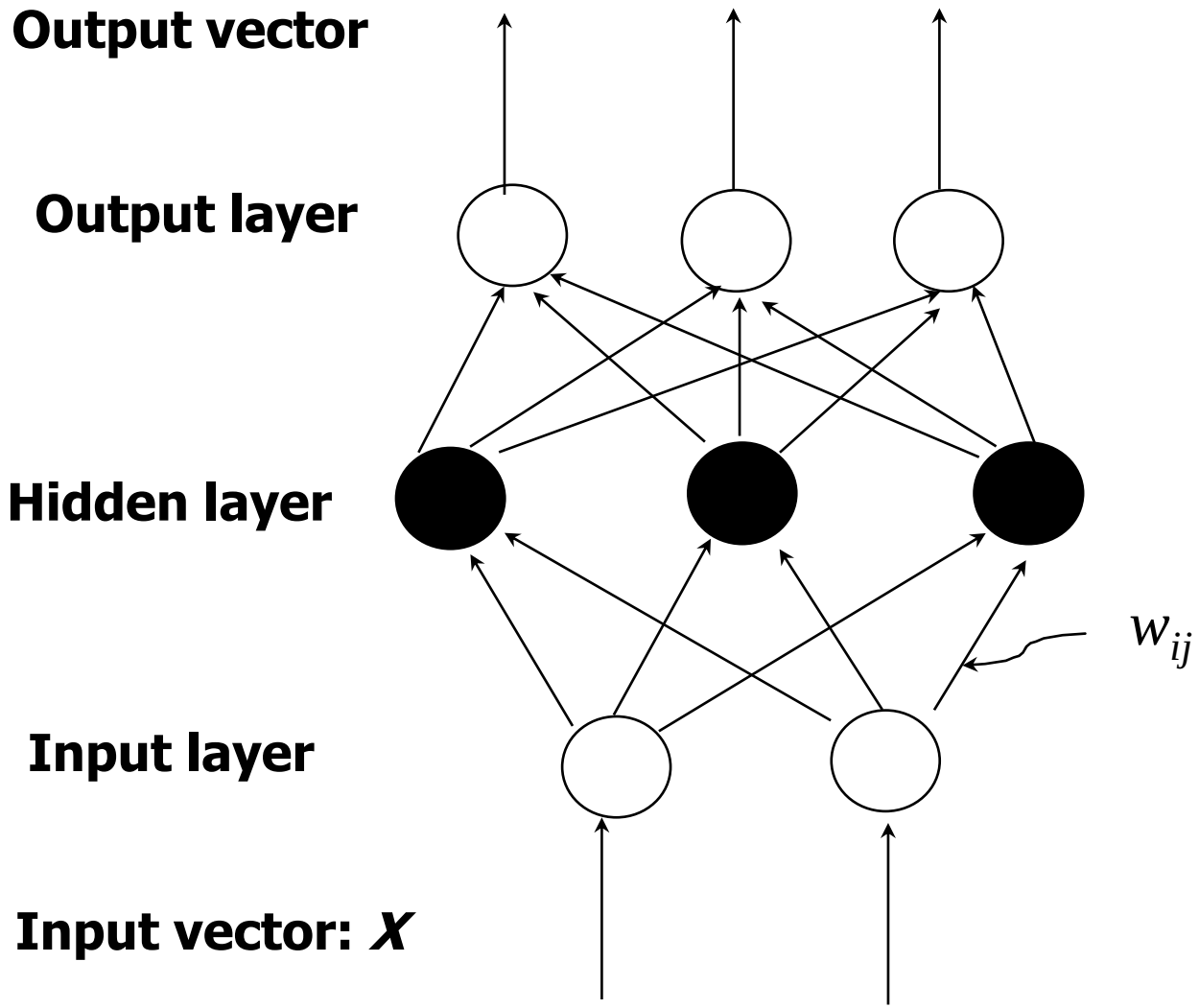


Why nonlinear?



What is a derivative of a sigmoid?

A multi-layer feed-forward NN

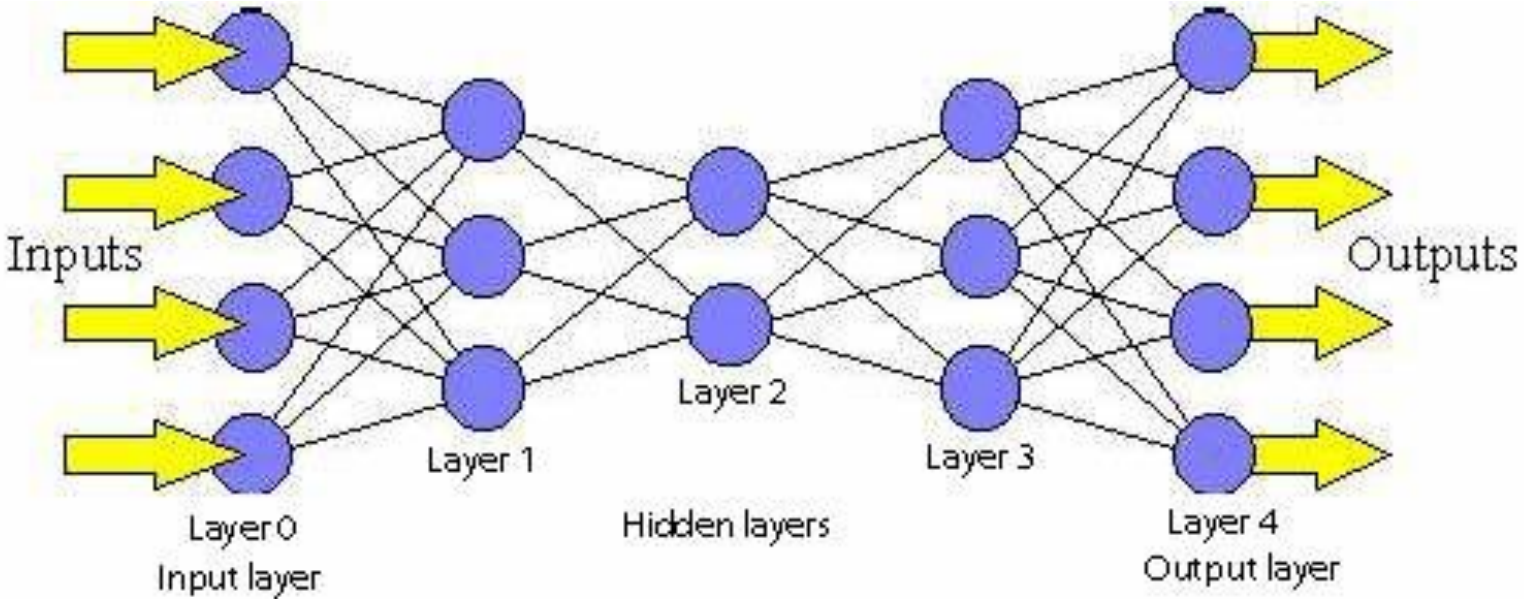


How a multi-layer NN works?

- The **inputs** to the network correspond to the attributes measured for each training tuple
- Inputs are fed simultaneously into the units making up the **input layer**
- They are then weighted and fed simultaneously to a **hidden layer**
- The number of hidden layers is arbitrary; if more than 1 hidden layer is used, the network is called deep neural network
- The weighted outputs of the last hidden layer are input to units making up the **output layer**, which emits the network's prediction
- The network is **feed-forward**: None of the weights cycles back to an input unit or to an output unit of a previous layer
- If we have backwards connections the network is called recurrent neural network
- From a statistical point of view, networks perform **nonlinear regression**: Given enough hidden units and enough training samples, they can closely approximate any function

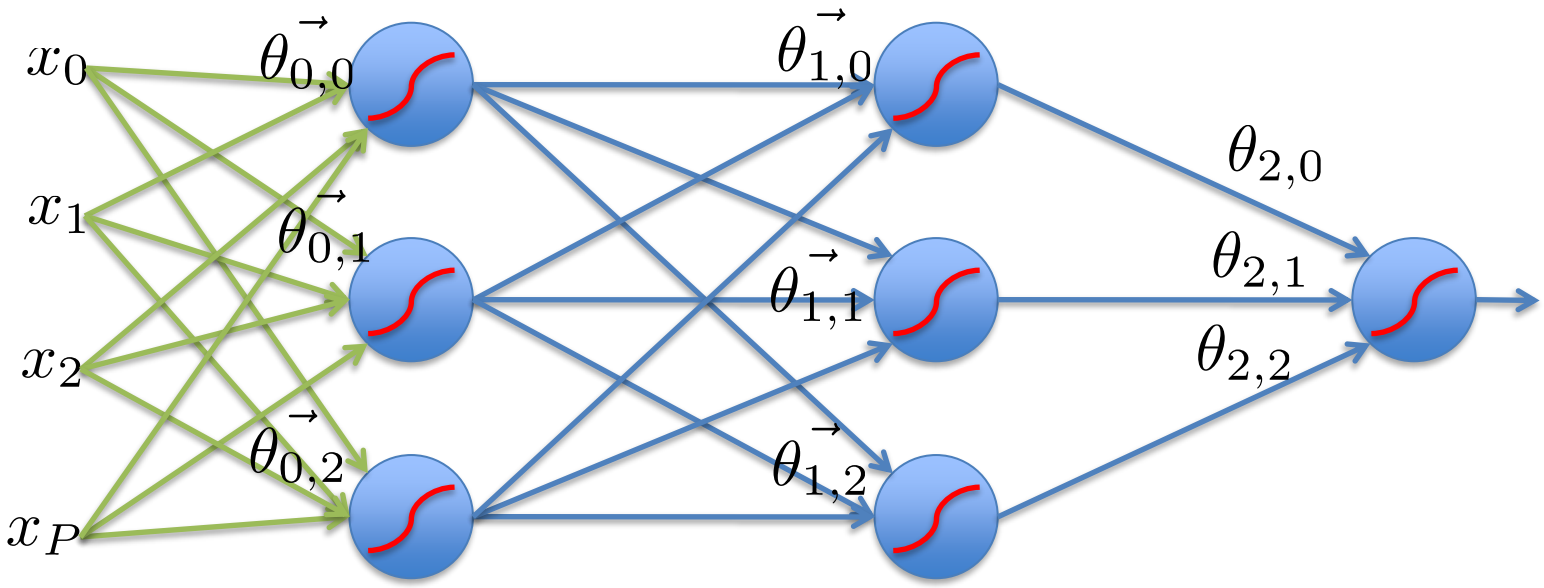
Feed-Forward Network

- neurons are activated progressively throug layers from input to output



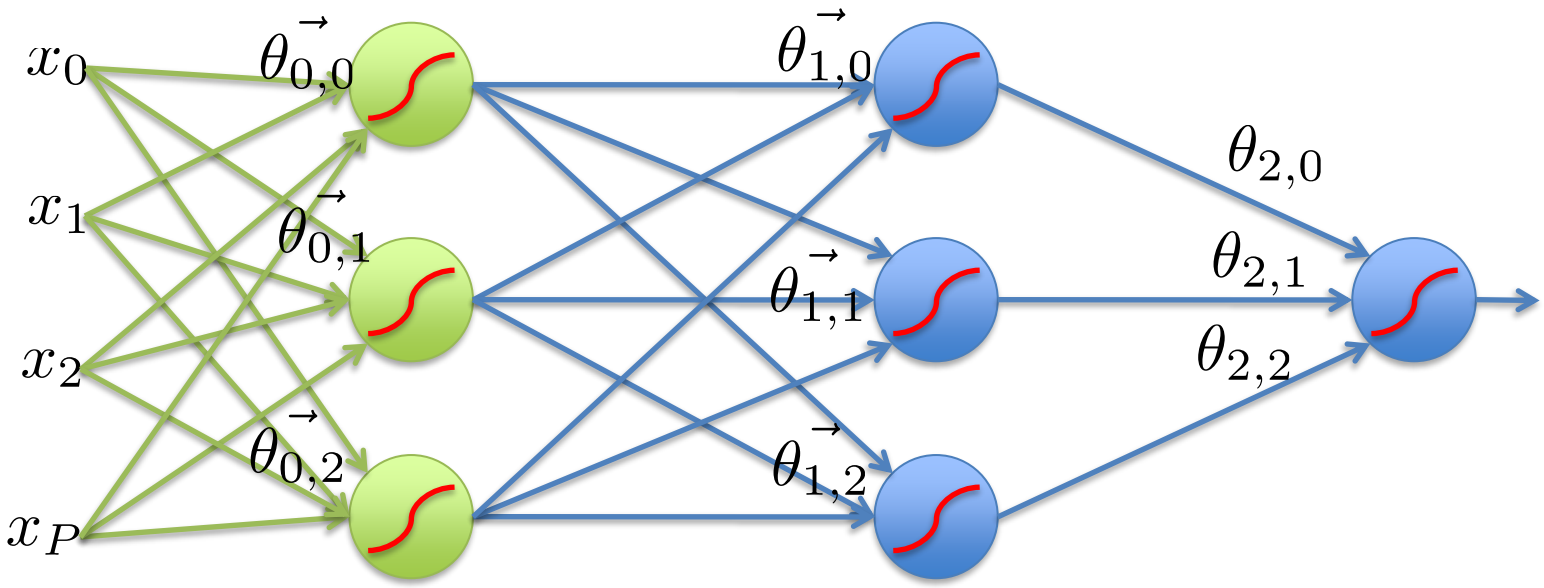
Feed-Forward Network

- Values are propagated through the network to the output, which returns the prediction

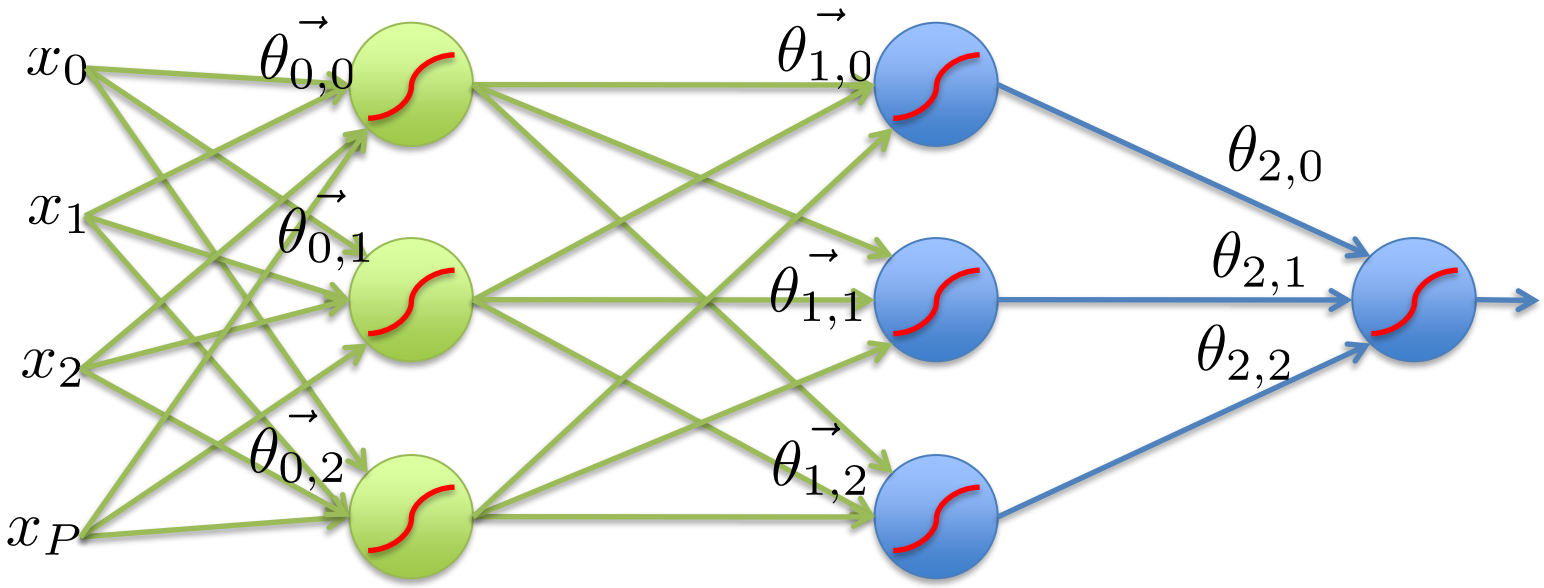


Next 6 slides by Andrew Rosenberg

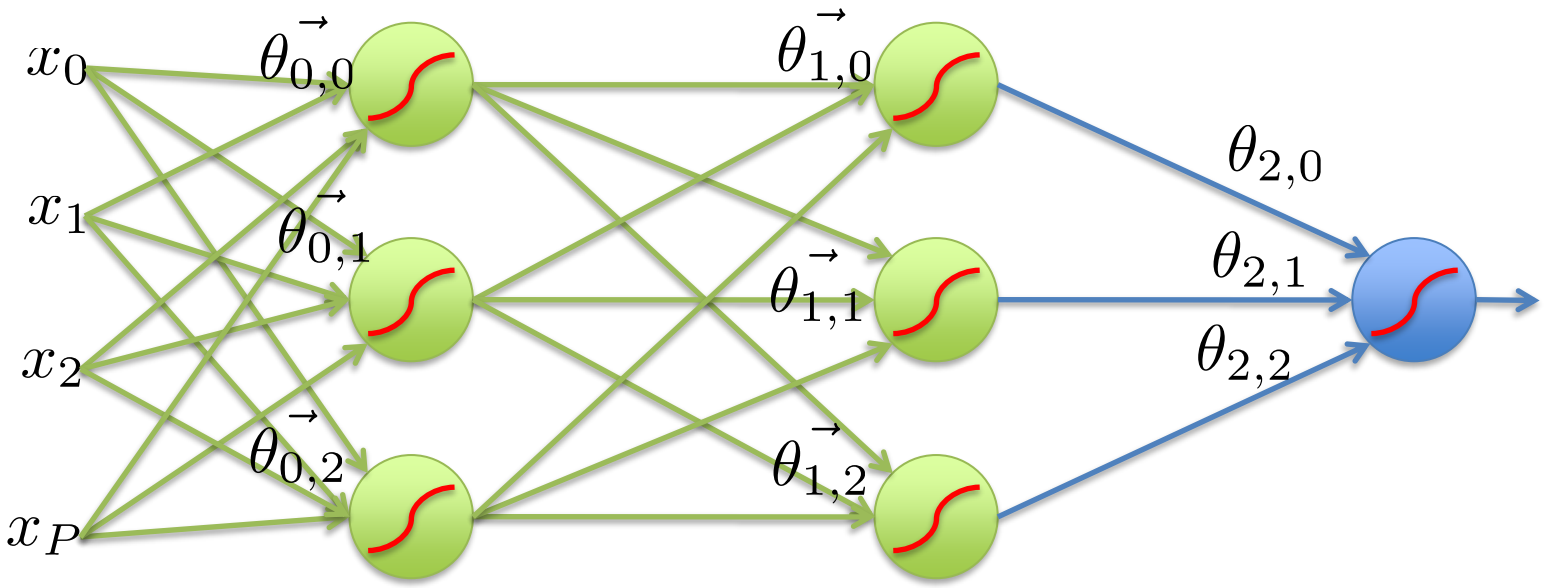
Feed-Forward Networks



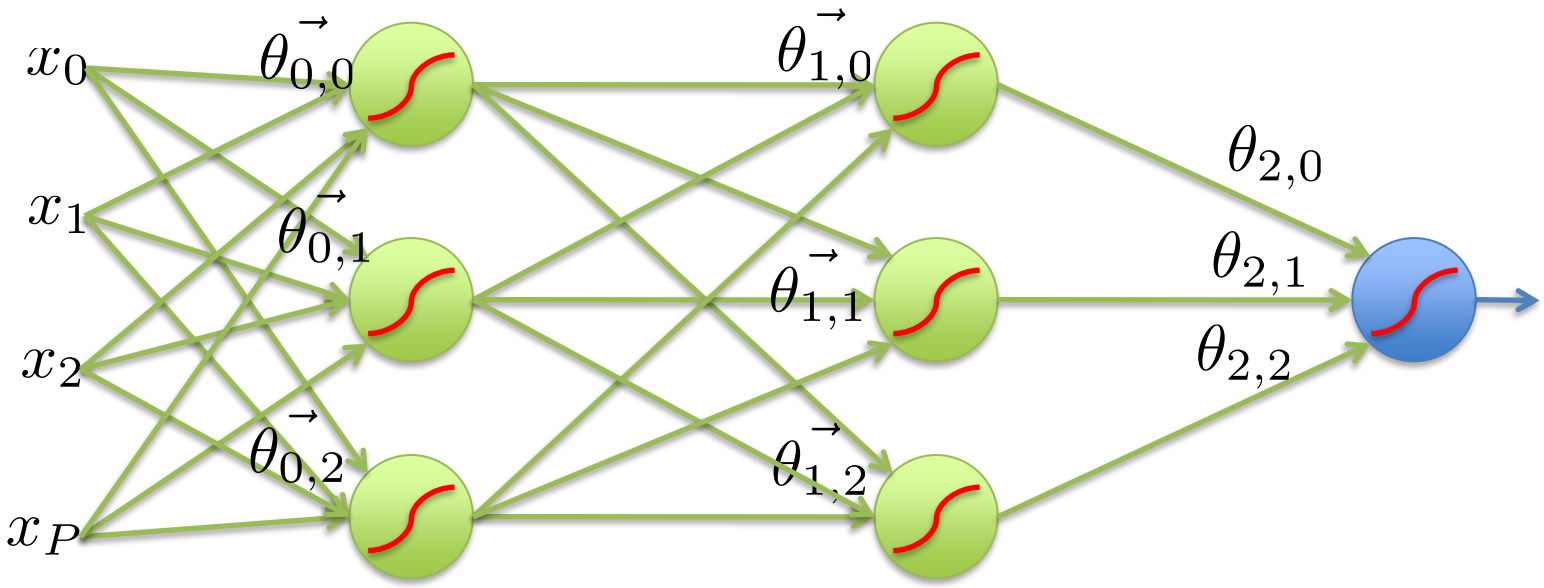
Feed-Forward Networks



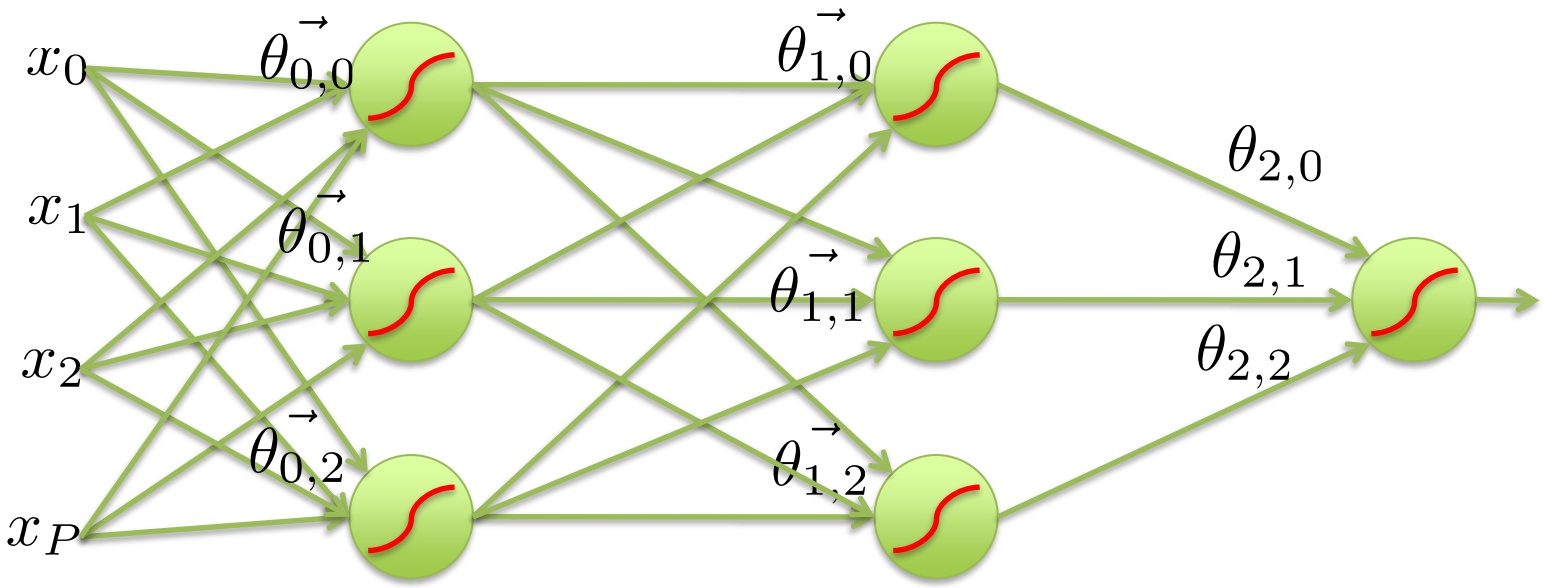
Feed-Forward Networks



Feed-Forward Networks



Feed-Forward Networks



Softmax is often used for the last layer

- normalizes the output scores to be a probability distribution (values between 0 and 1, the sum is 1)

$$y_i = \frac{e^{z_i}}{\sum_{j \in group} e^{z_j}}$$

$$\frac{\partial y_i}{\partial z_i} = y_i \left(1 - y_i\right)$$

Criterion function

- together with softmax we frequently use cross entropy as cost function C

$$C = -\sum_j t_j \log y_j$$

target value

$$\frac{\partial C}{\partial z_i} = \sum_j \frac{\partial C}{\partial y_j} \frac{\partial y_j}{\partial z_i} = y_i - t_i$$

Backpropagation learning algorithm for NN

- Backpropagation: a neural network learning algorithm
- Started by psychologists and neurobiologists to develop and test computational analogues of neurons
- A neural network: a set of connected input/output units where each connection has a weight associated with it
- During the learning phase, the network learns by adjusting the weights so as to be able to predict the correct class label of the input tuples
- Also referred to as connectionist learning due to the connections between units

Backpropagation algorithm

- Iteratively process a set of training tuples & compare the network's prediction with the actual known target value
- For each training tuple, the weights are modified to **minimize the mean squared error** between the network's prediction and the actual target value
- Modifications are made in the "**backwards**" direction: from the output layer, through each hidden layer down to the first hidden layer, hence "**backpropagation**"
- Steps
  - Initialize weights to small random numbers, associated with biases
  - Propagate the inputs forward (by applying activation function)
  - Backpropagate the error (by updating weights and biases)
  - Terminating condition (when error is very small, etc.)

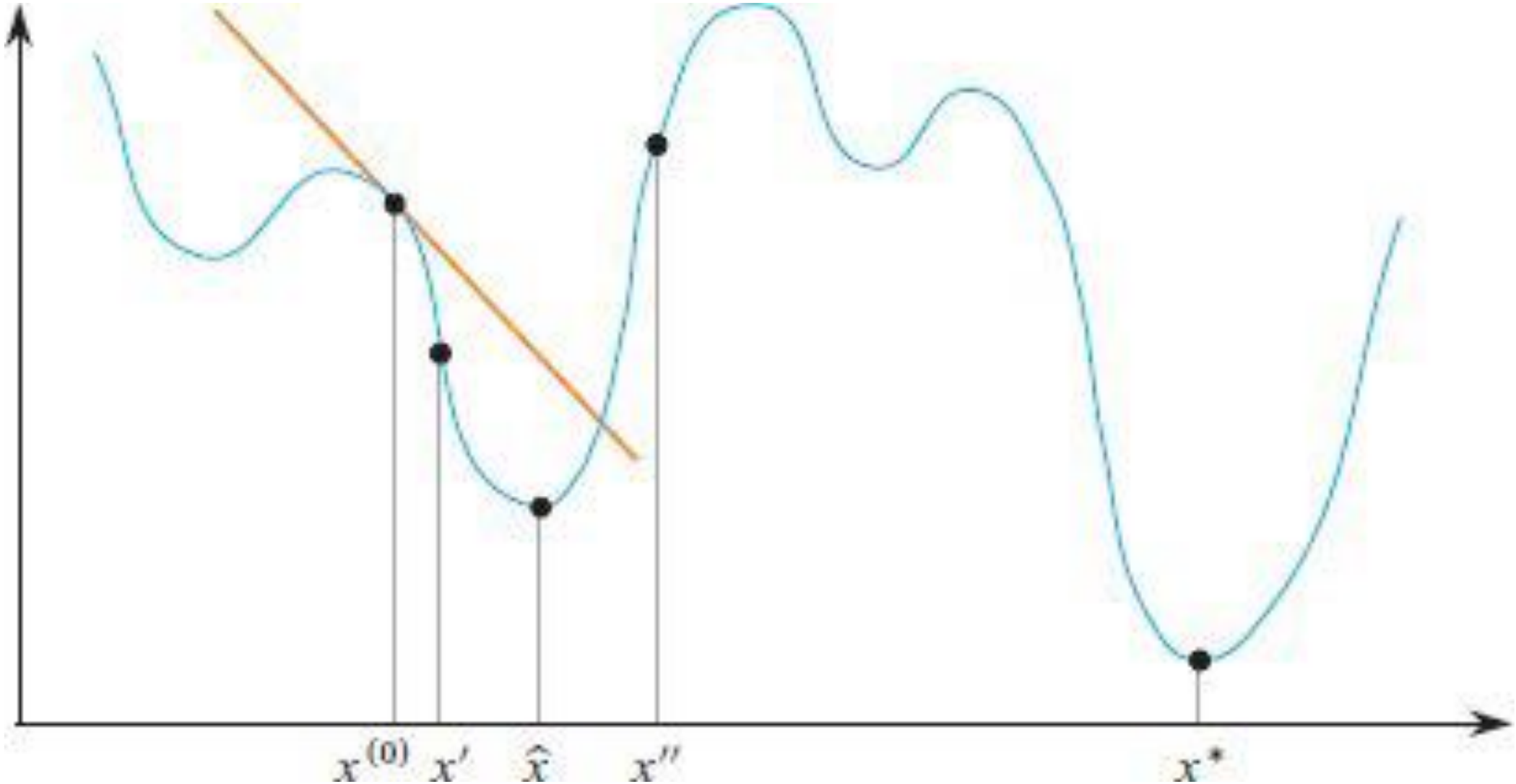
Gradient descent (GD)

- Gradient descent is an efficient local optimization in $\mathbb{R}^n$
- Local minimum of function *f:* $\mathbb{R}^n \rightarrow \mathbb{R}$ is a point ***x*** for which *f(x) ≤ f(x′)* for all ***x′*** that are "near" ***x***
- Gradient $\nabla f(x)$ is a function ∇f: $\mathbb{R}^n \rightarrow \mathbb{R}^n$ comprising *n* partial derivatives:

$$\nabla f(x) = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n})$$

- The GD optimization moves in the direction of -$\nabla f(x)$

Illustration of GD



GD algorithm

```
GRADIENT-DESCENT(f, x0, γ, T) {
  // function f,  initial value x0, fixed step size γ, number of steps T
  x_best = x = x0 ; // n-dimensional vectors, initially set to the initial value
  f_best = f_x = f(x_best) ;
  for t = 0 to T – 1 do {
    x_next = x – γ · ∇f(x);  // ∇f(x), x, and x_next are n-dimensional
    f_next = f(x_next)
    if (f_next < f_x)
      x_best = x_next ;
    x = x_next ;
    f_x = f_next ;
  }
  return x_best ;
}
```

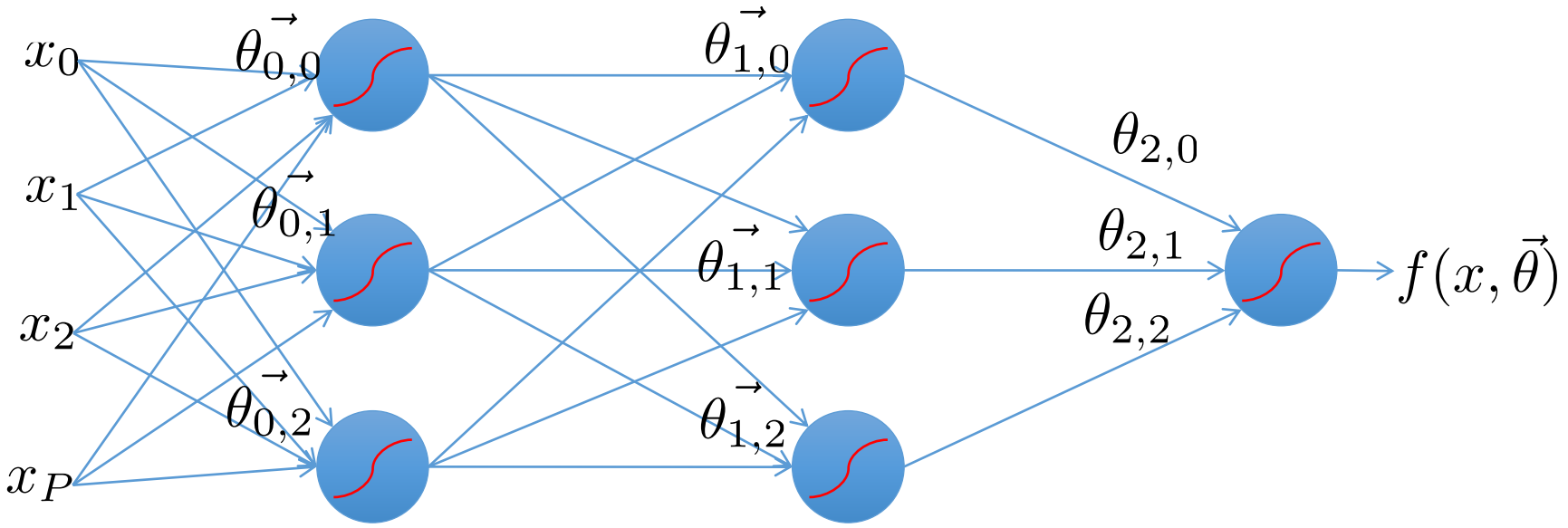
Chain rule of derivation

- In a network, the output of each neuron is a function of activation function and all its inputs, where the inputs may again be composite functions of neurons in previous layers
- To compute a gradient of a composite function, we use the chain rule of derivation

$$f\big(g(x)\big)' = f'\big(g(x)\big)g'(x)$$

Error Backpropagation

- We will do gradient descent on the whole network.
- Training will proceed from the last layer to the first.

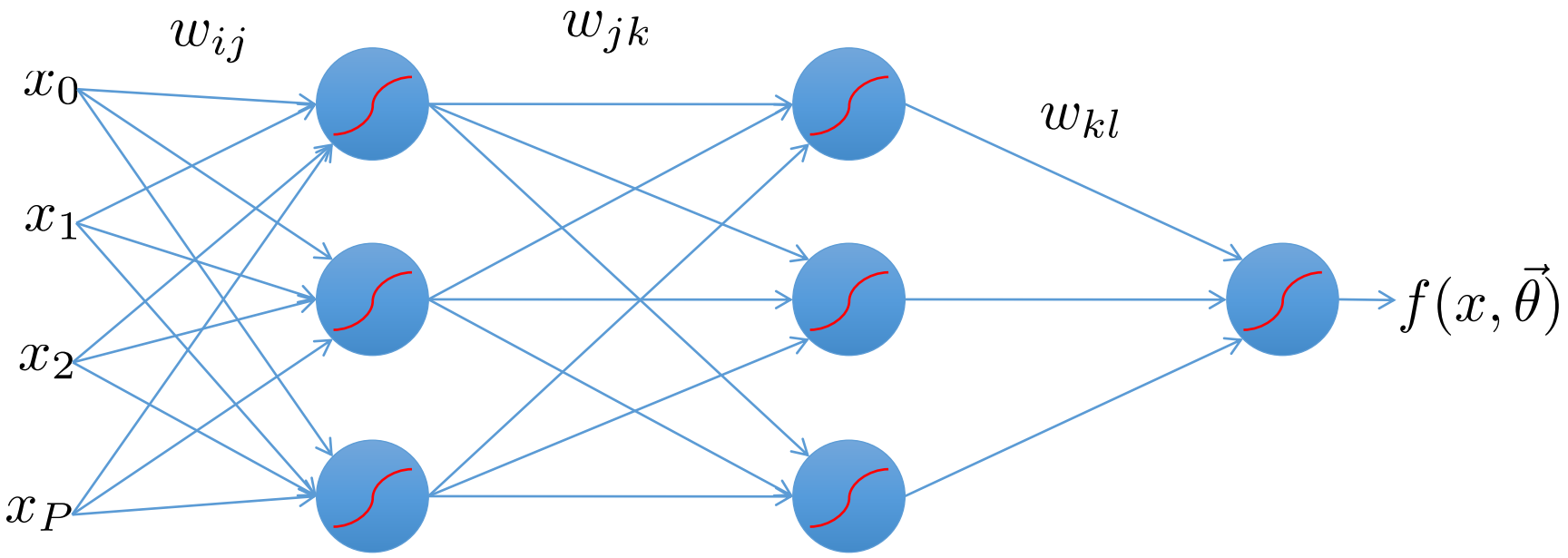


Next 18 slides by Andrew Rosenberg

Error Backpropagation

- Introduce variables over the neural network

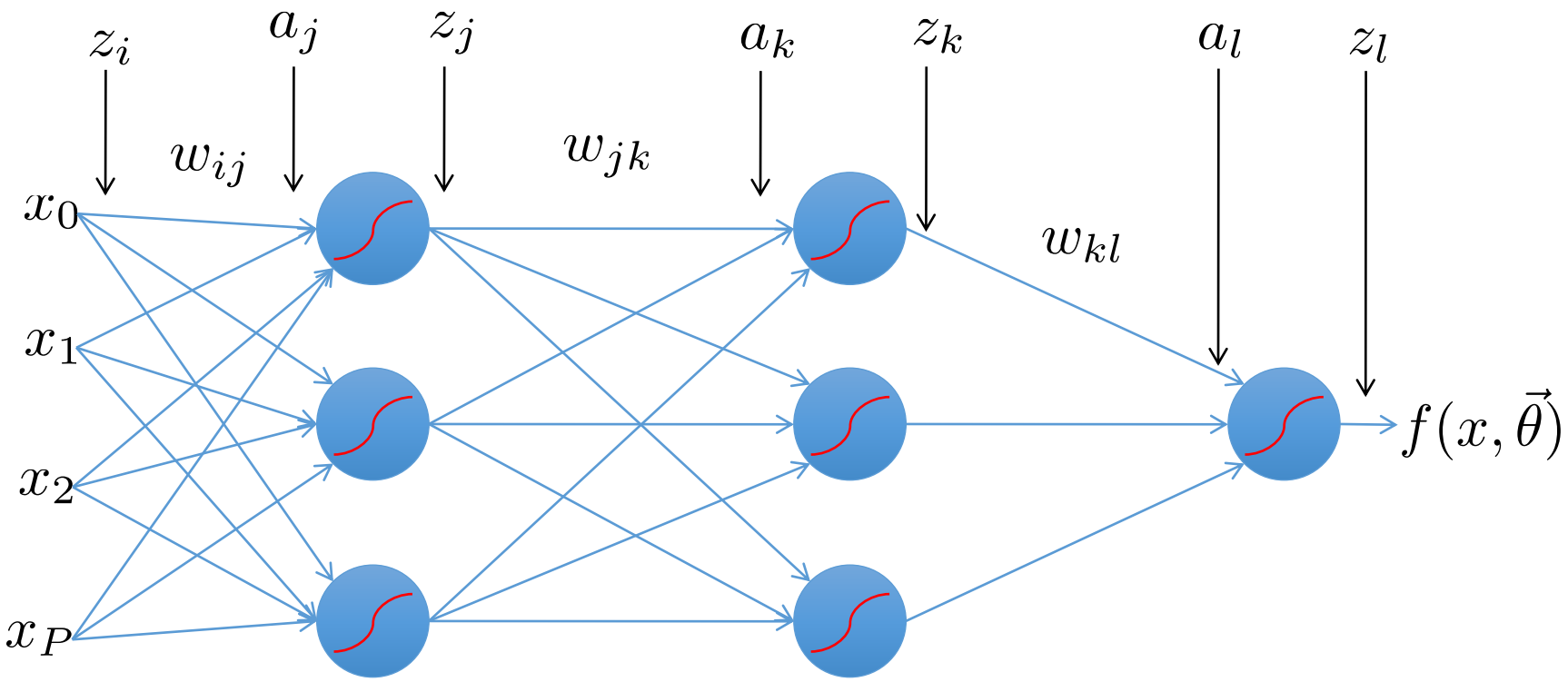
$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$



Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

- Introduce variables over the neural network
  - Distinguish the input and output of each node

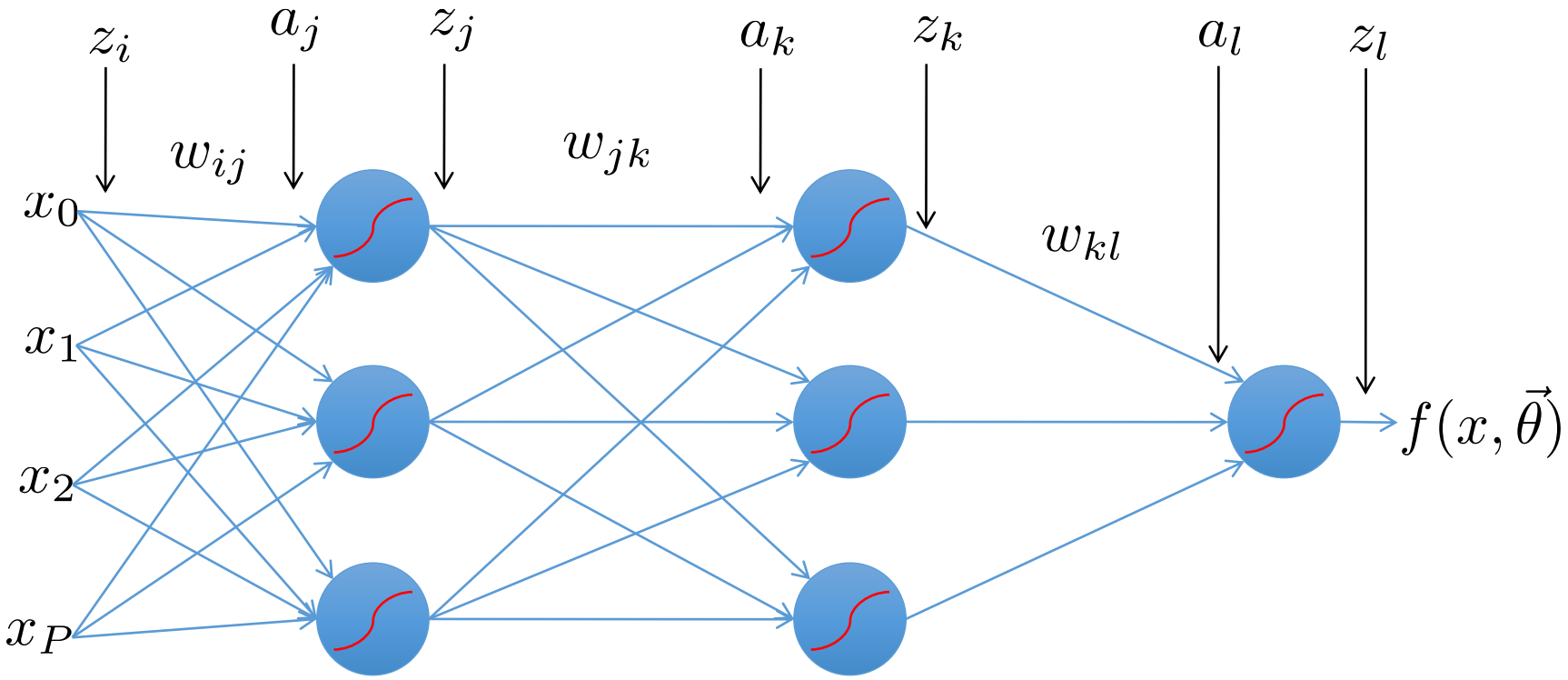


Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \sum_j w_{jk} z_j \qquad a_l = \sum_k w_{kl} z_k$$

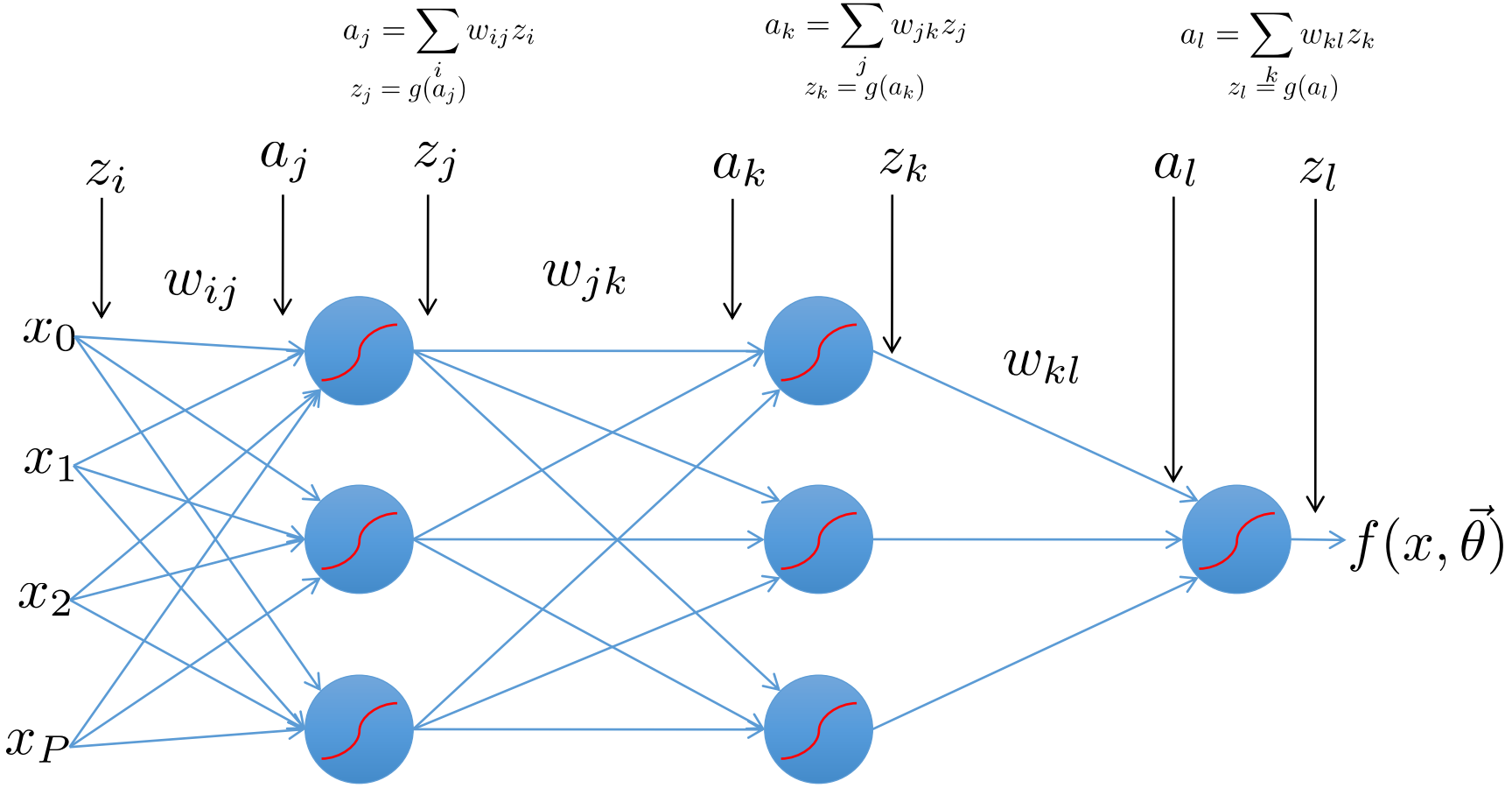
$$z_j = g(a_j) \qquad z_k = g(a_k) \qquad z_l = g(a_l)$$



Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

Training: Take the gradient of the last component and iterate backwards



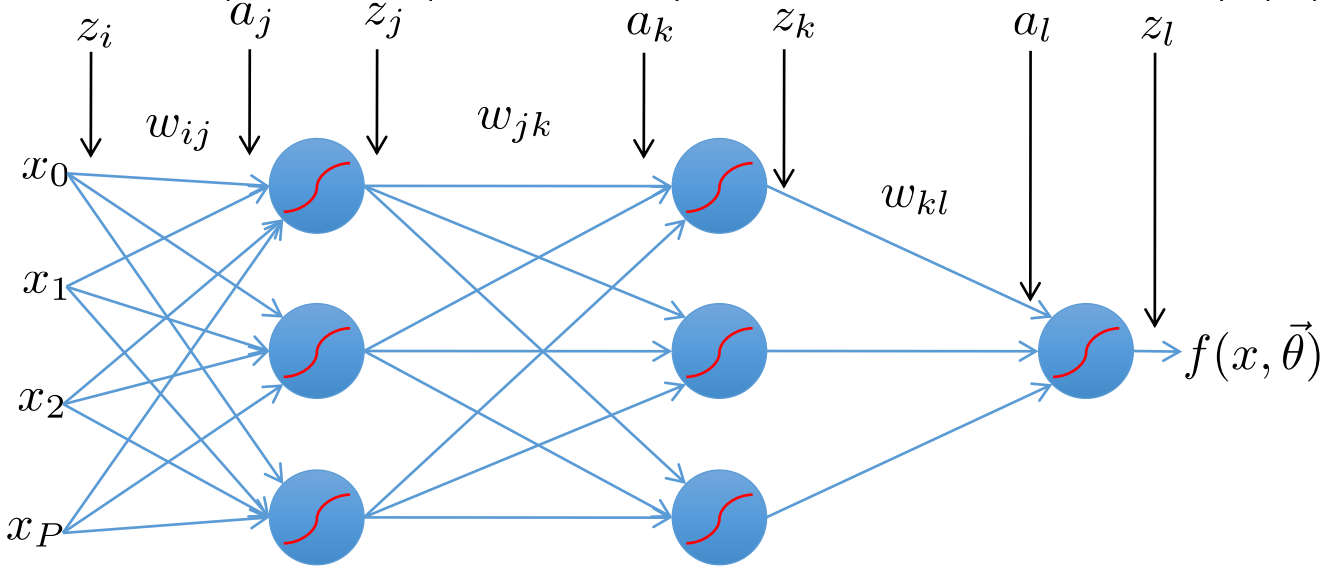
Error Backpropagation

$$R(\theta) = \frac{1}{N}\sum_{n=0}^{N} L(y_n - f(x_n))$$

Empirical Risk Function

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - g\left(\sum_k w_{kl} g\left(\sum_j w_{jk} g\left(\sum_i w_{ij} x_{n,i}\right)\right)\right)\right)^2$$



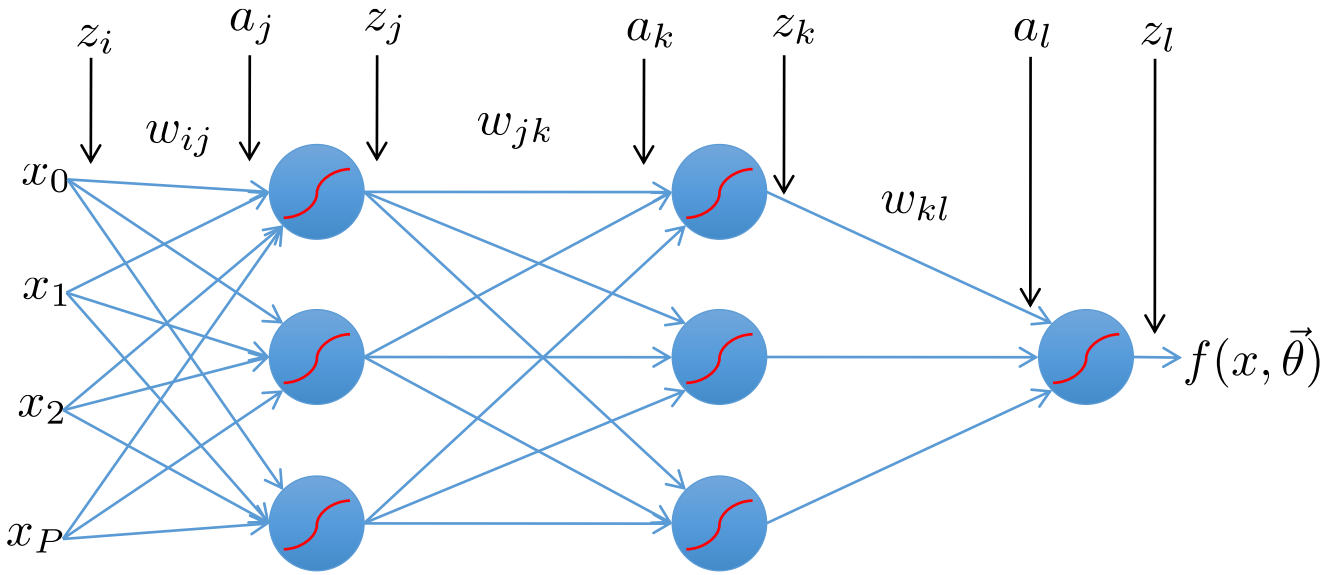
Error Backpropagation

Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule



Error Backpropagation

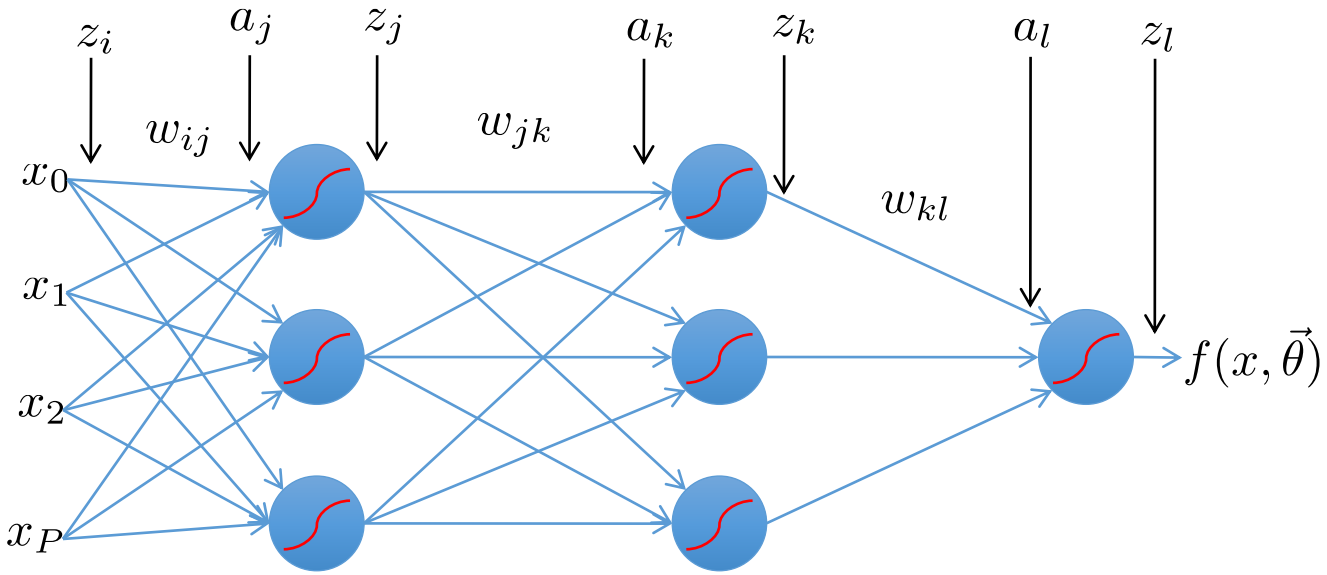
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$



Error Backpropagation

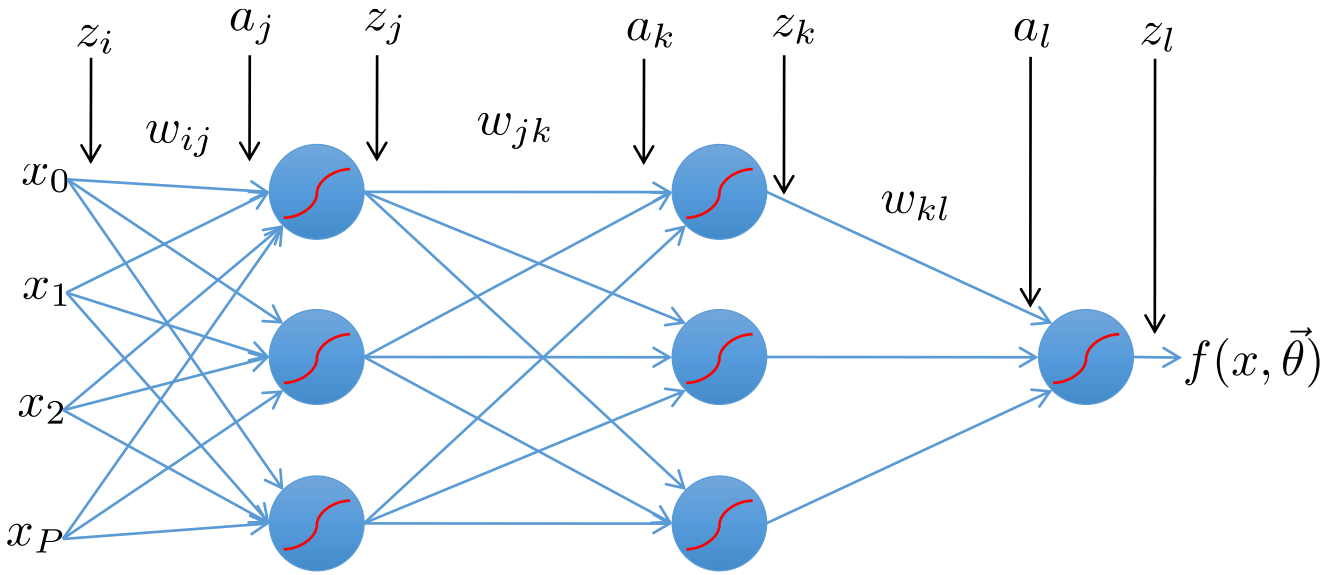
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right]$$



Error Backpropagation

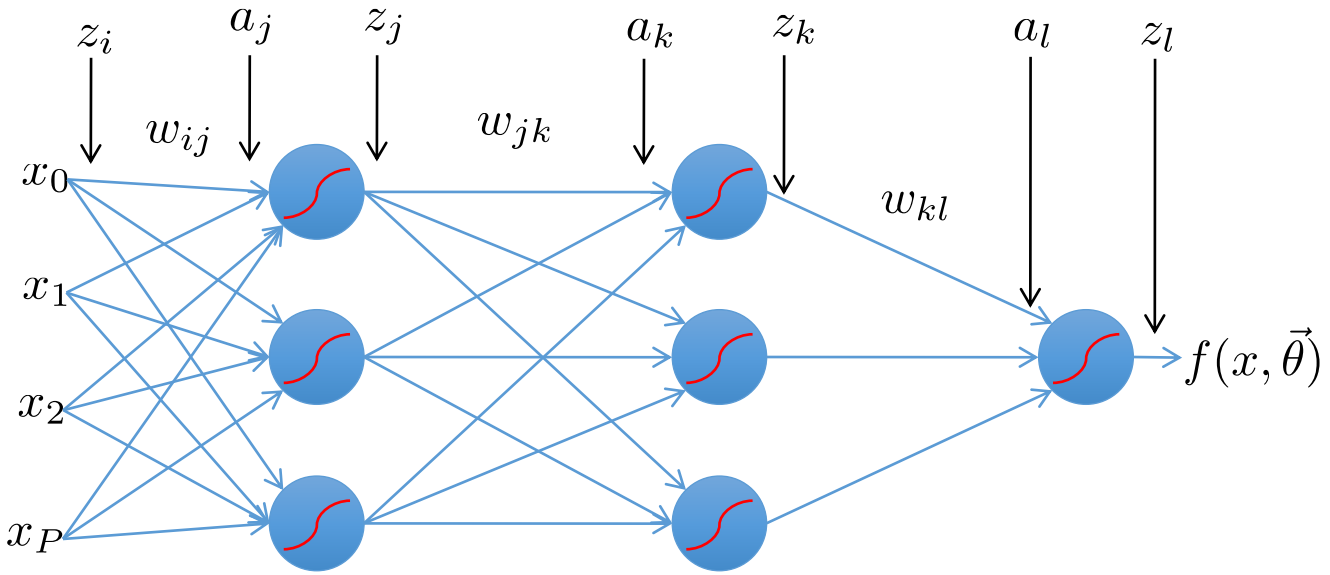
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n}$$



Error Backpropagation

Optimize last layer weights w_{kl}

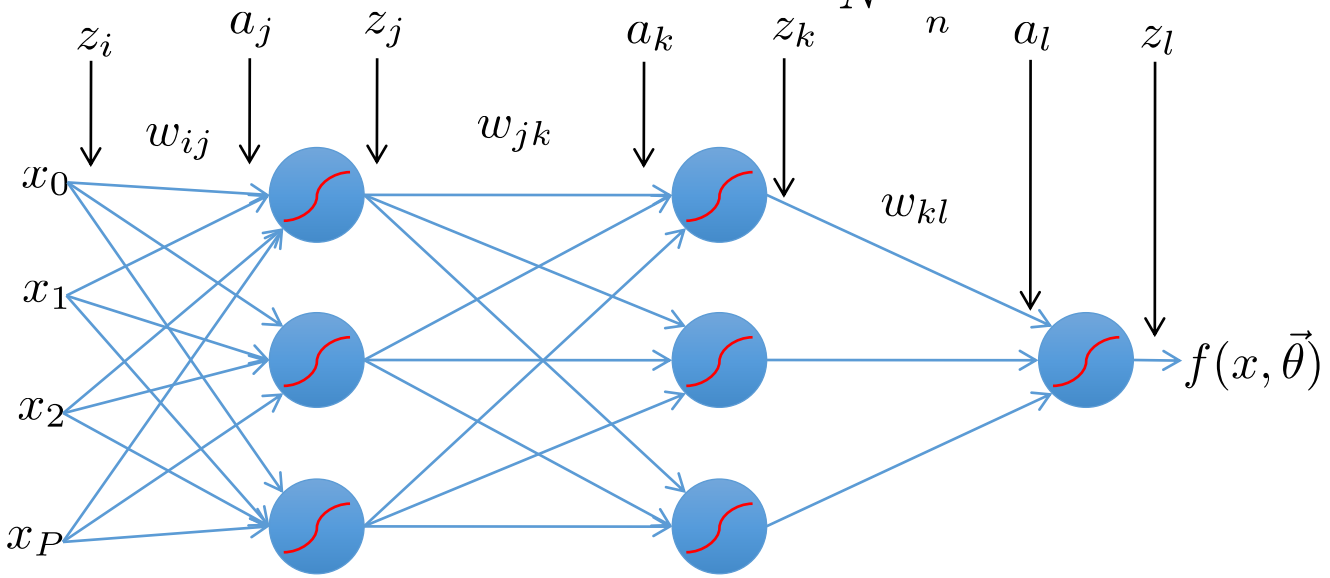
$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n}$$

$$= \frac{1}{N}\sum_n \delta_{l,n} n z_{k,n}$$

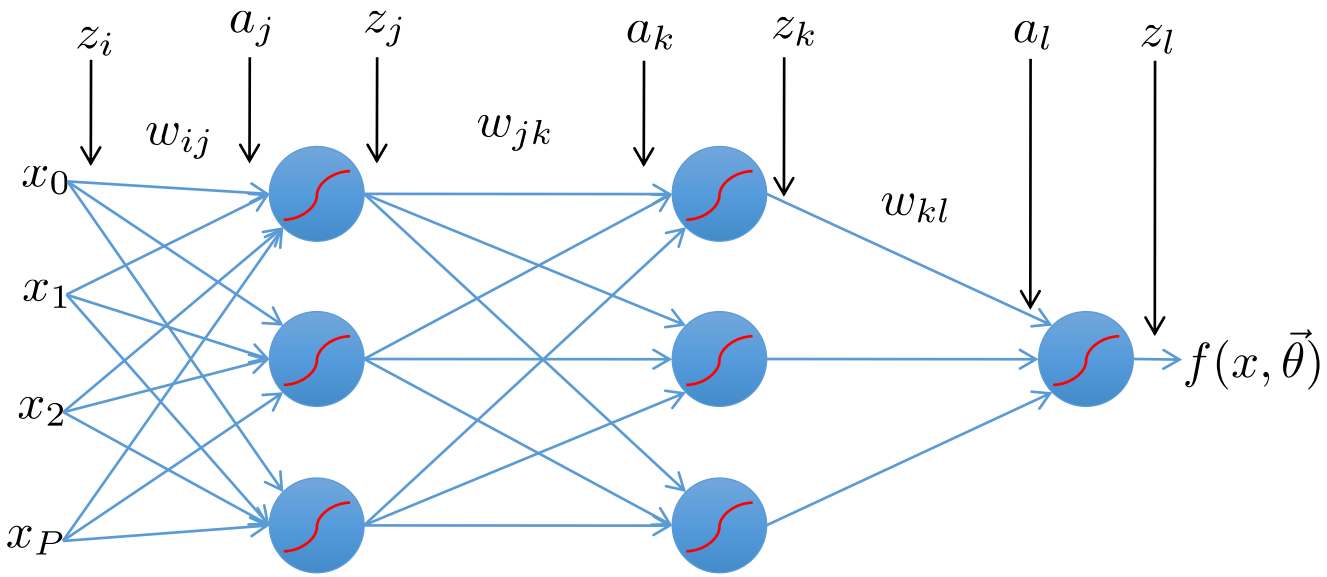


Error Backpropagation

Optimize last hidden weights w_{jk}

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$



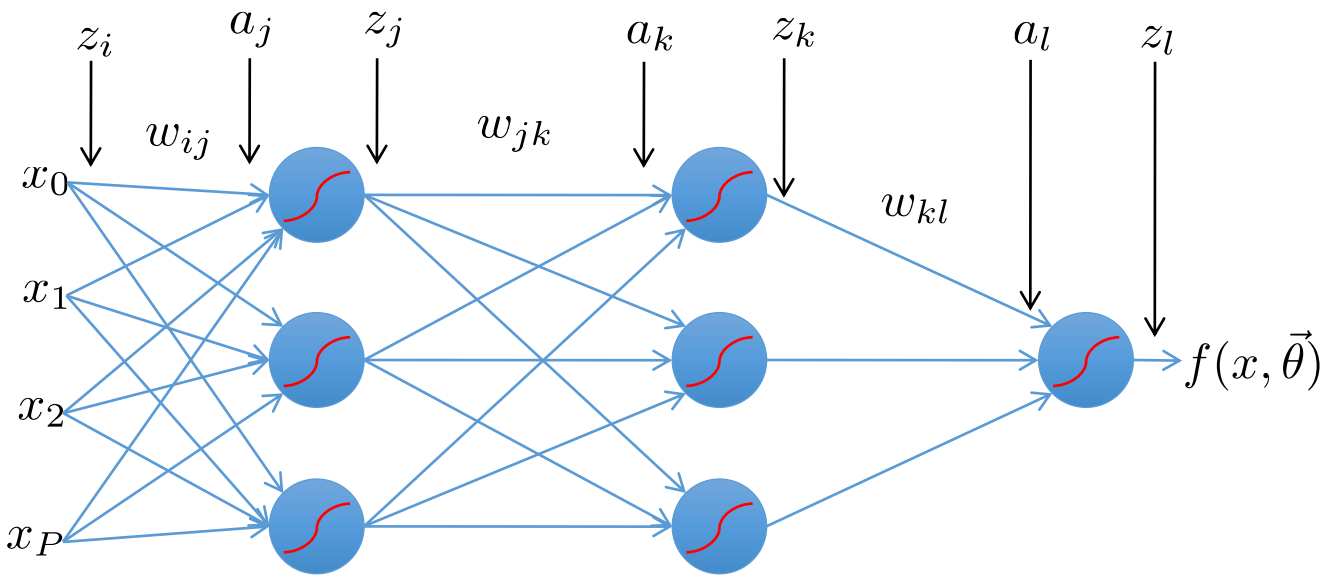
Error Backpropagation

Optimize last hidden weights w_{jk}

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] \left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule



Error Backpropagation

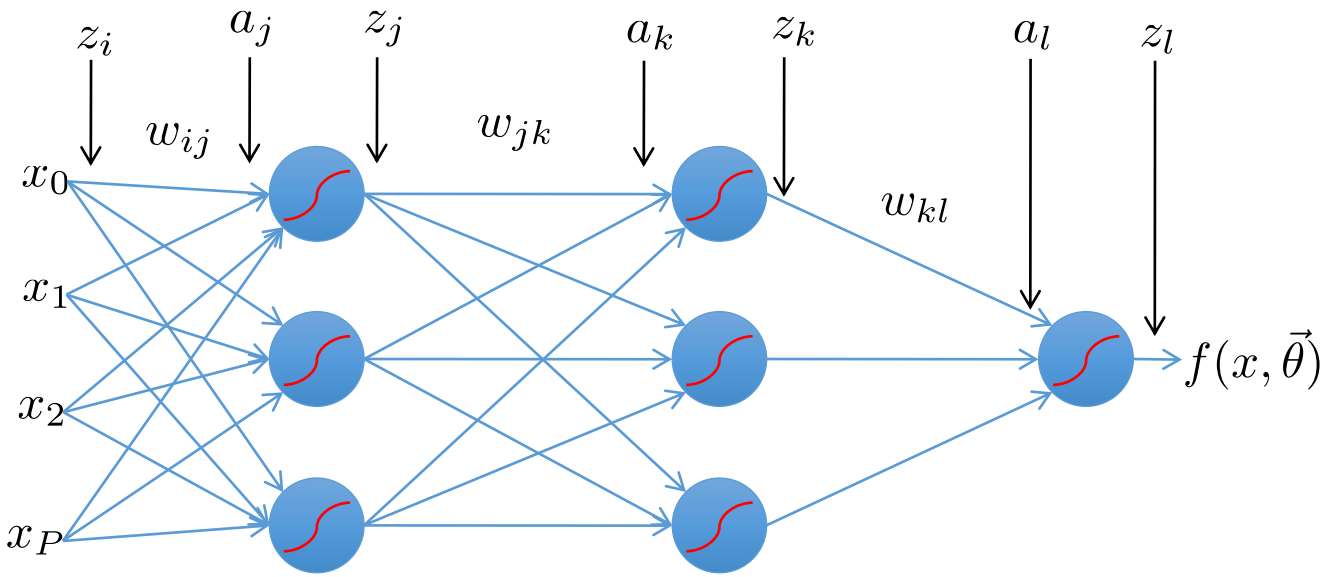
Optimize last hidden weights w_{jk}

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] \left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_l \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] [z_{j,n}]$$



Error Backpropagation

Optimize last hidden weights w_{jk}

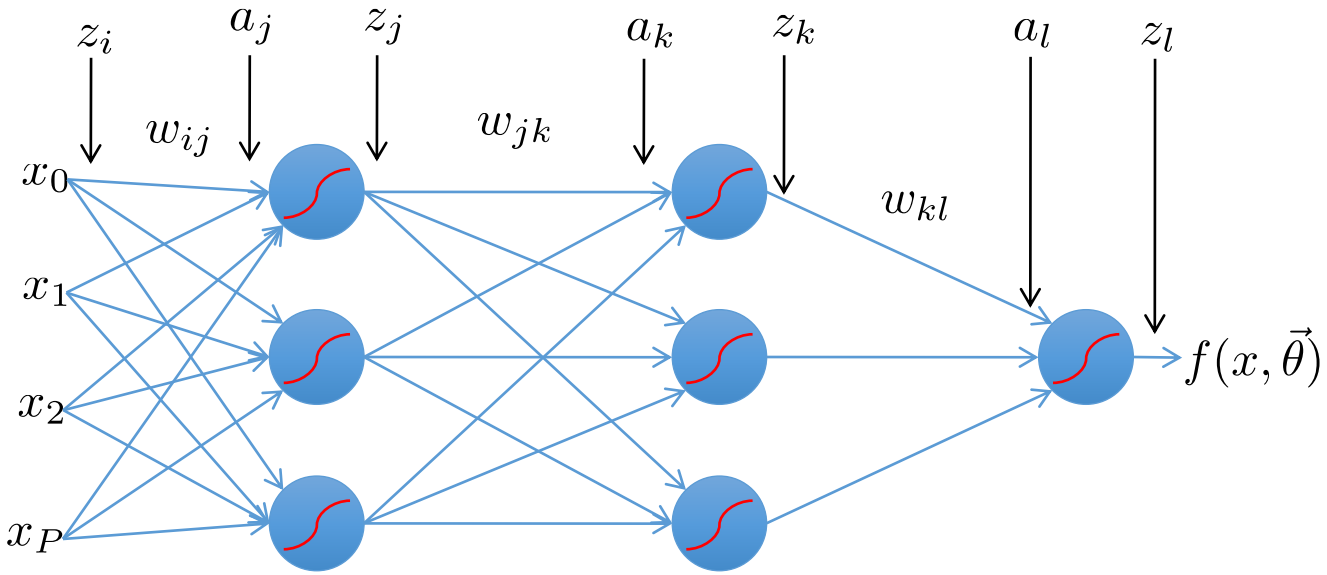
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] \left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_l \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] [z_{j,n}]$$

$$a_l = \sum_k w_{kl} g(a_k)$$



Error Backpropagation

Optimize last hidden weights w_{jk}

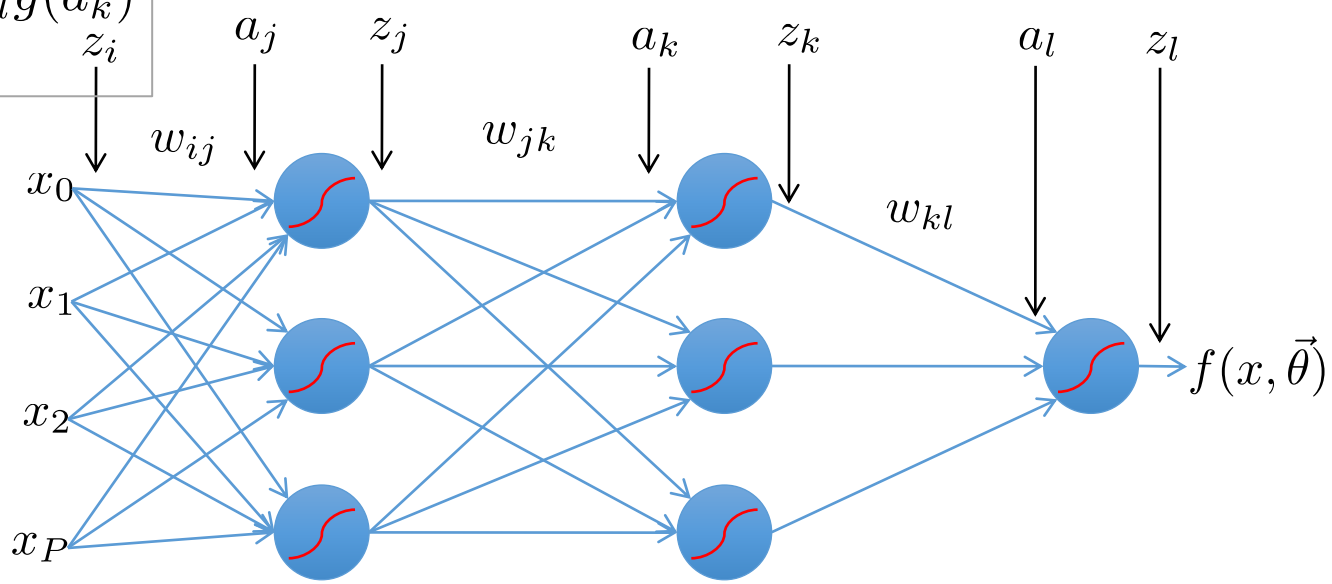
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}}\frac{\partial a_{l,n}}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_l w_{kl} g'(a_{k,n})\right][z_{j,n}] = \frac{1}{N}\sum_n [\delta_{k,n}][z_{j,n}]$$

$$a_l = \sum_k w_{kl} g(a_k)$$



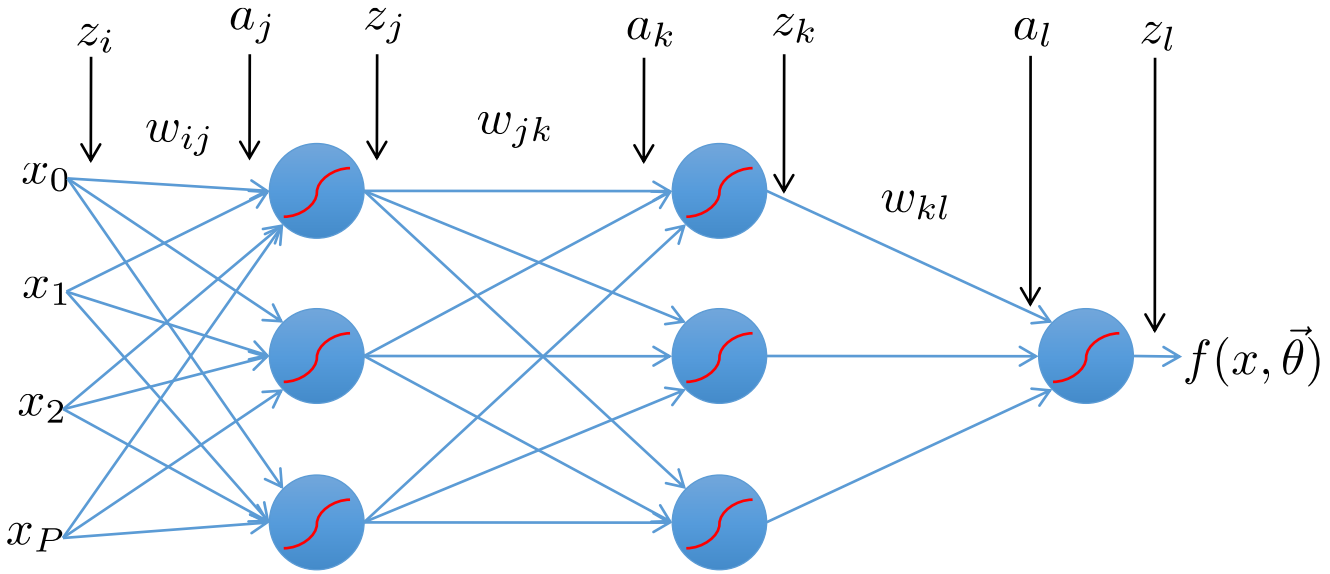
Error Backpropagation

Repeat for all previous layers

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right] = \frac{1}{N}\sum_n \left[\sum_l \delta_{l,n} w_{kl} g'(a_{k,n})\right] z_{j,n} = \frac{1}{N}\sum_n \delta_{k,n} z_{j,n}$$

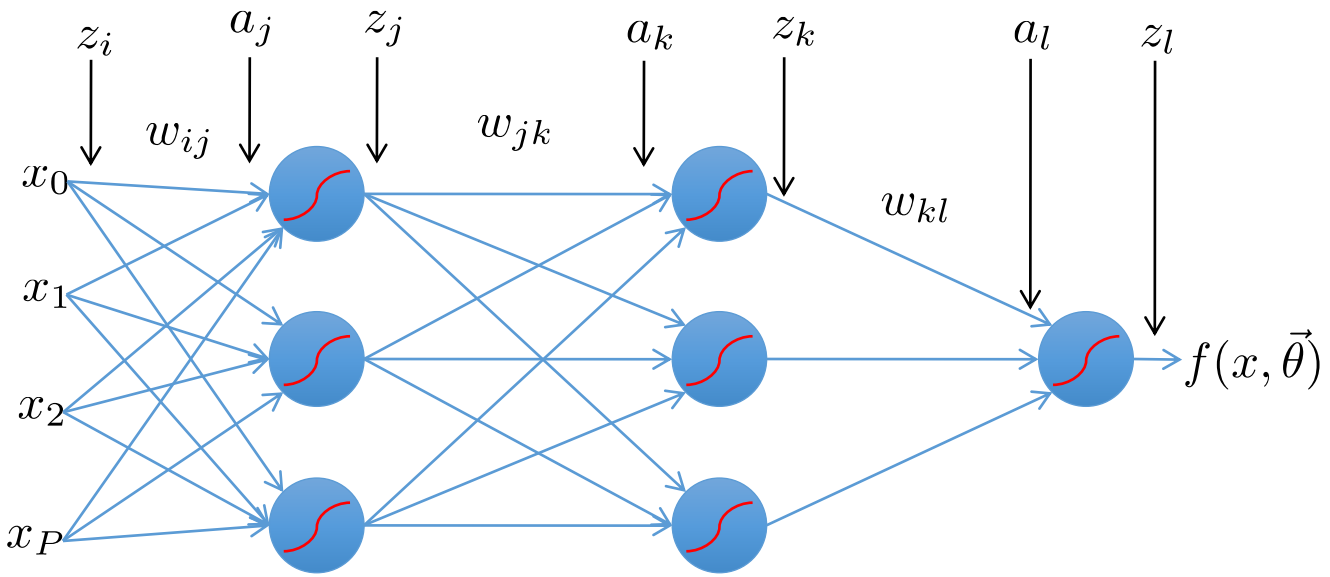
$$\frac{\partial R}{\partial w_{ij}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{j,n}}\right]\left[\frac{\partial a_{j,n}}{\partial w_{ij}}\right] = \frac{1}{N}\sum_n \left[\sum_k \delta_{k,n} w_{jk} g'(a_{j,n})\right] z_{i,n} = \frac{1}{N}\sum_n \delta_{j,n} z_{i,n}$$



Error Backpropagation

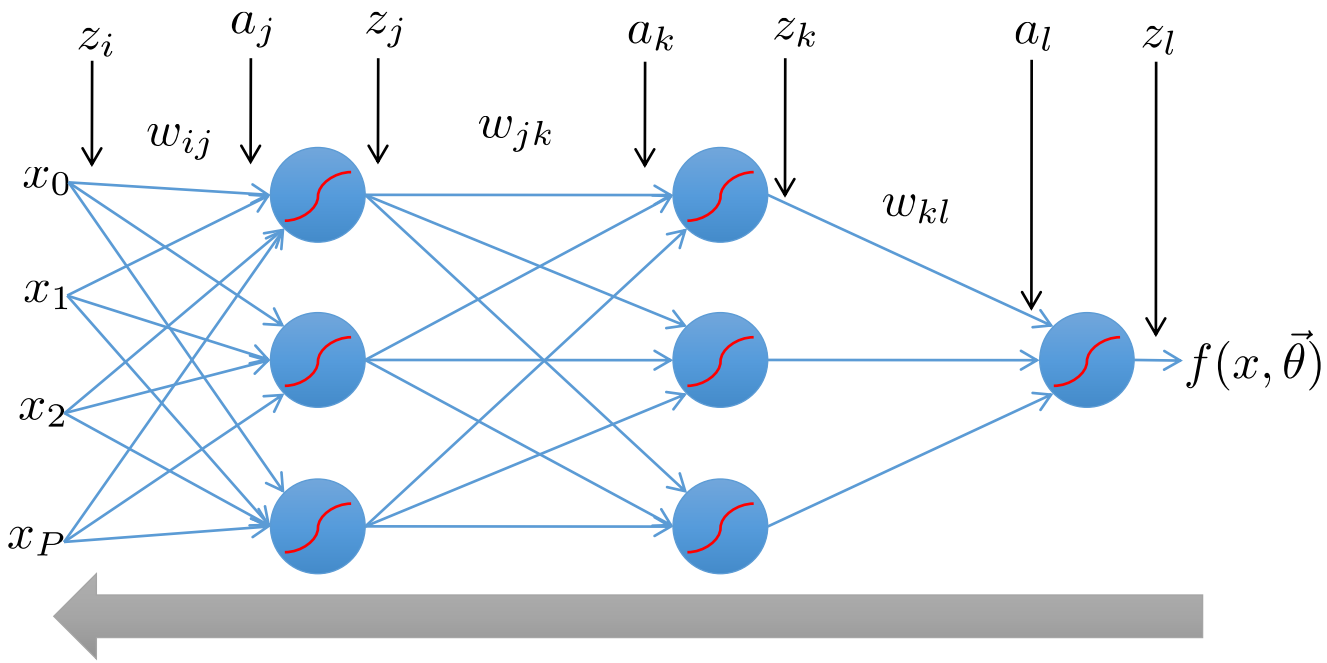
Now that we have well defined gradients for each parameter, update using Gradient Descent

$$
\begin{aligned}
w_{ij}^{t+1} &= w_{ij}^{t} - \eta \frac{\partial R}{w_{ij}} \\
w_{jk}^{t+1} &= w_{jk}^{t} - \eta \frac{\partial R}{w_{jk}} \\
w_{kl}^{t+1} &= w_{kl}^{t} - \eta \frac{\partial R}{w_{kl}}
\end{aligned}
$$



Error Back-propagation

- Error backpropagation unravels the multivariate chain rule and solves the gradient for each partial component separately.
- The target values for each layer come from the next layer.
- This feeds the errors back along the network.

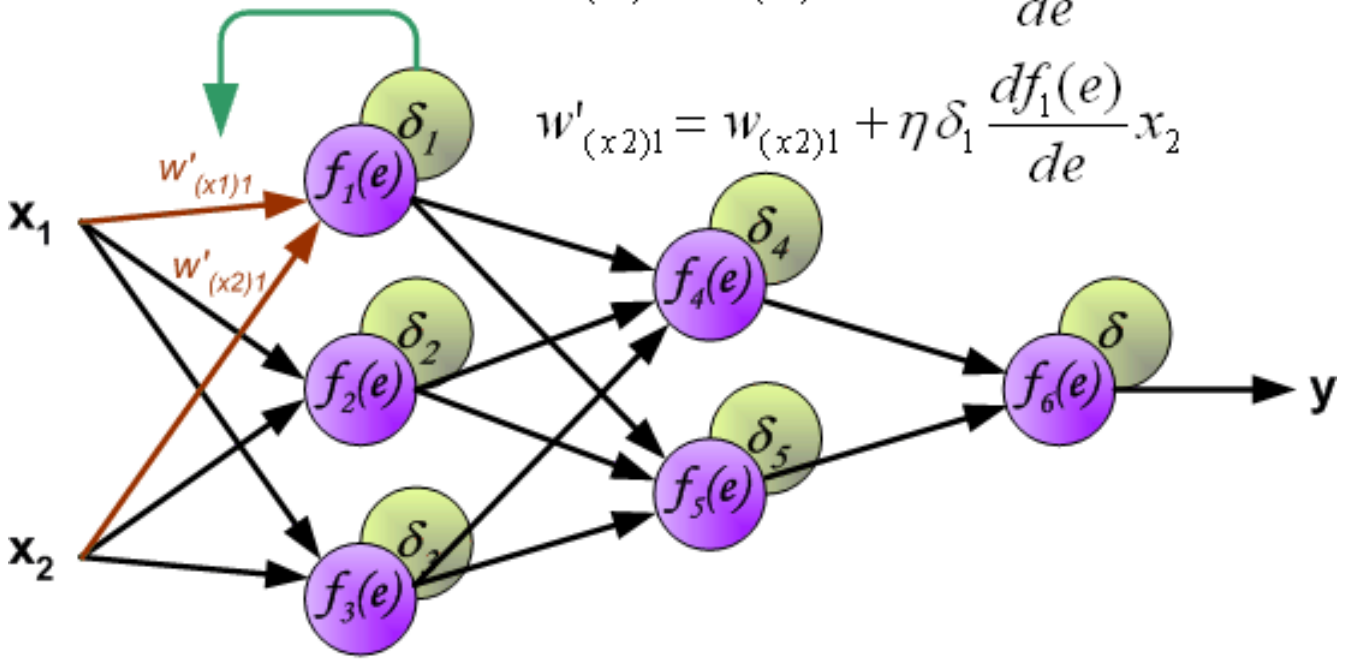


Learning with error backpropagation

- Backpropagation
- randomly initialize parameters (weights)
- compute error on the output
- compute contributions to error, δ_n, on each step backwards

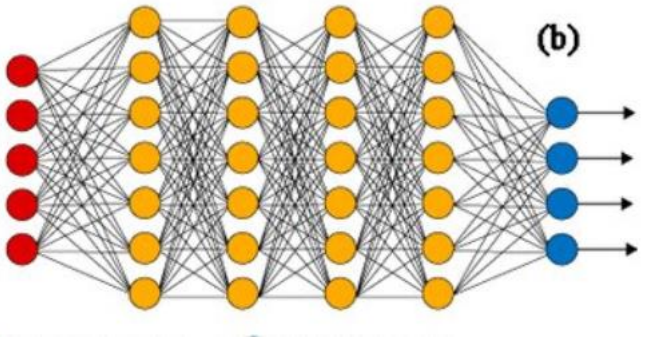
$$w'_{(x1)1} = w_{(x1)1} + \eta\delta_1 \frac{df_1(e)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta\delta_1 \frac{df_1(e)}{de} x_2$$



- step
- epoch
- batch
- minibatch

Defining a network topology



- Decide the network topology:
  Specify # of units in the *input layer*, # of *hidden layers* (if > 1), # of units in *each hidden layer*, and # of units in the *output layer*
- Normalize the input values for each attribute measured in the training tuples to [0.0—1.0]
- One input unit per discrete attribute value, 1-hot encoded
- For classification and more than two classes, one output unit per class is used
- Once a network has been trained and its accuracy is unacceptable, repeat the training process with a *different network topology* or a *different set of initial weights*

Neural network as a classifier for tabular data

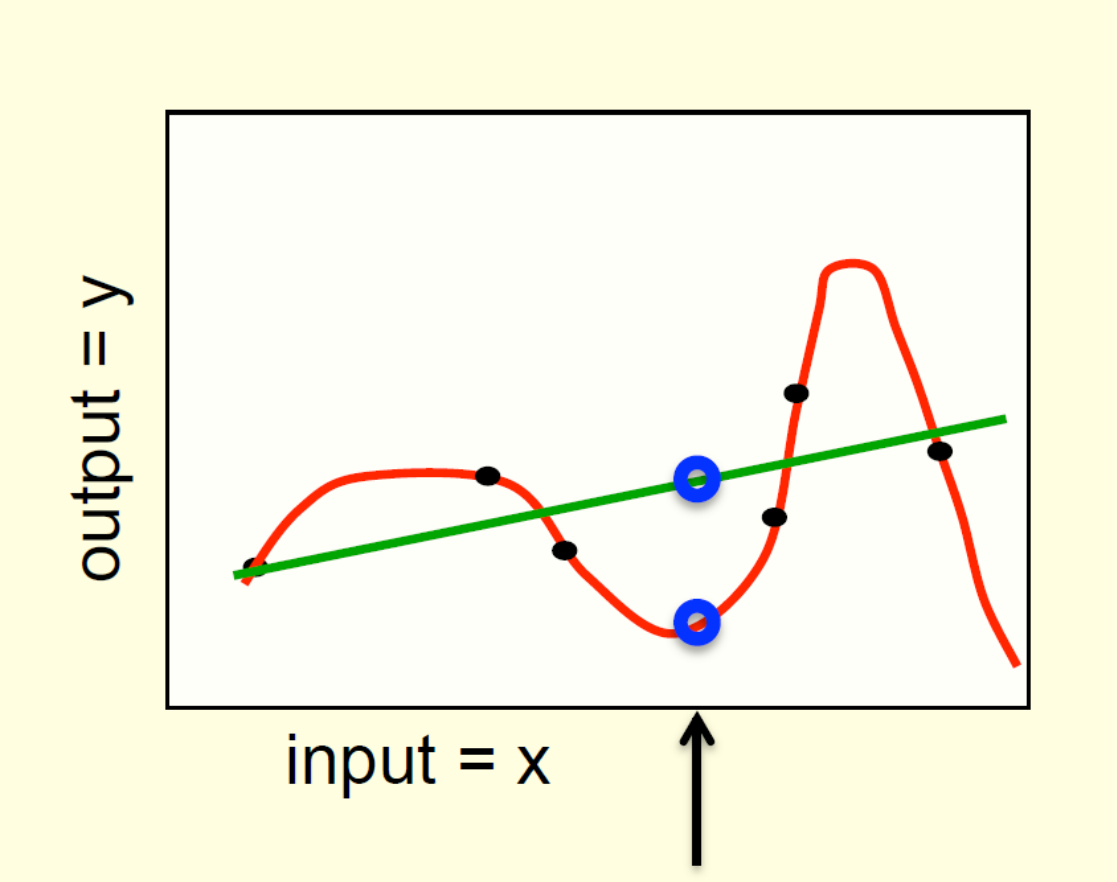
- Weakness
  - Long training time
  - Require a number of parameters typically best determined empirically, e.g., the network topology or “structure.”
  - Poor interpretability: difficult to interpret the symbolic meaning behind the learned weights and of “hidden units” in the network
  - Easy to overfit without evaluation set
- Strength
  - High tolerance to noisy data
  - Good generalization to untrained patterns
  - Well-suited for continuous-valued inputs *and outputs*
  - Successful on an array of real-world data, especially images, text, and time-series, e.g., one of early successful deep networks was applied to hand-written letters
  - Algorithms are inherently parallel
  - Builds more advanced representation
  - Techniques exist for the extraction of explanations from trained neural networks

Efficiency and interpretability

- Efficiency of backpropagation:
  Each epoch (one iteration through the training set) takes O(|D| * *w*), with |D| tuples and *w* weights, but # of epochs can be exponential to n, the number of inputs, in worst case
- For easier comprehension: Rule extraction by network pruning
  - Simplify the network structure by removing weighted links that have the least effect on the trained network
  - Then perform link, unit, or activation value clustering
  - The set of input and activation values are studied to derive rules describing the relationship between the input and hidden unit layers
- Sensitivity analysis: assess the impact that a given input variable has on a network output. The knowledge gained from this analysis can be represented in rules
- Recent attempts tend to learn interpretation along learning

Overfitting and model complexity

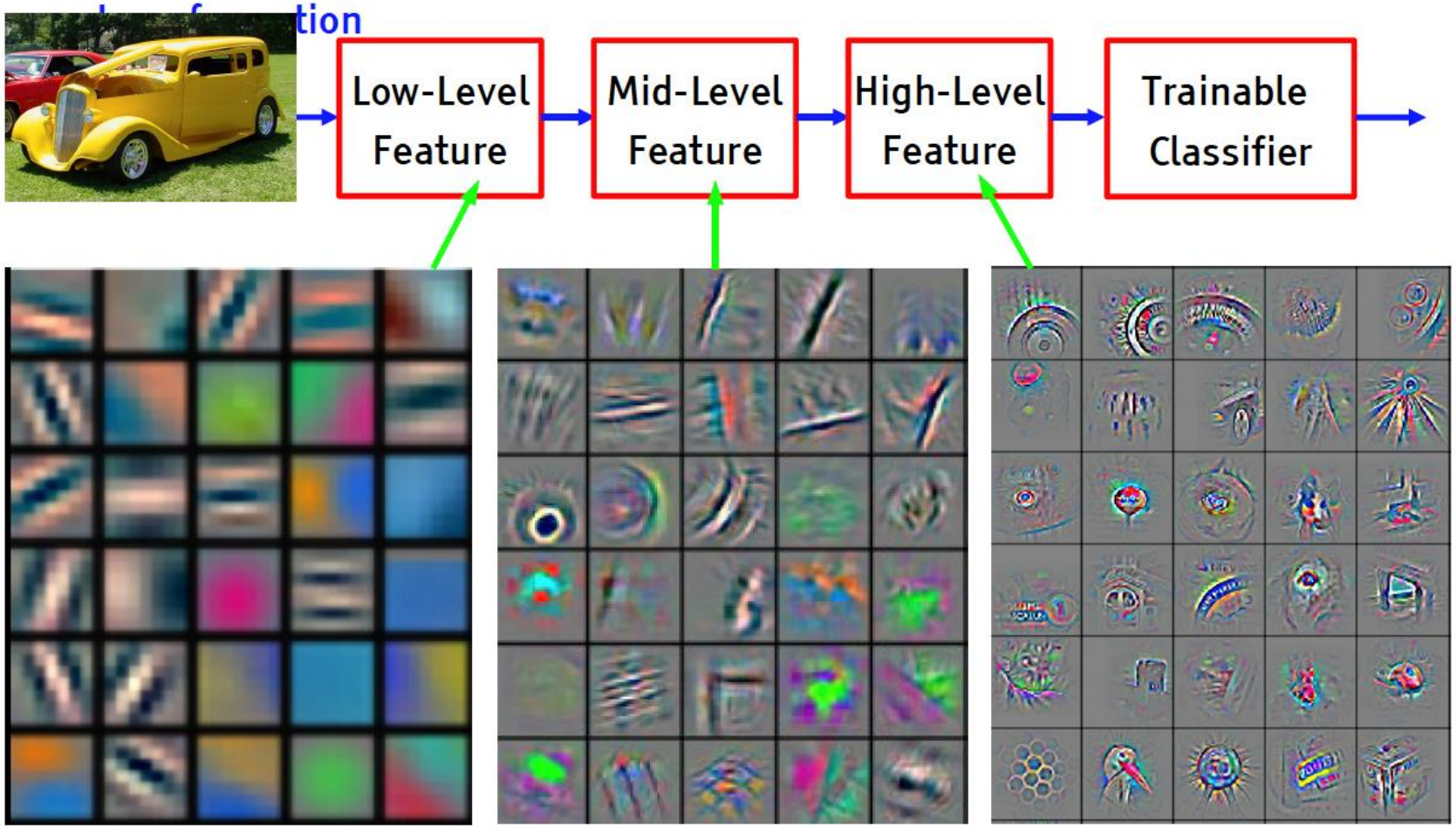
- which curve is more plausible given the data?
- overfitting
- neural nets are especially prone to overfitting
- why?



Prevention of overfitting

- Weight-decay
- Weight-sharing
- Early stopping
- Model averaging
- Bayesian fitting of neural nets
- Dropout
- Generative pre-training
- etc.

Deep learning = learning of hierarchical represenation



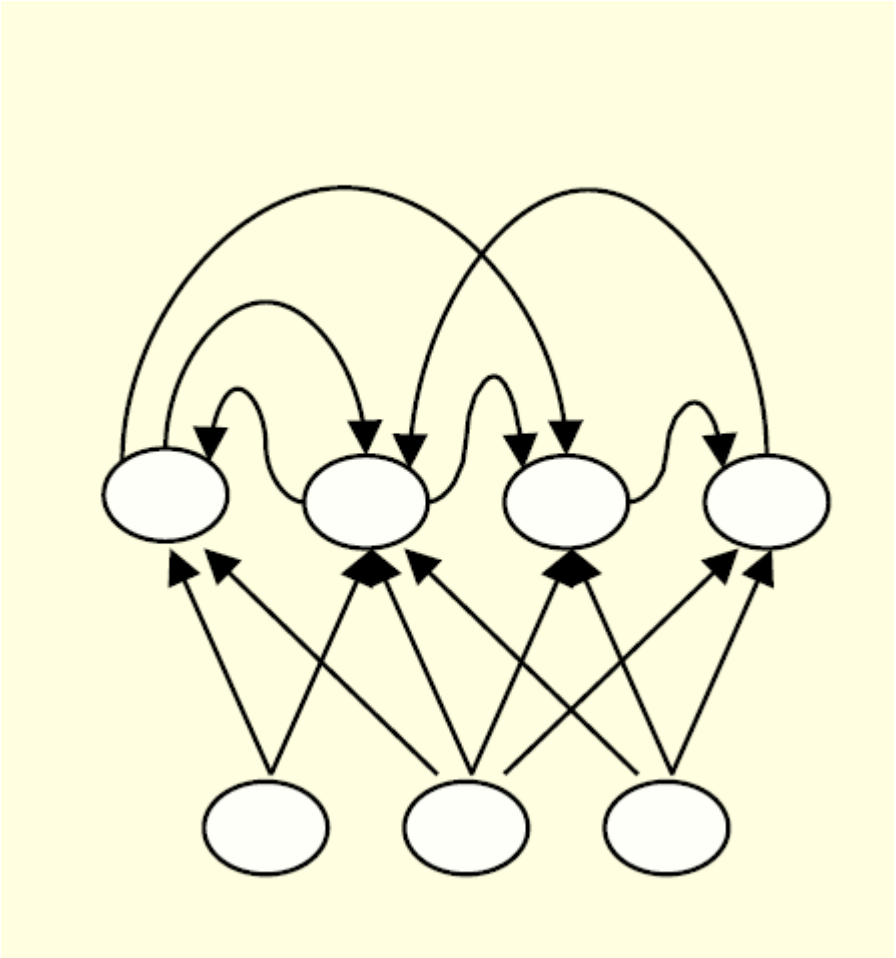
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Types of NN architectures

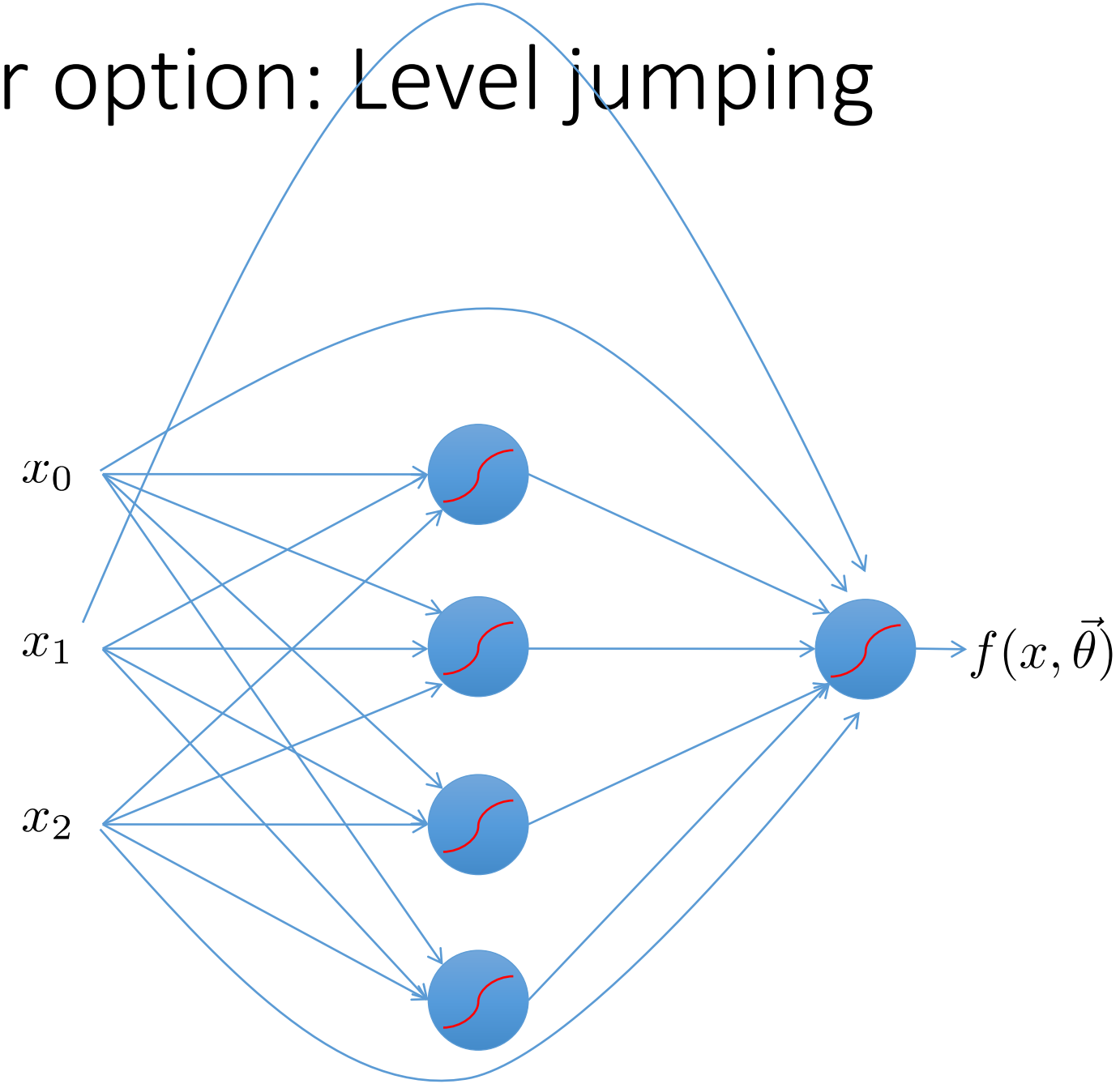
- Historically, feed-forward networks were the most commonly used; here neurons are activated progressively through layers from input to output
- However, we often combine different types of layers
- Examples of other architectures: recurrent, convolutional, transformer

Recurrent networks

- back connections
- biologically more realistic
- store information from the past
- more difficult to learn

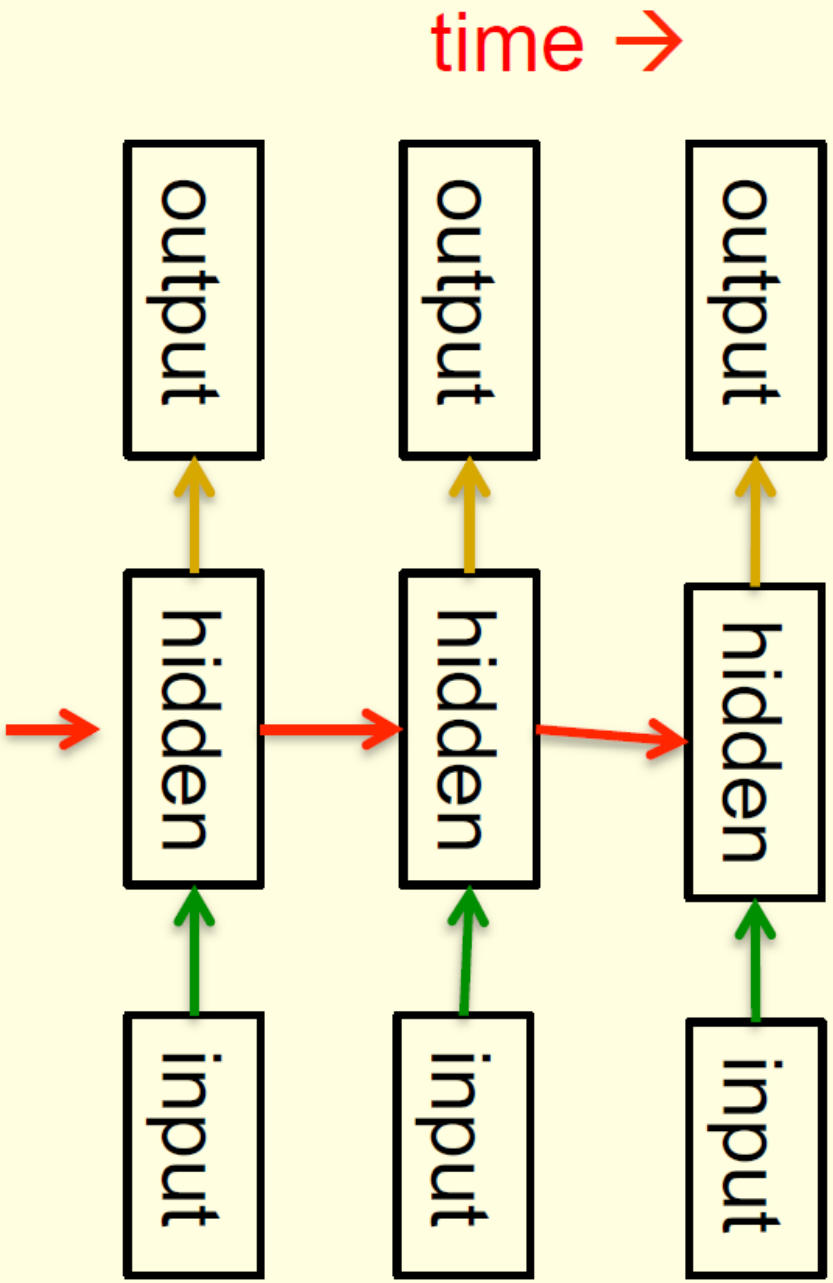


Another option: Level jumping

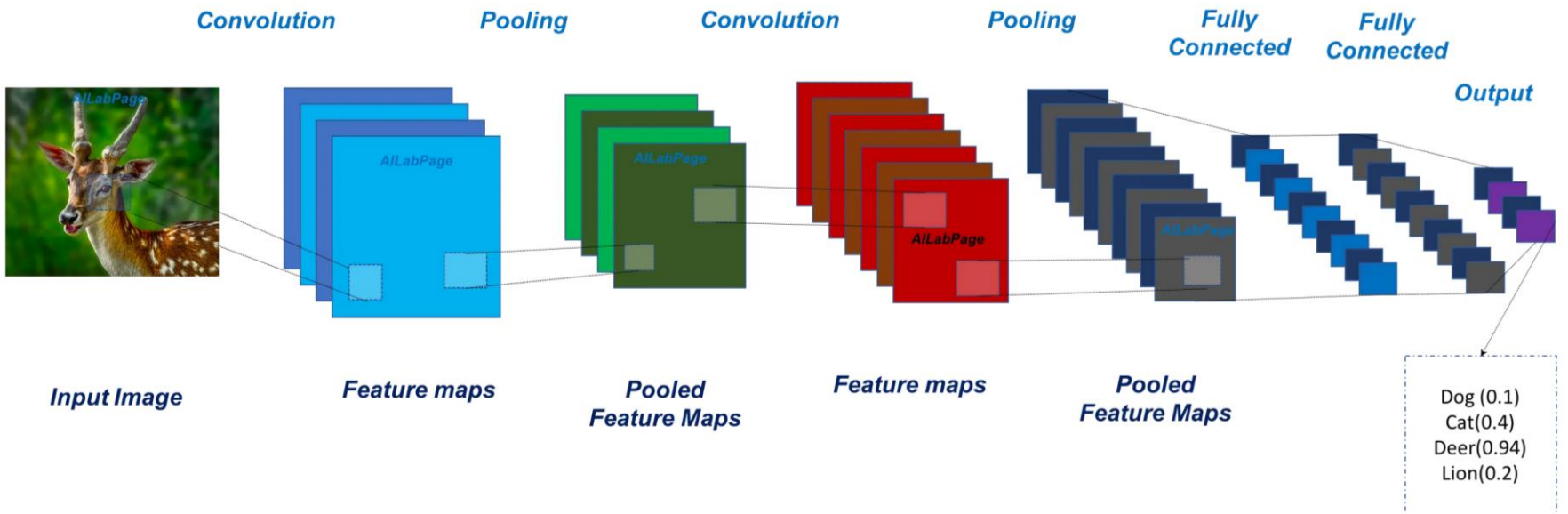


Recurrent networks for sequence learning

- unrolled network
- equivalent to deep networks with one hidden level per time slot
- but: hidden layers share weight (less parameters)



Convolutional neural networks (CNN)



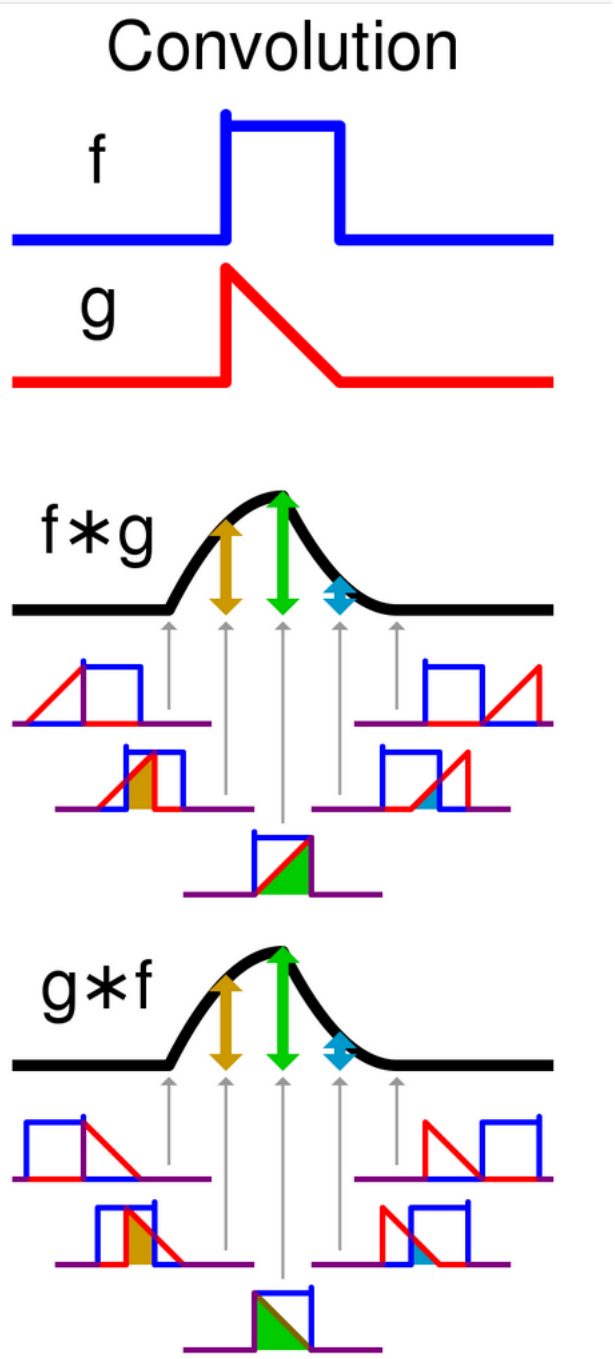
Convolution

- an operation on two functions (f and g) that produces a third function expressing how the shape of one is modified by the other.

$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau) g(t - \tau)\, d\tau.$$

- for discrete functions

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] g[n - m]$$

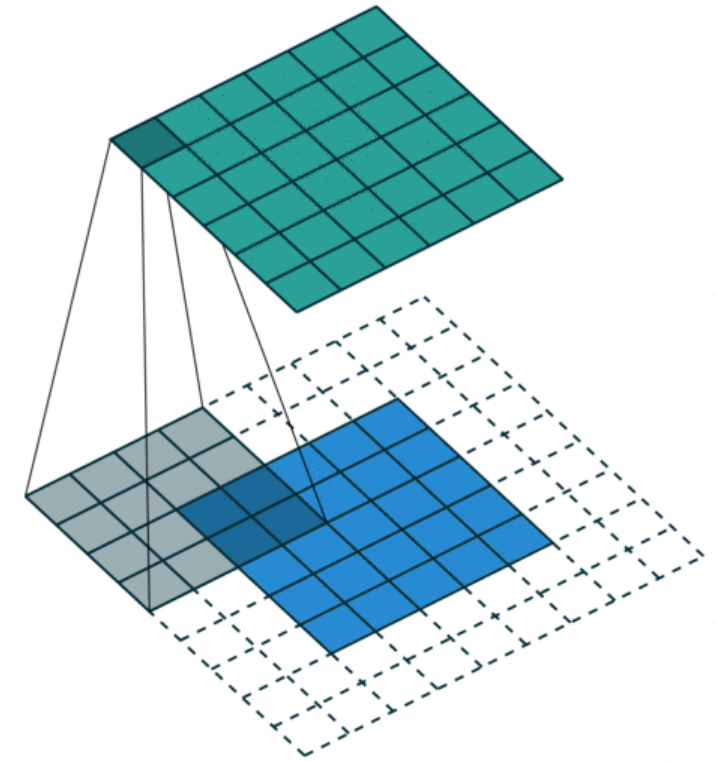


Convolutional Neural Network (CNN)

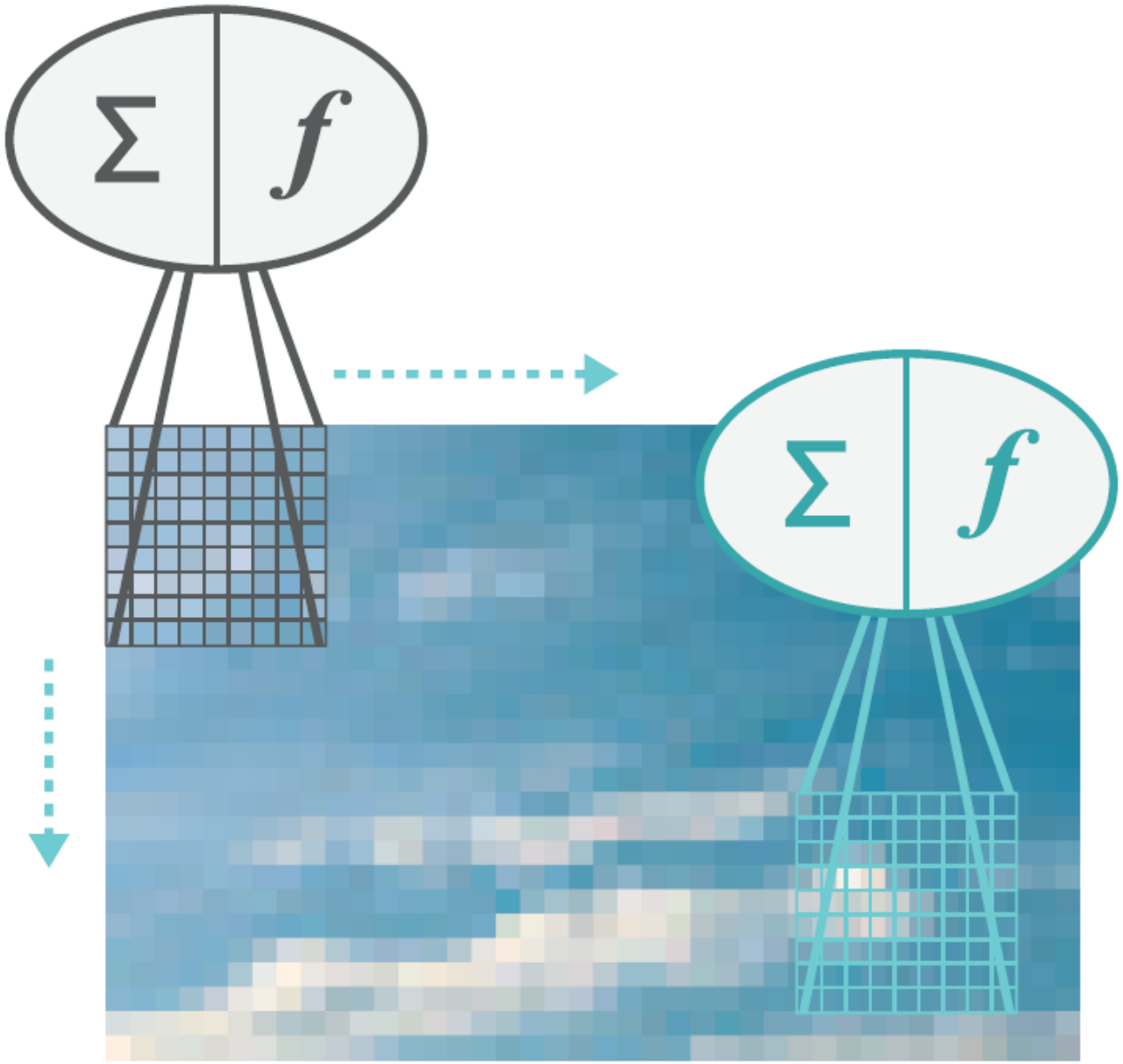
- Convolutional Neural Networks are inspired by mammalian visual cortex.
  - The visual cortex contains a complex arrangement of cells, which are sensitive to small sub-regions of the visual field, called a receptive field. These cells act as local filters over the input space and are well-suited to exploit the strong spatially local correlation present in natural images.
  - Two basic cell types:
    - Simple cells respond maximally to specific edge-like patterns within their receptive field.
    - Complex cells have larger receptive fields and are locally invariant to the exact position of the pattern.

Convolutional neural networks (CNN)

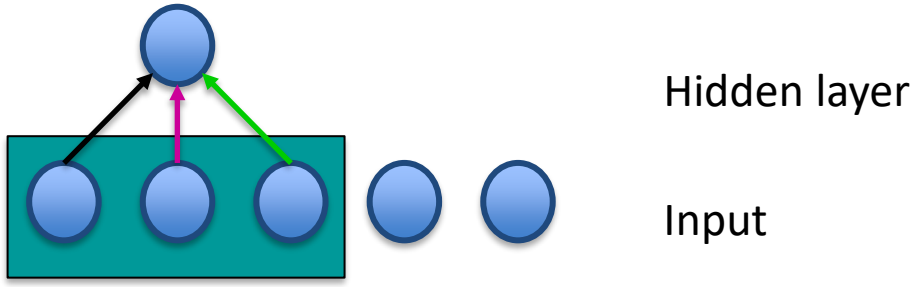
- currently, the most successful approach in image analysis, successful in language
- idea: many copies of small detectors used all over the image, recursively combined,
- detectors are learned, combination are learned



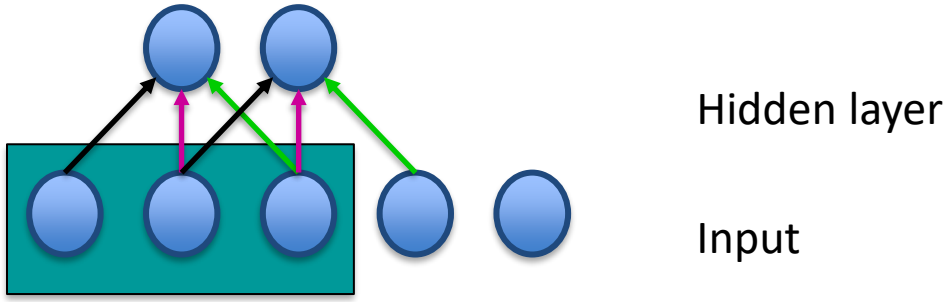
2d convolution for images



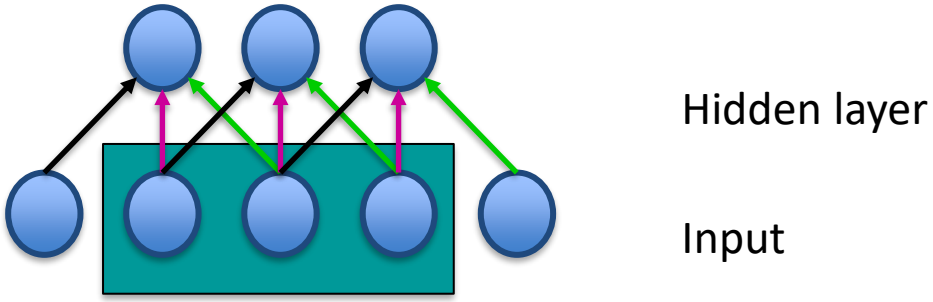
Basic Idea of CNNs



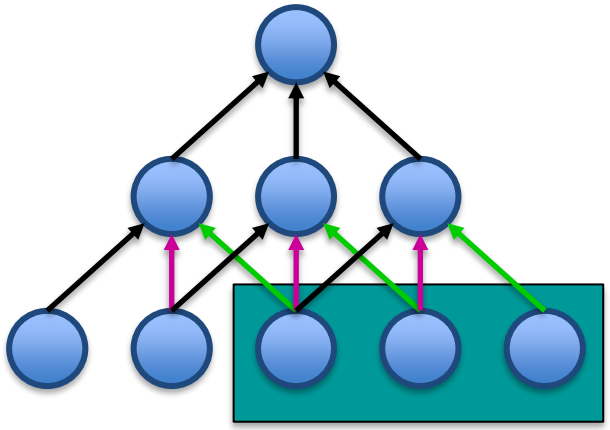
Basic Idea of CNNs



Basic Idea of CNNs



Basic Idea of CNNs

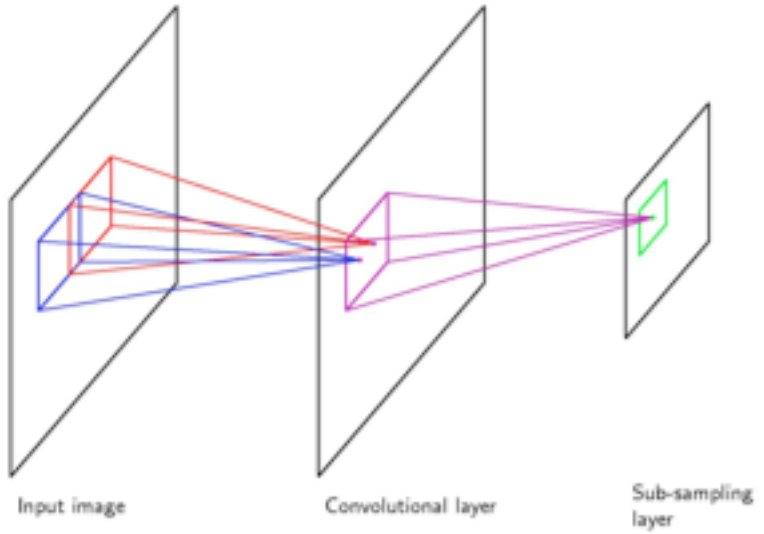


Hidden layer

Input

Convolutional Network

- The network is not fully connected.
- Different nodes are responsible for different regions of the image.
- This allows for robustness to transformations.
- Convolution is combined with subsampling.



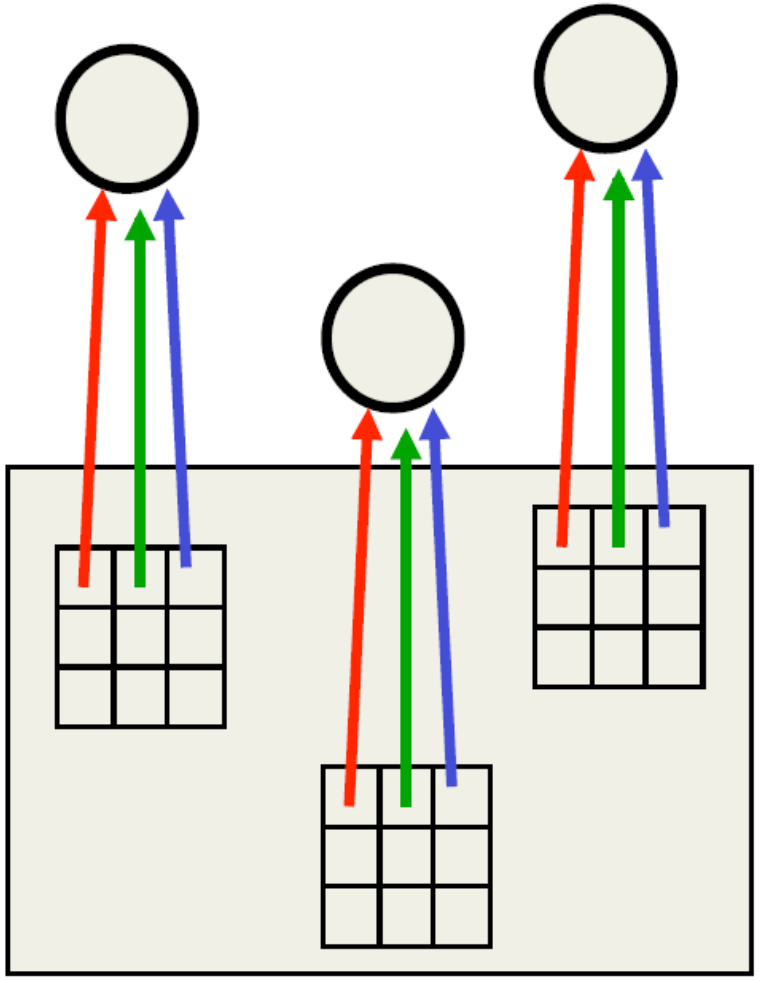
CNN Architecture: Convolutional Layer

- The core layer of CNNs
- The convolutional layer consists of a set of filters.
  - Each filter covers a spatially small portion of the input data.
- Each filter is convolved across the dimensions of the input data, producing a multidimensional feature map.
  - As we convolve the filter, we are computing the dot product between the parameters of the filter and the input.
- Intuition: the network will learn filters that activate when they see some specific type of feature at some spatial position in the input.
- The key architectural characteristics of the convolutional layer is local connectivity and shared weights.

Neural implementation of convolution

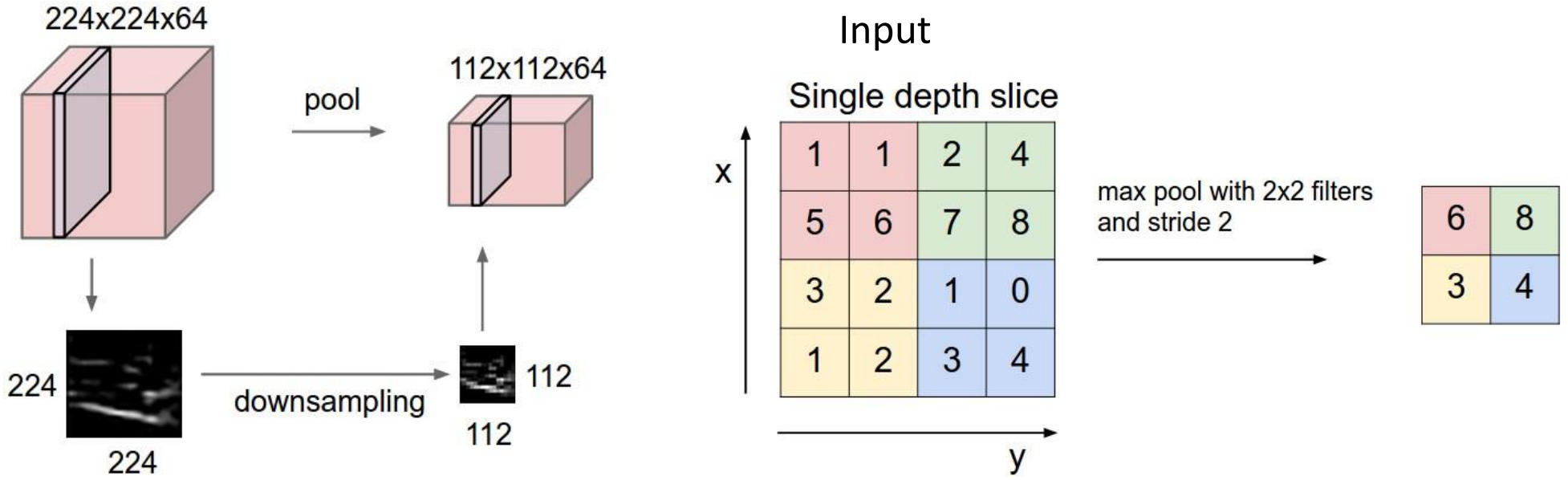
- weights of the same colors have equal weights
- adapted backpropagation

- images: 2d convolution
- languages: 1d convolution



CNN Architecture: Pooling Layer

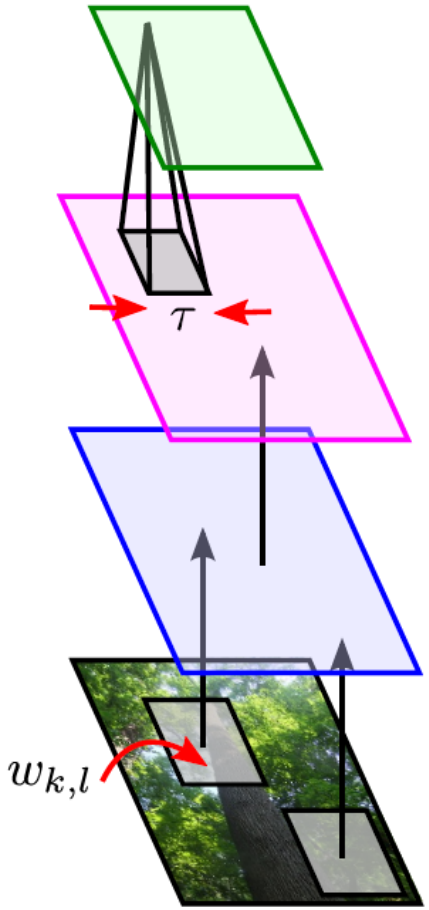
- Intuition: to progressively reduce the spatial size of the representation, to reduce the amount of parameters and computation in the network, and hence to also control overfitting
- Pooling partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum value of the features in that region.



CNN: pooling

- reduces the number of connections to the next layer (prevents the excessive number of parameters, speeds-up learning, reduces overfitting)
- max-pooling, min-pooling, average-pooling
- the problem: after several layers of pooling, we lose the information about the exact location of the recognized pattern and about spatial relations between different patterns and features, e.g., a nose on a forehead

Building-blocks for CNN's



$$x_{i,j} = \max_{|k|<\tau,|l|<\tau} y_{i-k,j-l}$$

mean or subsample also used

pooling stage

Feature maps of a larger region are combined.

$$y_{i,j} = f(a_{i,j})$$

e.g. $f(a) = [a]_+$

$f(a) = \text{sigmoid}(a)$

non-linear stage

Feature maps are trained with neurons.

Shared weights

$$a_{i,j} = \sum_{k,l} w_{k,l} z_{i-k,j-l}$$

only parameters

convolutional stage

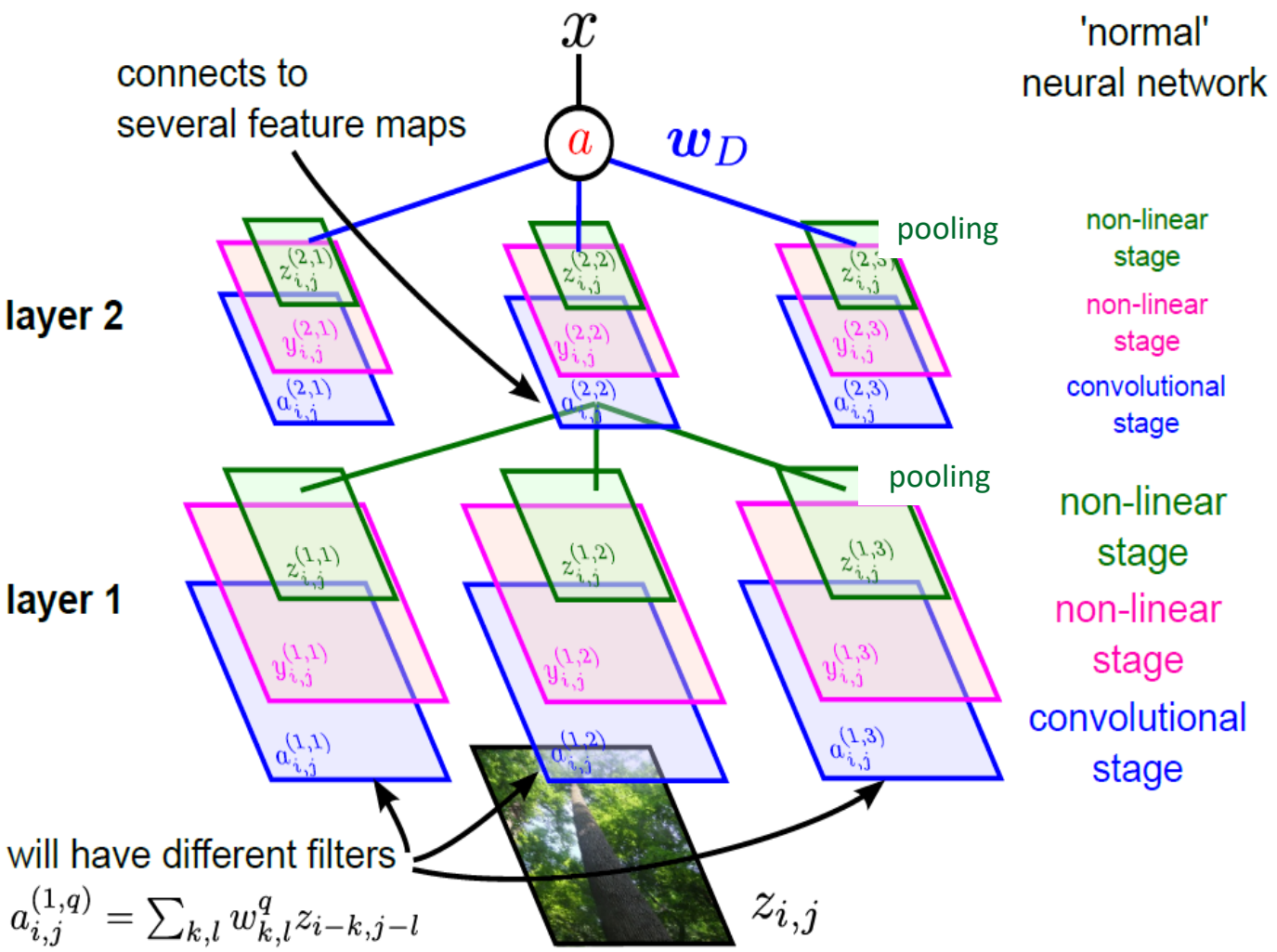
Each sub-region yields a feature map, representing its feature.

$z_{i,j}$

input image

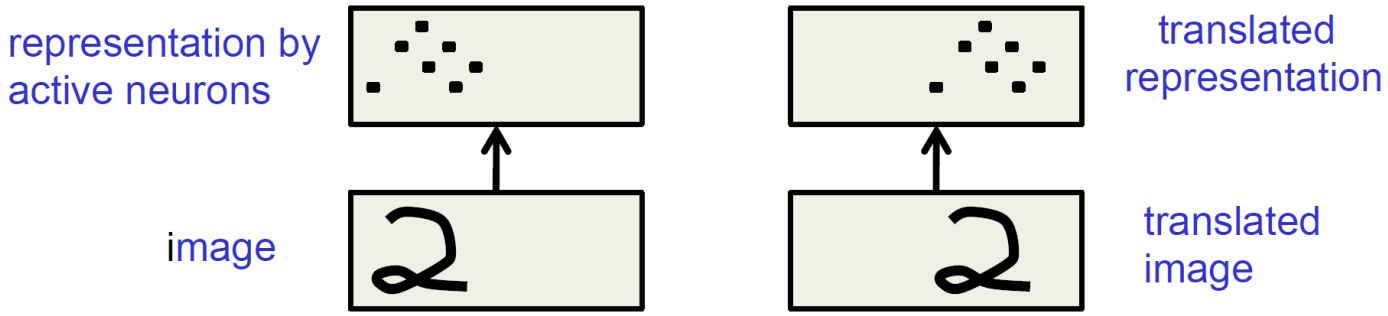
Images are segmented into sub-regions.

Full CNN


x
connects to
several feature maps
a
w_D
'normal'
neural network
pooling
non-linear
stage
non-linear
stage
convolutional
stage
layer 2
pooling
layer 1
non-linear
stage
non-linear
stage
convolutional
stage
will have different filters
$a_{i,j}^{(1,q)} = \sum_{k,l} w_{k,l}^{q} z_{i-k,j-l}$
$z_{i,j}$

Convolutional networks: illustration on image recognition

- a useful feature is learned and used on several positions
- prevents dimension hopping
- max-pooling

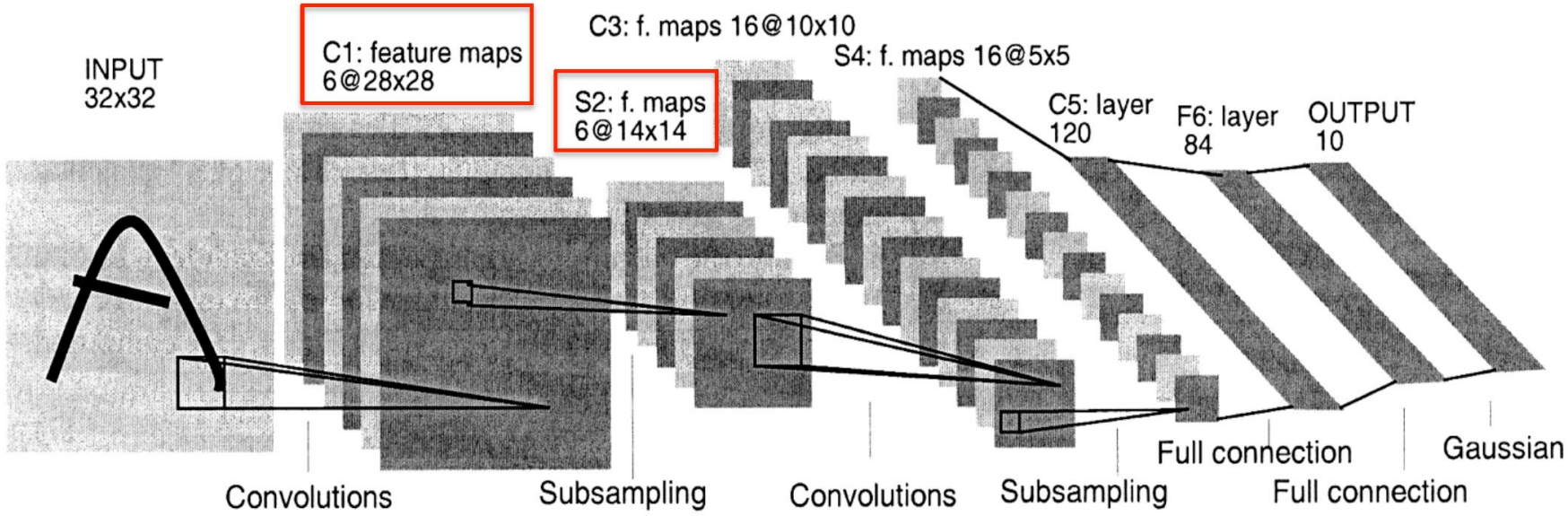


CNN success: LeNet

- handwritten digit recognition by Yann LeCun,
- several hidden layers
- several convolutional filters
- pooling
- several other tricks

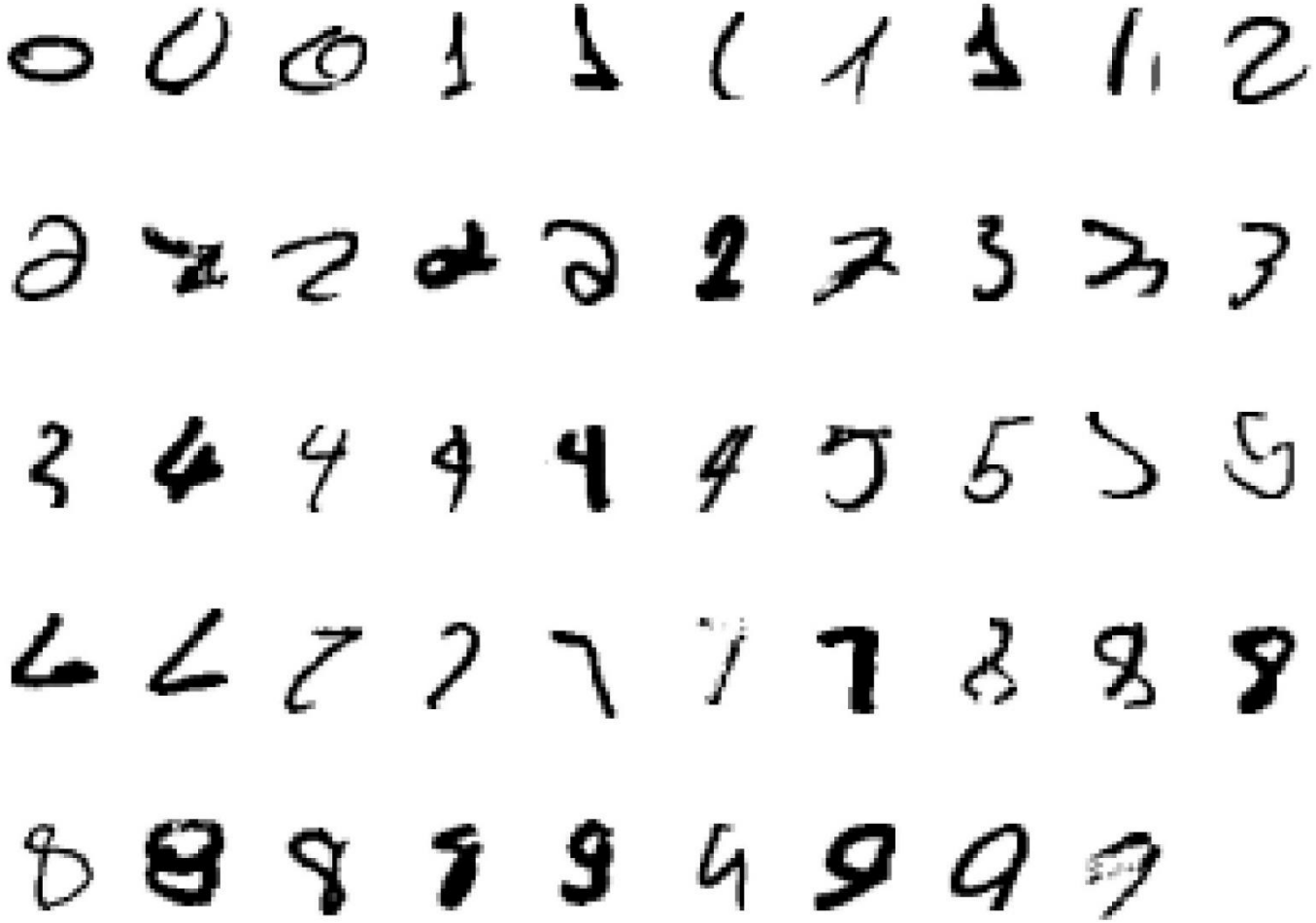
LeNet5 architecture

- handwritten digit recognition

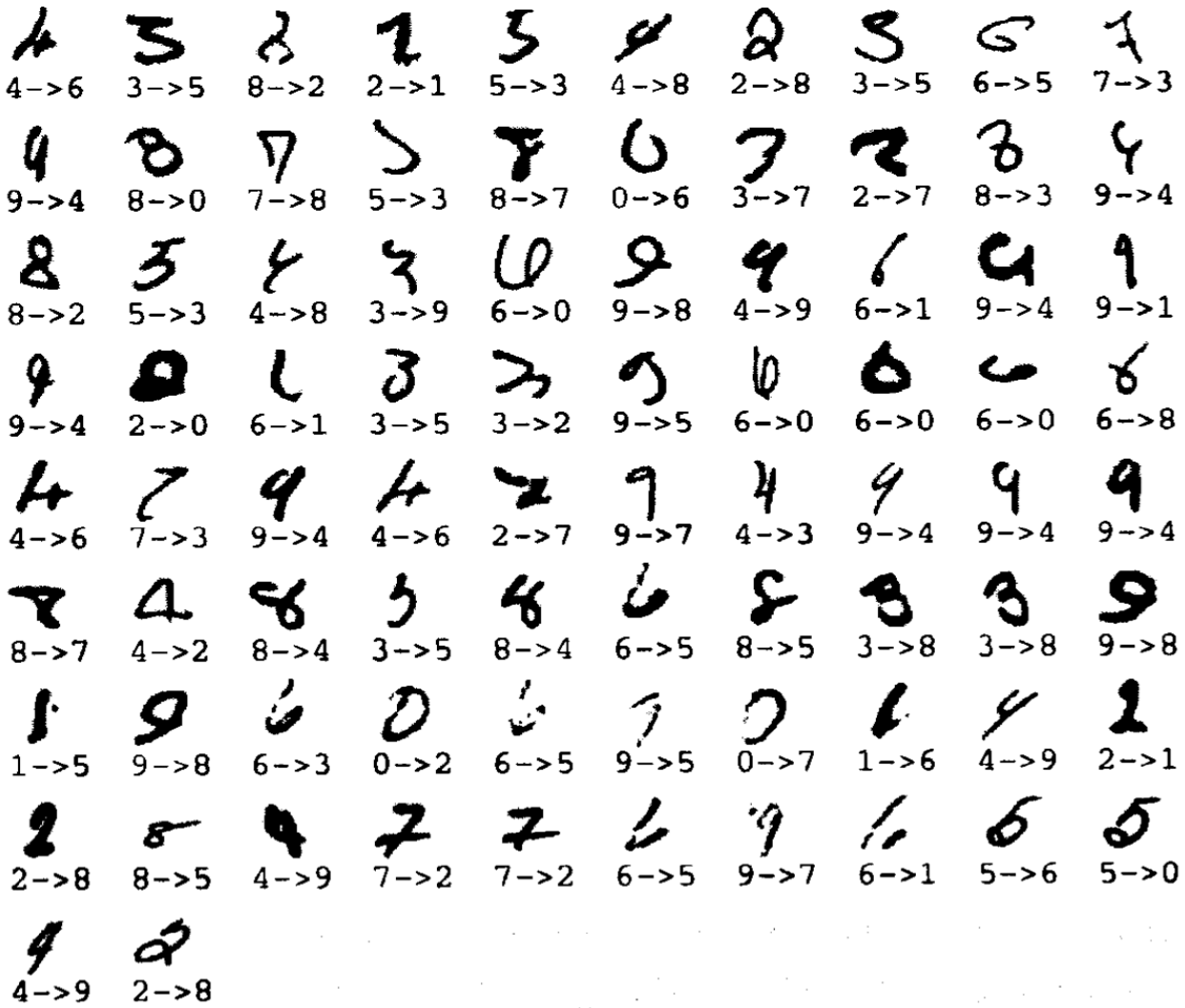


Hand-written Digit Recognition

- Input:



Errors of LeNet5

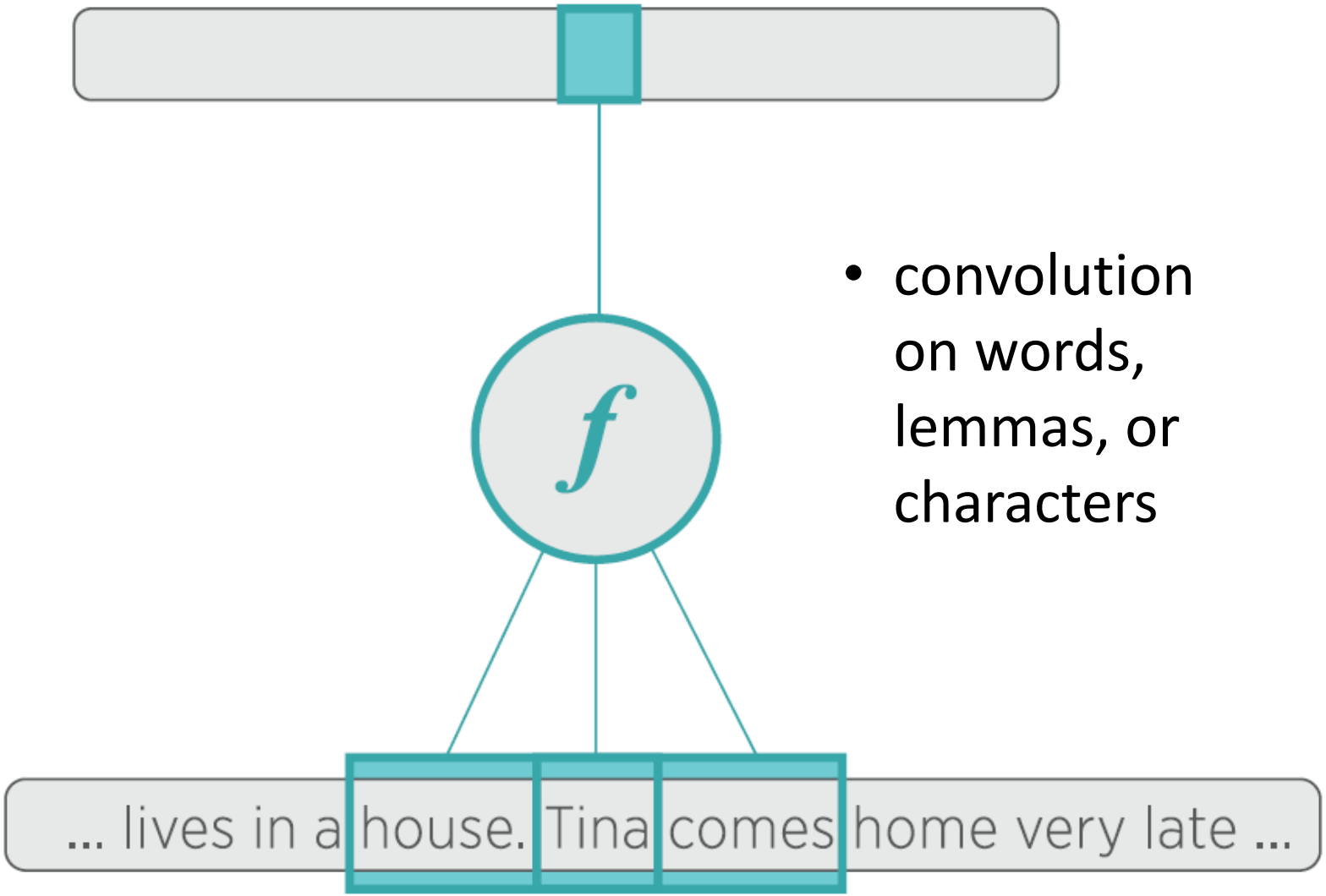


- 80 errors in 10,000 test cases

Benefits of CNNs

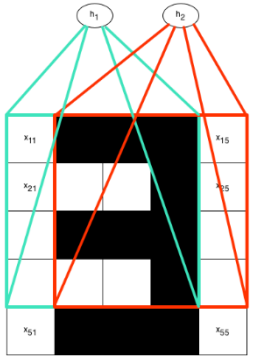
- The number of weights can be much less than 1 million for a 1 mega pixel image.
- The small number of weights can use different parts of the image as training data. Thus we have several orders of magnitude more data to train the fewer number of weights.
- We get translation invariance for free.
- Fewer parameters take less memory and thus all the computations can be carried out in memory in a GPU or across multiple processors.

1d convolution for text

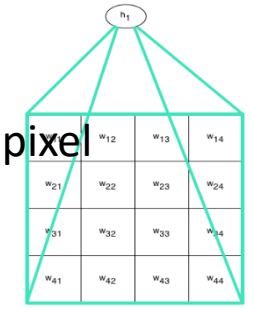


- convolution on words, lemmas, or characters

Example: what the following CNN returns 1/2



We have a convolutional neural network for images of 5 by 5 pixels.
In this network, each hidden unit is connected to a different 4 x 4 region of the input image:
The first hidden unit, h1, is connected to the upper left 4x4 portion of the input image (as shown).
The second hidden unit, h2, is connected to the upper right 4x4 portion of the input image (as shown).
The third hidden unit, h3, is connected to the lower left 4x4 portion of the input image (not shown in the diagram).
The fourth hidden unit, h4, is connected to the lower right 4x4 portion of the input image (not shown in the diagram).
Because it's a convolutional network, the weights (connection strengths) are the same for all hidden units: the only difference between the hidden units is that each of them connects to a different part of the input image.



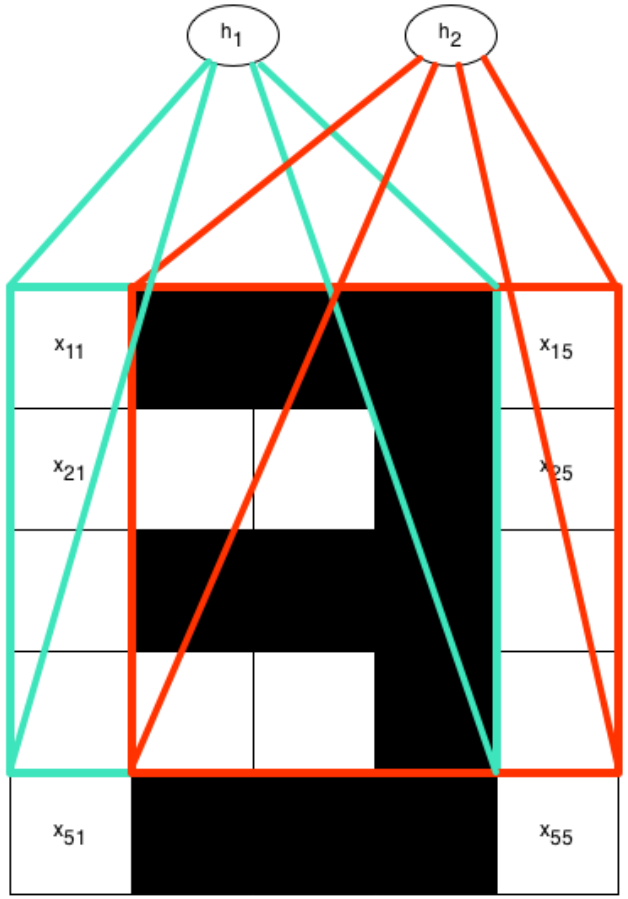
In the second diagram, we show the array of weights, which are the same for each of the four hidden units.
For h1, weight w11 is connected to the top-left pixel, i.e. x11, while for hidden unit h2, weight w11 connects to the pixel that is one to the right of the top left pixel, i.e. x12.

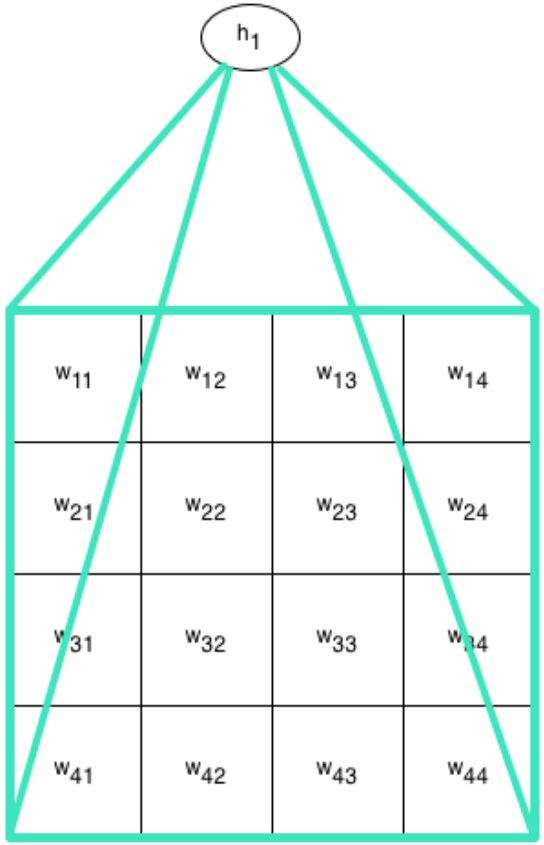
Imagine that for some training case, we have an input image where each of the black pixels in the top diagram has value 1, and each of the white ones has value 0. Notice that the image shows a "3" in pixels.

The network has no biases. The weights of the network are given as follows:
w11=1w12=1w13=1w14=0w21=0w22=0w23=1w24=0w31=1w32=1w33=1w34=0w41=0w42=0w43=1w44=0
The hidden units are *linear*.

For the training case with that "3" input image, what is the output y1, y2, y3, y4 of each of the four hidden units?

Example: what the following CNN returns 2/2

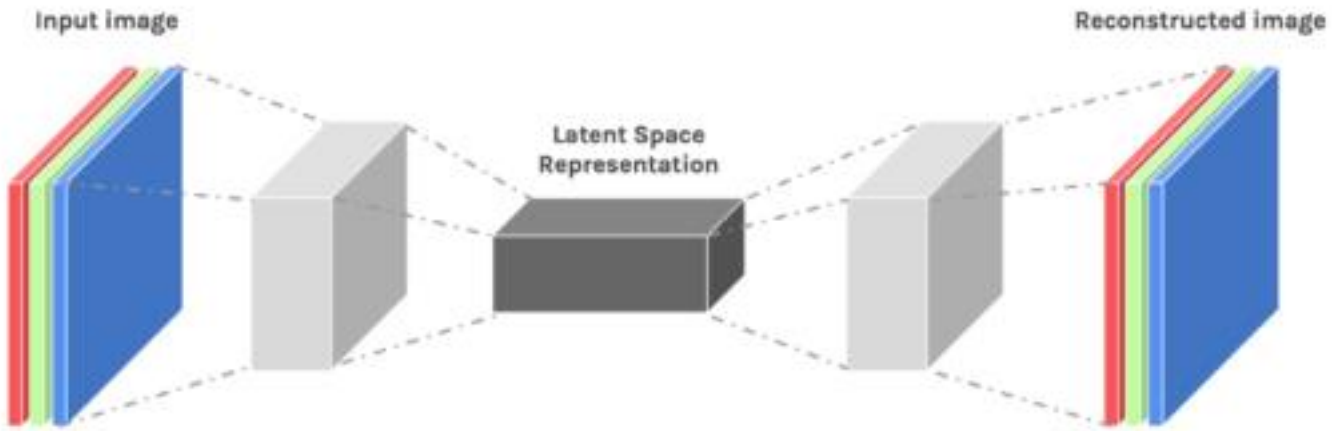




$$w_{11} = 1 \quad w_{12} = 1 \quad w_{13} = 1 \quad w_{14} = 0$$
$$w_{21} = 0 \quad w_{22} = 0 \quad w_{23} = 1 \quad w_{24} = 0$$
$$w_{31} = 1 \quad w_{32} = 1 \quad w_{33} = 1 \quad w_{34} = 0$$
$$w_{41} = 0 \quad w_{42} = 0 \quad w_{43} = 1 \quad w_{44} = 0$$

Autoencoders

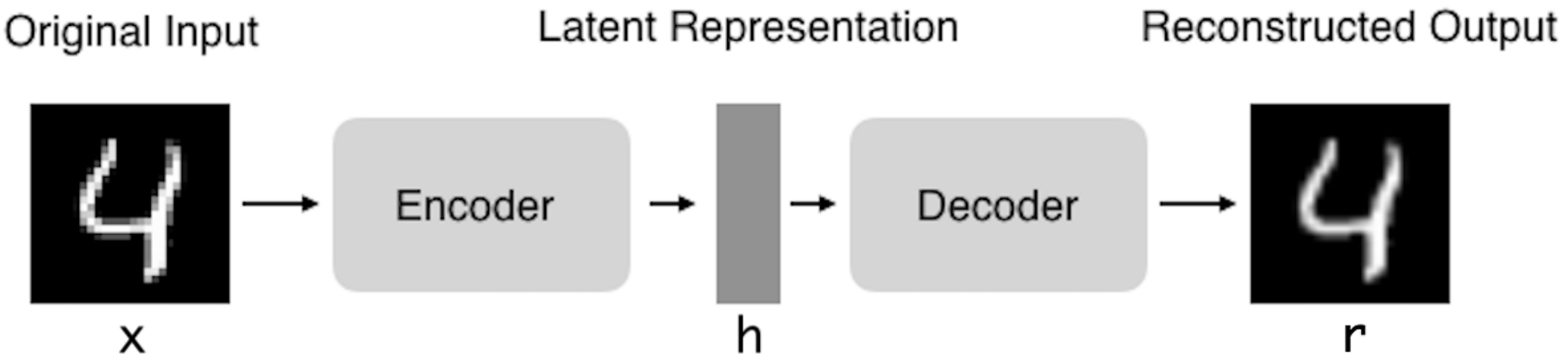
- Autoencoders are designed to reproduce their input, especially for images.
- The key point is to reproduce the input from a learned encoding.
- The loss function is the reproduction error



https://www.edureka.co/blog/autoencoders-tutorial/

Autoencoders: structure

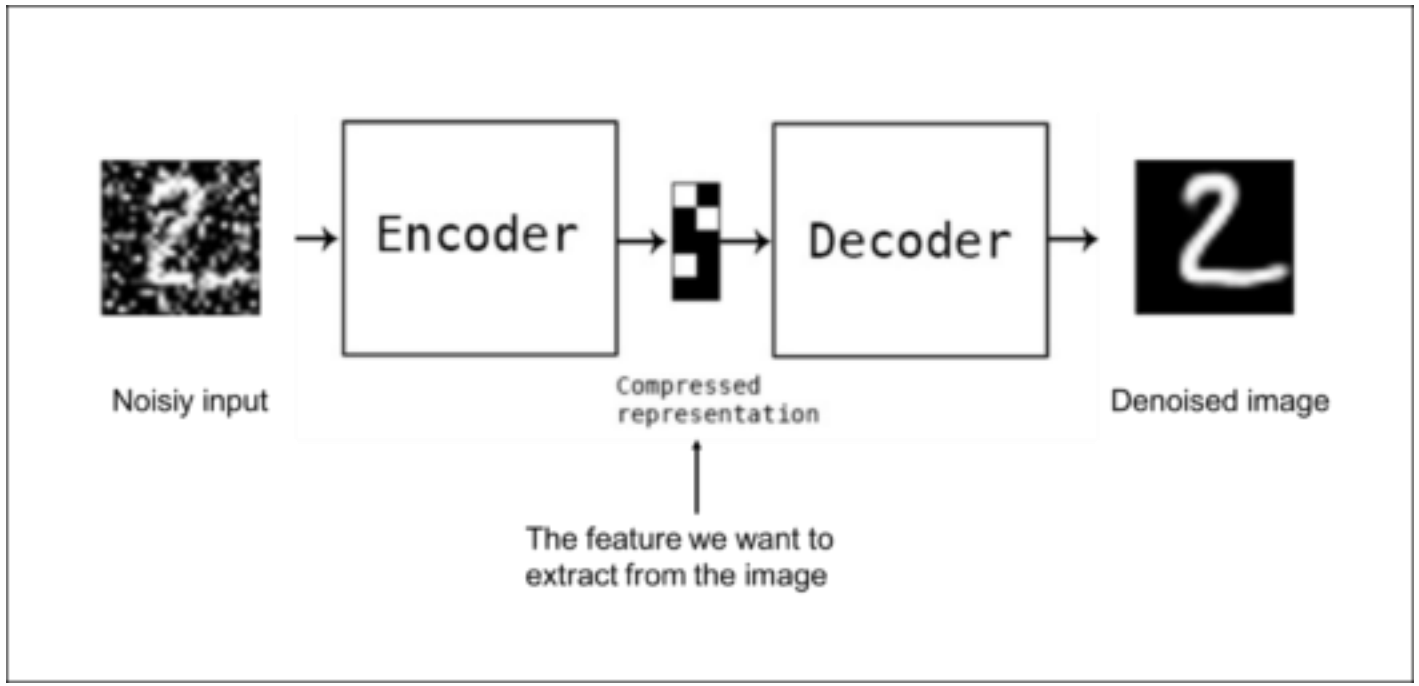
- Encoder: compress input into a latent-space of usually smaller dimension. h = f(x)
- Decoder: reconstruct input from the latent space. r = g(f(x)) with r as close to x as possible



https://towardsdatascience.com/deep-inside-autoencoders-7e41f319999f

Autoencoder applications: denoising

- Denoising: input clean image + noise and train to reproduce the clean image.



https://www.edureka.co/blog/autoencoders-tutorial/

Denoising autoencoders

- Basic autoencoder trains to minimize the loss between x and the reconstruction g(f(x)).
- Denoising autoencoders train to minimize the loss between x and g(f(x+w)), where w is random noise.
- Same possible architectures, different training data.

https://blog.keras.io/building-autoencoders-in-keras.html

Autoencoder applications: colorization

- Image colorization: input black and white and train to produce color images

https://www.edureka.co/blog/autoencoders-tutorial/

Autoencoder applications: watermark removal

- Watermark removal

https://www.edureka.co/blog/autoencoders-tutorial/

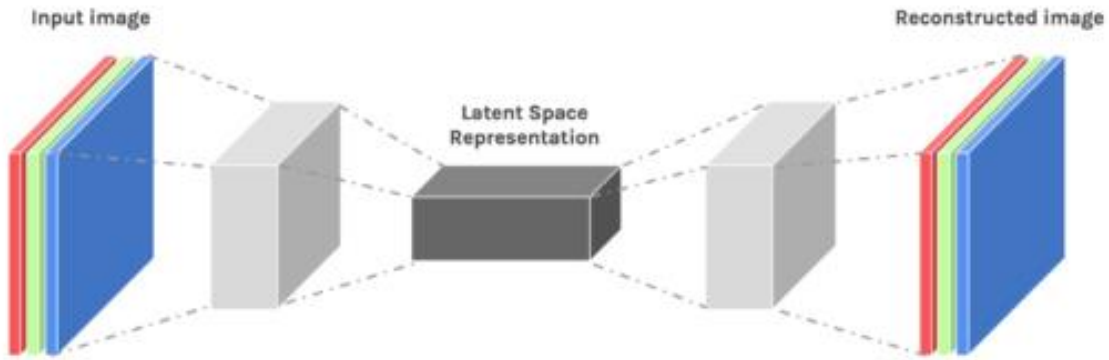
Properties of autoencoders

- **Data-specific**: Autoencoders are only able to compress data similar to what they have been trained on.
- **Lossy:** The decompressed outputs will be degraded compared to the original inputs.
- **Learned automatically from examples:** It is easy to train specialized instances of the algorithm that will perform well on a specific type of input.

https://www.edureka.co/blog/autoencoders-tutorial/

Bottleneck layer (undercomplete)

- Suppose input images are n x n and the latent space is $m < n \times n$.
- Then the latent space is not sufficient to reproduce all images.
- Needs to learn an encoding that captures the important features in training data, sufficient for approximate reconstruction.



GANs

- **Generative**
- Learn a generative model

- **Adversarial**
- Trained in an adversarial setting

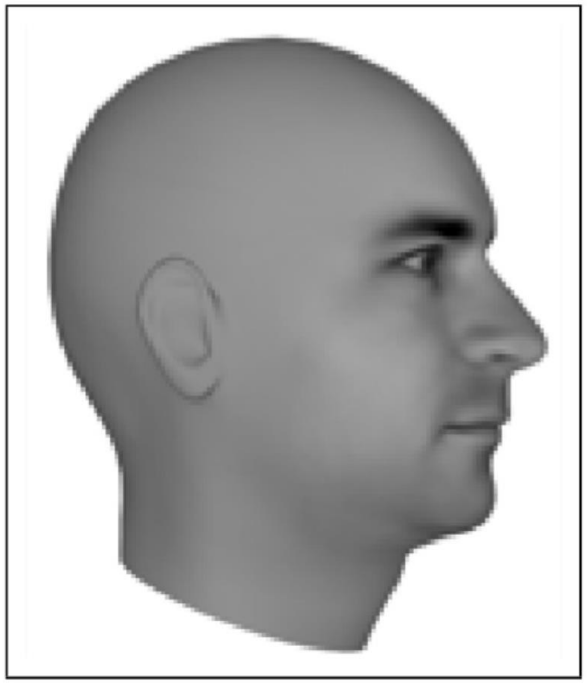
- **Networks**
- Use Deep Neural Networks

Why Generative Models?

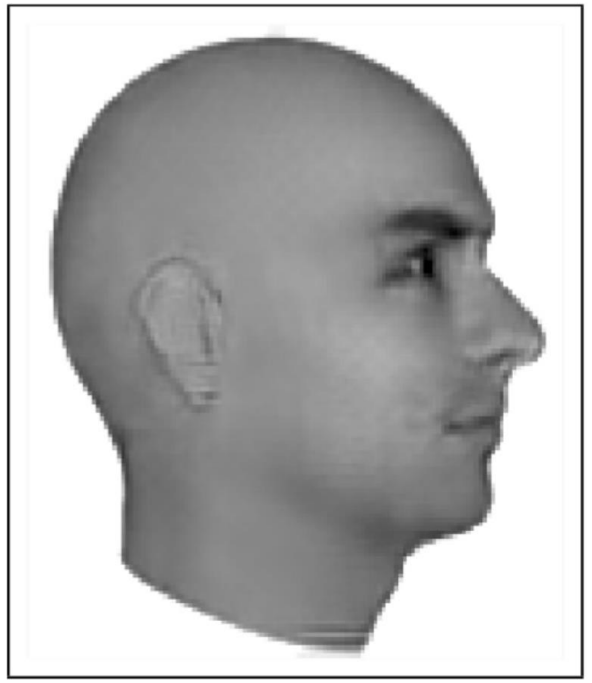
- We have only seen discriminative models so far
- Given an image X, predict a label Y
- Estimates P(Y|X)
- Discriminative models have several key limitations
- Cannot model P(X), i.e. the probability of seeing a certain image
- Thus, can't sample from P(X), i.e. can't generate new images
- Generative models (in general) cope with all of above
- Can model P(X)
- Can generate new images

What GANs can do

Ground Truth



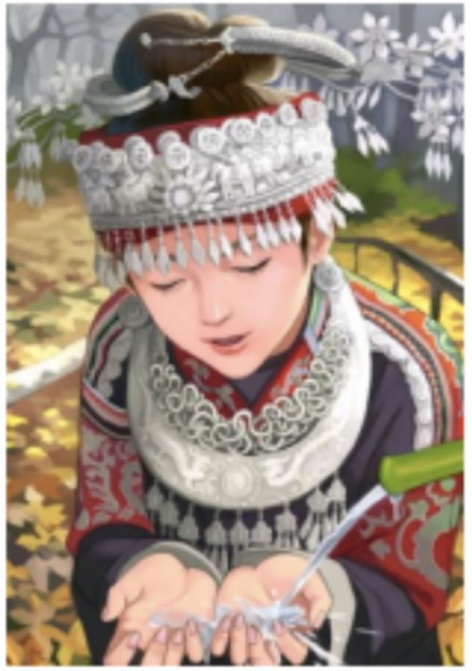
Adversarial



Lotter, William, Gabriel Kreiman, and David Cox. "Unsupervised learning of visual structure using predictive generative networks." *arXiv preprint arXiv:1511.06380* (2015).

GANs in action

Which one is Computer generated?

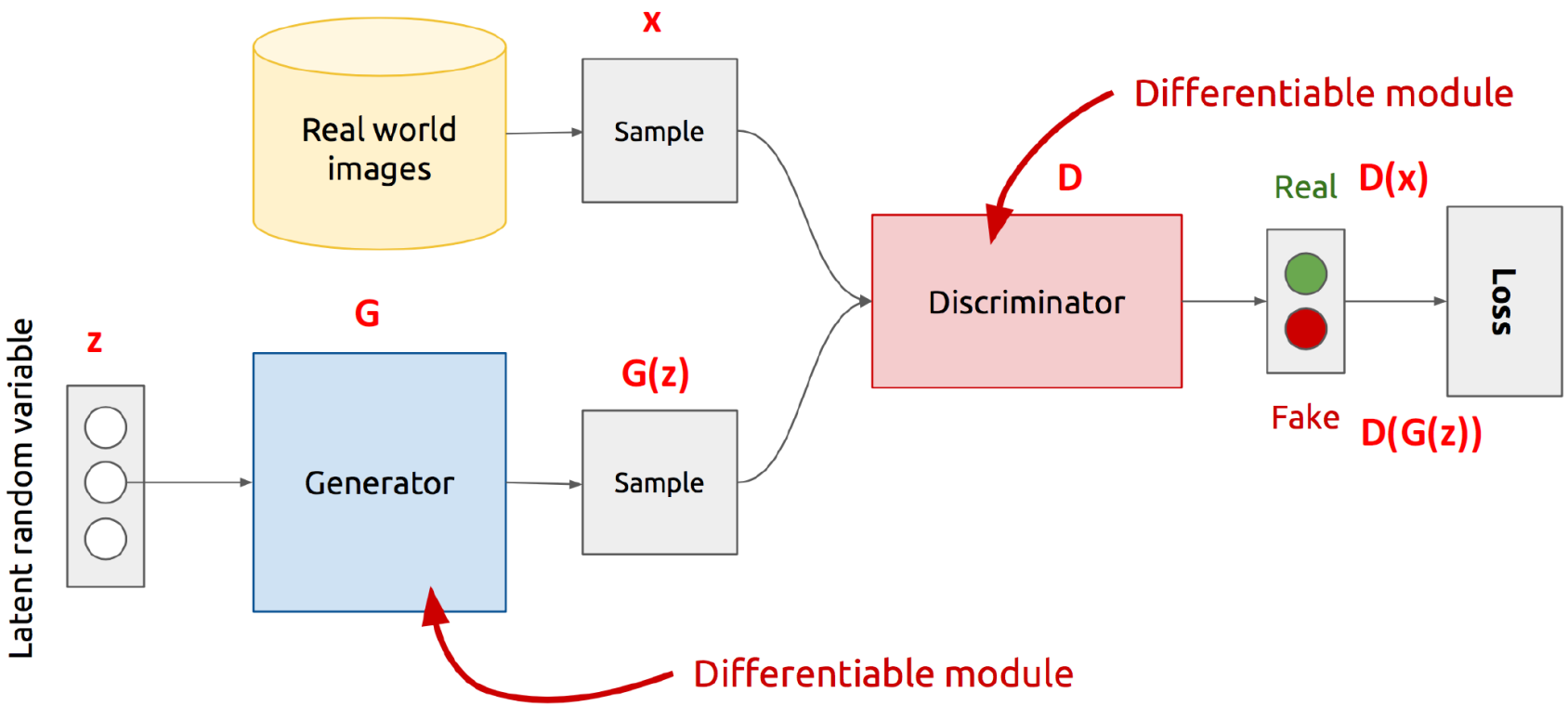


Ledig, Christian, et al. "Photo-realistic single image super-resolution using a generative adversarial network." *arXiv preprint arXiv:1609.04802* (2016).

Adversarial Training

- Generator: generate fake samples, tries to fool the Discriminator
- Discriminator: tries to distinguish between real and fake samples
- Train them against each other
- Repeat this and we get better Generator and Discriminator

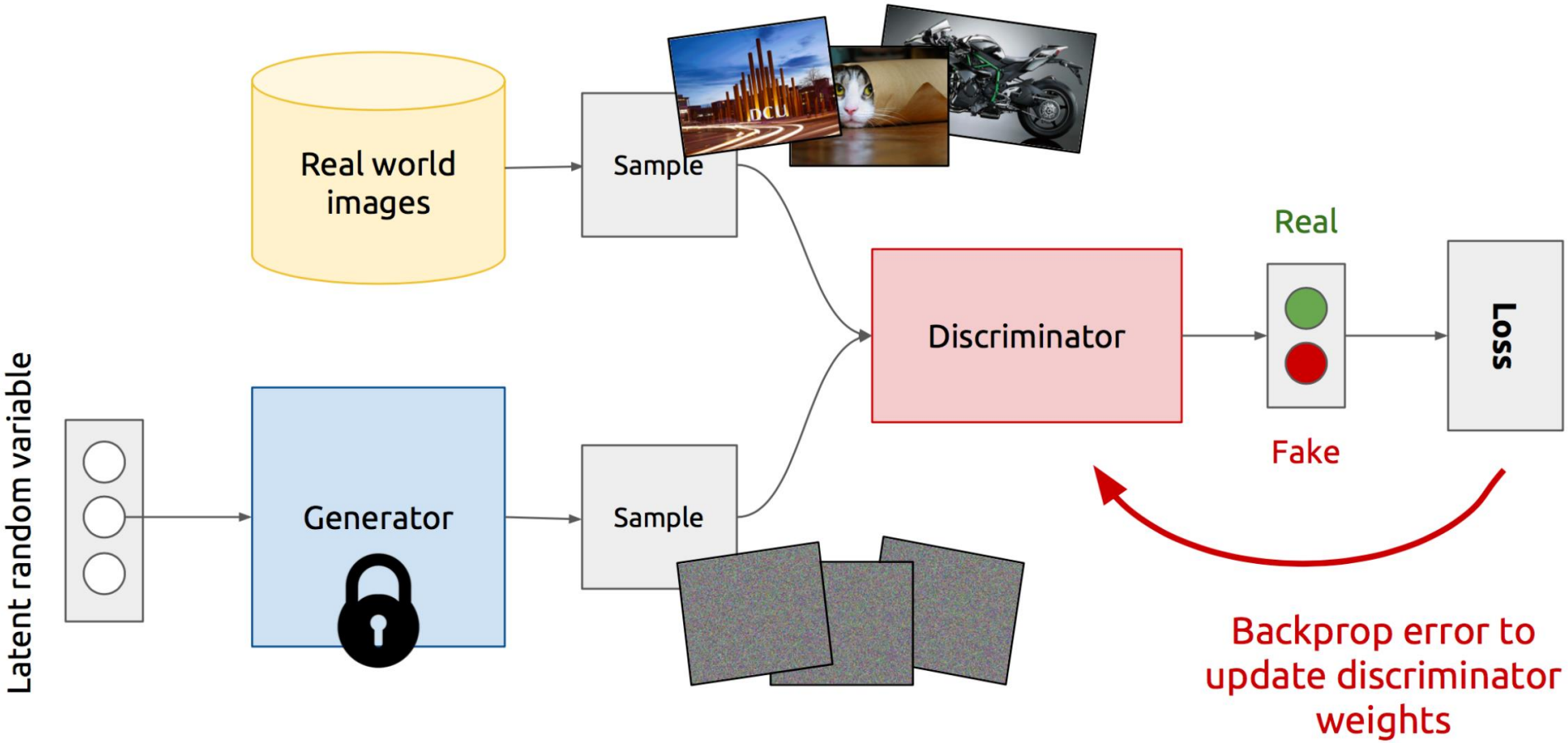
GAN's Architecture



- **Z** is some random noise (Gaussian/Uniform).
- **Z** can be thought as the latent representation of the image.

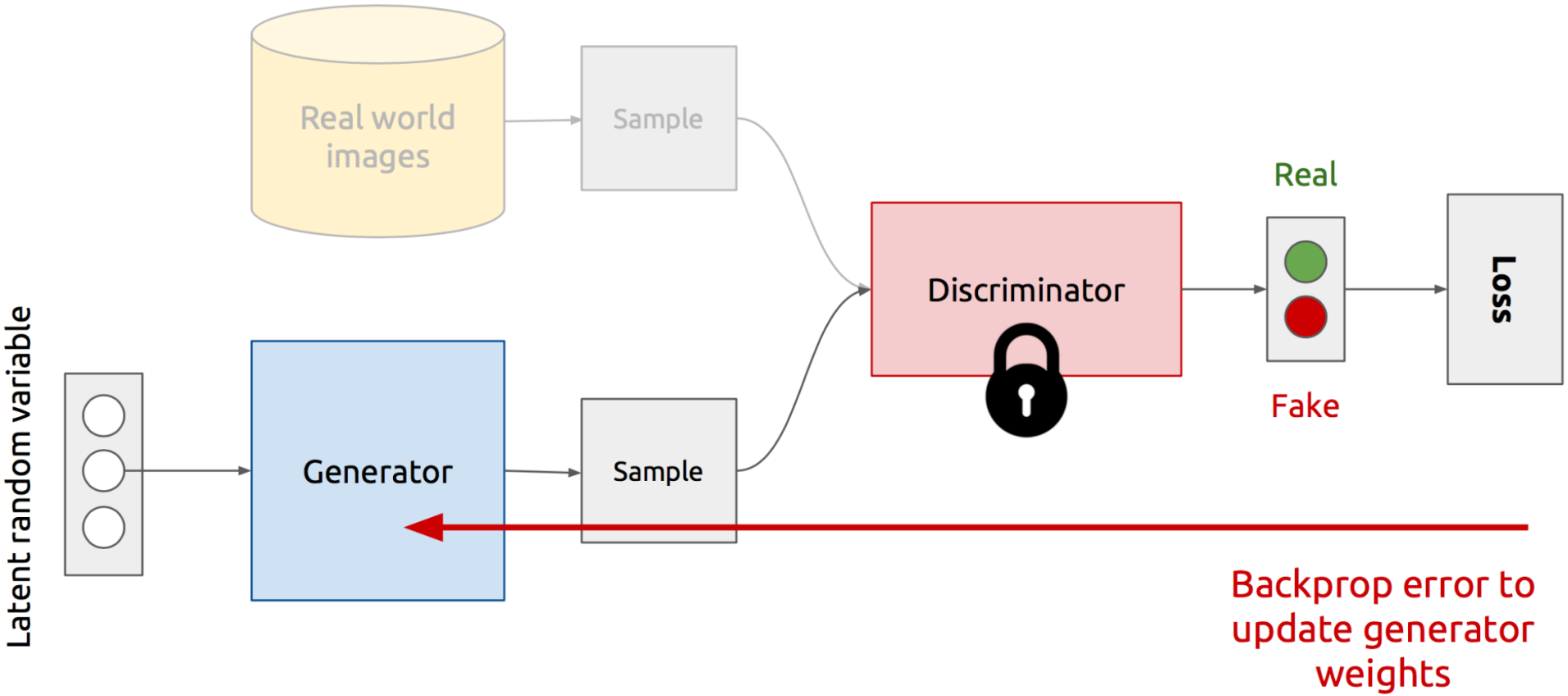
https://www.slideshare.net/xavigiro/deep-learning-for-computer-vision-generative-models-and-adversarial-training-upc-2016

Training Discriminator



https://www.slideshare.net/xavigiro/deep-learning-for-computer-vision-generative-models-and-adversarial-training-upc-2016

Training Generator



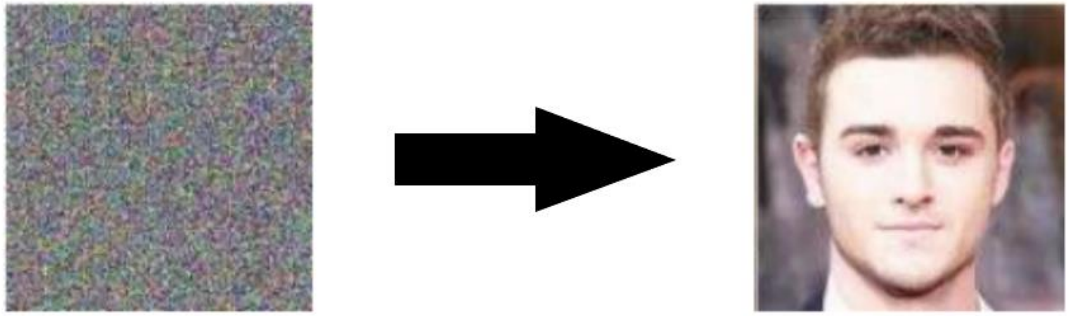
https://www.slideshare.net/xavigiro/deep-learning-for-computer-vision-generative-models-and-adversarial-training-upc-2016

Diffusion models intro

- Recent superior image generators,
- E.g., DALL-E is prompt based
- "a bowl of soup that is a portal to another dimension as digital art".

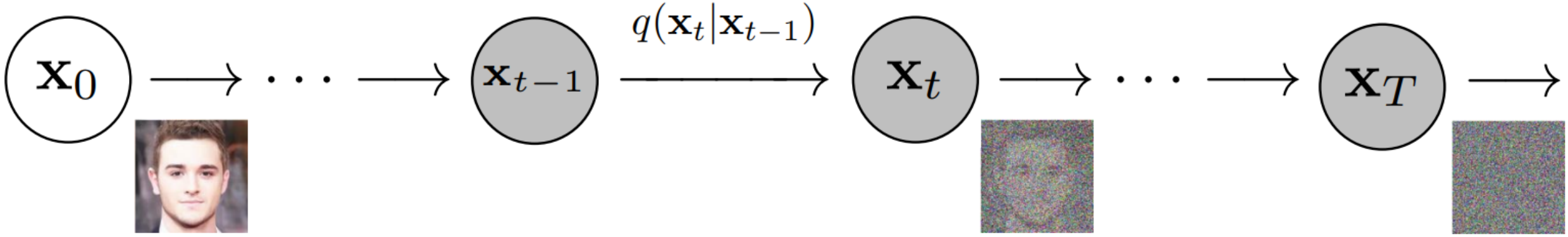
Idea of diffusion models

- generate data similar to the data on which they are trained
- destroy training data through the successive addition of Gaussian noise
- then learning to recover the data by *reversing* this noising process.
- After training, generate data by passing randomly sampled noise through the learned denoising process.



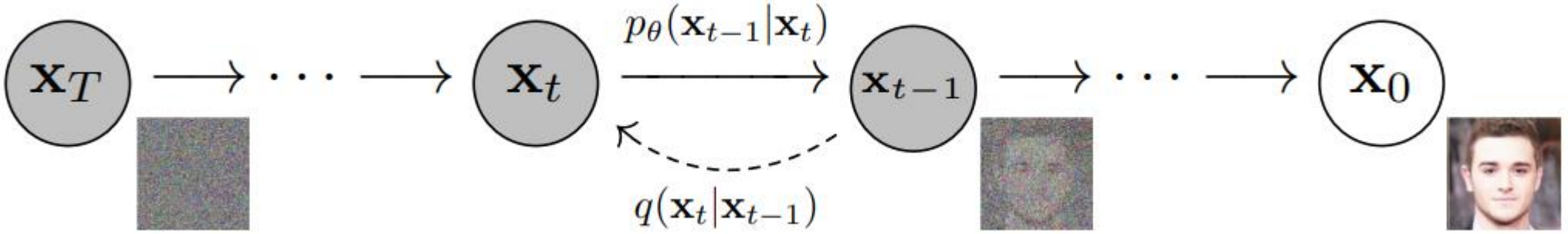
Diffusion models 1/2

- A diffusion model maps to the latent space using a fixed Markov chain. This chain gradually adds noise to the data in order to obtain the approximate posterior.



Diffusion models 2/2

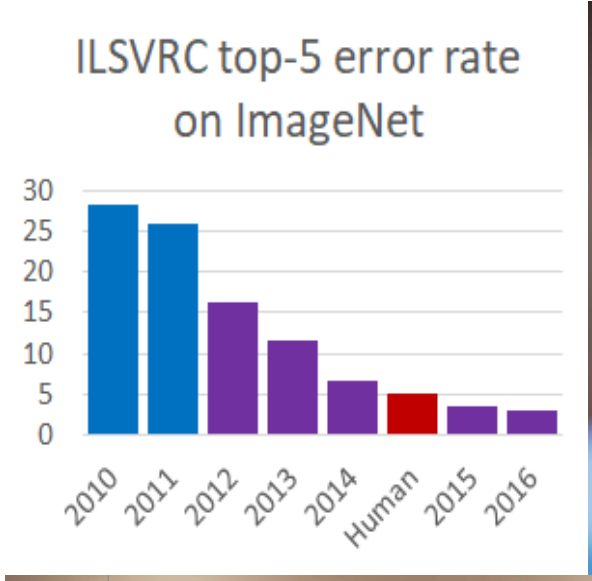
- A diffusion model is trained to **reverse** the process

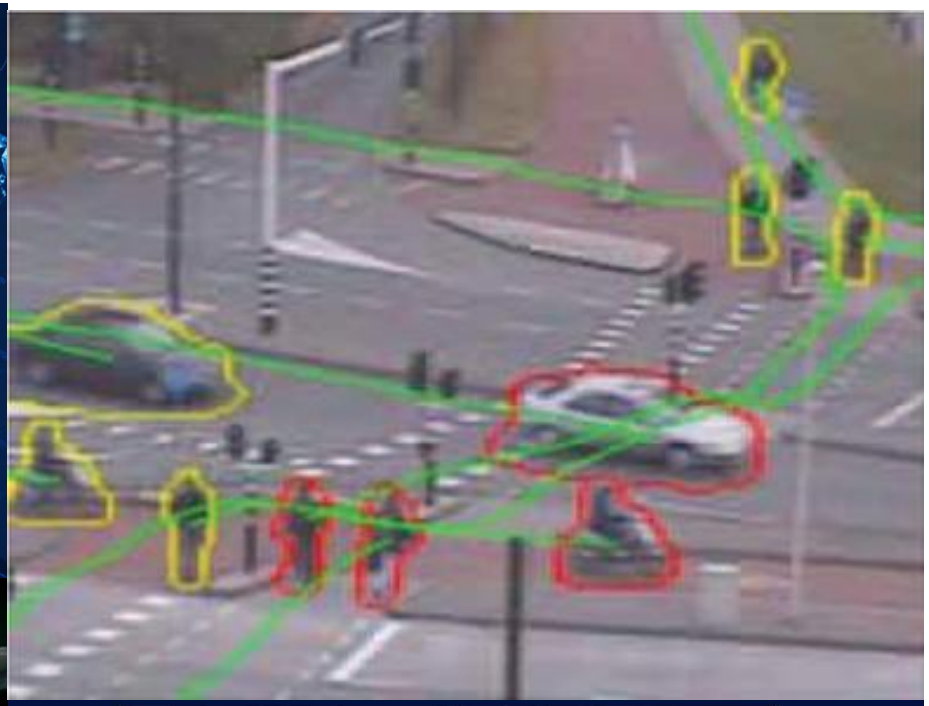


Sources

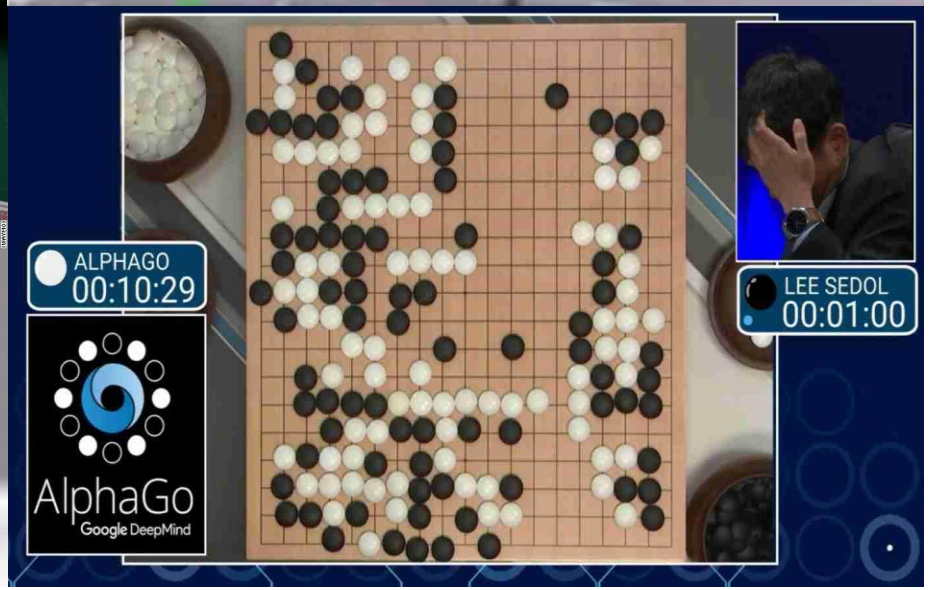
- Ian Goodfellow and Yoshua Bengio and Aaron Courville: *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org
- PyTorch
- HuggingFace library
- TensorFlow

Deep learnig

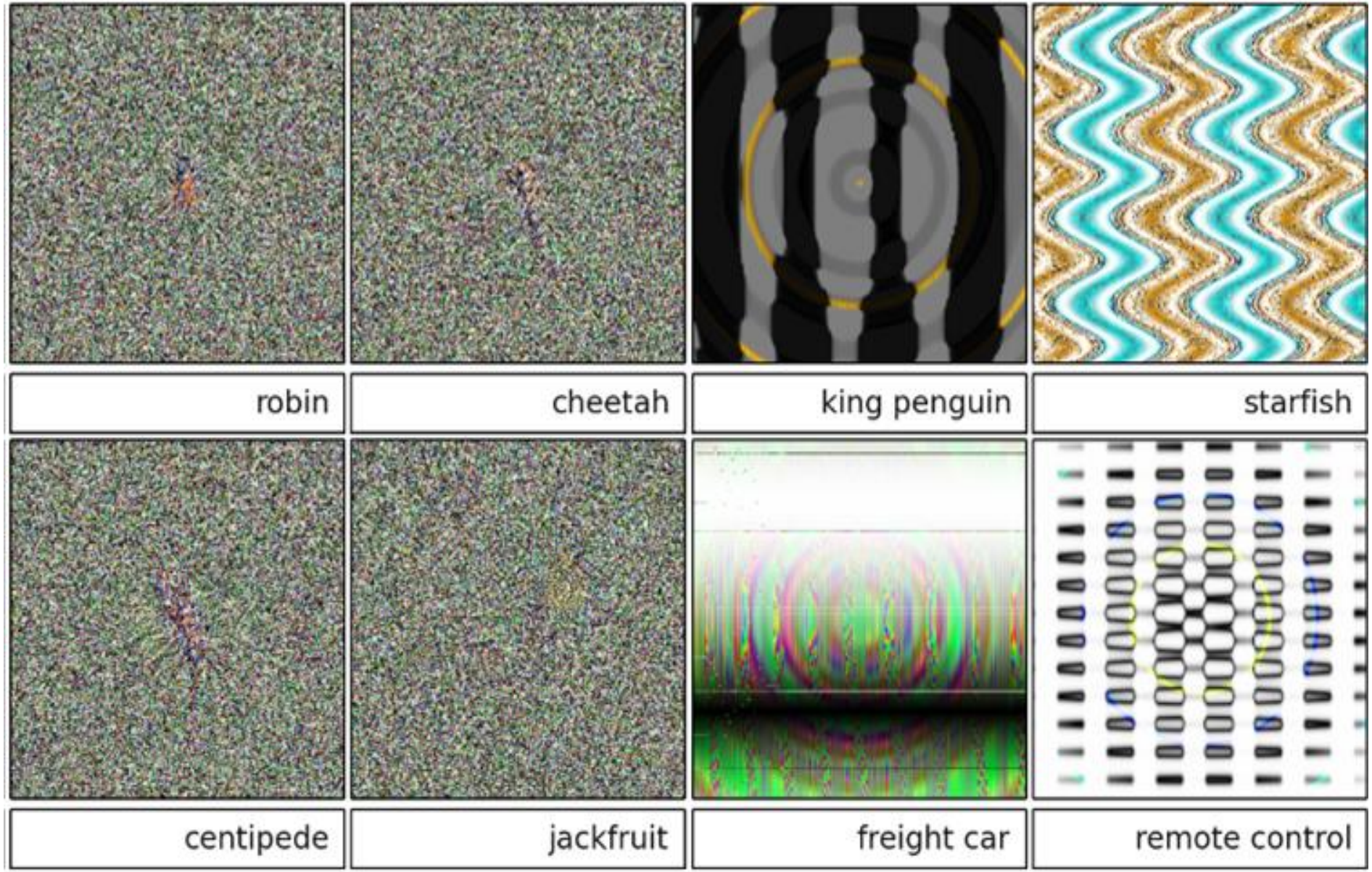




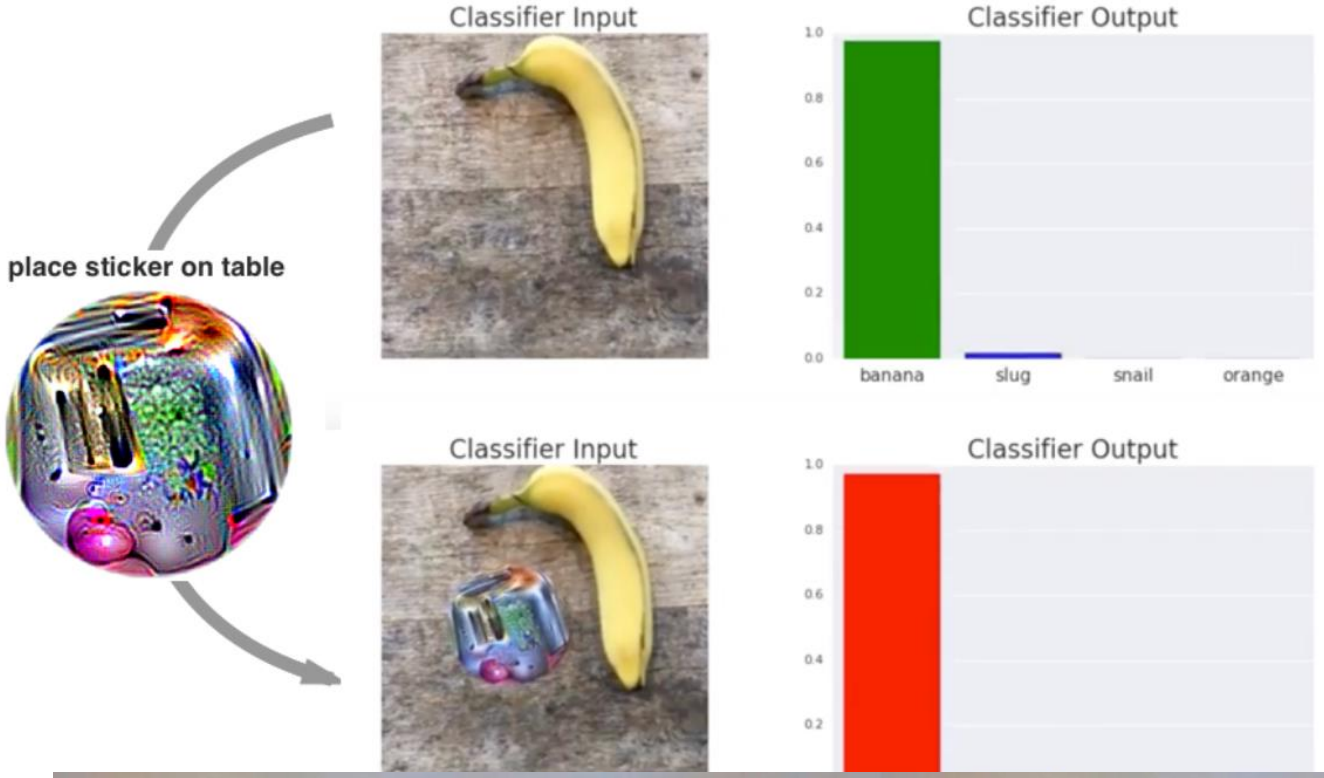
Microsoft claims new speech recognition record, achieving a super-human 5.1% error rate

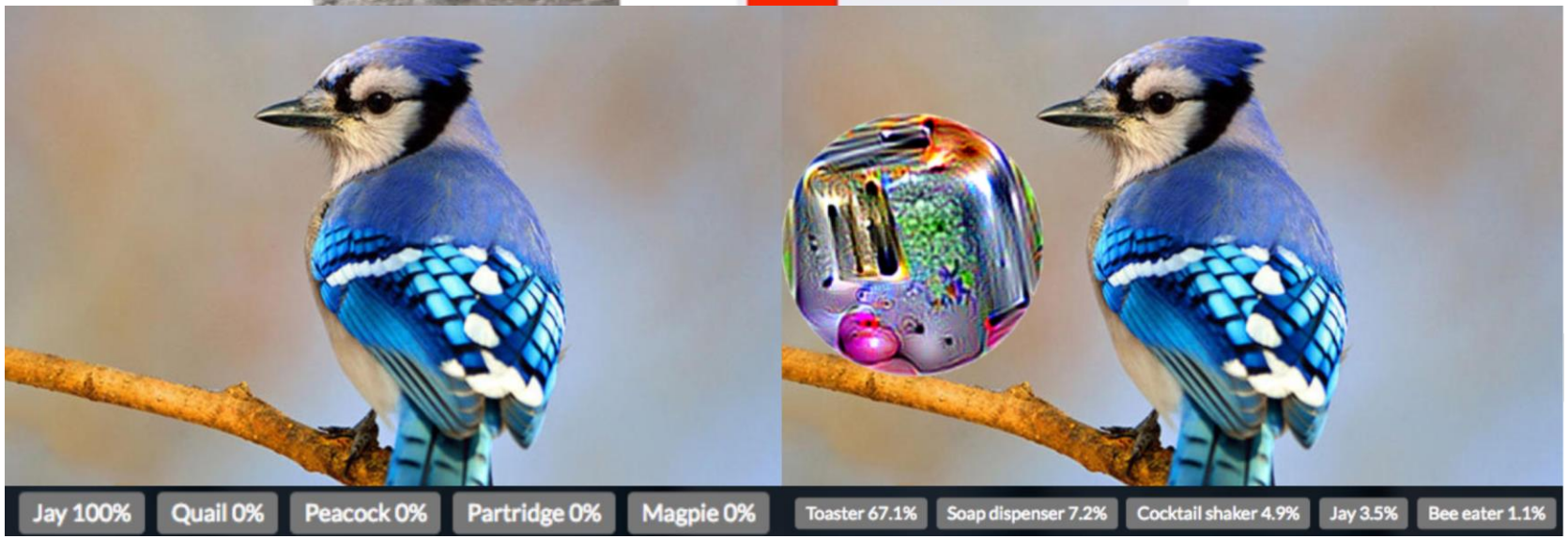


Weaknesses of deep learning



Attacks on neural networks





Failures on out-of-distribution examples

Michael A. Alcorn, Qi Li, Zhitao Gong, Chengfei Wang, Long Mai, Wei-Shinn Ku, Anh Nguyen (2018):
Strike (with) a Pose: Neural Networks Are Easily Fooled by Strange Poses of Familiar Objects. arXiv:1811.11553