Machine Learning in a Nutshell

15-488 Spring '20

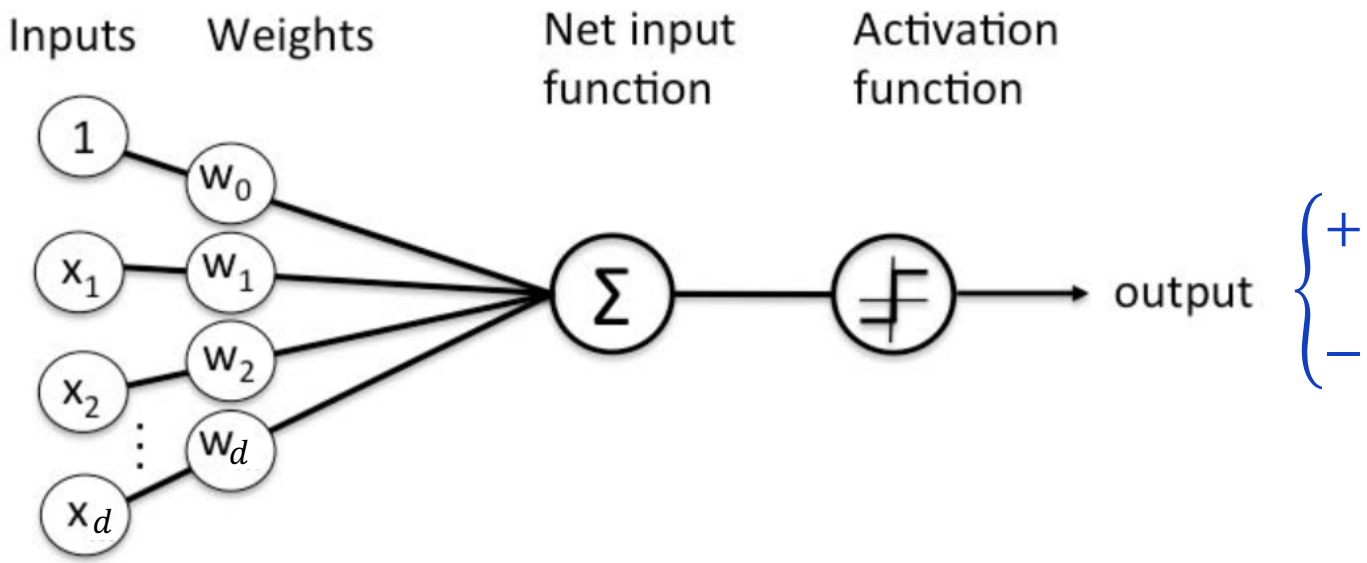
Lecture 32:
Neural Networks 1

Teacher:
Gianni A. Di Caro

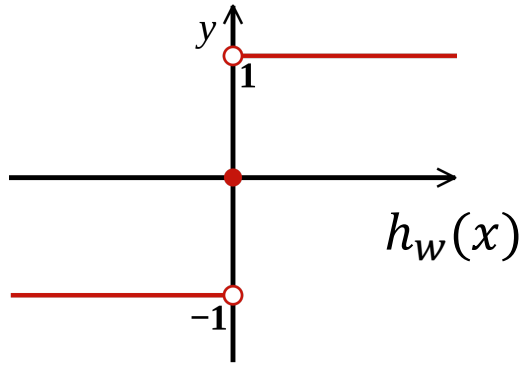
جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Linear models for Classification (focus on binary classification)

$$h_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$$

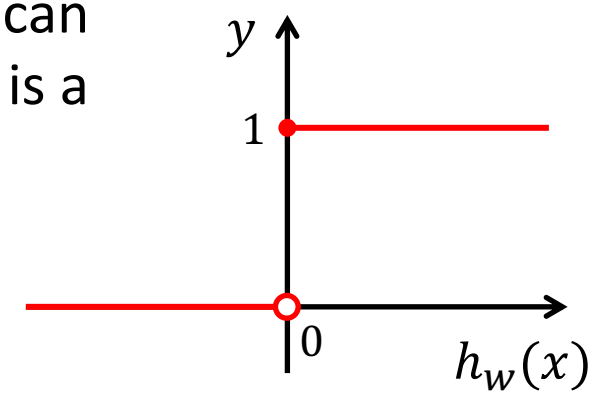


$$y = \text{sgn}(\boldsymbol{w}^T\boldsymbol{x}) = \text{sgn}\left(\sum_{j=1}^{d} w_j x_j + b\right)$$



Other transformations of $h_{\boldsymbol{w}}(\boldsymbol{x})$ can be used, as long as the output y is a two-value set

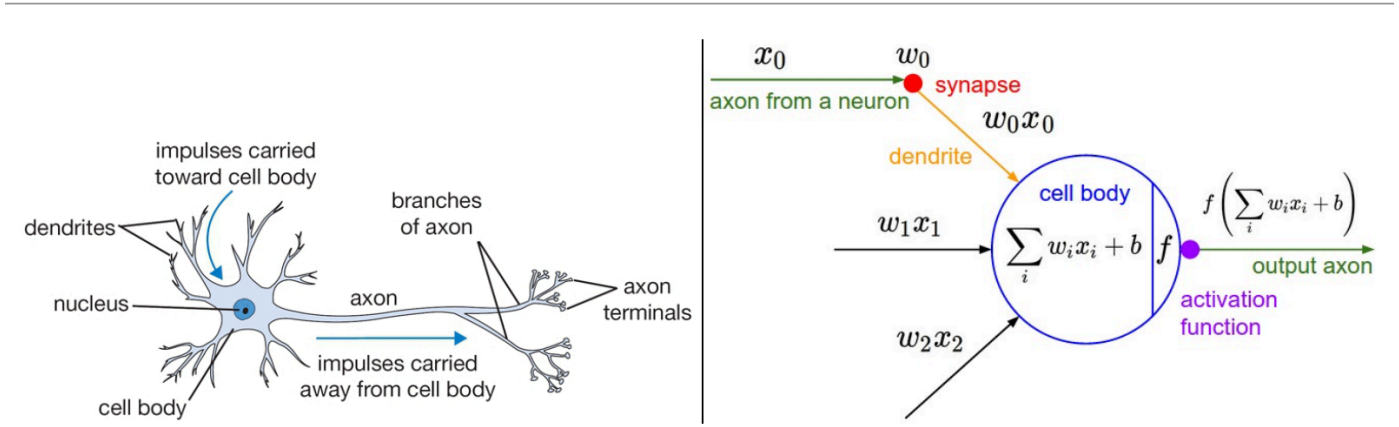
- **Class labels $\boldsymbol{y}$**: $\{0,1\}$, $\{-1,1\}$, ... $\{a,b\}$, $\{+,-\}$



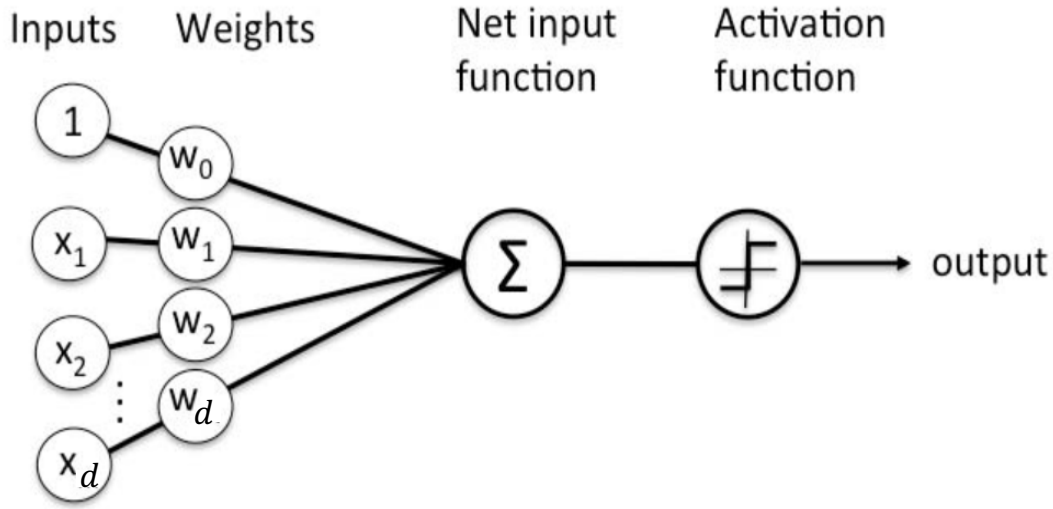
$$y = \begin{cases} 1 & if \quad \sum_{j=1}^{d} w_j x_j \geq threshold \\ -1 & if \quad \sum_{j=1}^{d} w_j x_j < threshold \end{cases}$$

threshold ≡ bias ≡ $-b$

Linear classifier: Perceptron, a long history rooted in neuroscience



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

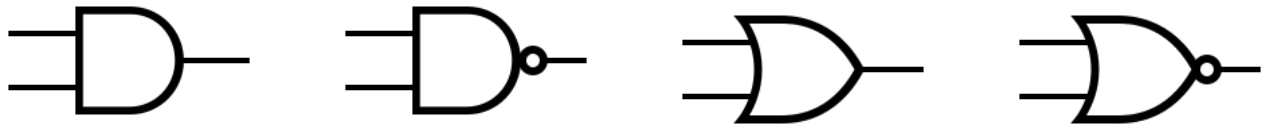


- First model of a linear classifier is the **perceptron** model, designed by F. Rosenblatt in the 50's-60's as the model of an *artificial neuron.*
- In turn, Rosenblatt, was inspired by the earlier seminal work of W. McCulloch and W. Pitts (1943) about the definition of *formal neurons* as threshold logic units for building a computing machine

- A neuron ***activates / fires*** (e.g., output passes from 0 to 1) if the sum of the inputs reaches a certain internal threshold.
- The perceptron machine (yes, hardware!) was developed in 1958 by F. Rosenblatt, designed after the model of Hebbian learning in **neurons**: Input weights can be learn to let the neuron responding according to a desired mapping between input and output

Perceptron and logic gates

- The original work of McCulloch and Pitts was about identifying threshold units to implement computing machines, that can be realized out of **logical gates**: AND, NAND, NOR, NOR



- In addition to make decisions (classification), perceptrons can be used to compute (or learn) elementary logical functions such as AND, OR, and NAND, that have linearly separable input domains

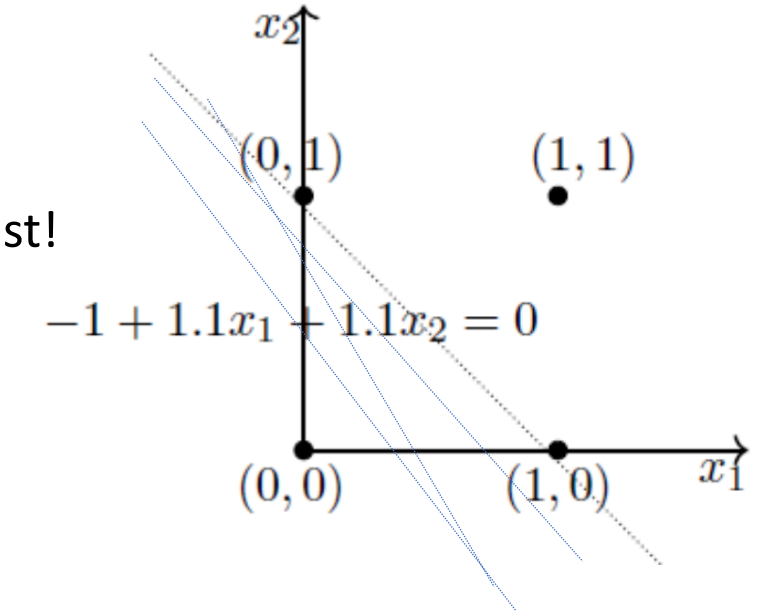
x_1	x_2	OR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0 \implies w_1 + w_2 > -w_0$$

One possible solution is:

$w_0 = -1, \quad w_1 = 1.1, \quad w_2 = 1.1$

Infinitely many other solutions exist!

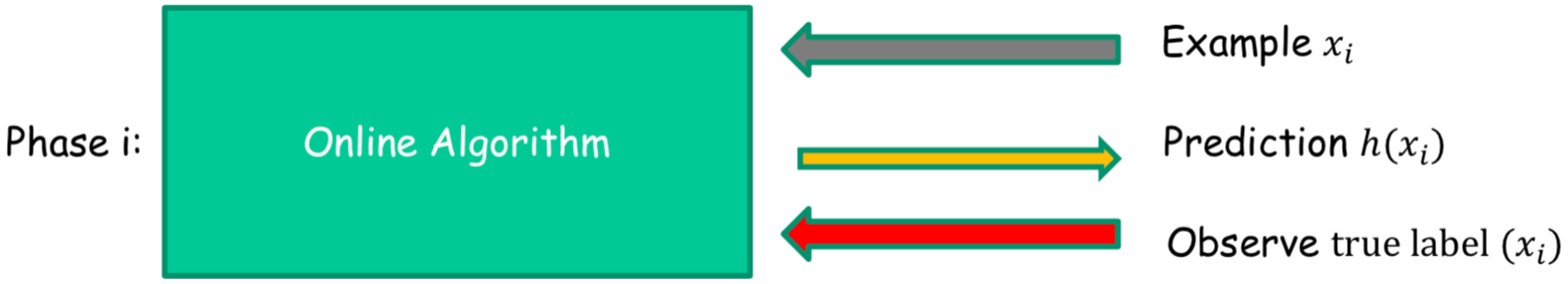


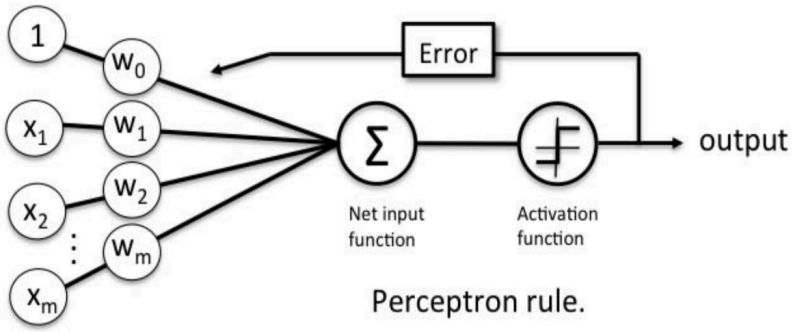
Perceptron algorithm: learning the weights

- Given α and training data $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$
- *Initialize*: $\boldsymbol{w}_0$ (e.g., $\boldsymbol{w}_0 = [0,0,\cdots,0]$)
- *Repeat* until stopping-criterion:
  - *Select* a random sample $(\boldsymbol{x}^{(k)}, y^{(k)}) \in \mathcal{D}$
  - *If* $y^{(k)} \neq \text{sgn}(\boldsymbol{w}_t^T \boldsymbol{x}^{(k)})$ *Update*: $\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha\, y^{(k)} \boldsymbol{x}^{(k)}$

Perceptron rule: on a mistake, update as follows

- Mistake on <u>positive:</u> $w_{t+1} \leftarrow w_t + x$
- Mistake on <u>negative:</u> $w_{t+1} \leftarrow w_t - x$





<u>Goal:</u> Iteratively adapt the weights to minimize classification mistakes

Perceptron algorithm: learning the weights

- ✓ **Natural greedy procedure:**
  - If true label of x is +1 and w_t is incorrect on x we have that $w_t \cdot x < 0$
  - After the update, w_{t+1} becomes equal to $w_{t+1} \leftarrow w_t + x$
  - Therefore, $w_{t+1} \cdot x = w_t \cdot x + x^2$ where the positive term x^2 has been added, such that there's more chance that now $w_{t+1} \cdot x$ is positive and would classify x correctly
  - Similarly for mistakes on negative examples.

Let's state perceptron procedure using slightly different words: at each step by use a random sample and use it to adapt the weights minimizing a loss function

- **Loss function**: squared error, $\ell\left(\boldsymbol{w}, \left(\boldsymbol{x}^{(i)}, y^{(i)}\right)\right) = \frac{1}{2}\left(y^{(i)} - \hat{y}^{(i)}(\boldsymbol{w})\right)^2$
- **Gradients**: of the squared error function
- **Stochastic gradient descent** (SGD) perform gradient descent on a single sample (approximated gradient)

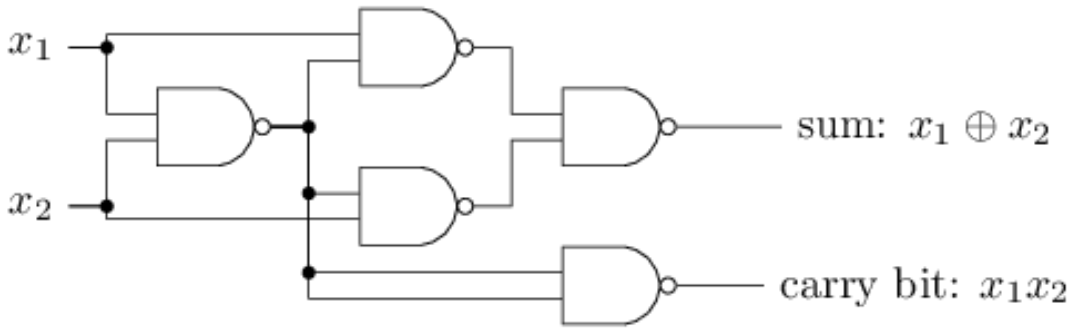
Beyond the perceptron

- The perceptron can learn a **linear function**
- It can used to learn or implement logic gates
- → In principle a **network of perceptrons** can be used to learn <u>any boolean function (non-linear)</u>

000	+
001	−
010	−
011	+
100	−
101	−
110	+
111	+

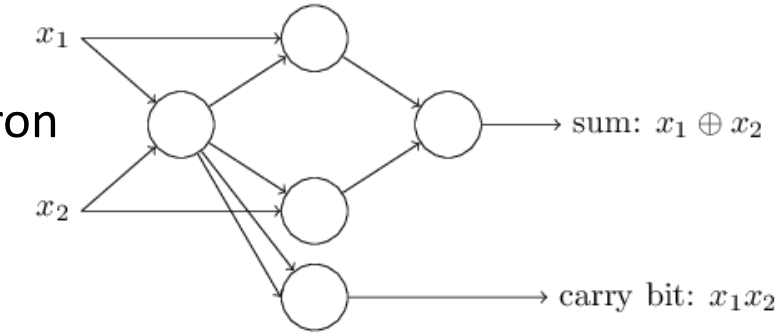
Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

A function adding two bit



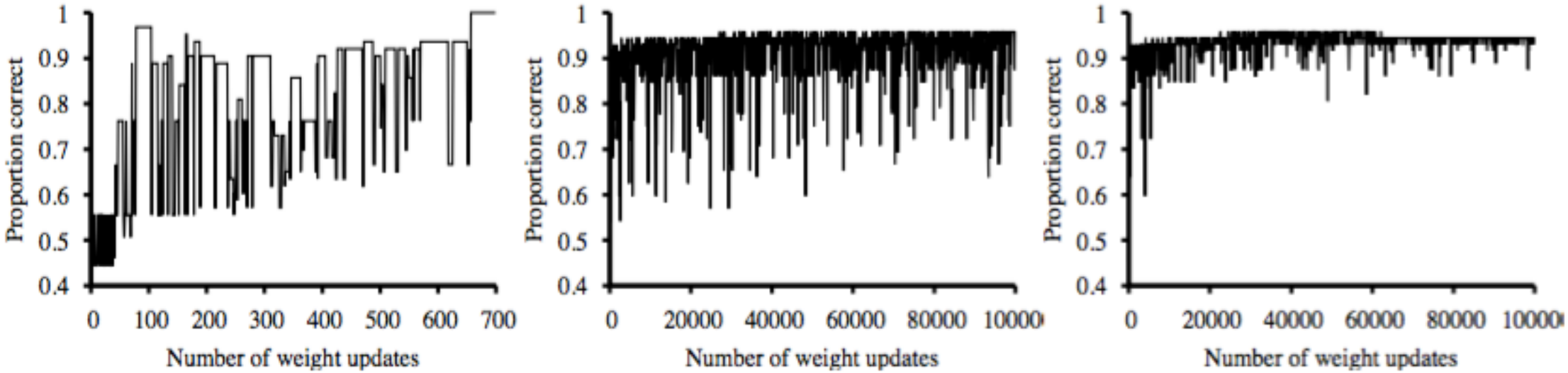
- ✓ How do we learn the weights in these complex networks of perceptron units for (non-linear) boolean functions?
- ✓ What about non-linear functions of continuous inputs?

Using perceptron units



Perceptron algorithm: properties

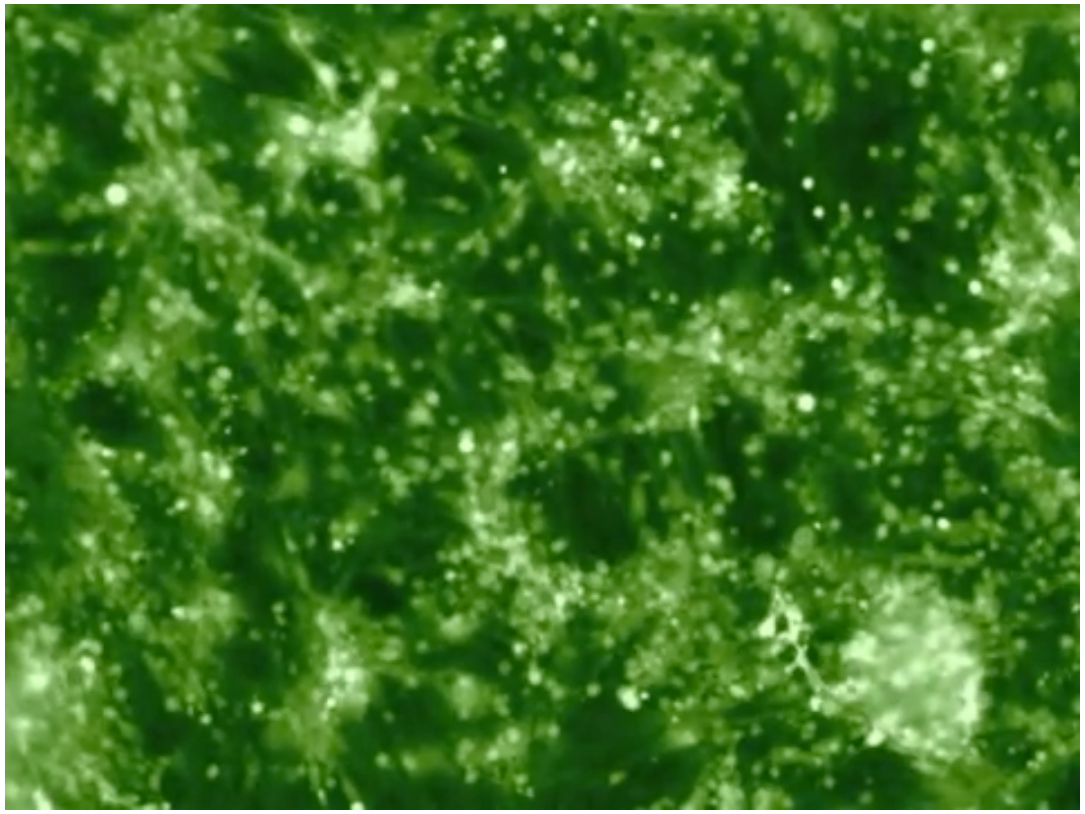
- If the problem is not linearly separable, the perceptron update rule may not converge
- Convergence to *some* linear separator is guaranteed if α decays as $O\left(\frac{1}{t}\right)$ and data are presented randomly



- Left: earthquake, separable data set
- Middle: earthquake, non-separable data set
- Right: earthquake, non-separable data set, $\alpha(t) = 1000/(1000 + t)$
- Observation: Quite hectic and unpredictable behavior

Artificial Neural Networks

- ✓ The idea of mimicking (to some extent) what's going on in the human brain makes sense
- ○ Parallel and Distributed Network of firing / activation units (neurons) performing (complex!) computations



Human brains

- Neuron switching time ~ .001 second
- Number of neurons ~ 10^{10}
- Connections per neuron ~ 10^{4-5}
- Scene recognition time ~ .1 second
- 100 inference steps doesn't seem like enough

→ much parallel computation

Design of Artificial Neural Networks

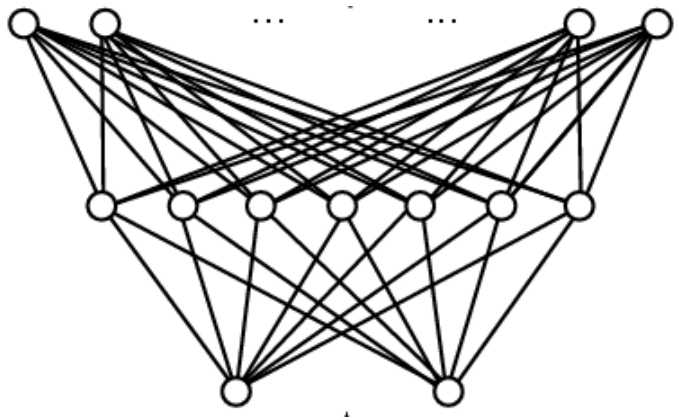
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process

Artificial Neural Networks for learning non-linear functions

❖ Goal: Learn a generic non-linear function, $f: X \rightarrow Y$

➢ Learn a parametric non-linear function, $f_w: X \rightarrow Y$

➢ Represent f_w by a (**multilayer**) **network of basic *units***



- X feature space: (vector of) continuous and/or discrete vars
- Y output space: (vector of) continuous and/or discrete vars
- f_w *(neural) network* of basic units

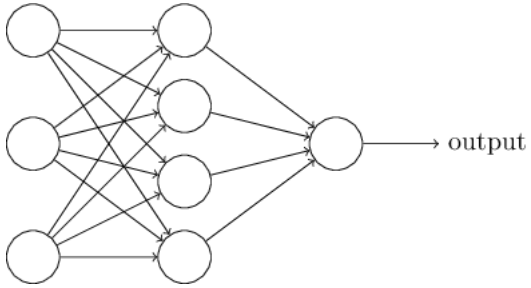
❖ A neural network is a (non-linear) parametric *function approximator*

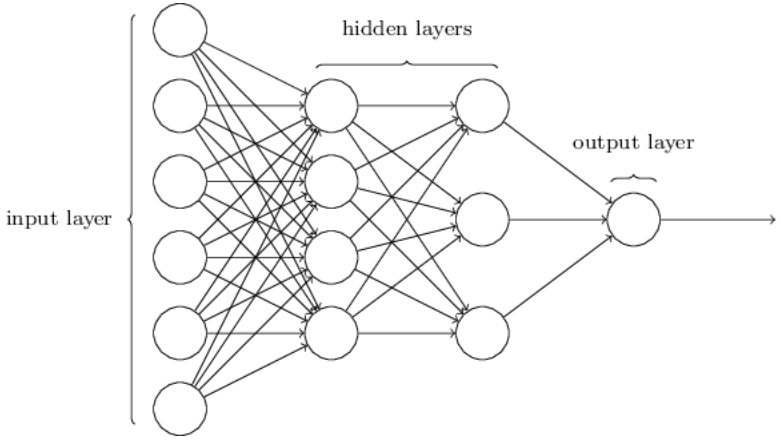
Learning algorithm: given $(x_d, t_d)_{d \in D}$, train weights w of all units to minimize sum of squared errors of predicted network outputs.

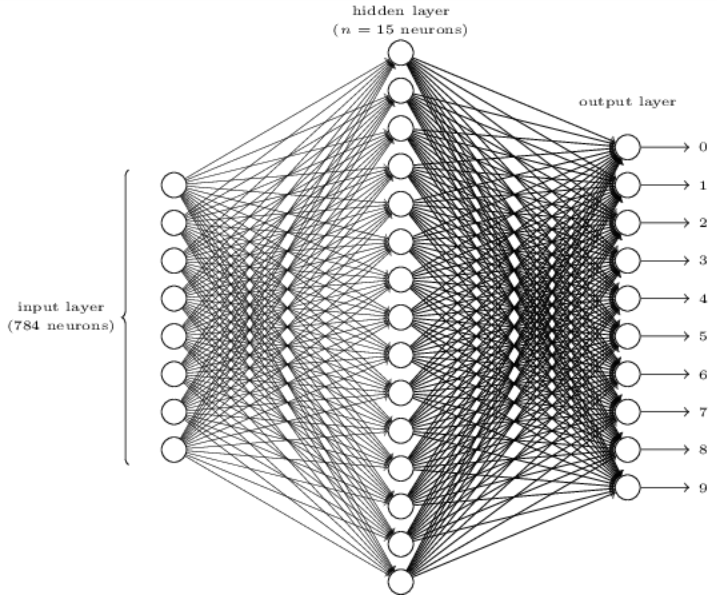
Find parameters w to minimize $\sum_{d \in D} (f_w(x_d) - t_d)^2$

✓ Gradient descent

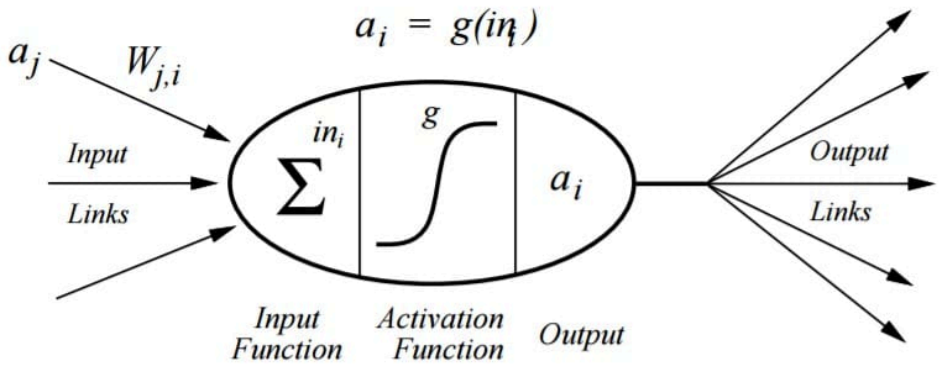
Architecture of a NN: Multi-Layer Perceptron, units







- ✓ One **input** layer
- ✓ One **output** layer
- ✓ One or more **hidden** layers
  - A single perceptron has no hidden layers
  - ❖ Hidden layers allow to learn non-linear functions

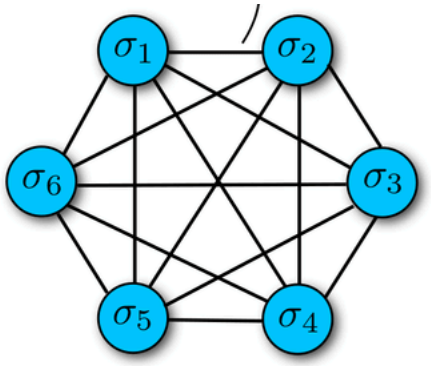


Feed-Forward MLP

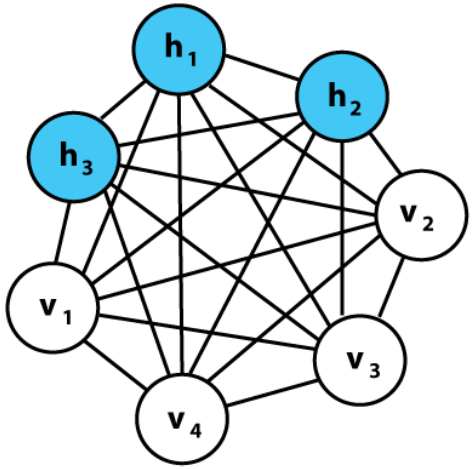
Deep network (> 3 hidden layers?)

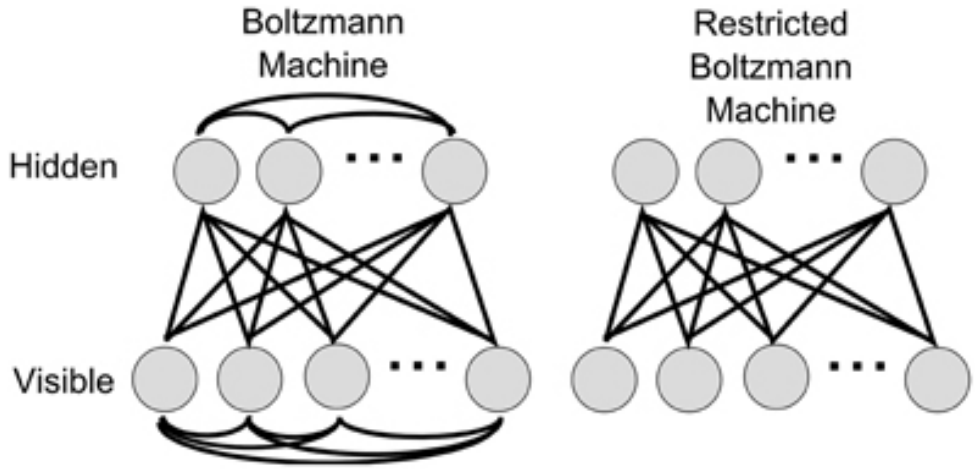
$$a_i = g(\sum_j W_{j,i} a_j)$$

Architecture of a NN: Recurrent NNs



Hopfield Network





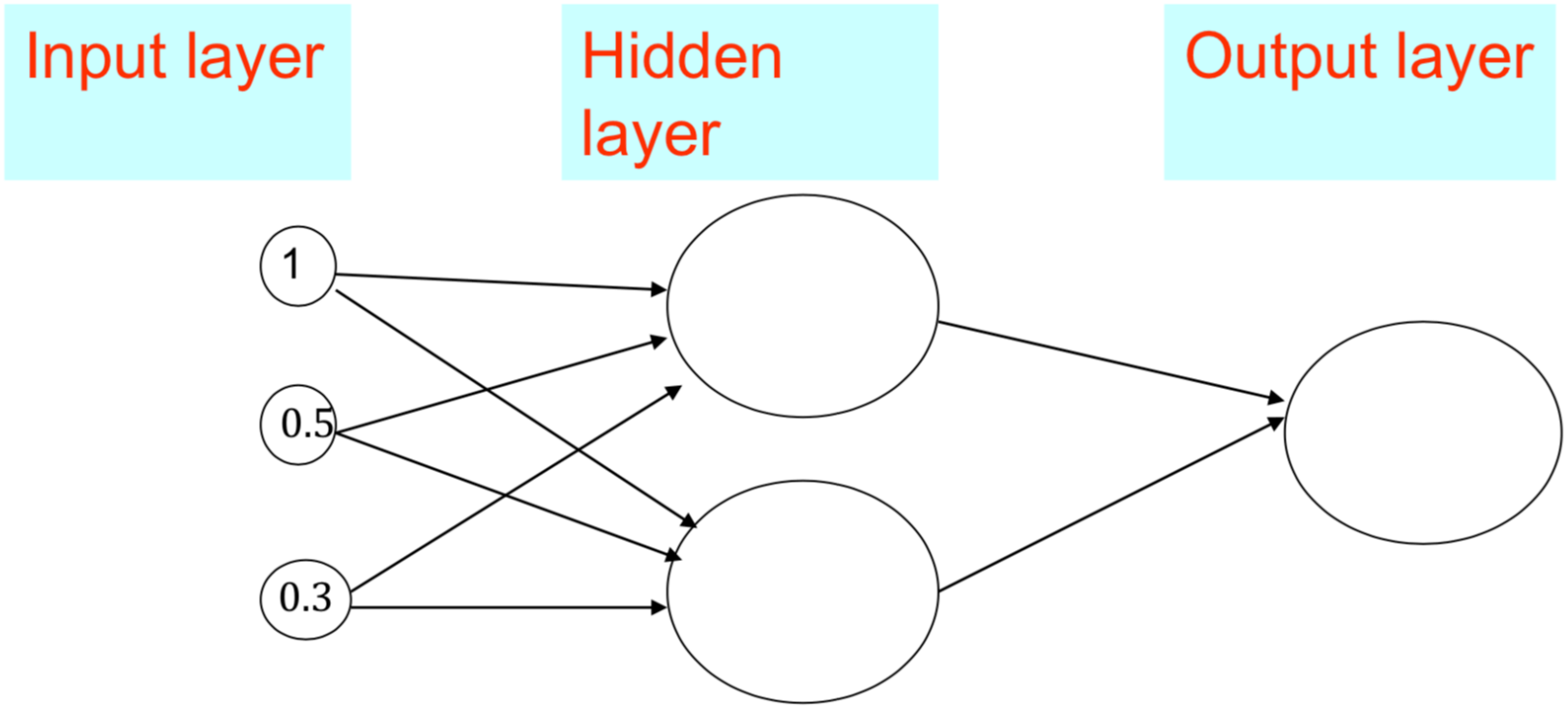
- ✓ A **set of visible** units
- ✓ Zero or more **sets of hidden** units
- ✓ **Undirected graph**, symmetric connections between the units
- ✓ The network has a **state**

- Unsupervised learning problems (learn input models)
- Optimization problems
- …

What type of units should we use? (MLPs)

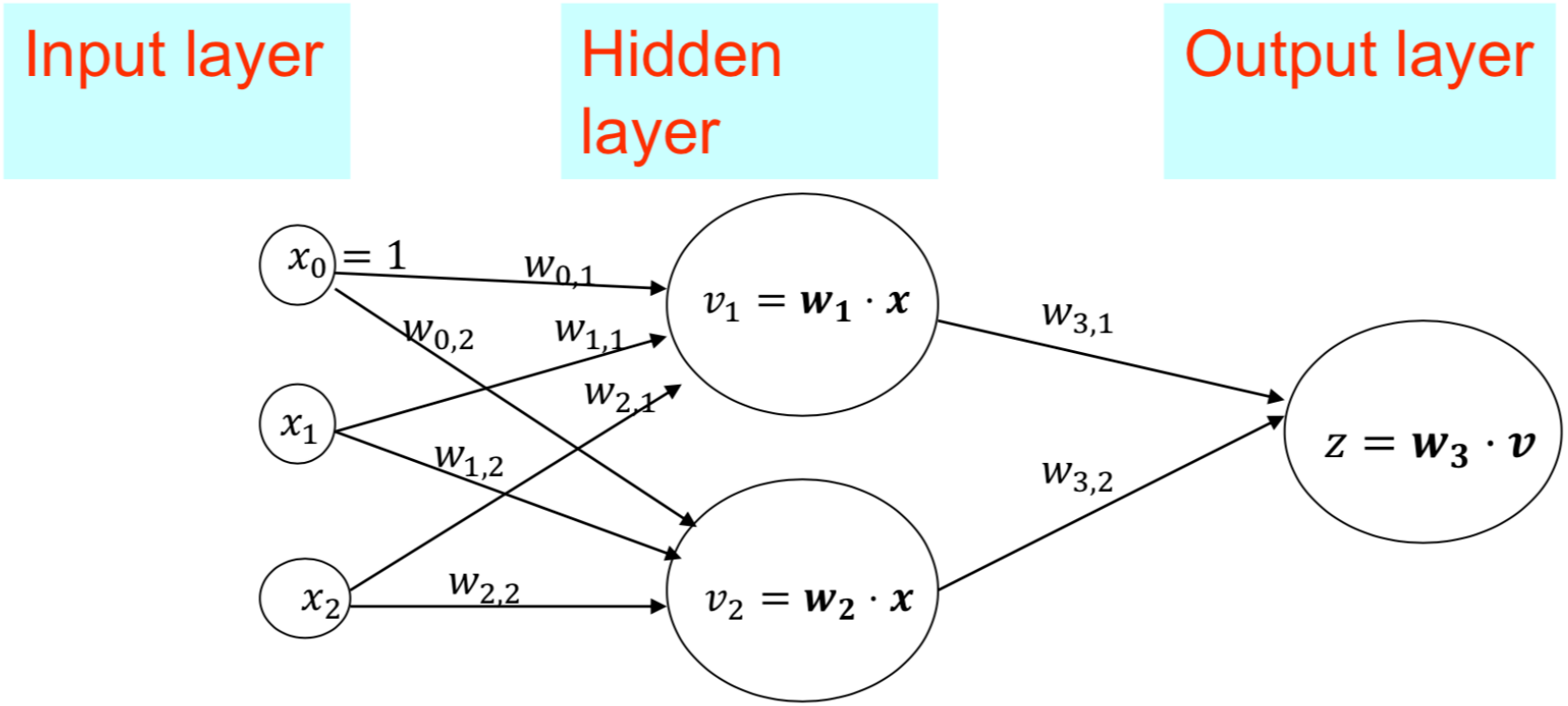
- The classifier / regressor is a multilayer network of units

  - Each *unit* takes some inputs and produces one output. Output of one unit can be the input of another.



Multi-layer network of linear units?

- Advantage: we know how to do gradient descent on linear units



Problem: linear of linear is just linear.

$$z = w_{3,1}(\boldsymbol{w_1} \cdot \boldsymbol{x}) + w_{3,2}(\boldsymbol{w_2} \cdot \boldsymbol{x}) = \left(w_{3,1}\boldsymbol{w_1} + w_{3,2}\boldsymbol{w_2}\right) \cdot \boldsymbol{x} = \text{linear}$$

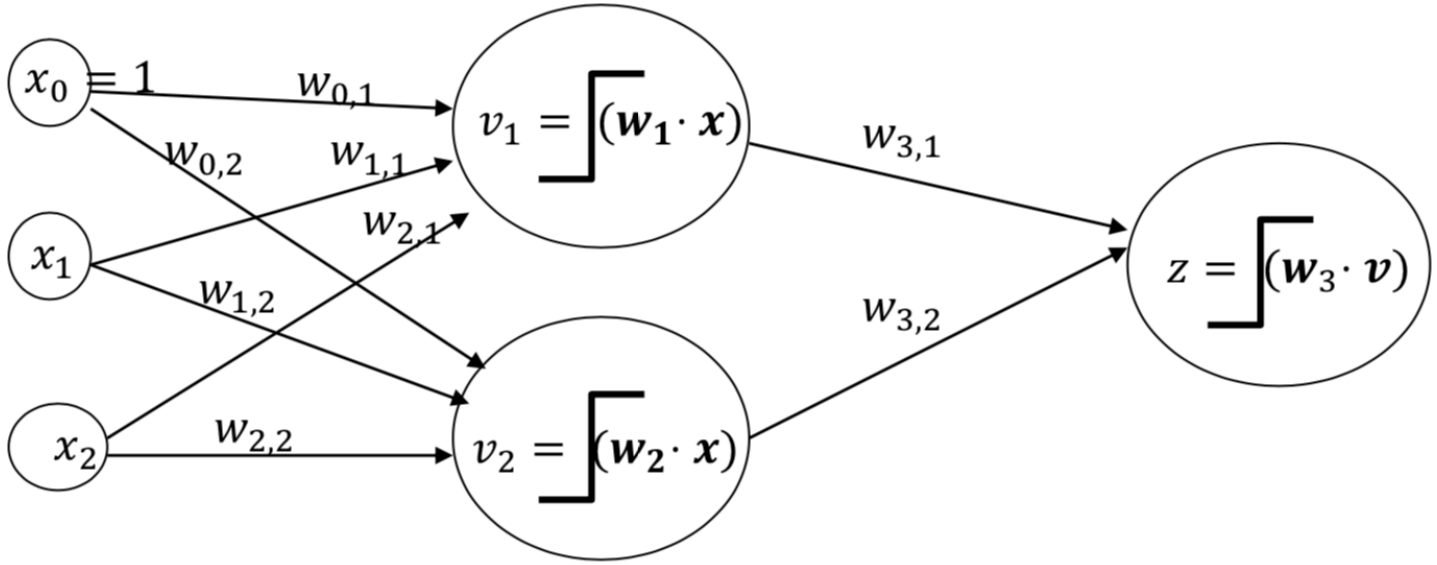
Multi-layer network of perceptron units?

- Advantage: Can produce highly non-linear decision boundaries!

Input layer

Hidden layer

Output layer



Threshold function: $\lceil x = 1$ if x is positive, 0 if x is negative.

Problem: discontinuous threshold is not differentiable. Can't do gradient descent.

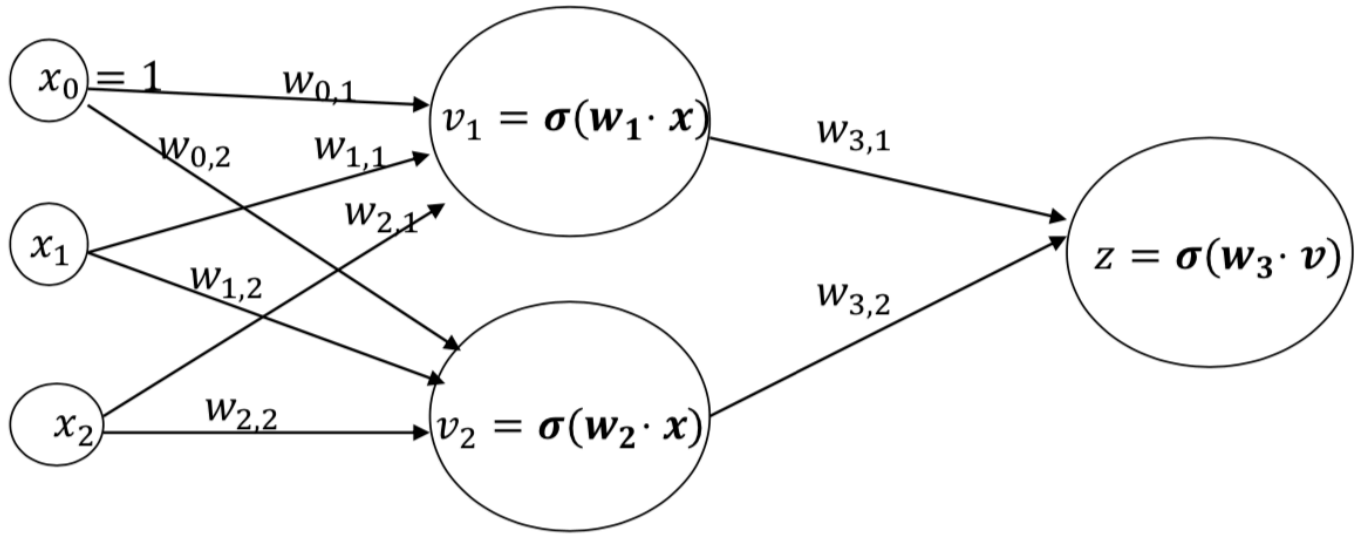
Multi-layer network of sigmoid units?

- Advantage: Can produce highly non-linear decision boundaries!
- Sigmoid is differentiable, so can use gradient descent

Input layer

Hidden layer

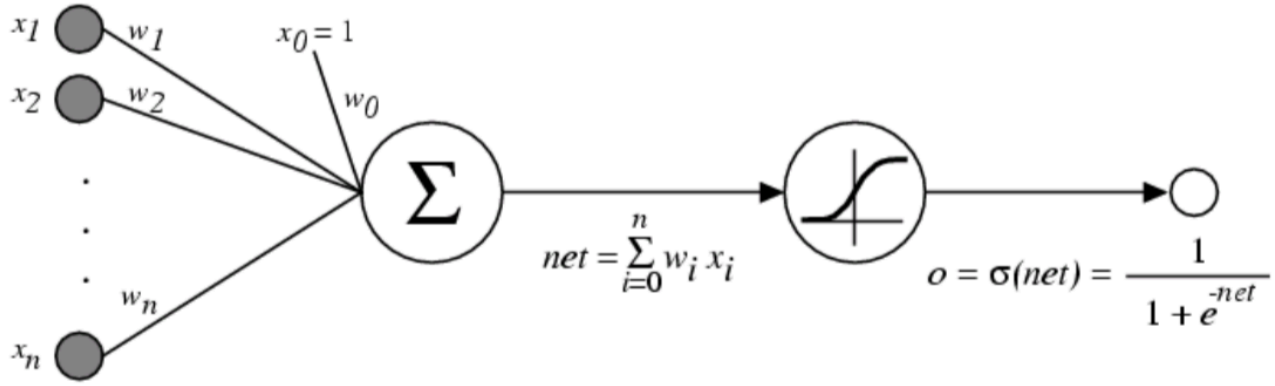
Output layer



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

Very useful in practice!

Sigmoid unit (Logistic regression!)



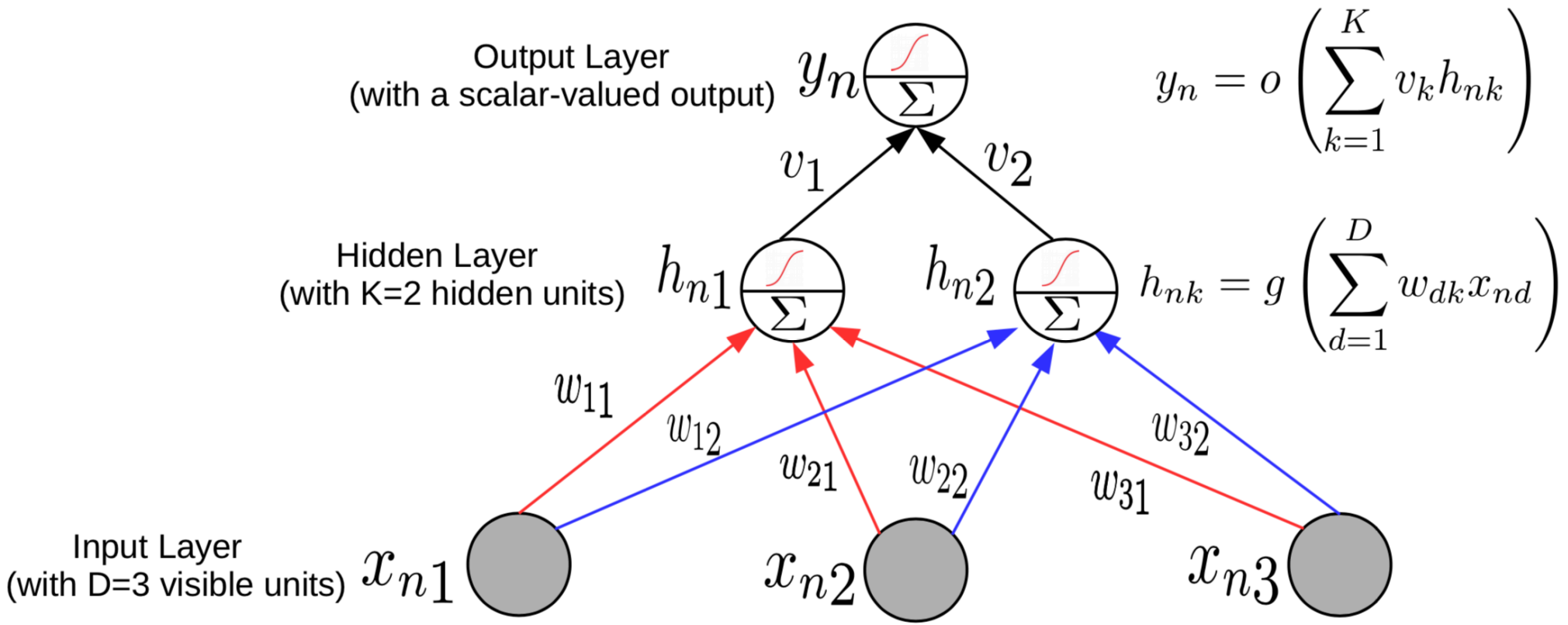
σ is the sigmoid function; $\sigma(x) = \frac{1}{1+e^{-x}}$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)\big(1 - \sigma(x)\big)$

We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

Let's give a more in-depth look to MLPs *(slides from Piyush Rai)*



- Each node (a.k.a. unit) in the hidden layer computes a nonlinear transform of inputs it receives
- Hidden layer nodes act as **features** in the final layer (a linear model) to produce the output
- The overall effect is a nonlinear mapping from inputs to outputs

Illustration: a NN with one hidden layer

- Each input $\boldsymbol{x}_n$ transformed into several "pre-activations" using linear models

$$a_{nk} = \boldsymbol{w}_k^\top \boldsymbol{x}_n = \sum_{d=1}^{D} w_{dk} x_{nd}$$

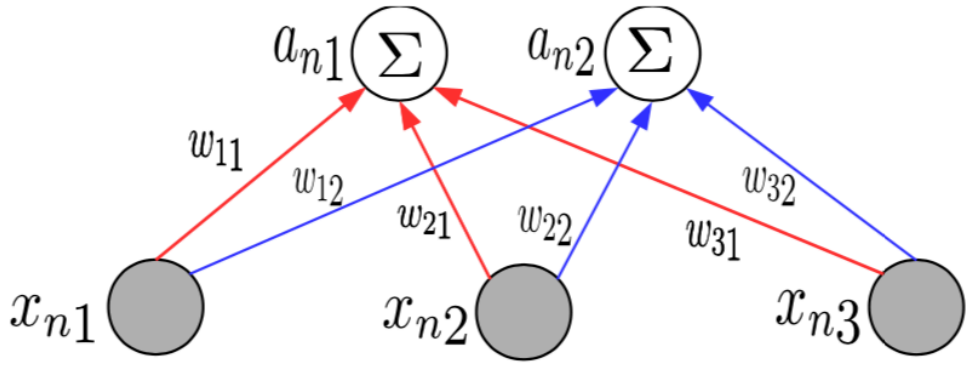


Illustration: a NN with one hidden layer

- Each input $\boldsymbol{x}_n$ transformed into several "pre-activations" using linear models

$$a_{nk} = \boldsymbol{w}_k^\top \boldsymbol{x}_n = \sum_{d=1}^{D} w_{dk} x_{nd}$$

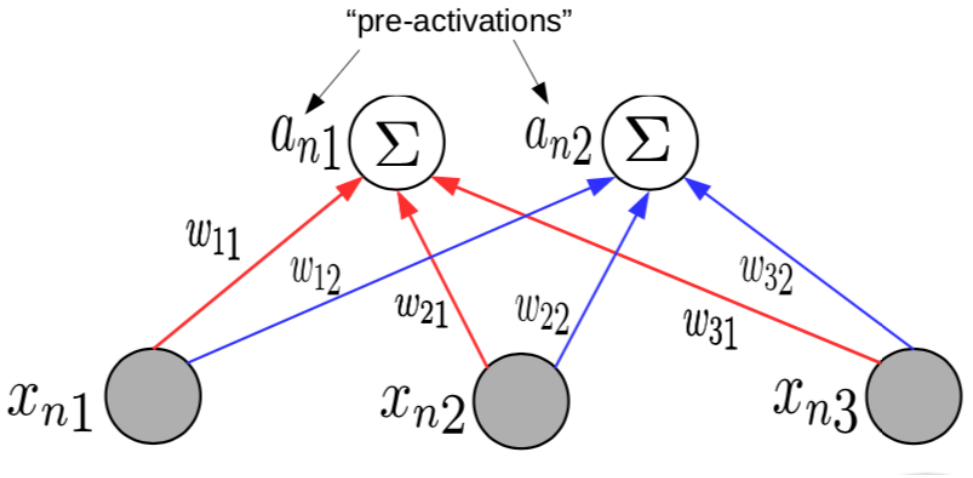


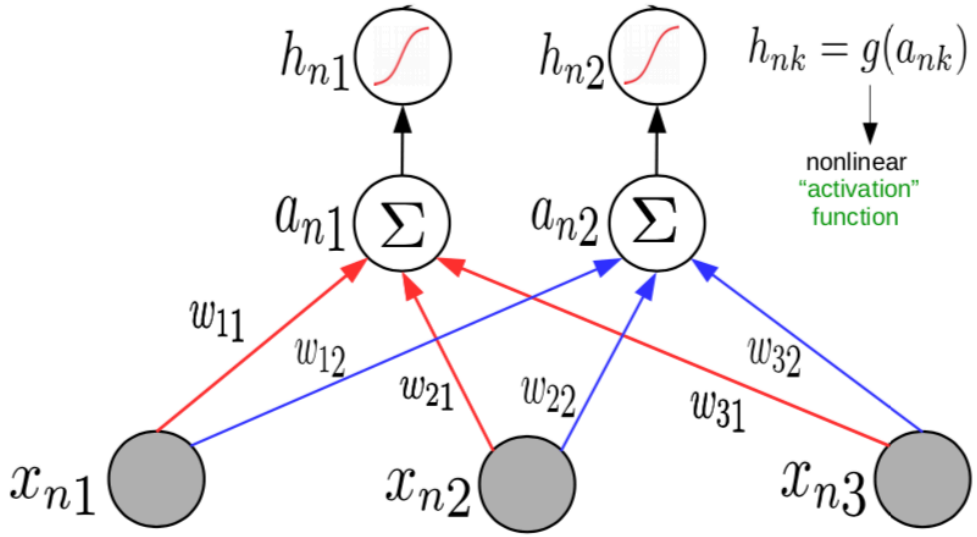
Illustration: a NN with one hidden layer

- Each input $\boldsymbol{x}_n$ transformed into several "pre-activations" using linear models

$$a_{nk} = \boldsymbol{w}_k^\top \boldsymbol{x}_n = \sum_{d=1}^{D} w_{dk} x_{nd}$$

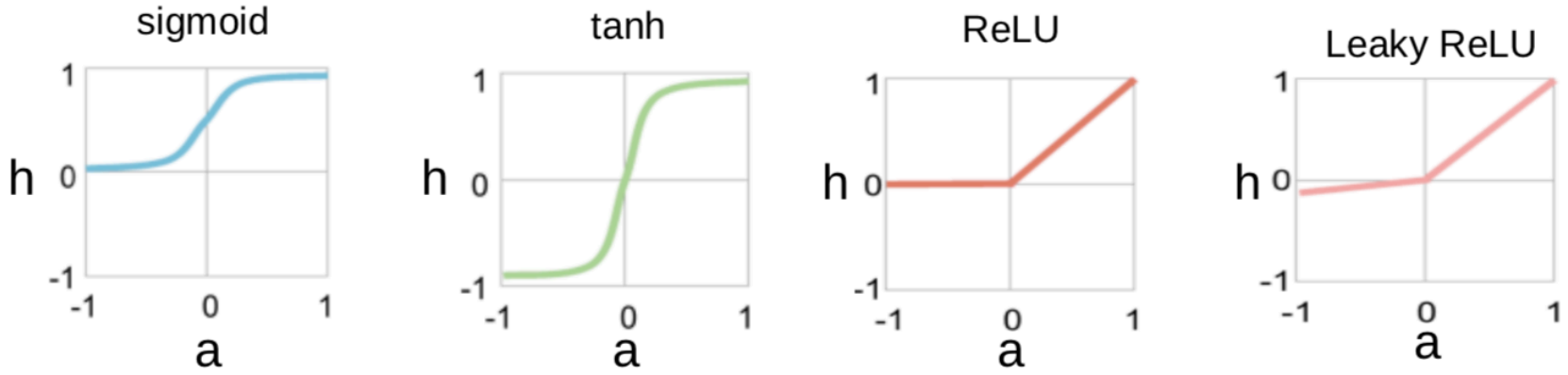
- Nonlinear activation applied on each pre-activation

$$h_{nk} = g(a_{nk})$$



Common activation functions: Optimization matters!

- Some common activation functions



- Sigmoid: $h = \sigma(a) = \frac{1}{1+\exp(-a)}$
- tanh (tan hyperbolic): $h = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = 2\sigma(2a) - 1$
- ReLU (Rectified Linear Unit): $h = \max(0, a)$
- Leaky ReLU: $h = \max(\beta a, a)$ where β is a small postive number
- Several others, e.g., Softplus $h = \log(1 + \exp(a))$, exponential ReLU, maxout, etc.
- Sigmoid, tanh can have issues during backprop (saturating gradients, non-centered)
- ReLU/leaky ReLU currently one of the most popular (also cheap to compute)

Illustration: a NN with one hidden layer

- Each input $\boldsymbol{x}_n$ transformed into several "pre-activations" using linear models

$$a_{nk} = \boldsymbol{w}_k^\top \boldsymbol{x}_n = \sum_{d=1}^{D} w_{dk} x_{nd}$$

- Nonlinear activation applied on each pre-activation

$$h_{nk} = g(a_{nk})$$

- A linear model applied on the new "features" $\boldsymbol{h}_n$

$$s_n = \boldsymbol{v}^\top \boldsymbol{h}_n = \sum_{k=1}^{K} v_k h_{nk}$$

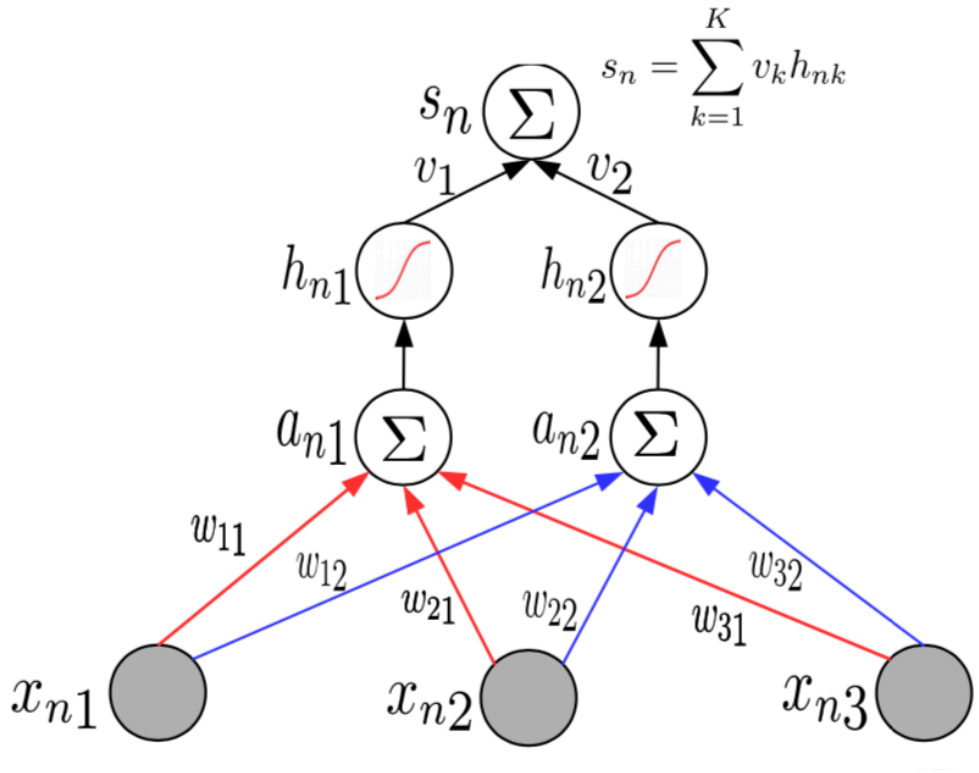


Illustration: a NN with one hidden layer

- Each input $\boldsymbol{x}_n$ transformed into several "pre-activations" using linear models

$$a_{nk} = \boldsymbol{w}_k^\top \boldsymbol{x}_n = \sum_{d=1}^{D} w_{dk} x_{nd}$$

- Nonlinear activation applied on each pre-activation

$$h_{nk} = g(a_{nk})$$

- A linear model applied on the new "features" $\boldsymbol{h}_n$

$$s_n = \boldsymbol{v}^\top \boldsymbol{h}_n = \sum_{k=1}^{K} v_k h_{nk}$$

- Finally, the output is produced as $y_n = o(s_n)$

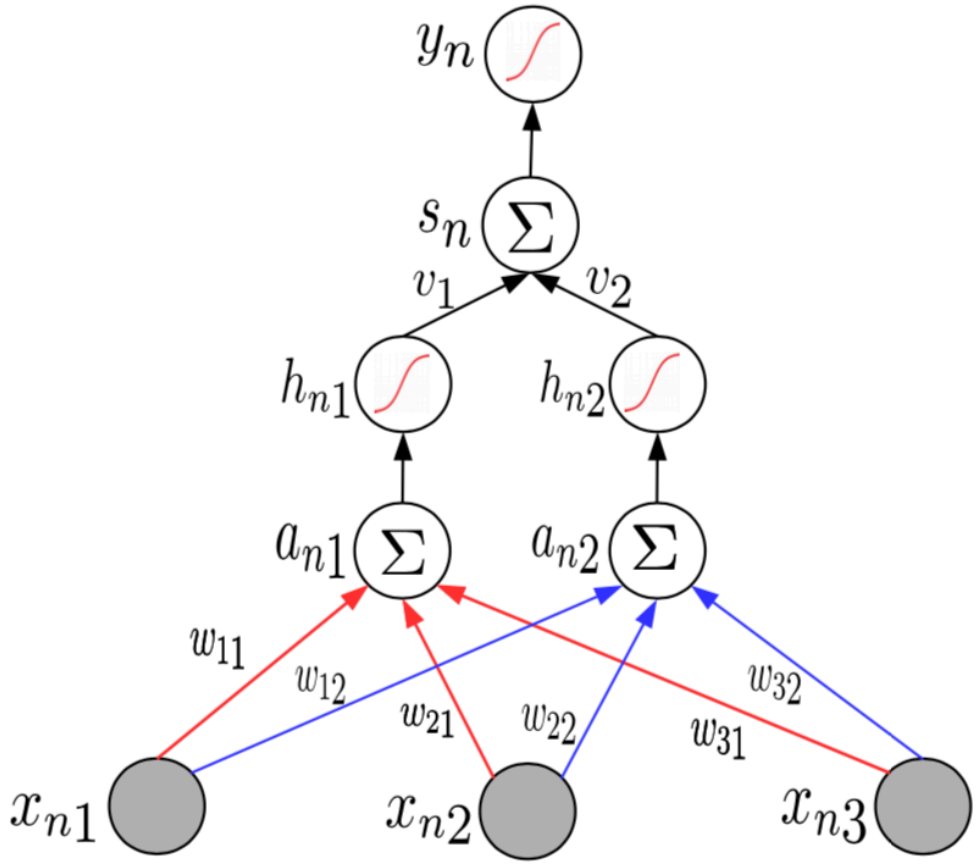


Illustration: a NN with one hidden layer

- Each input $\boldsymbol{x}_n$ transformed into several "pre-activations" using linear models

$$a_{nk} = \boldsymbol{w}_k^\top \boldsymbol{x}_n = \sum_{d=1}^{D} w_{dk} x_{nd}$$

- Nonlinear activation applied on each pre-activation

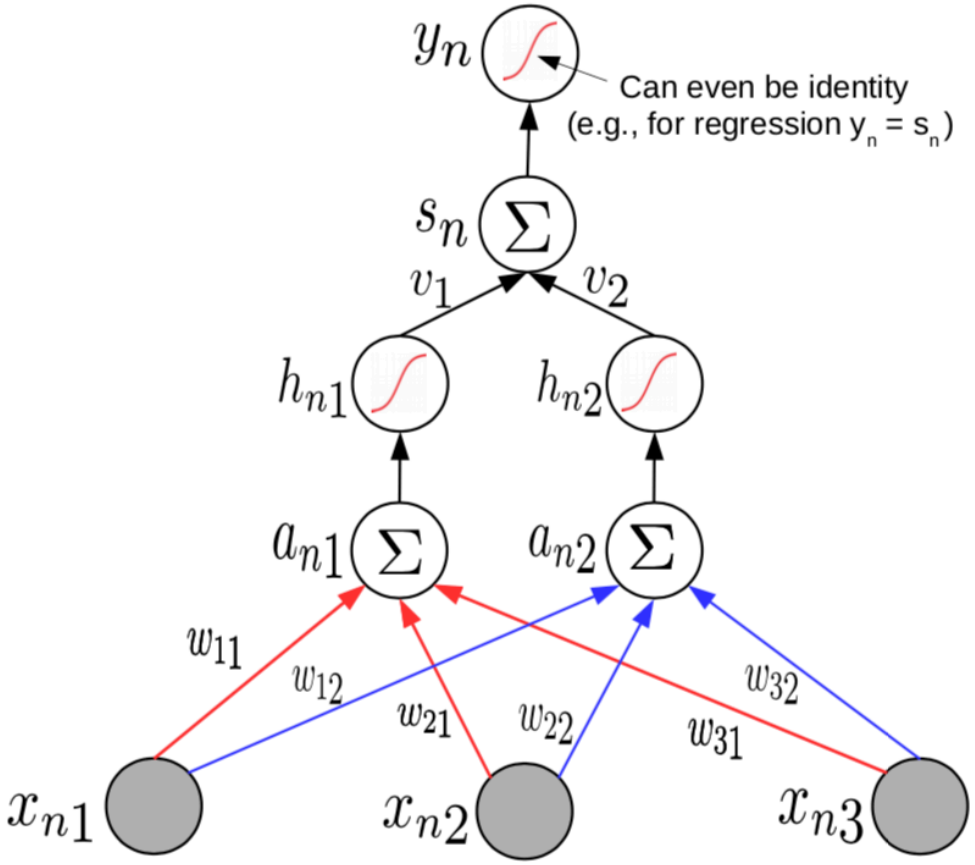
$$h_{nk} = g(a_{nk})$$

- A linear model applied on the new "features" $\boldsymbol{h}_n$

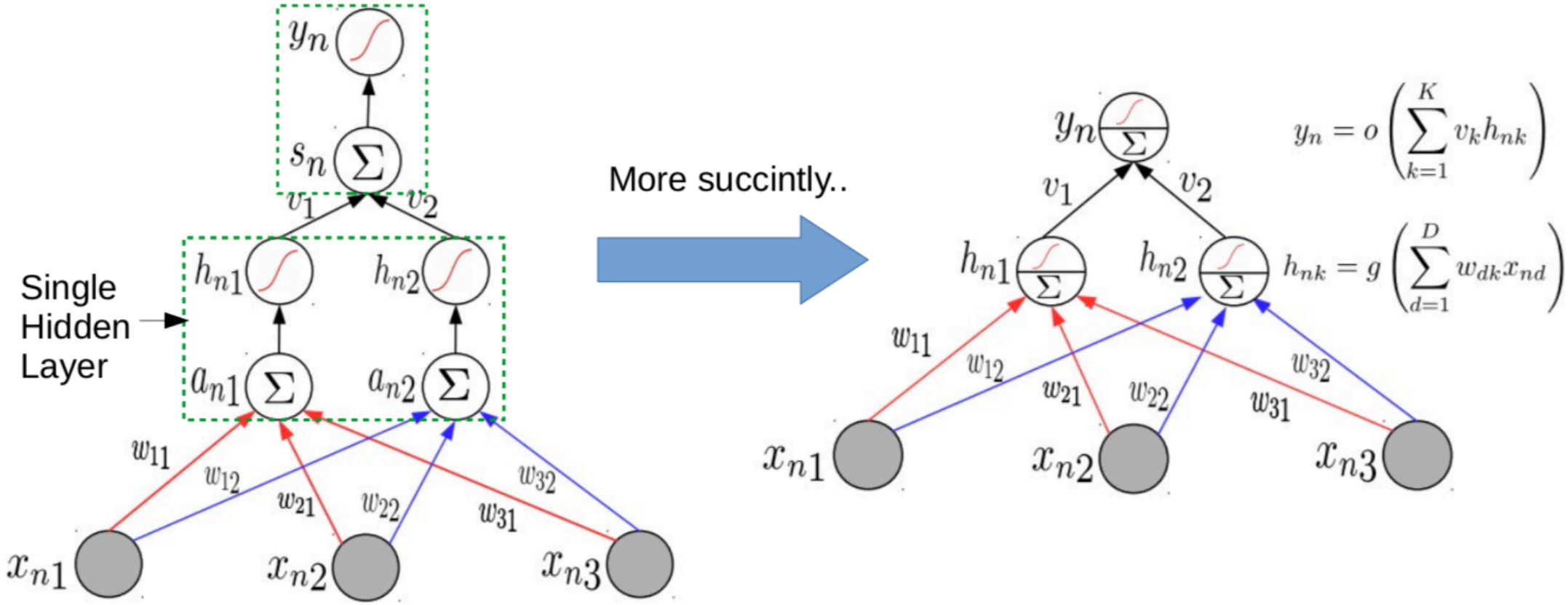
$$s_n = \boldsymbol{v}^\top \boldsymbol{h}_n = \sum_{k=1}^{K} v_k h_{nk}$$

- Finally, the output is produced as $y_n = o(s_n)$

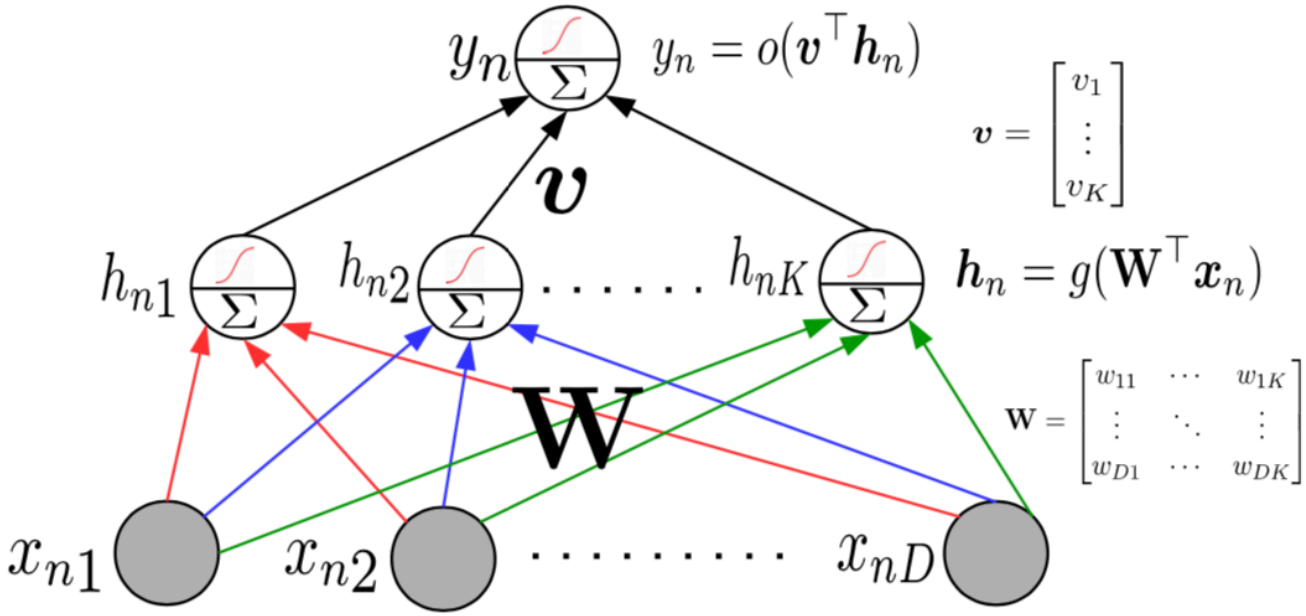
- Unknowns of the model ($\boldsymbol{w}_1, \ldots, \boldsymbol{w}_K$ and $\boldsymbol{v}$) learned by minimizing a loss $\mathcal{L}(\mathbf{W}, \boldsymbol{v}) = \sum_{n=1}^{N} \ell(y_n, o(s_n))$, e.g., squared, logistic, softmax, etc (depending on the output)



A more compact representation

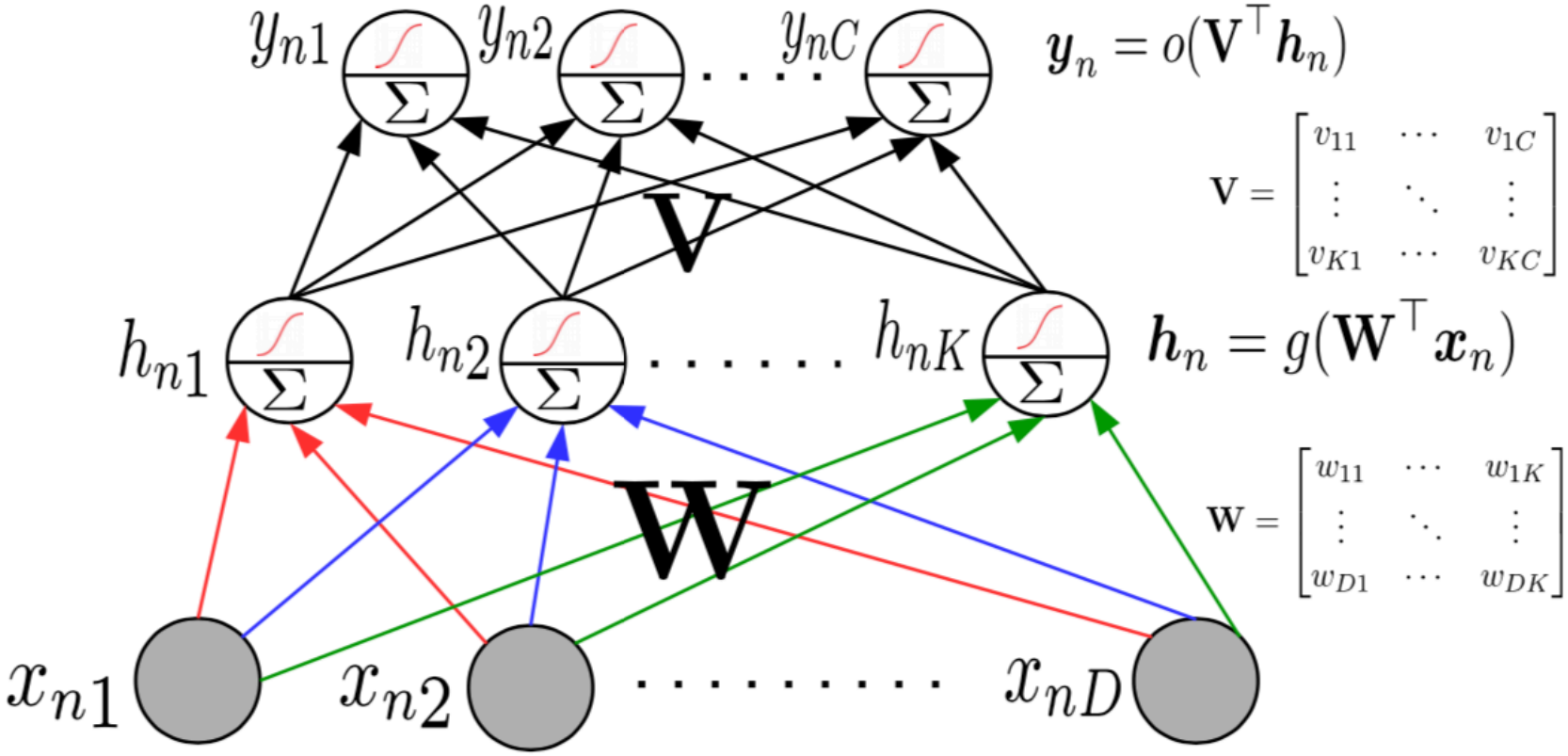


NN with one hidden layer and single output



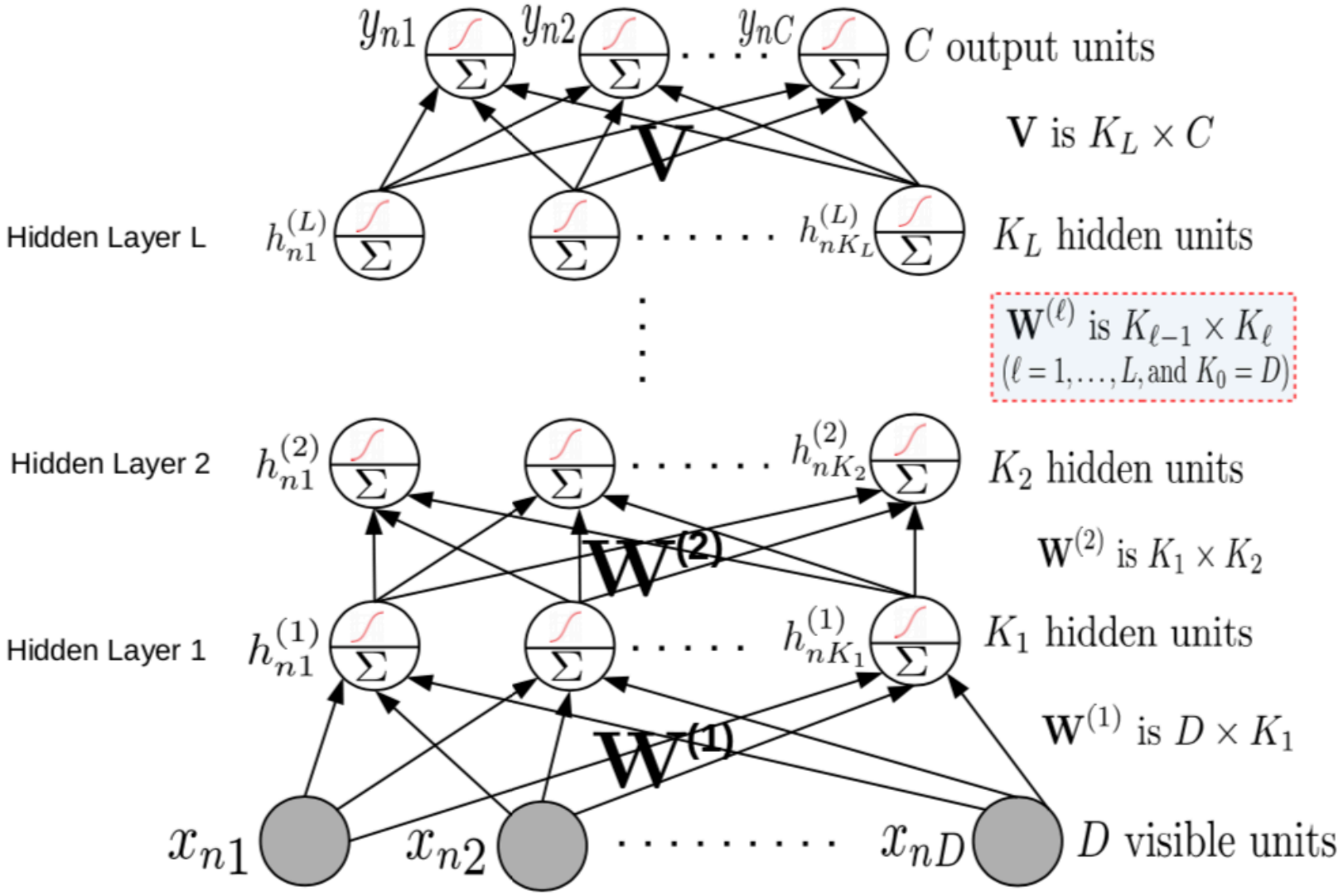
- Note: w_{dk} is the weight of edge between input layer node d and hidden layer node k
- $\mathbf{W} = [\boldsymbol{w}_1, \ldots, \boldsymbol{w}_K]$, with $\boldsymbol{w}_k$ being the weights incident on k-th hidden unit
- Each $\boldsymbol{w}_k$ acts as a "feature detector" or "filter" (and there are K such filters in above NN)

NN with one hidden layer and multiple outputs



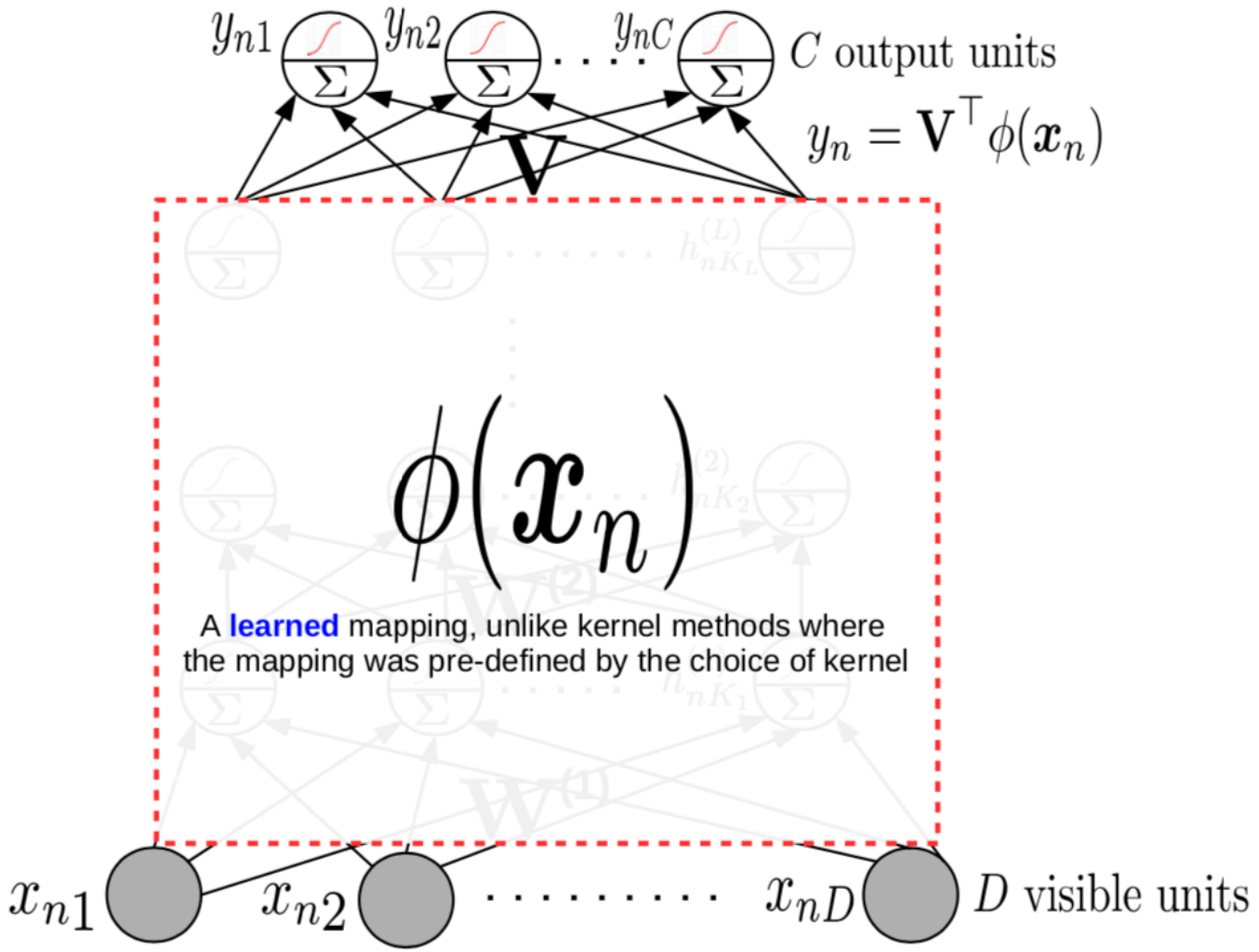
- Similar to multi-output regression or softmax regression on $\boldsymbol{h}_n$ as features

NN with multiple hidden layers and multiple outputs



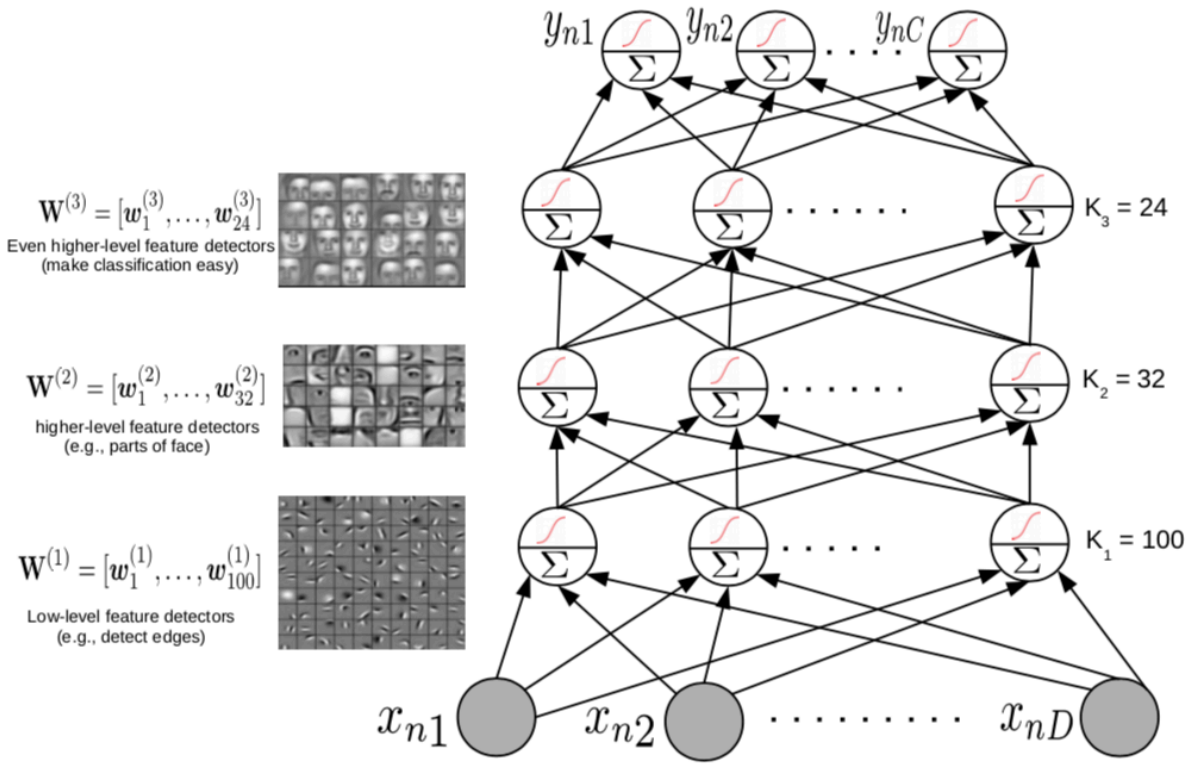
NN are feature learners!

❖ An NN (single/multiple hidden layers) tries to learn features that can predict the output well



Deep (Convolutional, especially) networks are good feature learners

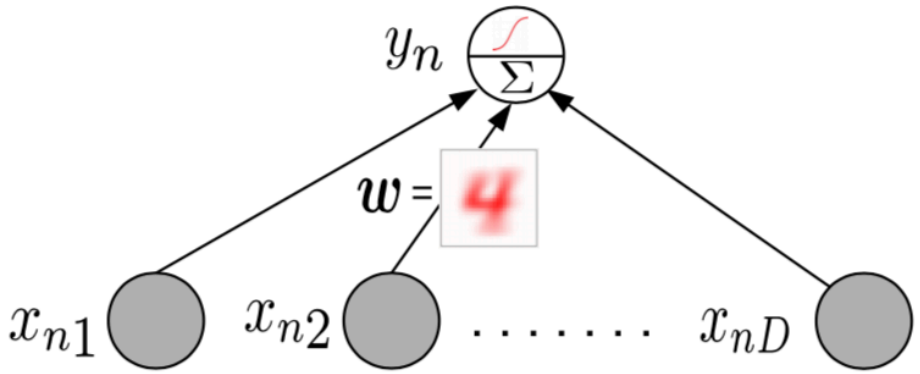
- Deep neural networks are good at detecting features at multiple layers of abstraction
- The connection weights between layers can be thought of as feature detectors or "filters"



- Lowest layer weights detect generic features, higher level weights detect more specific features
- Features learned in one layer are composed of features learned in the layer below

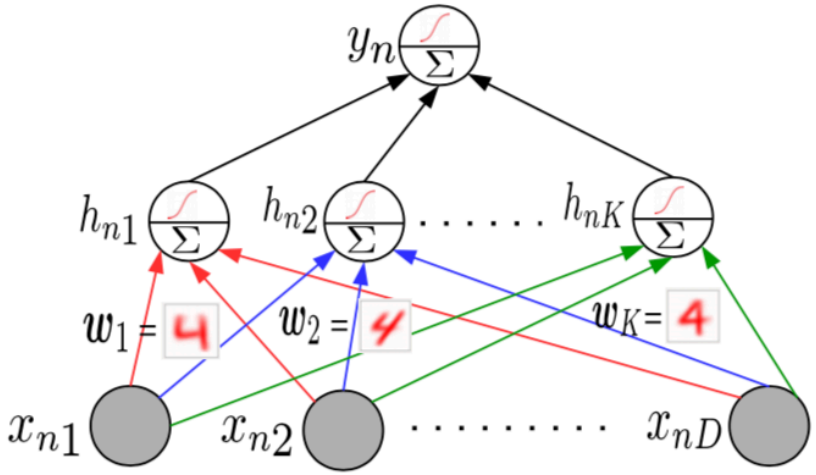
Why learning features with an MLP is *helpful*

- A single layer model will learn an "average" feature detector



- Can't capture subtle variations in the inputs

- An MLP can learn multiple feature detectors (even with a single hidden layer)



- Therefore even a single hidden layer helps capture subtle variations in the inputs

Expressive capabilities of NN: the good

Boolean functions:

- Every Boolean function can be represented by a network with a single hidden layer
- But might require exponential (in number of inputs) hidden units

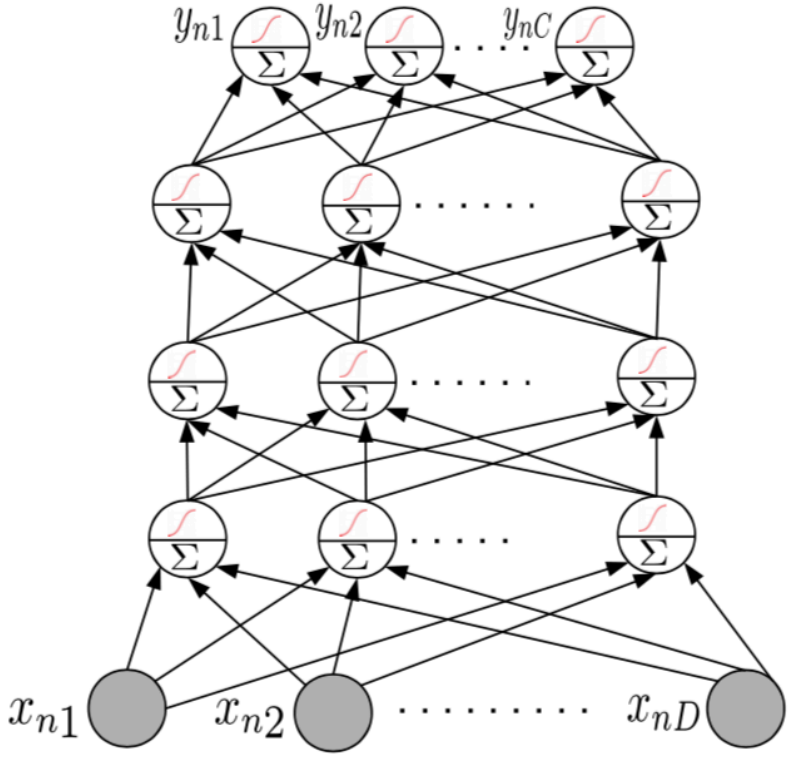
Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]

- Any function can be approximated to arbitrarily accuracy by a network with two hidden layers [Cybenko 1988]

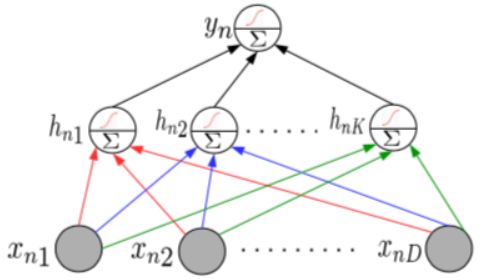
… the bad: design issues

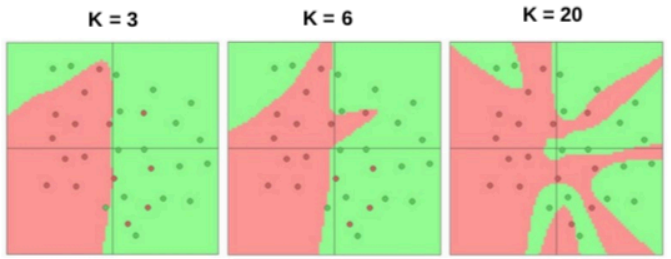
- Much of the magic lies in the hidden layer(s)
- As we've seen, hidden layers learn and detect good features
- However, we need to consider a few aspects
  - Number of hidden layers, number of units in each hidden layer
  - Why bother about many hidden layers and not use a single very wide hidden layer (recall Hornick's universal function approximator theorem)?
  - Complex network (several, very wide hidden layers) or simple network (few, moderately wide hidden layers)?
  - Aren't deep neural network prone to overfitting (since they contain a huge number of parameters)?



Representational power of hidden layers

- Consider an NN with a single hidden layer

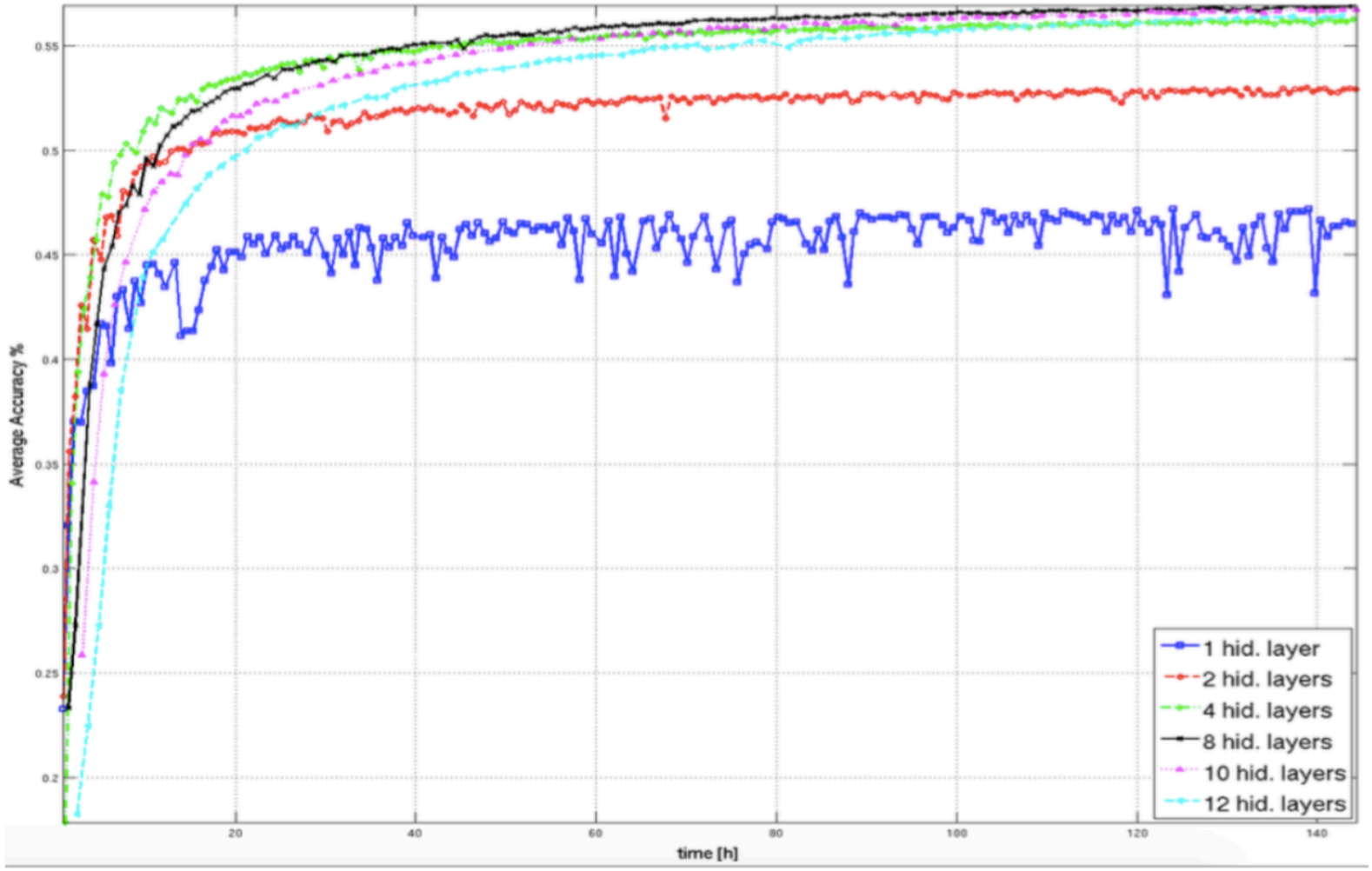




- Recall that each hidden unit "adds" a function to the overall function
- Increasing the number of hidden units will learn more and more complex function
- Very large K seems to overfit. Should we instead prefer small K?
- No! It is better to use large K and regularize well. Here is a reason/justification:
  - Simple NN with small K will have a few local optima, some of which may be bad
  - Complex NN with large K will have many local optimal, all equally good
  - Note: The above interesting behavior of NN has some theoretical justifications (won't discuss here)
- We can also use multiple hidden layers (each sufficiently large) and regularize well

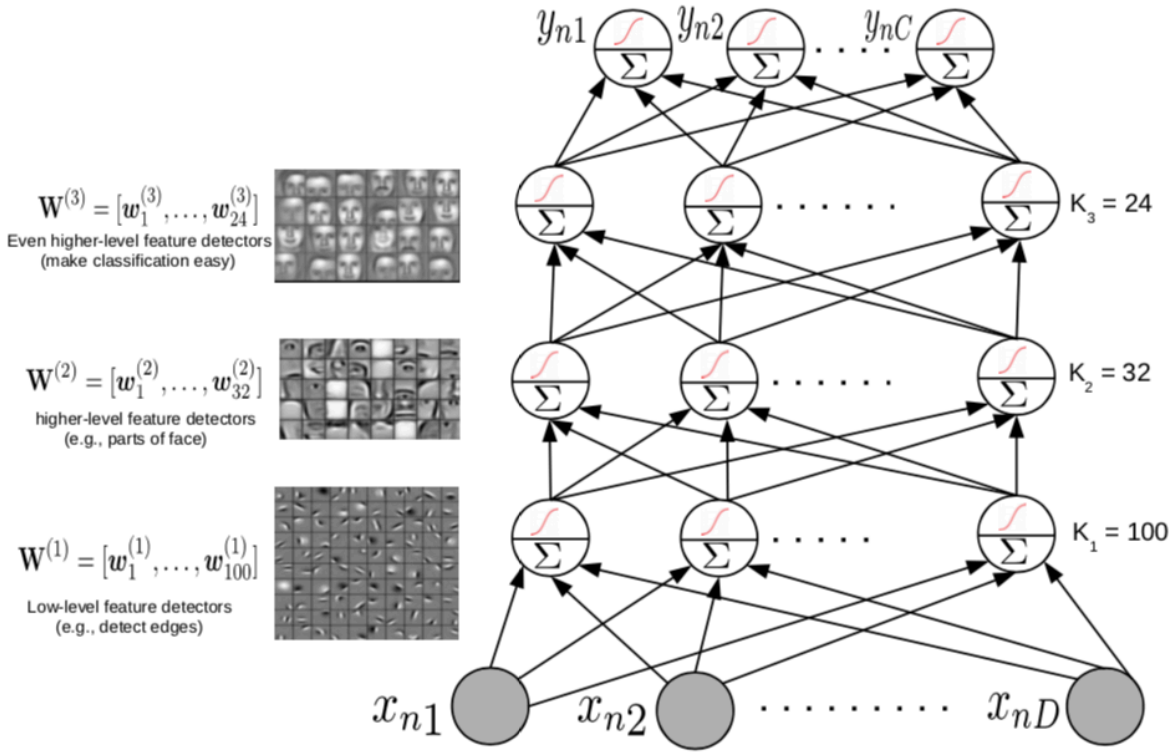
Effect of increasing the hidden layers

- Speech recognition task



Wide or deep?

- While very wide single hidden layer can approx. any function, often we prefer many hidden layers



- Higher layers help learn more directly useful/interpretable features (also useful for compressing data using a small number of features)

Wrap-up, Take-home messages

- Perceptron model for linear functions
- Perceptron algorithm for learning the weights (equivalent to SGD on squared loss function)
- Perceptrons and biological neural models
- Limitations of perceptron units
- From brain models to artificial neural networks for non-linear function approximation
- Structure of a unit, pre-activation and activation functions
- Multi-layer perceptrons, feed-forward networks
- Recurrent network models
- Hidden layers and feature aggregation and propagation, feature learning
- Weight matrices, number of parameters in a networks
- Theoretical properties of MLPs
- Design choices, overfitting, complexity