DATA MINING 2
Neural Networks

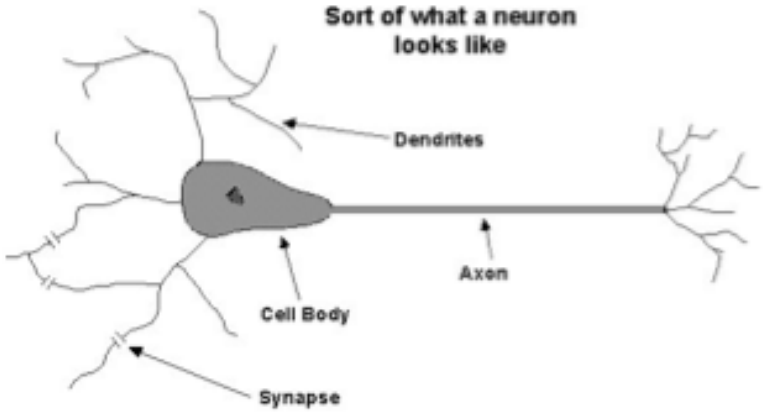
Riccardo Guidotti

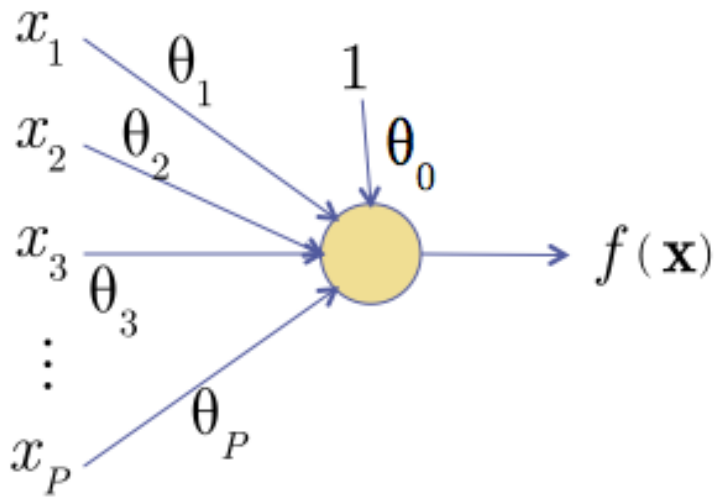
a.a. 2019/2020

UNIVERSITÀ DI PISA

The Neuron Metaphor

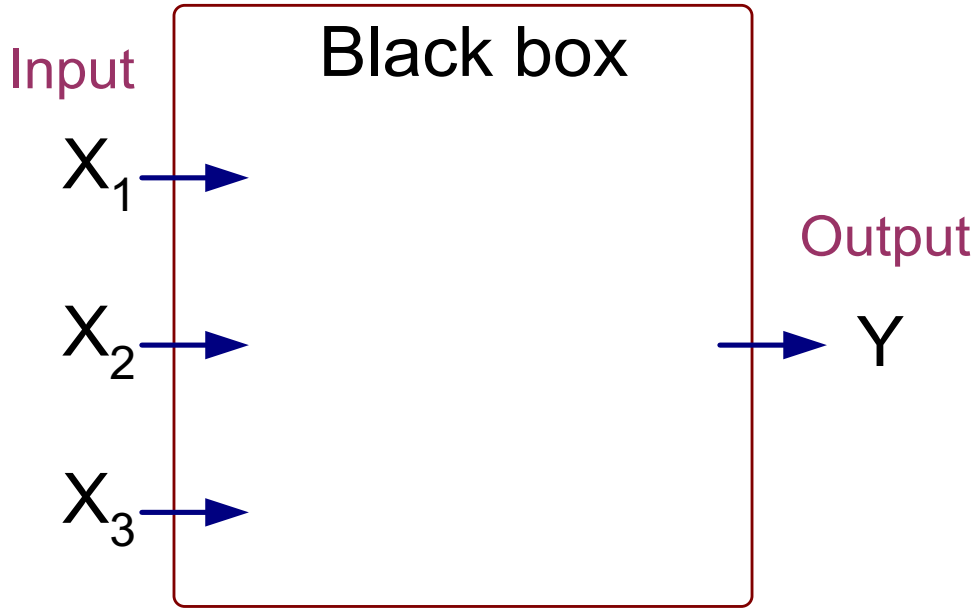
- Neurons
  - accept information from multiple inputs,
  - transmit information to other neurons.
- Multiply inputs by weights along edges
- Apply some function to the set of inputs at each node





Artificial Neural Networks (ANN)

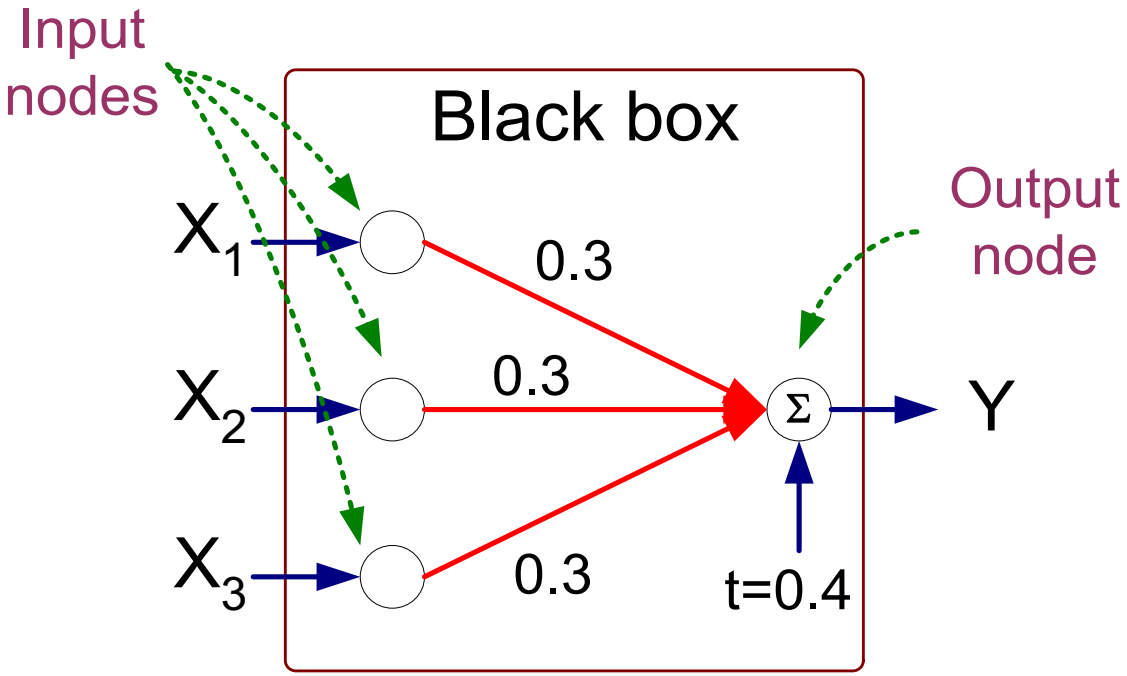
X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1



Output Y is 1 if at least two of the three inputs are equal to 1.

Artificial Neural Networks (ANN)

X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1

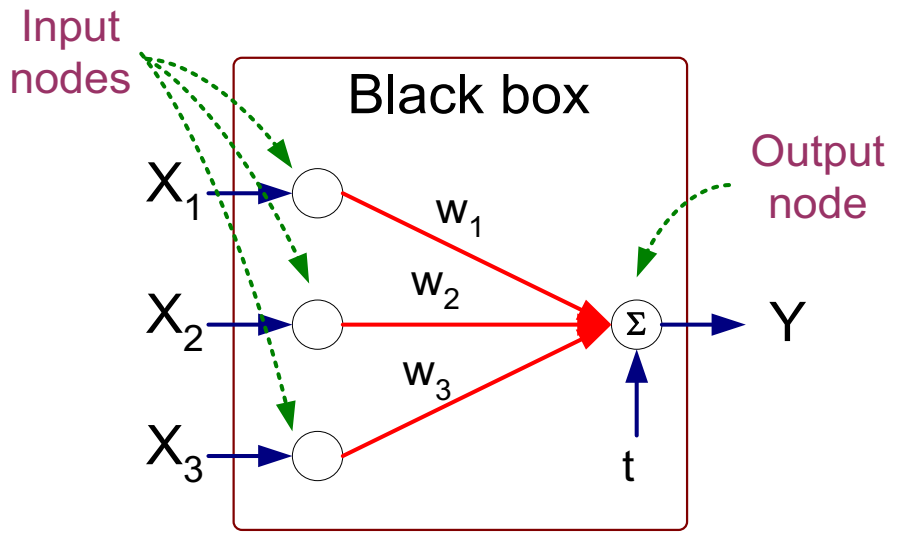


$$Y = sign(0.3X_1 + 0.3X_2 + 0.3X_3 - 0.4)$$

$$\text{where } sign(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Artificial Neural Networks (ANN)

- Model is an assembly of inter-connected nodes and weighted links
- Output node sums up each of its input value according to the weights of its links
- Compare output node against some threshold t (also named bias b)



$$Y = sign(\sum_{i=1}^{d} w_i X_i - t)$$

$$= sign(\sum_{i=0}^{d} w_i X_i)$$

Characterizing the Artificial Neuron

- Input/Output signal may be.
  - Real value.
  - Unipolar {0, 1}.
  - Bipolar {-1, +1}.
- **Weight** (w or sigma): ϑ_{ij} – strength of connection from unit *j* to unit *i*
- Learning amounts to **adjusting the weights** ϑ_{ij} by means of an **optimization algorithm** aiming to minimize a cost function, i.e., as in biological systems training a perceptron model amounts to adapting the weights of the links until they fit the input output relationships of the underlying data.

Characterizing the Artificial Neuron

- The bias *b* is a constant that can be written as $\vartheta_{i0}x_0$ with $x_0 = 1$ and $\vartheta_{i0} = b$ such that

$$net_i = \sum_{j=0}^{n} \vartheta_{ij} x_j$$

- The function $f(net_i(x))$ is the unit's **activation function**. In the simplest case, *f* is the identity function, and the unit's output is just its net input. This is called a ***linear unit.*** Otherwise we can have a ***sign unit,*** or a ***logistic unit.***

The Perceptron Classifier

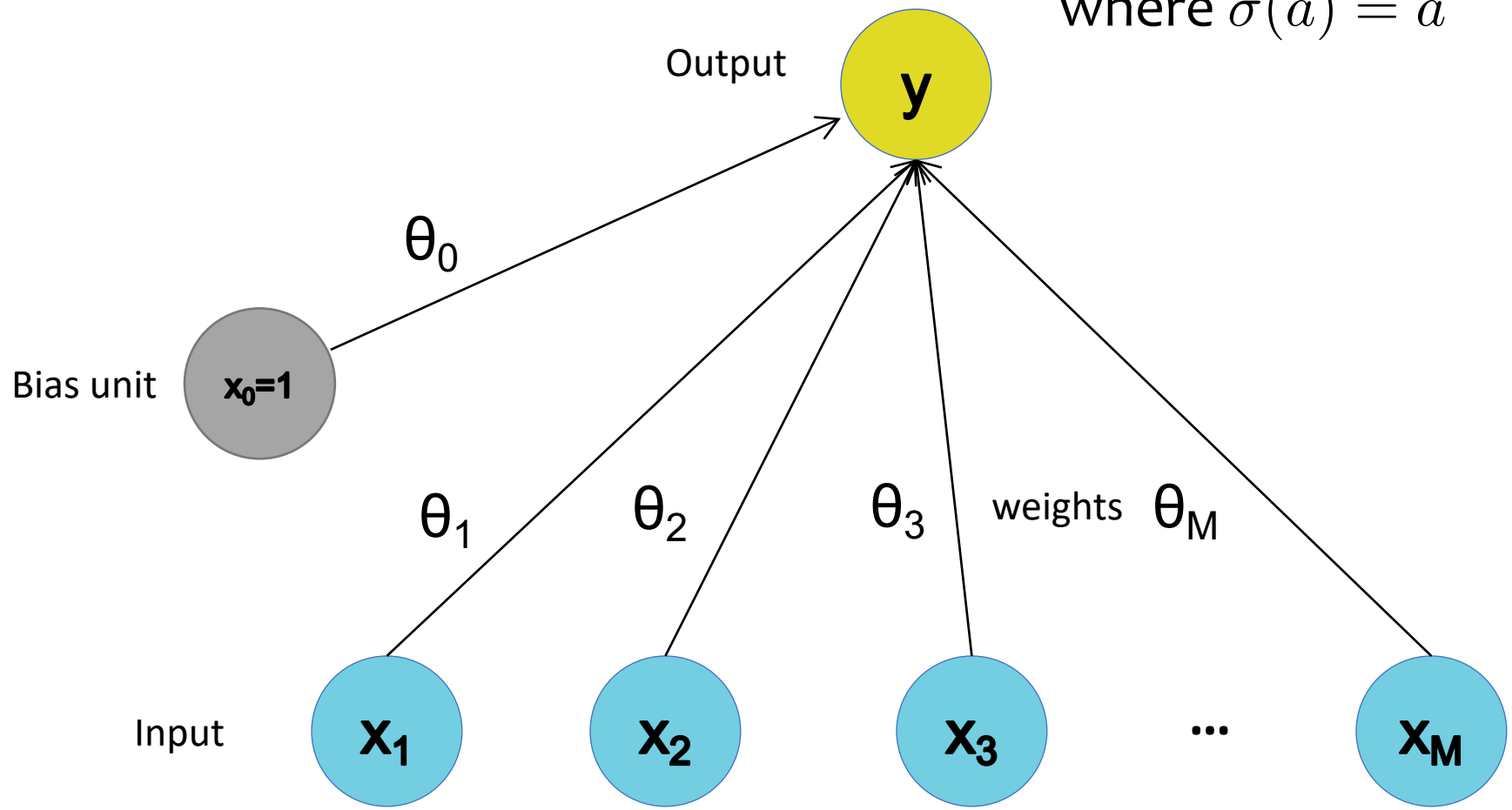
A Simple Linear Neuron

$$y = h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(\boldsymbol{\theta}^T \boldsymbol{x})$$

where $\sigma(a) = a$

net

Linear activation function



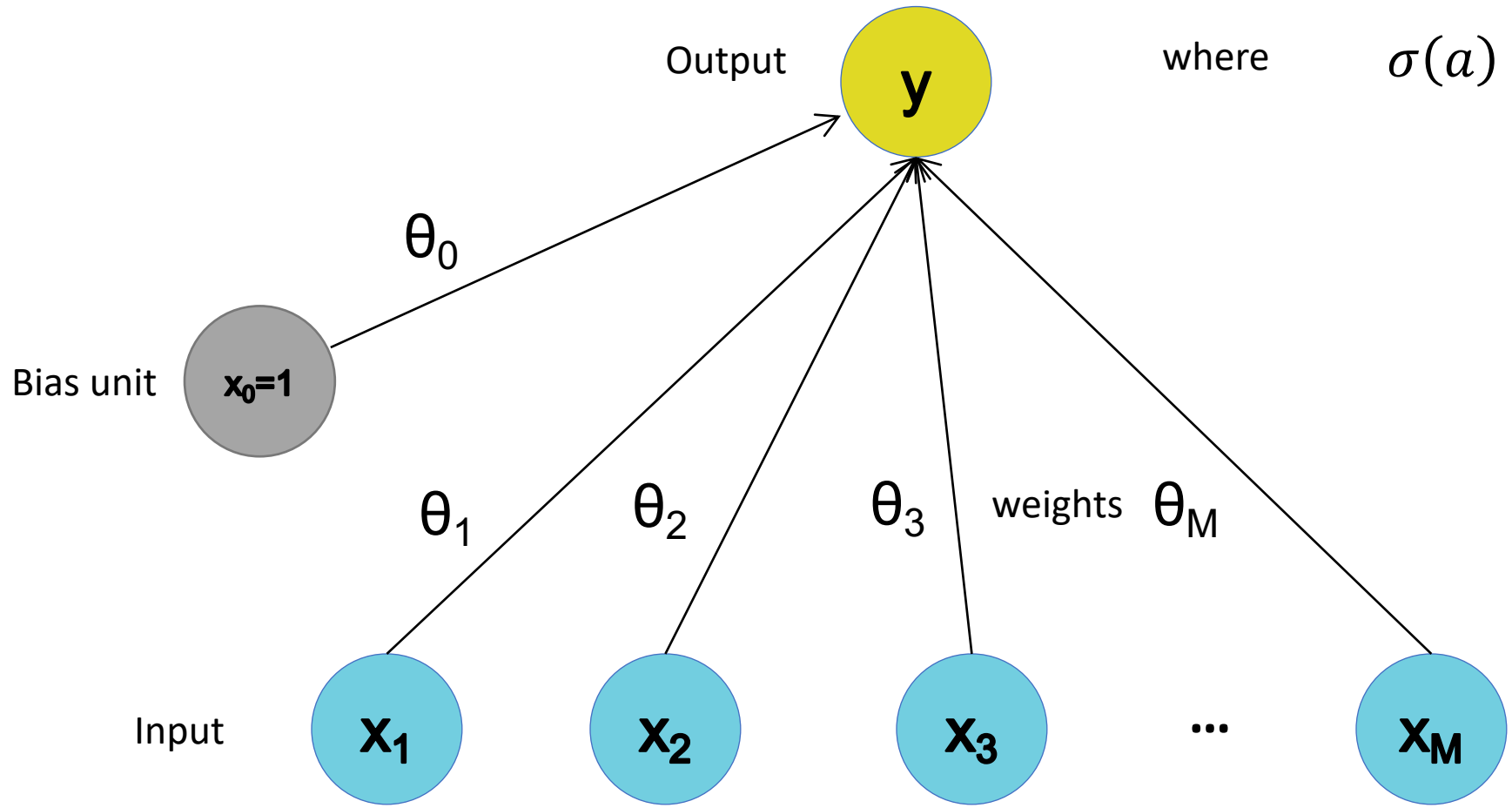
Linear Threshold Unit (a.k.a. Perceptron)

$$y = h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(\boldsymbol{\theta}^T \boldsymbol{x})$$

net

Sign activation function

where $\sigma(a) = \begin{cases} +1 \; a \geq 0 \\ -1 \; a < 0 \end{cases}$



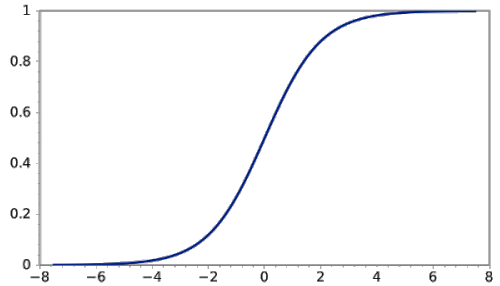
The Logistic Neuron

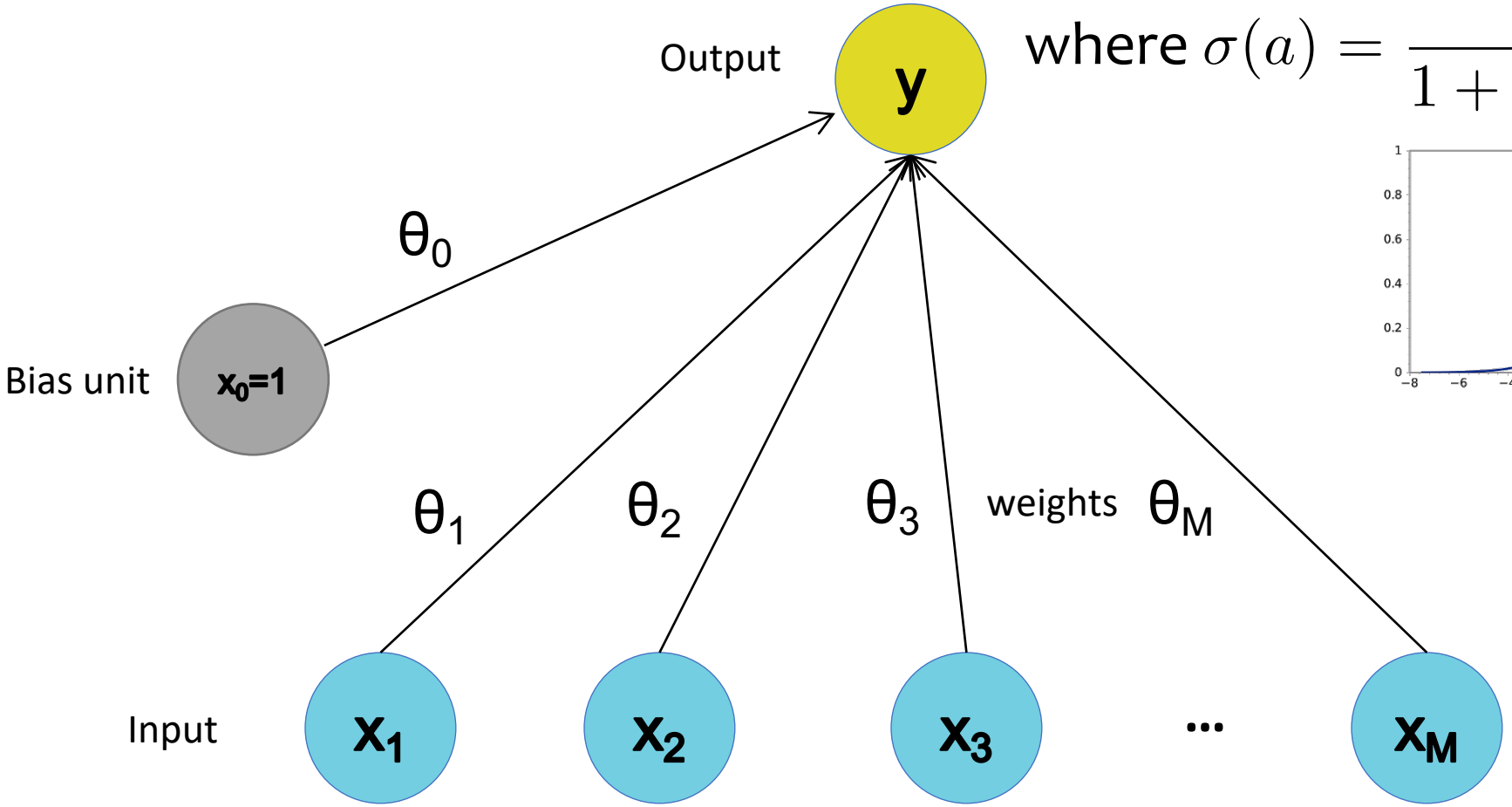
$$y = h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(\boldsymbol{\theta}^T \boldsymbol{x})$$

net

Sigmoid logistic activation function

where $\sigma(a) = \dfrac{1}{1 + \exp(-a)}$





Perceptron

- Single layer network
  - Contains only input and output nodes
- Activation function: f = sign(w•x)
- Applying model is straightforward

$$Y = sign(0.3X_1 + 0.3X_2 + 0.3X_3 - 0.4)$$

$$\text{where } sign(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

  - $X_1 = 1, X_2 = 0, X_3 = 1 \Rightarrow y = sign(0.2) = 1$

Learning Iterative Procedure

- During the training phase the weight parameters are adjusted until the outputs of the perceptron become consistent with the true outputs of the training examples.
- Initialize the weights (w_0, w_1, …, w_m)
- Repeat
  - For each training example (x_i, y_i)
    - Compute $f(w^{(k)}, x_i)$
    - Update the weights: $w^{(k+1)} = w^{(k)} + \lambda\left[y_i - f(w^{(k)}, x_i)\right]x_i$
- Until stopping condition is met

Iteration index

Learning rate

Perceptron Learning Rule

- Weight update formula:

$$w^{(k+1)} = w^{(k)} + \lambda\left[y_i - f(w^{(k)}, x_i)\right]x_i \;\; ; \;\; \lambda : \text{learning rate}$$

- Intuition:
  - Update weight based on error: $e = \left[y_i - f(w^{(k)}, x_i)\right]$
  - If y=f(x,w), e=0: no update needed
  - If y>f(x,w), e=2: weight must be increased so that f(x,w) will increase
  - If y<f(x,w), e=-2: weight must be decreased so that f(x,w) will decrease

The Learning Rate

- Is a parameter with value between 0 and 1 used to control the amount of adjustment made in each iteration.
- If is close to 0 the new weight is mostly influenced by the value of the old weight.
- If it is close to 1, then the new weight is mostly influenced by the current adjustment.
- The learning rate can be adaptive: initially moderately large and the gradually decreases in subsequent iterations.

Example of Perceptron Learning

$$w^{(k+1)} = w^{(k)} + \lambda\left[y_i - f(w^{(k)}, x_i)\right]x_i$$

$$Y = sign(\sum_{i=0}^{d} w_i X_i)$$

$$\lambda = 0.1$$

X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1

	w_0	w_1	w_2	w_3
0	0	0	0	0
1	-0.2	-0.2	0	0
2	0	0	0	0.2
3	0	0	0	0.2
4	0	0	0	0.2
5	-0.2	0	0	0
6	-0.2	0	0	0
7	0	0	0.2	0.2
8	-0.2	0	0.2	0.2

Epoch	w_0	w_1	w_2	w_3
0	0	0	0	0
1	-0.2	0	0.2	0.2
2	-0.2	0	0.4	0.2
3	-0.4	0	0.4	0.2
4	-0.4	0.2	0.4	0.4
5	-0.6	0.2	0.4	0.2
6	-0.6	0.4	0.4	0.2

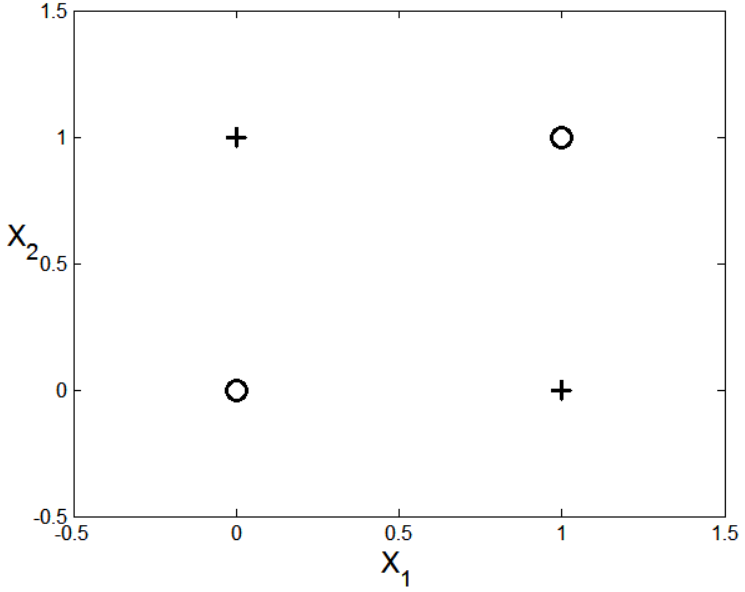
Nonlinearly Separable Data

- Since *f(w,x)* is a linear combination of input variables, decision boundary is linear.
- For nonlinearly separable problems, the perceptron fails because no linear hyperplane can separate the data perfectly.
- An example of nonlinearly separable data is the XOR function.

XOR Data

x_1	x_2	y
0	0	-1
1	0	1
0	1	1
1	1	-1

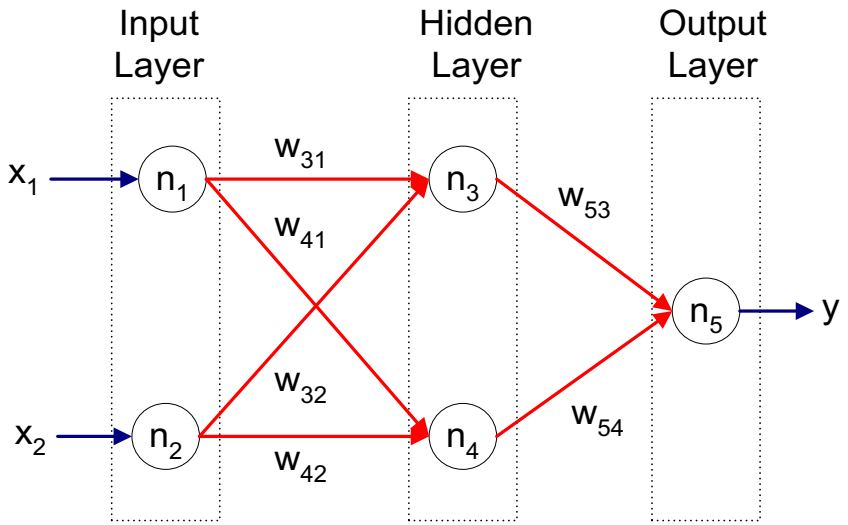
$$y = x_1 \oplus x_2$$



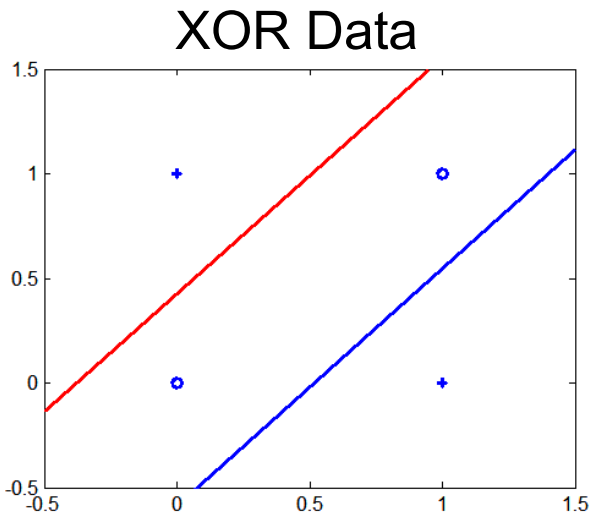
Multilayer Neural Network

Multilayer Neural Network

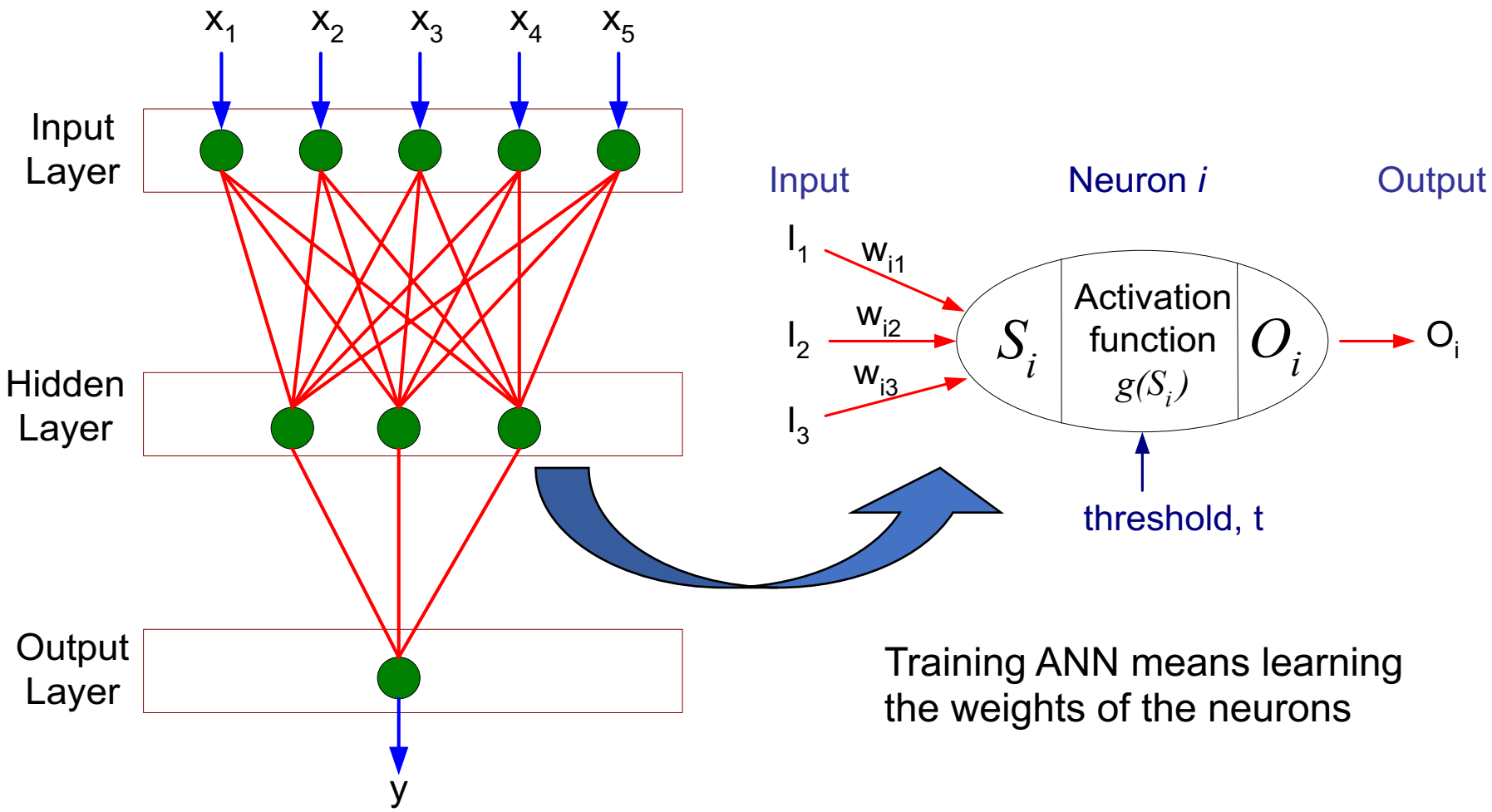
- **Hidden Layers**: intermediary layers between input and output layers.
- More general **activation functions** (sigmoid, linear, hyperbolic tangent, etc.).



- Multi-layer neural network can solve any type of classification task involving nonlinear decision surfaces.
- Perceptron is single layer.
- We can think to each hidden node as a perceptron that tries to construct one hyperplane, while the output node combines the results to return the decision boundary.



General Structure of ANN



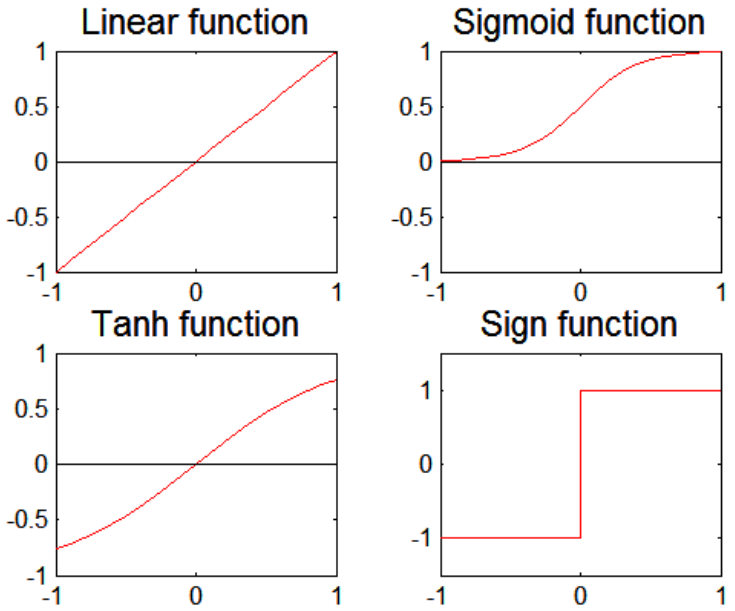
Training ANN means learning the weights of the neurons

Artificial Neural Networks (ANN)

- Various types of neural network topology
  - single-layered network (perceptron) versus multi-layered network
  - Feed-forward versus recurrent network

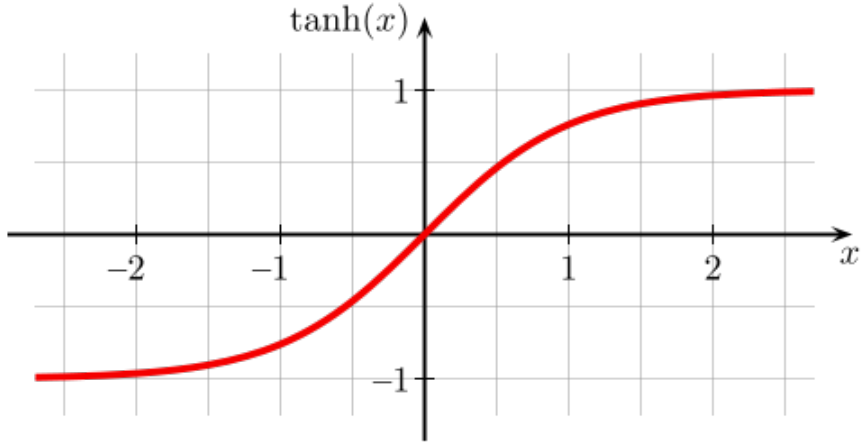
- Various types of activation functions (f)

$$Y = f(\sum_i w_i X_i)$$



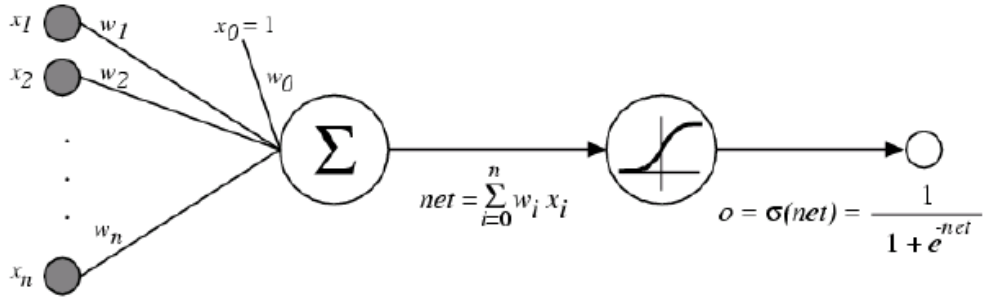
Activation Functions

- A new change: modifying the nonlinearity
  - The logistic is not widely used in modern ANNs

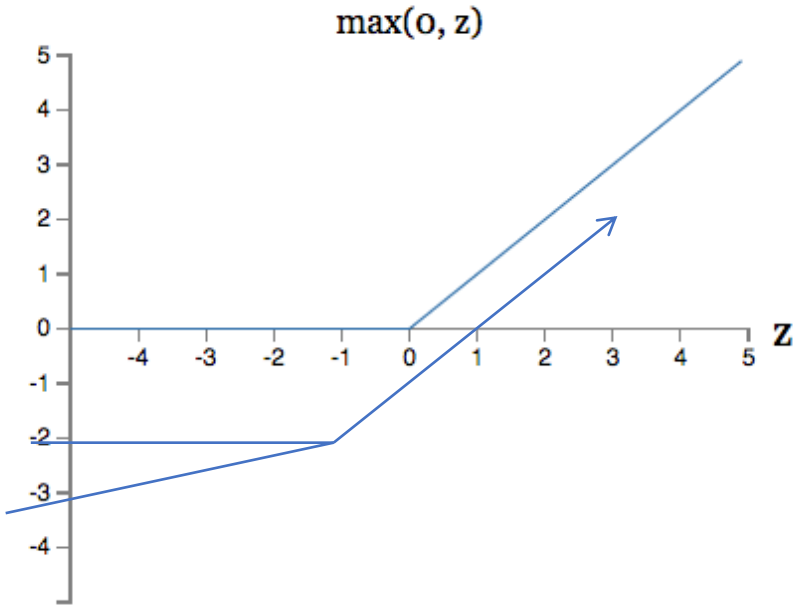


Alternative 1:
tanh

Like logistic function but shifted to range [-1, +1]



Activation Functions



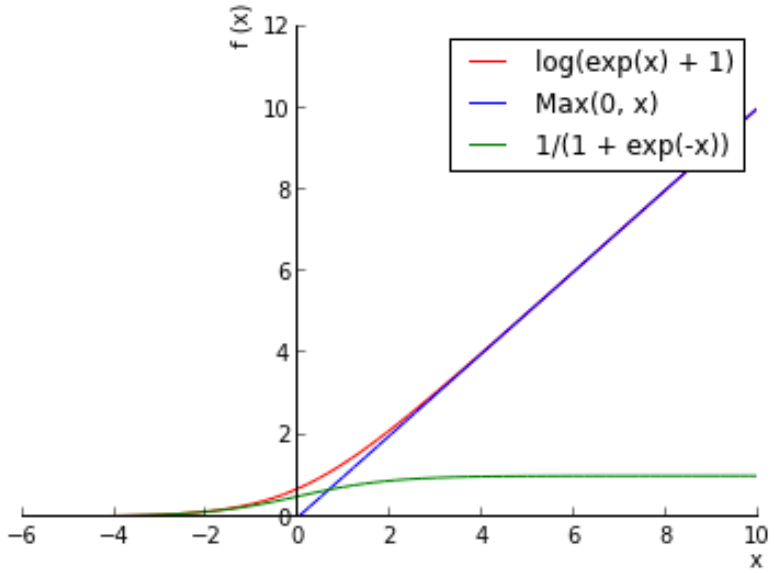
Alternative 2: rectified linear unit

Linear with a cutoff at zero

(Implementation: clip the gradient when you pass zero)

$$\max(0, w \cdot x + b).$$

Activation Functions



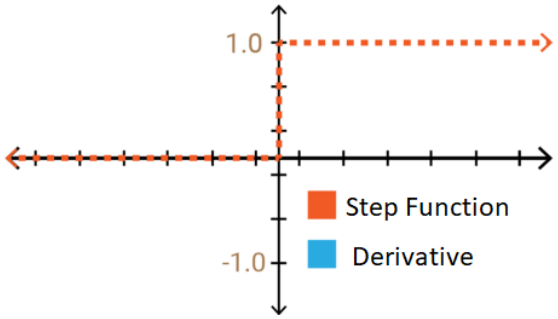
Alternative 3: soft exponential linear unit

Soft version: log(exp(x)+1)

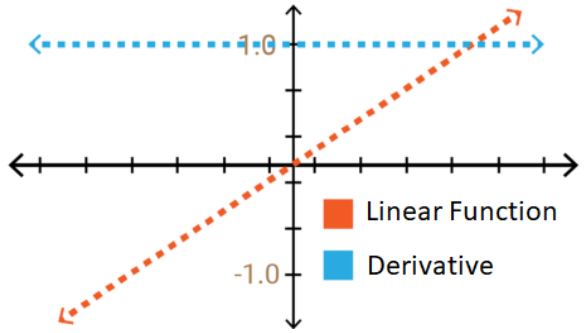
Doesn’t saturate (at one end)
Sparsifies outputs
Helps with vanishing gradient

Slide from William Cohen

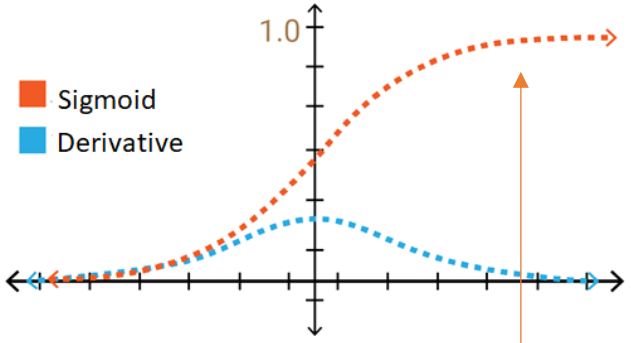
Activation Functions Summary



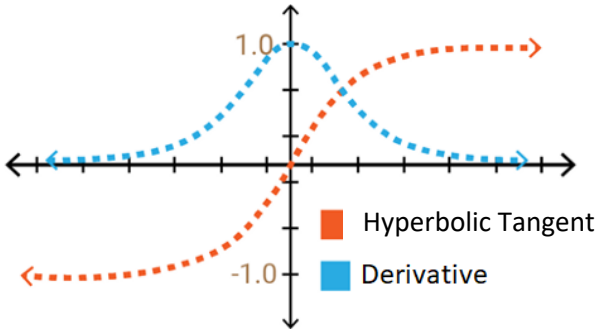
$$f(x) = \begin{cases} 0 \; for \; x < 0 \\ 1 \; for \; x \geq 0 \end{cases}$$



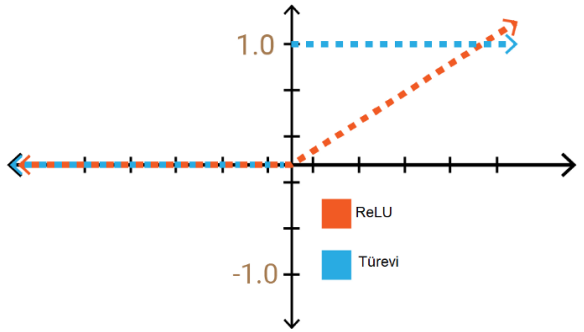
$$f(x) = x$$



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



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



$$f(x) = \begin{cases} 0 \; (or \; \epsilon) \; for \; x < 0 \\ x \qquad\qquad for \; x \geq 0 \end{cases}$$

$$f(x_j) = \frac{e^{x_j}}{\sum_k e^{x_k}}$$

Softmax Function

Learning Multi-layer Neural Network

- Can we apply perceptron learning to each node, including hidden nodes?
- Perceptron computes error $e = y - f(w,x)$ and updates weights accordingly
- Problem: how to determine the true value of y for hidden nodes?
- Approximate error in hidden nodes by error in the output nodes
- Problems:
  - Not clear how adjustment in the hidden nodes affect overall error
  - No guarantee of convergence to optimal solution

Gradient Descent for Multilayer NN

- Error function to minimize: $E = \frac{1}{2}\sum_{i=1}^{N}\left(y_i - f(\sum_j w_j x_{ij})\right)$

Quadratic function from which we can find a global minimum solution

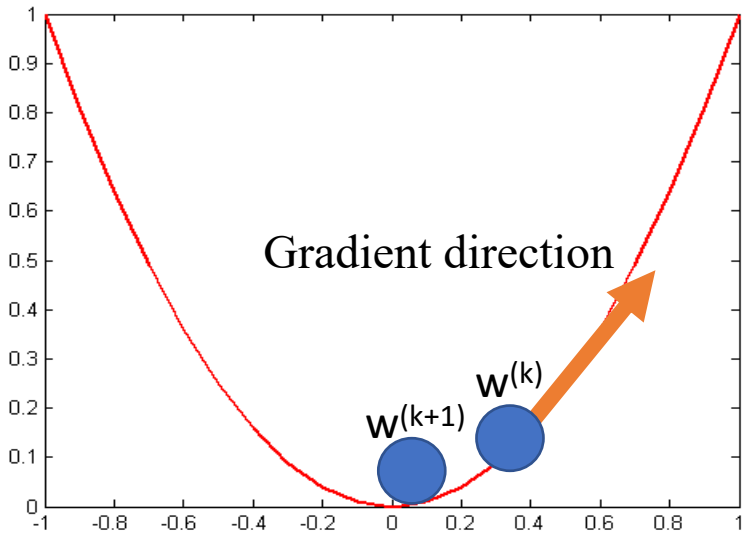
- Weight update: $w_j^{(k+1)} = w_j^{(k)} - \lambda \frac{\partial E}{\partial w_j}$
- Activation function *f* must be differentiable
- For sigmoid function: $w_j^{(k+1)} = w_j^{(k)} + \lambda \sum_i (y_i - o_i) o_i (1 - o_i) x_{ij}$
- Stochastic Gradient Descent (update the weight immediately)

Gradient Descent for Multilayer NN

- Weights are updated in the opposite direction of the gradient of the loss function.

$$w_j^{(k+1)} = w_j^{(k)} - \lambda \frac{\partial E}{\partial w_j}$$

- Gradient direction is the direction of uphill of the error function.
- By taking the negative we are going downhill.
- Hopefully to a minimum of the error.



Gradient Descent for Multilayer NN

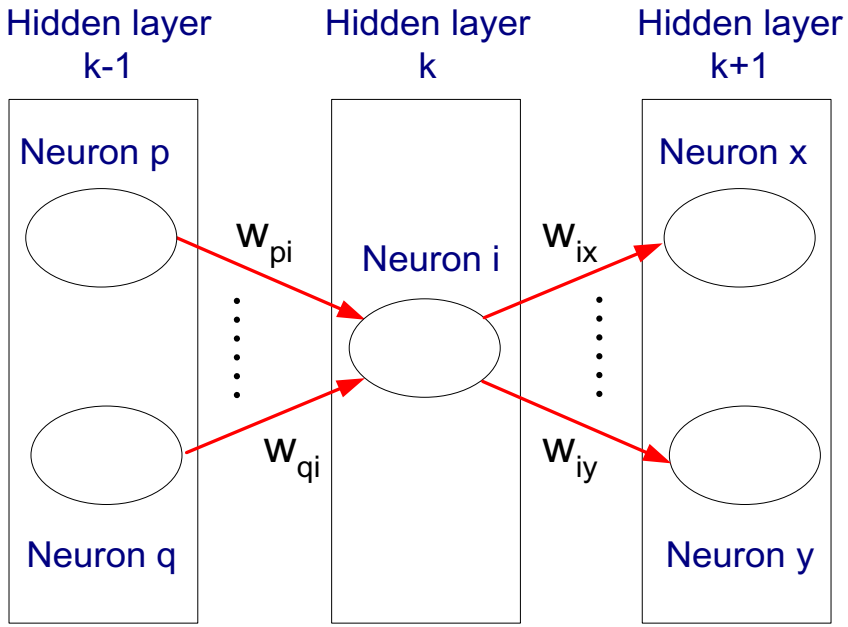
- For output neurons, weight update formula is the same as before (gradient descent for perceptron)

- For hidden neurons:

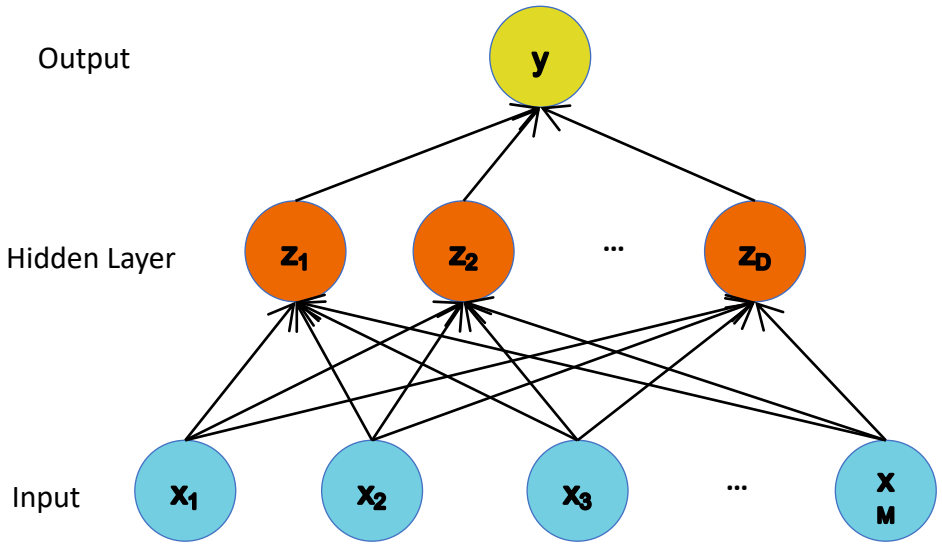
$$w_{pi}^{(k+1)} = w_{pi}^{(k)} + \lambda o_i (1 - o_i) \sum_{j \in \Phi_i} \delta_j w_{ij} x_{pi}$$

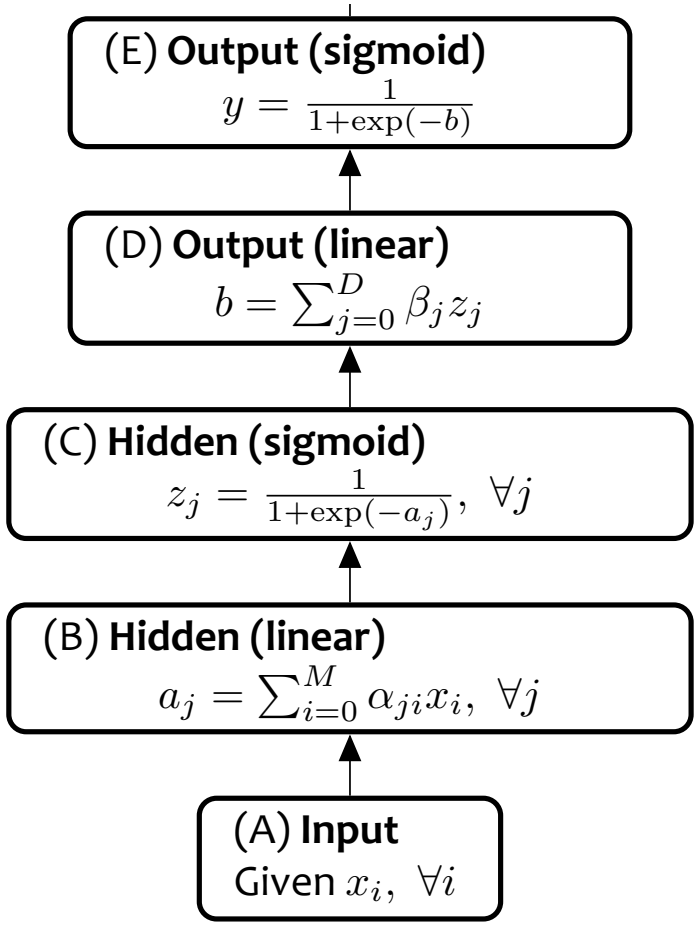
$$\text{Output neurons} : \delta_j = o_j (1 - o_j)(t_j - o_j)$$

$$\text{Hidden neurons} : \delta_j = o_j (1 - o_j) \sum_{k \in \Phi_j} \delta_k w_{jk}$$

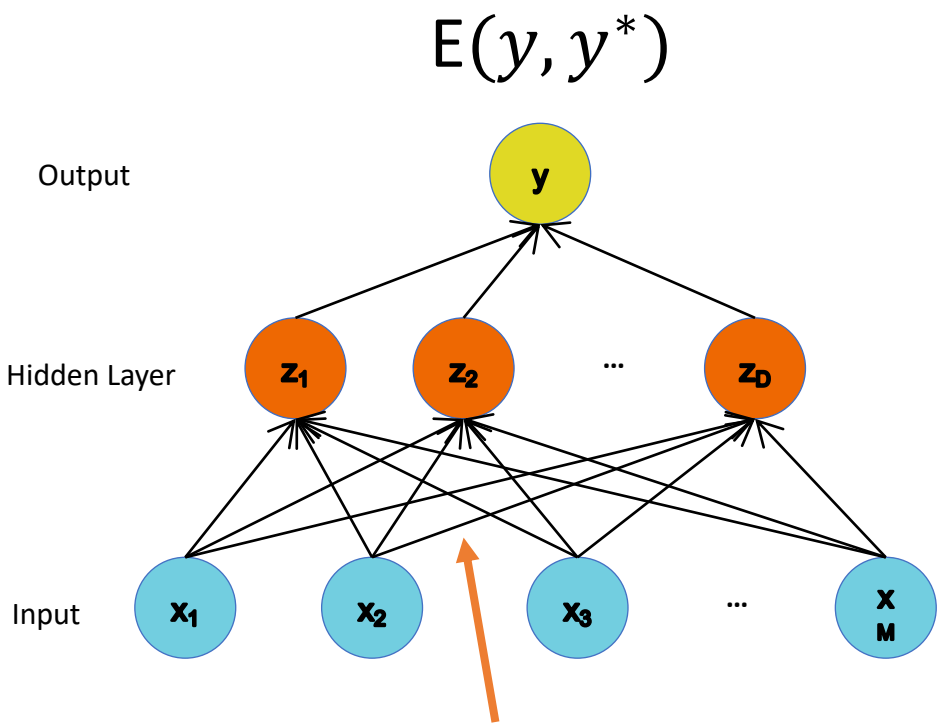


Training Multilayer NN

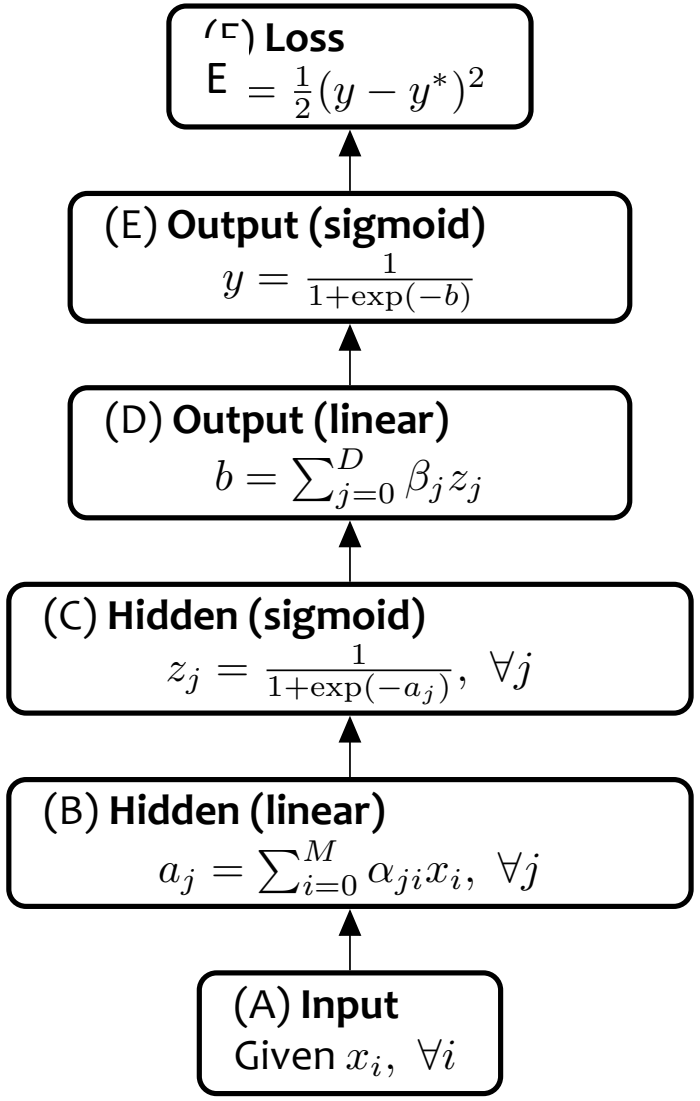




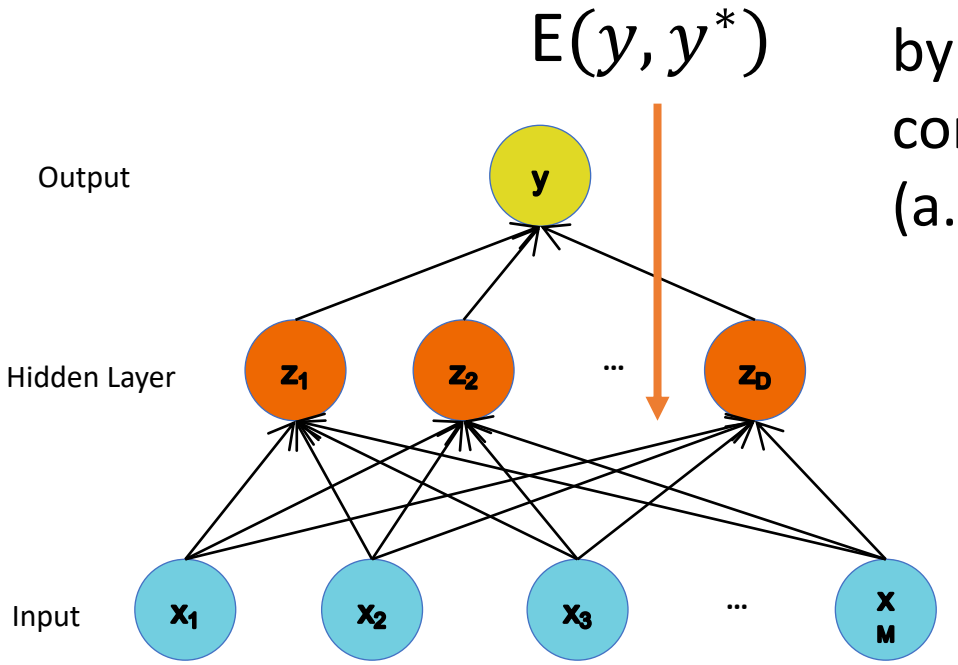
Training Multilayer NN



How do we update these weights given the loss is available only at the output unit?



Error Backpropagation



Error is computed at the output and propagated back to the input by chain rule to compute the contribution of each weight (a.k.a. derivative) to the loss

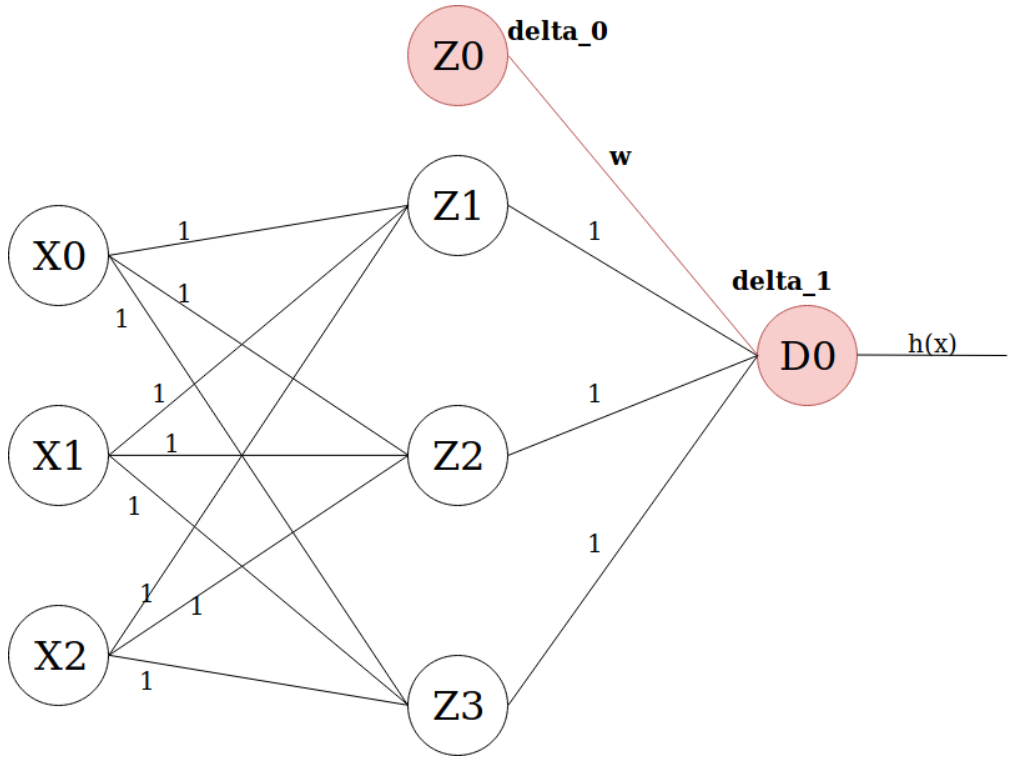
A 2-step process

1. Forward pass - Compute the network output
2. Backward pass – Compute the loss function gradients and update

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Backpropagation in other words

- In order to get the loss of a node (e.g. Z0), we multiply the value of its corresponding f’(z) by the loss of the node it is connected to in the next layer (delta_1), by the weight of the link connecting both nodes.
- We do the delta calculation step at every unit, back-propagating the loss into the neural net, and finding out what loss every node/unit is responsible for.



https://towardsdatascience.com/how-does-back-propagation-in-artificial-neural-networks-work-c7cad873ea7

On the Key Importance of Error Functions

- The error/loss/cost function reduces all the various good and bad aspects of a possibly complex system down to a single number, a scalar value, which allows candidate solutions to be compared.
- It is important, therefore, that **the function faithfully represent our design goals**.
- If we choose a poor error function and obtain unsatisfactory results, the fault is ours for badly specifying the goal of the search.

Objective Functions for NN

- Regression: A problem where you predict a real-value quantity.
  - Output Layer: One node with a linear activation unit.
  - Loss Function: Quadratic Loss (Mean Squared Error (MSE))
- Classification: Classify an example as belonging to one of K classes
  - Output Layer:
    - One node with a sigmoid activation unit (K=2)
    - K output nodes in a softmax layer (K>2)
  - Loss function: Cross-entropy (i.e. negative log likelihood)

J = E	Forward	Backward
Quadratic	$J = \frac{1}{2}(y - y^*)^2$	$\frac{dJ}{dy} = y - y^*$
Cross Entropy	$J = y^* \log(y) + (1 - y^*) \log(1 - y)$	$\frac{dJ}{dy} = y^* \frac{1}{y} + (1 - y^*) \frac{1}{y - 1}$

Design Issues in ANN

- Number of nodes in input layer
  - One input node per binary/continuous attribute
  - k or $log_2 k$ nodes for each categorical attribute with k values
- Number of nodes in output layer
  - One output for binary class problem
  - k or $log_2 k$ nodes for k-class problem
- Number of nodes in hidden layer
- Initial weights and biases

Characteristics of ANN

- Multilayer ANN are universal approximators but could suffer from ***overfitting*** if the network is too large.
- Gradient descent may converge to ***local minimum***.
- Model building can be very time consuming, but testing can be very fast.
- Can handle redundant attributes because weights are automatically learnt.
- Sensitive to noise in training data.
- Difficult to handle missing attributes.

Tips and Tricks of NN Training

Dataset Should Normally be Split Into

- ***Training set:*** use to update the weights. Records in this set are repeatedly in random order. The weight update equation are applied after a certain number of records.

- ***Validation set:*** use to decide when to stop training only by monitoring the error and to select the best model configuration

- ***Test set:*** use to test the performance of the neural network. It should not be used as part of the neural network development and model selection cycle

Before Starting: Weight Initialization

- Choice of ***initial weight values is important as this decides starting position in weight space.*** That is, how far away from global minimum
  - Aim is to select weight values which produce midrange function signals
  - Select weight values randomly from uniform probability distribution
  - Normalize weight values so number of weighted connections per unit produces midrange function signal

- Try different random initialization to
  - Assess robustness
  - Have more opportunities to find optimal results

Two learning fashion (plus one)

- **Sequential mode** (on-line, stochastic, or per-pattern)
  - Weights updated after each records is presented
  - Many weight updates, can quicker convergence but also make learning less stable

- **Batch mode** (off-line or per-epoch)
  - Weights updated after all records are presented
  - Can be very slow and lead to trapping in early local minima

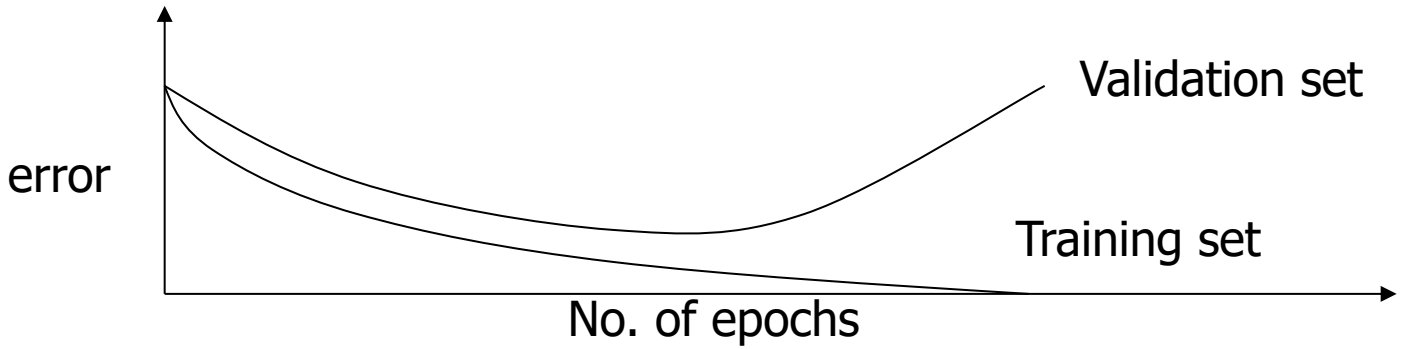
- **Minibatch mode** (a blend of the two above)
  - Weights updated after a few records (from tens to thousands) are presented
  - Best of both (and good for GPU)

Convergence Criteria

- Learning is obtained by repeatedly supplying training data and adjusting by backpropagation
  - Typically 1 training set presentation = **1 epoch**
- We need a stopping criteria to define convergence
  - Euclidean norm of the gradient vector reaches a sufficiently small value
  - Absolute rate of change in the average squared error per epoch is sufficiently small
  - **Validation for generalization performance: stop when generalization performance reaches a peak**

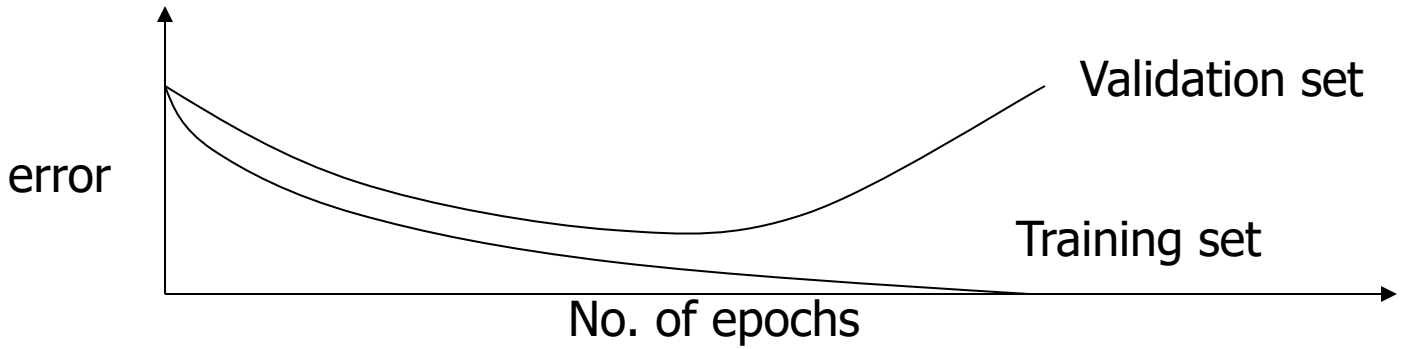
Early Stopping

- Running too many epochs may **overtrain** the network and result in **overfitting** and perform poorly in generalization
- Keep a hold-out validation set and test accuracy after every epoch. Maintain weights for best performing network on the validation set and stop training when error increases beyond this
- Always let the network run for some epochs before deciding to stop (**patience parameter**), then backtrack to best result



Model Selection

- **Too few hidden units** prevent the network from learning adequately fitting the data and learning the concept.
- **Too many hidden units** leads to overfitting, unless you regularize heavily (e.g. dropout, weight decay, weight penalties)
- Cross validation should be used to determine an appropriate number of hidden units by using the optimal validation error to select the model with optimal number of hidden layers and nodes.



Regularization

- Constrain the learning model to avoid overfitting and help improving generalization.
- Add **penalization terms** to the loss function that *punish* the model for excessive use of resources
  - Limit the **amount of weights** that is used to learn a task
  - Limit the **total activation of neurons** in the network

$$E' = E(y, y^*) + \lambda R(\cdot)$$

Hyperparameter to be chosen in model selection

$R(W_\theta)$ Penalty on parameters

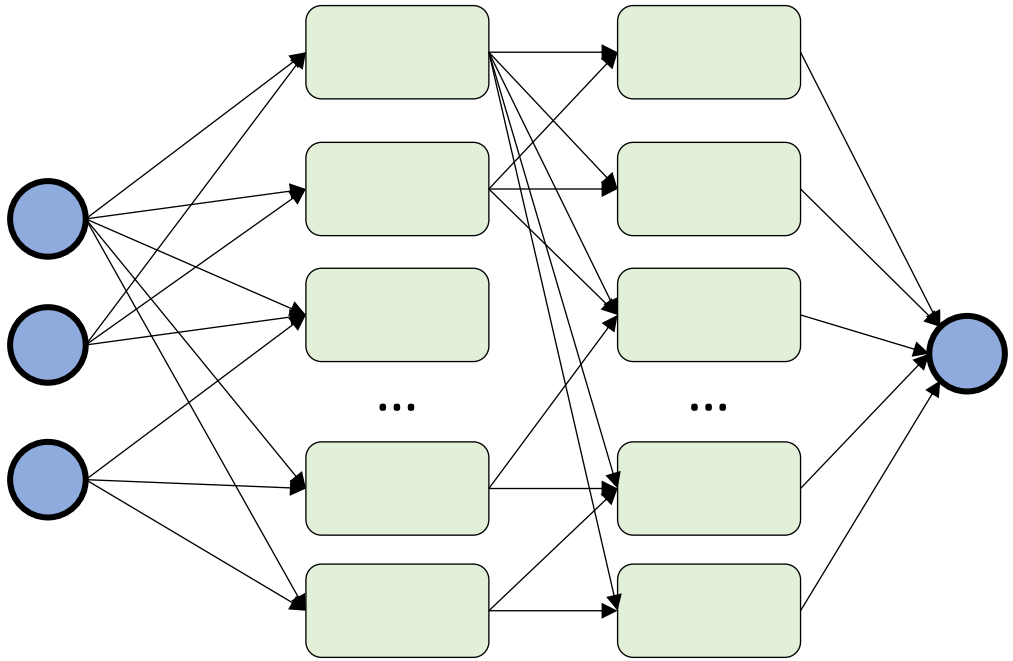
$R(Z)$ Penalty on activations

Common penalty terms (norms)

- 1-norm $||A||_1 = \sum_{ij} |a_{ij}|$
  - Parameters: $R(W_\theta) = ||W_\theta||_1^2$
  - Activations: $R(Z(X)) = ||Z(X)||_1^2$ (Z hidden unit activation)
- 2-norm $||A||_2 = \sqrt{\sum_{ij} a_{ij}^2}$
  - Parameters: $R(W_\theta) = ||W_\theta||_2^2$
  - Activations: $R(Z(X)) = ||Z(X)||_2^2$ (Z hidden unit activation)
- Any p-norm and more…

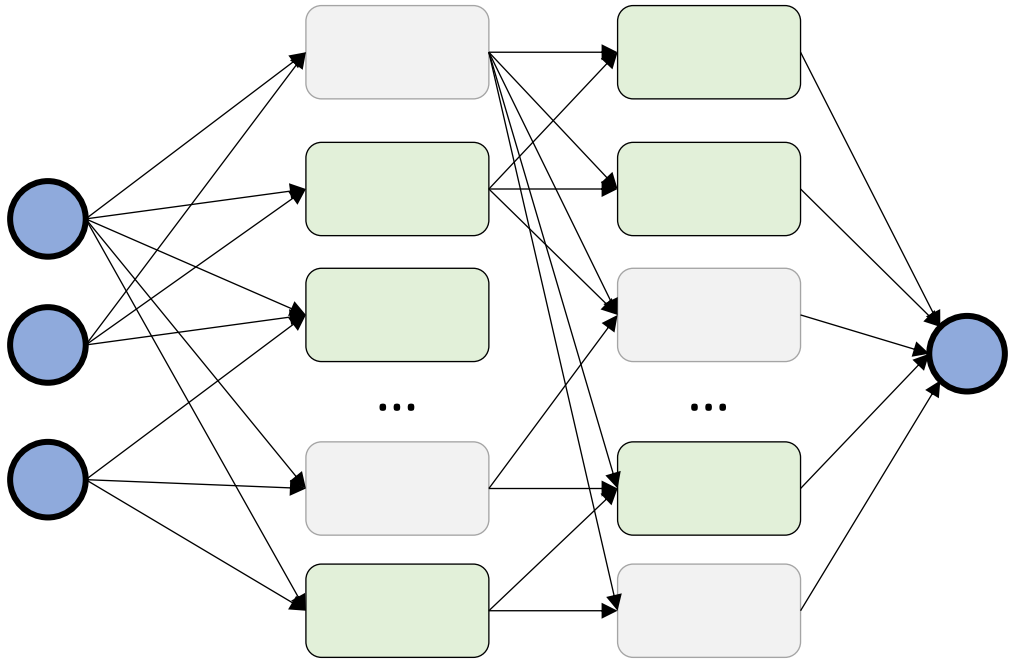
Dropout Regularization

Randomly disconnect units from the network during training



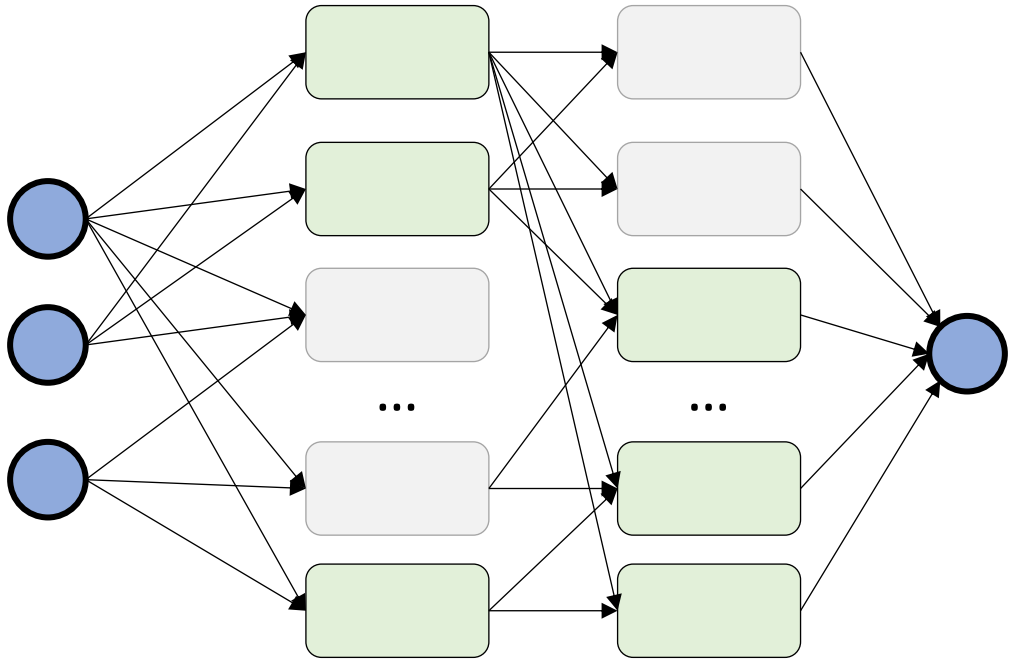
Dropout Regularization

Randomly disconnect units from the network during training



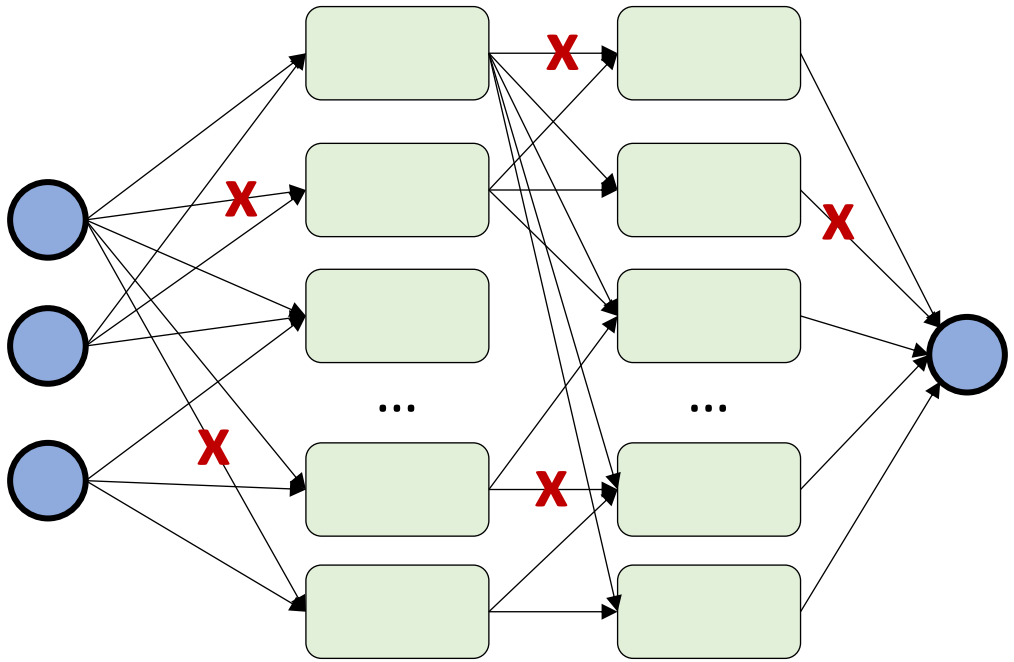
Dropout Regularization

Randomly disconnect units from the network during training



Dropout Regularization

Randomly disconnect units from the network during training



- Regulated by unit **dropping hyperparameter**
- Prevents unit **coadaptation**
- Committee machine effect
- Need to adapt **prediction phase**
- Used at prediction time gives **predictions with confidence intervals**

You can also **drop single connections** (dropconnect)

Choosing the Optimization Algorithm

- Standard Stochastic Gradient Descent (SGD)
  - Easy and efficient
  - Difficult to pick up the best learning rate
  - Unstable convergence
  - Often used with **momentum** (exponentially weighted history of previous weights changes)
- RMSprop
  - Adaptive learning rate method (reduces it using a moving average of the squared gradient)
  - Fastens convergence by having quicker gradients when necessary
- Adagrad
  - Like RMSprop with element-wise scaling of the gradient
- **ADAM**
  - **Like Adagrad but adds an exponentially decaying average of past gradients like momentum**

Momentum

- Adding a term to weight update equation to store an exponentially weight history of previous weights changes
- Reducing problems of instability while increasing the rate of convergence
  - If weight changes tend to have same sign momentum term increases and gradient decrease speed up convergence on shallow gradient
  - If weight changes tend have opposing signs momentum term decreases and gradient descent slows to reduce oscillations (stabilizes)
  - Can help escape being trapped in local minima

References

- Artificial Neural Network. Chapter 5.4 and 5.5. Introduction to Data Mining.
- Hands-on Machine Learning with Scikit-Learn, Keras & Tensorflow. A practical handbook to start wrestling with Machine Learning models (2nd ed).
- Deep Learning. Ian Goodfellow, Yoshua Bengio, and Aaron Courville. The reference book for deep learning models.