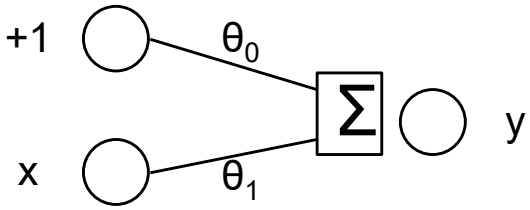
Perceptron and Linear Regression

TIN0255/09P9M80/D82 - Aprendizagem Profunda

2025.1 - Prof. Pedro Moura

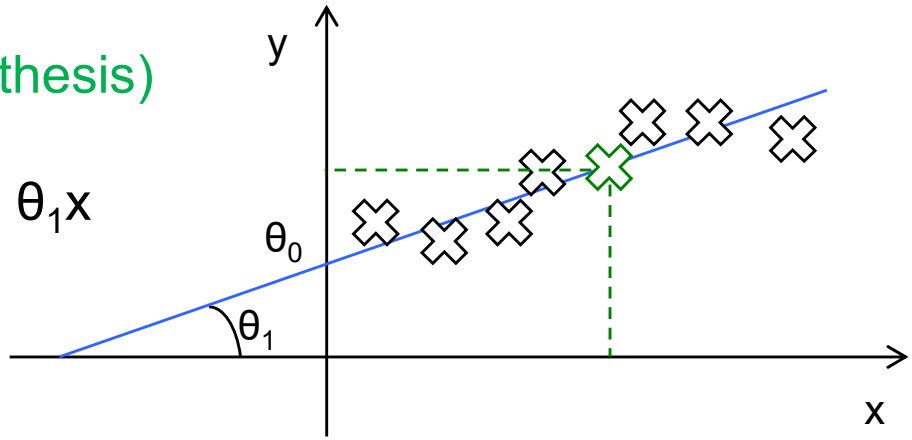
(Simple) Linear Regression (Prediction)

- Representation



Model (Hypothesis)

$h(x) = \theta_0 + \theta_1 x$



- Evaluation

Cost function : $J(\theta_0, \theta_1) = J_\theta = 1/m \ \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})^2$

Distance between predicted and real

- Optimization (Training)

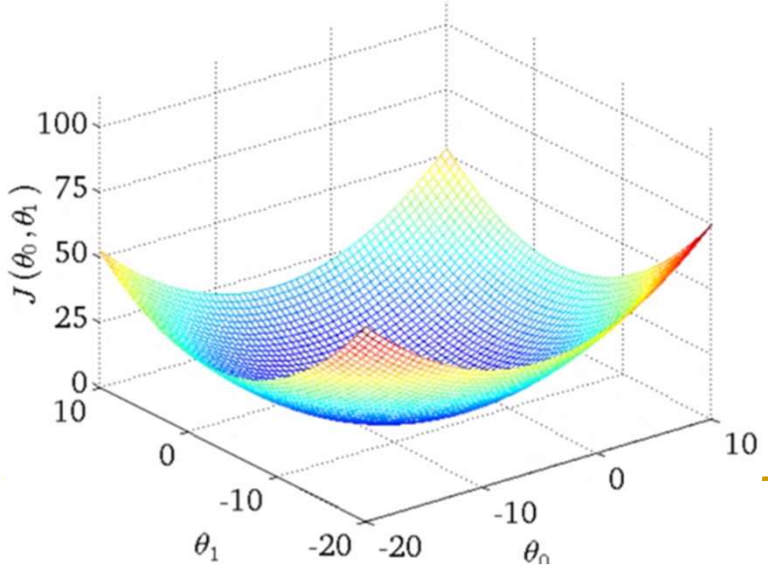
Objective: Find out "best" values of θ_0 and θ_1 to minimize Cost function

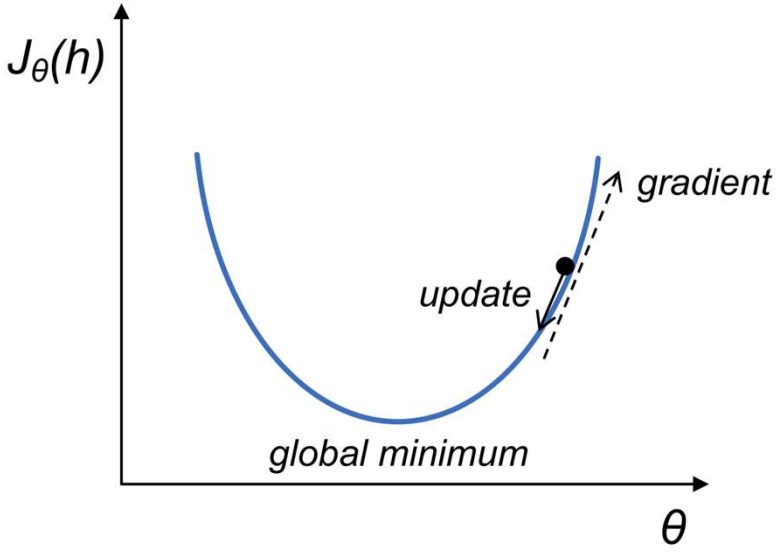
with:

	Example(1)	Example(2)		Example(m)
Input Dataset : X :	$x^{(1)}$	$x^{(2)}$	⋯	$x^{(m)}$
and Output Dataset : y :	$y^{(1)}$	$y^{(2)}$	⋯	$y^{(m)}$

Optimization

- Objective: Minimization of Cost function J_θ
- Usual/simple algorithm: Gradient descent
- Therefore, we need to compute the Cost, but also the Gradients (partial derivatives respective to each θ (weight)) in order to guide gradient descent
- As this Cost function is *convex*, there is a global minimum (and no local minima)





Optimization and Gradient

- We compute the gradient of the loss function with respect to the parameters o the network (in this case, **perceptron**)
- Let's recall the definition of the gradient
- Given a function $f : \mathbb{R}^N \rightarrow \mathbb{R}$ from a real-valued vector to a scalar
- If $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ and $y = f(\mathbf{x})$, the gradient of y with respect to x is given by:

$$\nabla_{\mathbf{x}} y = \left[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \ldots, \frac{\partial y}{\partial x_n}\right]$$

Optimization and Gradient

- We could let the function f be the loss function, e.g., the squared-error loss, which we denote by L . This loss is a scalar-valued function of the output $\mathbf{y}$:

$$L(\mathbf{y}) = \|\mathbf{y} - \hat{\mathbf{y}}\|^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

- The gradient of L with respect to $\mathbf{y}$ is then:

-

- $$\nabla_{\mathbf{y}} L = [2(y_1 - \hat{y}_1), (y_2 - \hat{y}_2), \ldots, 2(y_n - \hat{y}_n)] = 2(\mathbf{y} - \hat{\mathbf{y}})$$

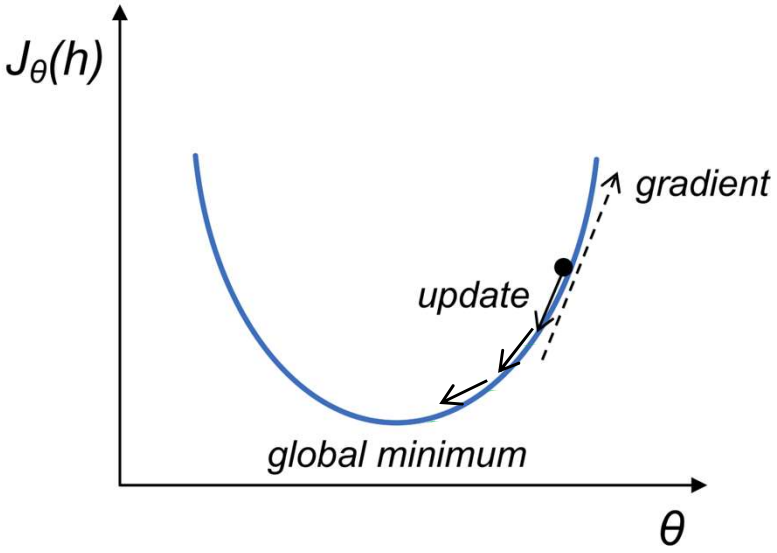
Gradient Descent Training Algorithm

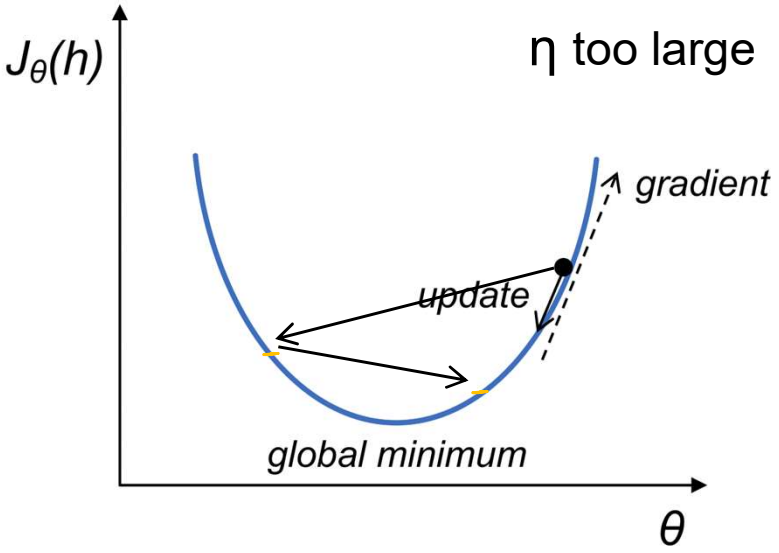
- Initialize each parameter θ_i and bias b to a random or some heuristic value
- Compute the values of the model *h* for all examples
- Compute the *cost* $J_\theta(h)$
- Compute also the Gradients (partial derivatives respective to each θ_i (weight)) in order to guide gradient descent
- **Update simultaneously** all parameters θ_i according to the update rule
- **Iterate** until the error between successive iterations change by only a tiny amount (i.e., we have reached a **local minimum**), or after a fixed number of iterations

Learning rate and Stochastic Gradient Descent (SGD)

Update Rule $\theta_0 := \theta_0 - \eta\, \partial J(\theta_0, \theta_1)/\partial\theta_0$
$\theta_1 := \theta_1 - \eta\, \partial J(\theta_0, \theta_1)/\partial\theta_1$

η : Learning rate → **It is important to pick the learning rate carefully**





Compute Cost and Gradients

Gradient Descent (SGD): for ALL examples
Stochastic Gradient Descent (SGD): for ONE example (randomly chosen)
MiniBatch: for a Batch (subset) randomly chosen

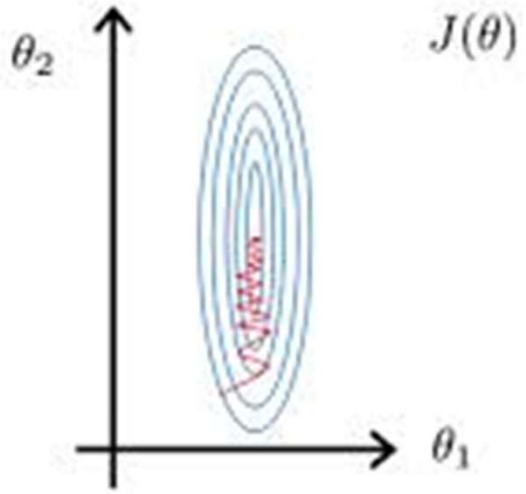
Learning rate and Stochastic Gradient Descent (SGD)

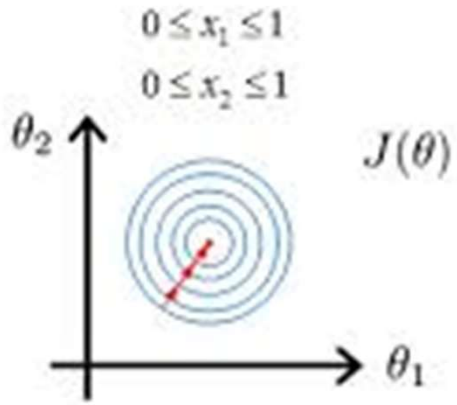
Some considerations regarding Gradient Descent

- Too small a value of η → might take **very large number of iterations to converge**
- Too large a value of η → might cause **large oscillations in the parameter values and never lead to converge**
- Picking the right value → matter of trial and error
- It is also common to vary the learning rate :
  - Start with an initial learning rate η
  - At each iteration, multiply by a factor β, $0 < \beta < 1$ until the learning rate reaches a sufficiently low value

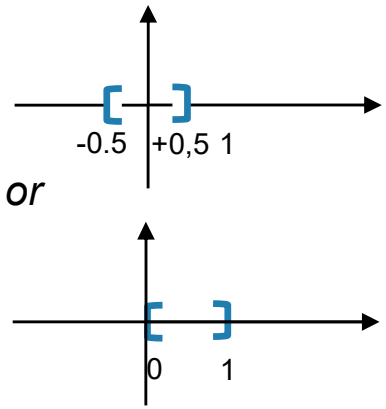
Scaling and Mean Normalization

- To improve learning convergence (faster), better to have features (x_1, x_2…) on a similar scale



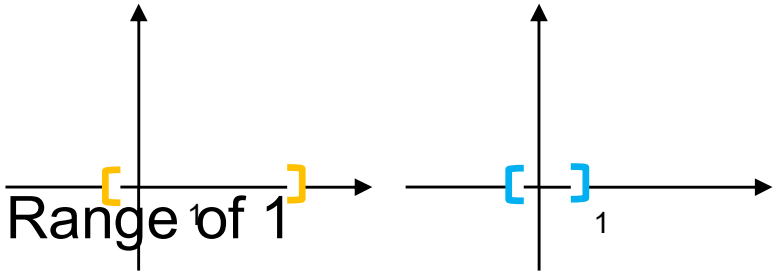


scaled and normalized



- Feature Scaling

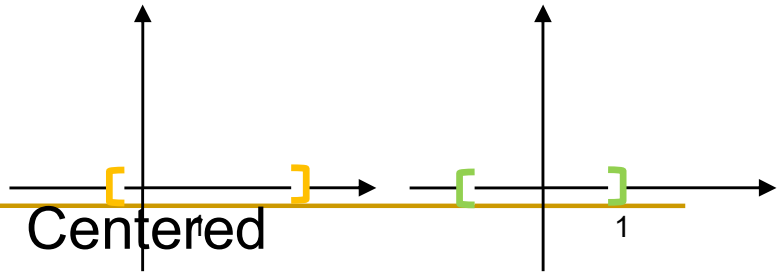
  x := x – min(x) / range(x) -> Range of 1



(Could also apply)

- Mean Normalization

  x := x – mean(x) / range(x) -> Centered



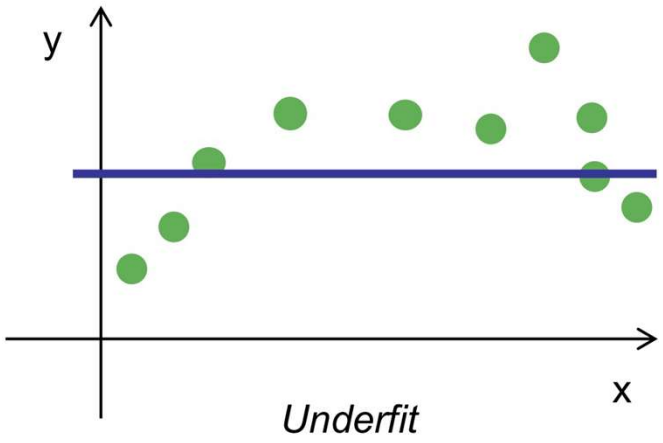
Best Hypothesis

- Objective: Find out "best" hypothesis (values of θ_0 and θ_1)

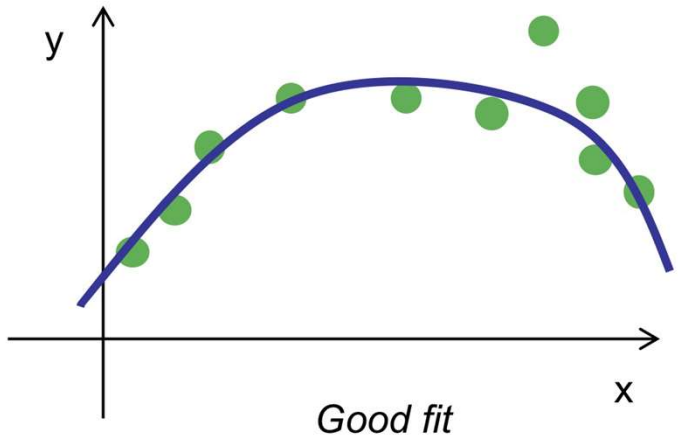
- Best hypothesis means best fit to data
- And not just best fit only to Train data
- Otherwise the best strategy would be to exactly memorize every entry

- Best hypothesis means best fit for future/yet unknown data: Test data
- i.e., with good capacity for Generalization

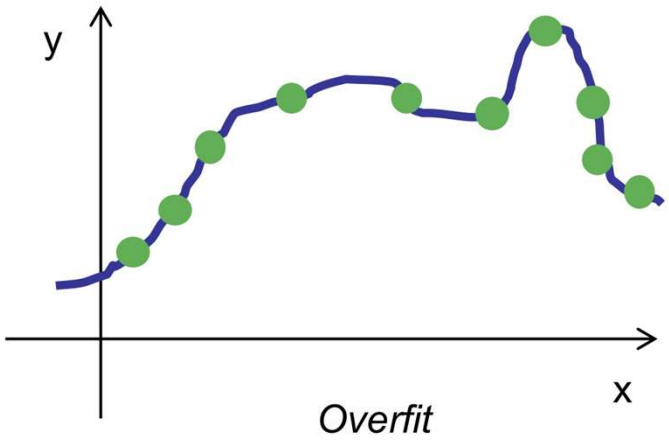
Best Hypothesis



(High Bias)



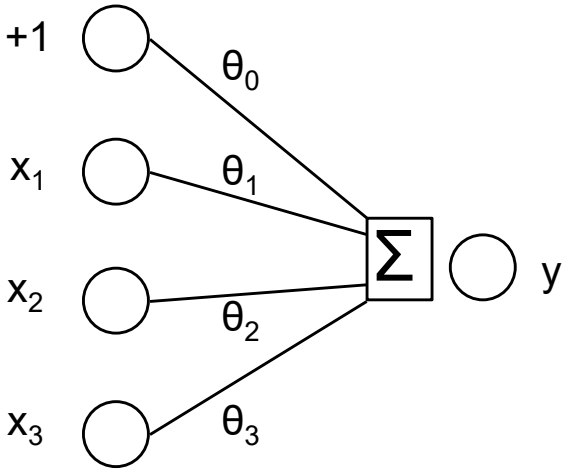
Objective



(High Variance)
Low Generalization

- Regularization: technique for reducing overfitness
  - Weights decay, to penalize over-preponderant weights
  - Also: Dropout, Early stopping, Dataset augmentation

Multiple Linear Regression – More than 1 Variable



$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$$

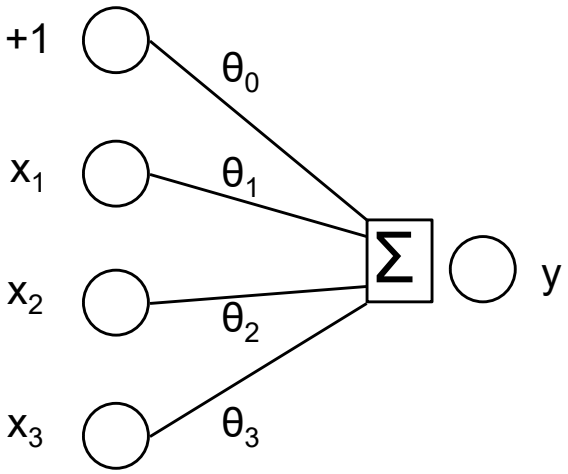
Input Dataset : X :

$$\begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{bmatrix} \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \\ x_3^{(2)} \end{bmatrix} \cdots \begin{bmatrix} x_1^{(m)} \\ x_2^{(m)} \\ x_3^{(m)} \end{bmatrix}$$

Output Dataset : y :

$$\begin{bmatrix} y^{(1)} \end{bmatrix} \begin{bmatrix} y^{(2)} \end{bmatrix} \cdots \begin{bmatrix} y^{(m)} \end{bmatrix}$$

Multiple Linear Regression Vectorization



$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$$
$$= \sum_{i=0}^{n} \theta_i x_i \quad \text{(with } x_0 = 1\text{)}$$
$$= [\theta_0\ \theta_1\ \theta_2\ \theta_3] * \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

*scalar(1x1) = vector(1x4) * vector(4x1)*

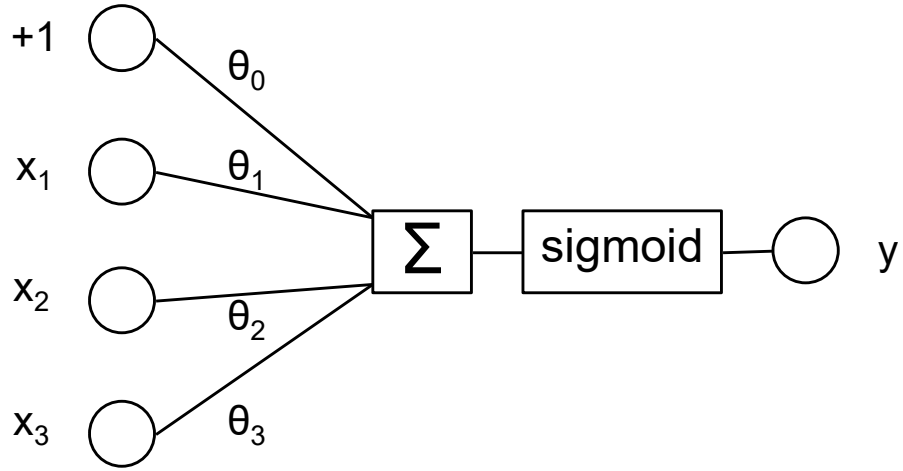
$$= \theta * x$$

Computing predictions for a set of examples X :

$$\theta * X = [\theta_0\ \theta_1\ \theta_2\ \theta_3] * \begin{bmatrix} 1 & \dots & 1 \\ x_1^{(1)} & \dots & x_1^{(m)} \\ x_2^{(1)} & \dots & x_2^{(m)} \\ x_3^{(1)} & \dots & x_3^{(m)} \end{bmatrix} = [h^{(1)} \dots h^{(m)}]$$

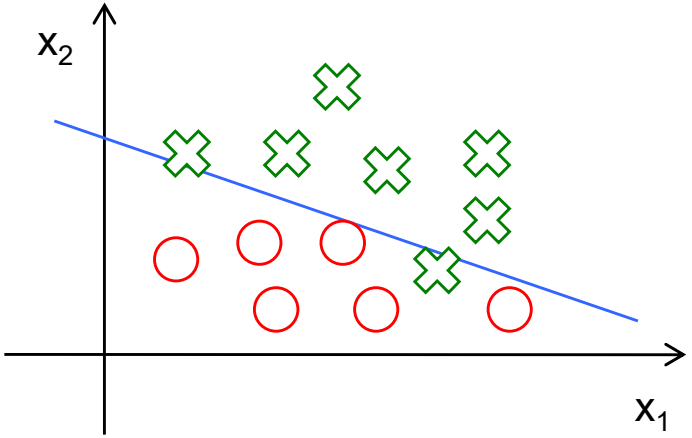
*vector(1x4) * matrix(4xm) = vector(1xm)*

Multiple Logistic Regression (Classification)

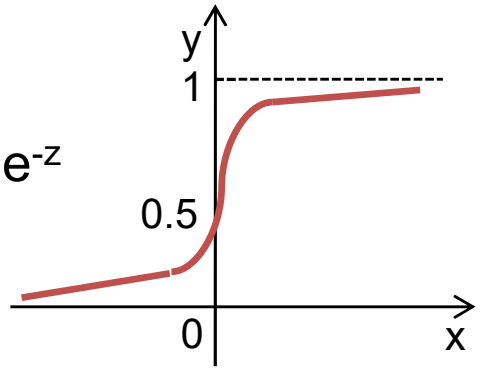


$h(x) \in \{0, 1\}$

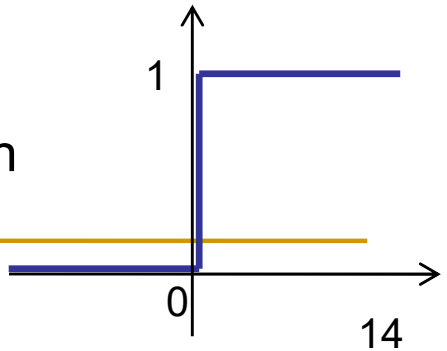
$h(x) = \text{sigmoid}(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3)$
$= \text{sigmoid}(\theta * x)$



$\text{sigmoid}(z) = \sigma(z) = 1 \,/\, 1 + e^{-z}$

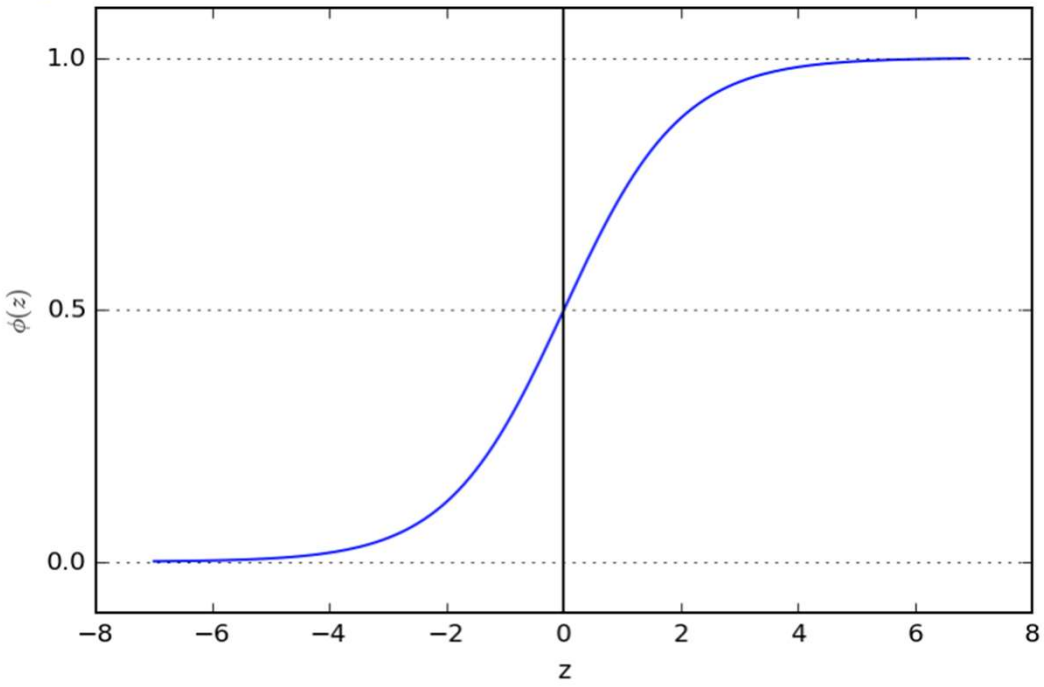


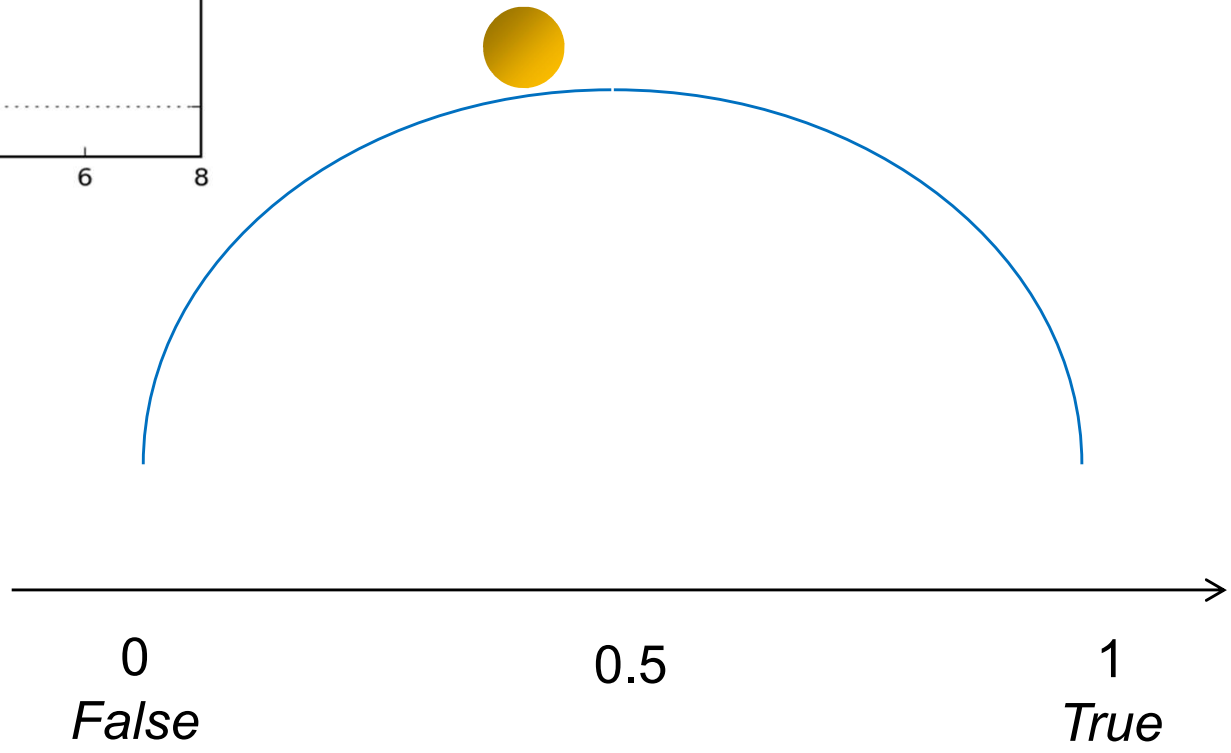
Perceptron [Rosenblatt 1957] used unit step function



Current Neural Networks use Softmax (transforms into probabilities)

Sigmoid

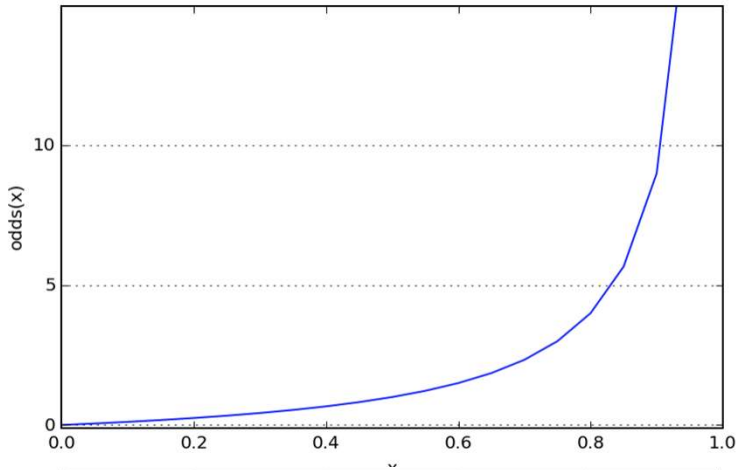




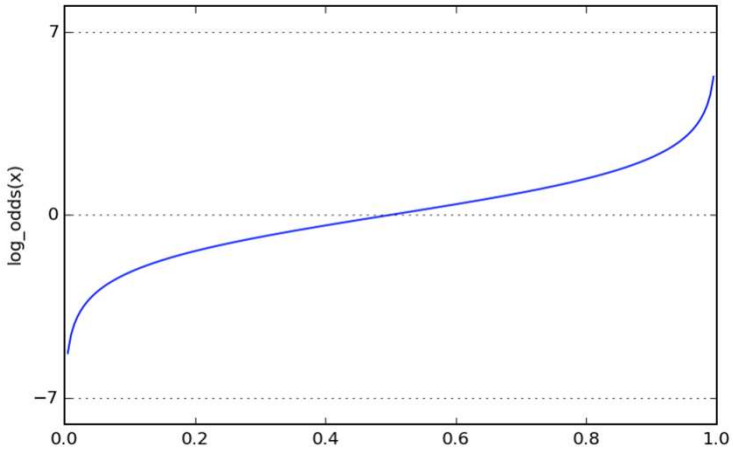
Sigmoid – Where does it come from? [Rosaen, 2016]

- Odds ratio = P / (1 - P)
- Log Odds ratio = Log(P / (1 - P))

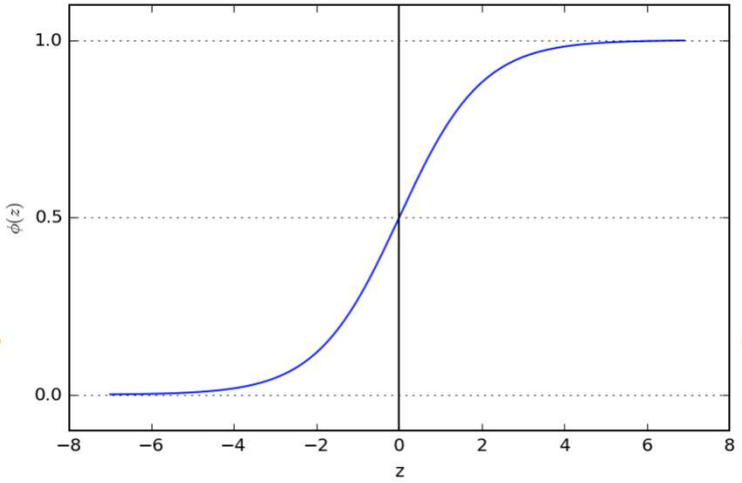
- (Log Odds ratio)$^{-1}$ = ??
- y = log(x / 1 - x)
- x = log(y /1 - y)
- e^x = y / 1 – y
- y = (1 – y) e^x
- y = e^x – y e^x
- y + y e^x = e^x
- y (1 + e^x) = e^x
- y = e^x / (1 + e^x)
- y = 1 / (1/e^x + 1)
- y = e^x / (1 + e^x)
- sigmoid(x) = σ(x) = e^x / (1 + e^x)



Odds



Log Odds



Sigmoid

Classifier Evaluation Metrics

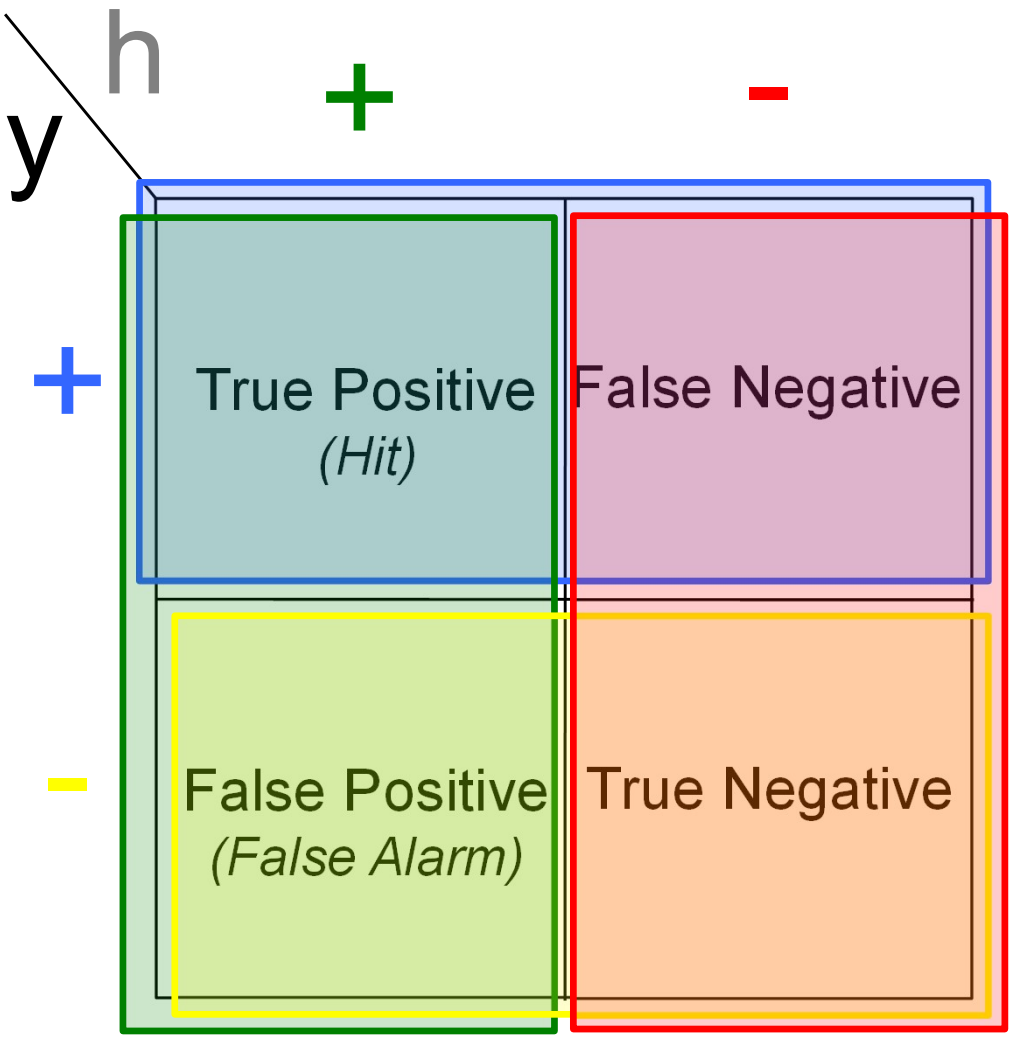
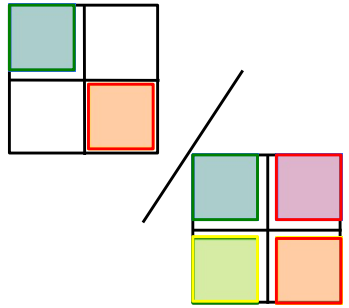


Table of Confusion
(Confusion Matrix)

Accuracy = TP + TN / ∀
% correctly classified



Classifier Evaluation Metrics

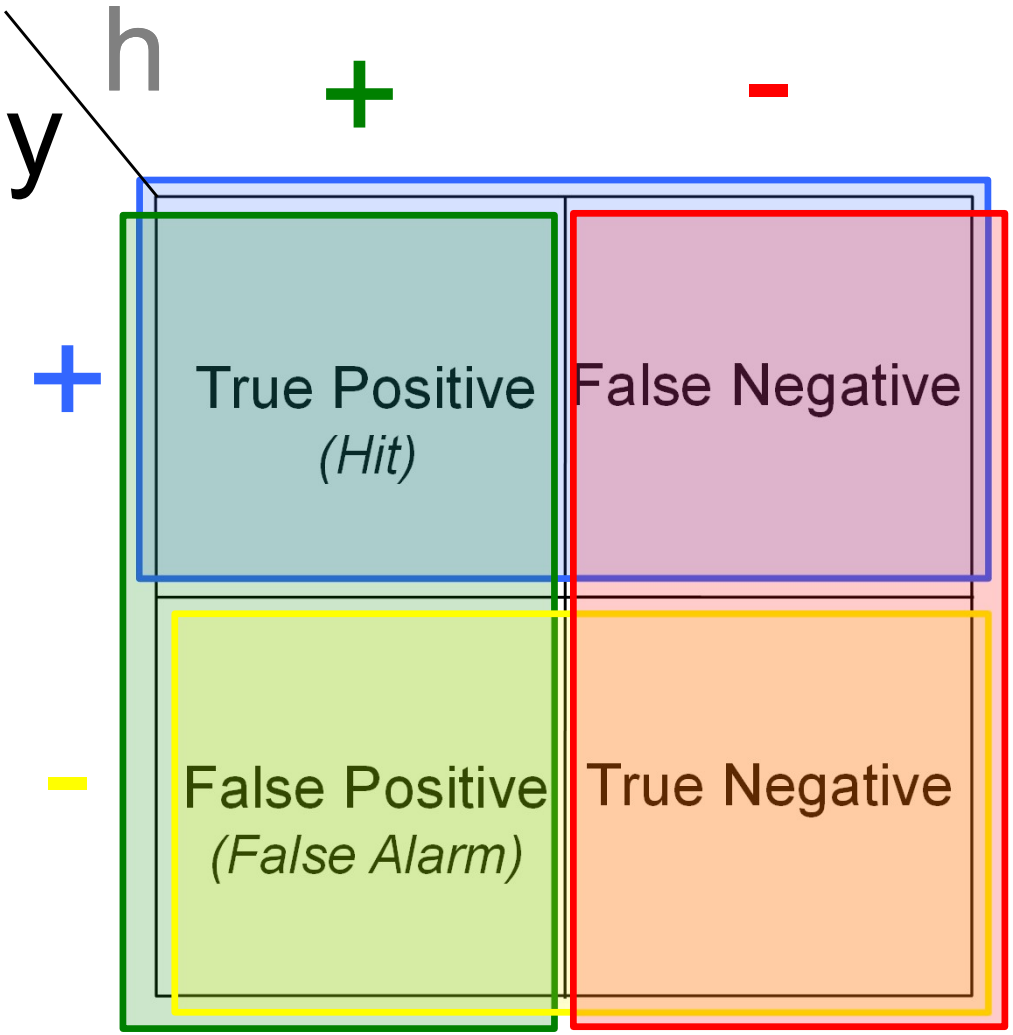
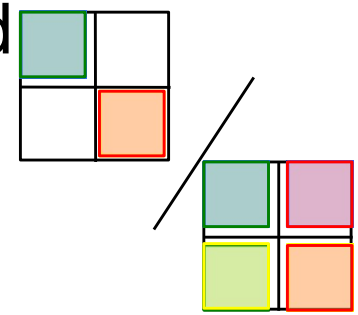
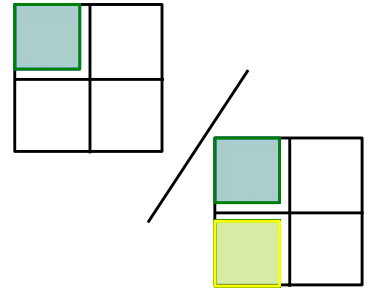


Table of Confusion
(Confusion Matrix)

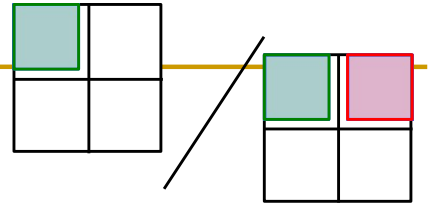
Accuracy = TP + TN / ∀
% correctly classified



Precision = TP / P (TP + FP)
true / classified positive
How useful the results



Recall = TP / P (TP + FN)
true / real positive
How complete the results



Skewed Classes

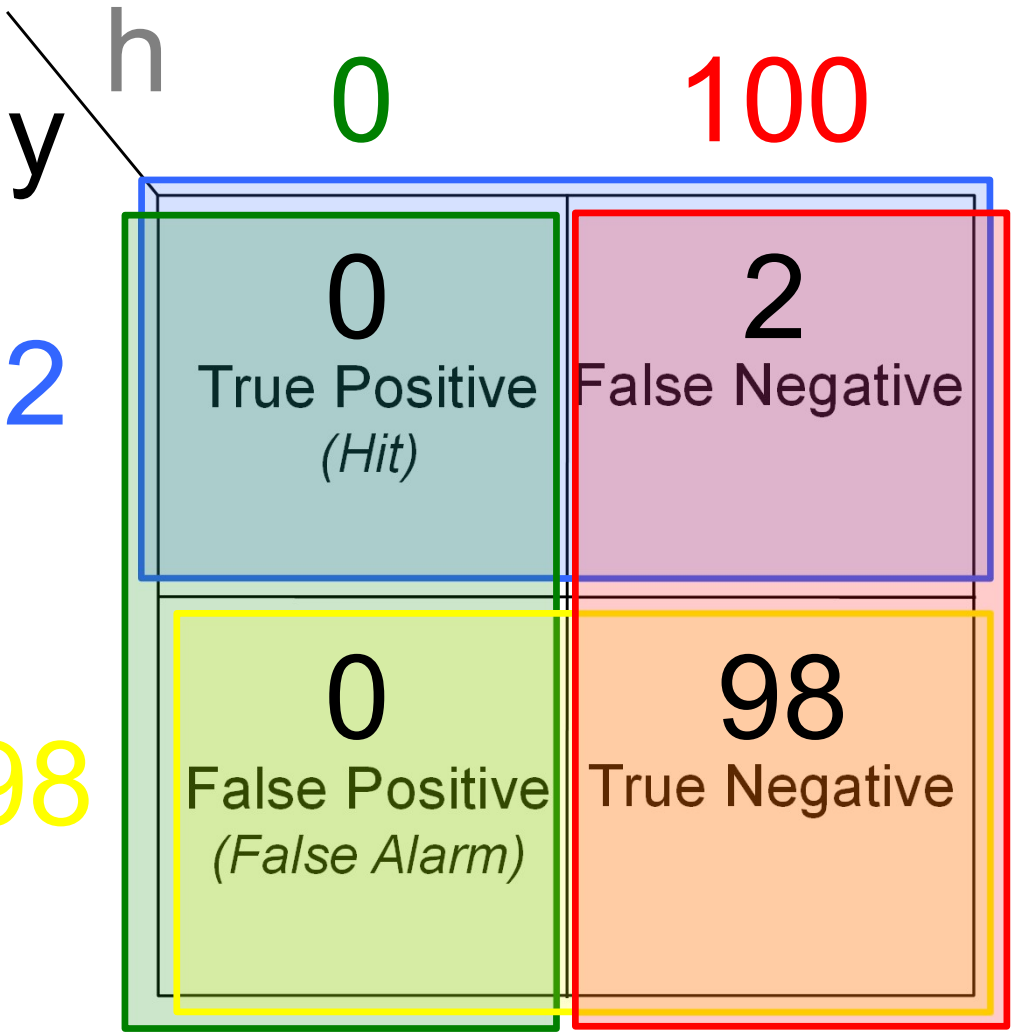
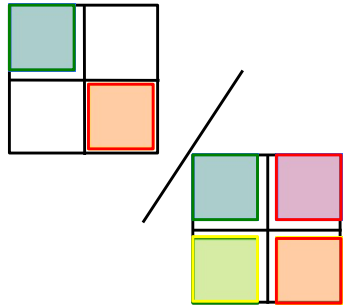


Table of Confusion
(Confusion Matrix)

Accuracy = TP + TN / ∀ = 0.98
% correctly classified



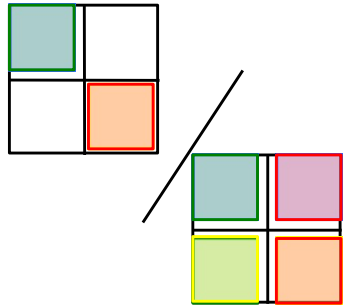
Ex: Maligne Cancer Prediction: 2% Rate
Optimist Hypothese: Always (100%) False

Skewed Classes

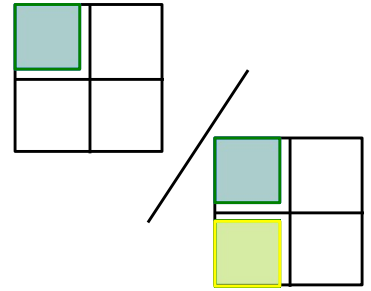
y \ h	0	100
2	0 True Positive *(Hit)*	2 False Negative
98	0 False Positive *(False Alarm)*	98 True Negative

Table of Confusion
(Confusion Matrix)

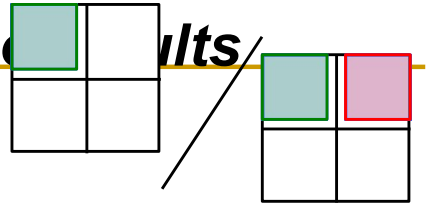
Accuracy = TP + TN / ∀ = 0.98
% correctly classified



Precision = TP / P (TP + FP) = 0
true / classified positive
How useful the results



Recall = TP / P (TP + FN) = 0
true / real positive
How complete the results



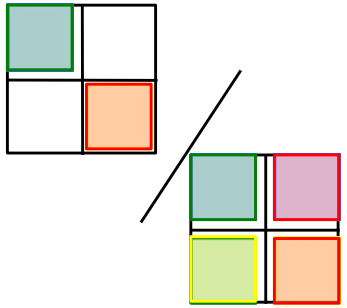
Ex: Maligne Cancer Prediction: 2% Rate
Optimist Hypothese: Always (100%) False

Skewed Classes

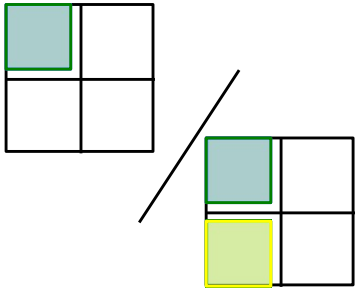
y	0	100
2	1 True Positive *(Hit)*	1 False Negative
98	2 False Positive *(False Alarm)*	96 True Negative

Table of Confusion
(Confusion Matrix)

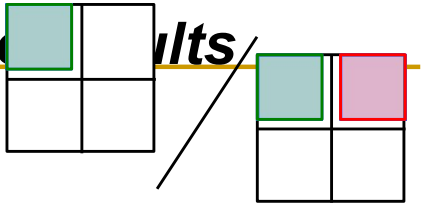
Accuracy = TP + TN / ∀ = 0.97
% correctly classified



Precision = TP / P (TP + FP) = 0
true / classified positive
How useful the results



Recall = TP / P (TP + FN) = 0.5
true / real positive
How complete the results



Ex: Maligne Cancer Prediction: 2% Rate
#1 Test

Skewed Classes

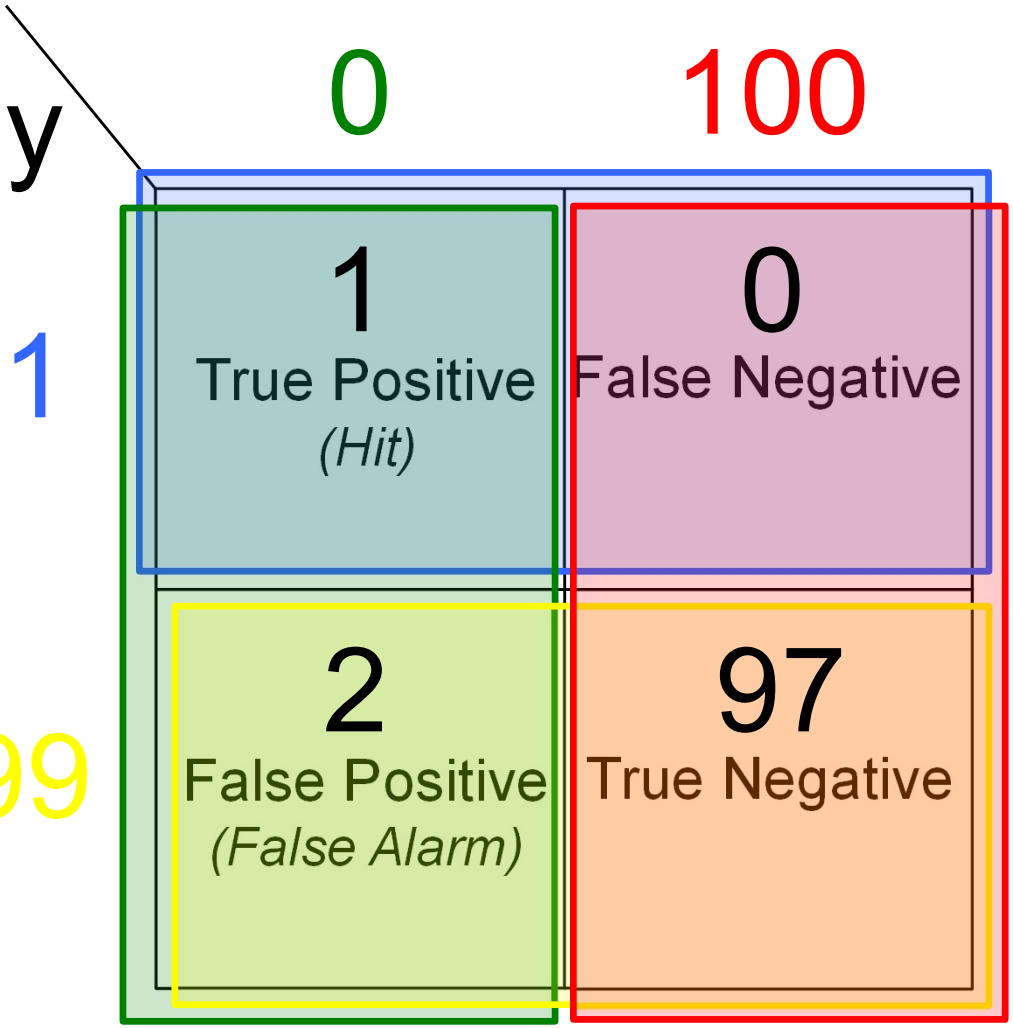
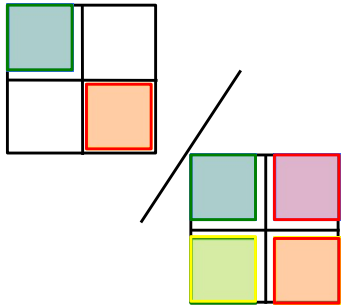
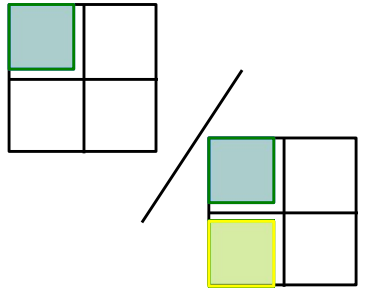


Table of Confusion
(Confusion Matrix)

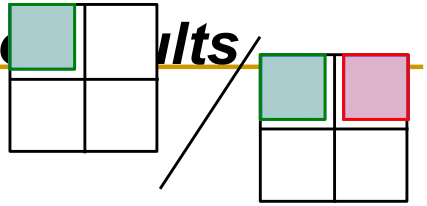
Accuracy = TP + TN / ∀ = 0.98
% correctly classified



Precision = TP / P (TP + FP) = 0
true / classified positive
How useful the results

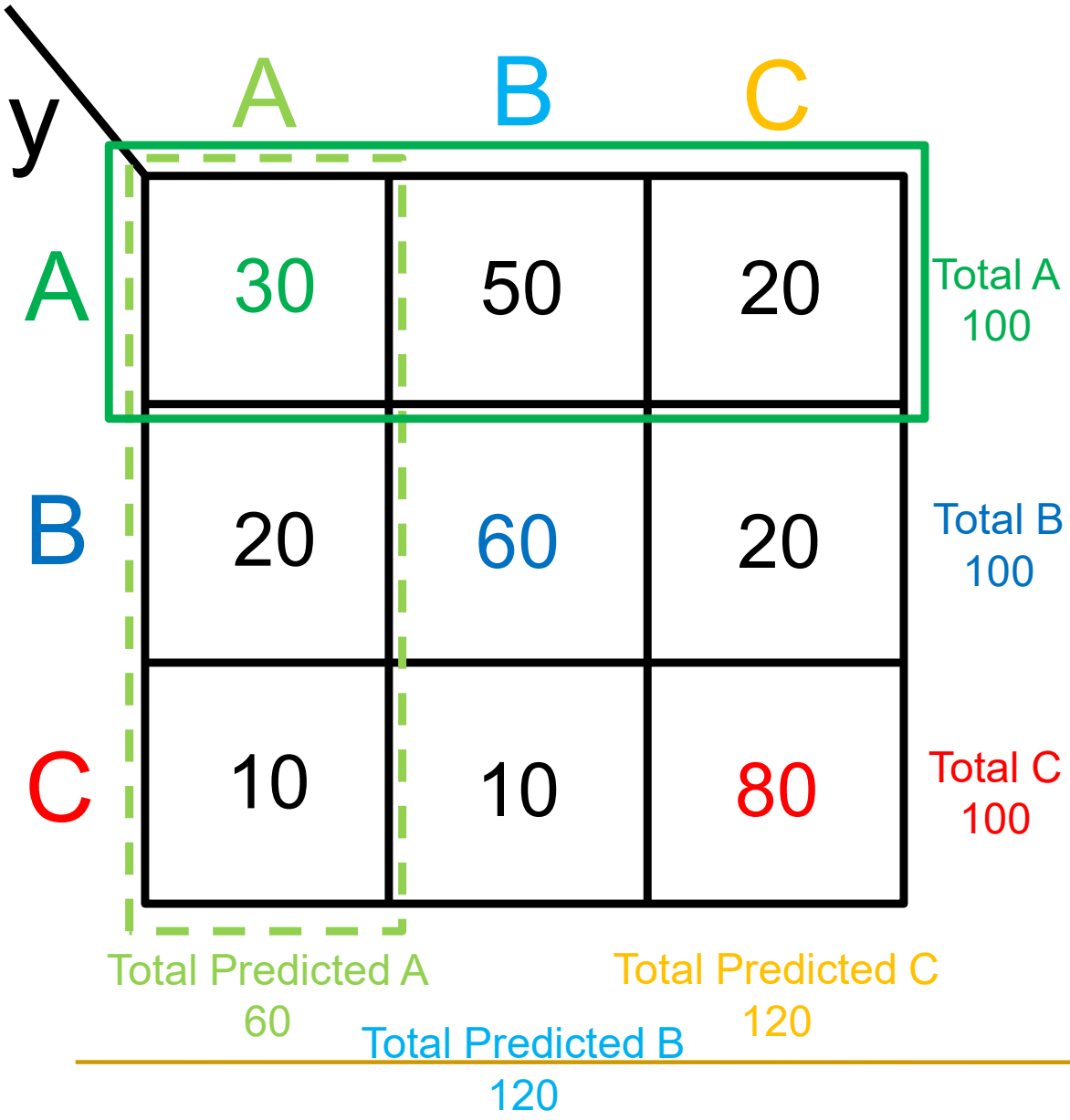


Recall = TP / P (TP + FN) = 1
true / real positive
How complete the results



Ex: Maligne Cancer Prediction: 2% Rate
#2 Test

Multilabel/Multiclass Confusion Matrix



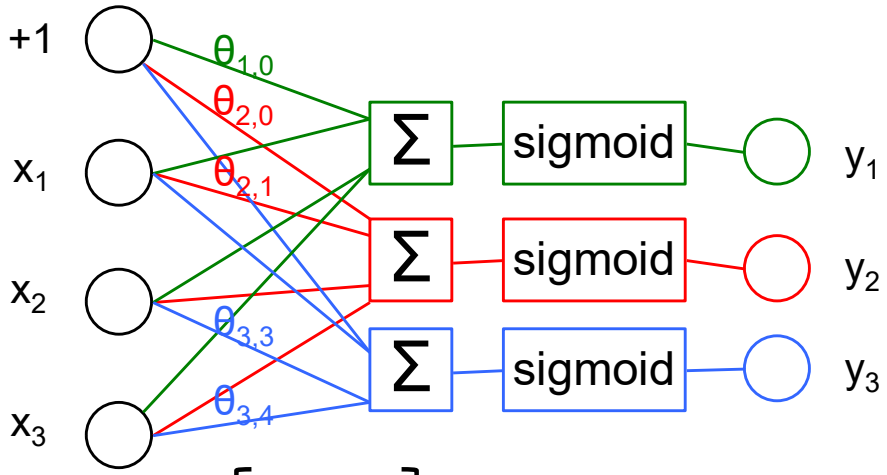
Precision(label X)
= True Positive(X) / Total Predicte

Ex: Precision(A)
= True Positive(A) / Total Predicte
= 30/60 = 0.5

Recall(label X)
= True Positive(X) / Total Label(X

Ex: Recall(A)
= True Positive(A) / Total Label(A
= 30/100 = 0.3

Multivariate Multiple Logistic Regression (Classification)



$h(x) = \text{sigmoid}(\theta * x)$

$$\theta = \begin{bmatrix} \cdots\ \theta_{i,j}\ \cdots \end{bmatrix}$$

$$X: \quad \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{bmatrix} \quad \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \\ x_3^{(2)} \end{bmatrix} \quad \cdots \quad \begin{bmatrix} x_1^{(m)} \\ x_2^{(m)} \\ x_3^{(m)} \end{bmatrix}$$

$$y: \quad \begin{bmatrix} y_1^{(1)} \\ y_2^{(1)} \\ y_3^{(1)} \end{bmatrix} \quad \begin{bmatrix} y_1^{(2)} \\ y_2^{(2)} \\ y_3^{(2)} \end{bmatrix} \quad \cdots \quad \begin{bmatrix} y_1^{(m)} \\ y_2^{(m)} \\ y_3^{(m)} \end{bmatrix}$$

Current Neural Networks use Softmax

Multivariate Multiple Classification Vectorization

$$h(x) = \begin{bmatrix} h_1(x) \\ h_2(x) \\ h_3(x) \end{bmatrix} = \begin{bmatrix} \theta_{1,0} + \theta_{1,1}x_1 + \theta_{1,2}x_2 + \theta_{1,3}x_3 \\ \theta_{2,0} + \theta_{2,1}x_1 + \theta_{2,2}x_2 + \theta_{2,3}x_3 \\ \theta_{3,0} + \theta_{3,1}x_1 + \theta_{3,2}x_2 + \theta_{3,3}x_3 \end{bmatrix} = \theta * x = \begin{bmatrix} \theta_{1,0} & \theta_{1,1} & \theta_{1,2} & \theta_{1,3} \\ \theta_{2,0} & \theta_{2,1} & \theta_{2,2} & \theta_{2,3} \\ \theta_{3,0} & \theta_{3,1} & \theta_{3,2} & \theta_{3,3} \end{bmatrix} * \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

*vector(3x1) = matrix(3x4) * vector(4x1)*

Computing predictions for a set of examples X :

$$\theta * X = \begin{bmatrix} \theta_{1,0} & \theta_{1,1} & \theta_{1,2} & \theta_{1,3} \\ \theta_{2,0} & \theta_{2,1} & \theta_{2,2} & \theta_{2,3} \\ \theta_{3,0} & \theta_{3,1} & \theta_{3,2} & \theta_{3,3} \end{bmatrix} * \begin{bmatrix} 1 & \dots & 1 \\ x_1^{(1)} & \dots & x_1^{(m)} \\ x_2^{(1)} & \dots & x_2^{(m)} \\ x_3^{(1)} & \dots & x_3^{(m)} \end{bmatrix} = \begin{bmatrix} h_1^{(1)} & \dots & h_1^{(m)} \\ h_2^{(1)} & \dots & h_2^{(m)} \\ h_3^{(1)} & \dots & h_3^{(m)} \end{bmatrix}$$

[]

$[class^{(1)} \dots class^{(m)}]$

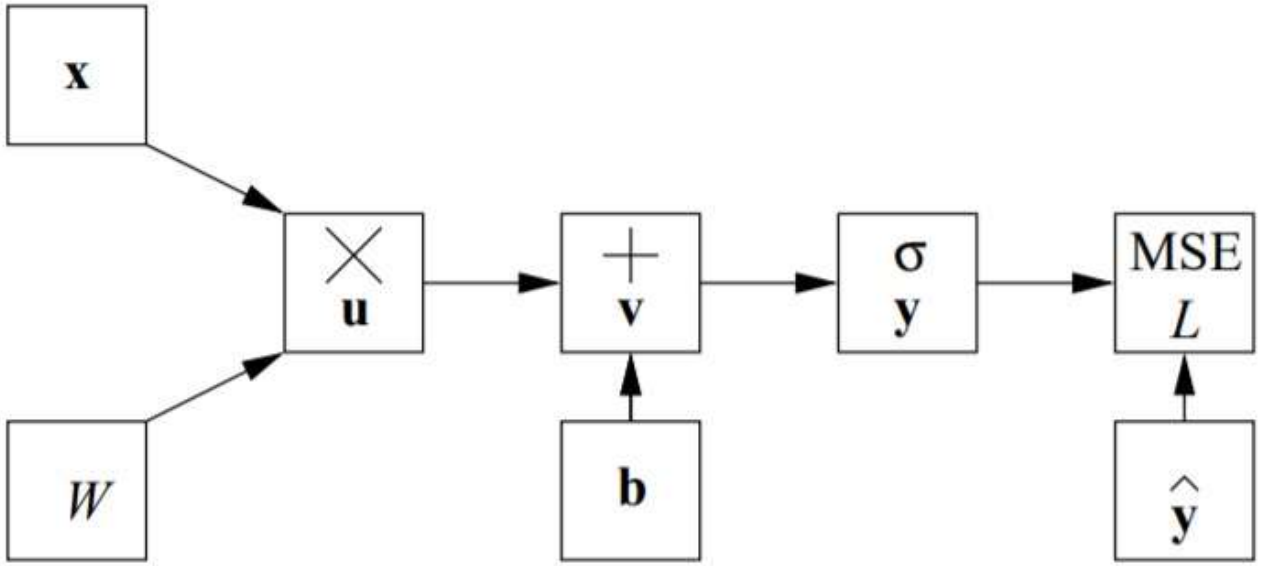
*matrix(3x4) * matrix(4xm) = matrix(3xm) vector(1xm)*

Compute graph for a single-layer dense network

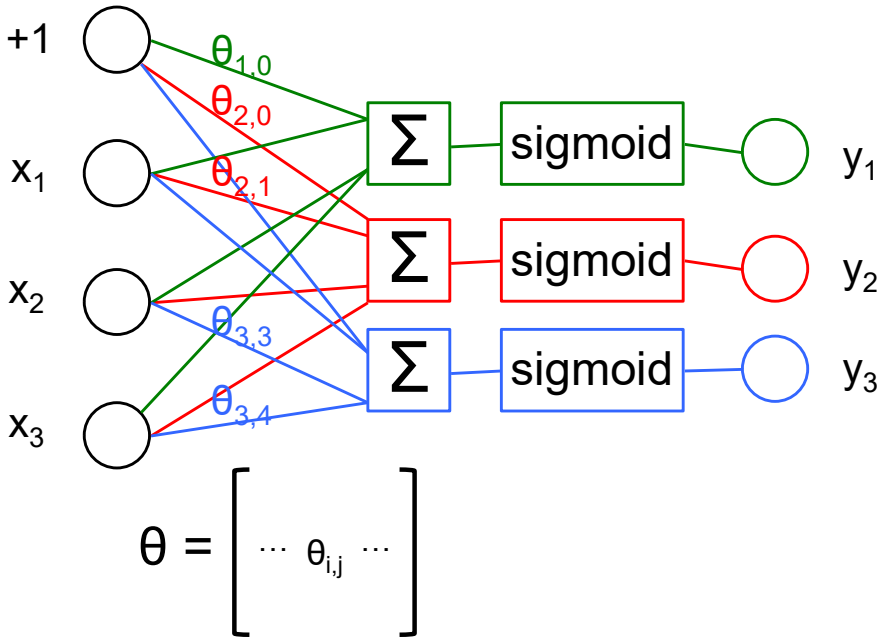
- A compute graph captures the **data flow** of a deep network
  - In fact, it is a Directed Acyclich Graph (DAG)
- Each node has an operand and, optionally, an operator
- Operand can be a **scalar**, a **vector**, or a **matrix**
- Operator is a **linear algebra operator** (such **+** or **x**), an **activation function** (such as σ) or a **loss function** (such as MSE)
- Next, I show the compute graph for a single-layer dense network described by $\boldsymbol{y = \sigma(Wx + b)}$, where **x** is the input and **y** is the output.
  - MSE loss computed against the training-set output $\hat{\mathbf{y}}$

Compute graph for a single-layer dense network

$$\mathbf{u} = W\mathbf{x}$$
$$\mathbf{v} = \mathbf{u} + \mathbf{b}$$
$$\mathbf{y} = \sigma(\mathbf{v})$$
$$L = \text{MSE}(\mathbf{y}, \hat{\mathbf{y}})$$



Modern Neural Network



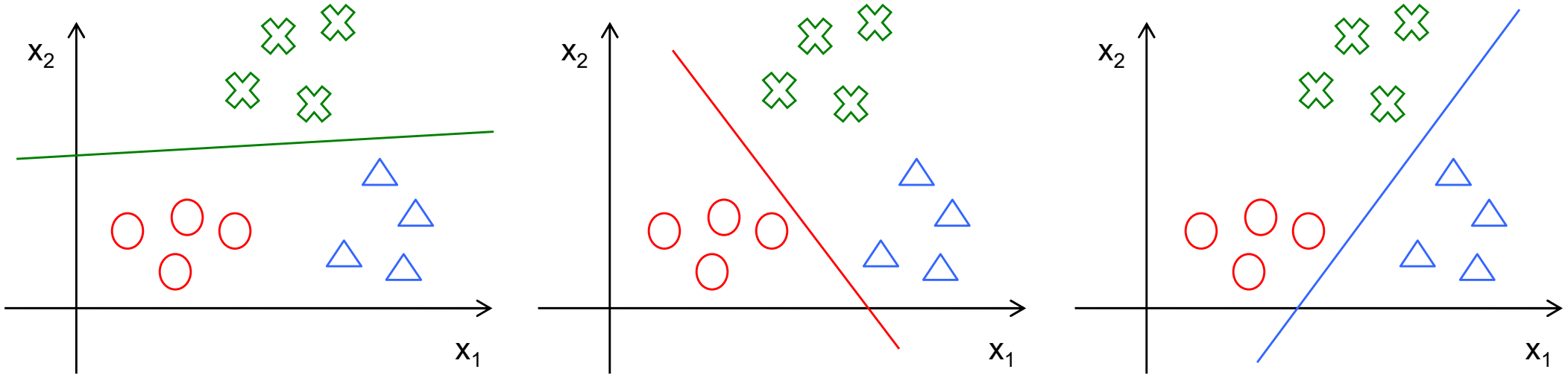
$$h(x) = \text{sigmoid}(\theta * x)$$

$$X: \quad \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{bmatrix} \quad \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \\ x_3^{(2)} \end{bmatrix} \quad \cdots \quad \begin{bmatrix} x_1^{(m)} \\ x_2^{(m)} \\ x_3^{(m)} \end{bmatrix}$$

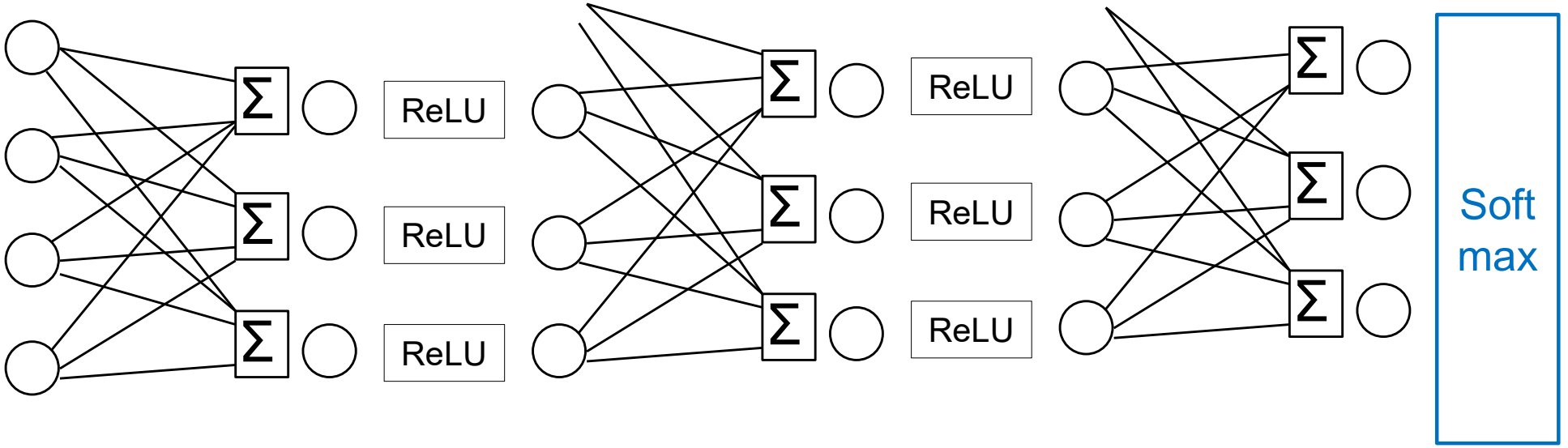
$$y: \quad \begin{bmatrix} y_1^{(1)} \\ y_2^{(1)} \\ y_3^{(1)} \end{bmatrix} \quad \begin{bmatrix} y_1^{(2)} \\ y_2^{(2)} \\ y_3^{(2)} \end{bmatrix} \quad \cdots \quad \begin{bmatrix} y_1^{(m)} \\ y_2^{(m)} \\ y_3^{(m)} \end{bmatrix}$$

Modern Neural Network

One vs All Algorithm: higher score = class recognized



Modern Neural Network



Softmax transforms values into probabilities

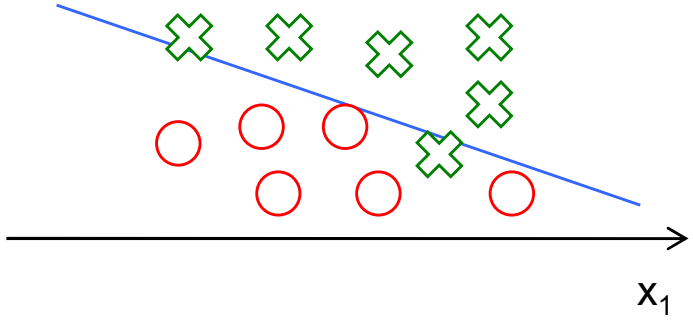
$$\sigma(z)_i = e^{z_i} / \Sigma\, e^{z_j}$$

Generalization of sigmoid

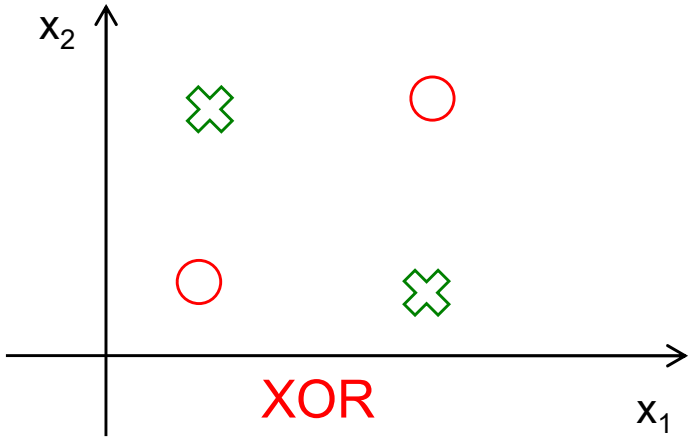
Associated cost function is cross entropy

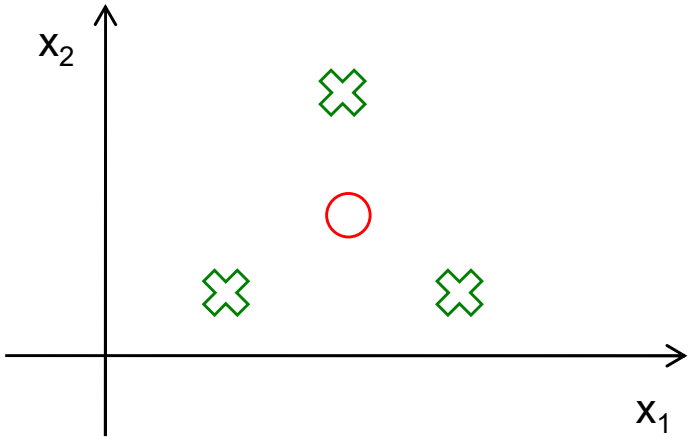
Linear vs Non Linear Decision Boundary

- Linear



- Non Linear

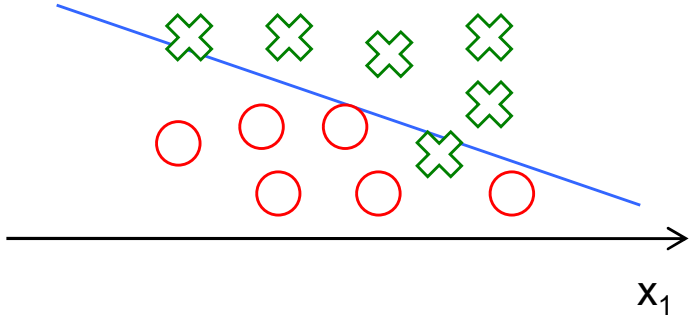




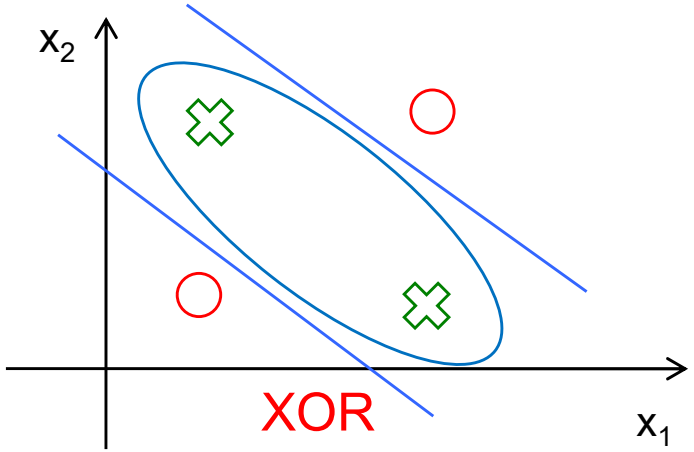
- Argument (XOR) used by [Minsky & Papert 1969] to criticize Perceptrons [Rosenblatt 1957] (and advocate Symbolic Artificial Intelligence)
- This stopped research on Perceptrons/Neural Networks for a long while
  - until Hidden Layers and Backpropagation or/and Kernel Trick (see later)

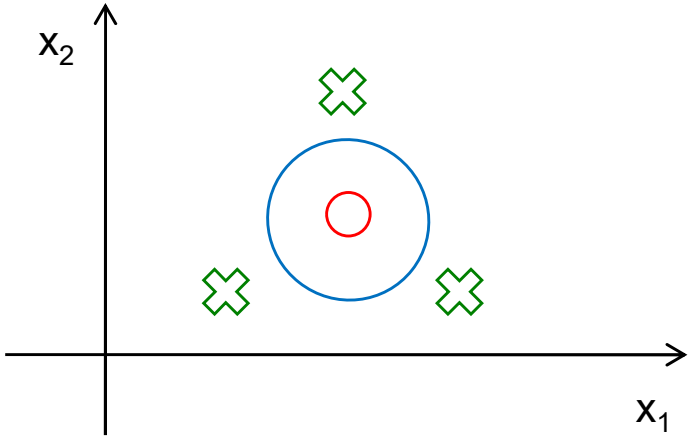
Linear vs Non Linear Decision Boundary

- Linear



- Non Linear

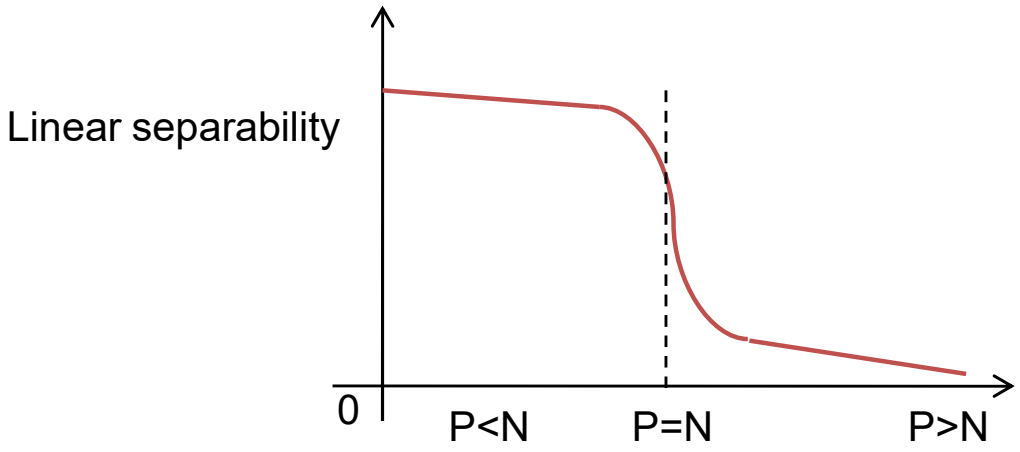




- Argument (XOR) used by [Minsky & Papert 1969] to criticize Perceptrons [Rosenblatt 1957] (and advocate Symbolic Artificial Intelligence)
- This stopped research on Perceptrons/Neural Networks for a long while
  - until Hidden Layers and Backpropagation or/and Kernel Trick (see later)

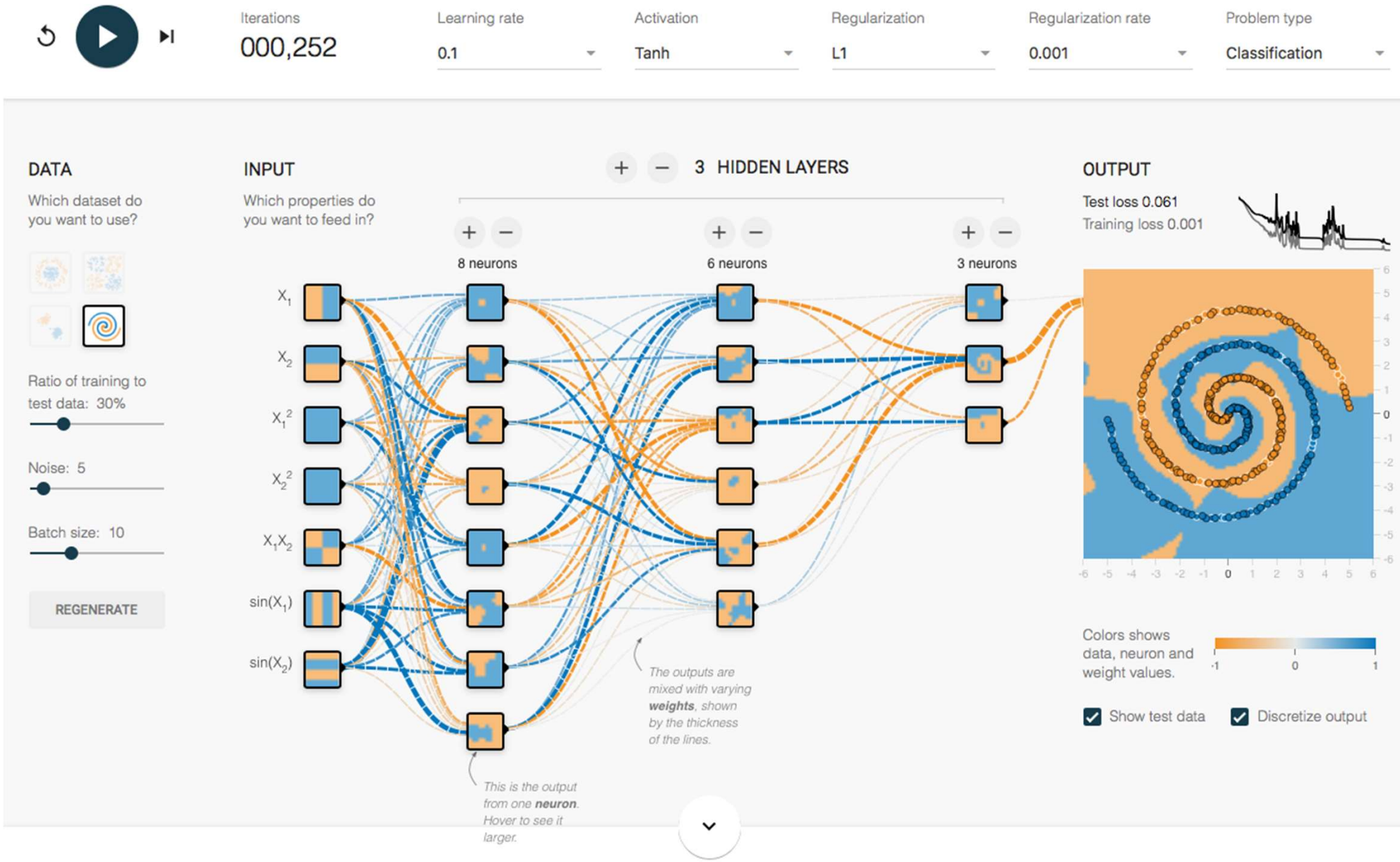
Linear vs Non Linear Decision Boundary

- A consequence of Cover's Theorem [Cover, 1966]
  - The probability that P samples (examples) of dimension N (number of parameters) are
  - linearly separable goes to zero very quickly as P grows larger than N

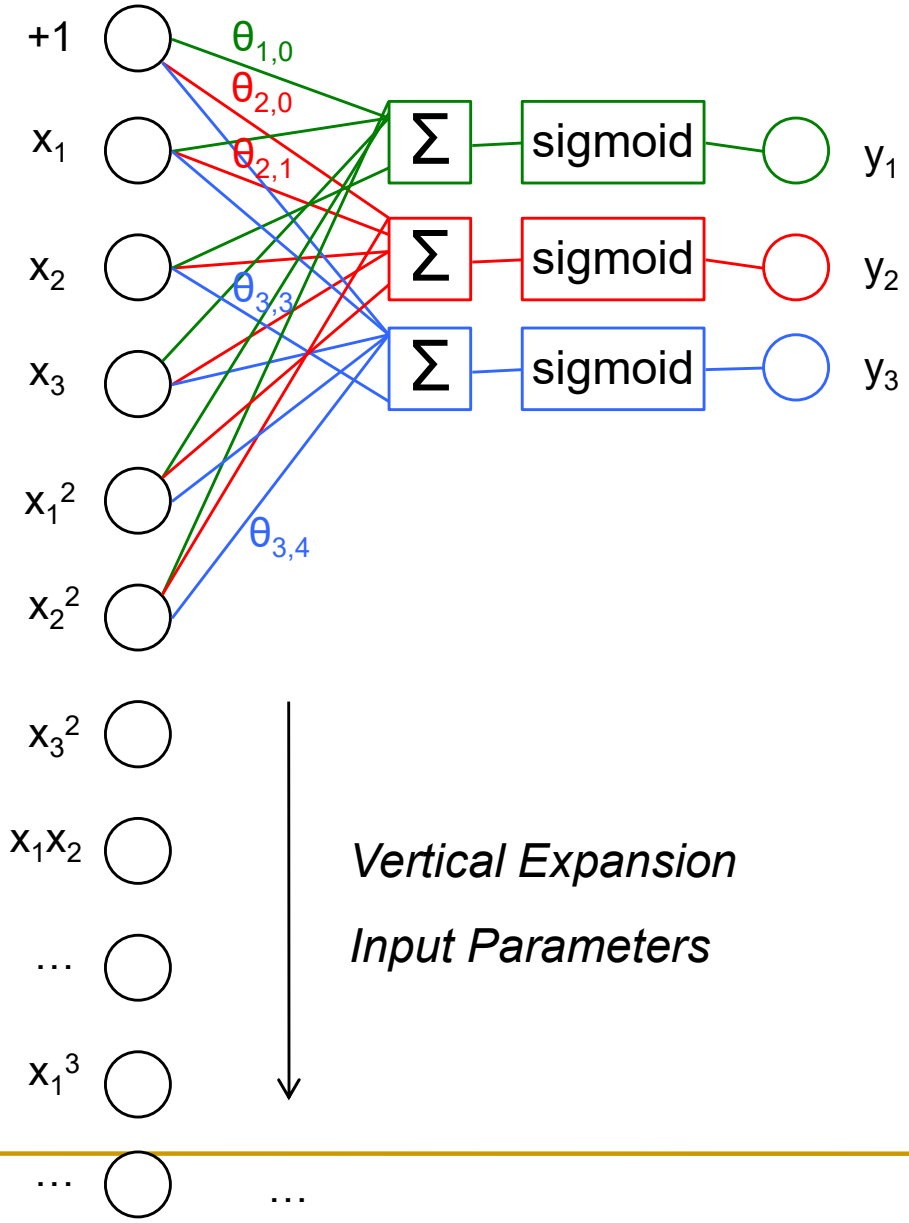


[LeCun 2016]

Example (TensorFlow PlayGround)



1st Approach: Polynomial Logistic Regression



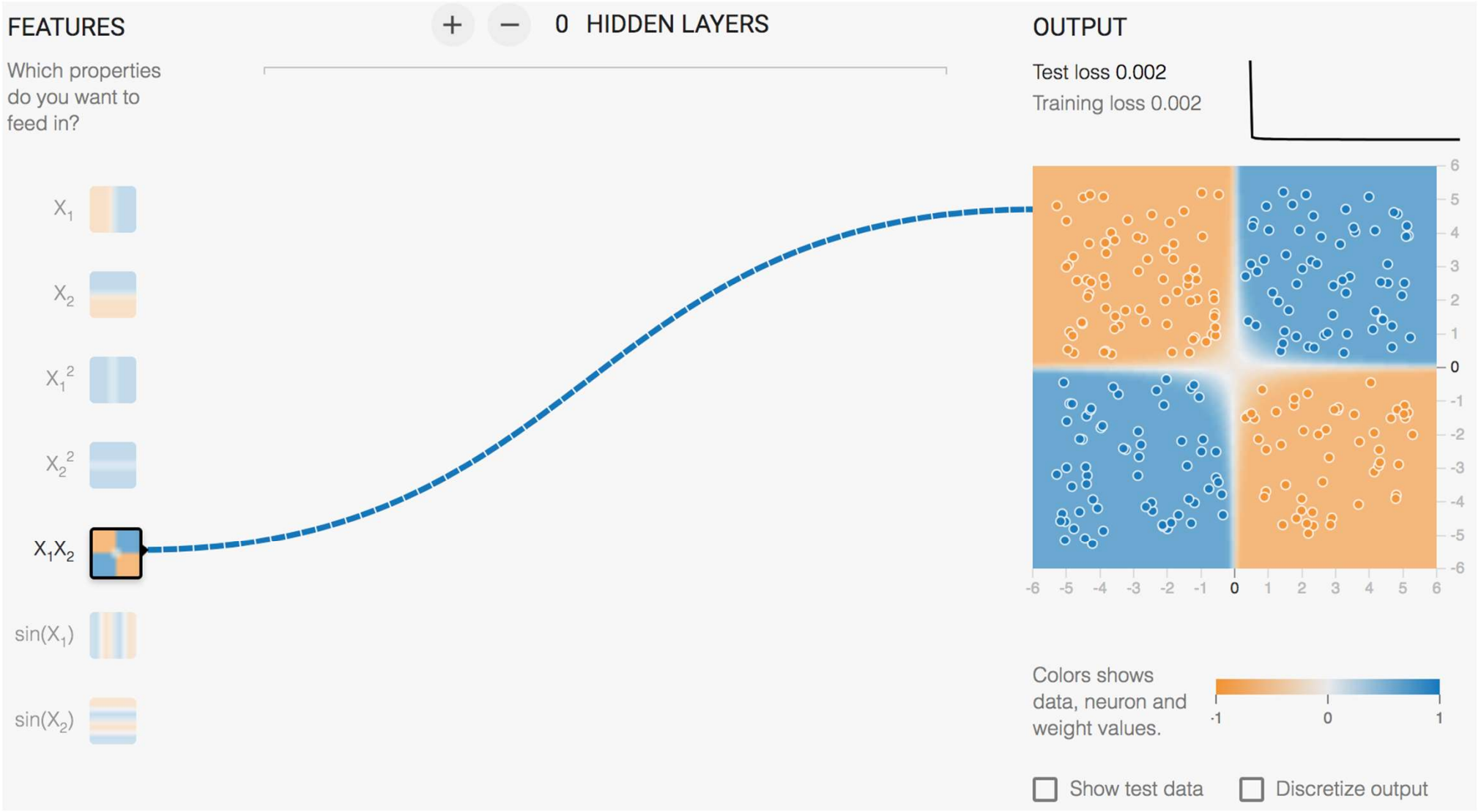
1st Approach: Polynomial Logistic Regression

- Idea:
  - Increase the number of Dimensions
  - By adding Polynomial Inputs
    - » Quadratic: x_1^2, x_2^2…
    - » Dot products: x_1x_2, x_1x_3…
    - » Cubic: x_1^3, x_2^3…
    - » Sinus: $\sin(x_1)$, $\sin(x_2)$...
    - » ...
- In order to find (explore) a higher Dimension Space in which could be
- found a Linear Decision Boundary

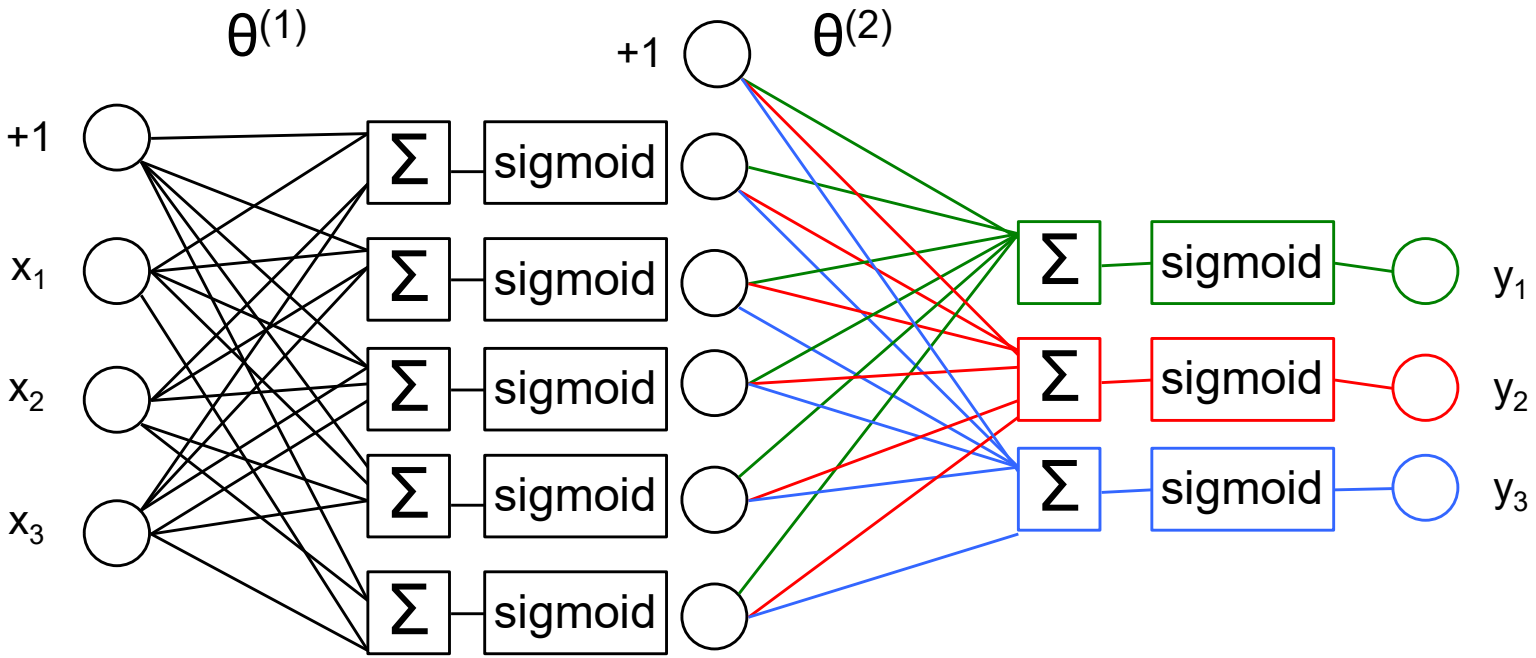
- Problems:
  - Select the Additional Inputs
  - Add them in a combinatorial way
  - The Number of Inputs could become Very Large
  - Computation and Data becomes more Heavy

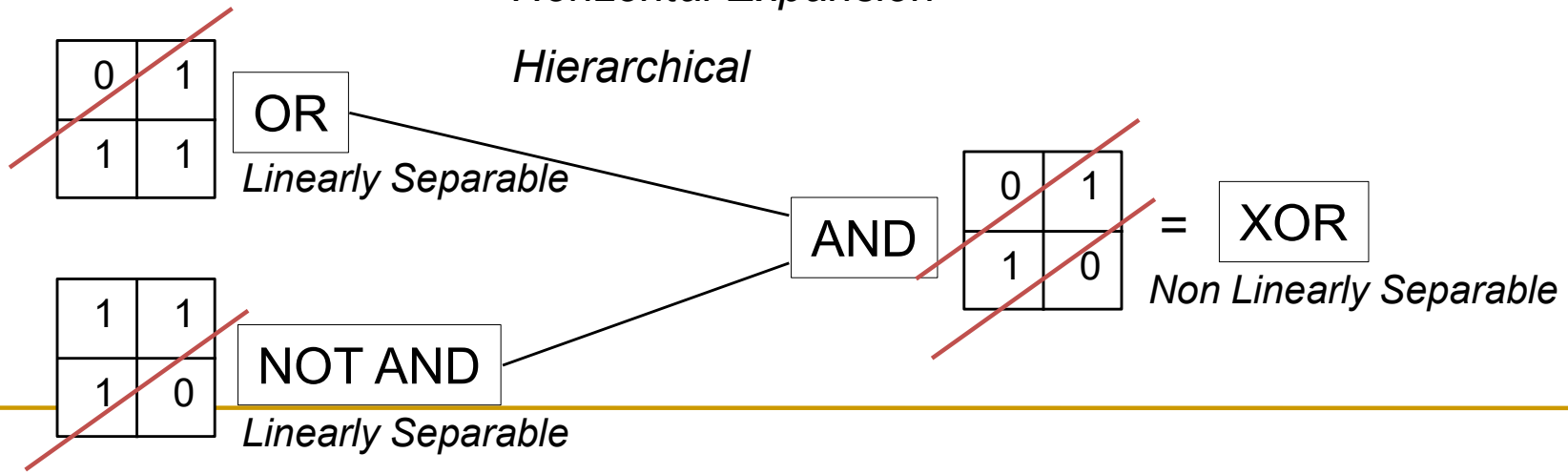
Example: x1x2

- Initial Representation $x_1 \; x \; x_2$
- New Representation x_1x_2

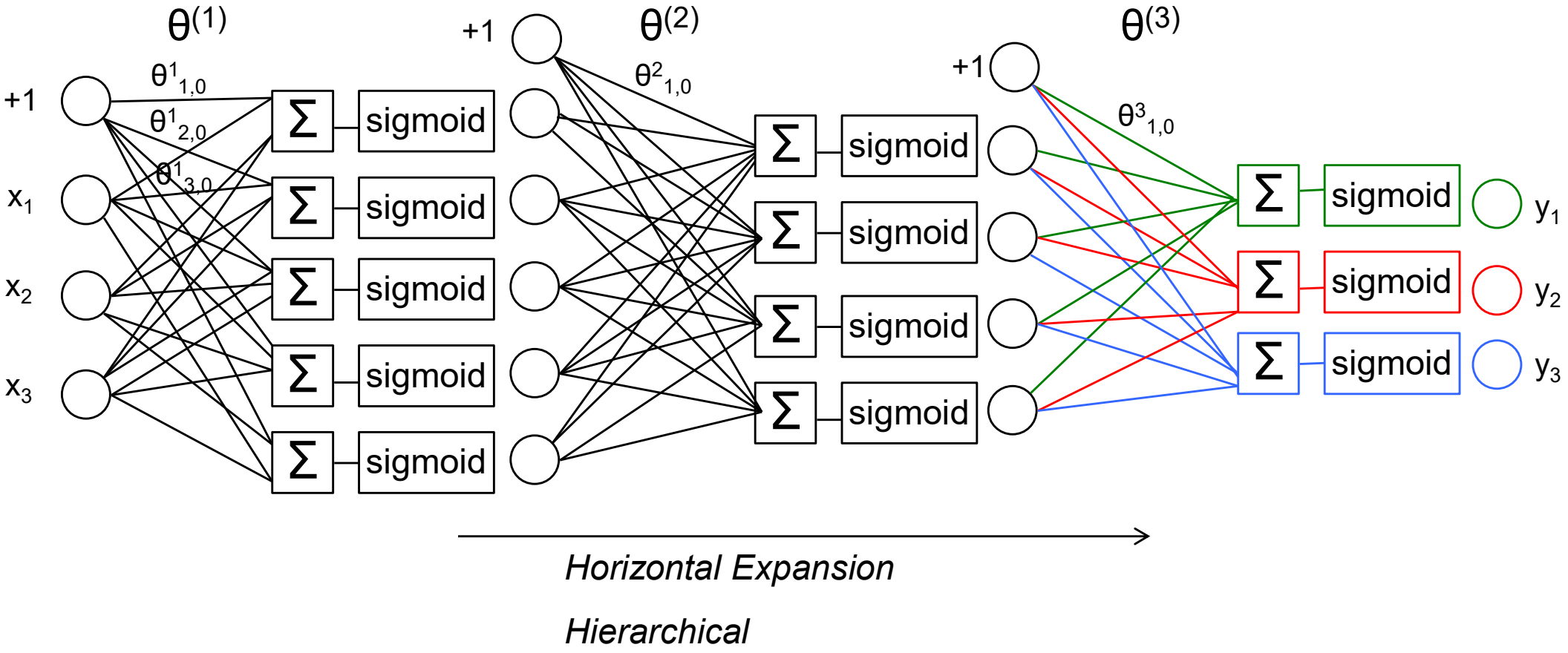


2nd Approach: Hidden Layers

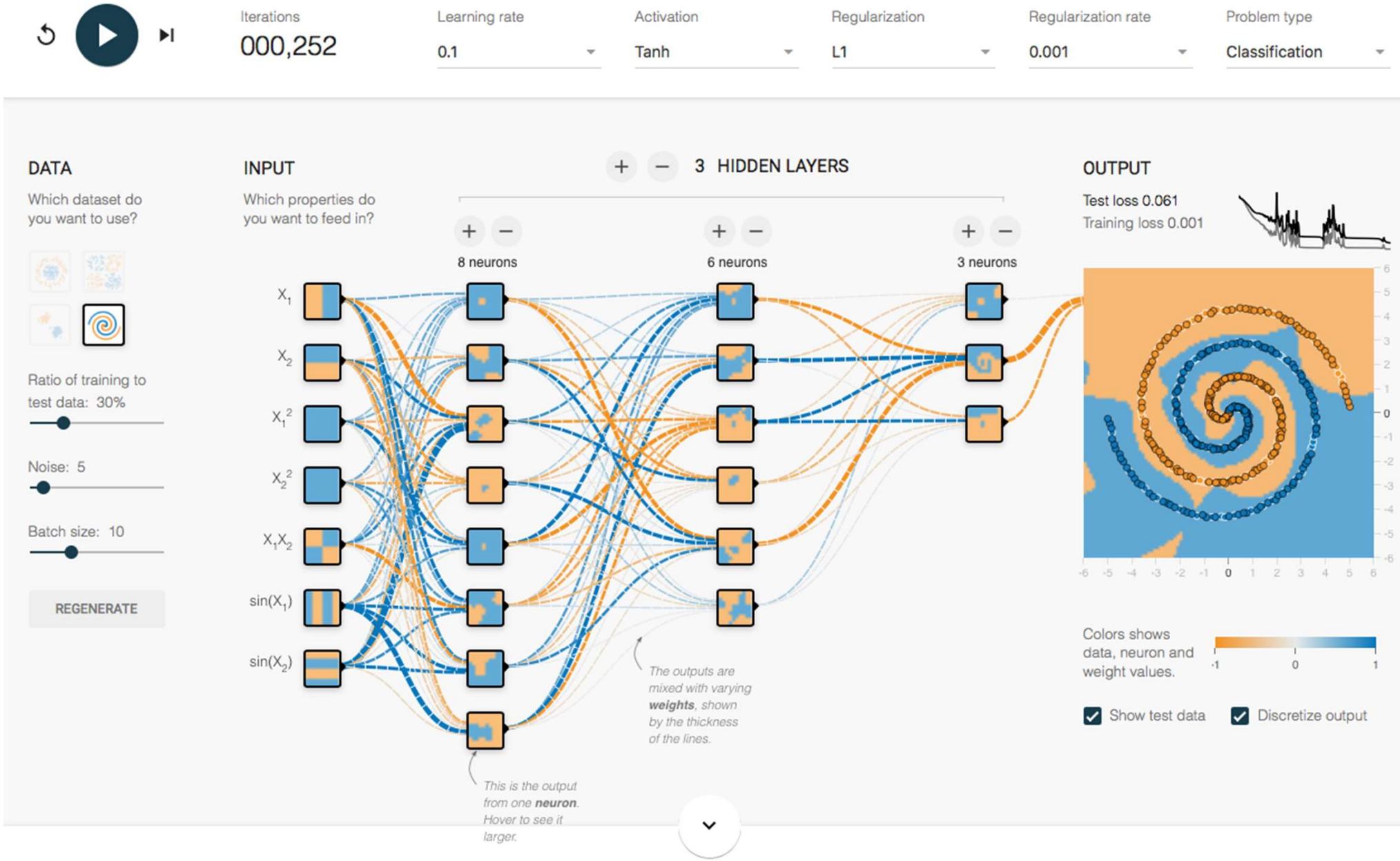




2nd Approach: Hidden Layers



Example (TensorFlow PlayGround)

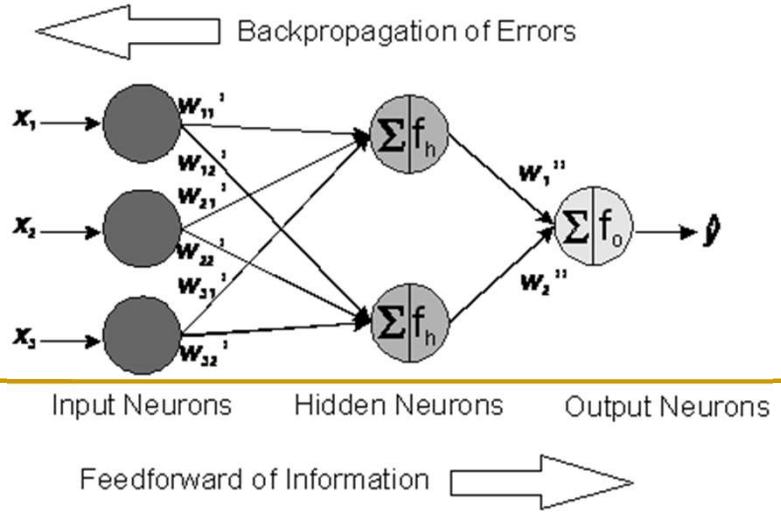


Backpropagation

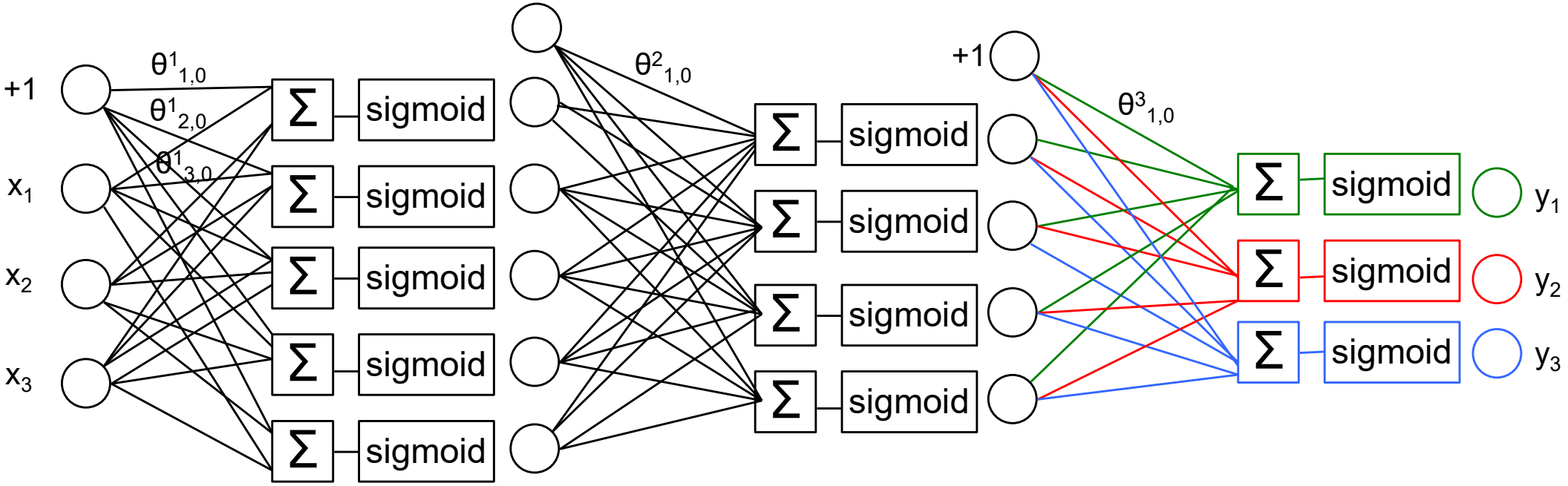
Issue:

- How to compute Gradients (partial derivatives respective to each $\theta^{(k)}_{i,j}$ (weight)) ?
- Now that there are several hidden layers…

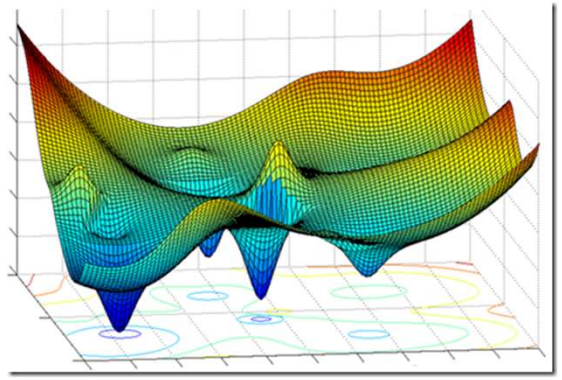
- Backpropagation (backward propagation of errors) algorithm [Rumelhart et al. 1986,
- initial ideas since 1960] estimates the gradients for each weight $\theta^{(k)}_{i,j}$ (in each layer *k*
- through a chain rule approach:
  - Feedforward the network to compute the output
  - Compute output layer units errors (deltas between predicted and actual values)
  - Propagate backward the errors (deltas) to each unit of previous layer
  - And then compute (accumulate) the gradients (relative to each weight)



2nd Approach: Hidden Layers – Limits



- Non Convex Optimization
  - Importance/Difficulty of Initialization of Weights ($\theta^{(n)}_{i,j}$)
  - Local Minimum



- Gradient Vanishing [Hochreiter 1991]
  - Backpropagation algorithm (chain rule based) to propagate gradients (from output errors)
  - Thinner Estimation (Increasing errors) of gradients while traversing layers

3rd Approach: Kernel (Trick)

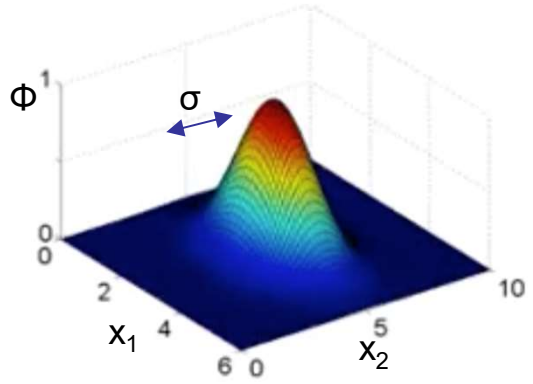
- Increase the number of Dimensions
- By a Transformation of Input Parameters into a Linearly Separable Higher Dimension Feature Space
- A Change of Representation

- This Transformation Function K is Named a *Kernel* Function

- A Kernel function $K(x^{(1)}, x^{(2)})$ represents the *Similarity* between two
- examples ($x^{(1)}$ and $x^{(2)}$)

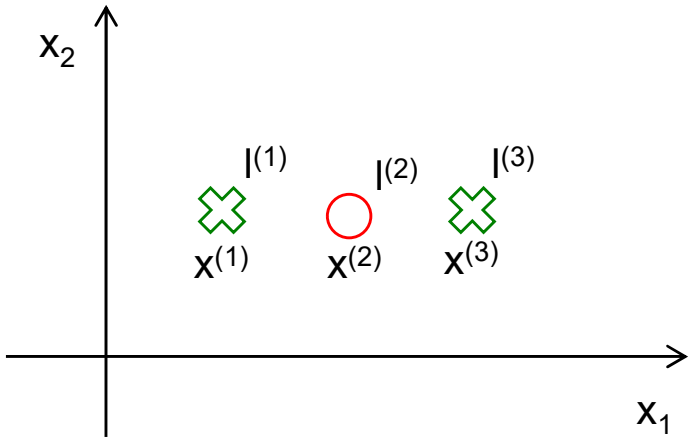
- Example of Kernel: Gaussian Kernel
  - also named Radial Basis Function (RBF) Kernel
  - $K(x^{(j)}, l^{(i)}) = \text{similarity}(x^{(j)}, l^{(i)}) = \exp(- ||x^{(j)} - l^{(i)}||^2 / 2\sigma^2)$

  - Other Kernels:
    - » Linear (Without) Kernel, Polynomial Kernel, Sigmoid Kernel…
    - » Homemade Kernel (must satisfy Mercer property [Mercer 1909])

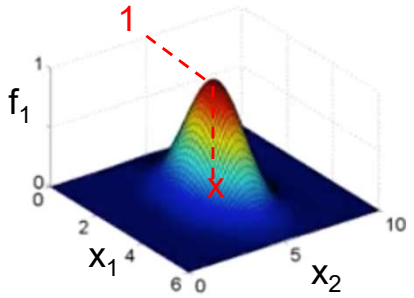


Kernel = Samples-Centered Feature Transformation

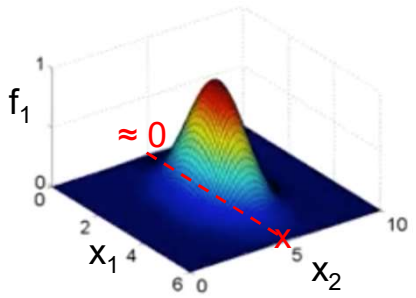
- Let's consider all (m = 3) examples $x^{(i)}$ as *Landmarks* $l^{(i)}$



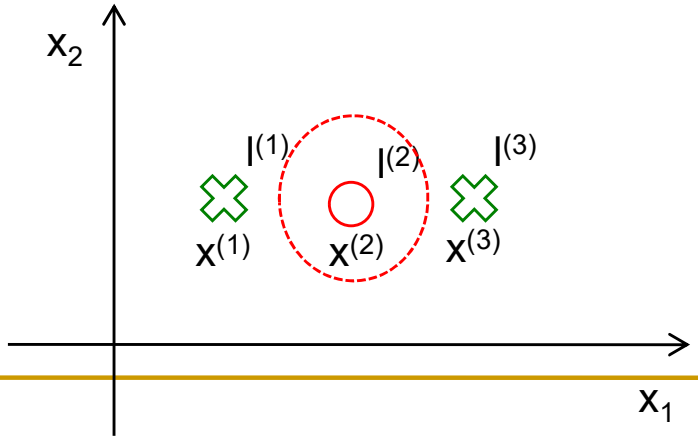
- Features are defined as following:
  - $f_1(x) = K(x, l^{(1)}) = \exp(- ||x - l^{(i)}||^2 / 2\sigma^2)$
  - $f_2(x) = K(x, l^{(2)})$
  - ...
  - $f_m(x) = K(x, l^{(m)})$

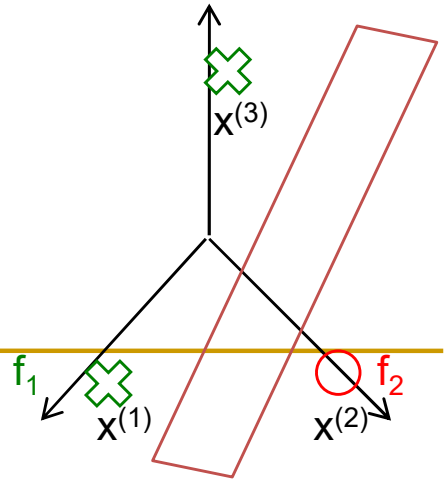


$f_1(x^{(1)}) = K(x^{(1)}, l^{(1)}) = 1$

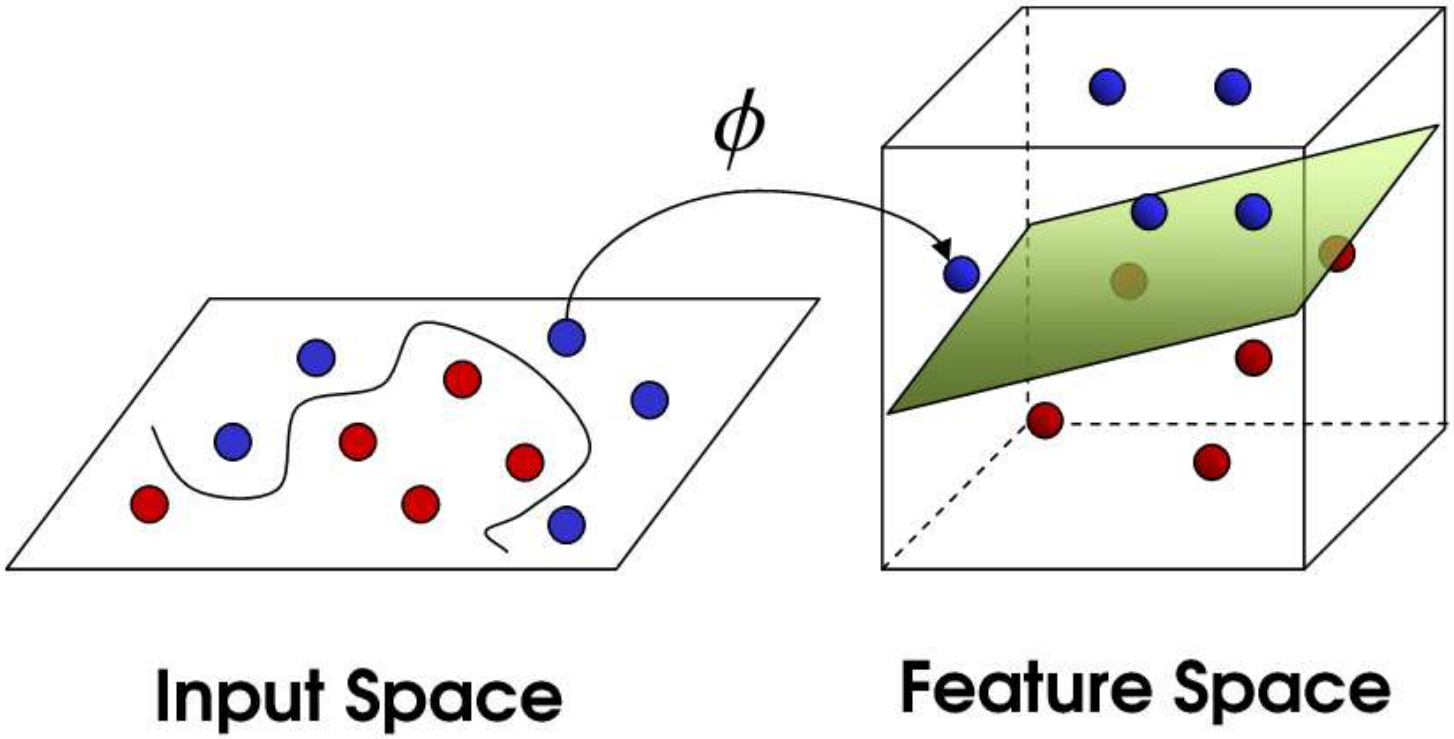


$f_1(x^{(2)}) = K(x^{(2)}, l^{(1)}) \approx 0$





Kernel Trick – Input Space x Feature Space



Kernel Trick

- Computing the Kernel is much easier (and more efficient) when it can be expressed as an inner product (in the feature space) of a kernel function f

- $K(x^{(1)}, x^{(2)}) = f(x^{(1)}).f(x^{(2)})$

- Ex: $K(x, y) = (\langle x_1, x_2 \rangle . \langle y_1, y_2 \rangle)^2$

  $= (x_1y_1 + x_2y_2)^2$

  $= x_1^2y_1^2 + 2x_1y_1x_2y_2 + x_2^2y_2^2$

  $= \langle x_1^2, \sqrt{2}\, x_1x_2, x_2^2 \rangle . \langle y_1^2, \sqrt{2}\, y_1y_2, y_2^2 \rangle$

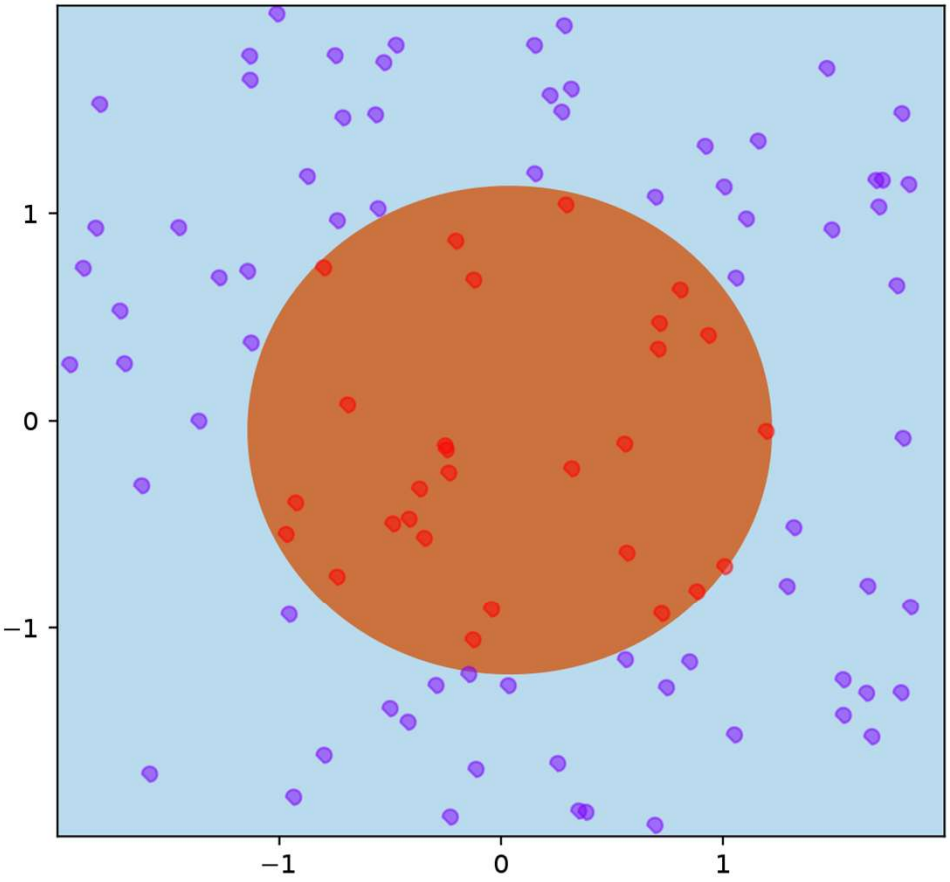
  $= f(x) . f(y)$

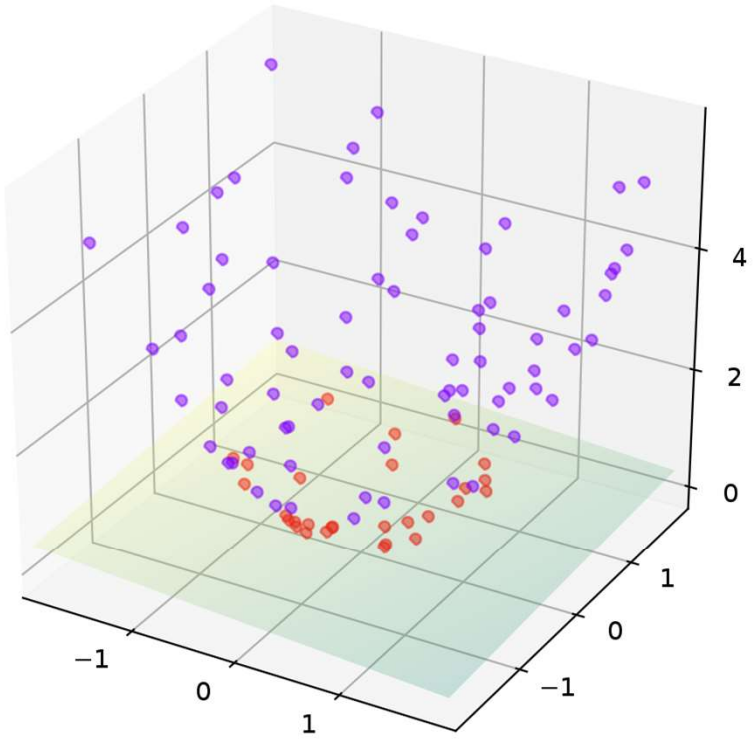
  with $f(x) = \langle x_1^2, \sqrt{2}\, x_1x_2, x_2^2 \rangle$

Mercer's Theorem: Characterization of a function as a kernel function

Kernel Trick

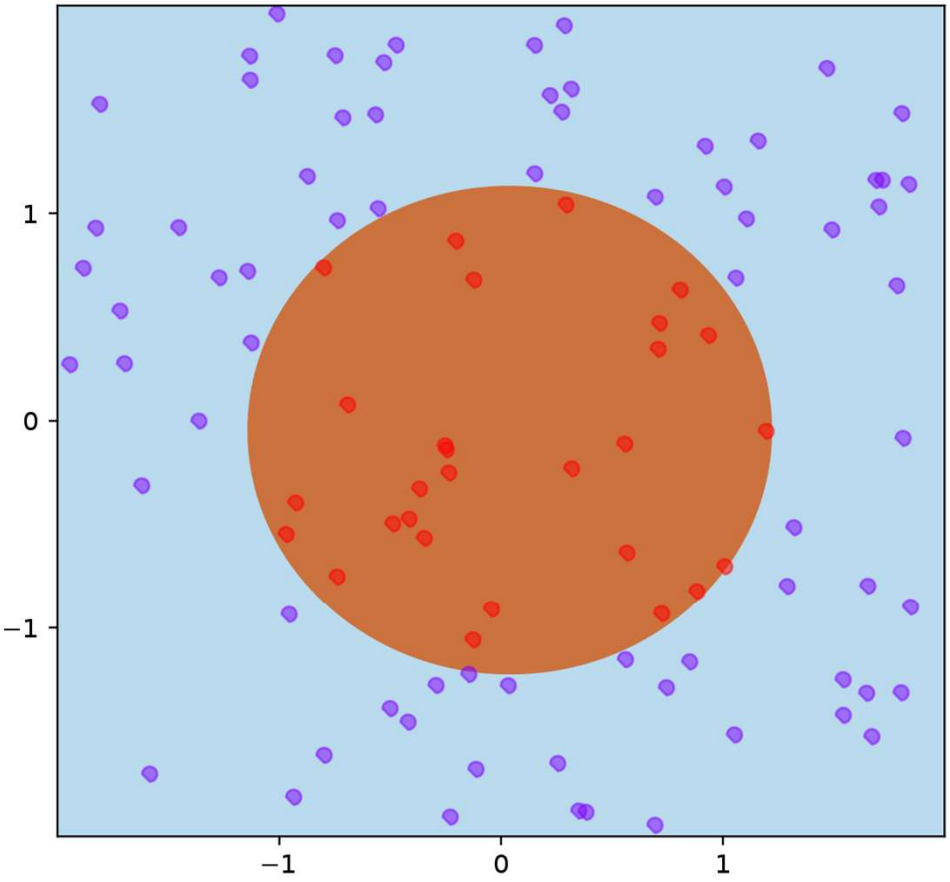
- Another example: Kernel transformation given by $\varphi((a, b)) = (a, b, a^2 + b^2)$

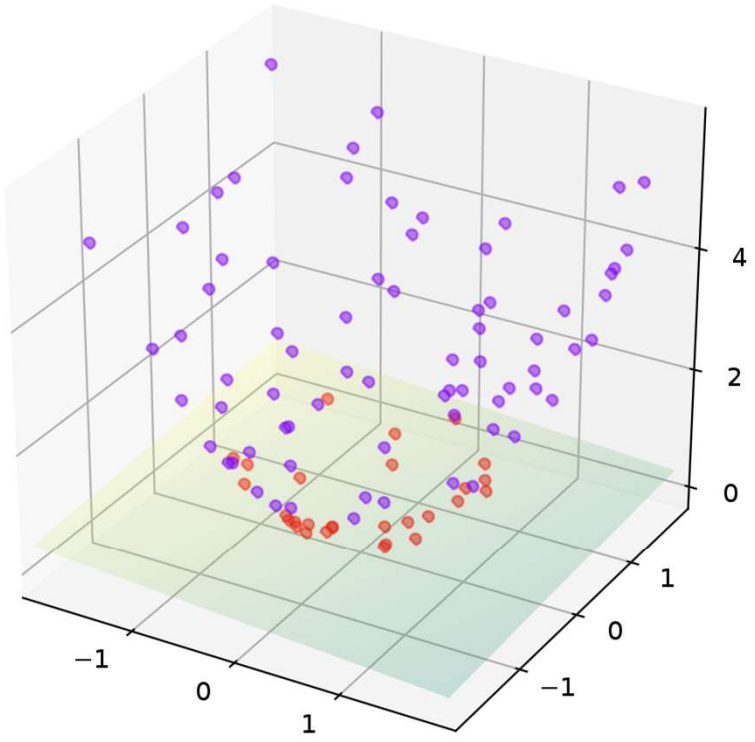




Kernel Trick

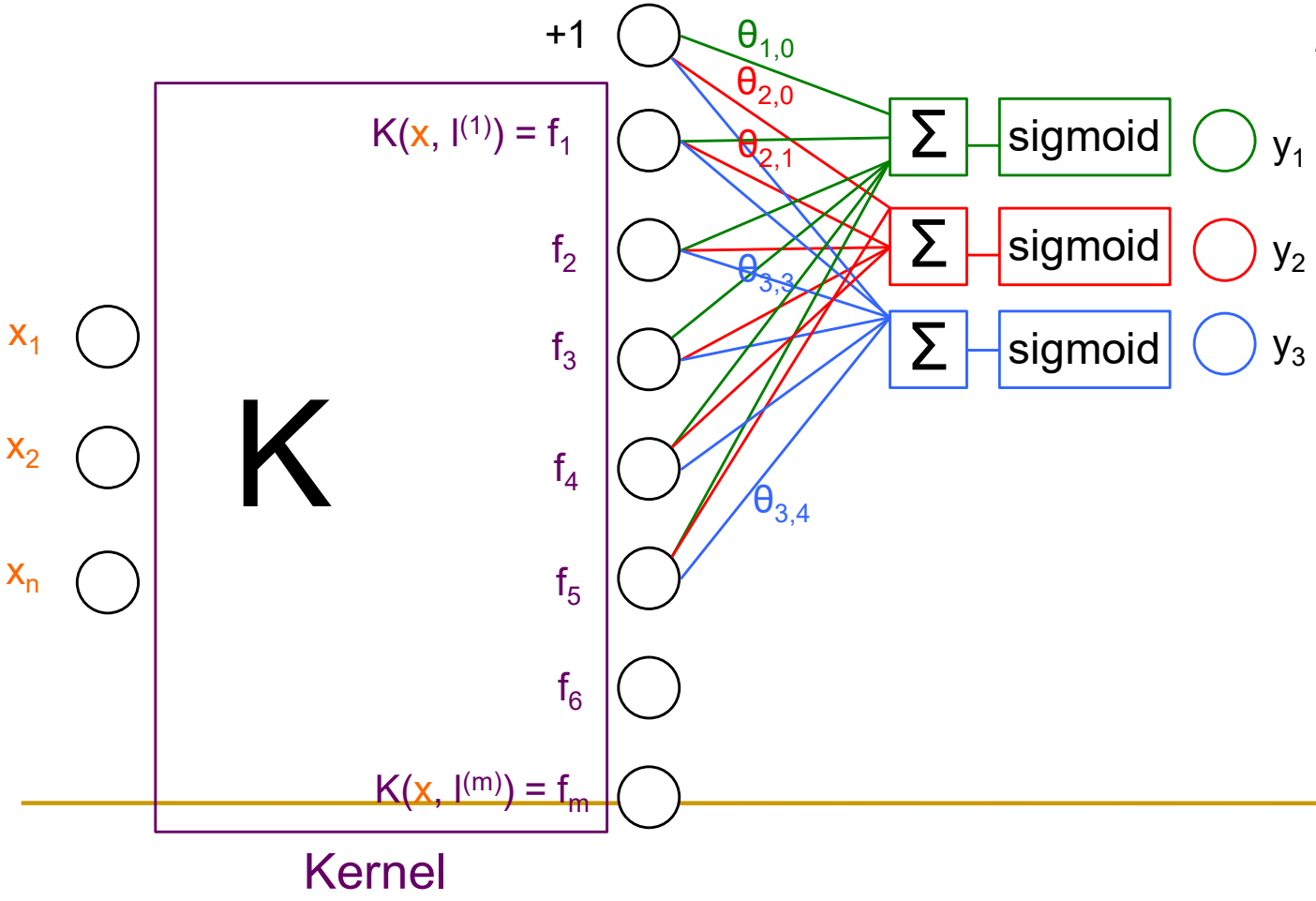
- Another example: Kernel transformation given by $\varphi((a, b)) = (a, b, a^2 + b^2)$





Kernel Trick

- The initial *n* dimension space (number of parameters) is tranformed into
- a *m* (number of Training examples) dimension space of features (new parameters), linearly separable
- This new space is independent from the number of parameters (*n*) and so depending on the number of training examples (*m*)



Note:

There are 2 phases:

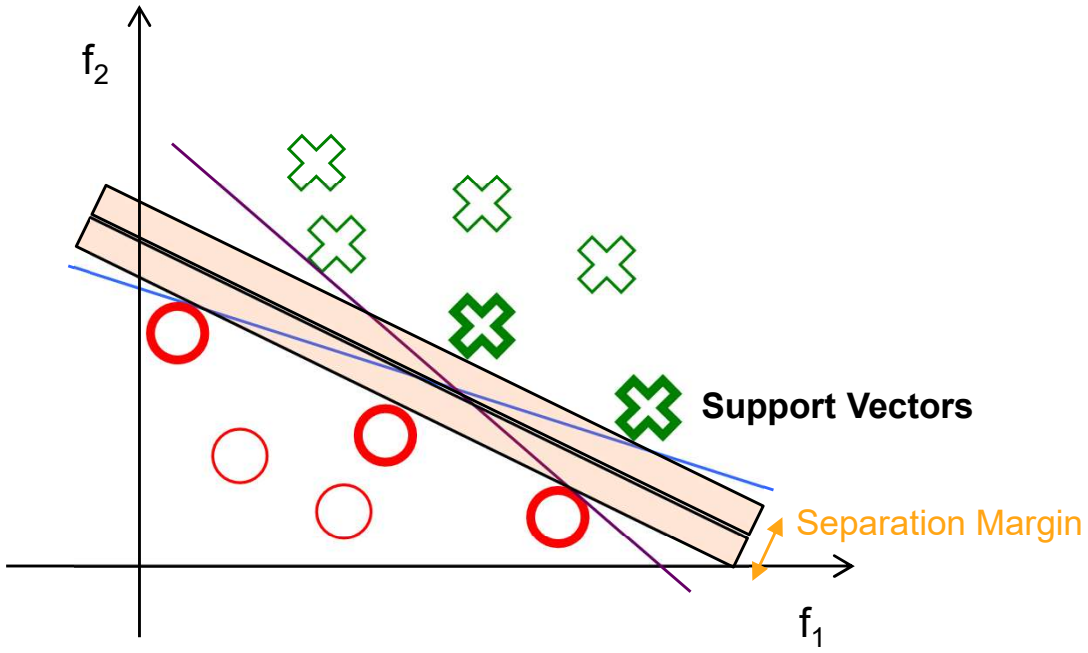
- *Configuration of the Kernel component with Training examples – landmarks, $l^{(1)}, \ldots l^{(m)}$ are set and stored into the Kernel component*
- *Application of the Kernel component as a dimension transformation on examples (X -> F)*

Support Vector Machine

- The same Kernel Trick is used by Support Vector Machines
- Support Vector Machine [Boser et al. 1992] [Vapnik 1995]

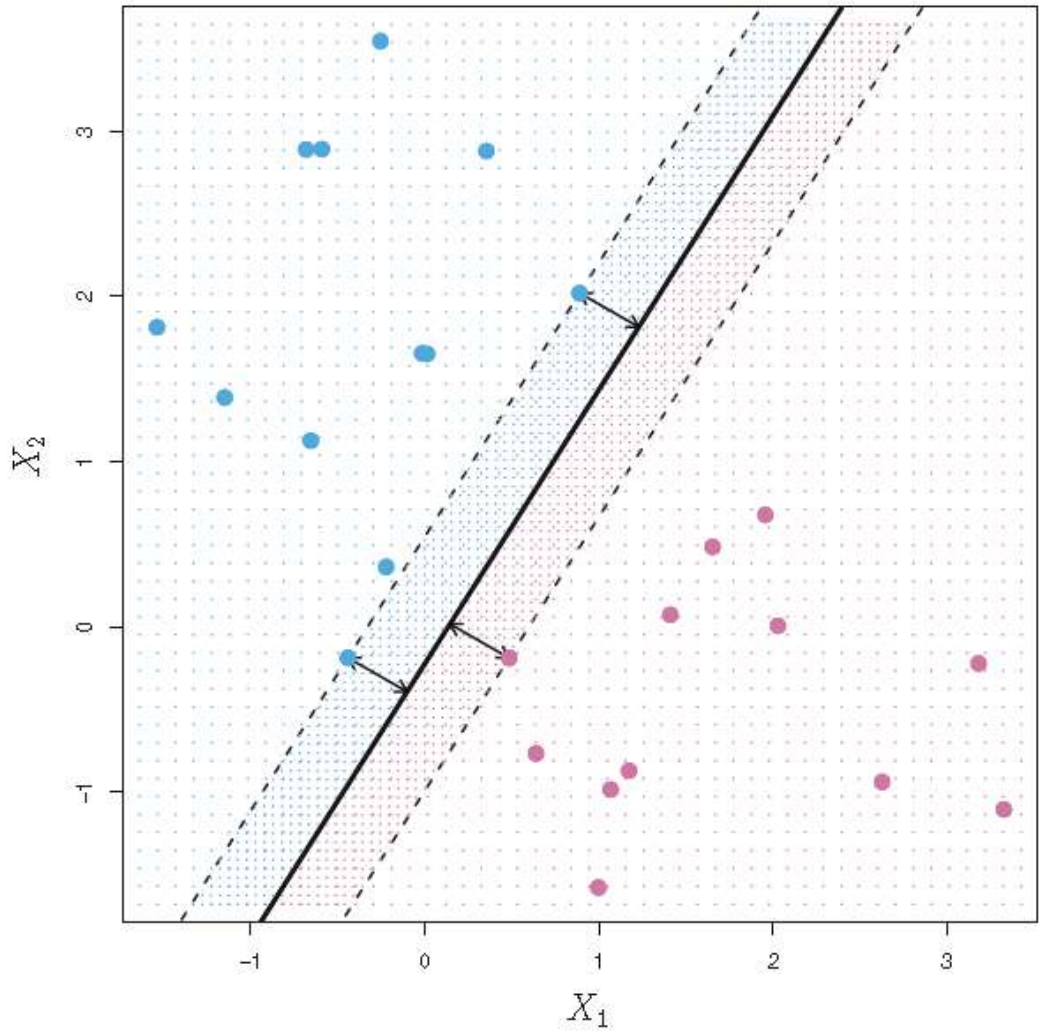
  = Kernel [Mercer 1909] + Linear Support Vector Machine [Vapnik & Lerner 1963]

- A Linear Support Vector Machine is good at maximizing the Separation Margin



Support Vector Machine

- From 2000's, SVMs outperformed Neural Networks and led to the decrease of research in Neural Networks
- Until the 2010's, when they returned through the idea of Deep Networks/Learning



Referências

1. Capítulo 5 do livro *“Deep Learning Techniques for Music Generation”* de Jean-Pierre Briot, Gaëtan Hadjeres e François-David Pachet.

2. Capítulo 13 (Seções 13.1, 13.2 e 13.3) do livro “*Mining of Massive Datasets*” de Jure Leskovec, Anand Rajaraman e Jeffrey David Ullman.