WELCOME

Introduction to Machine Learning

DBDA.X408.(33)

Instructor:

Bill Chen

UCSC Silicon Valley Extension
PROFESSIONAL EDUCATION

UCSC Silicon Valley Extension

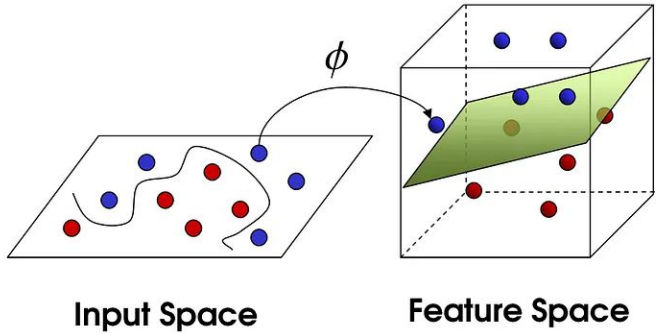
E: xch375@ucsc.edu

Week 7

Neural Network.

Backprop.

SVM Kernels.

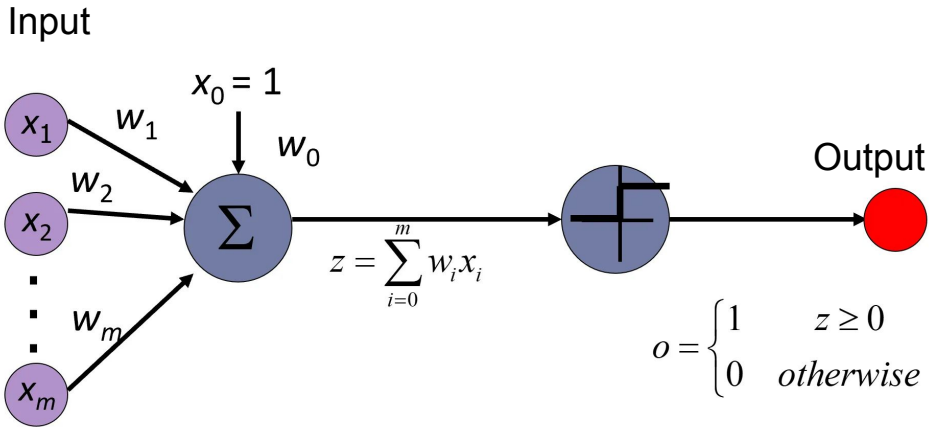


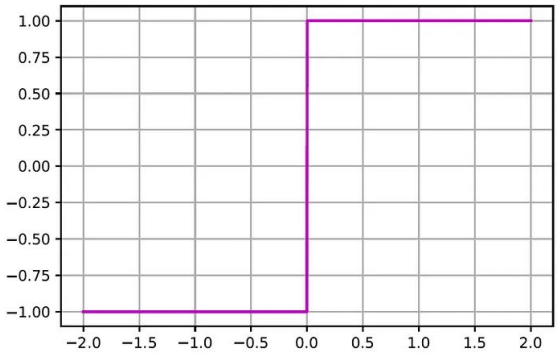
Neural Network

Perceptron Model

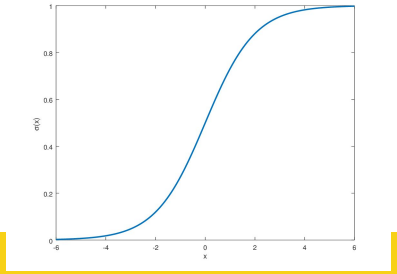
Each input neuron x_i is connected to the perceptron via a link whose strength is represented by a weight w_i. Inputs with higher weights have a larger influence on the perceptron output.

More generally, the perceptron applies an activation function f(z) on the net input that generates its output.

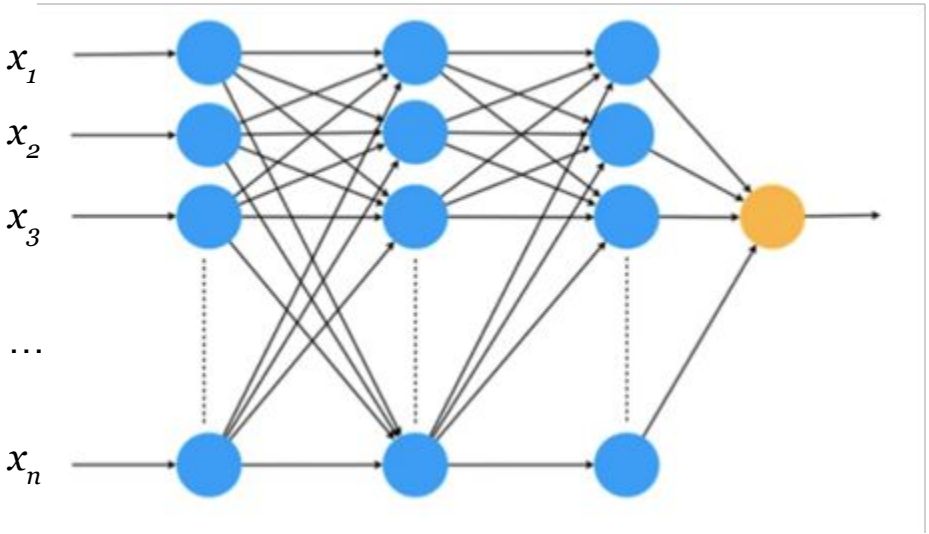




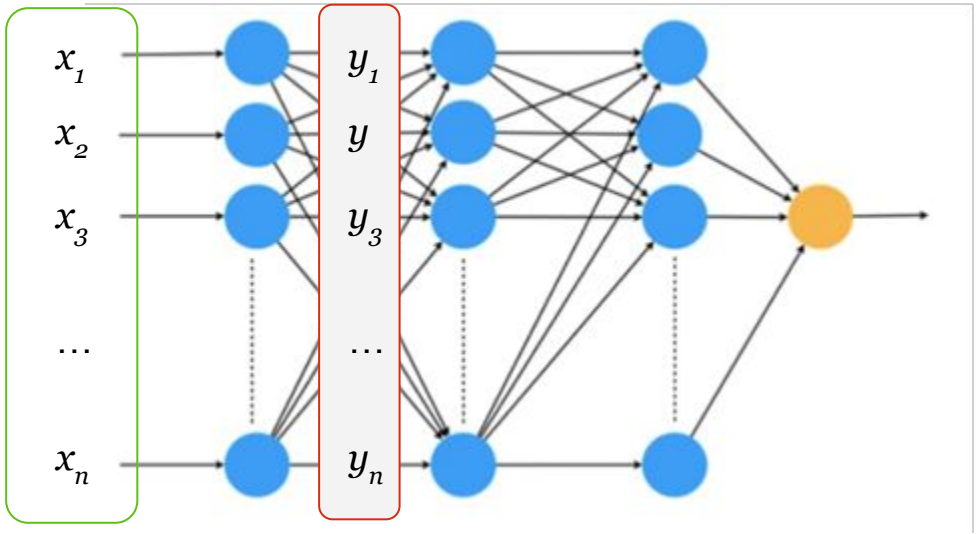
The sign function is a function whose value is -1 for negative inputs and 1 for non-negative inputs. We can also use sigmoid function.



Neural Model: Stack Neurons Together by Layers

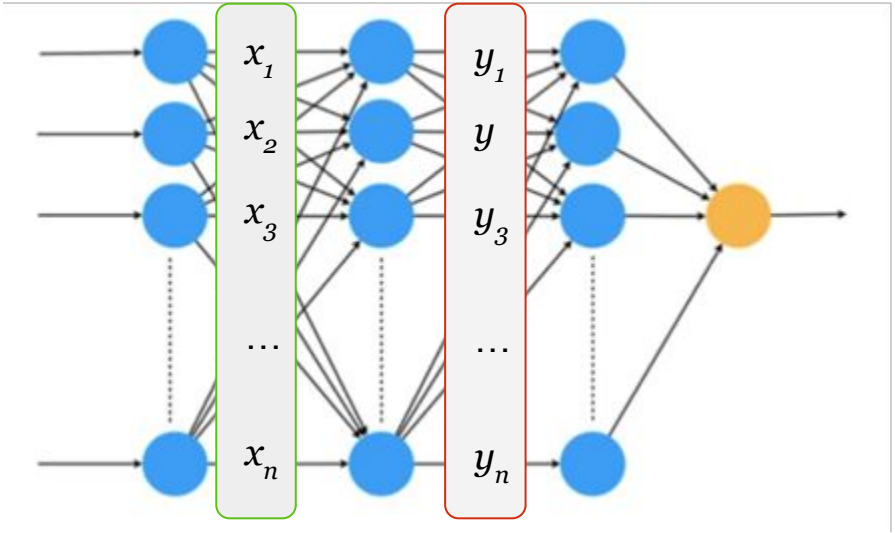


Neural Model: Stack Neurons Together by Layers

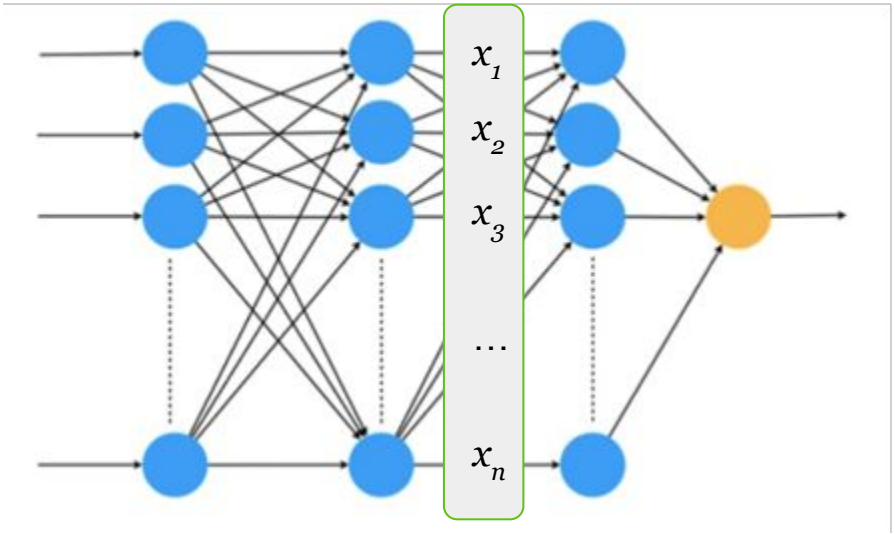


Inputs Outputs

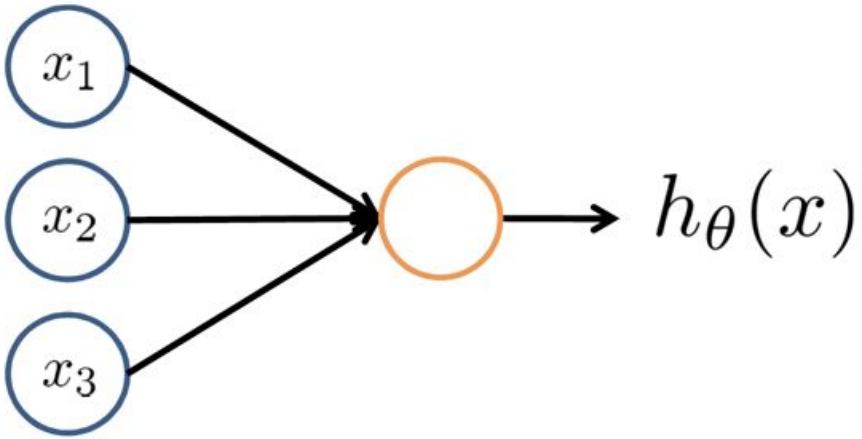
Neural Model: Stack Neurons Together by Layers



Neural Model: Stack Neurons Together by Layers



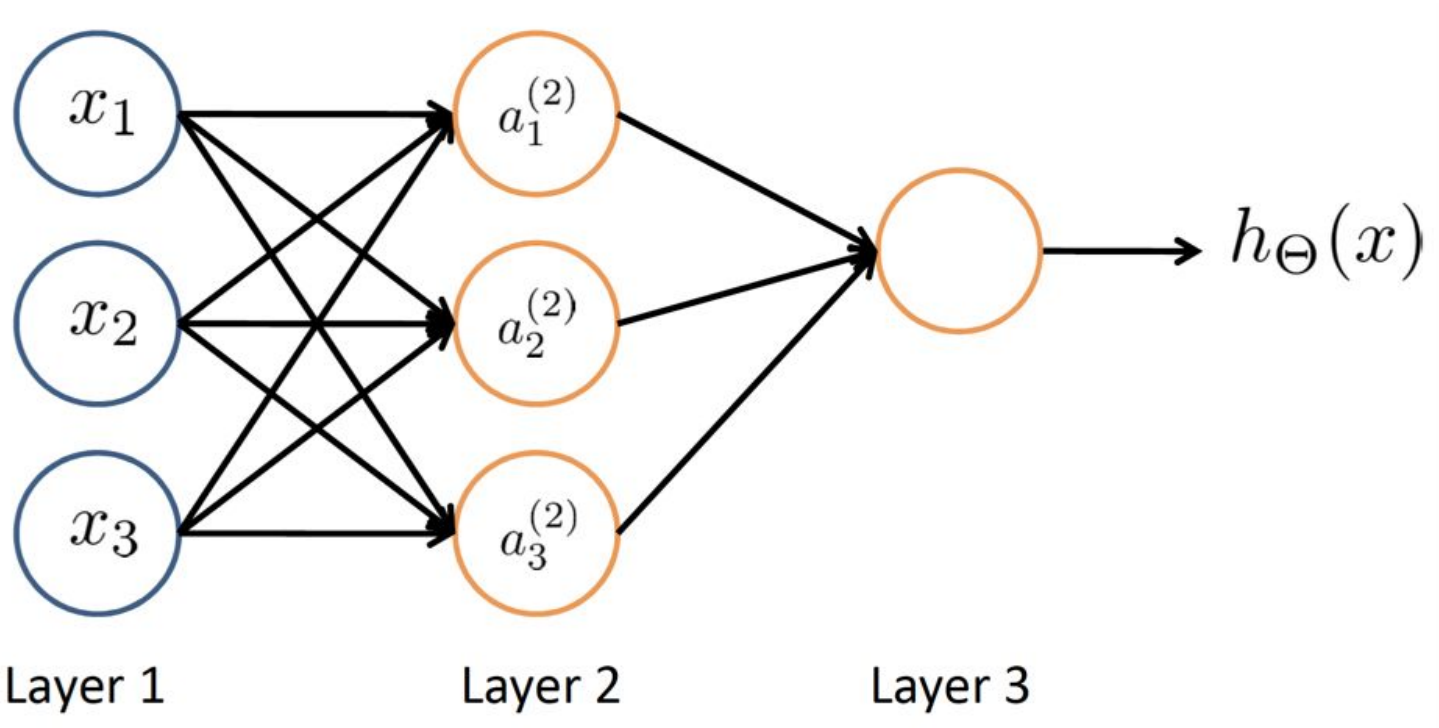
Neuron Model:



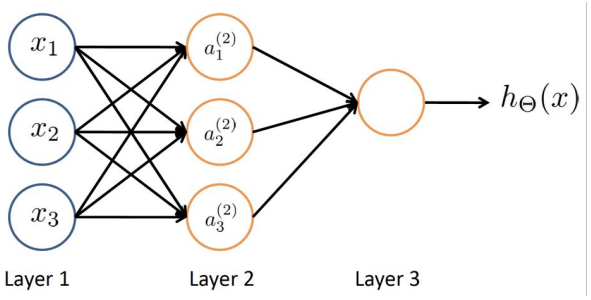
$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

Sigmoid (logistic) activation function.

Neural Network



Neural Network



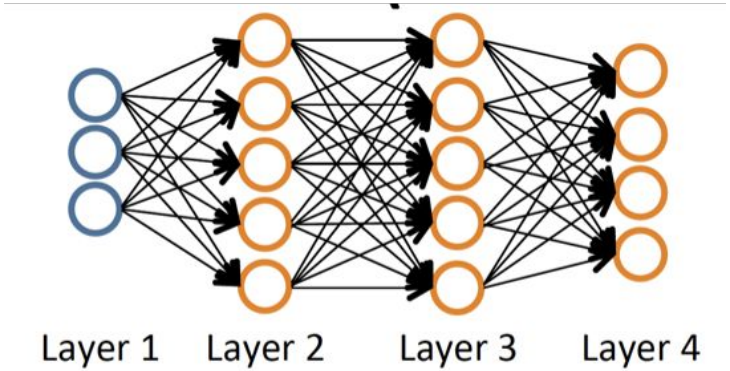
$a_i^{(j)}$ = "activation" of unit i in layer j

$\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$

$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

If network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$.

Neural Network - Classification



$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$$

$L =$ total no. of layers in network

$s_l =$ no. of units (not counting bias unit) in layer l

Binary Classification

$y = 0$ or 1

1 output unit

Multi-class Classification (k classes)

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

k output units

Cost Function

Logistic Regression:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

Neural Networks:

$$h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th} \text{ output}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log((h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right]$$

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

Backpropagation

Backpropagation Algorithm

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right]$$

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

$$\min_{\Theta} J(\Theta)$$

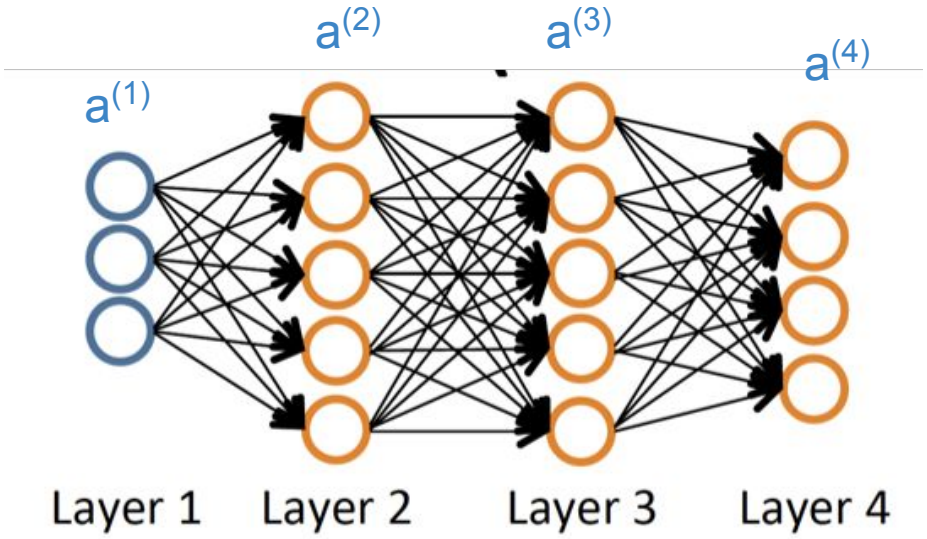
Need code to compute:

1. $J(\Theta)$
2. $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

Gradient Computation

Given one training example (x, y):

$$
\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)}) \quad (\text{add } a_0^{(2)}) \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
a^{(3)} &= g(z^{(3)}) \quad (\text{add } a_0^{(3)}) \\
z^{(4)} &= \Theta^{(3)} a^{(3)} \\
a^{(4)} &= h_\Theta(x) = g(z^{(4)})
\end{aligned}
$$



Gradient Computation: Backpropagation Algorithm

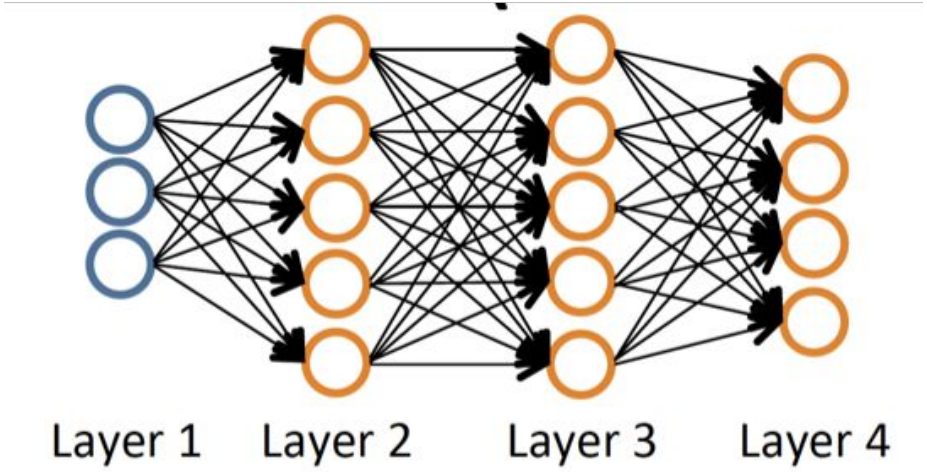
Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.

For each output unit (layer L = 4)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$



Backpropagation Algorithm

Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j)

For i = 1 to m

Set $a^{(1)} = x^{(i)}$

Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

Using $y^{(i)}$, compute $\delta^{(l)} = a^{(L)} - y^{(i)}$

Compute $\delta^{(l-1)}, \delta^{(l-2)}, \ldots \delta^{(2)}$

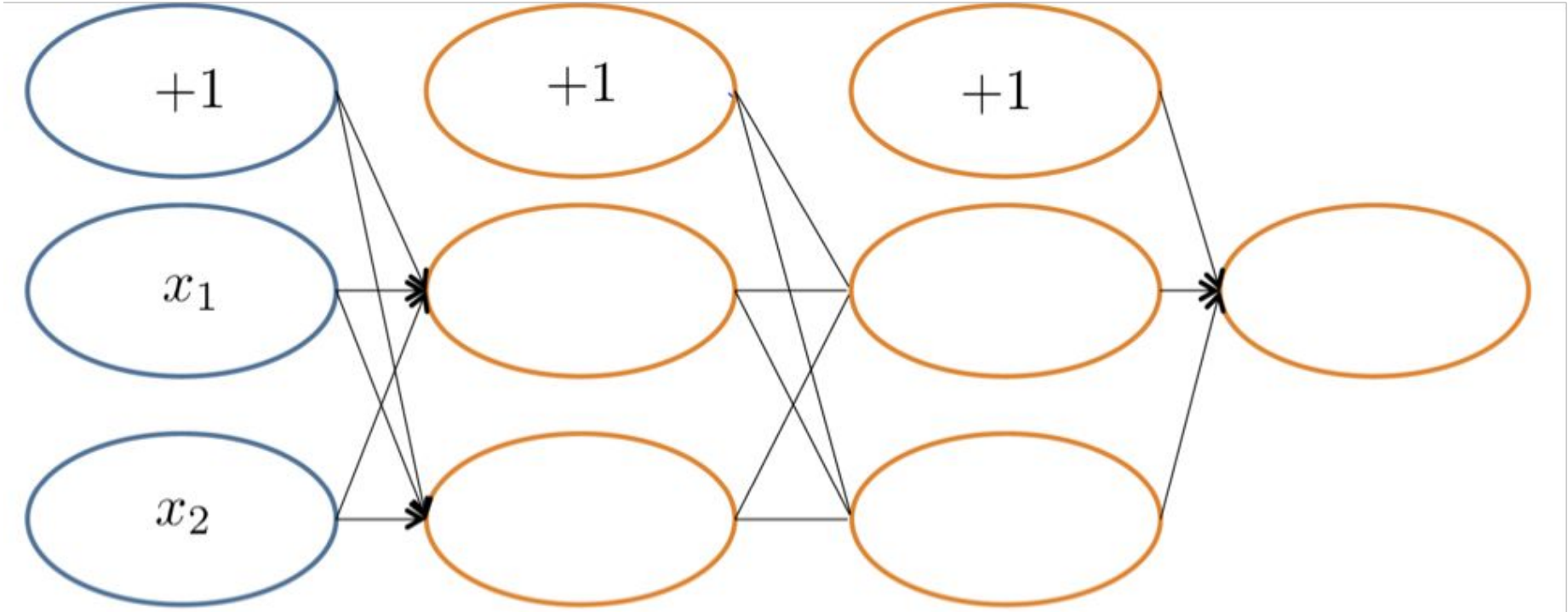
$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \text{ if } j \neq 0$$

$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} \text{ if } j = 0$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

Forward Propagation



Backpropagation

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right]$$

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

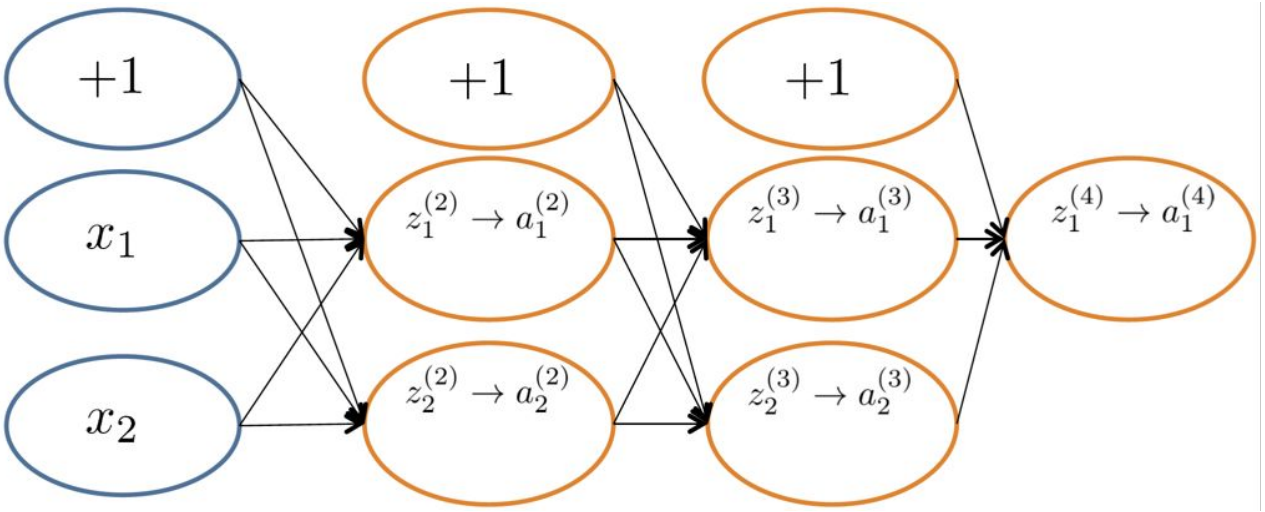
Focusing on a single example $x^{(i)}$, $y^{(i)}$, the case of 1 output unit, and ignoring regularization ($\lambda = 0$),

$$\text{cost(i)} = \; y^{(i)} \log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - (h_\Theta(x^{(i)})))$$

(Think of $\text{cost(i)} \approx (h_\Theta(x^{(i)}) - y^{(i)})$

I.e. how well is the network doing on example i?

Forward Propagation

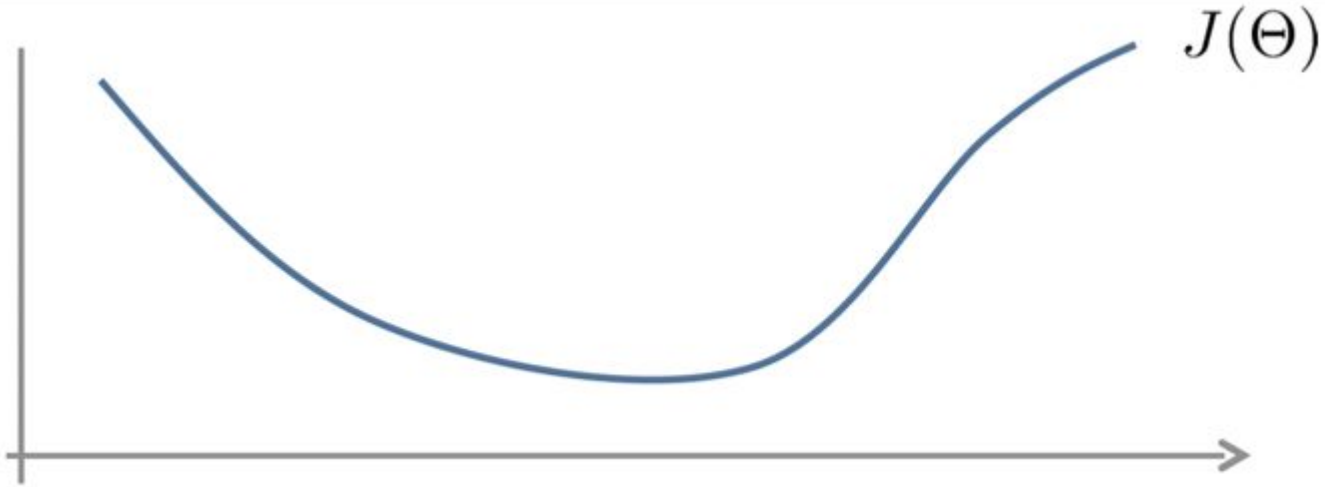


$\delta_j^{(l)}$ = "error" of cost for aj(l) (unit j in layer l).

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost(i)}$ (for j >=0), where

$$\text{cost(i)} = y^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log h_\Theta(x^{(i)})$$

Numerical Estimation of Gradients



Parameter Vector θ

$\theta \in \mathbb{R}^n$ (E.g. θ is "unrolled" version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$

$\theta = \theta_1, \theta_2, \theta_3, \ldots, \theta_n$

$$\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1+\epsilon,\theta_2,\theta_3,\ldots,\theta_n)-J(\theta_1-\epsilon,\theta_2,\theta_3,\ldots,\theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1,\theta_2+\epsilon,\theta_3,\ldots,\theta_n)-J(\theta_1,\theta_2-\epsilon,\theta_3,\ldots,\theta_n)}{2\epsilon}$$

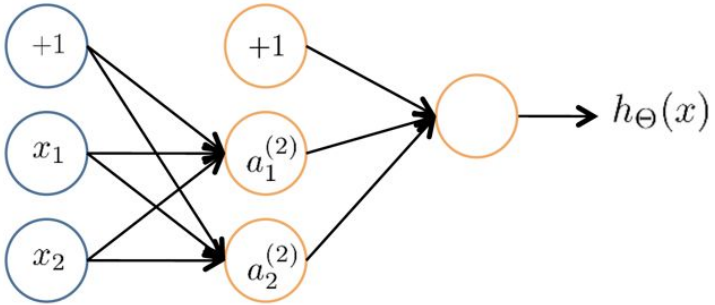
$$\vdots$$

$$\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1,\theta_2,\theta_3,\ldots,\theta_n+\epsilon)-J(\theta_1,\theta_2,\theta_3,\ldots,\theta_n-\epsilon)}{2\epsilon}$$

Initial Value of θ

For gradient descent and advanced optimization method, need initial value of θ.

Consider gradient descent, set the initial θ to zeros?



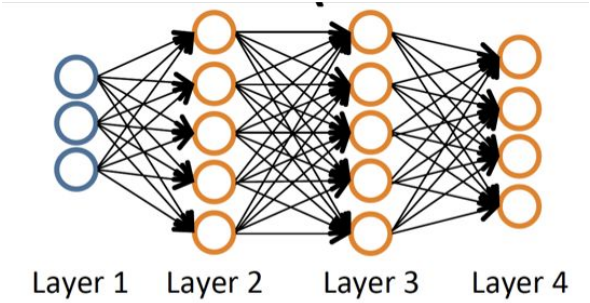
After each update, parameters corresponding to inputs going into each of two hidden units are identical.

Random Initialization: Symmetry Breaking

Initialize each $\theta_{ij}^{(l)}$ to a random value in $[-\epsilon,\epsilon]$ (i.e. $-\epsilon \leq \theta_{ij}^{(l)} \leq \epsilon$)

Put Together

Pick a network architecture (connectivity pattern between neurons)



No. of input units: Dimension of features $x^{(i)}$.

No. of output units: Number of classes.

Reasonable default: 1 hidden layer, or if >1 hidden layer, have same no. of hidden units in every layer (usually the more the better)

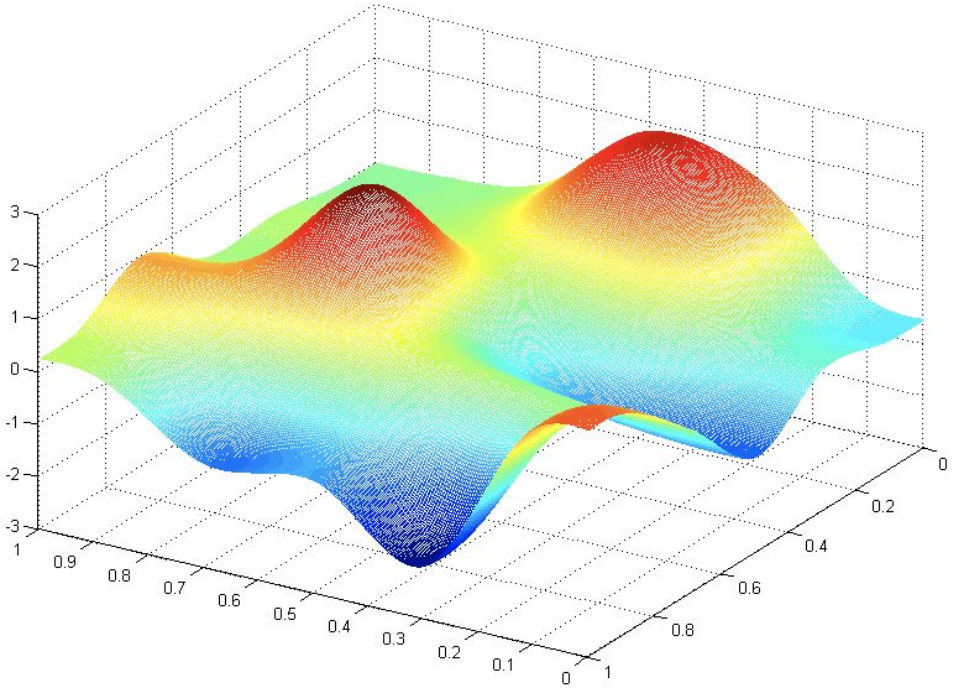
Training a Neural Network

1. Randomly initialize weights
2. Implement forward propagation to get $h_\theta(x^{(i)})$ for any $x^{(i)}$.
3. Implement code to compute cost function $J(\theta)$.
4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$
   for `i = i:m`

   Perform forward propagation and backpropagation using example ($x^{(i)}$, $y^{(i)}$)

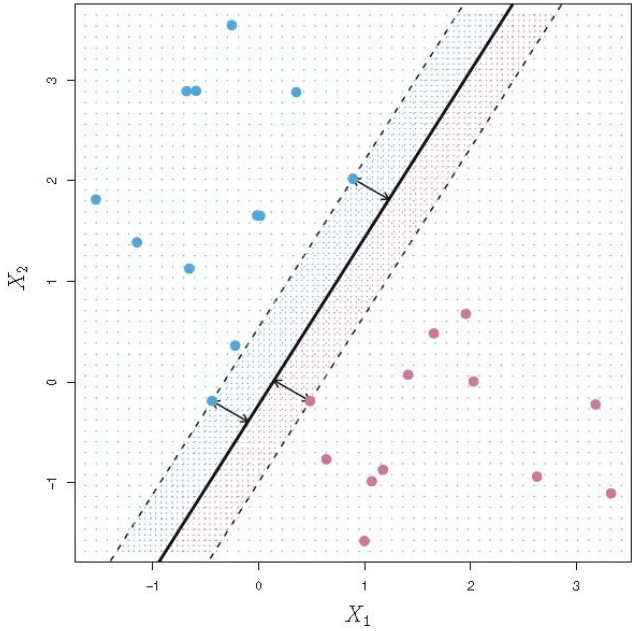
   (Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, ..., L$).
5. Using gradient checking to compute $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backpropagation vs. using numerical estimate of gradient of $J(\theta)$.
6. Using gradient descent or advanced optimization method with backpropagation to try to minimize $J(\theta)$ as a function of parameter θ.

Backpropagation Visualization



SVM Kernel Trick

SVM: Review



The support vectors are the points on the dashed lines. The distance from the dashed line to the solid line is the margin, represented by the arrows.

Equation of a Hyperplane in 2 dimensions

$$\beta_0 + \beta_1 \mathbf{X_1} + \beta_2 \mathbf{X_2} = 0$$

Equation of a Hyperplane in p-dimensions

$$\beta_0 + \beta_1 \mathbf{X_1} + \ldots + \beta_p \mathbf{X_p} = 0$$

We have points on the hyperplane:

$$\mathbf{X} = [\mathbf{X_1}, \ldots, \mathbf{X_p}]^T$$

where $\mathbf{X}$ is a feature vector

We can use this to make classifications by considering one class defined by:

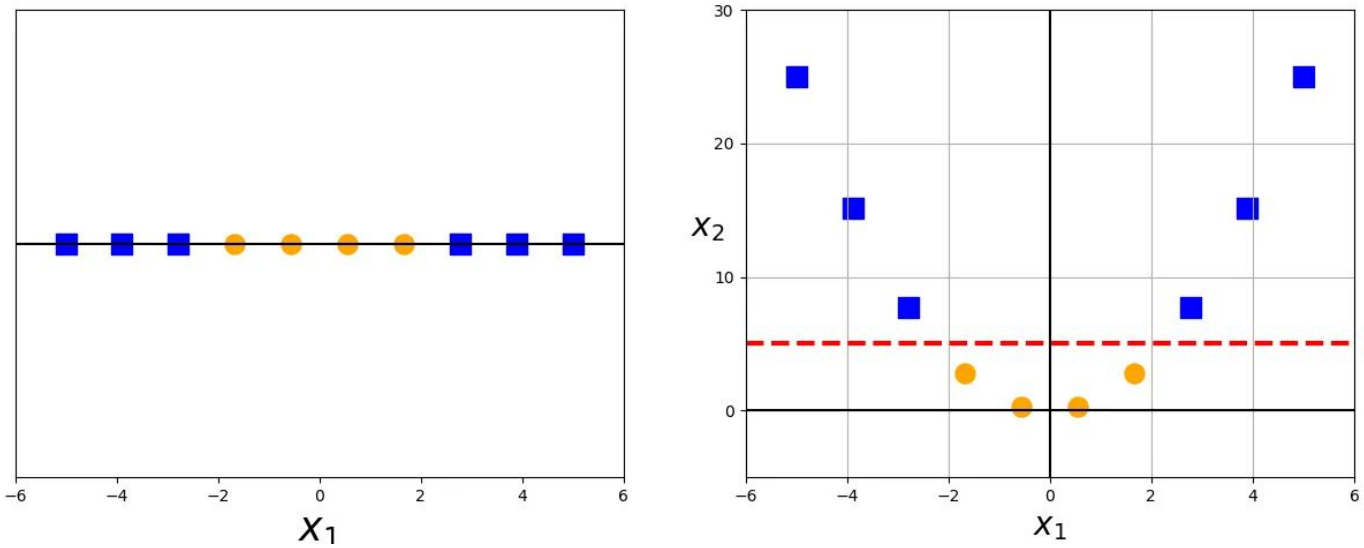
$$\beta_0 + \beta_1 \mathbf{X_1} + \ldots + \beta_p \mathbf{X_p} > 0$$

With the other class defined by:

$$\beta_0 + \beta_1 \mathbf{X_1} + \ldots + \beta_p \mathbf{X_p} < 0$$

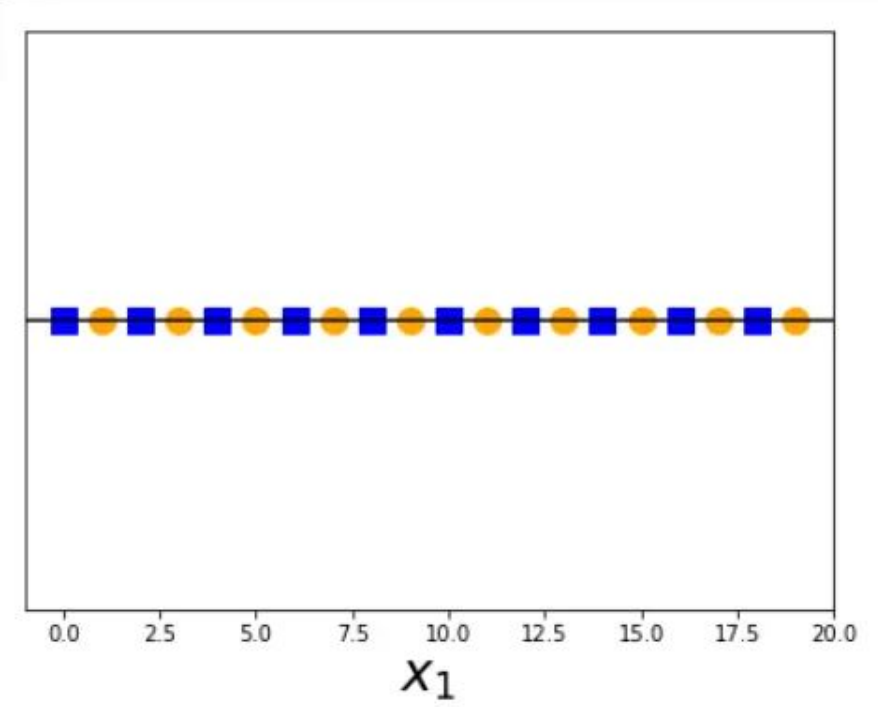
Support vector classification relies on this notion of linearly separable data. “Soft margin” classification can accommodate some classification errors on the training data, in the case where data is not perfectly linearly separable. However, in practice data is often very far from being linearly separable, and we need to transform the data into a higher dimensional space in order to fit a support vector classifier.

SVM: Non-linear Transformations



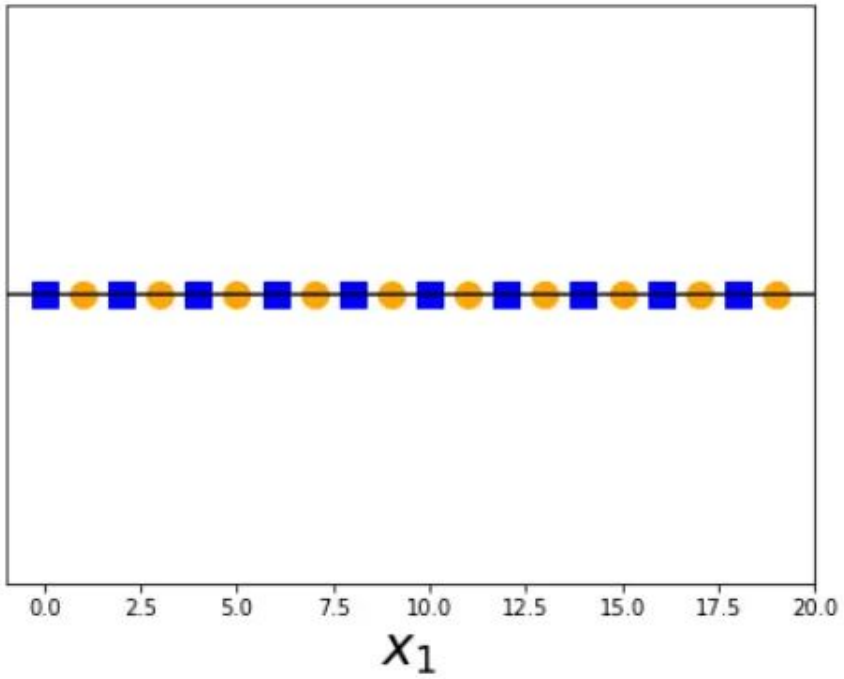
In 1-dimension, this data is not linearly separable, but after applying the transformation $\phi(x) = x^2$ and adding this second dimension to our feature space, the classes become linearly separable

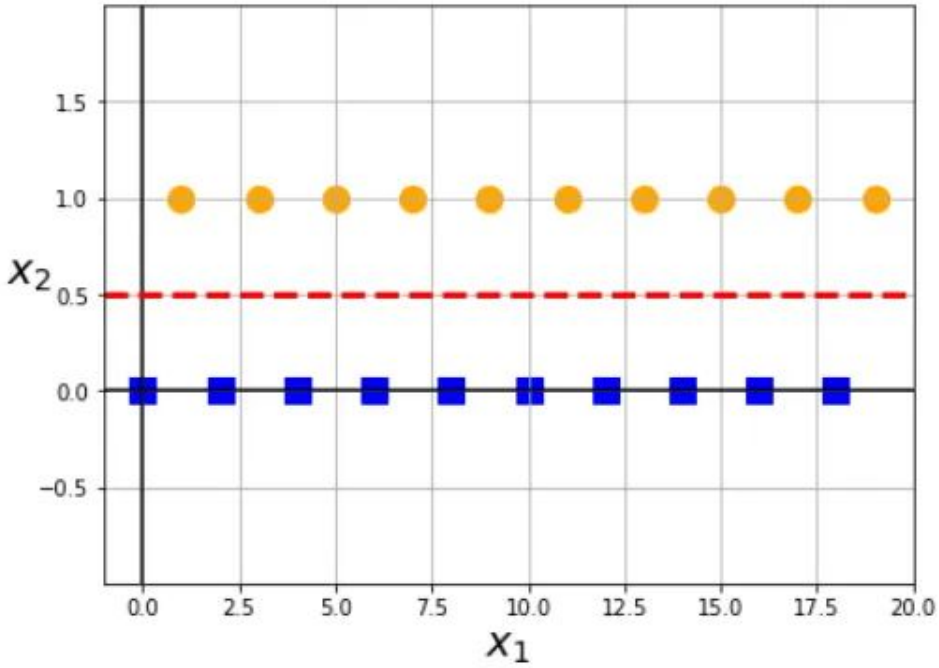
SVM: Non-linear transformations



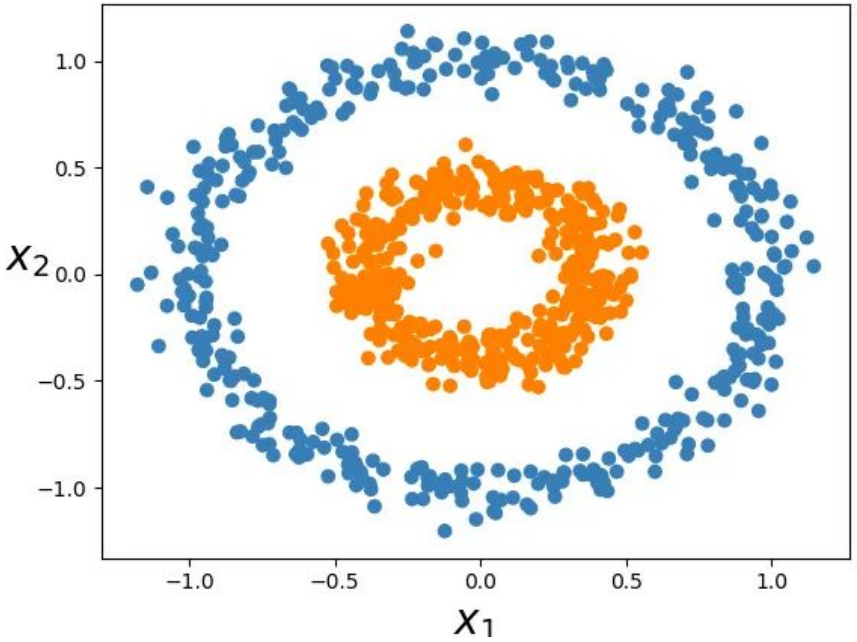
SVM: Non-linear transformations

$\phi(x) = x \bmod 2$



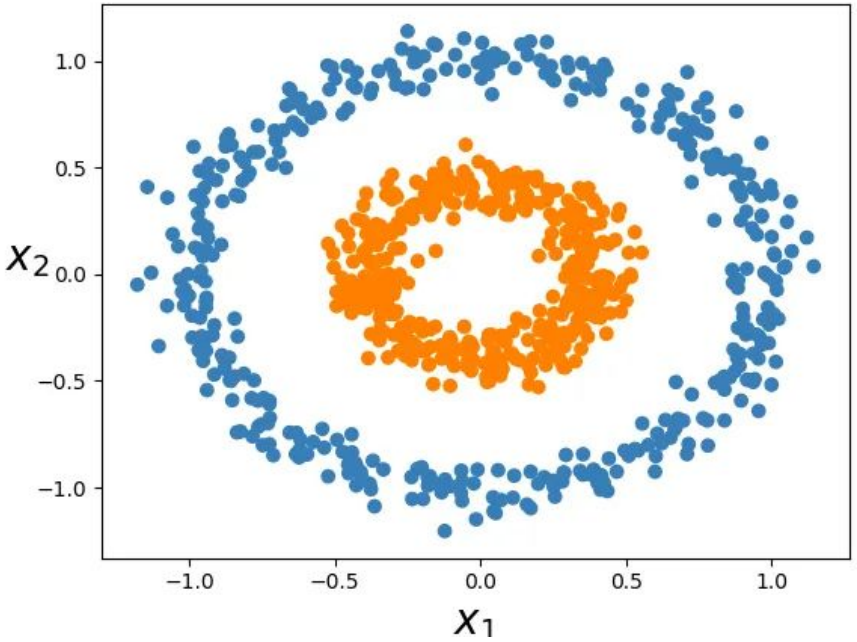


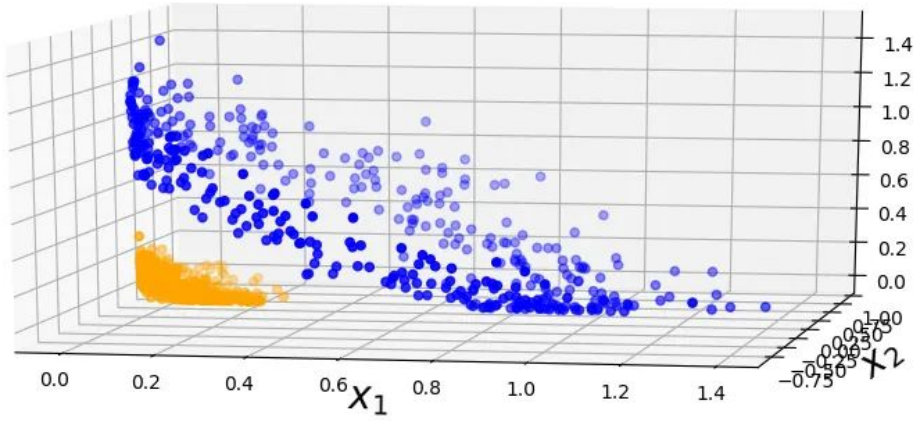
SVM: Non-linear Transformations



SVM: Non-linear Transformations

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2}\,x_1 x_2 \\ x_2^2 \end{pmatrix}$$





The Kernel Trick

It seems that in order to train a support vector classifier and optimize our objective function, we would have to perform operations with the higher dimensional vectors in the transformed feature space. In real applications, there might be many features in the data and applying transformations that involve many polynomial combinations of these features will lead to extremely high and impractical computational costs.

The kernel trick provides a solution to this problem. The “trick” is that kernel methods represent the data only through a set of pairwise similarity comparisons between the original data observations x (with the original coordinates in the lower dimensional space), instead of explicitly applying the transformations $\phi(x)$ and representing the data by these transformed coordinates in the higher dimensional feature space.

Kernel Definition

A function that takes as its inputs vectors in the original space and returns the dot product of the vectors in the feature space is called *kernel function*.

More formally, if we have data $x, z \in X$ and a map $\phi: X \rightarrow \mathcal{R}^N$ then

$$k(x, z) = \langle \phi(x), \phi(z) \rangle$$

is a kernel function.

There are also theorems which guarantee the existence of such kernel functions under certain conditions.

$$\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) = \begin{pmatrix} a_1^2 \\ \sqrt{2}\, a_1 a_2 \\ a_2^2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1^2 \\ \sqrt{2}\, b_1 b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2 a_1 b_1 a_2 b_2 + a_2^2 b_2^2$$

$$= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = \left(\mathbf{a}^T \cdot \mathbf{b} \right)^2$$

Quizzes and Assignments