MACHINE INTELLIGENCE

Instance based learning

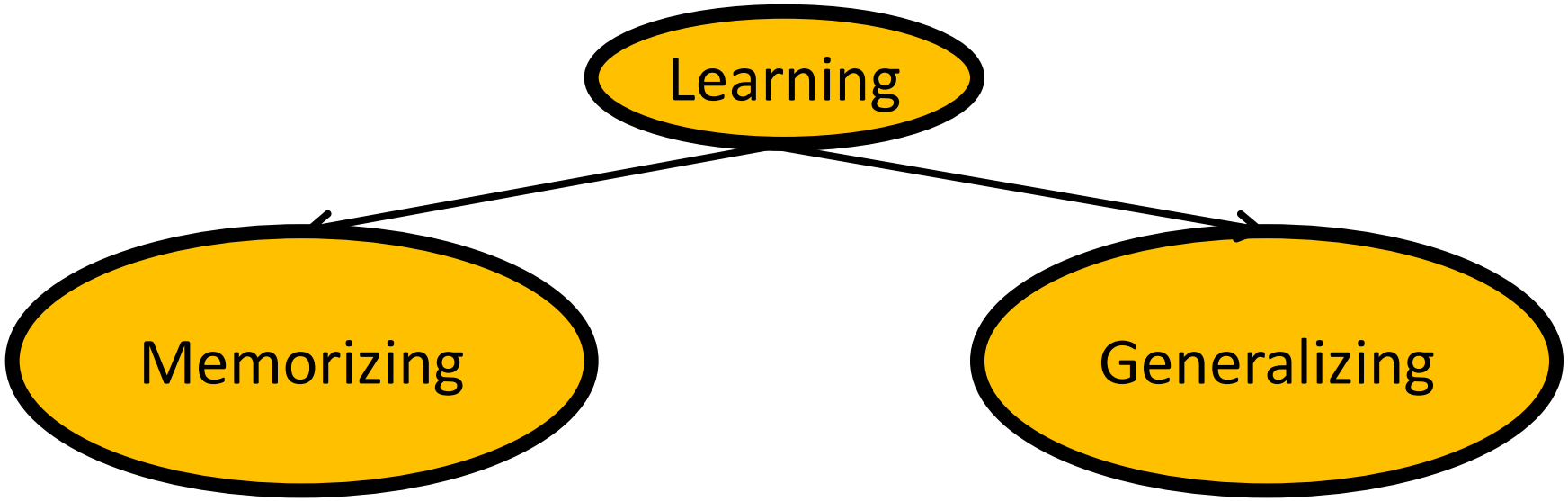
K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

Instance based learning

K.S.Srinivas
Department of Computer Science and Engineering

Instance Based learning



Instance based learning does Memorizing instead of Generalizing

Instance based learning

Lazy vs Eager Learning

Eager Learning	Lazy Learning
Simply stores training data (or does only minor processing) and waits until it is given a test tuple	Given a set of training set, constructs a classification model before receiving new (e.g., test) data to classify
Instance based Learning	Decision Tree learning

Lazy Learners

- less time in training but more time in predicting
- Lazy method effectively uses a **richer hypothesis space** since it uses **many local linear functions** to form its implicit global approximation to the target function
- Eager learning must commit to a **single hypothesis** that covers the entire instance space

Instance Based learning

Instance based Learning does storing of training examples and delay the processing ("lazy evaluation") until a new instance must be classified.
Typical Approaches are:

_k_-nearest neighbor approach

Instances represented as points in a Euclidean space.

Locally weighted regression

Constructs local approximation

Case-based reasoning

Uses symbolic representations and knowledge-based inference

The K-Nearest Neighbour Algorithm

- All instances correspond to points in the n-D space
- The nearest neighbor are defined in terms of distance measure, dist($\mathbf{X}_1$, $\mathbf{X}_2$)
- Target function could be **discrete or real valued**
- For **discrete-valued**, *k*-NN returns the **most common value** among the *k* training examples nearest to x_q
- k-NN for **real-valued prediction** for a given unknown tuple returns the **mean values of the *k* nearest neighbors**

Point to note about K

- K must be odd(ex. K=3,5,7.....),Why??
- we will keep this question for now and find its answer while solving a problem

- **Small k:**

Small K captures structure of problem space better. May be necessary for small training set. May be prone to noise.

- **Large K :**

1. Less sensitive to noise(particularly for class noise)
2. Gives better probability estimates for discrete classes.
3. Large training set allows use of larger K.

If K=1 the space is divided into several regions called as Veronoi Partition space.

The Algorithm

- **Training algorithm:**

  For each training example (x, f (x)), add the example to the list training examples.

- **Classification algorithm**:

Given a query instance xq to be classified

1. Let $X_1 \ldots X_k$ denote the k instances from training examples that are nearest to x_q
2. Return

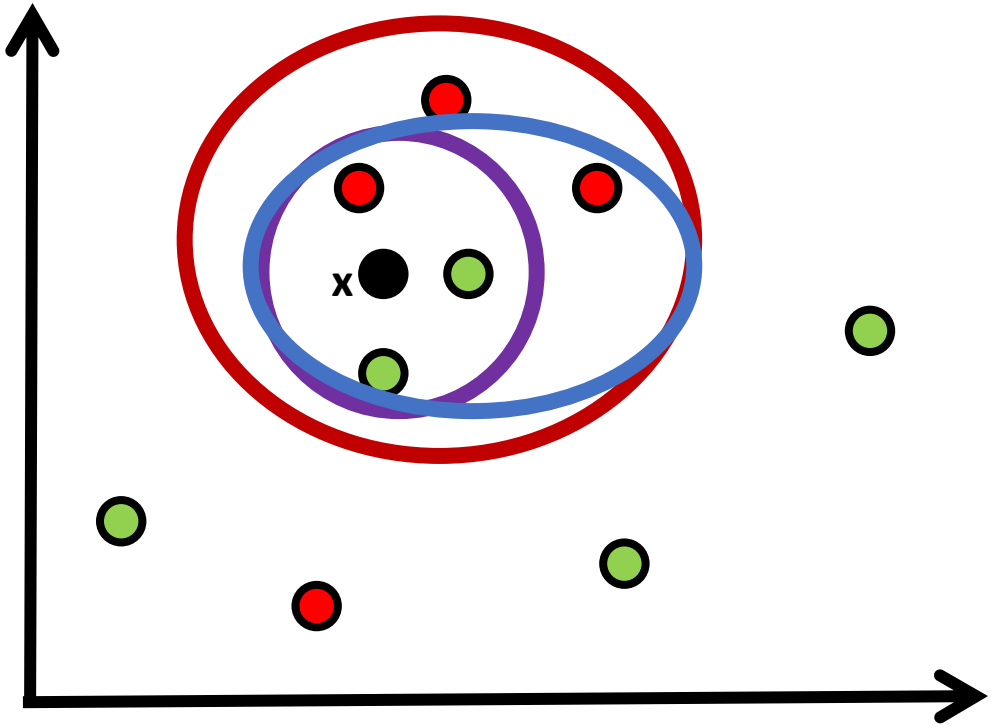
$$\hat{f}(x_q) <- \arg\max_{V \in v} \sum_{i=1}^{k} \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise

MACHINE INTELLIGENCE

Working of algorithm

consider the given plot of the data set with two classes,this completes the training algorithm
consider x to be the query point
with k=3 the query is classified as green class
with k=5 the query is classified as red class
we previously said K should be odd ,lets see why
with k=4 the query point can be classified since it has equal neighbour of both the class

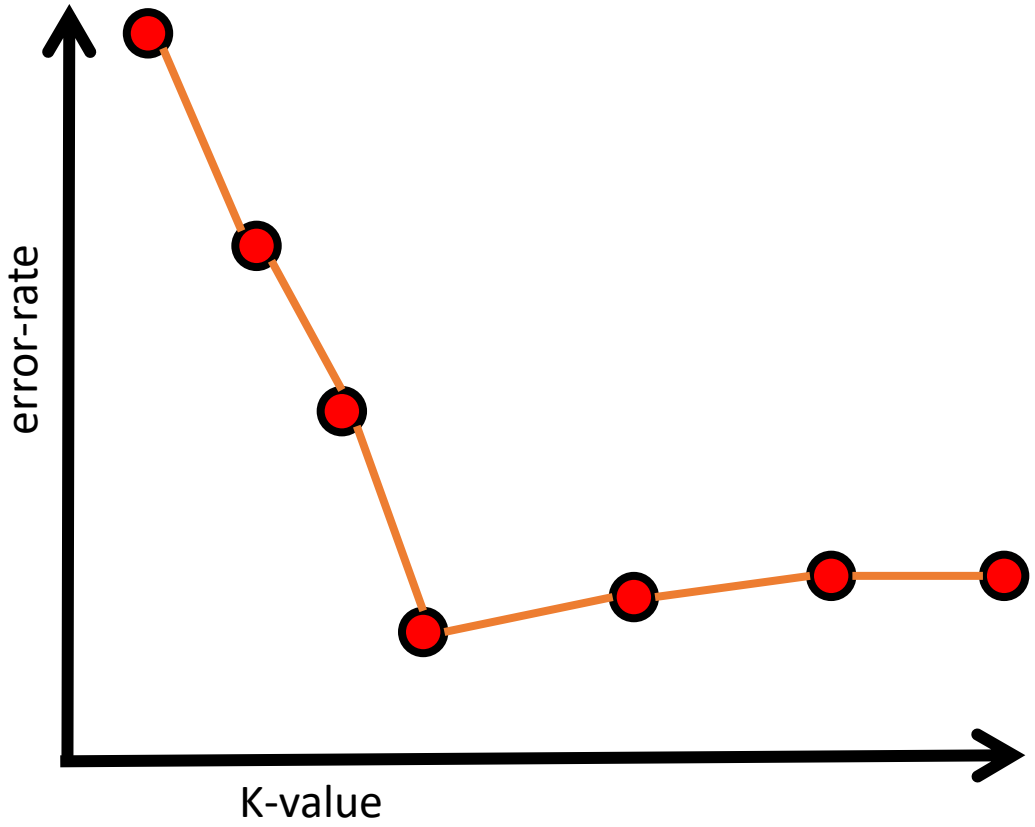


MACHINE INTELLIGENCE

Elbow Method

The question still remains unsolved that what value of k should we choose
We will use **Elbow Method** to determine the best **K** value

- calculate error rate for different K-value
- choose the optimal k value as elbow value of the curve
- Retrain with new K Value
- Retrain your model with the best K value and re-do the classification report and the confusion matrix



Return value for real valued class attributed

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

Here, we calculate the average of the target value for the k nearest neighbouring instances(for real valued class attribute)

How to do?

Step 1: Load Data (CSV, XLS FILE etc.)
Step 2: Initialize K(hyper parameter k=3,5,7,...)
Step 3: For each sample in the training data
    3.1 Calculate distance between query point and current point
    3.2 Add the distance and the index of the example to an ordered collection
Step 4: Sort the ordered collection of distances and indexes in ascending order
Step 5: Take first K entries from sorted collection
Step 6: Get the labels of selected k entries
Step 7: If classification return mode of K values
Step 8: If regression return mean of K values

THANK YOU

K.S.Srinivas
srinivasks@pes.edu
+91 80 2672 1983 Extn 701

MACHINE INTELLIGENCE

The *k*-Nearest Neighbor Example

K.S.Srinivas

Department of Computer Science and Engineering

MACHINE INTELLIGENCE

The *k*-Nearest Neighbor Example

K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

Hands on KNN

Lets take a simple example for understanding KNN

Attriubte1	Attribute2	class
7	7	False
7	4	False
3	4	True
1	4	True

consider the above data set with binary class and 2 attributes

AIM: Use KNN algorithm to determine the class of a queried object

Hands-on KNN

- We can take distance measure as

1. Euclidean Distance

2. Manhattan Distance

3. Minkowski Distance etc.

Assumptions

Equal weights to all attributes?

1. Only if the scale of the attributes are similar and differences.

2. Scale attribute to equal range and equal variance

3. Classes are spherical

Lets take Euclidean Distance

$$dist((x, y), (a, b)) = \sqrt{(x - a)^2 + (y - b)^2}$$

MACHINE INTELLIGENCE

Hands-on KNN

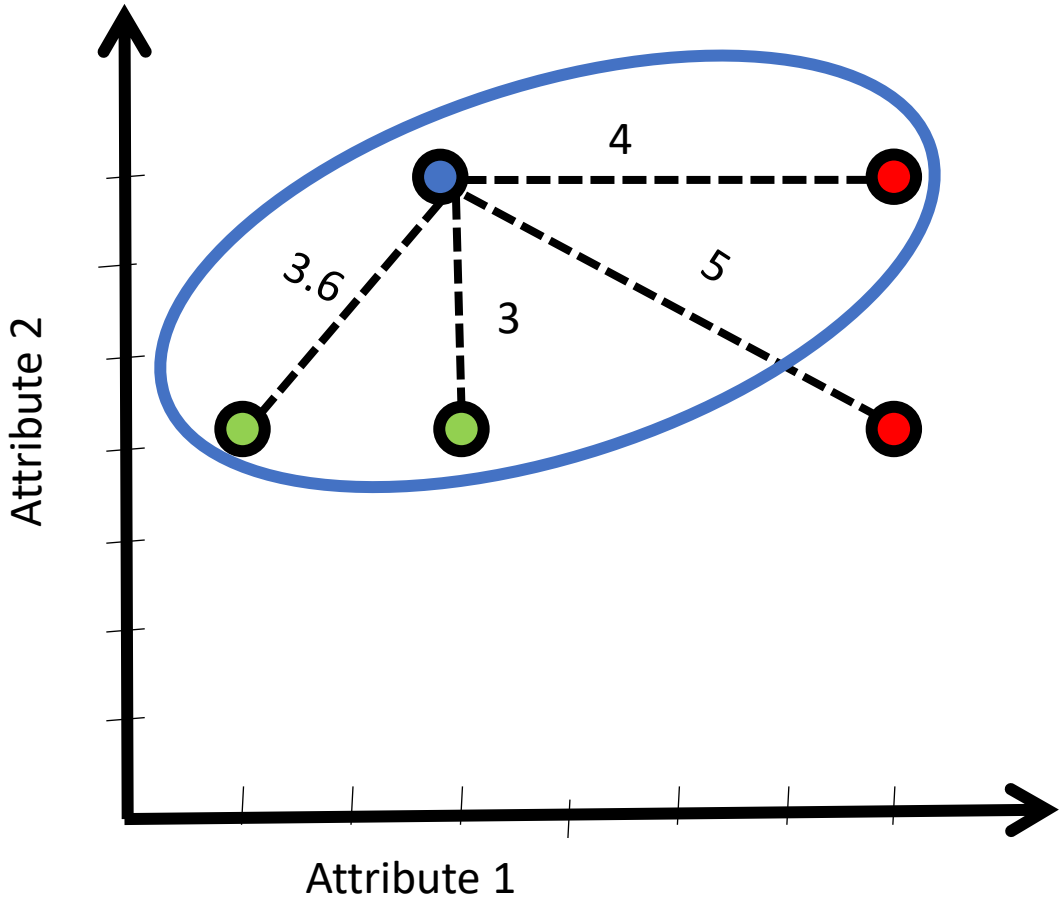
Problem: Perform KNN Classification on following data set and predict
class for x(Attribute1=3, Attribute2=7) where k=3

Attriubte1	Attribute2	class
7	7	False
7	4	False
3	4	True
1	4	True

MACHINE INTELLIGENCE

Hands-on KNN

we will first plot this on a 2-D surface
we will now calculate Euclidean distance from the queried point
with k=3 ,we get the following neighbour
with majority we assign true label to the x



Attriubte1	Attribute2	class
7	7	False
7	4	False
3	4	True
1	4	True

Problem: Perform KNN Classification on following data set and predict
class for x(Attribute1=3, Attribute2=7) where k=3

THANK YOU

K.S.Srinivas
srinivasks@pes.edu
+91 80 2672 1983 Extn 701

MACHINE INTELLIGENCE

Weighted K-Nearest Neighbour

K.S.Srinivas

Department of Computer Science and Engineering

MACHINE INTELLIGENCE

Weighted K-Nearest Neighbour

K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

Weighted KNN

- Distance-weighted nearest neighbor algorithm
- Weight the contribution of each of the k neighbors according to their distance to the query x_q
- Give greater weight to closer neighbors

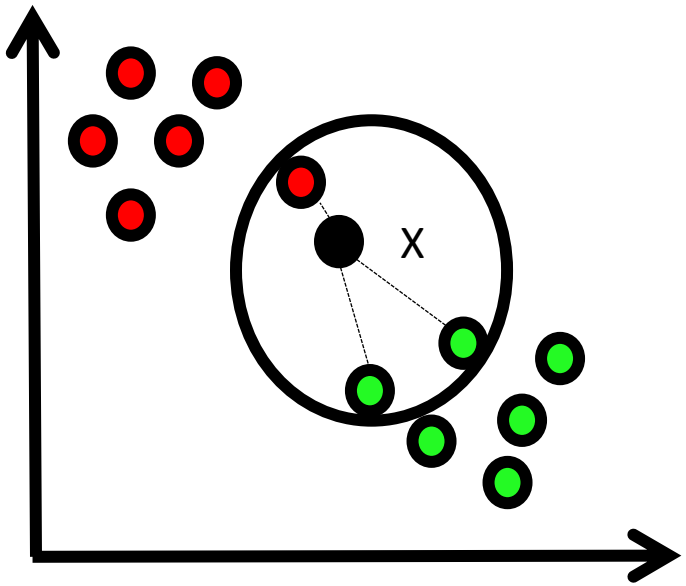
$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} w_i \delta(v, f(x_i))$$

$$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

MACHINE INTELLIGENCE

Why Weighted KNN

consider the given 2-D plot of a two class data set
consider the query point x
with normal KNN with K=3 x will be classified as green
with weighted KNN and K=3 x will be classified as red since its more close to the query x



Inductive Bias of KNN

The classification of an instance x, will be most similar to the classification of the K other instances that are nearby.

In simple English Birds of the same feather flock together

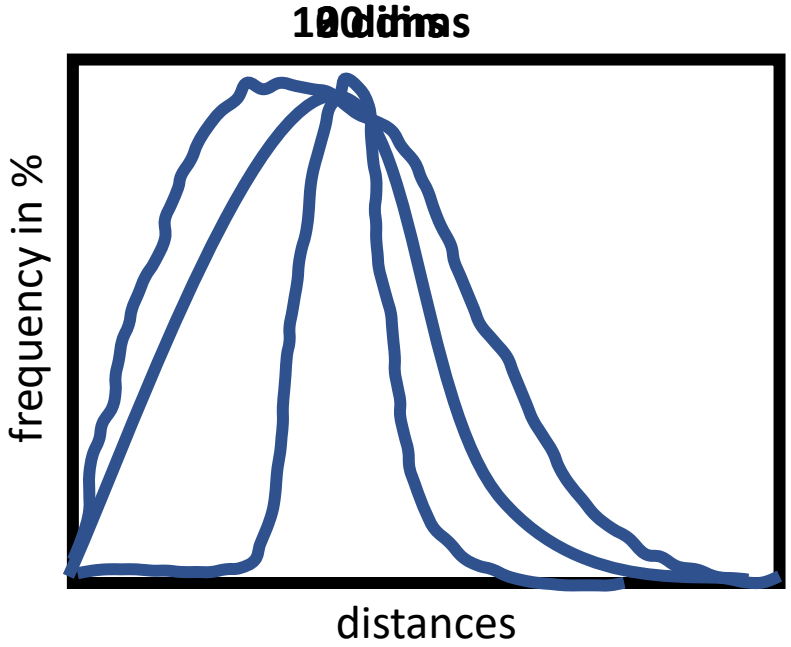
Issues with KNN

- **K-NN slow algorithm:**K-NN might be very easy to implement but as data set grows efficiency or speed of algorithm declines very fast.

- **Curse of Dimensionality:** KNN works well with small number of input variables but as the numbers of variables grow K-NN algorithm struggles to predict the output of new data point.

Curse of Dimensionality

- The Radius or circle of influence of each data point becomes smaller and smaller as we have more attributes.
- Rule of thumb – 5 data instances per attribute for learning



Overcoming Curse of Dimensionality

The curse of Dimensionality can be overcome by:

- Assigning weights to the attributes when calculating distances.
- Using the "**Leave-one-out**" approach. Iteratively, we leave out one of the attributes and test the algorithm by the cross-validation method. The exercise can then lead us to the best set of attributes.
- Rule of thumb – 5 data instances per attribute for learning

Issues with KNN

- **K-NN needs homogeneous features:** If you decide to build k-NN using a common distance, like Euclidean or Manhattan distances, it is completely necessary that features have the same scale, since absolute differences in features weight the same, i.e., a given distance in feature 1 must means the same for feature 2

Issues with KNN

- **Optimal number of neighbors:** One of the biggest issues with K-NN is to choose the optimal number of neighbors to be considered while classifying the new data entry
- **Imbalanced data causes problems:** k-NN doesn't perform well on imbalanced data. If we consider two classes, A and B, and the majority of the training data is labeled as A, then the model will ultimately give a lot of preference to A. This might result in getting the less common class B wrongly classified.

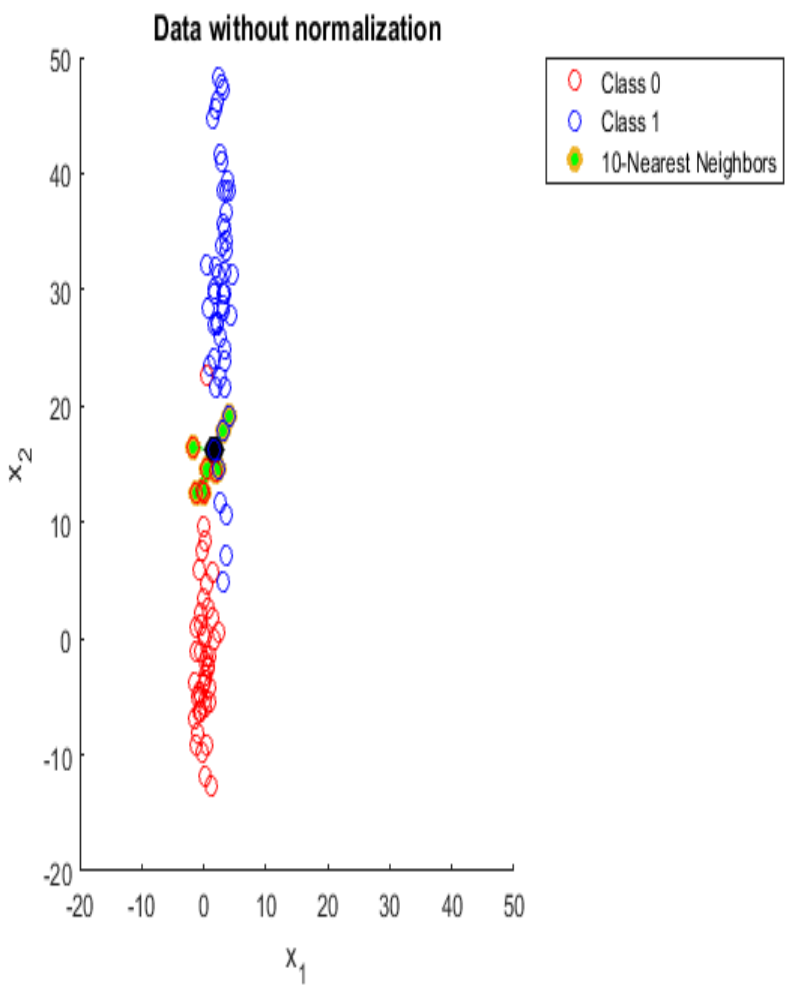
Issues with KNN

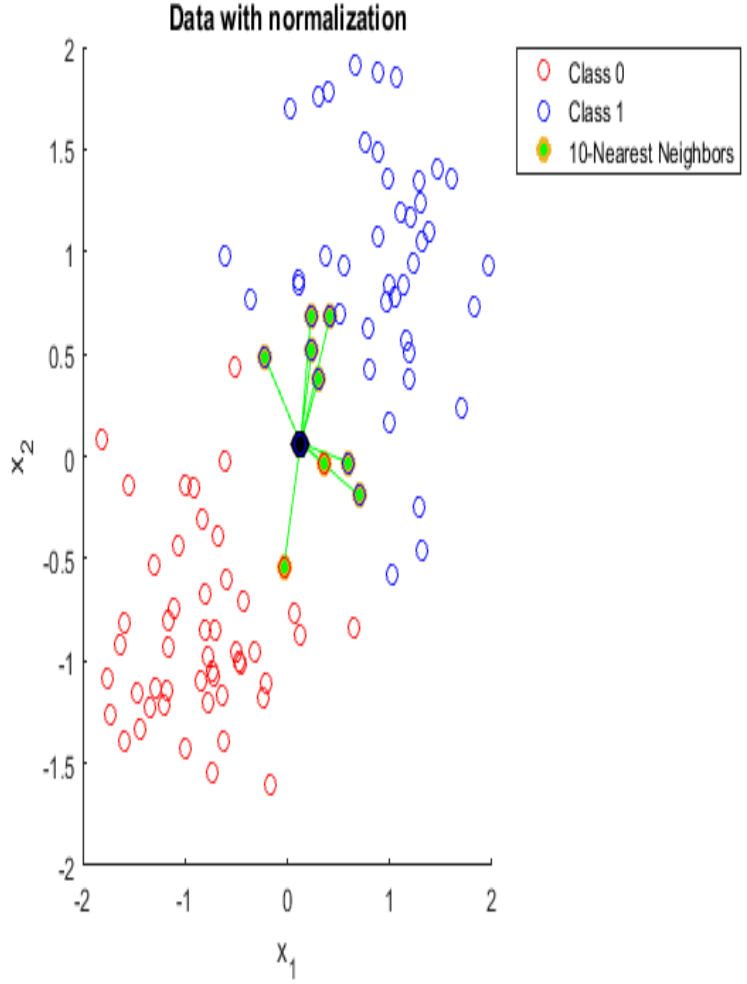
- **Outlier sensitivity:** K-NN algorithm is very sensitive to outliers as it simply chose the neighbors based on distance criteria.

- **Missing Value treatment:** K-NN inherently has no capability of dealing with missing value problem.

MACHINE INTELLIGENCE

Issues with KNN





KNN Computational Complexity

- Basic kNN algorithm stores all examples
- Suppose we have n examples each of dimension d
- ***O*(d)** to compute distance to one example
- ***O*(nd)** to find one nearest neighbor
- ***O*(knd)** to find k closest examples examples
- Thus complexity is ***O*(knd)**
- This is prohibitively expensive for large number of samples
- But we need large number of samples for kNN to work well!

THANK YOU

K.S.Srinivas
srinivasks@pes.edu
+91 80 2672 1983 Extn 701

MACHINE INTELLIGENCE

Artificial Neural Network

K.S.Srinivas

Department of Computer Science and Engineering

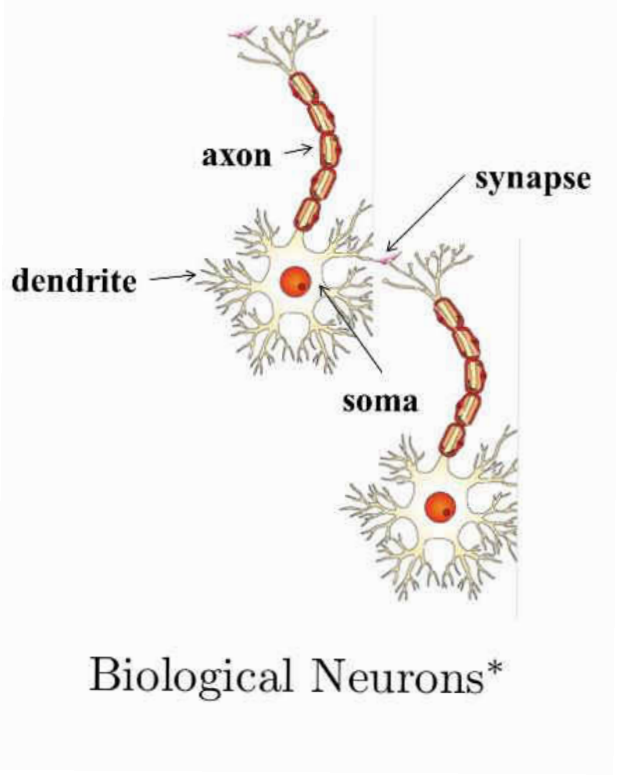
MACHINE INTELLIGENCE

Artificial Neural Network

K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

The Biological Neuron



Biological Neurons*

dendrite: receives signals from other neurons

synapse: point of connection to other neurons

soma: processes the information

axon: transmits the output of this neuron

MACHINE INTELLIGENCE

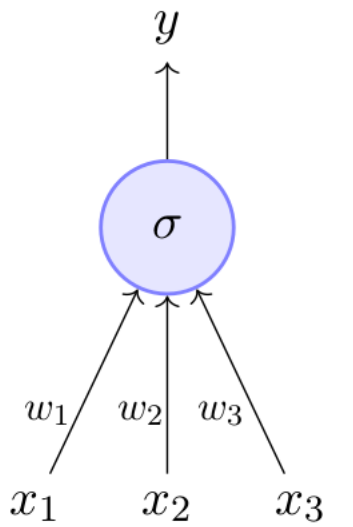
The Artificial Neural Network

The most fundamental unit of a deep neural network is called an artificial Neuron

Why is it called a neuron ? Where does the inspiration come from ?

The inspiration comes from biology (more specifically, from the brain)

biological neurons = neural cells = neural processing units



Artificial Neuron

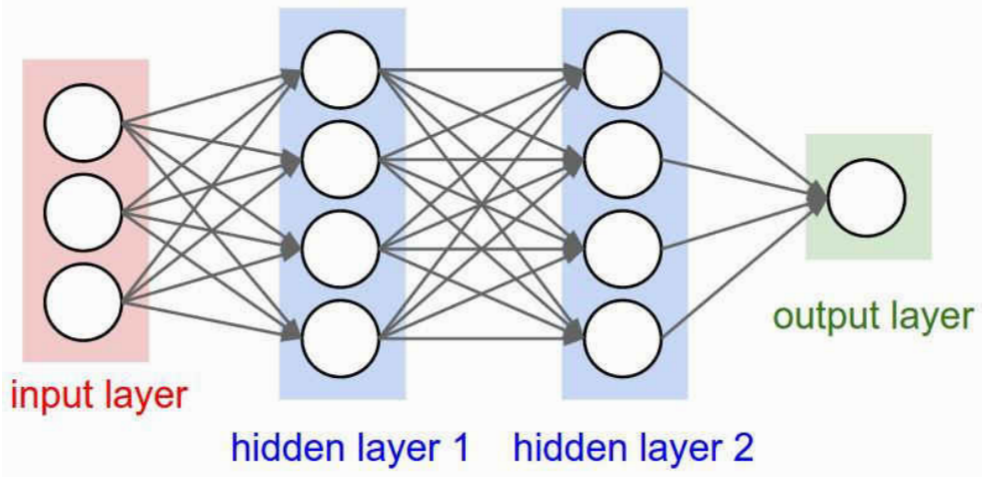
The Biological Neuron

- There is a massively parallel interconnected network of neurons. The sense organs relay information to the lowest layer of neurons
- An average human brain has around 10 power 11 (100 billion) neurons!
- This massively parallel network also ensures that there is division of work
- Each neuron may perform a certain role or respond to a certain stimulus

MACHINE INTELLIGENCE

A typical Artificial Neural Network

Neural network learning methods provide a robust approach to approximating **real-valued**, **discrete-valued**, and **vector** of real and discrete-valued functions.



Neural networks are most **effective** for many practical problems such as

1. learning to recognize handwritten characters
2. learning to recognize spoken words
3. learning to recognize faces
4. self-driven cars

For certain types of problems interpreting real-world **sensor data**, artificial neural networks are **effective** learning methods.

When to use ANN?

Instances are represented by **many attribute-value pairs.**

The **target function output** may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.

The training examples may contain **errors**.

THANK YOU

K.S.Srinivas
srinivasks@pes.edu
+91 80 2672 1983 Extn 701

MACHINE INTELLIGENCE

Perceptron

K.S.Srinivas

Department of Computer Science and Engineering

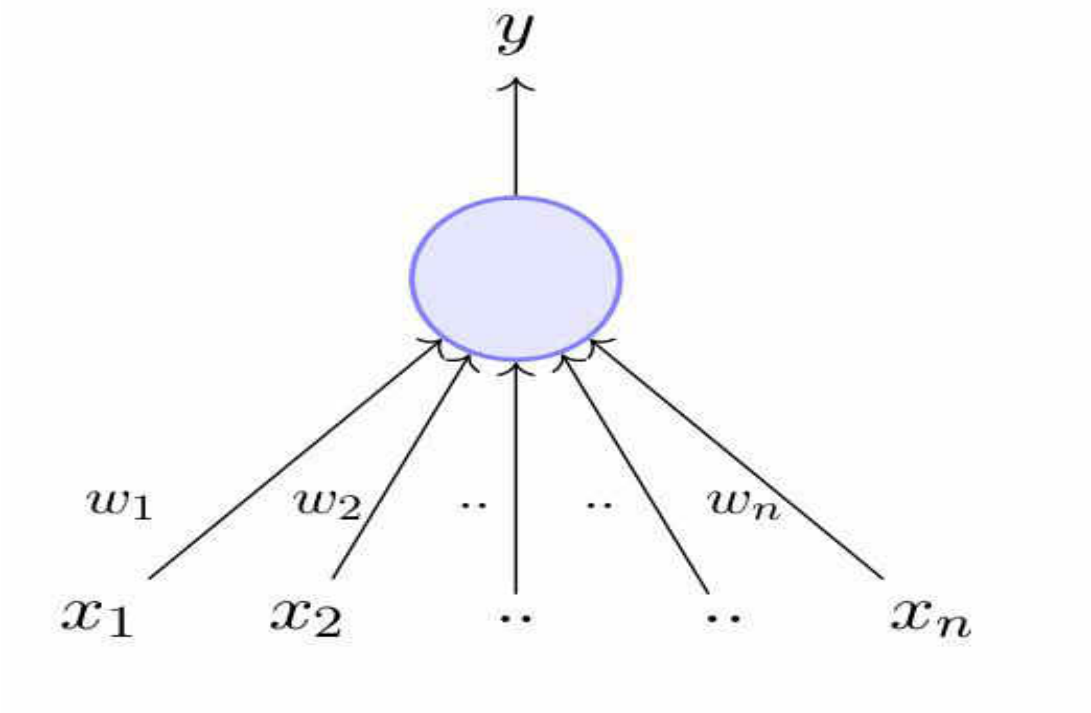
MACHINE INTELLIGENCE

Perceptron

K.S.Srinivas
Department of Computer Science and Engineering

The Perceptron

- A perceptron takes a **vector of real-valued inputs**, calculates a **linear combination** of these inputs, then outputs **1** if the result is greater than some threshold and **-1** otherwise.



The Perceptron

Frank Rosenblatt, an American psychologist, proposed the classical perceptron model (1958)

A more general computational model than McCulloch–Pitts neurons

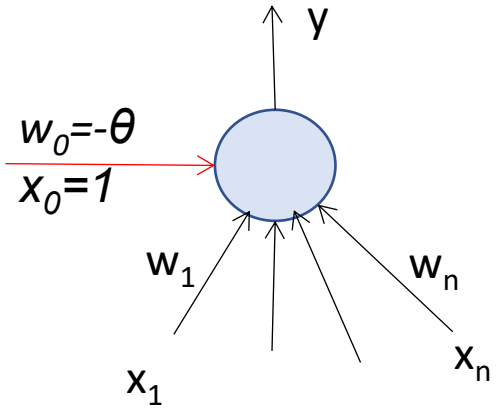
Main differences: Introduction of numerical weights for inputs and a mechanism for learning these weights

Inputs are no longer limited to boolean values

Refined and carefully analyzed by Minsky and Papert (1969) - their model is referred to as the perceptron model here

MACHINE INTELLIGENCE

The Perceptron



$$y = 1 \text{ if } \sum_{i=0}^{n} w_i * x_i \geq 0$$

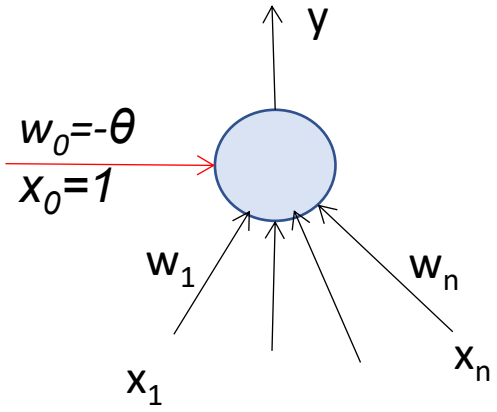
$$y = 0 \text{ if } \sum_{i=0}^{n} w_i * x_i < 0$$

where w_0 =-θ and x_0=1

- consider this single perceptron, we call w_0 as bias and X_0=1
- w_1.....w_n are weights associated with inputs x_1...x_n.
- y is the output
- let us try to frame equation for the output y

MACHINE INTELLIGENCE

The Perceptron



$$y = 1 \text{ if } \sum_{i=0}^{n} w_i * x_i \geq 0$$

$$y = 0 \text{ if } \sum_{i=0}^{n} w_i * x_i < 0$$

where $w_0 = -\theta$ and $x_0 = 1$

- $w_0 * x_0$ are known ,hence we can rewrite the equation as

$$y = 1 \text{ if } \sum_{i=0}^{n} w_i * x_i - \theta \geq 0$$

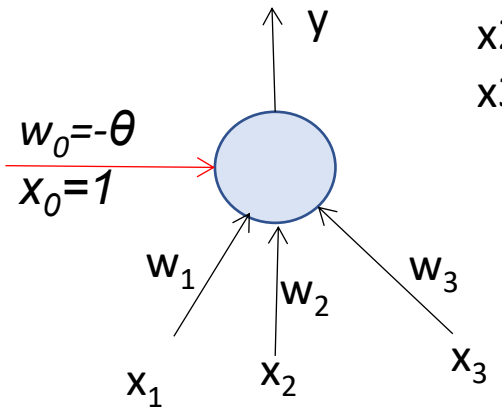
$$y = 0 \text{ if } \sum_{i=0}^{n} w_i * x_i - \theta < 0$$

$$y = 1 \text{ if } \sum_{i=0}^{n} w_i * x_i \geq \theta$$

$$y = 0 \text{ if } \sum_{i=0}^{n} w_i * x_i < \theta$$

MACHINE INTELLIGENCE

Simple Example Perceptron



x1 =actor
x2=genre
x3=director

Consider the task of predicting whether we would like a movie or not

Suppose, we base our decision on 3 inputs

Based on our past viewing experience (data), we may give a high weight to Director as compared to the other inputs

Specifically, even if the actor is not xyz and the genre is not thriller we would still want to cross the threshold θ by assigning a high weight to director

A movie buff may have a very low threshold and may watch any movie irrespective of the genre, actor, director [θ = 0]

MACHINE INTELLIGENCE

Logical OR learning

x_1	x_2	OR
0	0	0
1	0	1
0	1	1
1	1	1

$$\hat{y} = w_0 + \sum_{i=1}^{n} w_i x_i$$

$$y = 0 \text{ if } \hat{y} < 0$$

$$y = 1 \text{ if } \hat{y} \geq 0$$

consider the truth table of logic OR and our perceptron equation

let us frame the perceptron eqn for each input

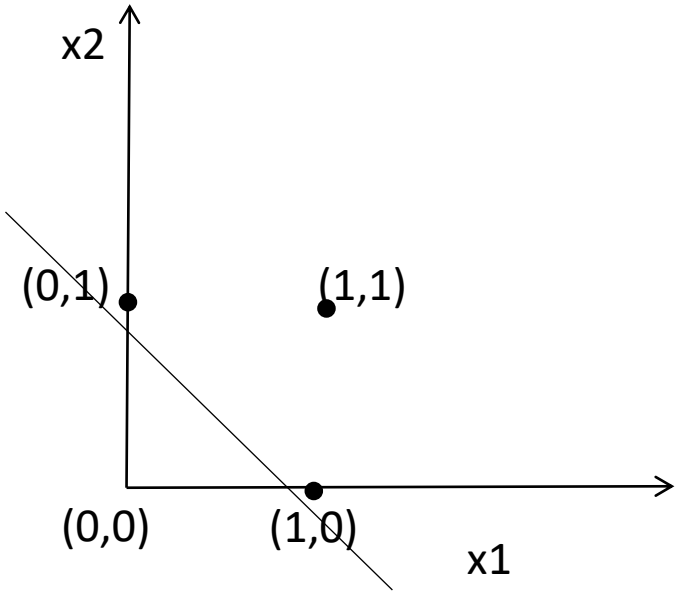
$w_0+w_1.0+w_2.0<0 \quad \Rightarrow w_0<0$

$w_0+w_1.1+w_2.0\geq 0 \quad \Rightarrow w_1\geq -w_0$

$w_0+w_1.0+w_2.1\geq 0 \quad \Rightarrow w_2\geq -w_0$

$w_0+w_1.1+w_2.1\geq 0 \quad \Rightarrow w_2+w_1\geq -w_0$

- One possible solution to this set of inequalities is $w_0 = -1, w_1 = 1.1, , w_2 = 1.1$ (and various other solutions are possible)



MACHINE INTELLIGENCE

Perceptron Learning Algorithm

$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$

Initialize $\mathbf{w}$ randomly;

while $!convergence$ **do**
  Pick random $\mathbf{x} \in P \cup N$;
  if $\mathbf{x} \in P \quad and \quad \sum_{i=0}^{n} w_i * x_i < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$;
  end
  if $\mathbf{x} \in N \quad and \quad \sum_{i=0}^{n} w_i * x_i \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$;
  end
end

//the algorithm converges when all the inputs are classified correctly

we will look at the concept of convergence later

MACHINE INTELLIGENCE

Revisiting the Perceptron Learning Concept

- consider two vector w and x

$\mathbf{w}=[w_0,w_1,.....,w_n]$
$\mathbf{x}=[x_0,x_1,.....,x_n]$

$\mathbf{w.x}= w^Tx = \sum w_i*x_i$

- we can thus rewrite the perceptron rule as

$y=1 \quad if \quad w^Tx \geq 0$
$=0 \quad if \quad w^Tx < 0$

- we are interested in finding the line $w^Tx=0$ which divides the input space into two halves
- Every point (x) on this line satisfies the equation $w^Tx=0$

- let us call the angle between w and any point x as α
- what could be value of α ???
- α=90 $(\cos\alpha = \frac{w^T x}{\| w \| \| x \|})$
- since the vector w is perpendicular to every point on the line it is actually perpendicular to the line itself

MACHINE INTELLIGENCE

Revisiting the Perceptron Learning Concept

Consider some points (vectors) which lie in the positive half space of this line (*i.e.*, $\mathbf{w^T x} \geq 0$)

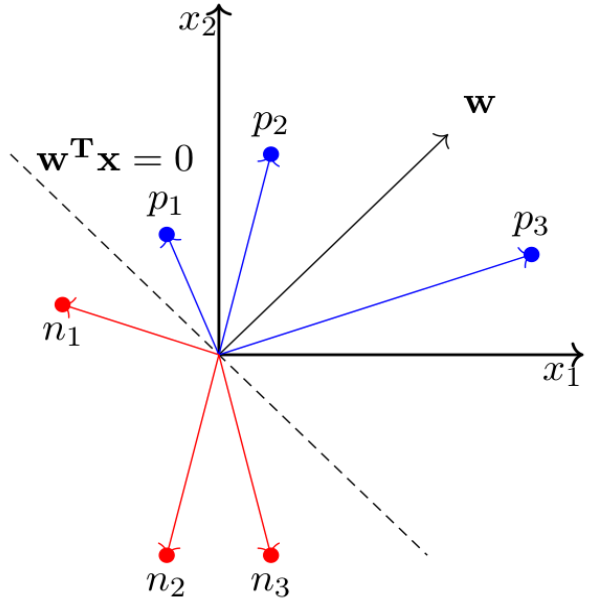
What will be the angle between any such vector and $\mathbf{w}$? Obviously, less than 90°

What about points (vectors) which lie in the negative half space of this line (*i.e.*, $\mathbf{w^T x} < 0$)

What will be the angle between any such vector and $\mathbf{w}$? Obviously, greater than 90°

Of course, this also follows from the formula ($cos\alpha = \frac{w^T x}{||w||||x||}$)

Keeping this picture in mind let us revisit the algorithm



MACHINE INTELLIGENCE

Perceptron Learning Algorithm

$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
while !*convergence* **do**
  Pick random $\mathbf{x} \in P \cup N$;
  if $\mathbf{x} \in P \quad and \quad \mathbf{w}.\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$;
  end
  if $\mathbf{x} \in N \quad and \quad \mathbf{w}.\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$;
  end
end
//the algorithm converges when all the inputs are classified correctly

$$cos\alpha = \frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||||\mathbf{x}||}$$

For $\mathbf{x} \in P$ if $\mathbf{w}.\mathbf{x} < 0$ then it means that the angle (α) between this $\mathbf{x}$ and the current $\mathbf{w}$ is greater than 90° (but we want α to be less than 90°)

What happens to the new angle (α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$\begin{aligned} cos(\alpha_{new}) &\propto \mathbf{w_{new}}^T\mathbf{x} \\ &\propto (\mathbf{w} + \mathbf{x})^T\mathbf{x} \\ &\propto \mathbf{w}^T\mathbf{x} + \mathbf{x}^T\mathbf{x} \\ &\propto cos\alpha + \mathbf{x}^T\mathbf{x} \\ cos(\alpha_{new}) &> cos\alpha \end{aligned}$$

Thus α_{new} will be less than α and this is exactly what we want

The Perceptron learning for Linear Surface

A perceptron is a thresholded linear unit (discrete-valued).

Linear Unit: A linear combination of weighted inputs (real-valued).

Perceptron offer a **linear decision surface**. This means a single perceptron can easily represent simple Boolean functions like AND, OR,NAND and NOR.

AND: $w_0 = -0.8$, $w_1 = w_2 = 0.5$
OR: $w_0 = -0.3$, $w_1 = w_2 = 0.5$

MACHINE INTELLIGENCE

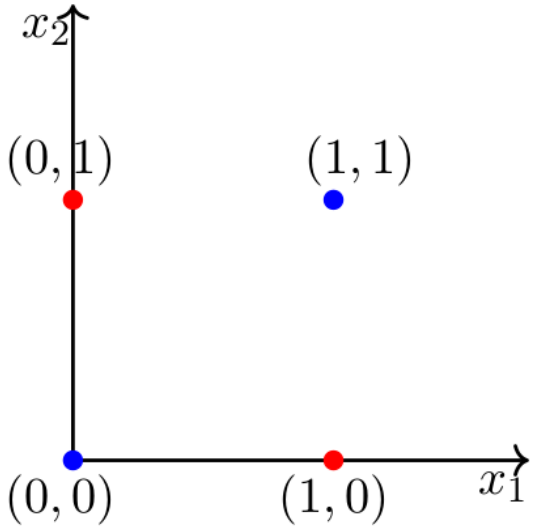
XOR

x_1	x_2	XOR
0	0	0
1	0	1
0	1	1
1	1	0

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

The fourth condition contradicts conditions 2 and 3

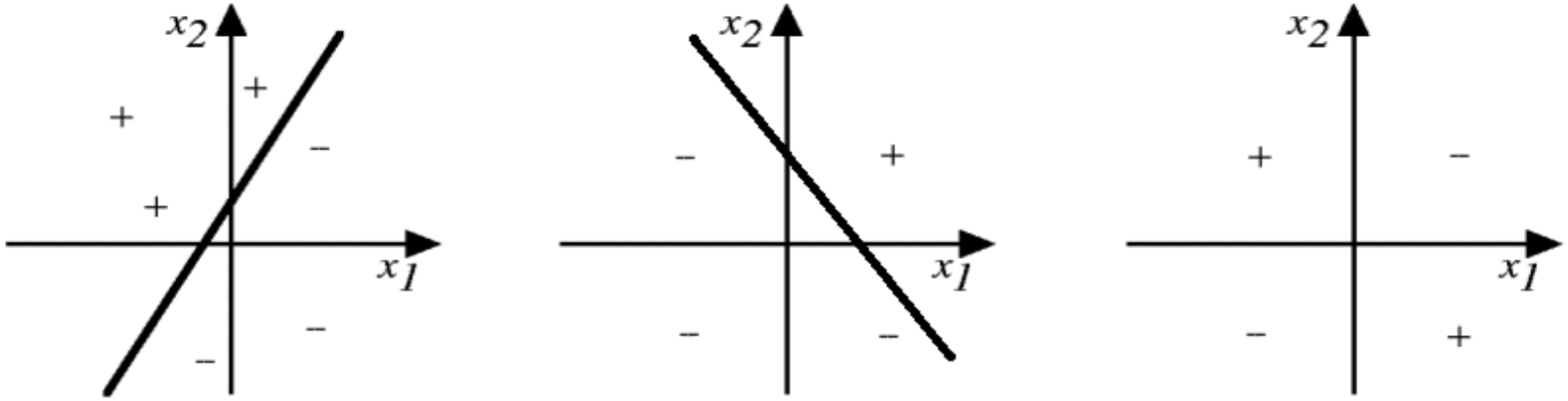
Hence we cannot have a solution to this set of inequalities



And indeed you can see that it is impossible to draw a line which separates the red points from the blue points

Perceptron fails here

If the data is **not linearly separable** as in XOR function, then a single perceptron is not enough to represent the functionality.



Linearly separable | Linearly separable | Not linearly separable

Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta (t - o) x_i$$

x_i inputs
w_i is the weight related to x_i
t is the target output for the current training example
o is the output generated by the perceptron
η is the learning rate, usually set to a small value (say 0.1)

The perceptron learns a linear decision surface of the form

$h(x) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + \ldots$

Delta Rule and Gradient Descent

When data is not linearly separable, the **perceptron rule** may **not converge** to a proper solution.

The **delta rule** converges toward a **best-fit approximation** to the target concept.

Gradient Descent searches the hypothesis space of possible weight vectors to find the weights that **best fit** the training examples.

We will properly try to explore what is Gradient Descent algorithm

MACHINE INTELLIGENCE

Gradient Descent

function ->>>> $J(\theta_0,\theta_1)$

required->>>> $\min_{\theta 0,\theta 1} J(\theta_0,\theta_1)$

Steps:

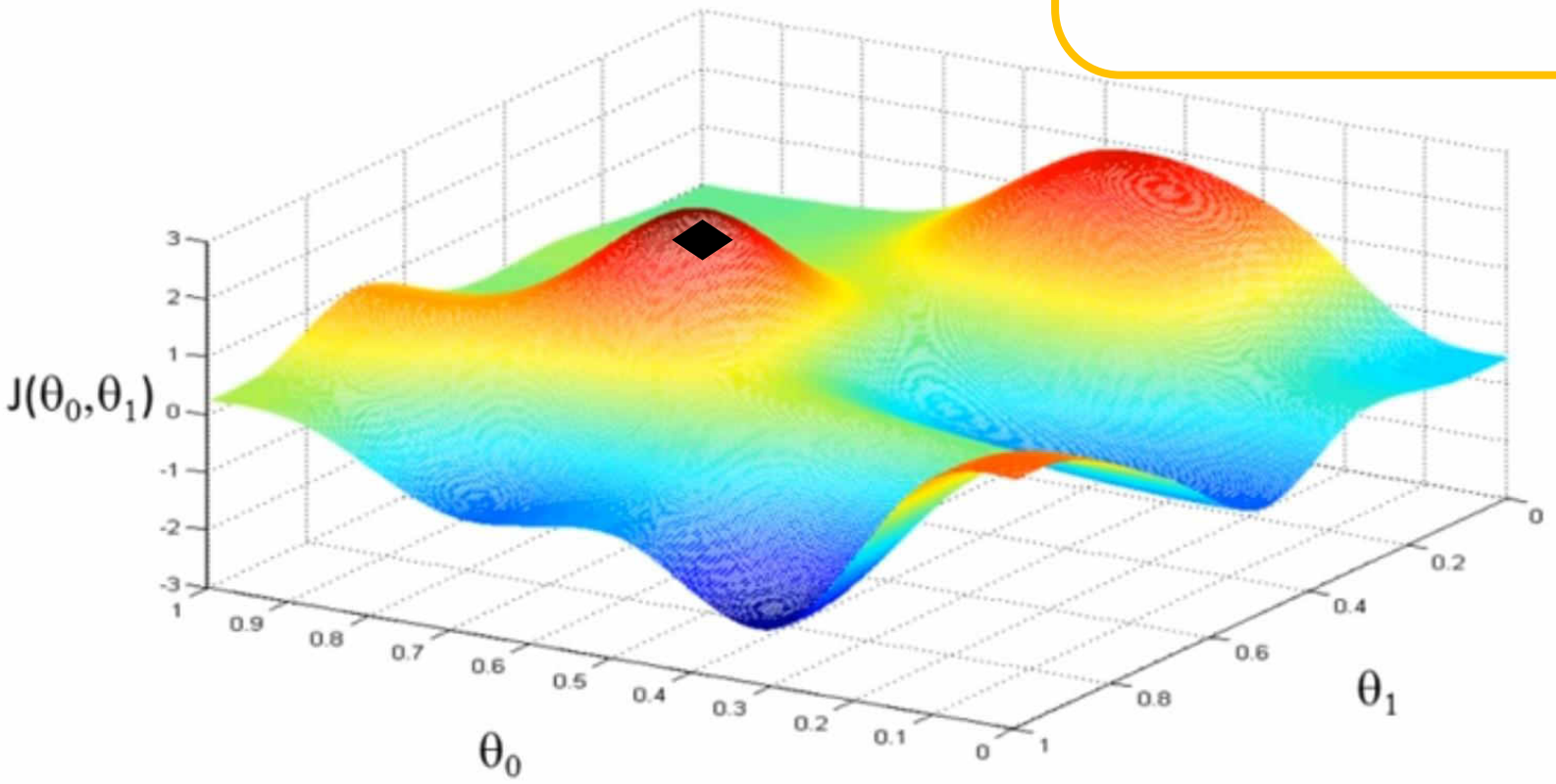
- start with some θ_0,θ_1
- keep changing θ_0,θ_1 to reduce $J(\theta_0,\theta_1)$
- do this until you reach minimum

- let us begin with a function $J(\theta_0,\theta_1)$

MACHINE INTELLIGENCE

Gradient Descent

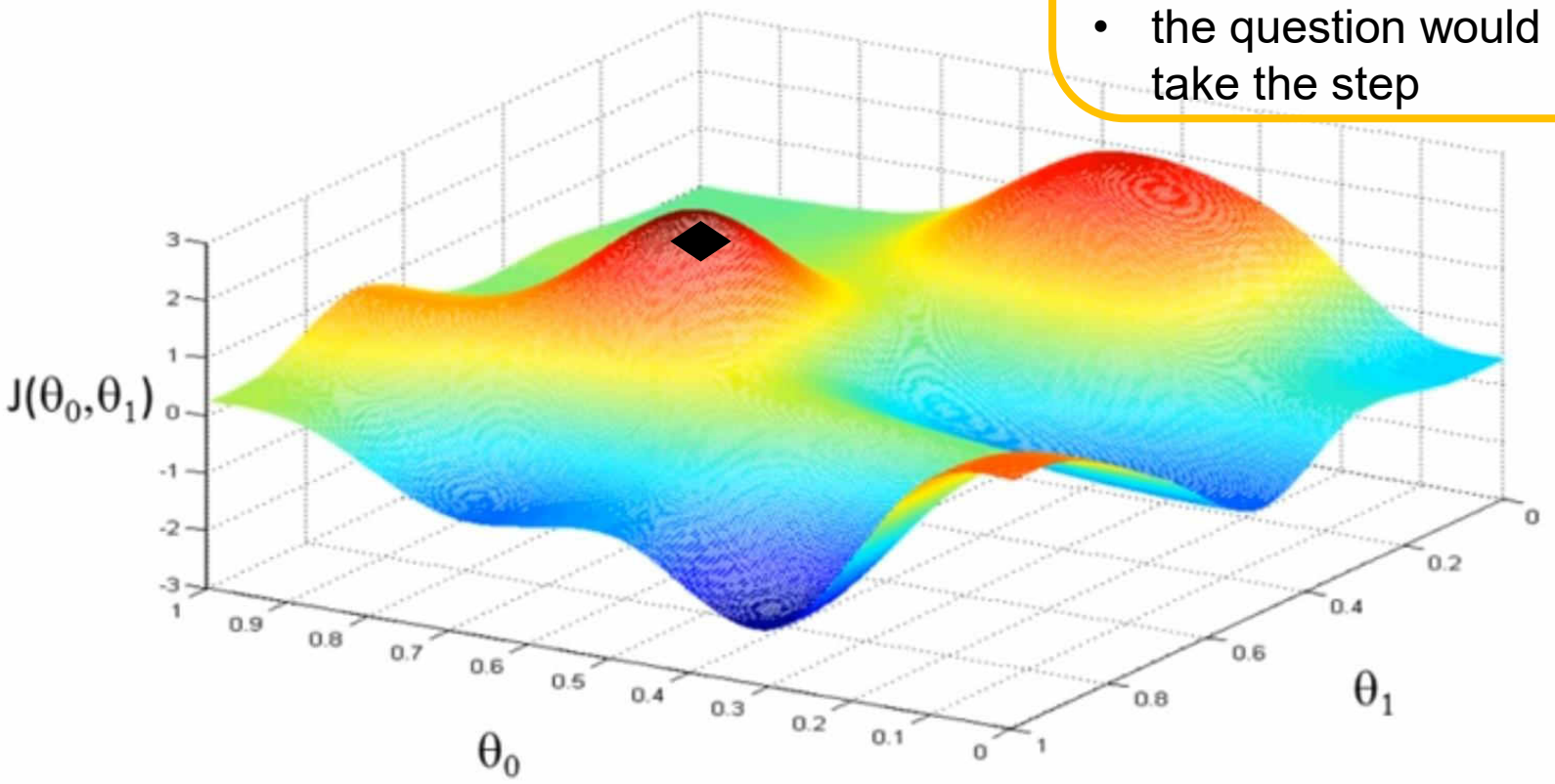
- let say we are trying to minimize this function
- we first start some random $J(\theta_0,\theta_1)$



MACHINE INTELLIGENCE

Gradient Descent

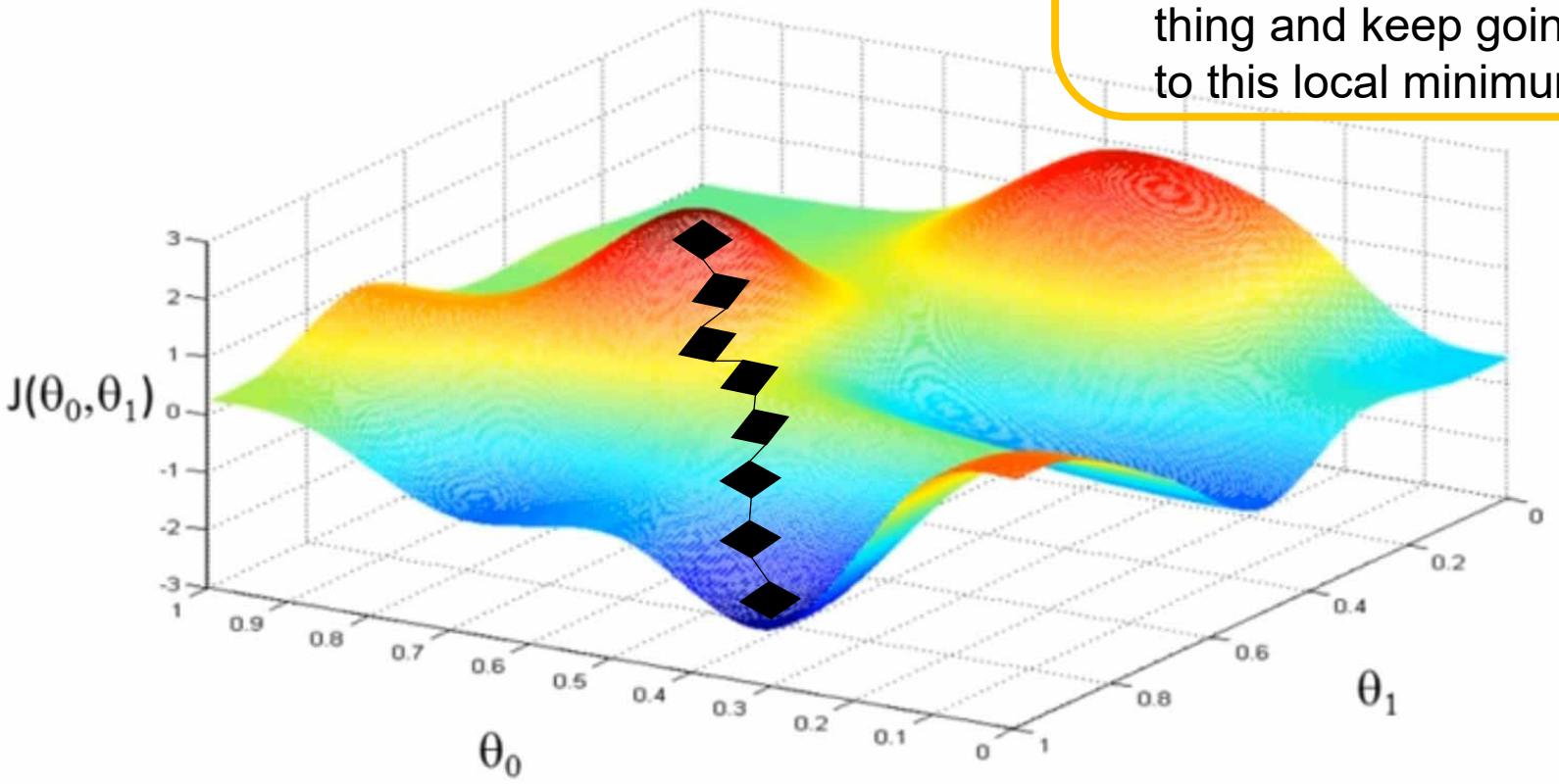
- imagine the surface as some trekking area and you are standing at that point
- in gradient descent what we do is we take a small step in some direction and go down as quickly as possible
- the question would be what direction to take the step



MACHINE INTELLIGENCE

Gradient Descent

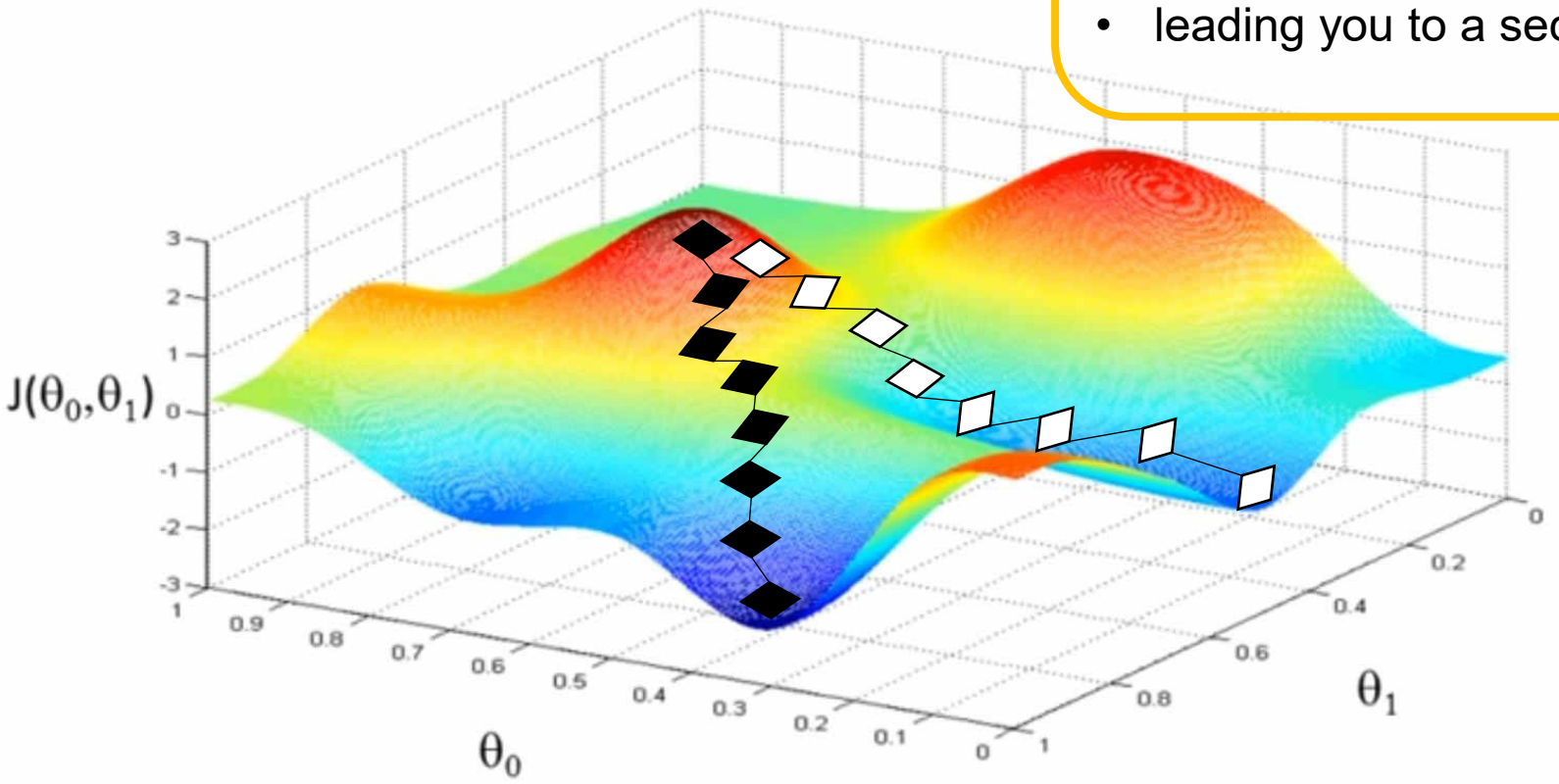
- what you would do is look around and take step
- suppose you take a step in the following direction
- again in this new point you do the same thing and keep going until you converge to this local minimum



MACHINE INTELLIGENCE

Gradient Descent

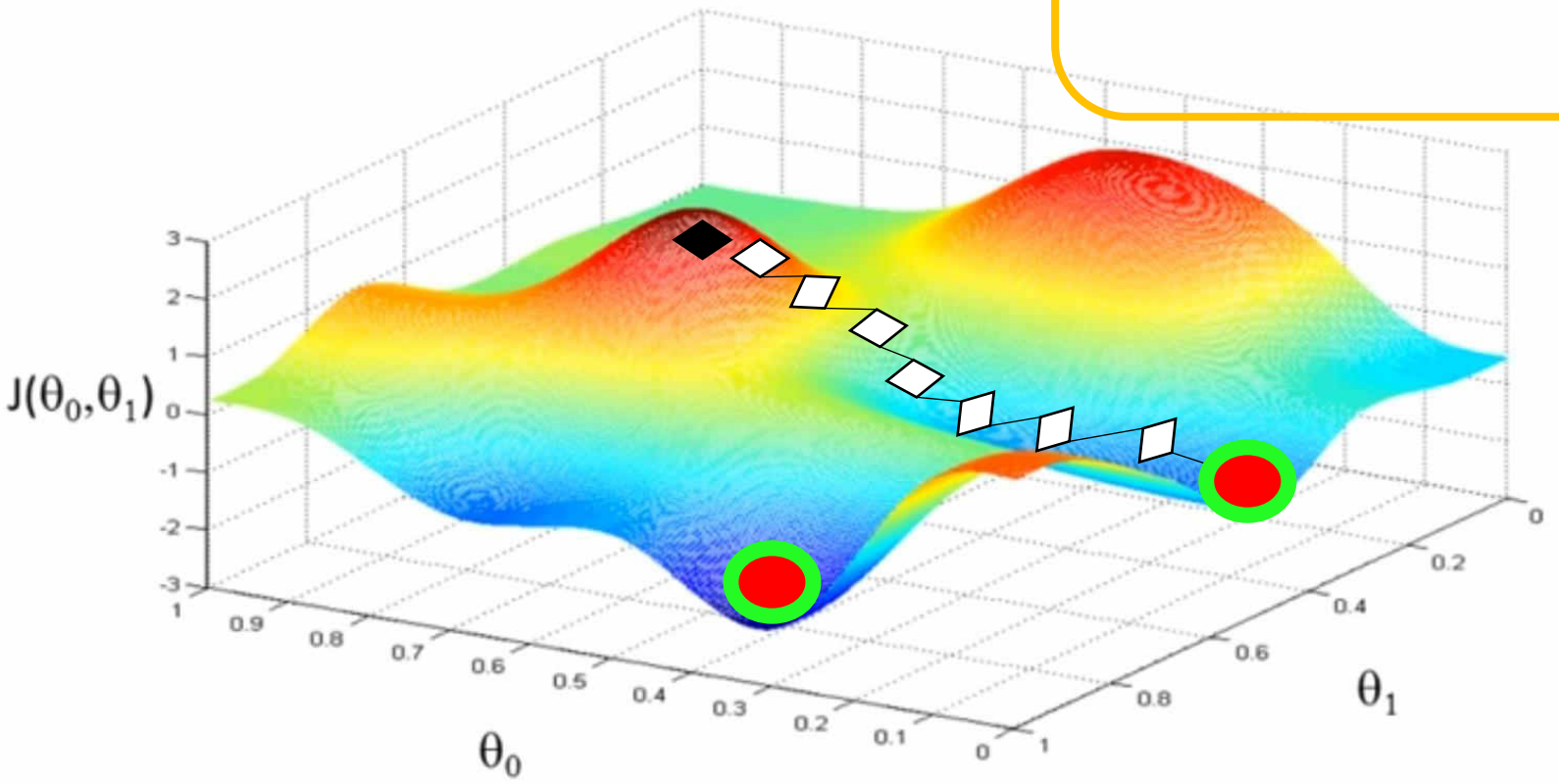
- now suppose you started with a point just right of our previous initial point
- what you would do is again check the steepest descent and proceed in that way
- leading you to a second local optimum



MACHINE INTELLIGENCE

Gradient Descent

- observe that when start with different starting point we have reached different local minimum
- this is one of the property of the gradient descent algorithm



MACHINE INTELLIGENCE

Gradient Descent

repeat until convergence

$$\{ \quad \theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j} \qquad (\text{for} \quad j = 0, 1)$$

}

- this is the algorithm of the gradient descent algorithm
- what we do is until convergence we update our parameters of J with the following rule

MACHINE INTELLIGENCE

Gradient Descent

repeat until convergence

$$\{ \quad \theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j} \qquad (\text{for} \quad j = 0, 1)$$

}

- we need to take care of one thing during updating the values
- although this algo is valid for j=0,1.n, we will limit our discussion for j=0,1

$$\theta_0 = \theta_0 - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0}$$

$$\theta_1 = \theta_1 - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1}$$

this wrong since, when you update θ1 you would be using new value of θ0

the correct understanding of algorithm would be

$$temp\ 0 = \theta_0 - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0}$$

$$temp\ 1 = \theta_1 - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1}$$

$$\theta_0 = temp\ 0$$

$$\theta_1 = temp\ 1$$

MACHINE INTELLIGENCE

Gradient Descent

repeat until convergence

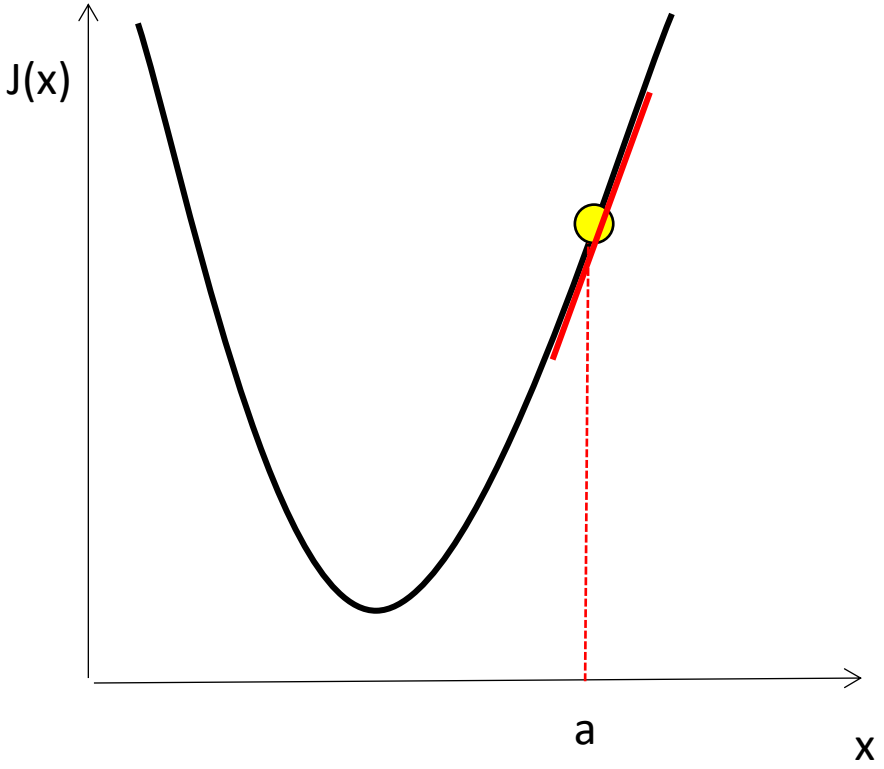
$$\{ \quad \theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j} \qquad (\text{for} \quad j = 0, 1)$$

}

- The alpha in the algorithm is called the learning rate
- this controls hoe bigger step we take downhill
- which means if alpha is bigger you take a bigger step down and visa versa
- we will later see how to set alpha later

MACHINE INTELLIGENCE

Gradient Descent



- consider this function J(x)
- we will start our algorithm and initialize our x to some value say a
- next step in our algorithm is to update our x according to the below equation
- the derivative term is nothing but the slope of the tangent at that point on the curve

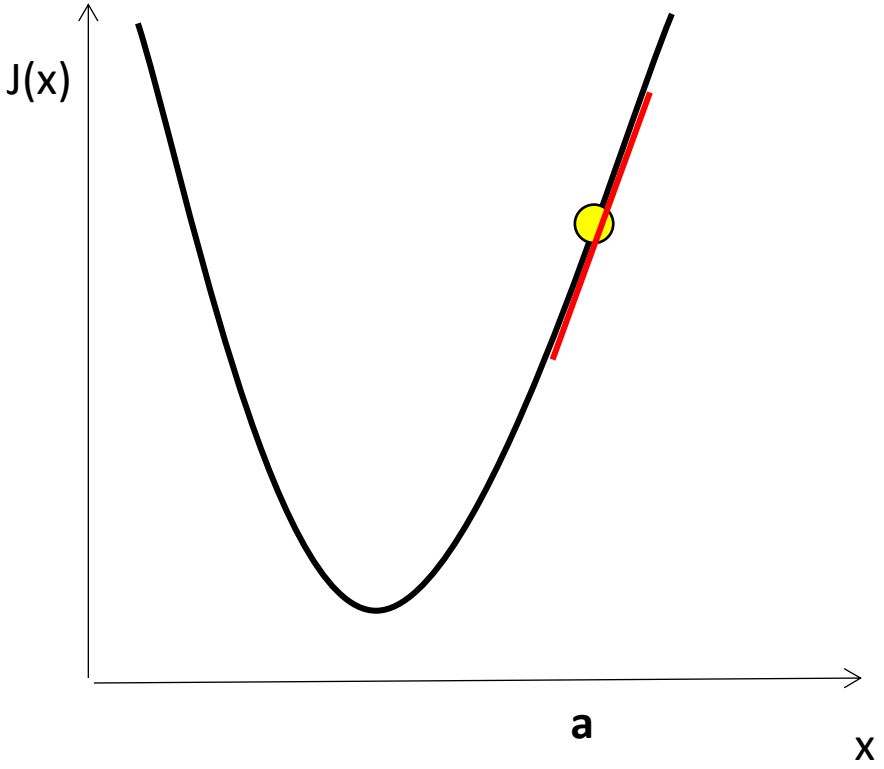
repeat until convergence

$$\{ \quad \theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j} \qquad (\text{for } j = 0, 1)$$

$$\}$$

MACHINE INTELLIGENCE

Gradient Descent



- this line now has a positive slope and hence it has a positive derivative
- so our update of x will be
- we end moving our x to the left,and it is right since its in the direction of the minimum
- and gradient descent seems to be doing well since its making us move in the direction of minimum

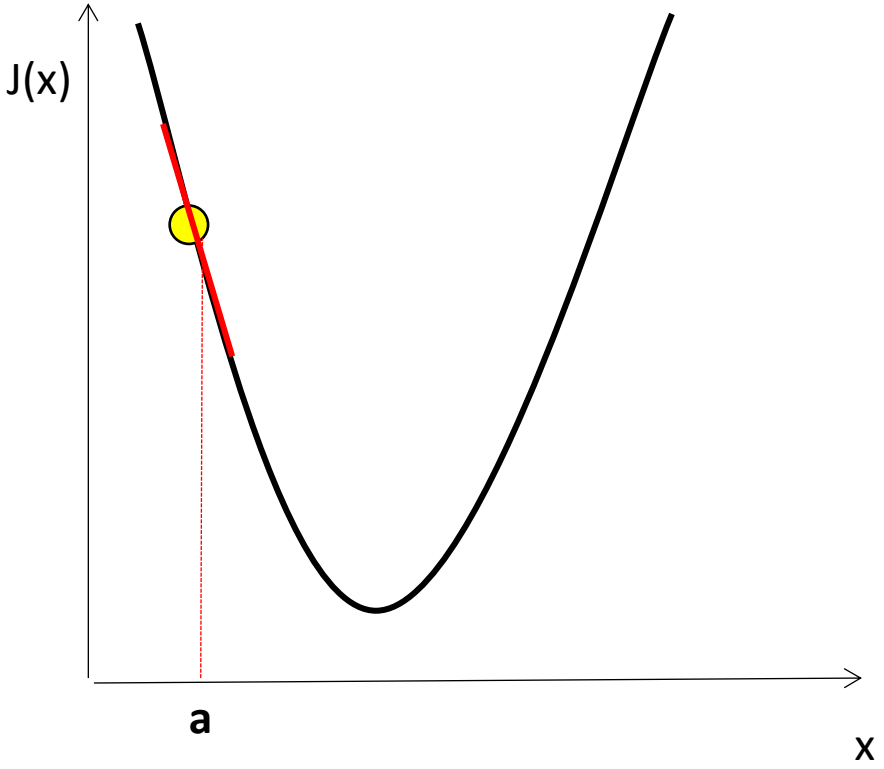
repeat until convergenc e

$$\{ \quad \theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j} \qquad (\text{for } j = 0, 1)$$

}

MACHINE INTELLIGENCE

Gradient Descent



- now let us begin again with initializing x at point b
- the slope of the tangent here is negative and hence our derivative term is negative too
- our x updation would be
- we end um moving right side which is again towards our minimum
- hence gradient descent is been doing well till now

repeat until convergenc e

$$\{ \quad \theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j} \qquad (\text{for } j = 0, 1)$$

}

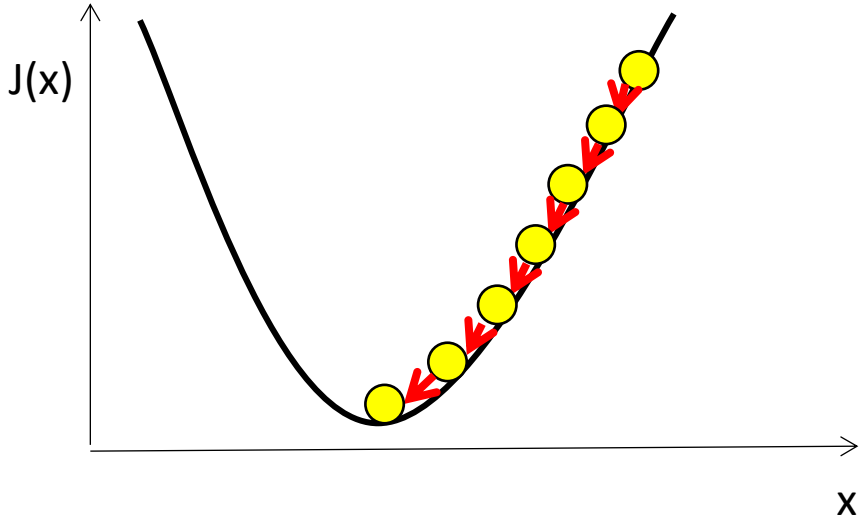
$$x = x - \alpha(negative \text{ number})$$

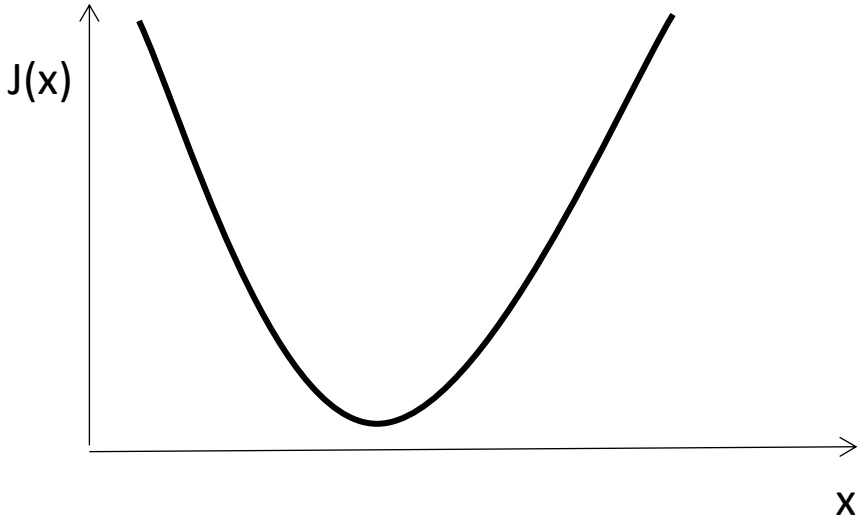
$$x = x - (-\beta) = x + \beta$$

$$(\alpha \text{ is always } + \text{ve})$$

MACHINE INTELLIGENCE

Gradient Descent





- now let us see what role does alpha play
- suppose we keep our alpha too small
- lets say we start a point x'
- if alpha is too small on each updation we are changing our x with some small value i.e taking small steps
- and again from new point we take again small steps to get to minimum
- so if alpha is small gradient descent can be slow to reach minimum

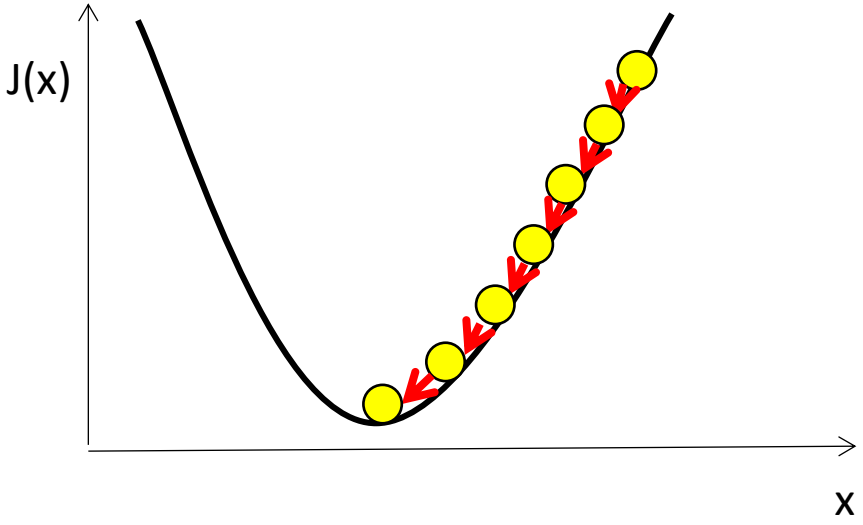
repeat until convergenc e

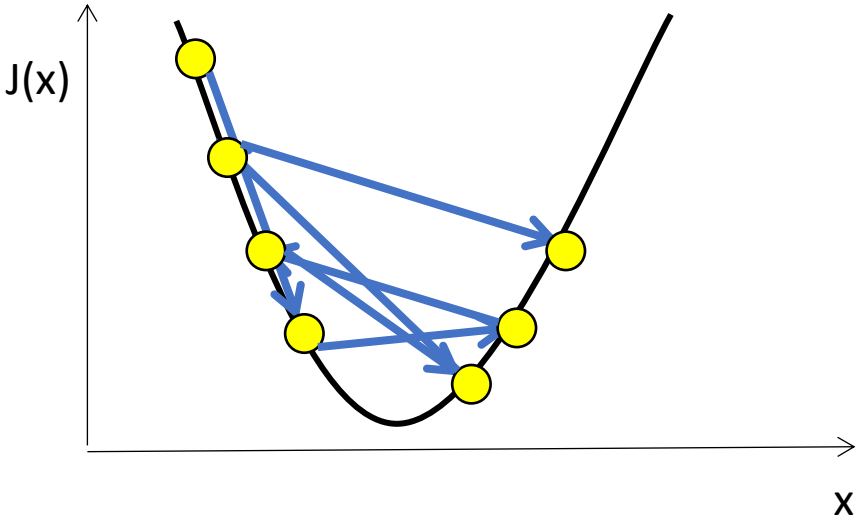
$$\{ \quad \theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j} \qquad (\text{for } j = 0, 1)$$

$$\}$$

MACHINE INTELLIGENCE

Gradient Descent





- if alpha is too large then gradient may overshoot to minimum or may never reach minimum
- suppose we start at point x''
- if alpha is too large we are taking a huge step
- say we take this step and reach at new point
- on next updation we take a huge step and go the right far then the minimum
- this may continue and we may never end up reaching minimum

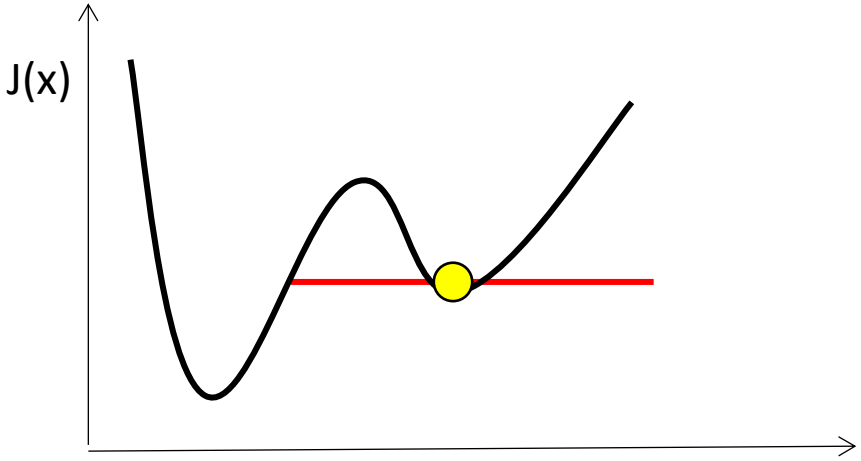
repeat until convergenc e

$$\{ \quad \theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j} \qquad (\text{for } j = 0, 1)$$

$$\}$$

MACHINE INTELLIGENCE

Gradient Descent



- now suppose our x is already at local minimum what would a gradient descent step do?
- at local minimum the slope would be zero
- and hence the derivative term is zero
- and hence it leaves our x unchanged
- this is the one problem with the gradient discent ,i.e you may get stuck in local minimum instead of global minimum

repeat until convergenc e

$$\{ \quad \theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j} \qquad (\text{for } j = 0, 1)$$

}

MACHINE INTELLIGENCE

Gradient Descent in optimizing perceptron learning

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$$

E($\vec{w}$) is the error in the output. We use the square of the error mainly for ease of differentiation.

t_d is the training output (expected) for the current data row.

o_d is the actual output of the perceptron for the "current" choice of weights.

- Let us define a error function for our perceptron learning
- E is the function of w ,i,e our weights
- Let us consider two input perceptron
- let x1,and x2 be the inputs and w_0, w_1, w_2 be bias and weights
- we represent $W = w_0 + w_1 * x_1 + w_2 * x_2$

MACHINE INTELLIGENCE

Gradient Descent in optimizing perceptron learning

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$$

- we will use gradient descent to move to minimum of our error function updating value of w_0, w_1, w_2 using the update rule
- thus we will reach minimum of the error function and end up finding optimal value for w_0, w_1, w_2

- Let us define a error function for our perceptron learning
- E is the function of w ,i,e our weights
- Let us consider two input perceptron
- let x1,and x2 be the inputs and w_0, w_1, w_2 be bias and weights
- we represent $\boldsymbol{W}=w_0+w_1*x_1+w_2*x_2$

THANK YOU

K.S.Srinivas
srinivasks@pes.edu
+91 80 2672 1983 Extn 701

MACHINE INTELLIGENCE

Multi layer Network and Back propagation

K.S.Srinivas

Department of Computer Science and Engineering

MACHINE INTELLIGENCE

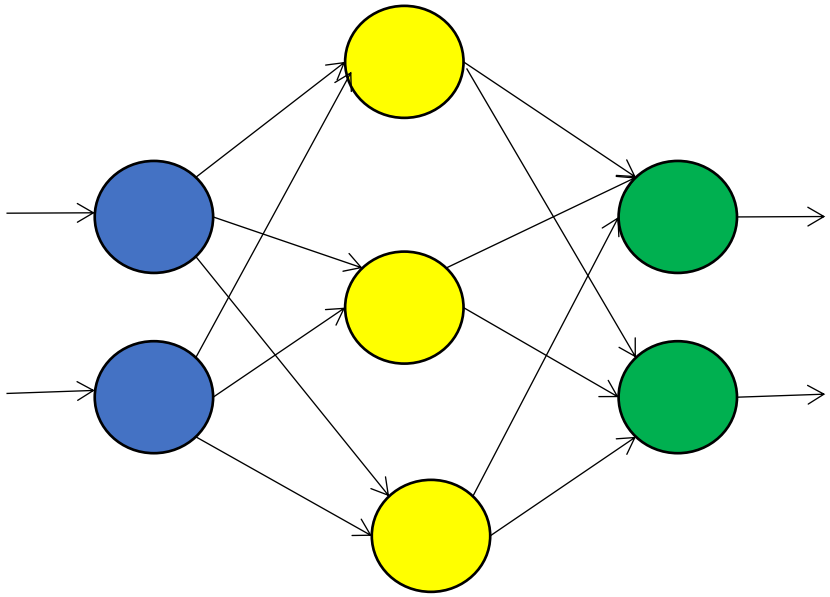
Multi layer Network and Back propagation

K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

Multi layer Network

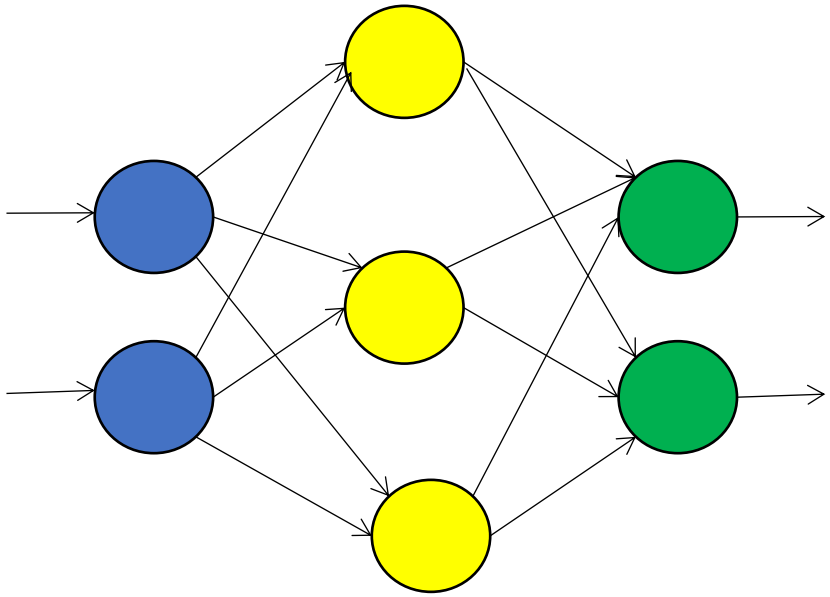
- while neurons are really cool,we cannot just use a single neuron to perform complex tasks.
- This is the reason our brain has billions of neuron,stacked in layers forming a network .
- Similarly ,artificial neurons are arranged in layers.
- Each and every layer will be connected in such a way that information is passed from one layer to another.
- A typical ANN consists of the following layers:
  - INPUT LAYER
  - HIDDEN LAYER
  - OUTPUT LAYER
- Each layer has a collection of neurons and the neurons in one layer interact with all the neurons in other layers.
- We use the term **nodes or units** to represent the neurons in an ANN



MACHINE INTELLIGENCE

INPUT LAYER

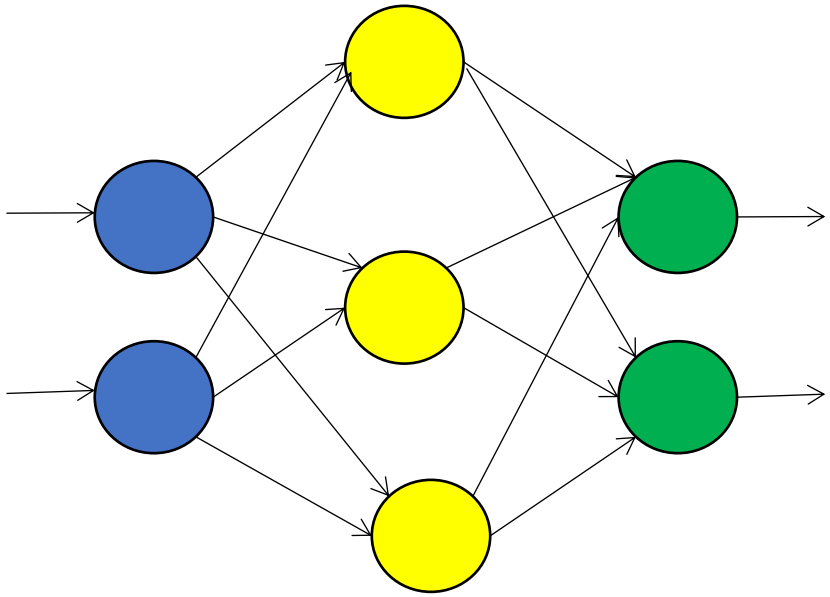
- The input layer is where we feed in inputs to the network .
- The number of units in input layer is the number of inputs we feed to the network.
- No computation is performed in the input layer



MACHINE INTELLIGENCE

HIDDEN LAYER

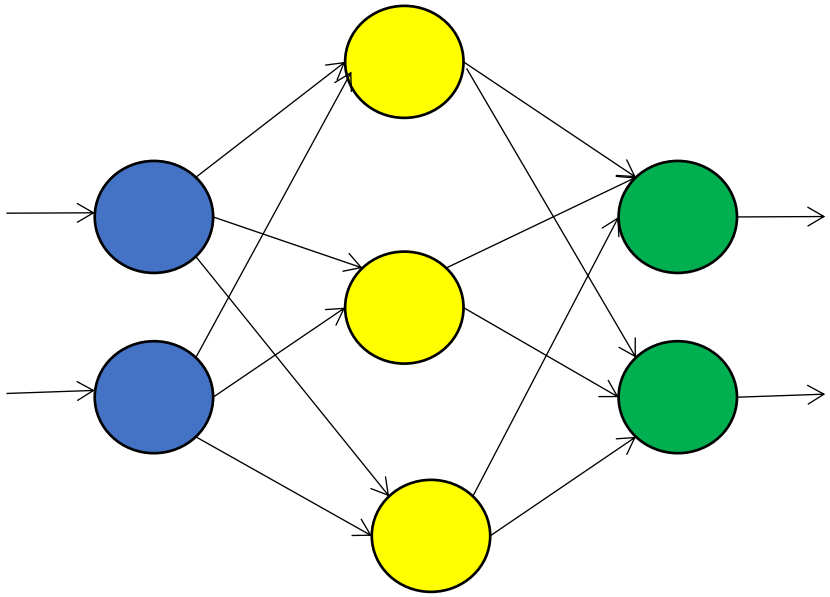
- Any layer between the input layer and the output layer is called hidden layers.
- It process the input received from input layer.
- The hidden layer is responsible for deriving complex relationships between input and output .
- That is hidden layer identifies the pattern in the data set.
- Main core for learning
- There can be n number of hidden layer according to our use case.



MACHINE INTELLIGENCE

OUTPUT LAYER

- After processing the input the hidden layer sends its results to the output layer.
- As the name suggests the output layer emits output



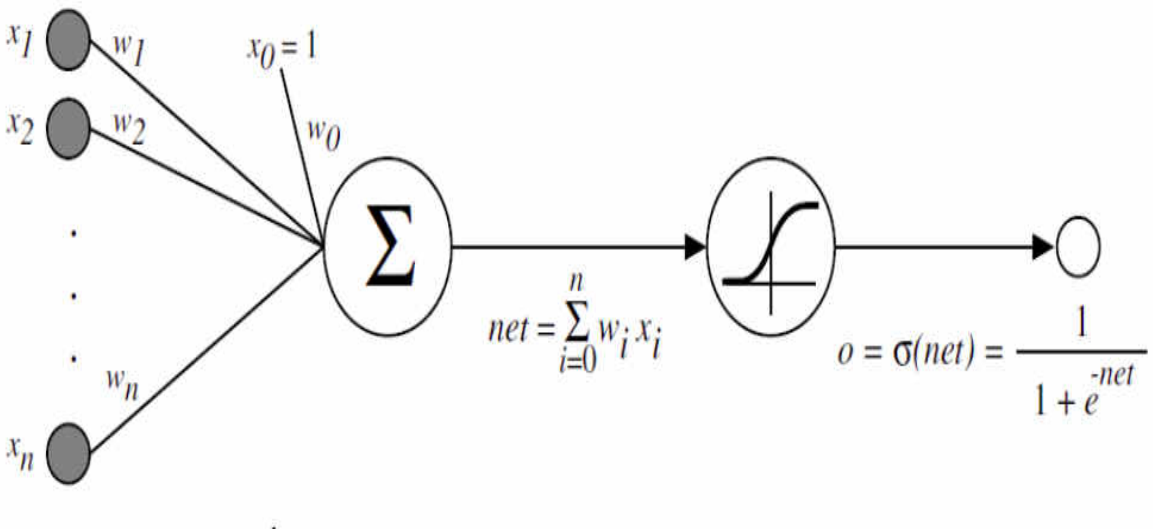
MACHINE INTELLIGENCE

Activation Functions

- An activation function,also known as transfer function plays vital role in neural networks.
- It is used to introduce a non-linearity in neural networks.
- We apply activation function to the inputs which is multiplied by weights and added to bias ,i.e f(z),where z=(inputs*weights)+bias
- If we do not apply the activation function,then a neuron simply resembles the linear regression.
- ex for an activation function is
  - Sigmoid Activation Function:

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

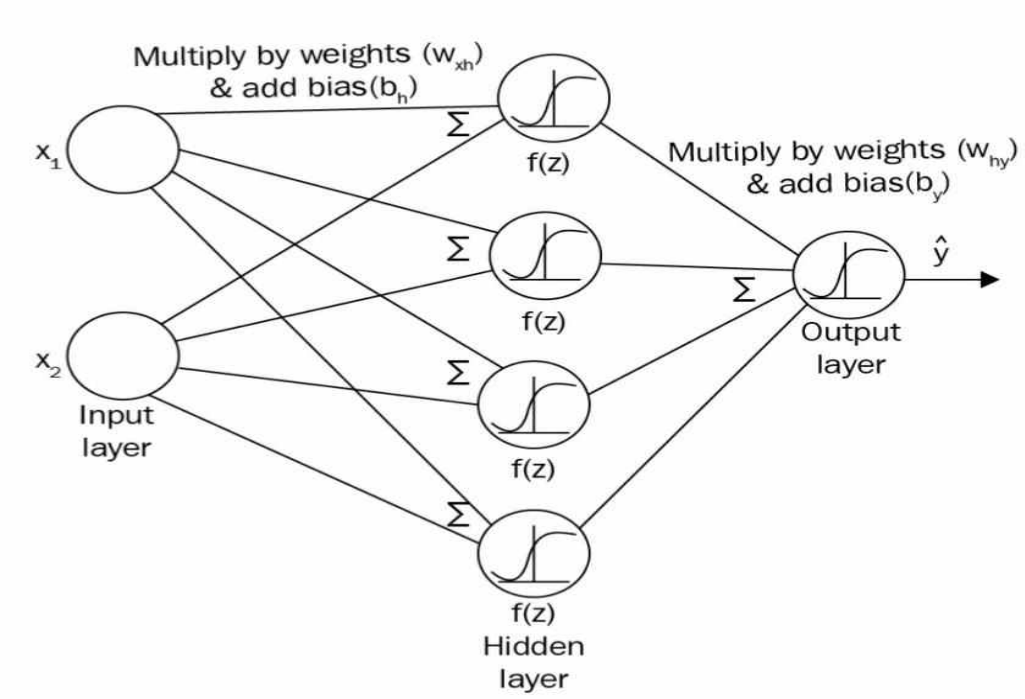
  - we will learn in deep about activation function in upcoming sessions



MACHINE INTELLIGENCE

Forward propagation in ANN

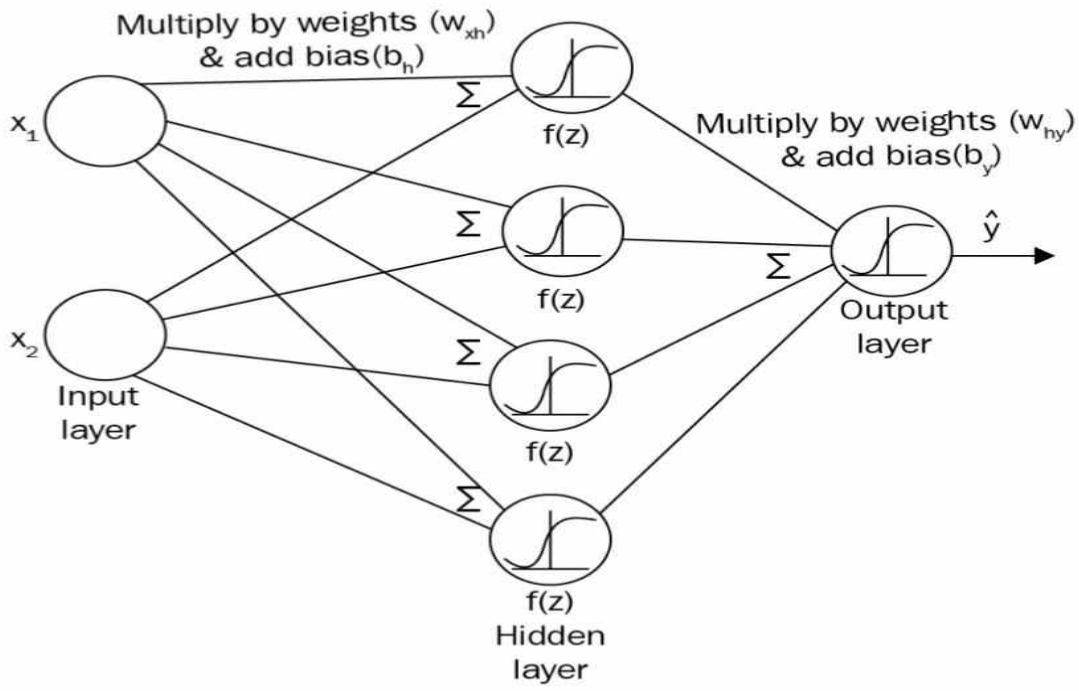
- the number of layer in an ANN is number of hidden layer +output layer
- consider the following two layer neural network
- inputs--->2 (x1,x2)
- hidden layer units --->>4
- output layer units -->>>1
- Now the inputs will be multiplied by weights and we add bias and propagate the result value to the hidden layer where the activation function will be applied
- Before that we need to initialize the weight matrix.
- In real world we don't know which input is more important than the other so that we can weight them accordingly to compute the output.
- hence we randomly initialize weights and the bias value.



MACHINE INTELLIGENCE

Forward propagation in ANN

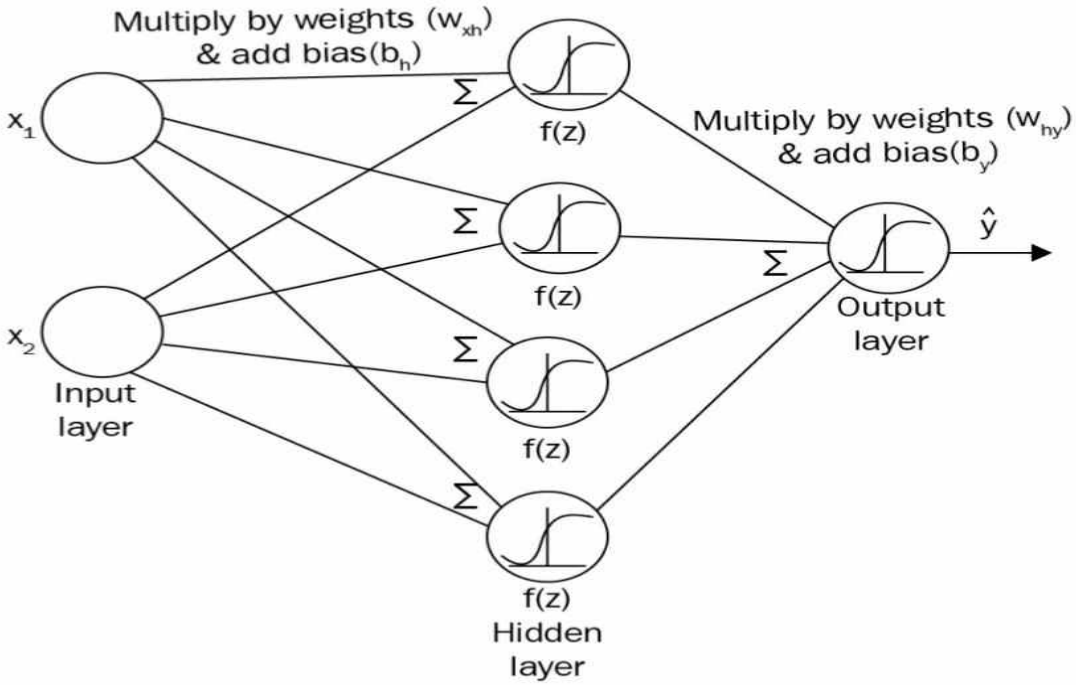
- Let the weight and bias between input and the hidden layer be represented as $\mathbf{W_{xh}}$ **and** $\mathbf{b_h}$ respectively.
- let $\mathbf{Z1=XW_{xh}+b_h}$
- now this is passed through the hidden layer ,we apply activation function here (sigmoid)
- $\mathbf{\alpha 1=\sigma(z1)}$
- This makes our inputs to the output layer ready
- we then multiply this $\mathbf{\alpha 1}$ with weight matrix $\mathbf{W_{hy}}$ and add with bias $\mathbf{b_y}$,which is weight matrix and bias between hidden layer and output layer.
- we get $\mathbf{Z2=\alpha 1 W_{hy}+b_y}$
- which is then passed to sigmoid activation function in the output layer
- we get our final output as
- $\mathbf{\hat{y}=\sigma(z2)}$



MACHINE INTELLIGENCE

Forward propagation in ANN

- This whole process from the input layer to the output layer is known as forward-propagation.
- Forward Propagation is cool,but how do we know whether the output generated by the neural network is correct?
- The answer is defining a cost function and backpropgating the error



MACHINE INTELLIGENCE

Backward propagation in ANN

- Iteratively process a set of training tuples & compare the network's prediction with the actual known target value

- For each training tuple, the weights are modified to **minimize the loss function** between the network's prediction and the actual target value

- Modifications are made in the "**backwards**" direction: from the output layer, through each hidden layer down to the first hidden layer, hence "**back propagation**"

Backward propagation in ANN

- let us first define our loss function

Error as a function the weight vector for a single output unit:

Stochastic approximation:

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$$

Error function for multiple output units:

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$$

Derivation of Back propagation Rule

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- $Downstream(j)$ = the set of units whose immediate inputs include the output of unit j

Derivation of Back propagation Rule

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \qquad E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} x_{ji}$$

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$$

Derivation of Back propagation Rule

Case 1: Training Rule for Output Unit Weights.

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k \in outputs}(t_k - o_k)^2 = \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2$$

$$= \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j}$$

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j} = o_j(1 - o_j)$$

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\, o_j(1 - o_j)$$

$$\Delta w_{ji} = \eta\, \delta_j\, x_{ji}$$

$$\Delta w_{ji} = -\eta\frac{\partial E_d}{\partial w_{ji}} = \eta\, (t_j - o_j)\, o_j(1 - o_j)x_{ji}$$

Derivation of Back propagation Rule

Case 2: Training Rule for Hidden Unit Weights.

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \frac{\partial o_j}{\partial net} = \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)$$

using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k \, w_{kj}$$

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$$

Backward propagation in ANN

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji} \quad \text{where} \quad \Delta w_{ji} = \eta\, \delta_j\, x_{ji}$$

Over fitting

Analogous to the Decision Tree case.

With several weights to update and several iterations, the Back propagation Algorithm over fits.

Over fitting occurs because the weights are being **tuned to fit idiosyncrasies** of the training examples that are not representative of the general distribution of examples.

A solution:

Maintain a different **validation set**, apart from training set and test set.

Every now and then measure the accuracy over the validation set.

Choose the set of weights that produce the **least validation-set error**.

THANK YOU

K.S.Srinivas
srinivasks@pes.edu
+91 80 2672 1983 Extn 701

MACHINE INTELLIGENCE

Back propagation

K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

Back propagation

K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

Back Propagation

- Before we jump into back propagation ,lets first check derivative of the sigmoid activation that we previously learned about

MACHINE INTELLIGENCE

Derivative of Sigmoid

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

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

$$= \frac{[(1+e^{-x})*\frac{d(1)}{dx}]-[1*\frac{d(1+e^{-x})}{dx}]}{(1+e^{-x})^2}$$

$$= \frac{[(1+e^{-x})*0]-[1*-e^{-x}]}{(1+e^{-x})^2}$$

$$= \frac{e^{-x}}{(1+e^{-x})^2} \qquad = \frac{e^{-x}+1-1}{(1+e^{-x})^2}$$

$$= \frac{e^{-x}+1}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} \qquad = \frac{1}{(1+e^{-x})} - \frac{1}{(1+e^{-x})^2}$$

$$= \frac{1}{(1+e^{-x})}\left(1 - \frac{1}{(1+e^{-x})}\right) \qquad = \sigma(x)*\ (1-\sigma(x))$$

RECALL

Quotient Rule

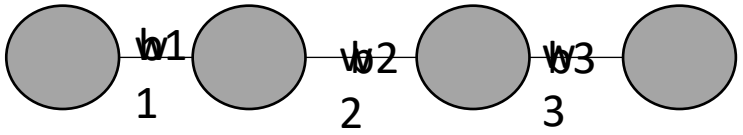
$$\frac{d}{dx}\frac{f(x)}{g(x)} = \frac{(g(x)-f'(x))-(f(x)-g'(x))}{g(x)^2}$$

Your task!!!

try finding derivative of tanh function

MACHINE INTELLIGENCE

Back Propagation

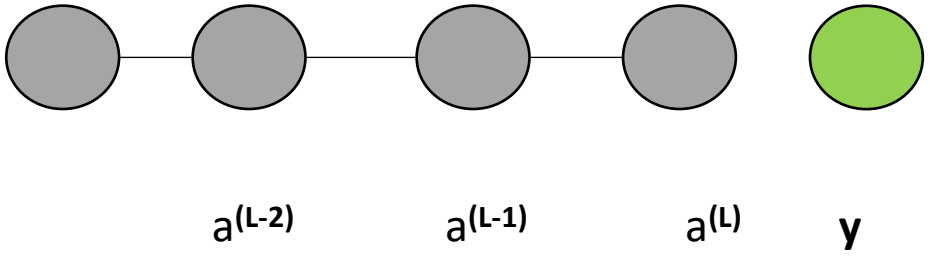


E(w1,b1,w2,b2,w3,b4)

- let us start with a simple network,where each layer has a single perceptron,
- determined by three weights and three bias
- our go is to understand how sensitive is our cost function is to these variables
- that way we know which adjustment to these terms is going to cause the most efficient decrease to the cost function

MACHINE INTELLIGENCE

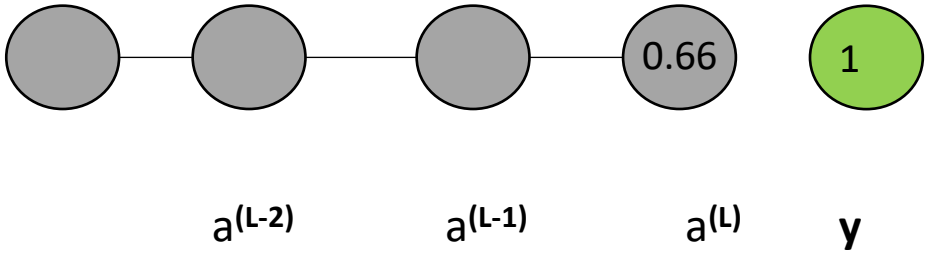
Back Propagation



- Lets label the activation of that last neuron a with a superscript L,indicating which layer its in.
- Lets say that the value we want the last activation to be for a given training example is y

MACHINE INTELLIGENCE

Back Propagation



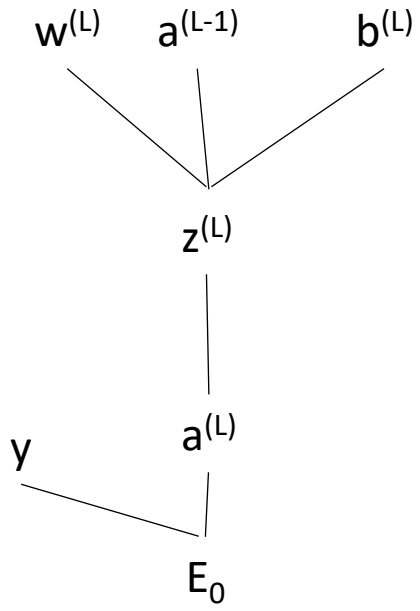
- y can be 0 or 1,so the cost for this simple network for a single training example is $(a^L-y)^2$
- Remember what is a^L
- Let us make things simpler
- we can generalize it in the following way

$E_0(.....)= (0.66-1)^2$

$a^{(L)}= \sigma(w3\ a^{(L-1)}+b3)$

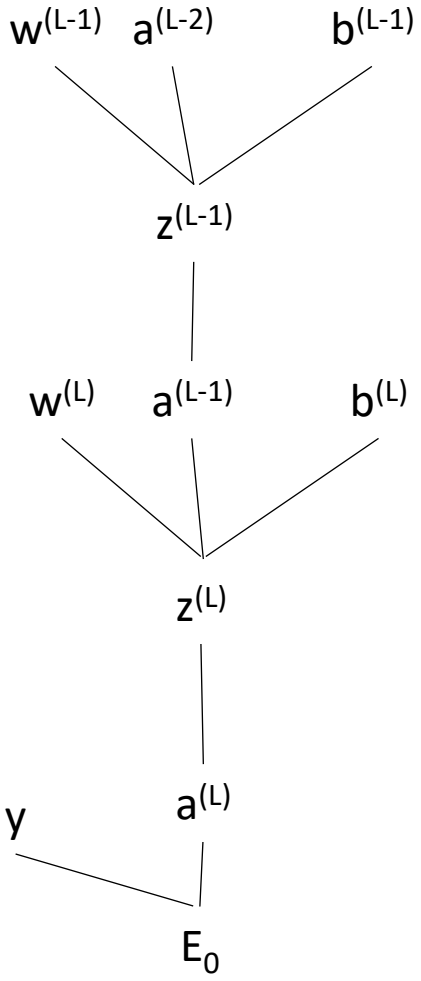
$z^{(L)}= w3\ a^{(L-1)}+b3$

$a^{(L)}= \sigma\ (z^{(L)})$



MACHINE INTELLIGENCE

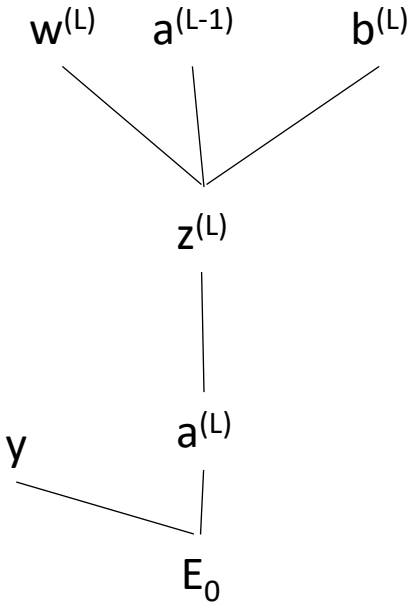
Back Propagation



- also a(L-1) is influenced by its own weight
- Now lets not focus on it,
- The thing is all of this is numbers
- our first goal is to understand how sensitive our cost function is to small changes in our weight

MACHINE INTELLIGENCE

Back Propagation

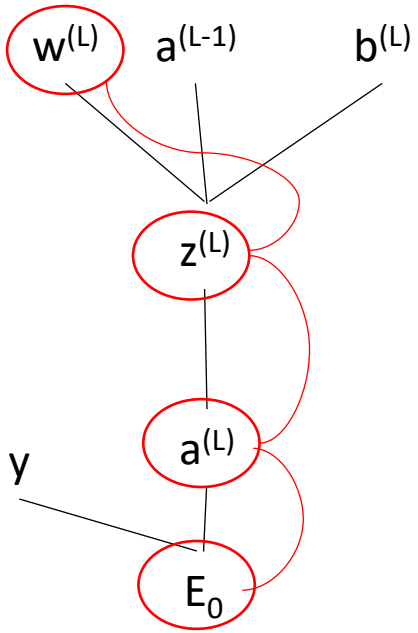


$$\frac{\partial E_0}{\partial w^{(L)}}$$

- In other words what is the derivative of E with respect to W(L)
- this ∂w can be said as small nudge in w like ,change by 0.01
- and ∂E is whatever the resulting nudge to cost E
- what we want is their ratio

MACHINE INTELLIGENCE

Back Propagation

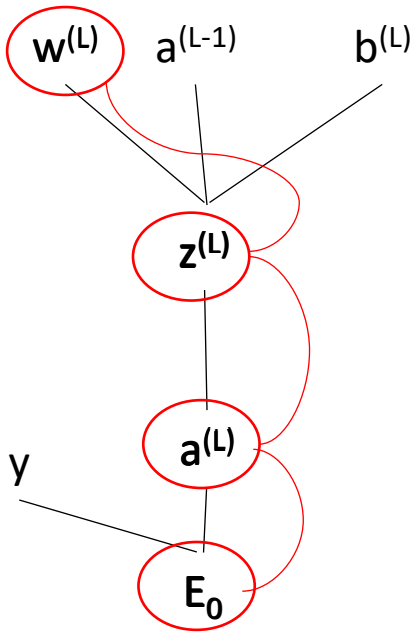


$$\frac{\partial E_0}{\partial w^{(L)}}$$

- conceptually this tiny nudge to w^(L) causes some nudge to z^(L) ,
- which in turn causes some nudge to a^(L)
- which directly influences the cost

MACHINE INTELLIGENCE

Back Propagation



- so with that let us break things up
- This is nothing but the chain rule

$$\frac{\partial E_0}{\partial w^{(L)}} = \frac{\partial z^L}{\partial w^{(L)}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial E_0}{\partial a^{(L)}}$$

Back Propagation

$$\frac{\partial E_0}{\partial w^{(L)}} = \frac{\partial z^L}{\partial w^{(L)}} \quad \frac{\partial a^L}{\partial z^{(L)}} \quad \frac{\partial E_0}{\partial a^{(L)}}$$

- Let us recall eqn of E
- The derivative of this wrt to a^L would be

$$E_0 = (a^L - y)^2$$

$$\frac{\partial E_0}{\partial a^{(L)}} = 2(a^L - y)$$

Back Propagation

$$\frac{\partial E_0}{\partial w^{(L)}} = \frac{\partial z^L}{\partial w^{(L)}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial E_0}{\partial a^{(L)}}$$

- Let us recall eqn of a^(L)
- The derivative of this wrt to z^L would be

$$a^L = \sigma(z^L)$$

$$\frac{\partial a^L}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

$$\frac{\partial E_0}{\partial a^{(L)}} = 2(a^L - y)$$

Back Propagation

$$\frac{\partial E_0}{\partial w^{(L)}} = \frac{\partial z^L}{\partial w^{(L)}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial E_0}{\partial a^{(L)}}$$

- now lets check the equation of z^L
- The derivative of this wrt to w^L would be

$$z^L = w^L . a^{L-1} + b^L$$

$$\frac{\partial z^L}{\partial w^{(L)}} = a^{L-1}$$

$$\frac{\partial E_0}{\partial a^{(L)}} = 2(a^L - y)$$

$$\frac{\partial a^L}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

Back Propagation

$$\frac{\partial E_0}{\partial w^{(L)}} = \frac{\partial z^L}{\partial w^{(L)}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial E_0}{\partial a^{(L)}}$$

- putting all this together

$$\frac{\partial E_0}{\partial w^{(L)}} = a^{L-1} \sigma'(z^{(L)}) 2(a^L - y)$$

$$\frac{\partial E_0}{\partial a^{(L)}} = 2(a^L - y)$$

$$\frac{\partial a^L}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

$$\frac{\partial z^L}{\partial w^{(L)}} = a^{L-1}$$

Back Propagation

$$\frac{\partial E_0}{\partial w^{(L)}} = \frac{\partial z^L}{\partial w^{(L)}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial E_0}{\partial a^{(L)}}$$

- putting all this together
- this is for one example ,generalizing for n examples

$$\frac{\partial E_0}{\partial w^{(L)}} = a^{L-1} \sigma'(z^{(L)}) 2(a^L - y)$$

$$\frac{\partial E}{\partial w} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial E_k}{\partial w}$$

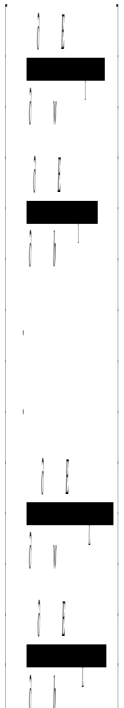
MACHINE INTELLIGENCE

Back Propagation

$$\frac{\partial E_0}{\partial w^{(L)}} = \frac{\partial z^L}{\partial w^{(L)}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial E_0}{\partial a^{(L)}}$$

$$\frac{\partial E}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial E_k}{\partial w^{(L)}}$$

- this is just one component of the gradient
- even though its one of many partial derivatives we need its more than 50% of our work



Back Propagation

$$\frac{\partial E_0}{\partial w^{(L)}} = \boxed{\frac{\partial z^L}{\partial w^{(L)}}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial E_0}{\partial a^{(L)}}$$

- the derivative wrt to bias is almost identical
- we just need to replace this with derivative of z^L wrt to b^L
- and that turns out to be 1

$$z^L = w^L . a^{L-1} + b^L$$

$$\frac{\partial z^L}{\partial b^{(L)}} = 1$$

$$\frac{\partial E_0}{\partial b^{(L)}} = 1\, \sigma'(z^{(L)}) 2(a^L - y)$$

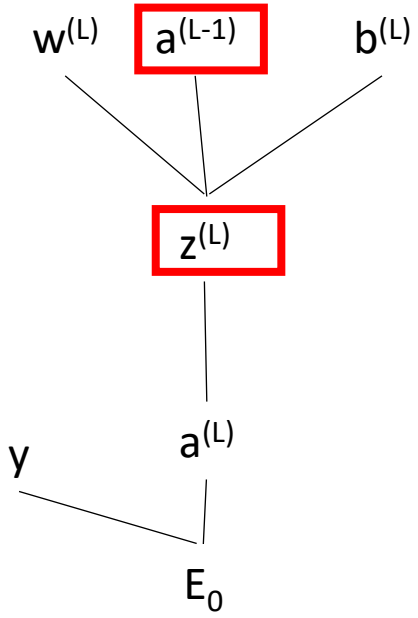
$$\frac{\partial E}{\partial w} = \frac{1}{n}\sum_{k=0}^{n-1} \frac{\partial E_k}{\partial w}$$

$$\frac{\partial E_i}{\partial w^{(L)}} = a^{L-1}\, \sigma'(z^{(L)})\, 2(a^L - y)$$

MACHINE INTELLIGENCE

Back Propagation

$$\frac{\partial E_0}{\partial w^{(L)}} = \frac{\partial z^L}{\partial w^{(L)}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial E_0}{\partial a^{(L)}}$$



- this were topic of back propagation comes in

$$\frac{\partial E}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial E_k}{\partial w^{(L)}}$$

$$\frac{\partial E_i}{\partial w^{(L)}} = a^{L-1} \sigma'(z^{(L)}) \, 2(a^L - y)$$

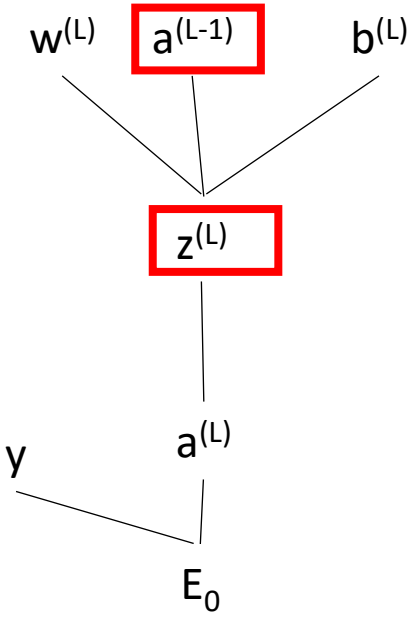
$$\frac{\partial E}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial E_k}{\partial w^{(L)}}$$

$$\frac{\partial E_i}{\partial w^{(L)}} = a^{L-1} \sigma'(z^{(L)}) \, 2(a^L - y)$$

MACHINE INTELLIGENCE

Back Propagation

$$\frac{\partial E_0}{\partial a^{(L-1)}} = \frac{\partial z^L}{\partial a^{(L-1)}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial E_0}{\partial a^{(L)}}$$



- this were topic of back propagation comes in

$$\frac{\partial E}{\partial w^{(L)}} = \frac{1}{n_{L-1}} \sum_{k=0}^{n_{L-1}} \frac{\partial E_k}{\partial w^{(L)}}$$

$$\frac{\partial E_i}{\partial w^{(L)}} = a^{L-1} \sigma'(z^{(L)}) \, 2(a^L - y)$$

$$\frac{\partial E}{\partial w^{(L)}} = \frac{1}{n_{L-1}} \sum_{k=0}^{n_{L-1}} \frac{\partial E_k}{\partial w^{(L)}}$$

$$\frac{\partial E_i}{\partial w^{(L)}} = a^{L-1} \sigma'(z^{(L)}) \, 2(a^L - y)$$

Back Propagation

$$\frac{\partial E_0}{\partial a^{(L-1)}} = \frac{\partial z^L}{\partial a^{(L-1)}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial E_0}{\partial a^{(L)}}$$

- let us bring back the equation again and see the derivative

$$z^L = w^L . a^{L-1} + b^L$$

$$\frac{\partial z^L}{\partial a^{(L-1)}} = w^L$$

$$\frac{\partial E_0}{\partial a^{(L-1)}} = w^L \ \sigma'(z^{(L)}) 2(a^L - y)$$

$$\frac{\partial E}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial E_k}{\partial w^{(L)}}$$

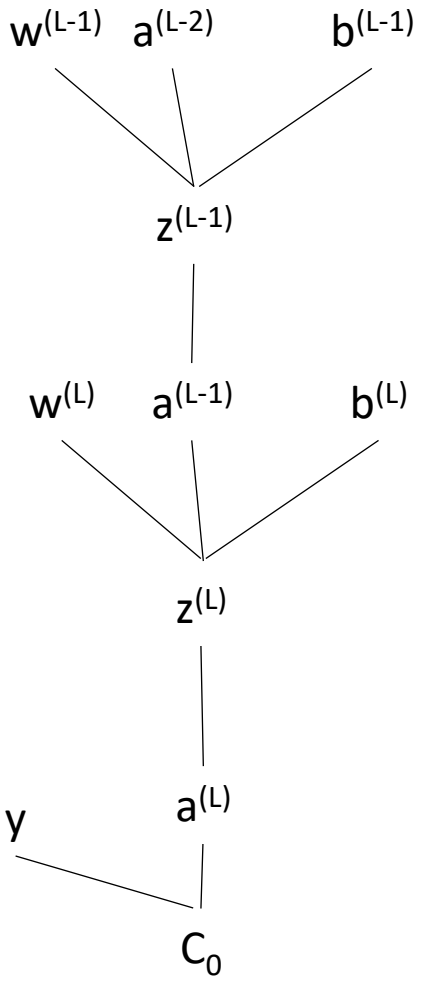
$$\frac{\partial E_i}{\partial w^{(L)}} = a^{L-1} \ \sigma'(z^{(L)}) \ 2(a^L - y)$$

$$\frac{\partial E}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial E_k}{\partial w^{(L)}}$$

$$\frac{\partial E_i}{\partial w^{(L)}} = a^{L-1} \ \sigma'(z^{(L)}) \ 2(a^L - y)$$

MACHINE INTELLIGENCE

Back Propagation



- now we can keep iterating this chain rule idea backwards,to see how sensitive the cost function is to the previous weights and biases

$$\frac{\partial E}{\partial w} = \frac{1}{n}\sum_{k=0}^{n-1}\frac{\partial E_k}{\partial w}$$

$$\frac{\partial E_i}{\partial w^{(L)}} = a^{L-1}\,\sigma'(z^{(L)})\,2(a^L - y)$$

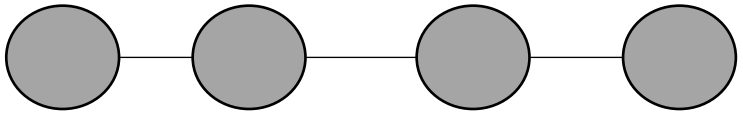
$$\frac{\partial E}{\partial w} = \frac{1}{n}\sum_{k=0}^{n-1}\frac{\partial E_k}{\partial w}$$

$$\frac{\partial E_i}{\partial w^{(L)}} = a^{L-1}\,\sigma'(z^{(L)})\,2(a^L - y)$$

$$\frac{\partial E_i}{\partial a^{(L-1)}} = w^L\,\sigma'(z^{(L)})\,2(a^L - y)$$

MACHINE INTELLIGENCE

Back Propagation

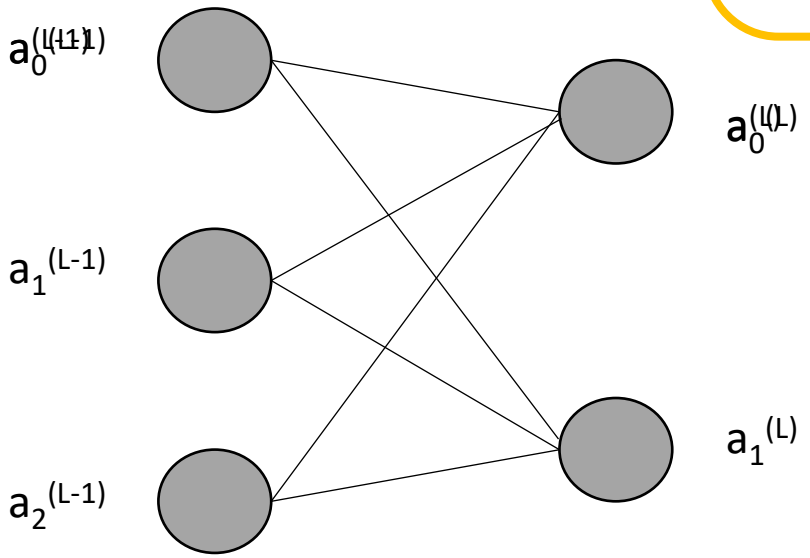


- this look simple since all layers all one neuron
- and things is going to get exponentially complicated when we give more layers to neuron
- But honestly,not that much changes
- its just few more indices we need to keep track of

MACHINE INTELLIGENCE

Back Propagation

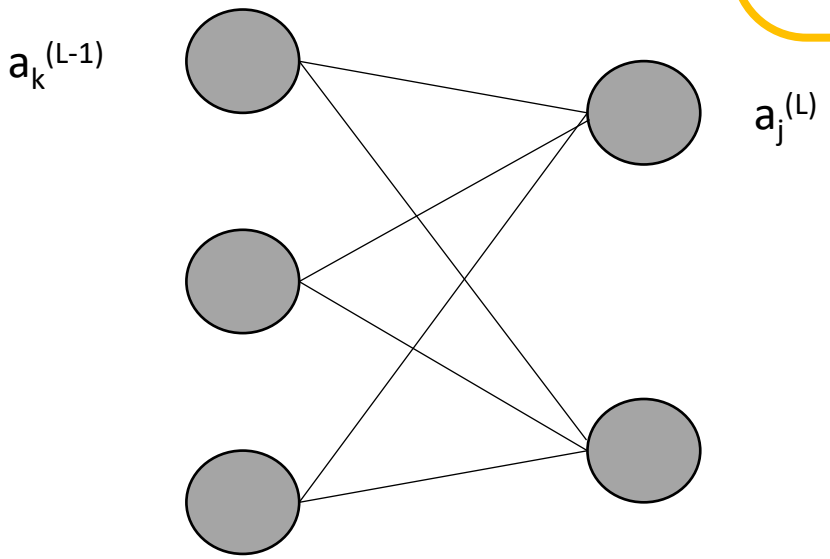
- rather than activation of a given layer being a^L,its also going it have a subscript indicating which neuron of that layer it is



MACHINE INTELLIGENCE

Back Propagation

- lets use letter k to index the layer L-1 and letter j to index the layer L

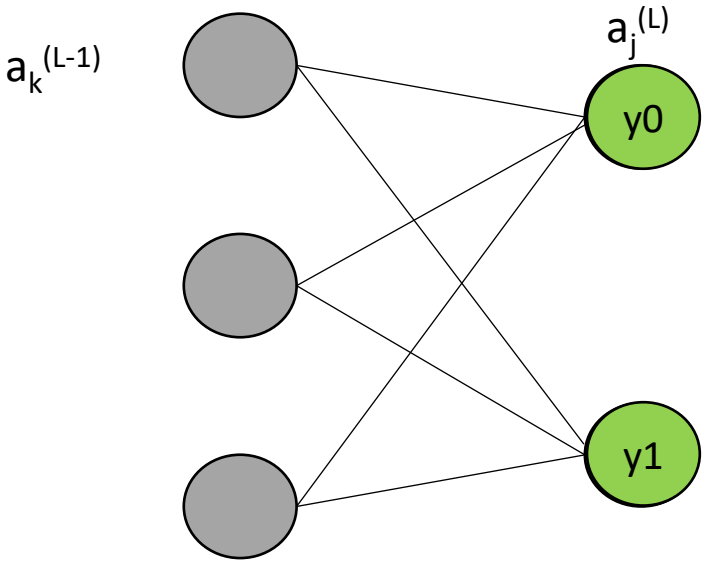


MACHINE INTELLIGENCE

Back Propagation

$$E_0 = \sum_{j=0}^{n_L - 1} (a^L{}_j - y_i)^2$$

- for the cost ,again we look at what the desired output is ,but this time we add up the squares of the differences between the last layers activation and the desired output

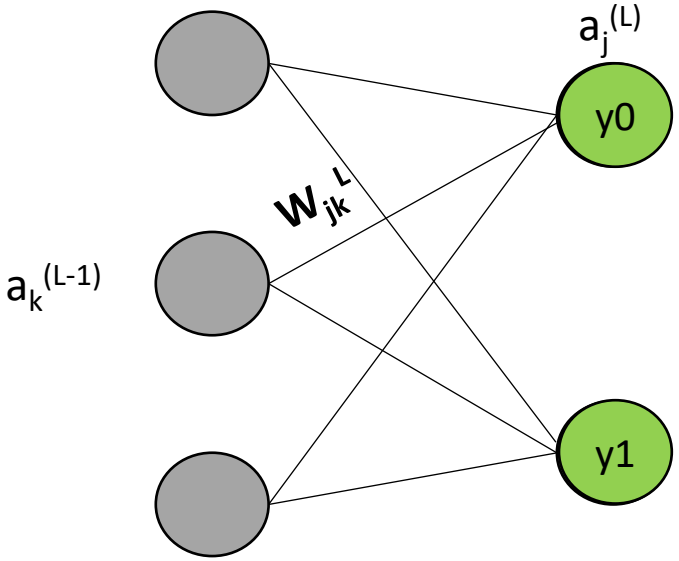


MACHINE INTELLIGENCE

Back Propagation

$$E_0 = \sum_{j=0}^{n_L - 1} (a^L{}_j - y_i)^2$$

- since there lot more weights ,each one has to have couple more indices to keep track of where it is
- let call the weight of the edge connecting the kth neuron in L-1 layer to jth neuron in L layer as W_{jk}^L

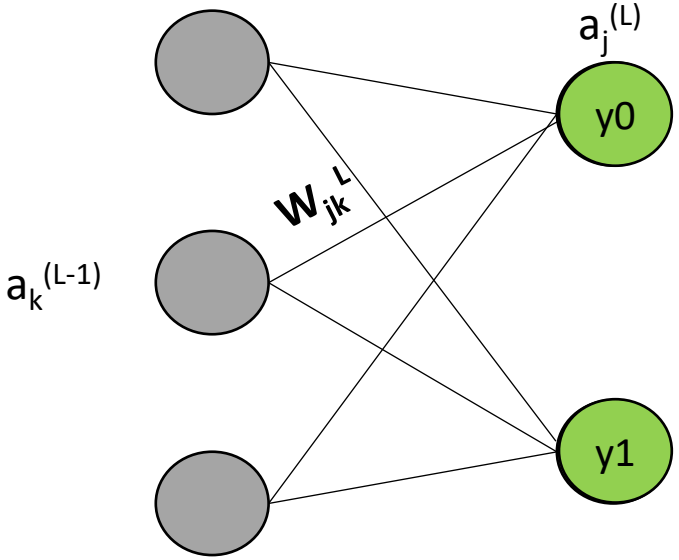


MACHINE INTELLIGENCE

Back Propagation

$$z_j^L = w_{j0}^L a_0^{L-1} + w_{j1}^L a_1^{L-1} + w_{j2}^L a_2^{L-1} + b_j^L$$

$$a_j^L = \sigma(z^L{}_j)$$



- just as before ,its nice to give name to relevant weighted sum,like z
- so that activation of the last layer from the j neuron is just our special function applied to z

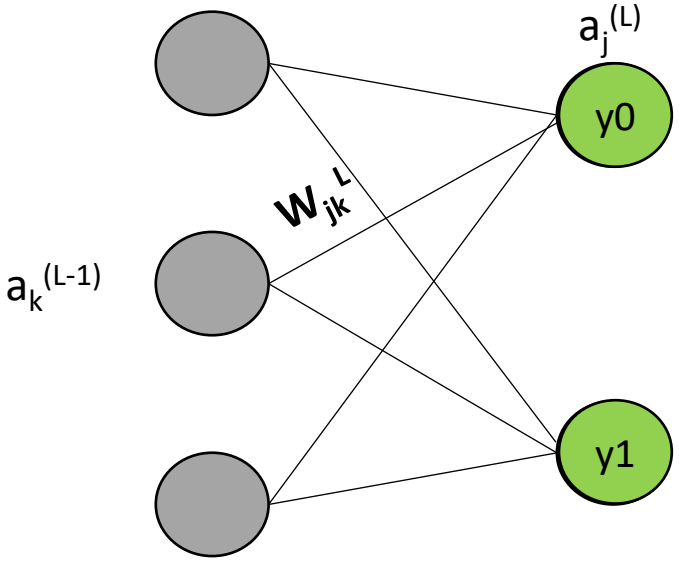
$$E_0 = \sum_{j=0}^{n_L - 1} (a^L{}_j - y_i)^2$$

MACHINE INTELLIGENCE

Back Propagation

$$\frac{\partial E_0}{\partial w_{jk}^L} = \frac{\partial z_j^L}{\partial w_{jk}^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial E_0}{\partial a_j^L}$$

- similarly our derivative terms that we derived previously gets indices ,take a pause and analyse



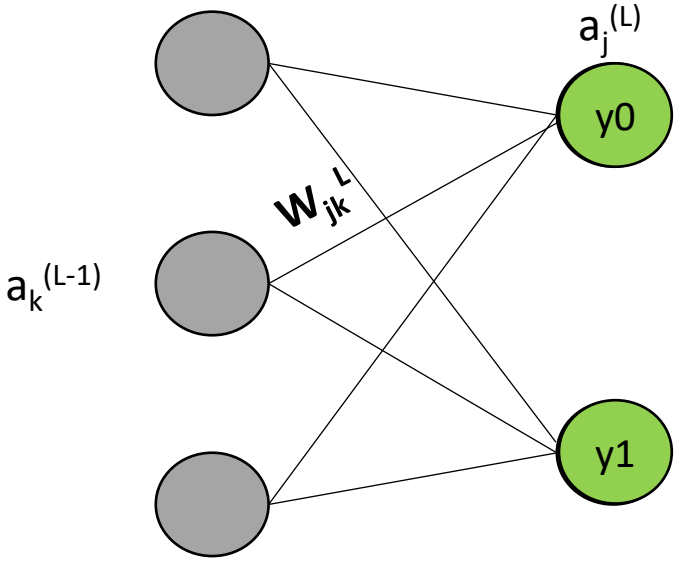
$$E_0 = \sum_{j=0}^{n_L - 1} (a^L{}_j - y_i)^2$$

MACHINE INTELLIGENCE

Back Propagation

$$\frac{\partial E_0}{\partial a_k^{L-1}} = \sum_{j=0}^{n_L-1} \frac{\partial z_j^L}{\partial w_{jk}^L} \frac{\partial a_j^L}{\partial z_j^L} \quad \frac{\partial E_0}{\partial a_j^L}$$

- what doe change is derivative of cost with respect to one of the activation in the layer (L-1)
- in this case the difference is the neuron influences the cost function through multiple paths

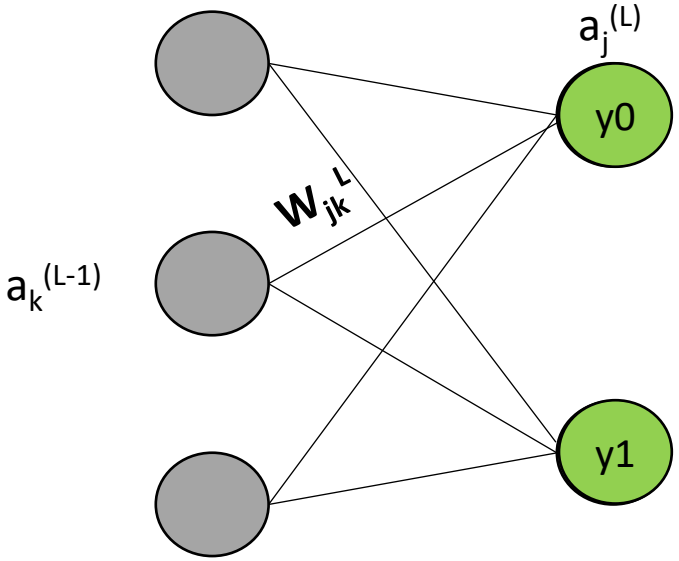


$$E_0 = \sum_{j=0}^{n_L-1} (a^L{}_j - y_i)^2$$

$$\frac{\partial E_0}{\partial w_{jk}^L} = \frac{\partial z_j^L}{\partial w_{jk}^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial E_0}{\partial a_j^L}$$

$$\frac{\partial E_0}{\partial a_k^{L-1}} = \sum_{j=0}^{n_L-1} \frac{\partial z_j^L}{\partial w_{jk}^L} \frac{\partial a_j^L}{\partial z_j^L} \quad \frac{\partial E_0}{\partial a_j^L}$$

- once we know how sensitive the cost function is to the previous layer we can just repeat the process for other previous layer



$$E_0 = \sum_{j=0}^{n_L-1} (a^L{}_j - y_i)^2$$

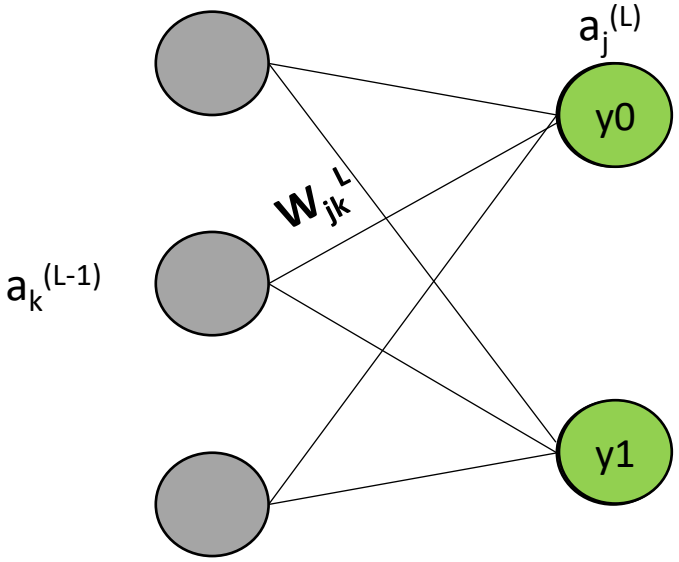
$$\frac{\partial E_0}{\partial w_{jk}^L} = \frac{\partial z_j^L}{\partial w_{jk}^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial E_0}{\partial a_j^L}$$

MACHINE INTELLIGENCE

Back Propagation

$$\frac{\partial E_0}{\partial a_k^{L-1}} = \sum_{j=0}^{n_L-1} \frac{\partial z_j^L}{\partial w_{jk}^L} \frac{\partial a_j^L}{\partial z_j^L} \quad \frac{\partial E_0}{\partial a_j^L}$$

- once we know how sensitive the cost function is to the previous layer we can just repeat the process for other previous layer



$$E_0 = \sum_{j=0}^{n_L-1} (a^L{}_j - y_i)^2$$

$$\frac{\partial E_0}{\partial w_{jk}^L} = \frac{\partial z_j^L}{\partial w_{jk}^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial E_0}{\partial a_j^L}$$

MACHINE INTELLIGENCE

Back Propagation

$$\frac{\partial E}{\partial w_{ij}} = \left[\sum_k \frac{\partial E}{\partial a_j}\right] \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

- once before we conclude let us see the back propagation between the input layer and first hidden layer .
- where j is the first hidden layer and i is the input layer,and let k be the output layer
- lets us use the same chain rule friend

MACHINE INTELLIGENCE

Back Propagation

$$\frac{\partial E}{\partial w_{ij}} = \left[\sum_k \frac{\partial E}{\partial a_j}\right] \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

$$= \left[\sum_k \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial z_k} \frac{\partial z_k}{\partial a_j}\right] \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

$$= \left[\sum_k (y_k - a_k)\sigma'(z_k) w_{jk}\right] \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

$$= \left[\sum_k (y_k - a_k)\sigma'(z_k) w_{jk}\right] \sigma'(z_j) a_i$$

- let us expand it more again using chain rule
- we have already derived similar equation previously ,we will use the same knowledge

$$\frac{\partial E}{\partial w_{ij}} = \left[\sum_k (y_k - a_k)\sigma'(z_k) w_{jk}\right] \sigma'(z_j) a_i$$

MACHINE INTELLIGENCE

Back Propagation

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial z_k}\frac{\partial z_k}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial a_k} = 2(a_k - y)$$

$$\frac{\partial z_k}{\partial w_{jk}} = a_j$$

$$\frac{\partial E}{\partial w_{jk}} = 2(a_y - y)\sigma'(z_k)a_j$$

$$\frac{\partial E}{\partial w_{jk}} = \delta_k a_j$$

- lets recall once what we have learned till now
- for a hidden layer and output layer
- where j represents a hidden layer neuron and k represents a last layer neuron

MACHINE INTELLIGENCE

Back Propagation

$$\frac{\partial E}{\partial w_{ij}} = \left[\sum_k \frac{\partial E}{\partial a_j}\right] \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

$$= \left[\sum_k \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial z_k} \frac{\partial z_k}{\partial a_j}\right] \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

$$= \left[\sum_k (y_k - a_k)\sigma'(z_k) w_{jk}\right] \sigma'(z_j) a_i$$

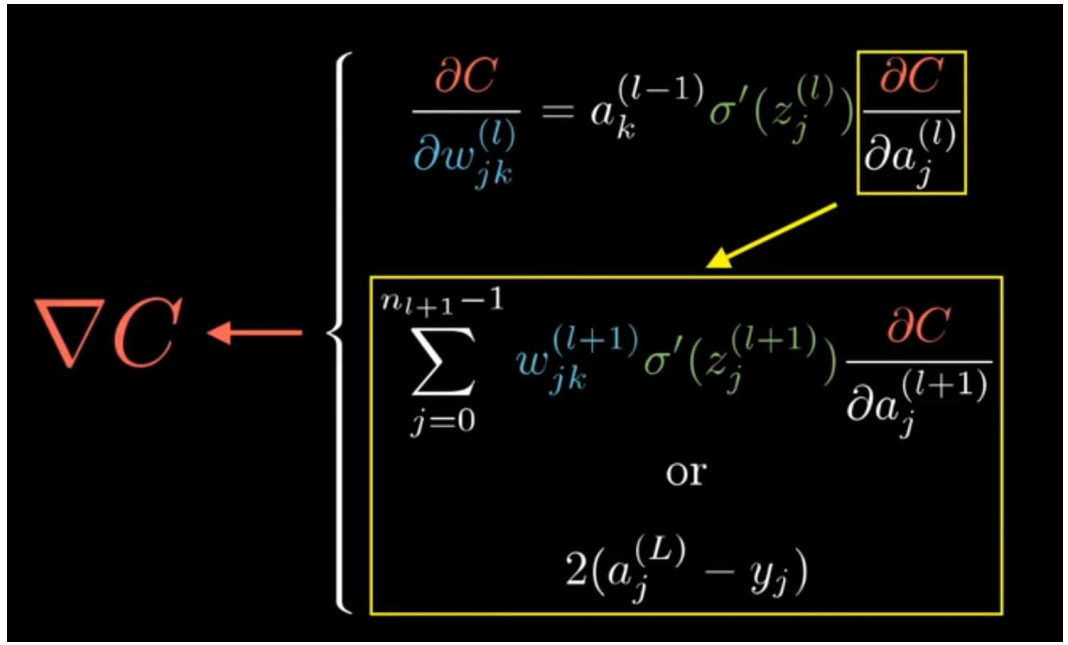
$$= \left[\sum_k \delta_k w_{jk}\right] \sigma'(z_j) a_i$$

$$= \delta_j a_i$$

- for a input layer and hidden layer layer
- where j represents a hidden layer neuron and i represents a input layer neuron

MACHINE INTELLIGENCE

Back Propagation



- If all of this makes sense ,we have now looked deep into the heart of back propagation,
- these chain rule expression gives us the derivative that determines each component in the gradient,that helps minimize the cost of the network by repatedly stepping downhill

THANK YOU

K.S.Srinivas
srinivasks@pes.edu
+91 80 2672 1983 Extn 701

MACHINE INTELLIGENCE

Activation functions

K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

Activation functions

K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

Activation Functions

- we had already seen what is an activation function and what role it plays in a artificial neural network
- we will now look at some of the different activation functions which we will need for different application

MACHINE INTELLIGENCE

Sigmoid Function

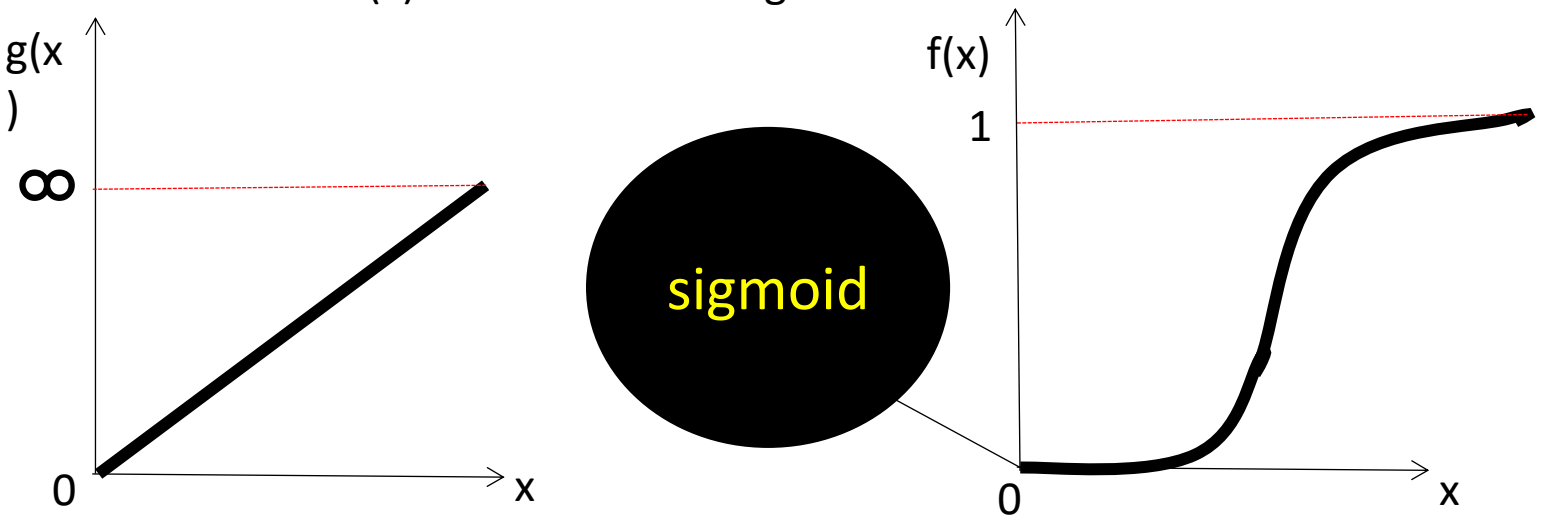
- The sigmoid functions one of the most commonly used activation function.
- The sigmoid can be defined as follows

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

- it scales the value between 0 and 1

let g(x) be linear function i.e g(x) =x

the softmax f(x) does the following transformation



MACHINE INTELLIGENCE

Sigmoid Function

- The function is monotonic ,which implies it is either entirely non-decreasing or non-increasing.
- Observe that this scales the value between 0 and 1. Can you think of something else that lies between 0 and 1 ??
- **Probability.** Hence , this can be used for predicting probability of the output

MACHINE INTELLIGENCE

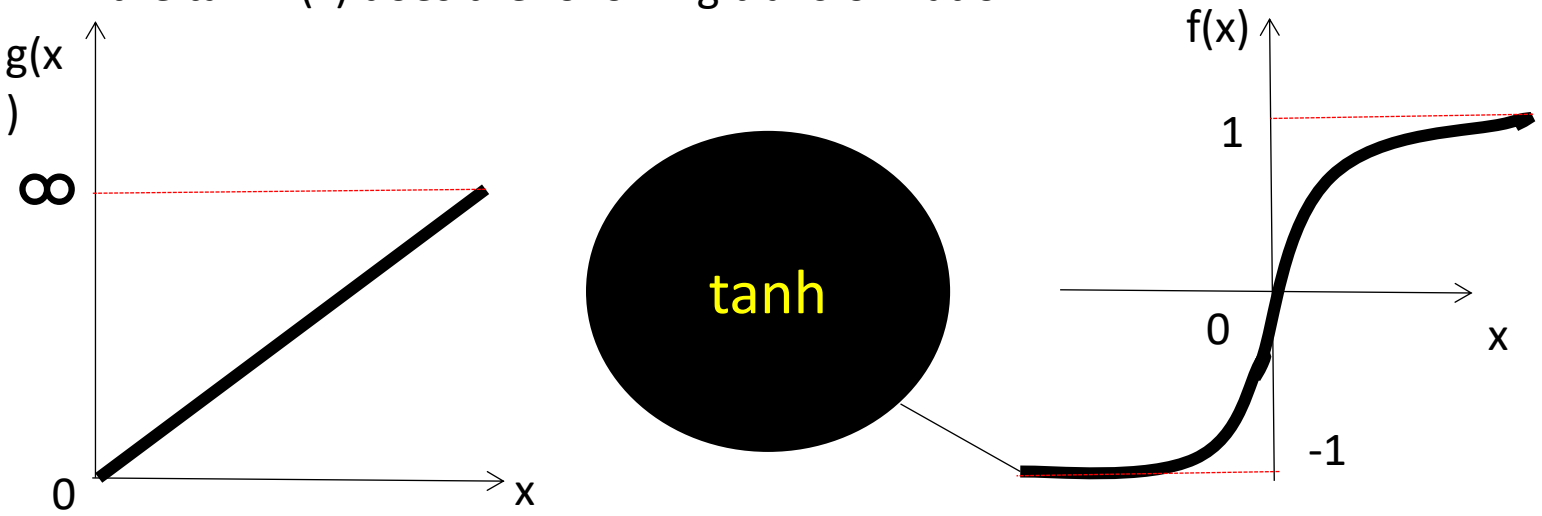
Tanh Function

- The hyperbolic tangent function outputs the value between -1 to +1 and is expressed as follows:

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

let g(x) be linear function i.e g(x) =x

the tanh f(x) does the following transformation



MACHINE INTELLIGENCE

Tanh Function

- The function is monotonic.
- The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph

MACHINE INTELLIGENCE

ReLU Function

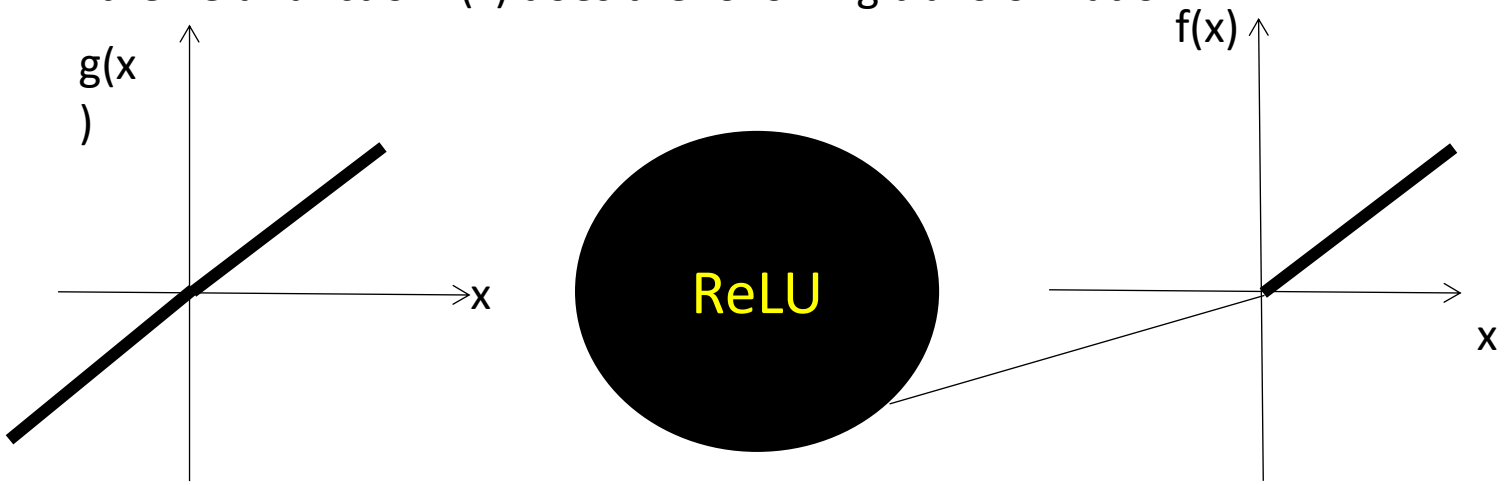
- The **Rectified Linear** Unit function is another one of the most commonly used activation functions.
- It outputs the value from 0 to infinity
- It is basically a peicewise function and can be expressed as follows:

$$f(x) = \{ 0 \quad \text{for } x < 0 \qquad x \quad \text{for } x \geq 0$$

- When we feed any negative input to RELU function it converts it to zero.
- The snag for being zero for all negative values is a problem called **dying RELU,** and a neuron is said to be dead if it always outputs zero.

let g(x) be some function

the Relu function f(x) does the following transformation



MACHINE INTELLIGENCE

Leaky ReLU Function

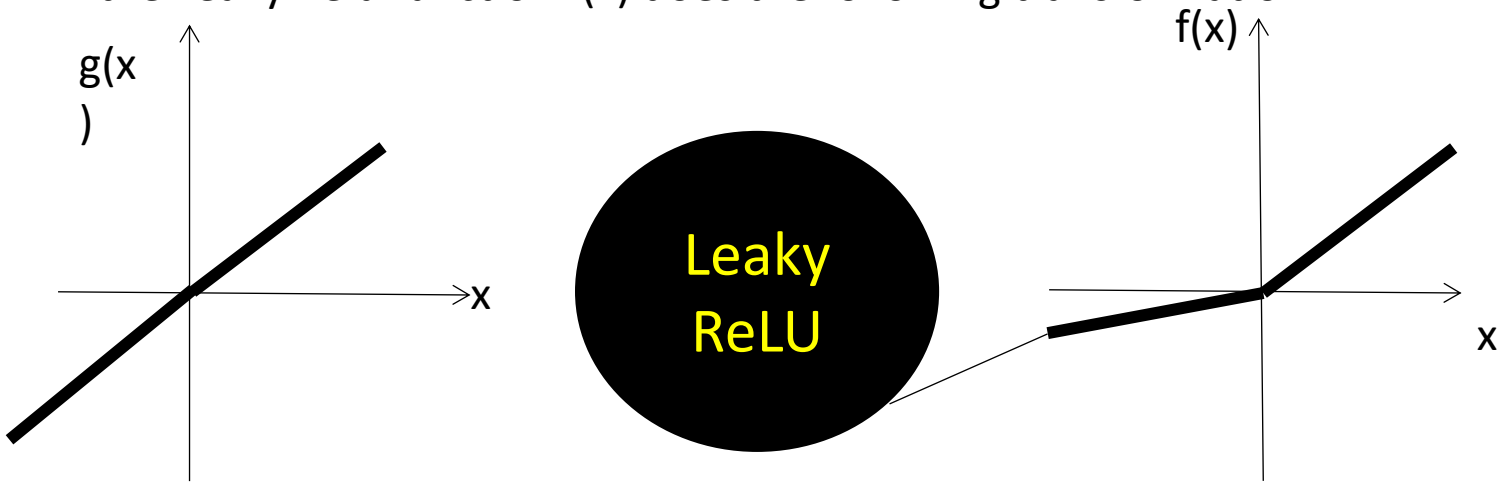
- The **Leaky Rectified Linear** is variant of the ReLU that solves the dying ReLU problem.
- it can be expressed as follows:

$$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

- the value of alpha is typically set to some low value say 0.01

let g(x) be some function

the Leaky Relu function f(x) does the following transformation



- instead of setting some default value to α, we can send them as a parameter to a neural network and make the network learn the optimal value for it.

MACHINE INTELLIGENCE

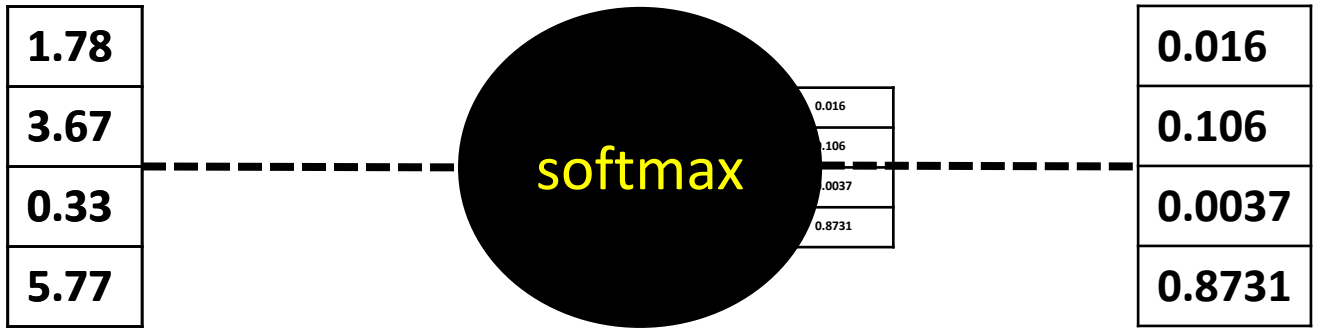
Softmax Function

- The softmax function is basically generalization of the sigmoid function
- usually used in the final layer of the ANN while performing multi-class classification tasks
- It gives the probabilities of each class for being the output and thus the sum of the softmax values will always be equal to 1.
- It can be represented as

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

let x be the output of summation computation at output layer

x=	1.78	3.67	0.33	5.77



THANK YOU

K.S.Srinivas
srinivasks@pes.edu
+91 80 2672 1983 Extn 701

MACHINE INTELLIGENCE

VANISHING AND EXPLOIDING GRADIENTS

K.S.Srinivas

Department of Computer Science and Engineering

MACHINE INTELLIGENCE

VANISHING AND EXPLOIDING GRADIENTS

K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

Exploding and Vanishing Gradient

- one of the problem with training our deep neural network is that is vanishing and exploiding gradients
- which means when we calculate our derivative ,some times it may become very big or some times very small,which makes training difficult

MACHINE INTELLIGENCE

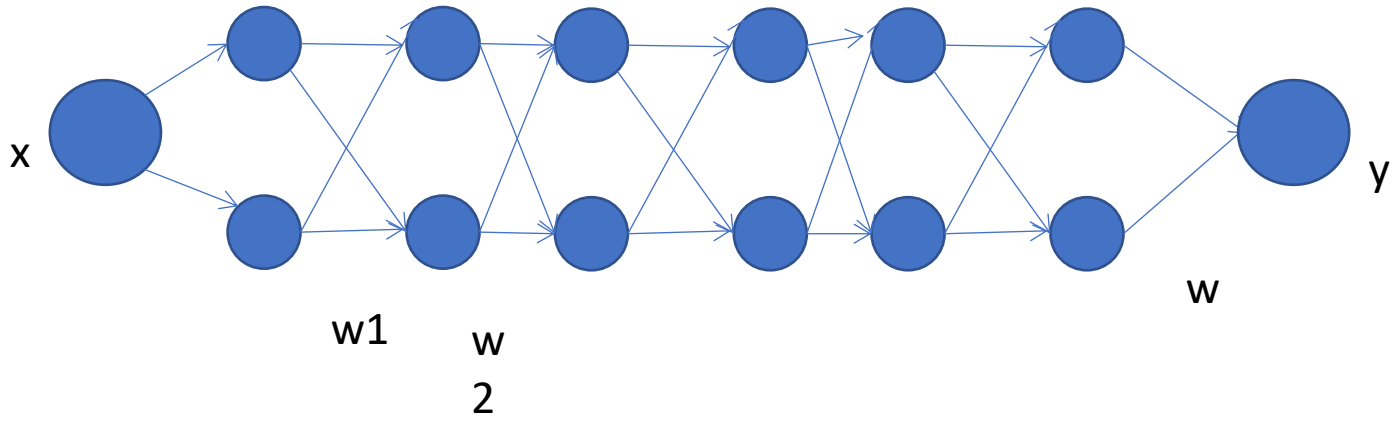
Exploding and Vanishing Gradient

$y=W^{L}.W^{L-1}.....W^{1}.[x]$

$Z_1=W^{1}\cdot x$

$a_1=a(Z_1)=Z_1 \quad a_2=a(Z_2)=a(W^2a_1) \quad AL=a(Z_L)=a(Wall_{-1})$

- consider deep neural network
- this network will have the following parameter
- For our simplicity lets say all bias are 0,and our activation function is linear that is a(x)=x
- then we can show that
- similarly



MACHINE INTELLIGENCE

Exploding and Vanishing Gradient

$y=W^{L}.W^{L-1}.....W^{1}.[x]$

$Z_1=W^{1}.x$

$a_1=a(Z_1)=Z_1 \quad a_2=a(Z_2)=a(W^2a_1) \quad a_L=a(Z_L)=a(W^La_{L-1})$

$$W^{L}=\begin{bmatrix}1.5 & 0\\ 0 & 1.5\end{bmatrix}$$

$$y=\begin{bmatrix}1.5 & 0\\ 0 & 1.5\end{bmatrix}\begin{bmatrix}1.5 & 0\\ 0 & 1.5\end{bmatrix}\cdots\begin{bmatrix}1.5 & 0\\ 0 & 1.5\end{bmatrix}x$$

$$y=1.5^{L-1}x$$

- suppose our each matrix is the following
- so our y would be
- for a very deep neural network the value of L is big and our y would just explode
- conversely if it was 0.5 then it would decrease exponentially
- even though we used activation as increasing or decreasing exponentially as function of L,a similar argument can be used to show that derivatives we use in gradient descent will also decrease or increase exponentially as function of L

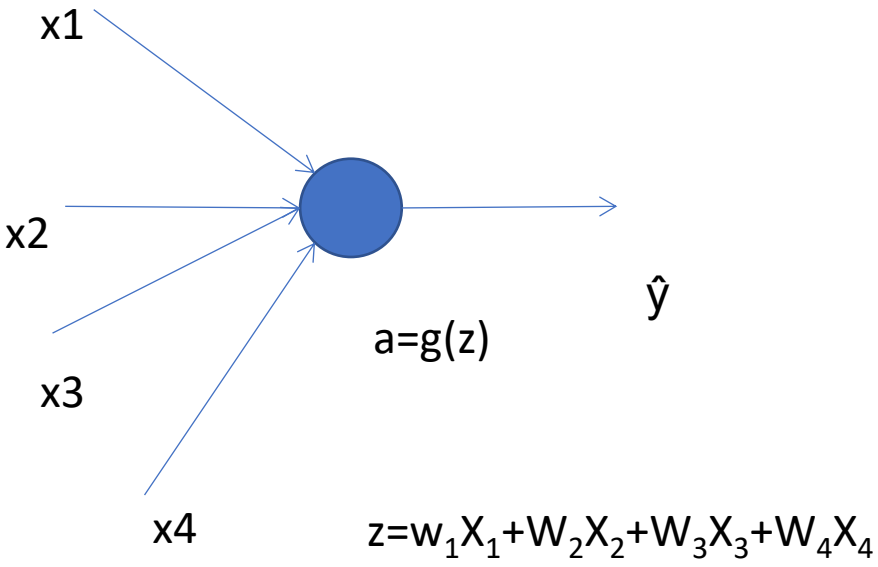
MACHINE INTELLIGENCE

Exploding and Vanishing Gradient

- A partial solution to this is better or more careful choice of random initialization for our neural network

MACHINE INTELLIGENCE

Exploding and Vanishing Gradient

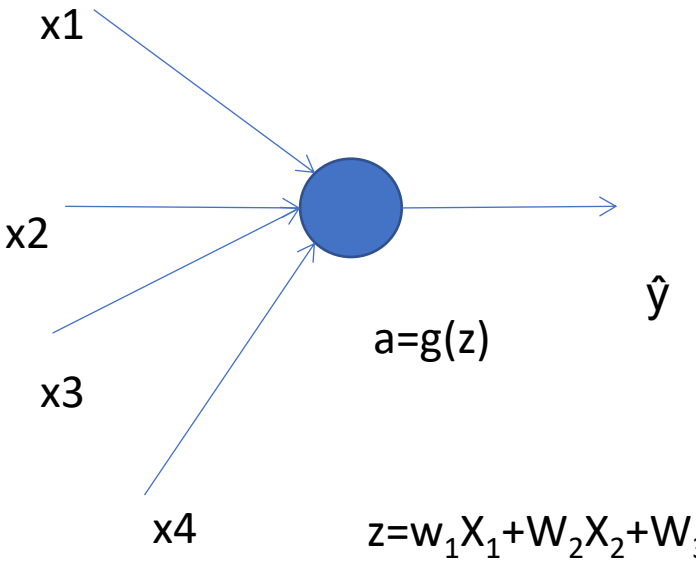


$z=w_1X_1+W_2X_2+W_3X_3+W_4X_4$

- lets start with initializing weight for a single neuron and then we go to generalize this to a deep network
- consider this network with a single neuron
- lets consider b=0

MACHINE INTELLIGENCE

Exploding and Vanishing Gradient



- in order to z not blow up and not become too small,notice that larger n the smaller you want w_i to be small
- one solution is
- so the solution would be
- it turns out that if we are using ReLU activation function then rather then (1/n),setting (2/n) gives better result

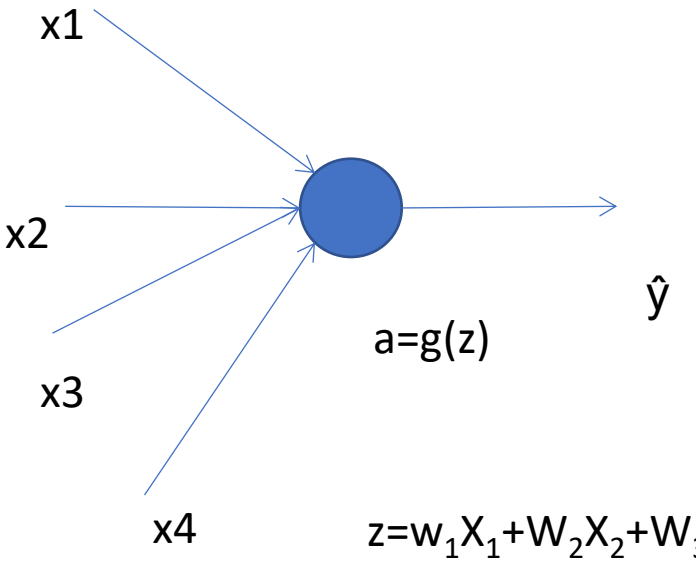
$z=w_1X_1+W_2X_2+W_3X_3+W_4X_4+.....W_nX_n$

$var(w_i)=1/n$

for a given layer W^L=np.random.(shape)*np.sqrt($1/n^{[L-1]}$)

MACHINE INTELLIGENCE

Exploding and Vanishing Gradient



- for tanh this variance works well

$z=w_1X_1+W_2X_2+W_3X_3+W_4X_4+.....W_nX_n$

$var(w_i)=1/n$

for a given layer W^L=np.random.(shape)*np.sqrt($1/n^{[L-1]}$)

for tanh activation function sqrt($1/n^{L-1}$)

THANK YOU

K.S.Srinivas
srinivasks@pes.edu
+91 80 2672 1983 Extn 701

MACHINE INTELLIGENCE

Support Vector Machine

K.S.Srinivas

Department of Computer Science and Engineering

MACHINE INTELLIGENCE

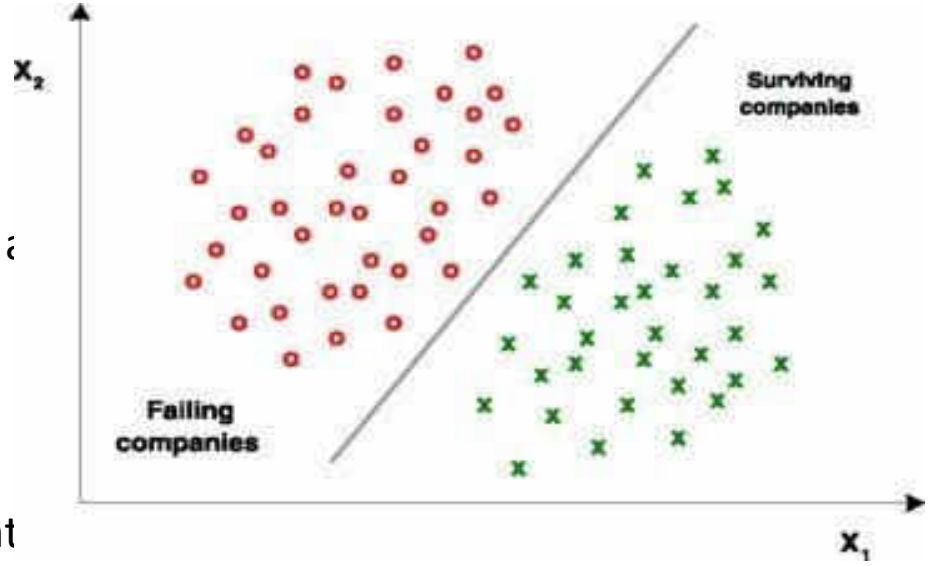
Support Vector Machine

K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

Support Vector Machine

- This is one of the most sophisticated yet simple to implement to algorithms.
- Support Vector Machines is largely thought of as a linear classifier.
- Certain myths about SVMs
    1. Linear Classifier -> Nope it can handle non-linearity a well
    2. Single linear Classifier -> We would do multi – classification
    3. Classifier-> It can do regression as well
    4. SVM are computationally expensive -> with the right and γ **they work like charm**
- **Not a myth – Math heavy and fundamentally provable**

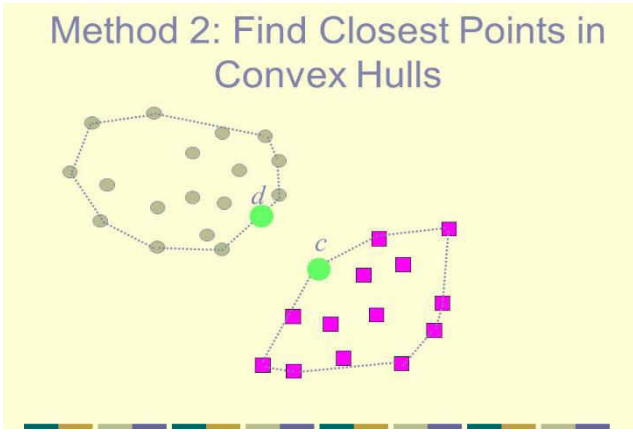


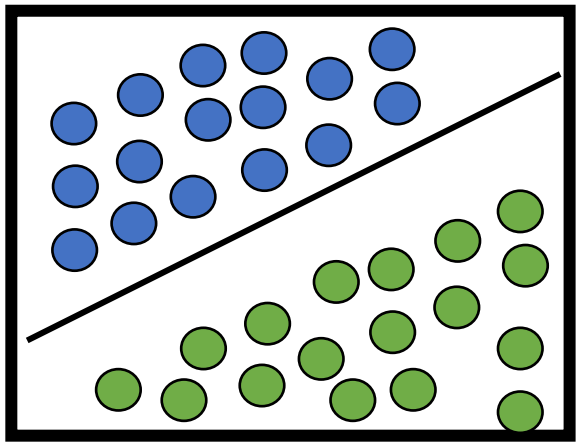
- A data set is said to be linearly separable if we can have a hyper plane that cans separate them.

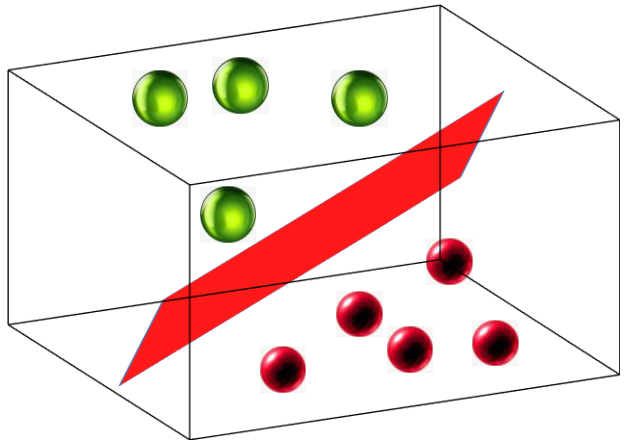
MACHINE INTELLIGENCE

Linearly Separable Data

- Lets look at this Euclidean plane with 2 sets of data points.
- The line that separates them is the decision boundary.
- In a 3 dimensions that would be a plane
- These points that are separable usually form a convex hull



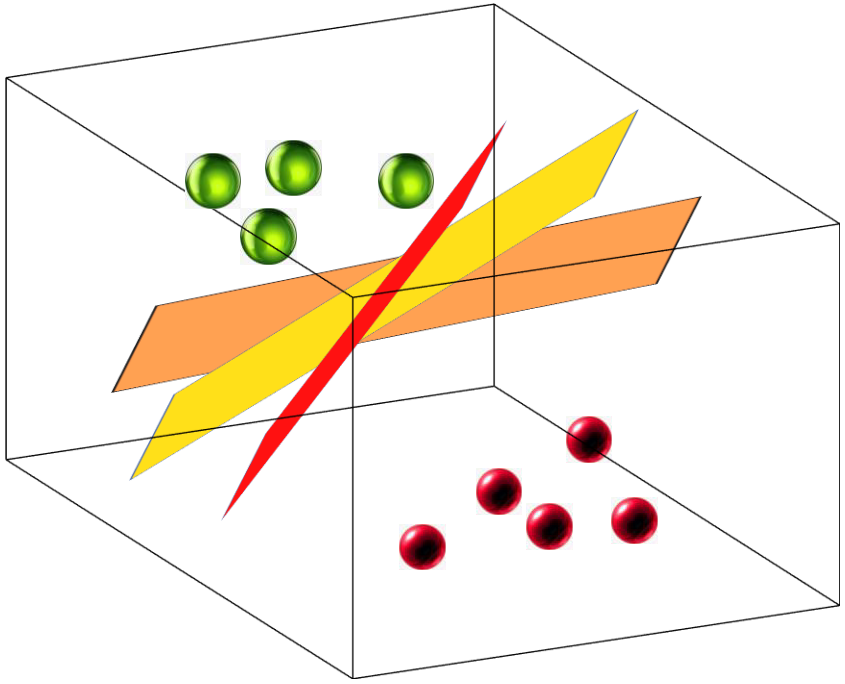


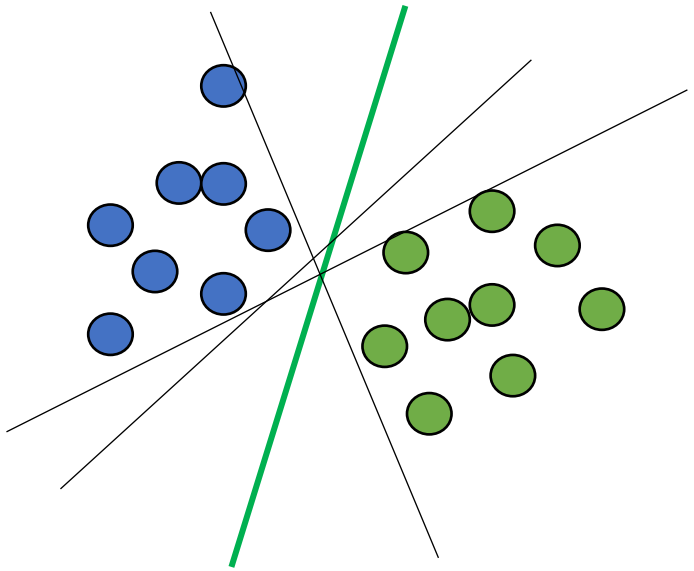


MACHINE INTELLIGENCE

Example of Linearly Separable Data

- But we know that are many planes that can separate these instances
- The real question is which of these planes is the best
- You can see that the green line seems to be the best – Why?
- Because the green line seems to be the farthest away from the closest points

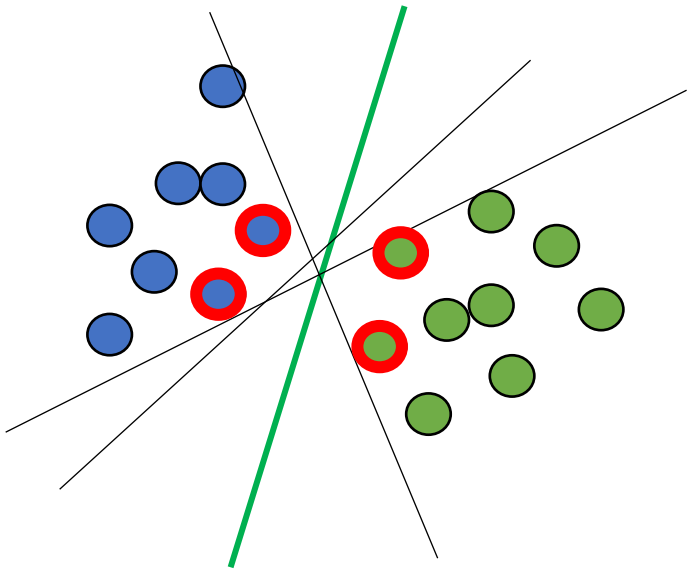




MACHINE INTELLIGENCE

Example of Linearly Separable Data

- But we know that are many planes that can separate these instances
- The real question is which of these planes is the best
- You can see that the green line seems to be the best – Why?
- Because the green line seems to be the farthest away from the closest points

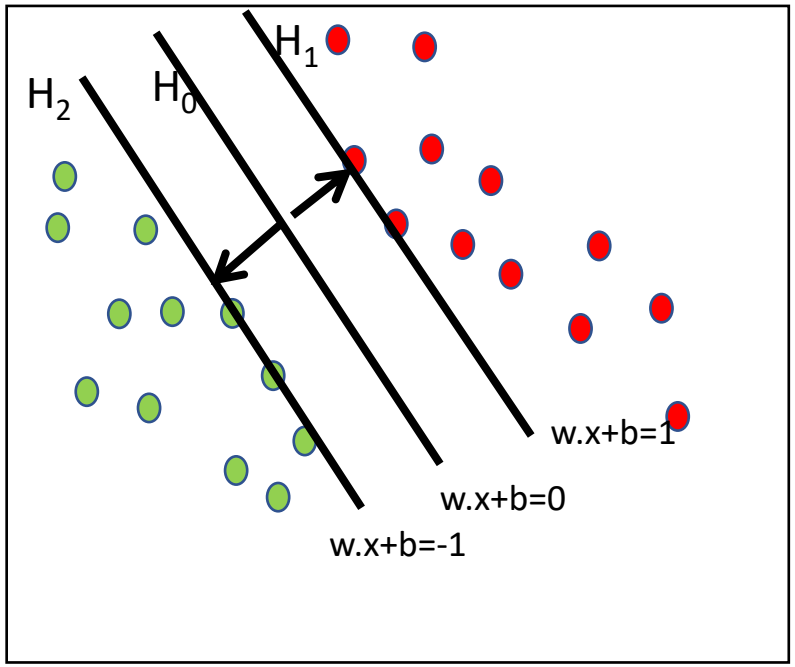


- These points marked in red play a role in defining the hyper plane
- These are the Support Vectors

MACHINE INTELLIGENCE

Hyper planes

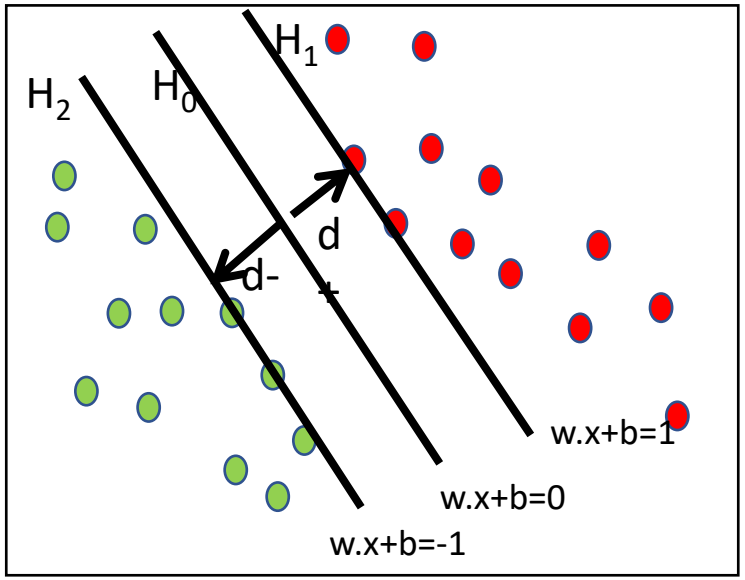
- So the cool thing is this plane separate the instance set maximally.
- Lets draw this plane H0 that separates them maximally – Say the median (as in road median)
- Lets also draw equidistant margin H1 and H2 such that
  The margins are parallel to the median (aka the gutter)
  The closest points on either side lie on the margin
- Our goal now simplifies to making the distance say 2d (sum (d1+d2)) as large as possible
- SVM is a maximum margin classifier – one of the most elegant ideas that have come about in computer science



MACHINE INTELLIGENCE

Hyper planes

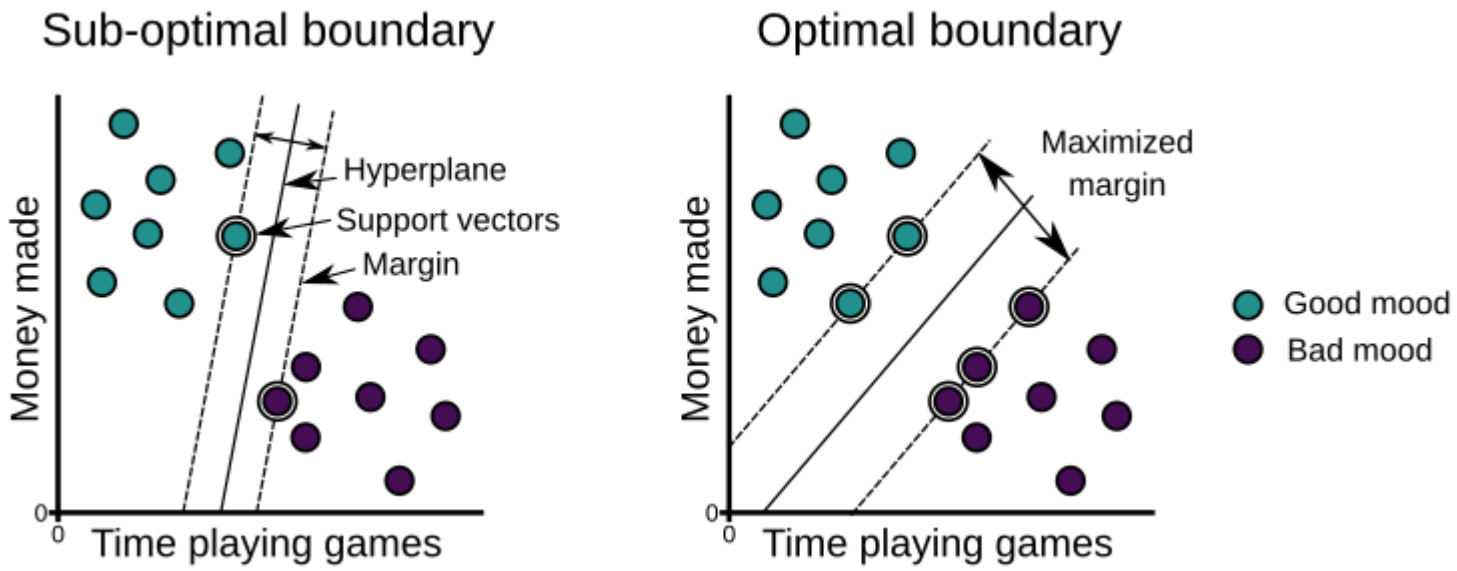
- A ‘N’ number of hyper planes can be traced between data which can be linearly separable.
- There are 3 significant hyper planes defined namely , H0 which acts as a median to H1 and H2 hyper planes which are the closest to the data sets on both planes such that the margin between them is maximum.
- The distance between the two extreme hyper planes can be given by ‘2d’



MACHINE INTELLIGENCE

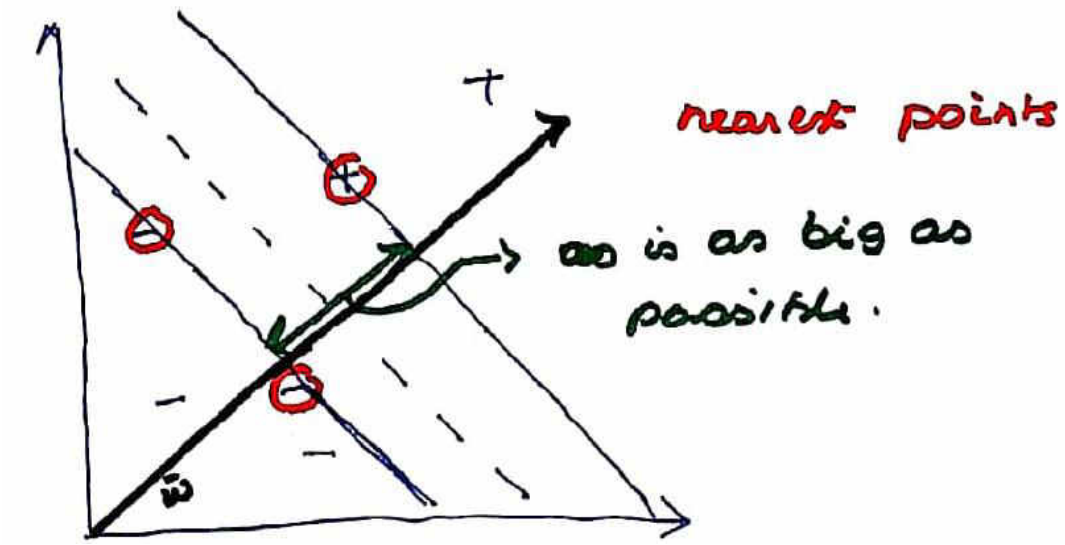
Optimal vs Sub optimal

- We saw some equations earlier around these medians and margins but let the geometry sink in a little



Geometry of the Median

- Lets draw this line w perpendicular to the street (perforce median and gutter)
- Lets also insist that the closest points lie on the margins(gutters)
- Our Goal as stated before is to make the street as big as possible
- We don't know anything about the length of the vector w



MACHINE INTELLIGENCE

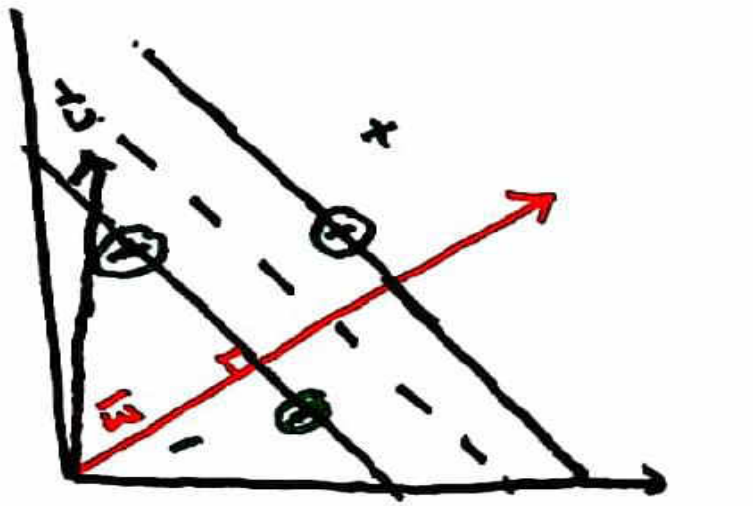
Finding the distance of an unknown point

- Lets get an unknown vector u
- We know quite intuitively that the point is either a + sample or a – sample by looking at the projection of u on w
- The farther u is on the right side of w we know it is a + sample and a –ve sample other wise
- To compute how far it is on w all we need to do
- Is take a dot product of w.u which is a scalar quantity

$\vec{w} \cdot \vec{u}$

- Lets actually insist that for it to be a + sample it must

$\vec{w} \cdot \vec{u} \geq c$ for +ve sample



Creating our decision rule

- So our decision rule is

$\vec{w} \cdot \vec{v} \geq c$ for +ve sample

- Without any loss of generality we can say

$\vec{w} \cdot \vec{v} + b \geq 0$ by putting $c = -b$

- This is our decision rule 1/5 which we will use for our final equation

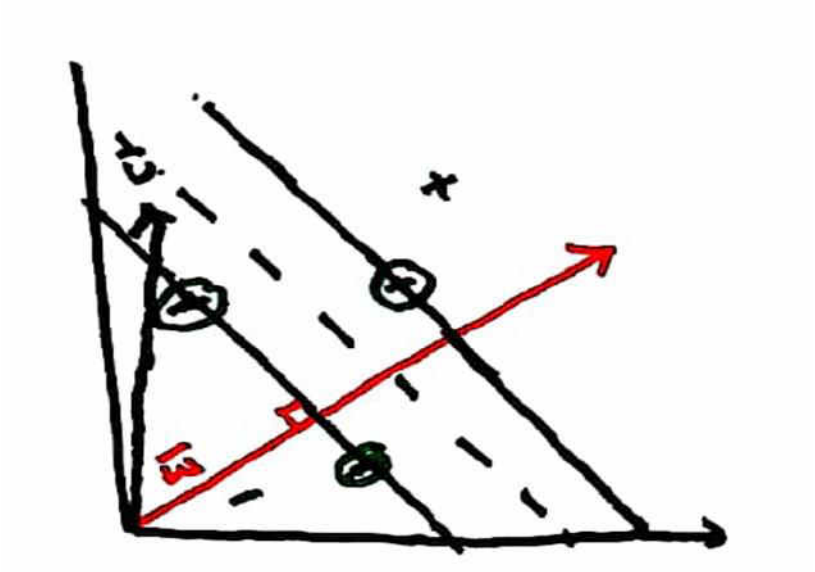
Enhancing our decision Rule

- Our Decision rule is $\vec{w} \cdot \vec{u} + b \geq 0$

- We are not just going to leave the decision rule but make it HARD
- We insist that for all + samples that the decision rule must be >= 1
- We insist that for all – samples that our decision rule must be <= -1

$\vec{w} \cdot \vec{u} + b \geq 1$

$\vec{w} \cdot \vec{u} + b \leq -1$



Defining the margins

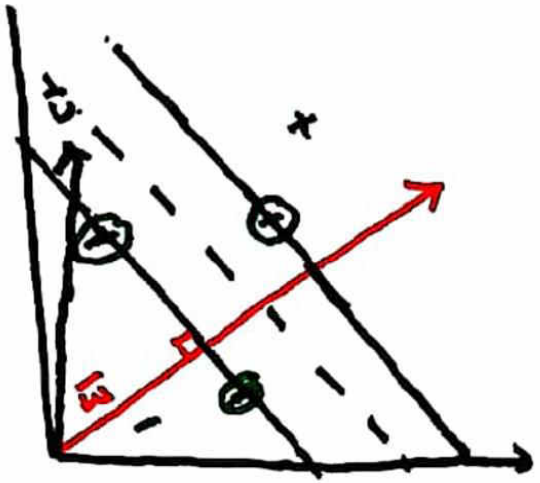
- Lets also harden the margins by saying for the margins the equations are exactly +1 and -1
- Therefore the equations of the points on margins are

$$\vec{w} \cdot \vec{u}^{+} + b = 1$$

for + ve distances on the margin

$$\vec{w} \cdot \vec{u}^{-} + b = -1$$

for -ve distances on the margin.

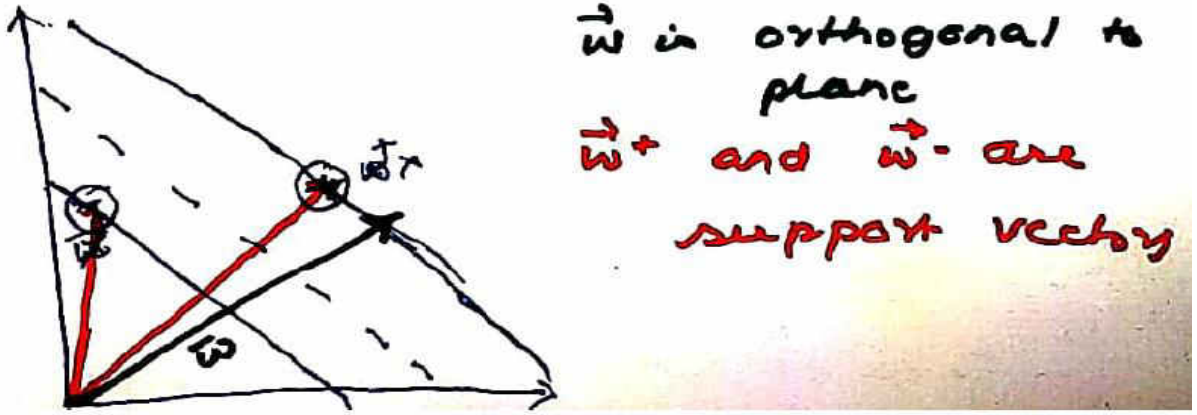


Defining the Support Vectors

- The samples on the margins are the support vector that satisfy the equations

$\vec{w} \cdot \vec{u}^{+} + b = 1$ for + ve instances on the margin

$\vec{w} \cdot \vec{u}^{-} + b = -1$ for -ve instances on the margin.



$\vec{w}$ is orthogonal to plane

$\vec{w}^{+}$ and $\vec{w}^{-}$ are support vectors

MACHINE INTELLIGENCE

A mathematical convenience

- Lets introduce another quantity y_i for every instance .
- The property **of y_i** that for every sample is

$y_i = 1$ if + sample

$y_i = -1$ if - sample .

- This could help us in rewriting our equation as for all +ve and –ve samples as

$\vec{w} \cdot \vec{x}^+ + b \geq 1$

$\vec{w} \cdot \vec{x}^- + b \leq -1$

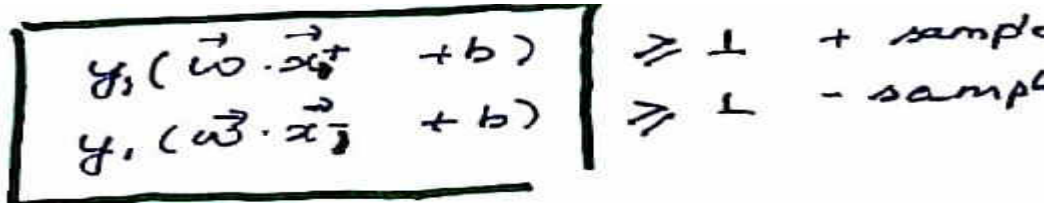
→ $y_i(\vec{w} \cdot \vec{x}_i^+ + b) \geq 1$ + sample

$y_i(\vec{w} \cdot \vec{x}_j^- + b) \geq 1$ - sample

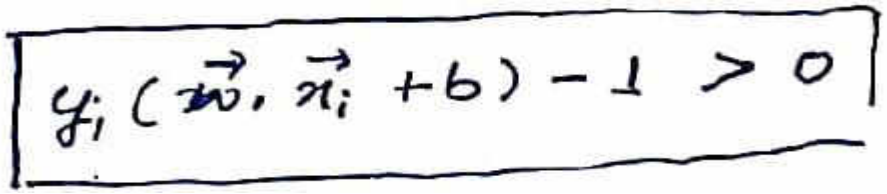
Not a mistake

Some rearrangements

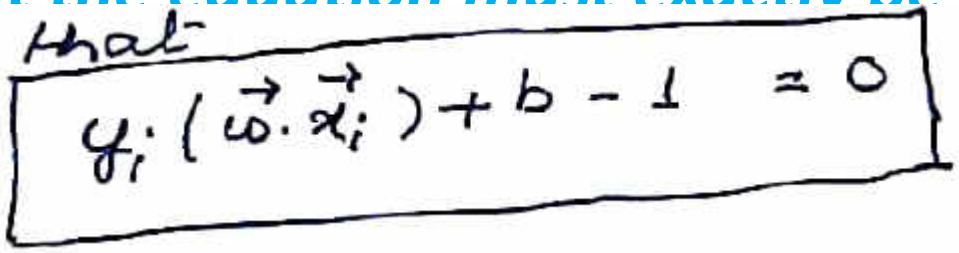
- Lets move 1 to the left



- That would make the equation to for all + and – samples

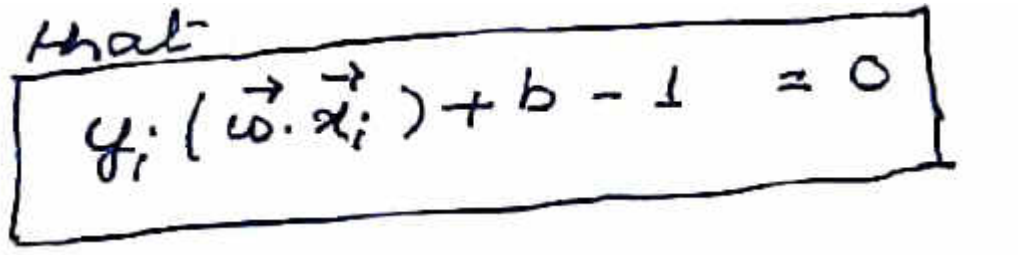


- ***But for all instance that lie in the gutter we will insist that the equation must exactly be equal to 0***



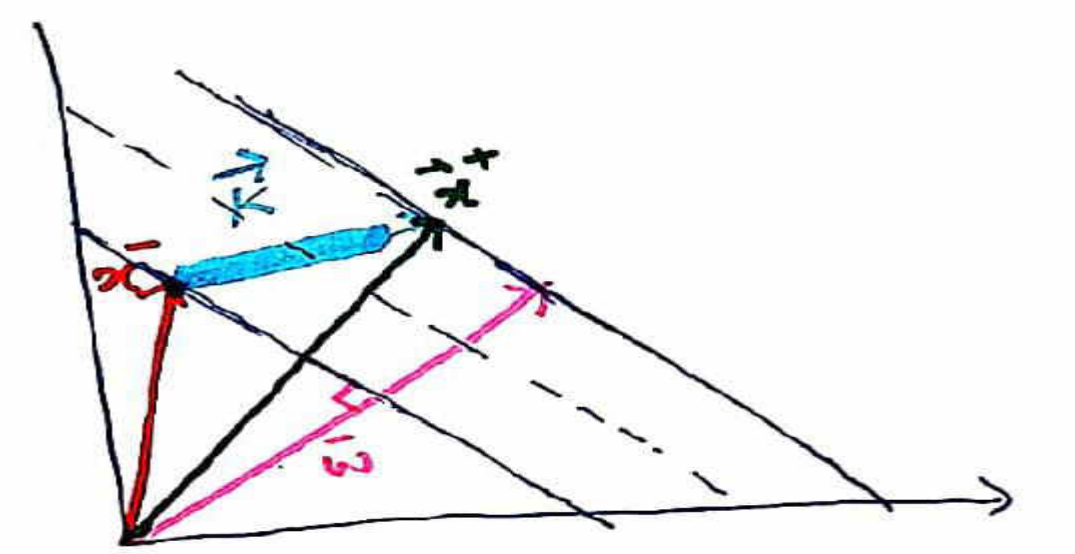
Lost track??

- Our original problem was to find a street that maximally separated the samples
- i.e as we break it down
  - Find margins
  - Maximize the street
  - AND NOW LET THE MARGINS BE CONSTRAINED BY



Lets find the quantity we want to maximize

- If we have a x- on the margin and another x+ on the other margin
- The difference between these 2 vectors is (x+ - x-) as indicated by the blue indicates the width of the street
- These would need to be projected onto the vector w



MACHINE INTELLIGENCE

Lets compute the margin

- We known that for + support vector the equation (y_i=+1)
- We known that for - support vector the equation (y_i=-1)
- What we want to compute is

$$\frac{\vec{w}}{\|w\|} \cdot (\vec{x}^+ - \vec{x}^-) = \text{width}$$

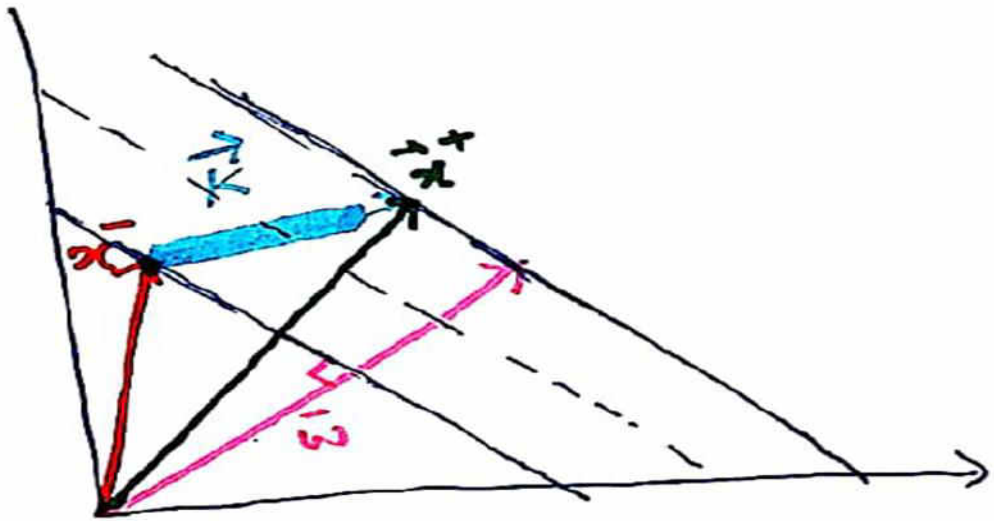
that

$$y_i(\vec{w} \cdot \vec{x}_i) + b - 1 = 0$$

that

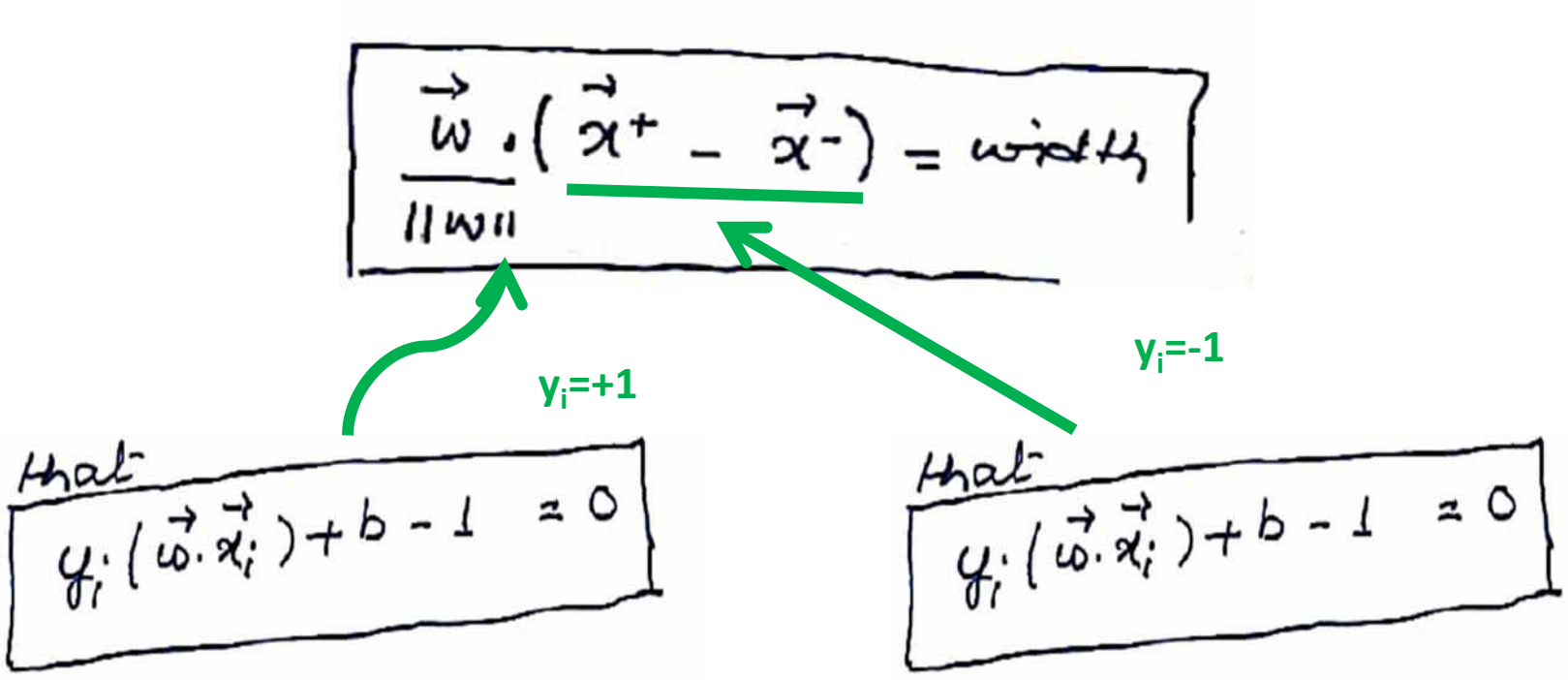
$$y_i(\vec{w} \cdot \vec{x}_i) + b - 1 = 0$$

We have just made the vector W an unit vector



MACHINE INTELLIGENCE

Lets simplify the equation



That gives us the margin to maximized as

$$\frac{\vec{w}}{\|w\|} \left((1-b) - (1+b) \right) = \frac{2}{\|w\|}$$

MACHINE INTELLIGENCE

Lets add some more mathematical convenience

- We wish to maximize $= \frac{2}{\|w\|}$
- Which is the same as minimizing $\min \frac{1}{2} \|w\|^2$
- Subject to $y_i(\vec{w}\vec{x_i} + b) - 1 \geq 0$
- This is the primal form

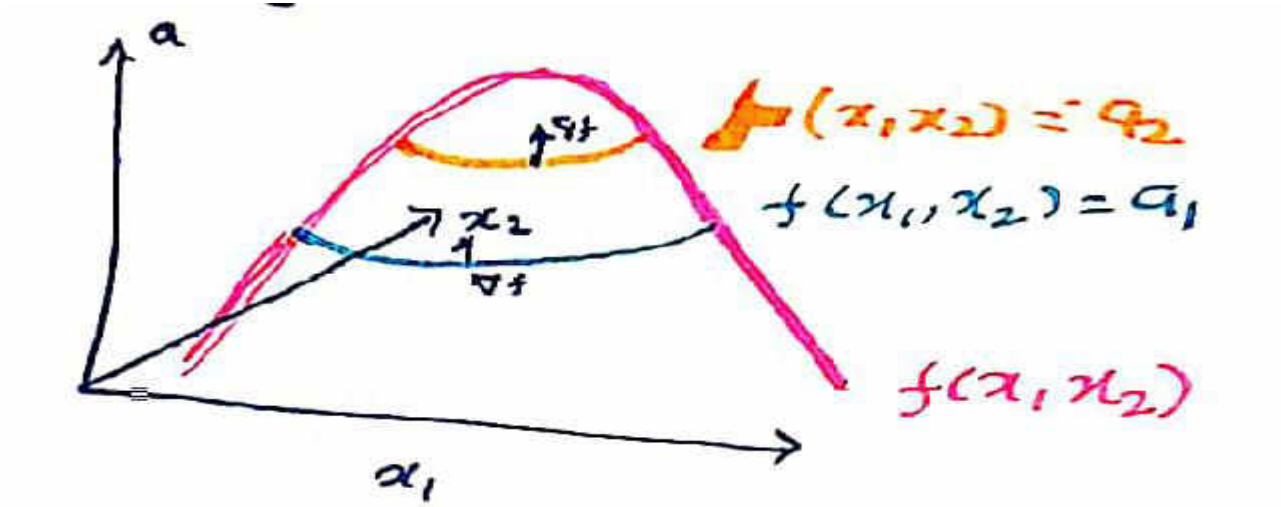
This is super critical

$\min \frac{1}{2} \|w\|^2$ s.t $y_i(\vec{w} \cdot \vec{x_i} + b) - 1 \geq 0$

MACHINE INTELLIGENCE

A small detour into constrained optimization

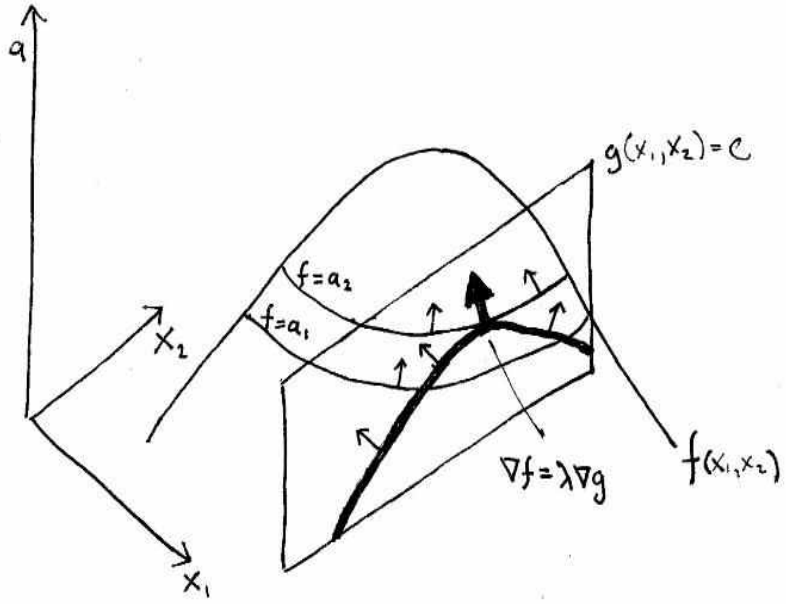
- Lets say there is a function f(x1,x2) that we wish to maximize subject to a function g(x1,x2) = c
- Please note the equality constraint which will need to be relaxed later.
- We need to take the ∇f which gives us the steepest direction
- Let us visualize the plot f = a1 all along the blue line and f=a2 along the orange line
- The function f being represented by the pink line



MACHINE INTELLIGENCE

Constrain added in

- We start climbing the hill moving from left to right on the constraint curve
- At each point we check ∇g and check ∇f
- When $\nabla g > \nabla f$ we keep moving
- At a point where $\nabla g = \nabla f$ we have reached the maximum
- Any further move will result in declining f



MACHINE INTELLIGENCE

Constrain added in

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) = 0, \quad i = 1, \ldots, p \\ & h_i(x) \leq 0, \quad i = 1, \ldots, m \end{aligned}$$

- We can rewrite

$$\nabla f = \lambda(x) * \nabla g$$

$$\nabla f - \lambda \nabla g = 0$$

$$g = c$$

$$L = f - \lambda(g - c)$$

Algorithm

- Construct the Lagrangian
- One Lagrangian multiplier per constraint
- Which means one α per instance
- Compute ∇L
- Set $\nabla L = 0$ and solve for this

Final Lagrangian

$$g = c$$

$$L = f - \lambda(g - c)$$

Which for our problem is

$$L = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{M} \alpha_i \left[y_i (w x_i + b) - 1 \right]$$

and we want to minimize $\frac{1}{2}\|w\|^2$

$$\min L_p = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{M} \alpha_i \left[y_i (w x_i + b] - 1 \right]$$

So we solve for Dual

- It is said that it is easier to solve the dual than the primal
- The Lagrangian dual states that instead of **solving for minimizing w,b subject to the constraints involving α i.e $\alpha>=0$**
- Solve instead by maximizing α subject to the previously defined constraints
- What that means is
  - Define L
  - Take partials with respect to w and b
  - Set them to 0
  - Replace them in the original equation

So the solution is...

$$f(x) \equiv \frac{1}{2}\|w\|^2 \qquad g(x) \equiv y_i(wx_i + b) - 1$$

$$\min L_p = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{m} \alpha_i \left[y_i (w x_i + b) - 1 \right]$$

$$\nabla L_p = 0$$

$$\frac{\delta L_p}{\delta w} = w - \sum_{i=1}^{m} \alpha_i x_i y_i = 0$$

$$w = \sum_{i=1}^{m} \alpha_i x_i y_i$$

$$\frac{\delta L_p}{\delta b} = \sum_{i=1}^{m} \alpha_i y_i = 0$$

$$\sum_{i=1}^{m} \alpha_i y_i = 0$$

So the solution is...

Primal problem:

$$\min L_P = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{l} a_i y_i (\mathbf{x}_i \cdot \mathbf{w} + b) + \sum_{i=1}^{l} a_i$$

s.t. $\forall i$ $a_i \geq 0$

$$\mathbf{w} = \sum_{i=1}^{l} a_i y_i \mathbf{x}_i, \qquad \sum_{i=1}^{l} a_i y_i = 0$$

Dual problem:

Dual problem:

$$\max L_D(a_i) = \sum_{i=1}^{l} a_i - \frac{1}{2}\sum_{i=1}^{l} a_i a_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

$$\text{s.t.} \sum_{i=1}^{l} a_i y_i = 0 \;\&\; a_i \geq 0$$

So the solution is...

The Lagrangian Dual Problem: instead of minimizing over w, b, subject to constraints involving a's, we can maximize over a (the dual variable) subject to the relations obtained previously for w and b

Our solution must satisfy these two relations:

$$\mathbf{w} = \sum_{i=1}^{I} a_i y_i \mathbf{x}_i, \quad \sum_{i=1}^{I} a_i y_i = 0$$

We will now solve for the a's by differentiating the dual problem wrt a, and setting it to zero. Most of the a's will turn out to have the value zero. The non-zero a's will correspond to the support vectors

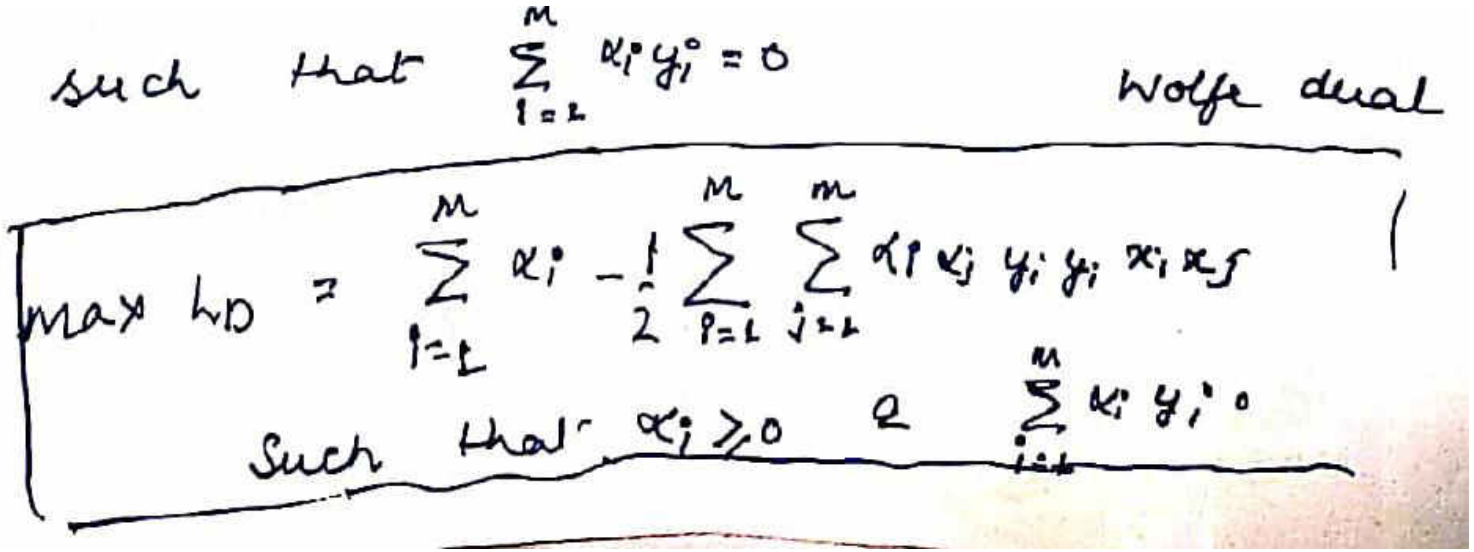
By substituting for w and b back in the original eqn we can get rid of the dependence on w and b.

Note first that we already now have our answer for what the weights w must be: they are a linear combination of the training inputs and the training outputs, xi and yi and the values of a

MACHINE INTELLIGENCE

Wolf Dual replacement

Which when we replace we get Wolfe Dual – - Straightforward



KKT

- Slack condition gives us the formulation that for all points other than the support vectors alpha is 0
- Since KKT is satisfied we have g = 0 changed to g > 0 as the solution

The Karush-Kuhn-Tucker conditions are:

- Stationarity condition:

$$\nabla_{\mathbf{w}}\mathcal{L} = \mathbf{w} - \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i = \mathbf{0}$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{m} \alpha_i y_i = 0$$

- Primal feasibility condition:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0 \qquad \text{for all } i = 1, \ldots, m$$

- Dual feasibility condition:

$$\alpha_i \geq 0 \qquad \text{for all } i = 1, \ldots, m$$

- Complementary slackness condition:

$$\alpha_i[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1] = 0 \qquad \text{for all } i = 1, \ldots, m$$

THANK YOU

K.S.Srinivas
srinivasks@pes.edu
+91 80 2672 1983 Extn 701

MACHINE INTELLIGENCE

Kernel

K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

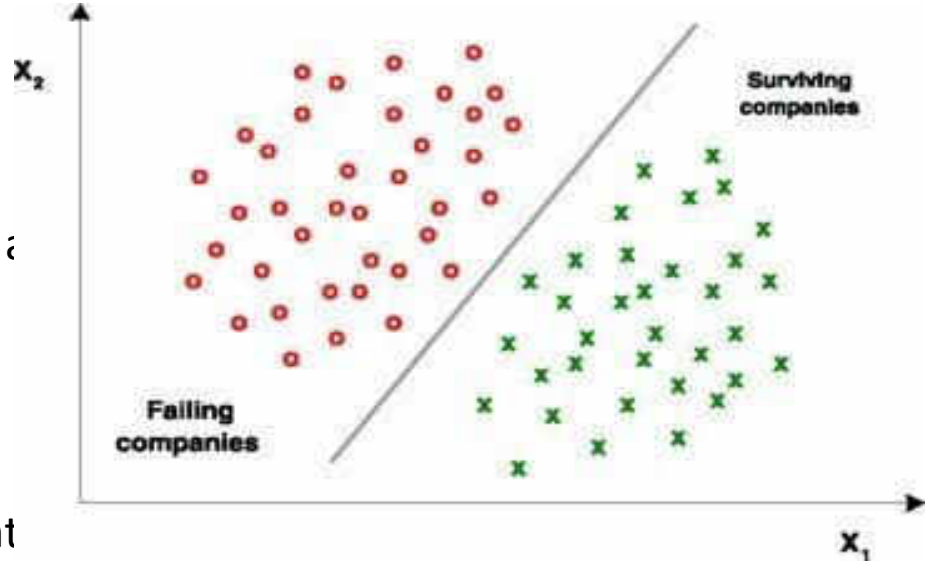
Kernel

K.S.Srinivas
Department of Computer Science and Engineering

MACHINE INTELLIGENCE

Support Vector Machine

- This is one of the most sophisticated yet simple to implement to algorithms.
- Support Vector Machines is largely thought of as a linear classifier.
- Certain myths about SVMs
    1. Linear Classifier -> Nope it can handle non-linearity a well
    2. Single linear Classifier -> We would do multi – classification
    3. Classifier-> It can do regression as well
    4. SVM are computationally expensive -> with the right and γ **they work like charm**
- **Not a myth – Math heavy and fundamentally provable**



- A data set is said to be linearly separable if we can have a hyper plane that cans separate them.

MACHINE INTELLIGENCE

Some basic Formulation

$$\max_{\alpha} \quad L = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i x_j$$

- Subject to

1) $\sum_{i=1}^{N} \alpha_i y_i = 0$
2) $0 \leq \alpha_i \leq C$

The beauty of the algorithm.

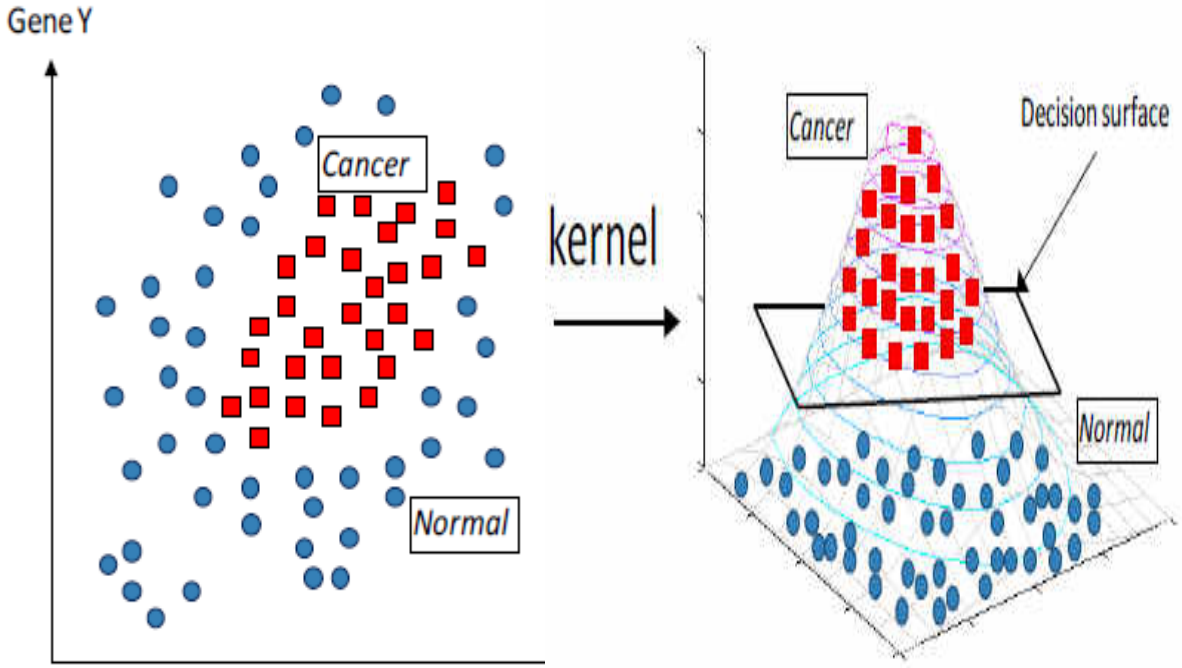
Only Scalar quantities except for $x_i.x_j$

Would be computationally inexpensive if all wee had to was just a dot product of TWO VECTORS

MACHINE INTELLIGENCE

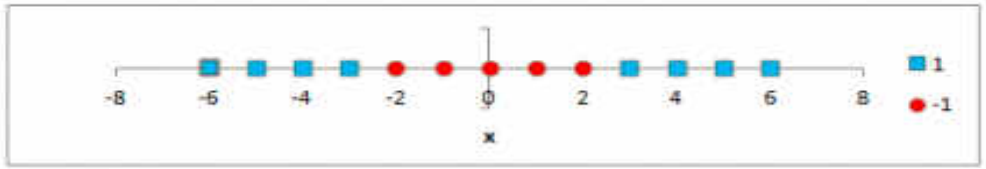
Non Linearity

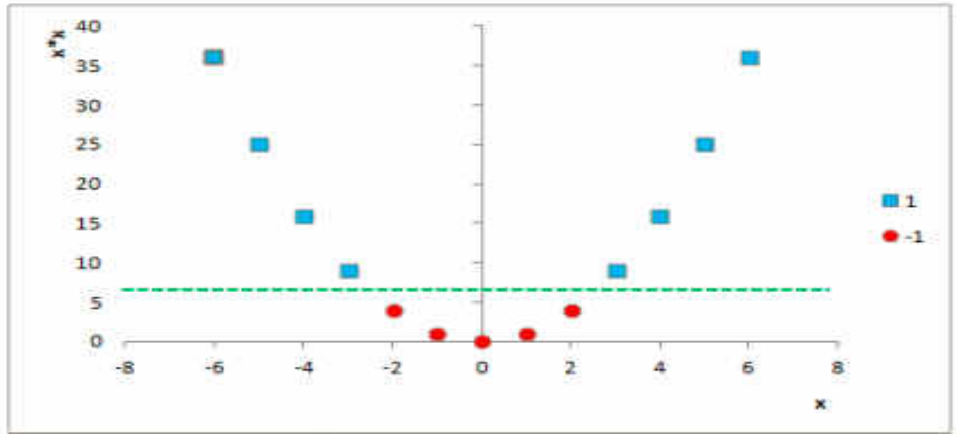
- How do we separate such data ?
- Maybe by adding another dimension.
- One more dimension – Yup.
- But which Dimension
- How many dimensions
- More dimensions means more computation
- Real Life data has 1000 dimensions and fo separation we may need to scale by 1000.



Finding dimensions

- That does not seem to be separable given only 1d.
- What if we converted this to a 2 feature vector z = [x, x*x]
- Clearly Separable
- Given this can find a set of dimensions to separate out this data



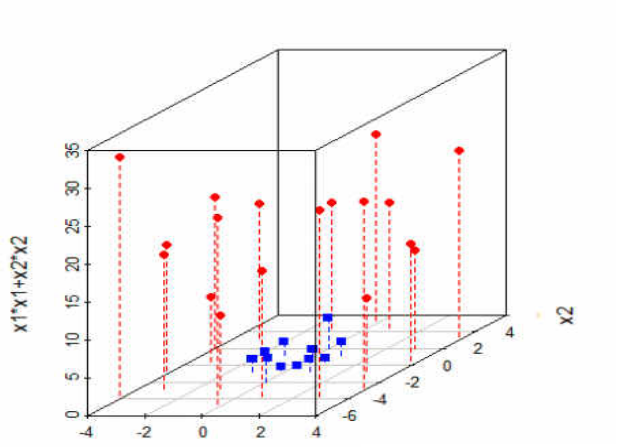


MACHINE INTELLIGENCE

The transformation trick

- Sure, but is that computationally expensive
- Not Really , With a new trick in our tool bag called as the kernel trick.
- Recollect that

$$\max_{\alpha} \; L = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i x_j$$



$$\phi: x \rightarrow z = [x, x^2] \qquad \phi: x \rightarrow z = [x_1, x_2, x_1^2 + x_2^2]$$

- Which has just a dot product
- φ is the symbol we using for feature engineering

Modified Dual Form

- So the new dual is a kernel function applied on the dot product terms

$$\max_{\alpha} \; L = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \phi(x_i) \phi(x_j)$$

- This would mean going to those higher dimensions and then computing the dot product
- Not really – Lets see this with an example

Example

- Dot These

- Now compute $(a.b+1)^2$
- Both have the same value.
- That's the neat trick. We can compute the value by one dot + some constant squared

Polynomial Kernel

- The trick we used in the earlier example is a polynomial kernel

More broadly, the kernel $K(x,y) = (x^T y + c)^d$ corresponds to a feature mapping to an (n+d choose d) feature space, corresponding of all monomials of the d form $x_i^1 x_i^2 ... x_i^l$ that are up to order d. However, despite working in this $O(n^d)$-dimensional space, computing $K(x,y)$ still takes only $O(n)$ time, and hence we never need to explicitly represent feature vectors in this very high dimensional feature space.

Consider polynomial kernel: $K(\vec{x}_i, \vec{x}_j) = (1 + \vec{x}_i \cdot \vec{x}_j)^3$

Assume that we are dealing with 2-dimensional data (i.e., in $\mathbb{R}^2$). Where will this kernel map the data?

2-dimensional space

$x_{(1)}$ $x_{(2)}$

kernel

10-dimensional space

1 $x_{(1)}$ $x_{(2)}$ $x_{(1)}^2$ $x_{(2)}^2$ $x_{(1)}x_{(2)}$ $x_{(1)}^3$ $x_{(2)}^3$ $x_{(1)}x_{(2)}^2$ $x_{(1)}^2x_{(2)}$

MACHINE INTELLIGENCE

Polynomial Kernel Expansion for d=2

$$\left(\sum_{i=1}^{D} x_i x_i'\right)\left(\sum_{j=1}^{D} x_j x_j'\right) = \sum_{i=1}^{D}(x_i x_i')^2 + \sum_{i=1}^{D-1}\sum_{j=i+1}^{D} 2x_i x_i' x_j x_j'$$

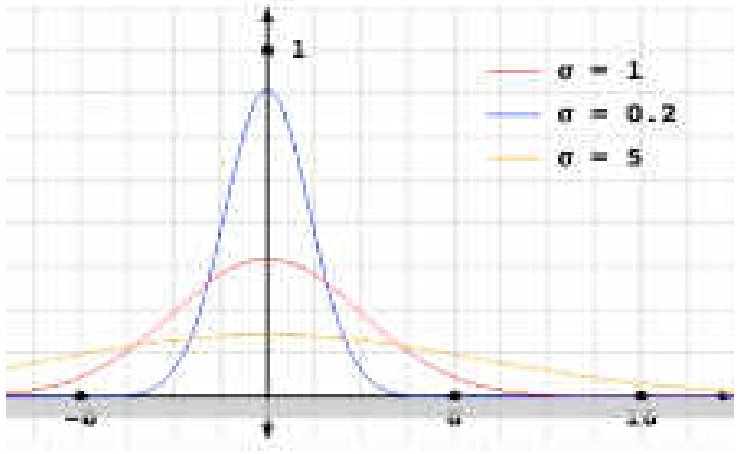
$$\left(\sum_{i=1}^{D} x_i x_i' + c\right)\left(\sum_{j=1}^{D} x_j x_j' + c\right) = \sum_{i=1}^{D}(x_i x_i')^2 + \sum_{i=1}^{D-1}\sum_{j=i+1}^{D} 2x_i x_i' x_j x_j' + 2c\sum_{i=1}^{D} x_i x_i' + c^2$$

Gaussian Kernel

- This is the most popular kernel
- Synonymous with SVM and the default
- This has the same shape as a Gaussian , but is not a PDF since it is not normalized

$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right) \qquad K(\mathbf{x}, \mathbf{x}') = \exp\left(-\gamma\|\mathbf{x} - \mathbf{x}'\|^2\right) \qquad \gamma = \frac{1}{2\sigma^2}$$

- High Variance – Low precision
- Low Variance – High Precision



MACHINE INTELLIGENCE

Behaviour of Gaussian Kernel

- Measures the similarity between x and x'.
- Recollect $K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma\|\mathbf{x} - \mathbf{x}'\|^2)$

- If x and x' are similar we get exp(0)
- If x and x' are different we get values from 0 to < 1.
- The further the points are exp gets smaller.
- But the key point is that it only depends on the radial distance between the 2 and is independent of relative position
- Hence the name radial basis function

MACHINE INTELLIGENCE

Beauty of Gaussian Kernel

- For any kernel to be valid it must be expressed in terms of a dot

$$K[\phi(x_1)\,\phi(x_2)] = e^{-\gamma\|x_1 - x_2\|^2}$$

Let $\gamma = 1/2$

$$\Rightarrow e^{-\gamma\|x-x'\|^2} = e^{-\frac{1}{2}\langle (x-x')(x-x')\rangle}$$

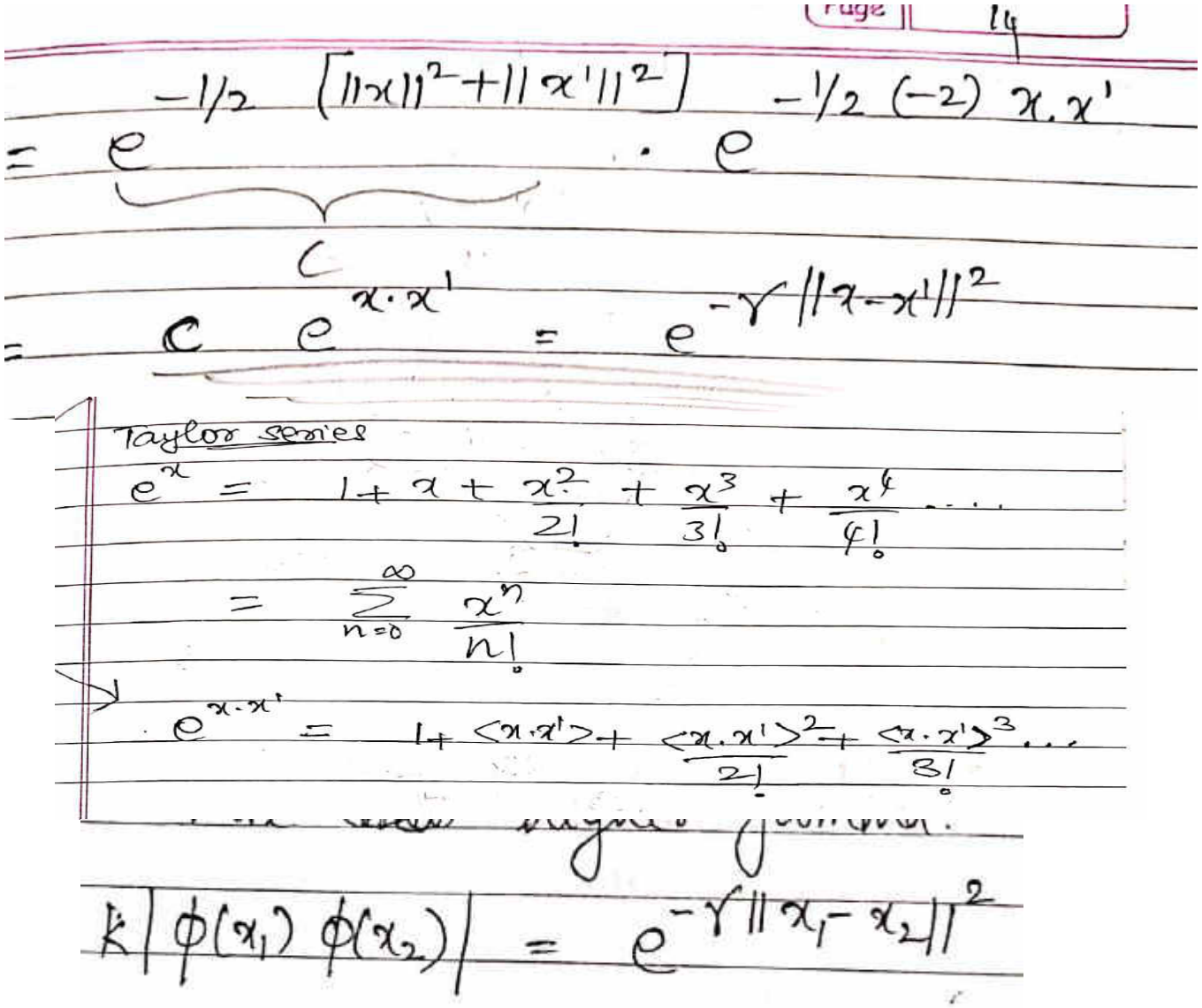
$$= e^{-\frac{1}{2}[x(x-x') - x'(x-x')]}$$

$$= e^{-\frac{1}{2}[xx - xx' - x'x + x'x']}$$

$$= e^{-\frac{1}{2}[\|x\|^2 - 2x\cdot x' + \|x'\|^2]}$$

MACHINE INTELLIGENCE

RBF



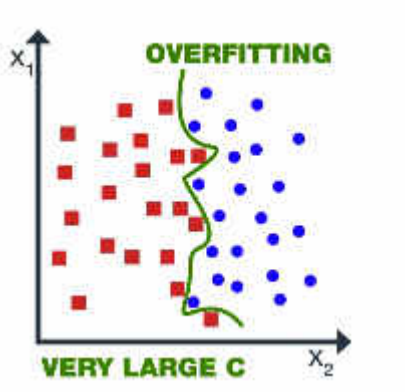
You can see that this takes you to infinite dimensions

Effect of Gamma

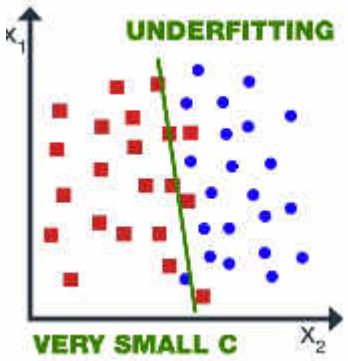
Technically, the gamma parameter is the inverse of the standard deviation of the RBF kernel (Gaussian function), which is used as similarity measure between two points. Intuitively, a small gamma value define a Gaussian function with a large variance. In this case, two points can be considered similar even if are far from each other. In the other hand, a large gamma value means define a Gaussian function with a small variance and in this case, two points are considered similar just if they are close to each other.

MACHINE INTELLIGENCE

Intuition behind C and Gamma



If the value of Gamma is high, then our decision boundary will depend on points close to the decision boundary and nearer points carry more weights than far away points due to which our decision boundary becomes more wiggly.



If the value of Gamma is low, then far away points carry more weights than nearer points and thus our decision boundary becomes more like a straight line.

THANK YOU

K.S.Srinivas
srinivasks@pes.edu
+91 80 2672 1983 Extn 701