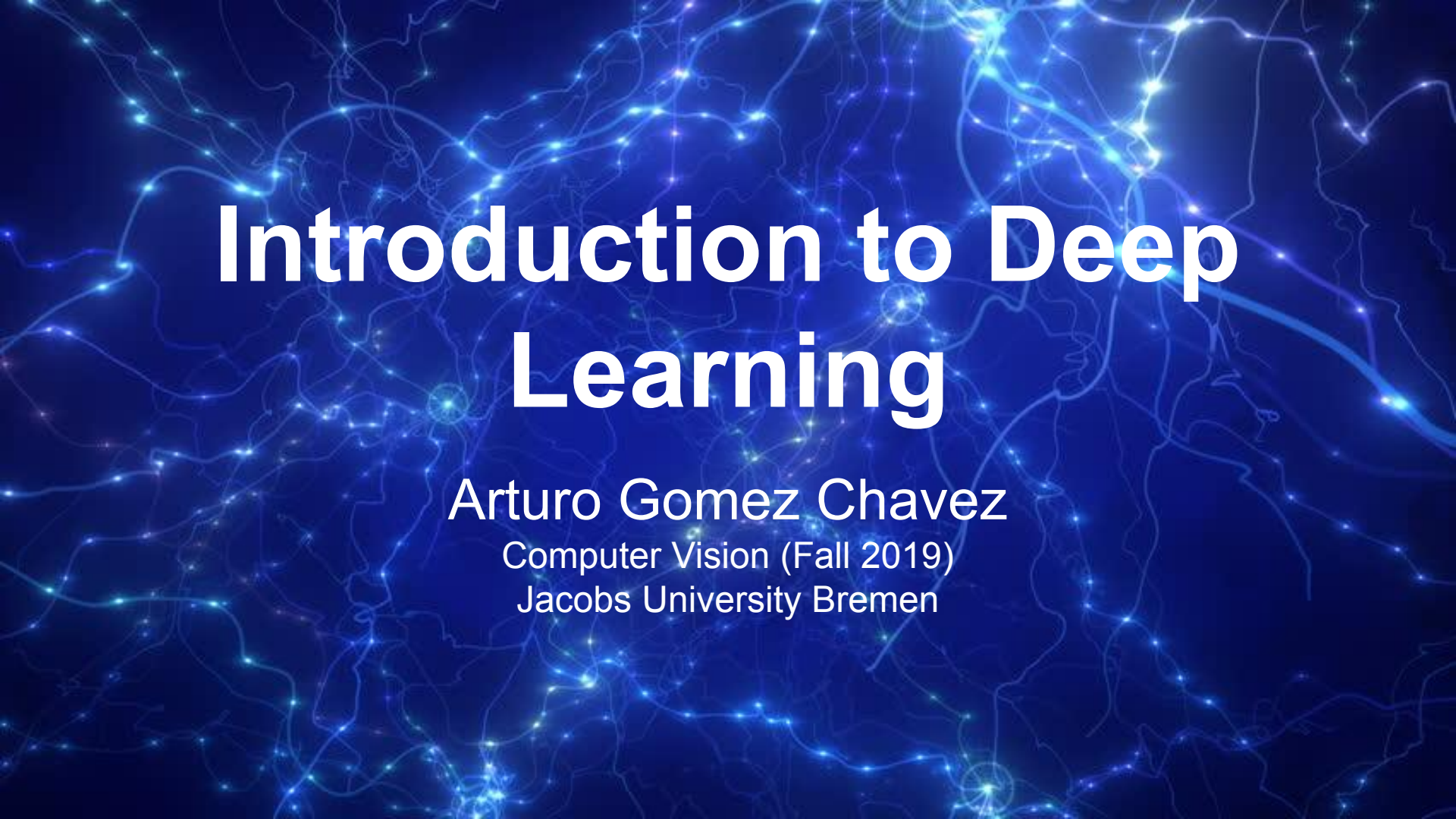
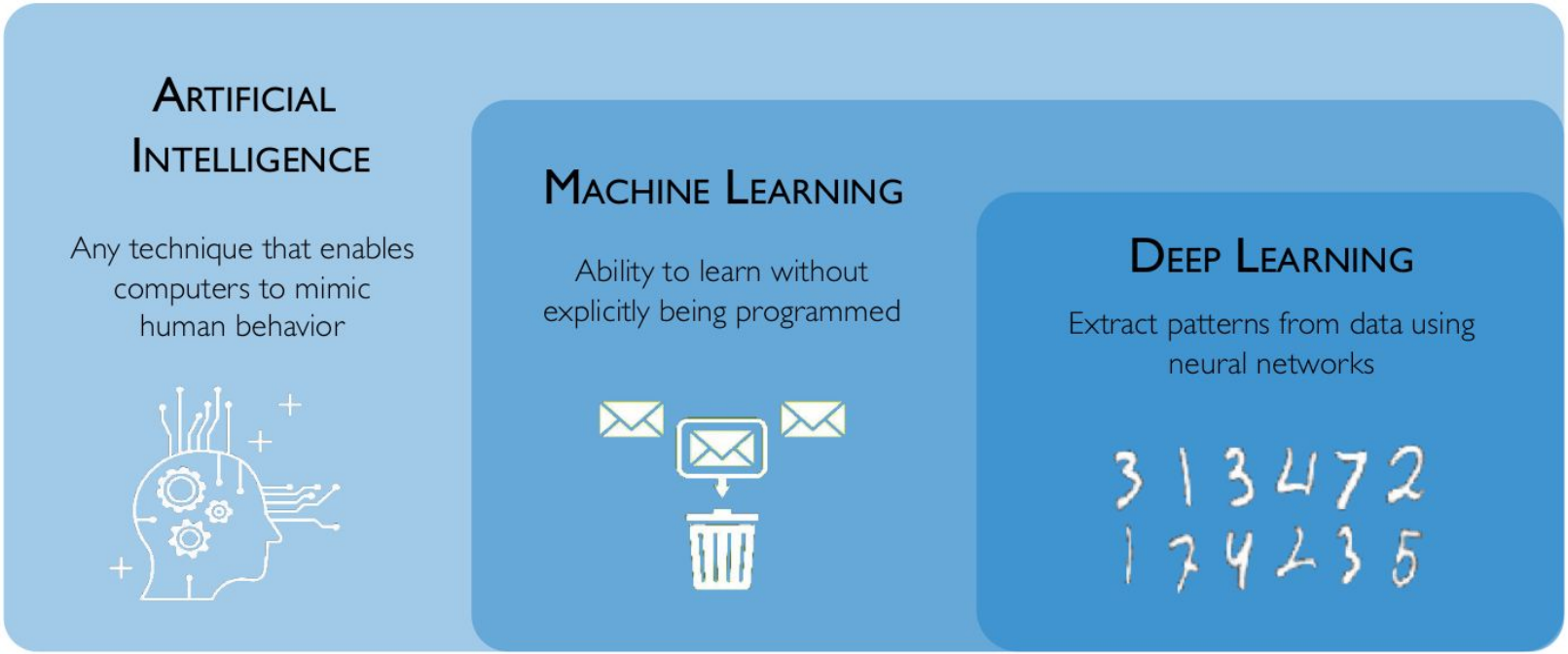


Introduction to Deep Learning

Arturo Gomez Chavez
Computer Vision (Fall 2019)
Jacobs University Bremen

The Rise of Deep Learning

WHAT is Deep Learning?



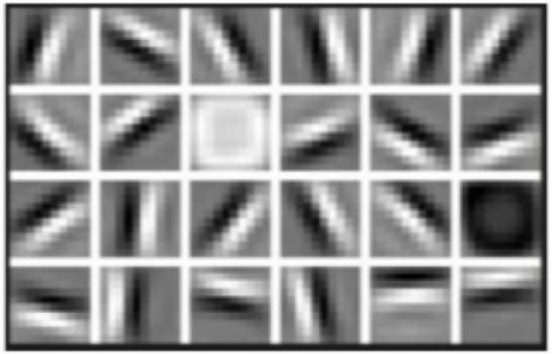
Source: MIT 6.S191 Introduction to Deep Learning http://introtodeeplearning.com/

WHY Deep Learning?

Hand engineered features are time consuming, brittle and not scalable in practice

Can we learn the **underlying features** directly from data?

Low Level Features



Lines & Edges

Mid Level Features

Eyes & Nose & Ears

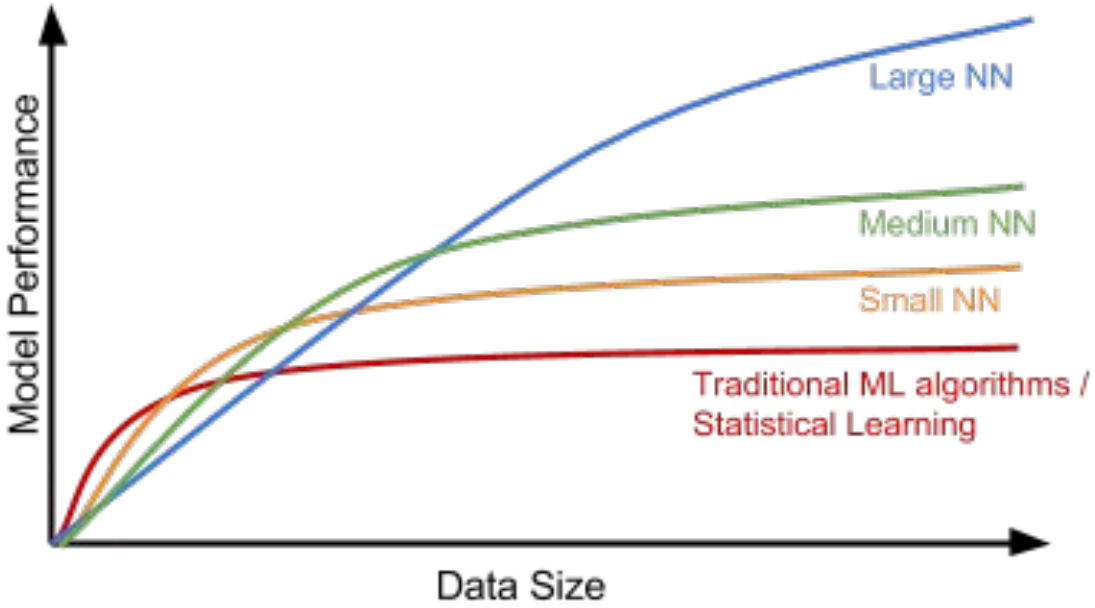
High Level Features

Facial Structure

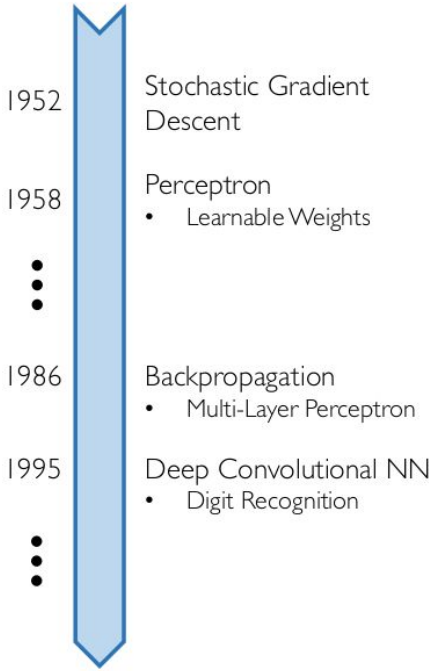
Source: MIT 6.S191 Introduction to Deep Learning http://introtodeeplearning.com/

WHY Deep Learning?

- Deep Neural Networks improve accuracy as more data is available, unlike traditional ML methods.



WHY NOW ?



Neural Networks date back decades, so why the resurgence?

1. Big Data

- Larger Datasets
- Easier Collection & Storage

IMAGENET

2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable

3. Software

- Improved Techniques
- New Models
- Toolboxes

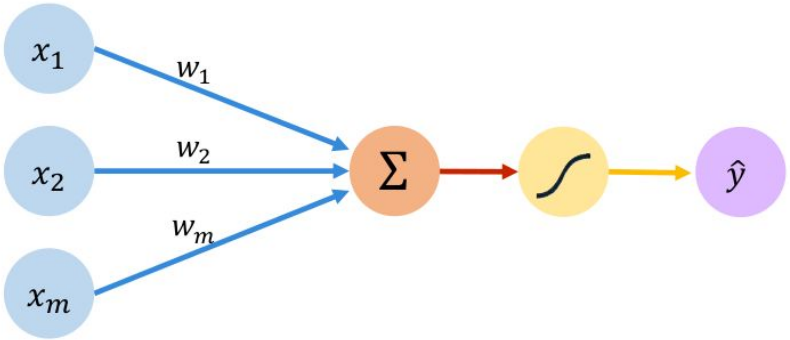
TensorFlow

Source: MIT 6.S191 Introduction to Deep Learning http://introtodeeplearning.com/

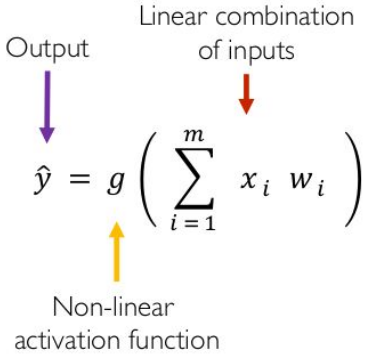
The perceptron

The structural building block of deep learning

Perceptron - Feed Forward NN

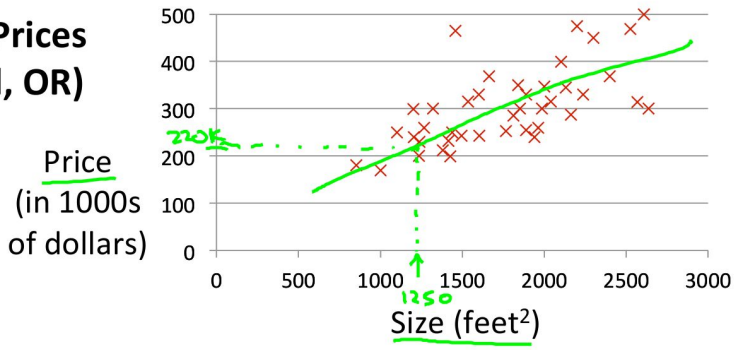


Inputs Weights Sum Non-Linearity Output



Basics first - Linear Regression

Housing Prices (Portland, OR)



Supervised Learning

Given the "right answer" for each example in the data.

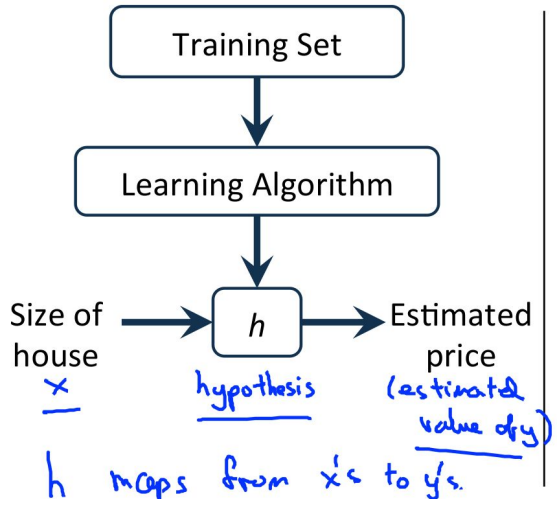
Regression Problem

Predict real-valued output

Classification: Discrete-valued output

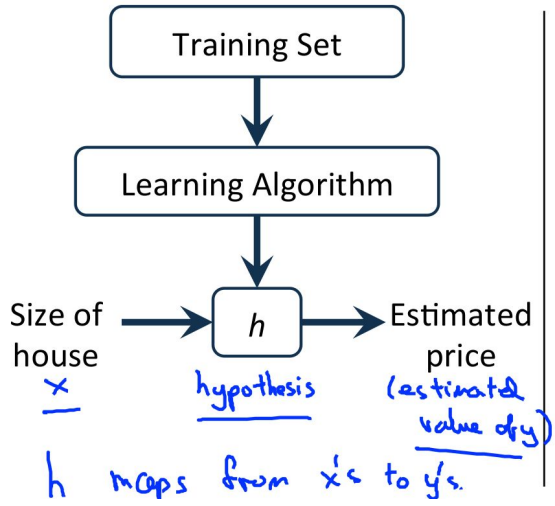
- In order to understand the advantages of Neural Networks and how they work, it is good to remember how Linear Regression works.
- Given labeled data (supervised learning), we can try to create a function that predicts the behavior of the data (regression).

Basics first - Linear Regression



- Any learning algorithm, including linear regression, uses available input data (training set) to try and come up with a hypothesis *h*.
- The hypothesis is then used to mapped new inputs to outputs, i.e., predict outcomes given new data, based on a collection of past inputs/outputs.

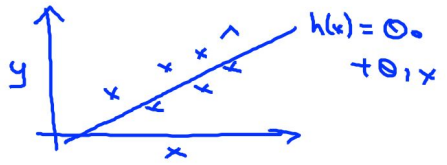
Basics first - Linear Regression



How do we represent *h* ?

$h_\theta(x) = \theta_0 + \theta_1 x$

Shorthand: $h(x)$



Linear regression with one variable. (x)
Univariate linear regression.

- For linear regression with one variable, the hypothesis looks like the equation of of a line $y=mx+b$ (adjust to notation)

Basics first - Normal equation

Examples: $m = 4$.

x_0	Size (feet²) x_1	Number of bedrooms x_2	Number of floors x_3	Age of home (years) x_4	Price ($1000) y
1	2104	5	1	45	460
1	1416	3	2	40	232
1	1534	3	2	30	315
1	852	2	1	36	178

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix} \quad y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

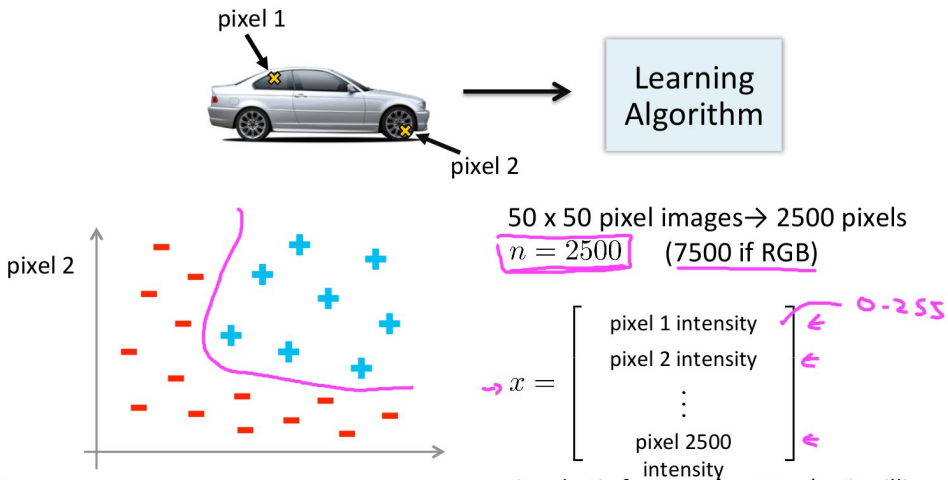
m x (n+1)

m-dimensional vector

$$\theta = (X^T X)^{-1} X^T y$$

- It is important to think about the data available before deciding WHICH learning algorithm to implement, and HOW.
- To apply minimum square error to data in linear algebra fashion, the normal equation is sufficient.
- This outputs the optimized values for the equation modelling the data (theta in this example).
- Computation is immediate, no learning time required (this does not refer to computation time, i.e., time requires to perform matrix operations)

Basics first - Normal equation ???



- The problem arises when the number of training examples and variables becomes larger and larger.
- This is often the case in computer vision, where technically each pixel is an input variable.
- Even for small gray-scale images (50x50 pixels) the number of variables is 500 greater than the house-price example from the previous slide.
- In these cases, the calculation of the normal equation becomes too computationally expensive.

Basics first - Gradient Descent

m **training examples,** n **features.**

Gradient Descent	Normal Equation
• Need to choose α.	• No need to choose α.
• Needs many iterations.	• Don't need to iterate.
• Works well even when n is large.	• Need to compute $(X^T X)^{-1}$
	• Slow if n is very large.

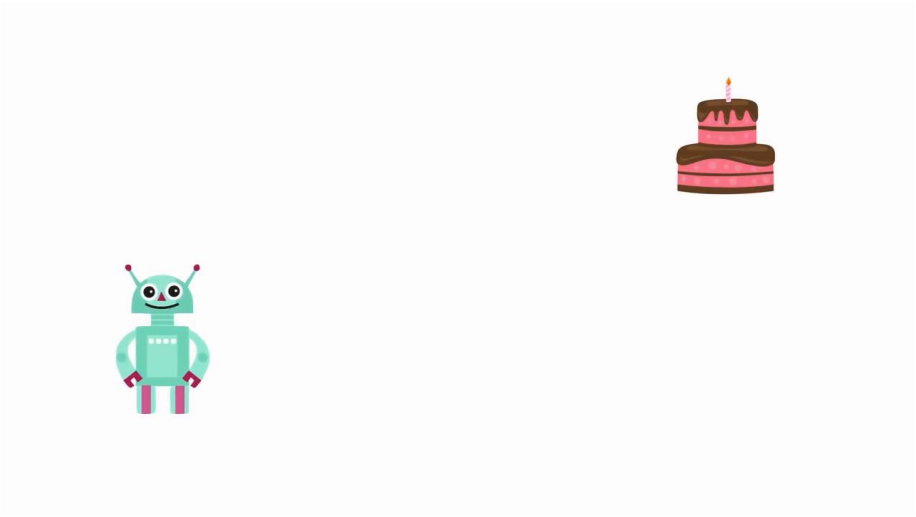
$n = 10^6$

$n \times n$ $O(n^3)$

$n = 100$
$n = 1000$
$n = 10000$

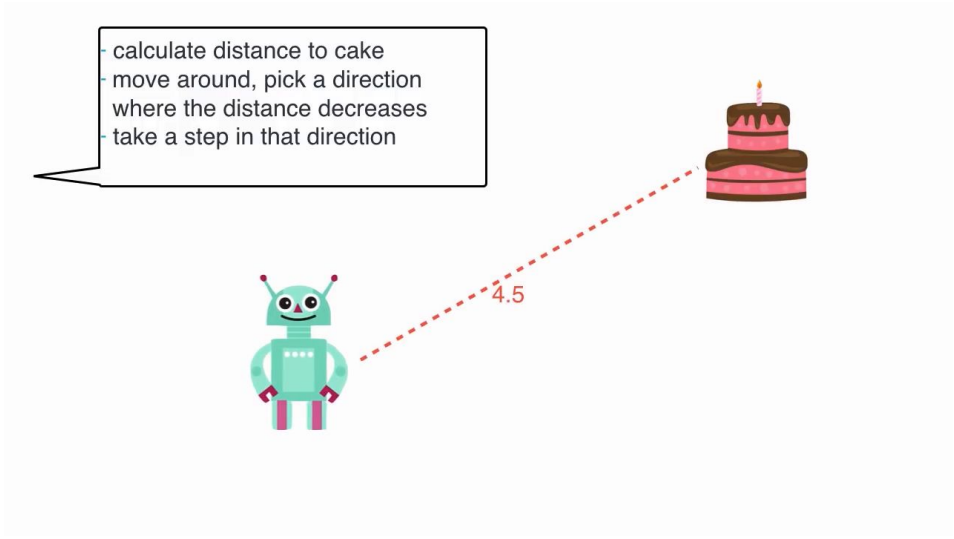
- Here alpha refers to learning rate, which is addressed in the next slides.
- This slide presents another option to the Normal Equation: Gradient Descent.
- Although Gradient Descent requires several number of iterations, it can handle well large number of samples *(m)* and variables *(n)*.
- The normal equation has complexity O(n^3), n= number of features and variables.
- As a rule of thumb (you are welcome to experiment yourself), when the number of features is larger than 10K, use Gradient Descent. Normal Equation, otherwise.

Gradient Descent - Help the robot



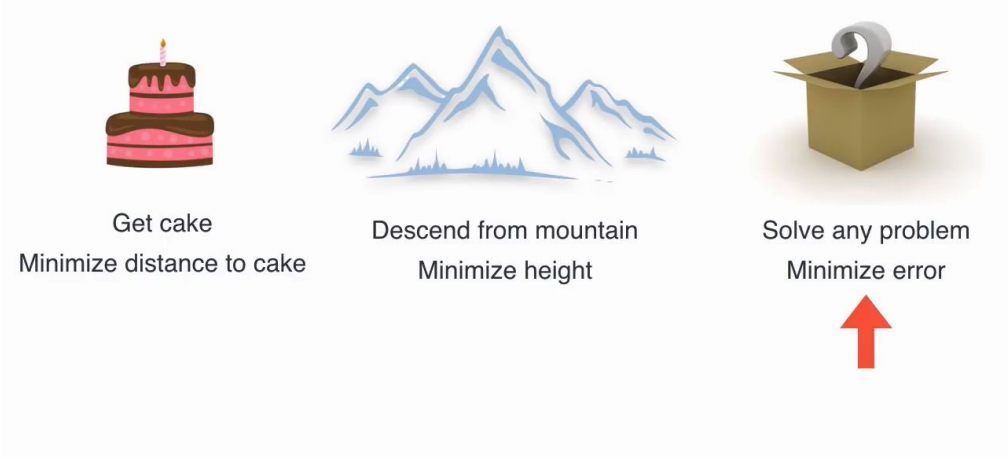
- To introduce GD, think of this: How would you tell a robot go the cake? Or think of a person who is blindfolded and only knows two words (close and far).

Gradient Descent - Help the robot



- The most simplistic approach would be for the robot to make an initial guess, make a sensor reading to the cake, if the distance decreases → move in the same direction, if the distance increases → pick a different direction.

Gradient Descent



- Broadly speaking, this is the same approach used to compute Gradient Descent, which objective is minimize the error of our hypothesis.
- Another analogy is finding the shortest path to climb down a mountain.

Gradient Descent

Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

Parameters: θ_0, θ_1

Cost Function: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$

Goal: $\underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

- These are the important factors (variables, parameters, hypothesis, cost function).
- This is the notation that we will follow.

Gradient Descent

Have some function $J(\theta_0, \theta_1)$ $J(\theta_0, \theta_1, \theta_2, \ldots, \theta_n)$

Want $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$ $\min_{\theta_0 \ldots \theta_n} J(\theta_0, \ldots, \theta_n)$

Outline:

- Start with some θ_0, θ_1 (Say $\theta_0 = 0, \theta_1 = 0$)
- Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$

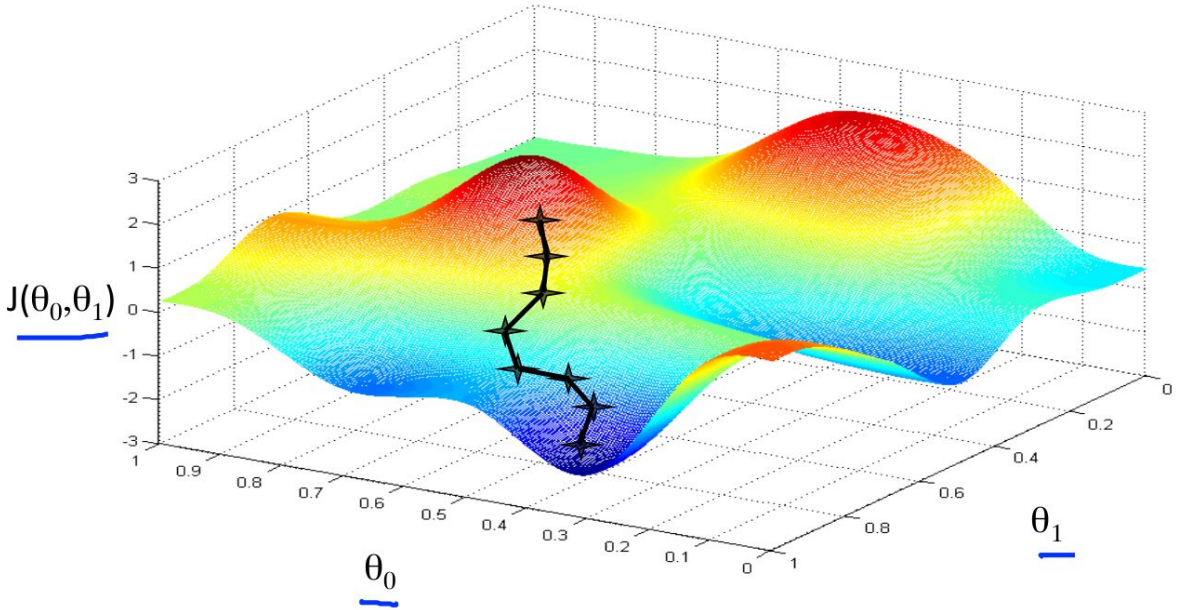
  until we hopefully end up at a minimum

(Remember how the robot works to find the cake)

- The objective of GD is to minimize the cost function by finding the optimal parameters (theta) that achieve this.
- To do this, GD initially guesses a value for each theta. The it computes the value of the cost function and if the value decreases, then it continues changing the values of theta in the same way as before.

Gradient Descent

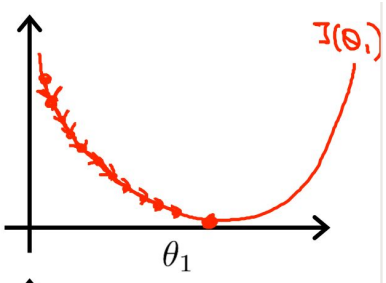
- Visual representation of how GD, changes the values of the parameters to find the minimum value of the cost function.



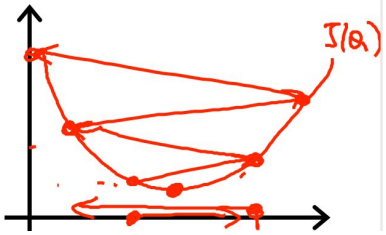
Gradient Descent

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

If α is too small, gradient descent can be slow.



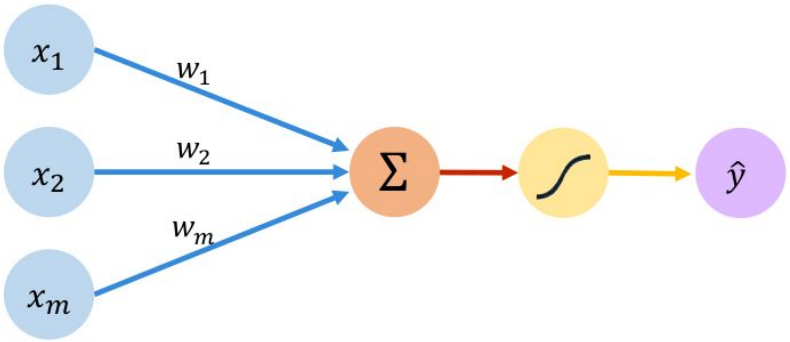
If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.

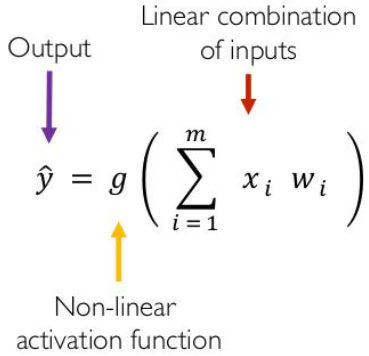


- The formula here shows the formula used to change the values of the parameters (theta) based on the cost function.
- Here is where the learning rate (alpha) comes into play.
- The derivative of the cost function dictates the “direction” (positive or negative) in which the value of theta will change.
- The value of learning rate influences how rapid the value fo the parameters (theta) changes.

Perceptron - Feed Forward NN

- Gradient Descent is the method used to adjust the weights *w* in a perceptron, based on the predicted values *y*.





Perceptron - Feed Forward NN

- As an example, we use a case where we want to split (classify data).

Get cake

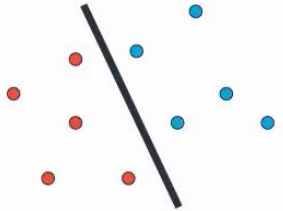
Distance to cake

continuous function

Descend from mountain

Height

continuous function



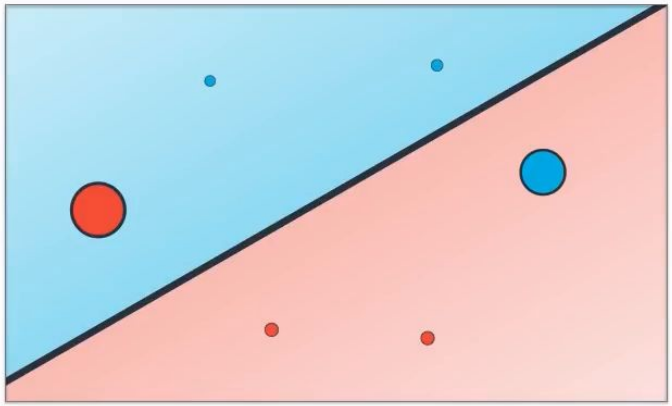
Split data

Number of errors

discrete function

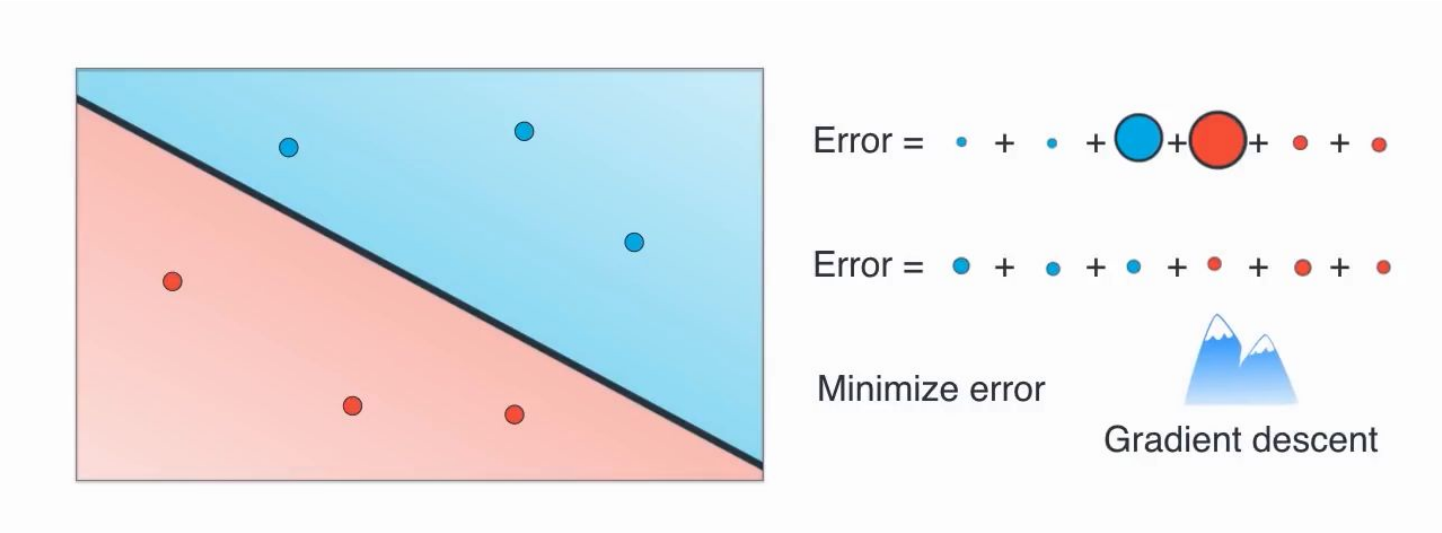
Perceptron - Example

- Blue dots (samples) represent dogs, red dots represent cats.
- The black line is our hypothesis (function,decision boundary) used to classified our samples.
- The error is computed as the distance from the samples to the decision boundary, the misclassified samples will weight more in the error calculation.



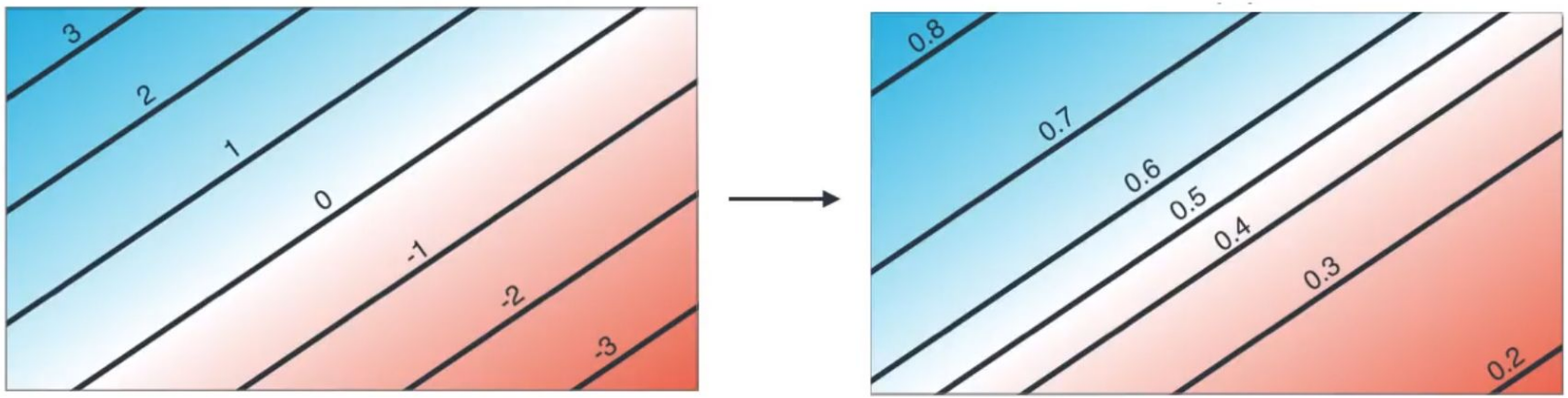
Perceptron - Example

- Given the past error, we perform GD and adjust the decision boundary.
- If done well, the new decision boundary will have a smaller error and not misclassified samples.



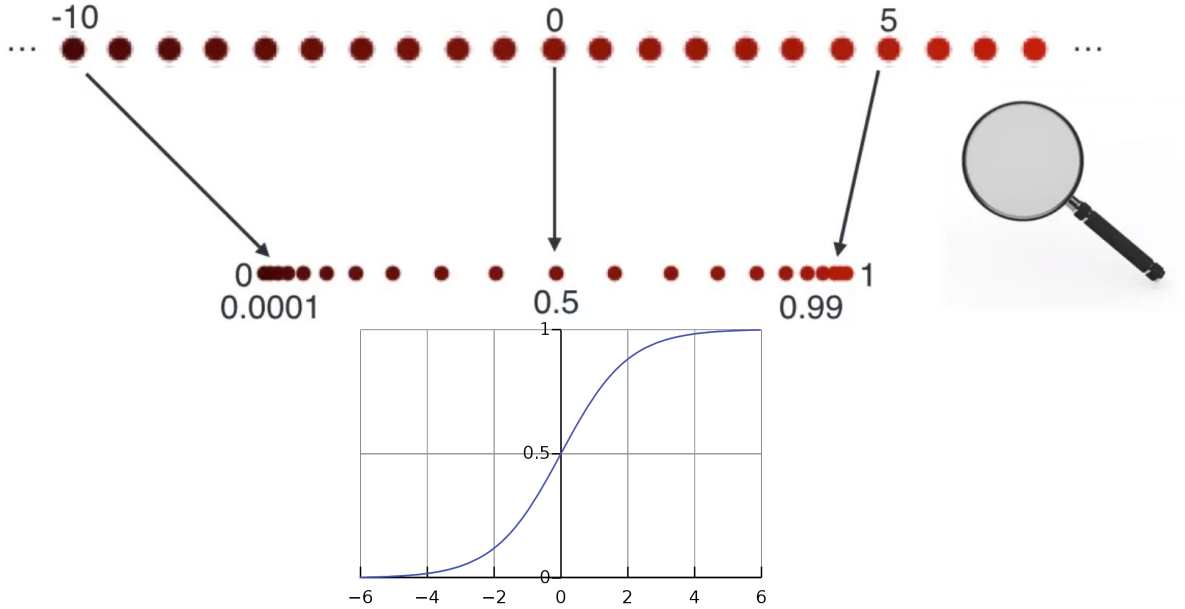
Perceptron - Error function?

- As mentioned *J* (error or cost function) is based on “distance” which range goes from -inf to +inf.
- However, for classification probabilities are better suited.
- We can map the previous *J* value to probabilities (how likely is it dog)



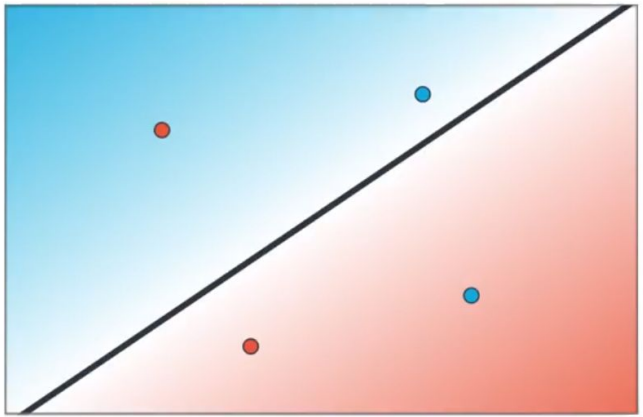
Perceptron - Activation Function

- To achieve this, we apply an activation function *g(x)* to map linear value to probabilities.
- Usually the sigmoid function is used in these cases.



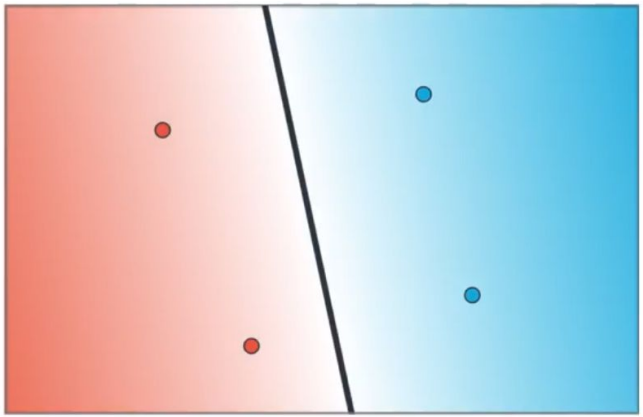
Perceptron - Error function (Intuition)

- Now, we have probabilities for each sample.
- To compute the OVERALL probability of the decision boundary, probabilities need to be multiply. Logarithmic functions are used for scale (see logistics regression).



0.6*0.2*0.1*0.7 = 0.0084

-log(0.6) - log(0.2) - log(0.1) - log(0.7) = 4.8

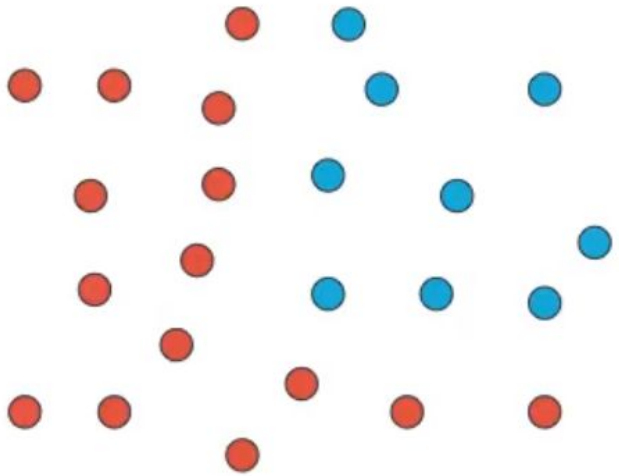


0.7*0.9*0.8*0.6 = 0.3024

-log(0.7) - log(0.9) - log(0.8) - log(0.6) = 1.2

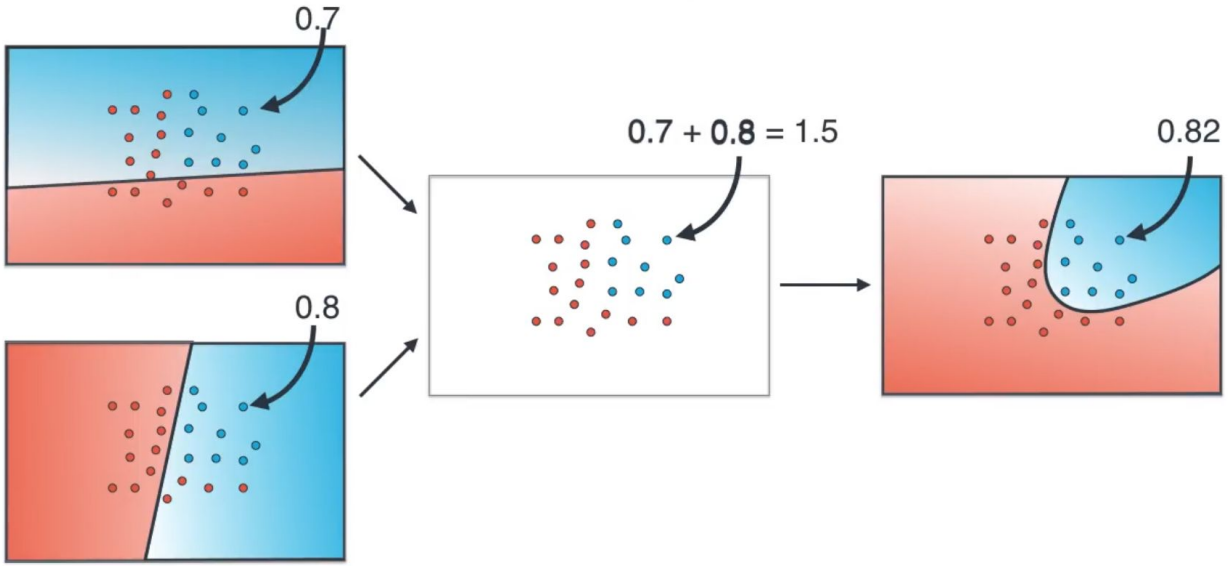
Perceptron - How to solve this?

- How to solve (create a decision boundary) for very complex data?



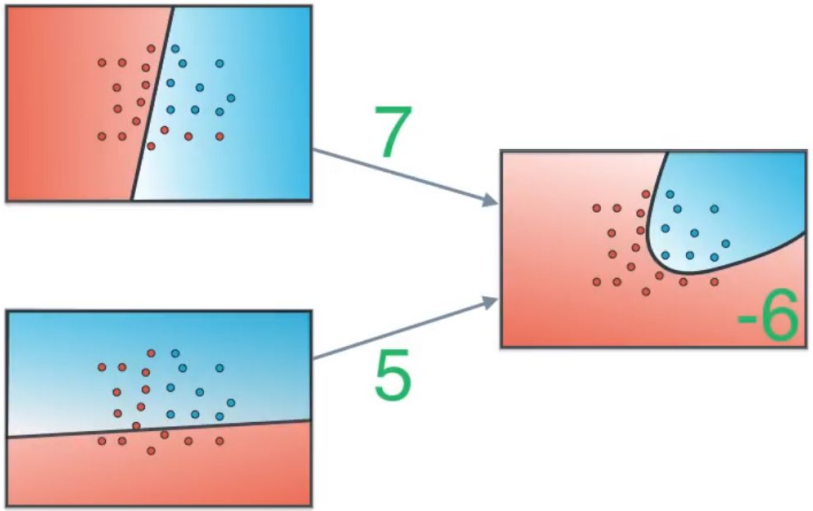
Perceptron - Non linear regions

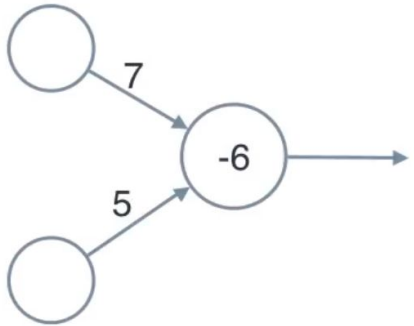
- How about we create 2 decision boundaries and combine their predictions?
- Each error (log probability) is added, which in turn clusters the data in non-linear ways.



Perceptron - Non linear regions

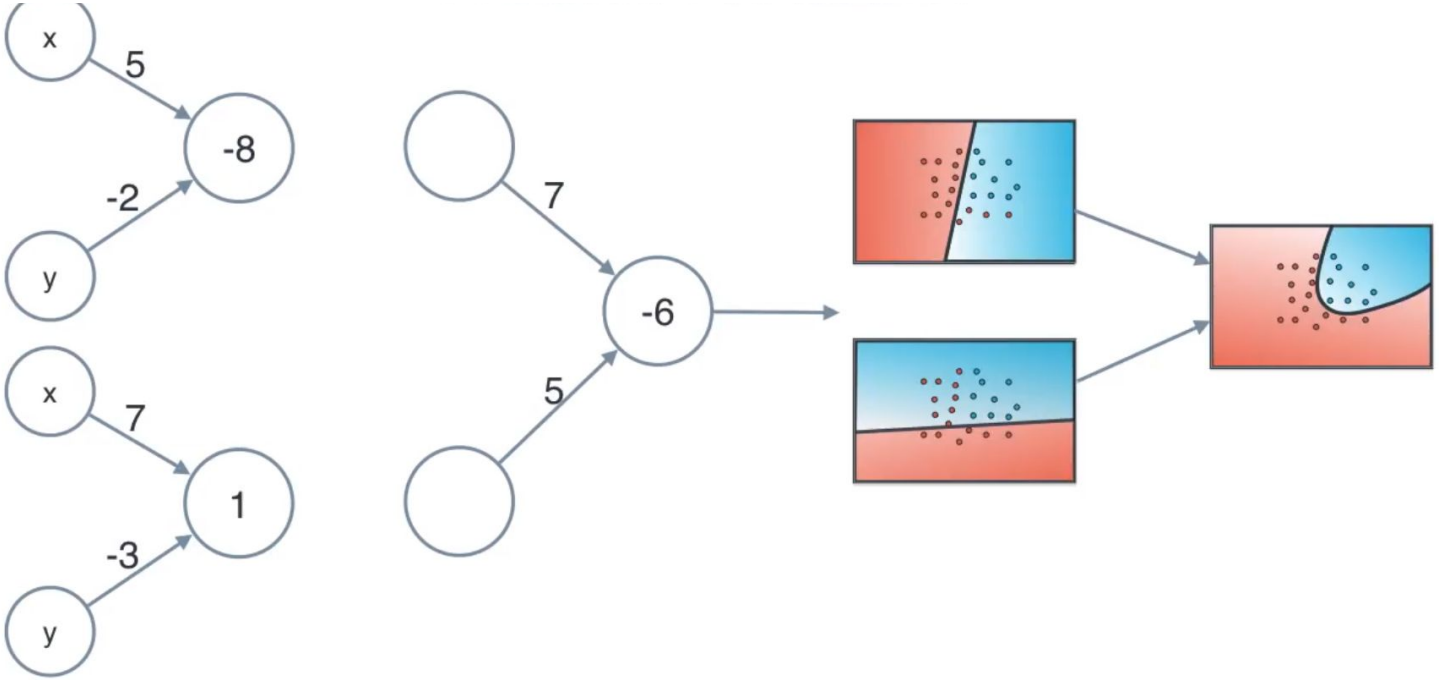
- We can also weight the “predictions” to influence the shape of the non-linear decision boundary.
- In the right, we can see the Neural Network representation of this.





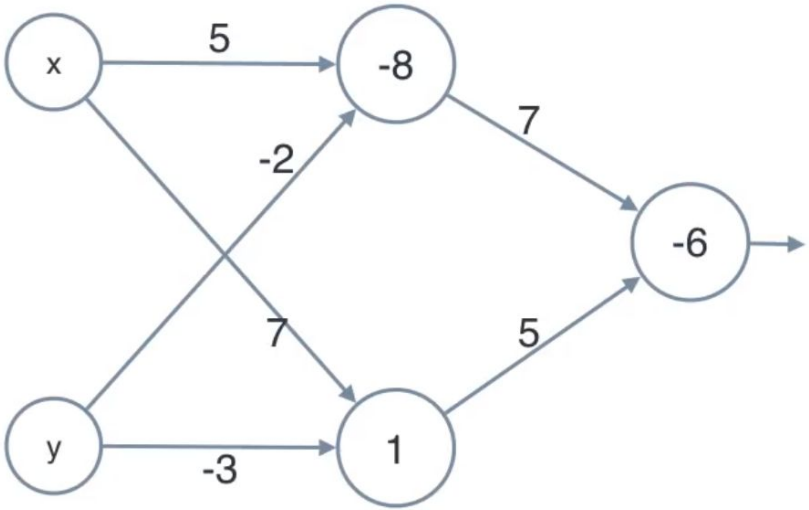
Perceptron - Non linear regions

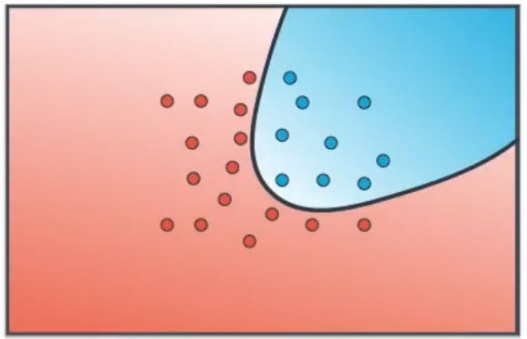
- Neural network representation of this example.



Perceptron - Non linear regions

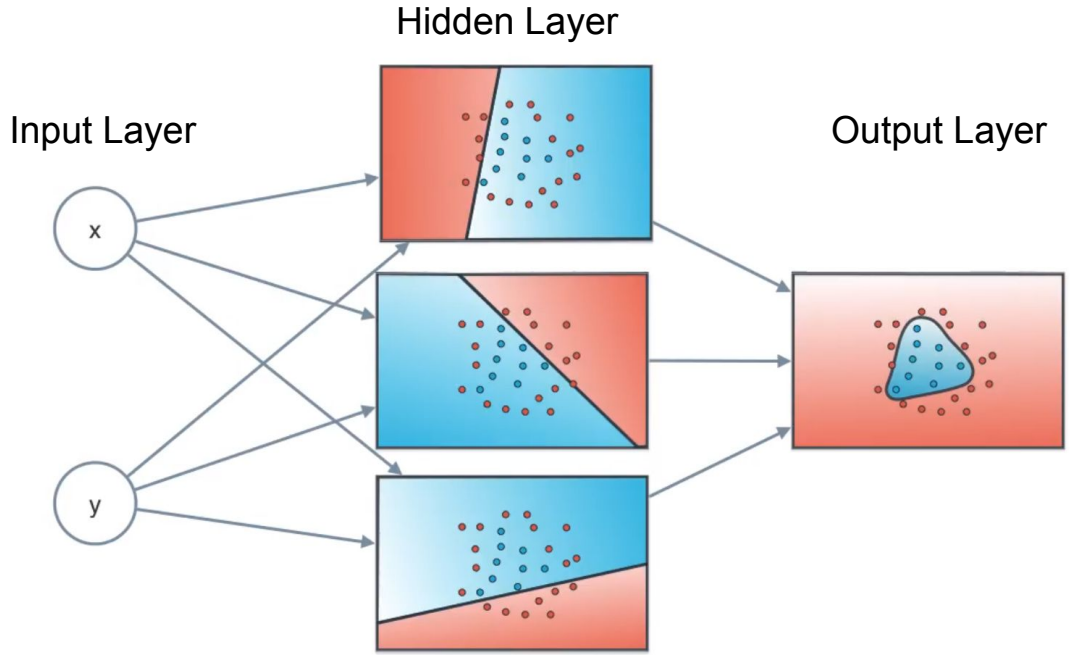
- We can further clean up the NN representation to be more clear.





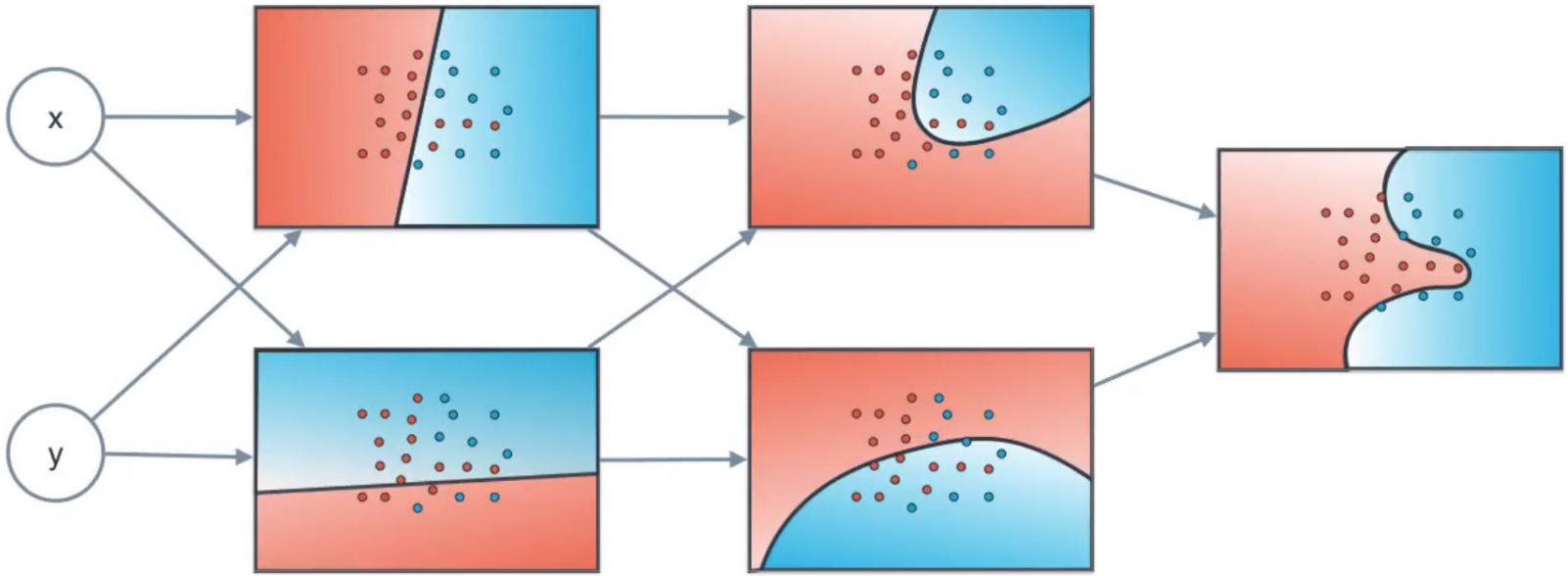
Perceptron - Non linear regions

- We can combine as many “predictions” as needed to create different decision boundaries. These predictions lie in the called **Hidden Layer**.



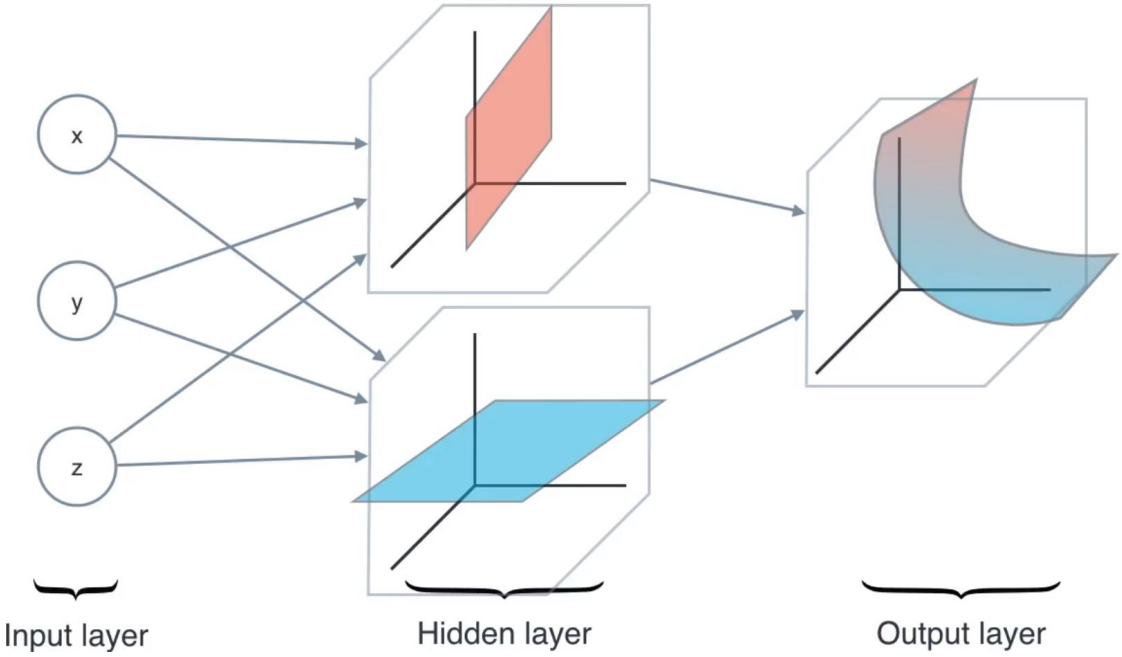
Perceptron - Non linear regions

- The hidden layer can also consist of several layers in series, which in turn will create more complex decision boundaries (“make it deeper”).



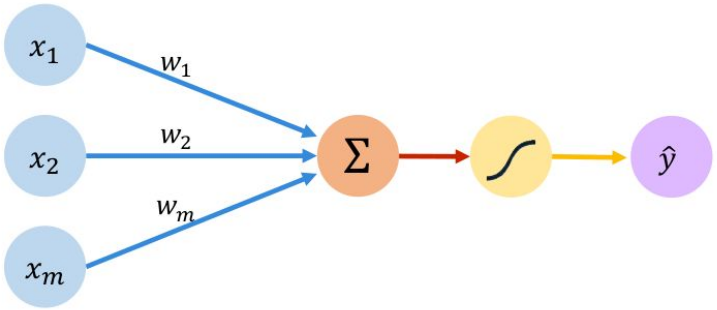
Perceptron - Non linear regions

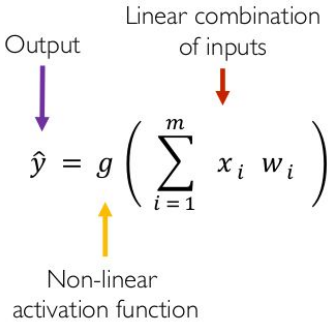
- We used an example with 2 variables for visualization, but the input layer can have any number of inputs (e.g. number of pixels in an image).



Perceptron - Feed Forward NN

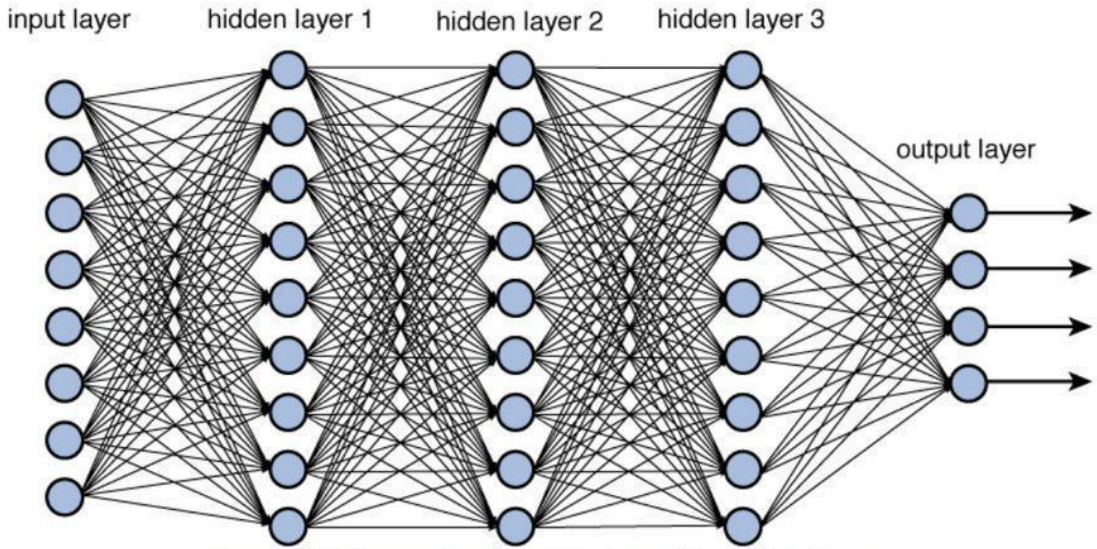
- Summary:
  - Perceptron combines a multitude of linear functions to adapt to the complexity of the data
  - To make the functions non-linear, an activation function is added.
  - Based on the error, the weights *w,* for each decision boundary and input is adjusted using Gradient Descent due to the usually large number of inputs *x_m*.





Perceptron - Feed Forward NN

- Then you can create something like this!!!!



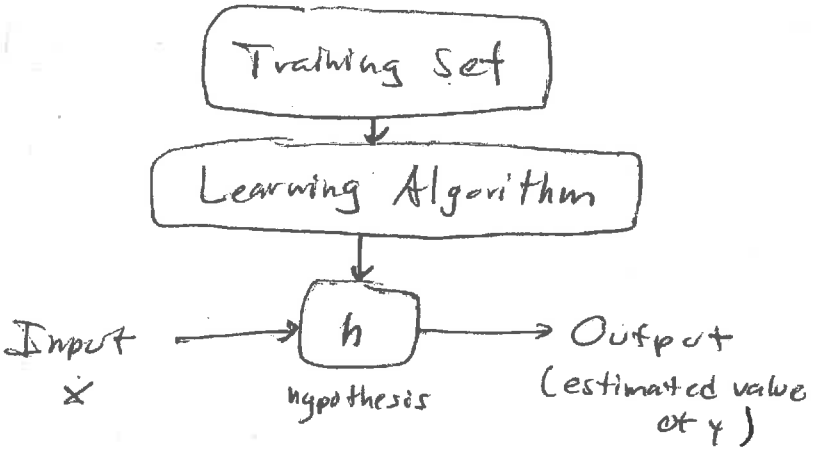
Thanks

* Linear Regression

- Supervised Learning.
- Prediction of real-valued output.

Notation:

- m = # of training examples.
- x's = "input" variable / features
- y's = "output" variable / "target" variable.



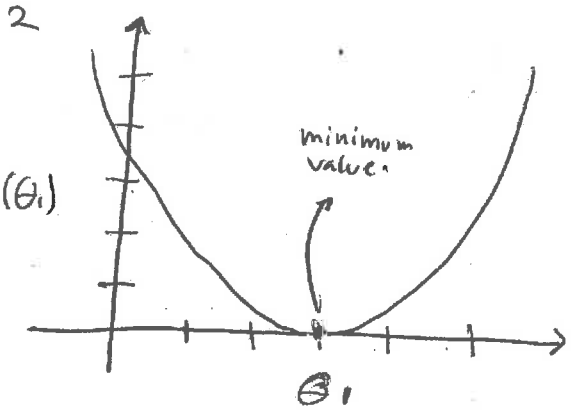
h maps from x's to y's. Ej: $h_\theta(x) = \theta_0 + \theta_1 x$. ($\theta$: Parameters.)

* Cost Function

$$\underset{\theta_0, \theta_1}{\text{minimize}} \; \underbrace{\frac{1}{2m}}_{\text{It is added just to deal with smaller values.}} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

$$\Rightarrow J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

$$\Rightarrow \underset{\theta_0, \theta_1}{\text{minimize}} \; \underbrace{J(\theta_0, \theta_1)}_{\text{Cost Function (Squared error function).}}$$



* Gradient Descent

- Have some function $J(\theta_0, \theta_1)$
- Want $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$.

- Algorithm:

repeat until convergence {

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

simultaneous update.

}

α → learning rate.

Correct: Simultaneous Update.

$$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$

$$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$

$$\theta_0 := \text{temp0}$$

$$\theta_1 := \text{temp1}$$

① If α is too small, gradient descent can be slow.

② If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.

③ α should not be variable necessarily, because GD will automatically take smaller steps ~~because~~ when it approaches a Local minimum.

* GD for Linear Regression

$$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) = \frac{\partial}{\partial \theta_j} \left[\frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \right]$$

$$= \frac{\partial}{\partial \theta_j} \left[\frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2 \right]$$

$$j=0: \frac{\partial}{\partial \theta_0} J(\theta_0,\theta_1) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\theta_0+\theta_1 x^{(i)}}_{h_\theta(x^{(i)})} - y^{(i)}$$

$$j=1: \frac{\partial}{\partial \theta_1} J(\theta_0,\theta_1) = \frac{1}{m}\sum_{i=1}^{m} \left(\underbrace{\theta_0+\theta_1 x^{(i)}}_{h_\theta(x^{(i)})} - y^{(i)}\right) x^{(i)}$$

* The cost function for linear regression is always a Convex Function, which only has 1 local optima = global optima.

"Batch" Gradient Descent

↳ Each step of GD uses all training examples ($\sum_{i=1}^{m}$).

* LINEAR REGRESSION W/ MULTIPLE VARIABLES

* Multiple Features

Notation:

n = number of features

$x^{(i)}$ = input (features) of i^{th} training example.

$x_j^{(i)}$ = value of feature j in i^{th} training example.

Hypothesis:

Previously: $h_\theta(x) = \theta_0 + \theta_1 x$

$\rightarrow h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$

- For convenience, define $x_0 = 1$.

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

$$\Rightarrow \boxed{h_\theta(x) = \theta^T x.}$$

* Gradient Descent for Multiple Variables

New algorithm ($n \geq 1$):

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

}

* Gradient Descent in Practice I – Feature Scaling

- Idea: Make sure features are on a similar scale.

  Get every feature into approximately a $-1 \leq x_i \leq 1$ range.

  Ex. (Range no larger than $|3|$ or less than $\left|\frac{1}{3}\right|$

- Mean normalization.

$$x_1 \leftarrow \frac{x_1 - \mu_1}{s_1}$$

→ GD converges quicker

s_1 → range of values. (max − min) / std. deviation.

* GD – Learning Rate

- $J(\theta)$ should decrease after every iteration.
- The number of iterations needed for the algorithm to converge varies a lot depending on the application.
- Look at plots! instead of defining a threshold
- To choose α, try multiples of 10 of α (Init: 0.001). (or 3):

Features and Polynomial Regression.

→ You can combine features into one to reduce the complexity of the algorithm if they're correlated.

→ You can modify the hypothesis $h_\theta(x)$, into another function.

Ex: $h_\theta(x) = \theta_0 + \theta_1 \text{size}$.

$h_\theta(x) = \theta_0 + \theta_1 \text{size} + \theta_2 \sqrt{\text{size}}$.

$h_\theta(x) = \theta_0 + \theta_1 \text{size} + \theta_2 \text{size}^2 + \theta_3 \text{size}^3$

↳ In this type. Feature scaling becomes more important.

* Normal Equation

- Method to solve for θ analytically.

x_0	Feature 1	Feature 2	. . .	Feature N	y
1	a	b		n	
1	a_2	b_2		n_2	
.	.	.		.	

$$\Rightarrow X = \begin{bmatrix} 1 & a_1 & b_1 & \cdots & n_1 \\ 1 & a_2 & b_2 & \cdots & n_2 \\ \vdots & \vdots & \vdots & & \\ 1 & a_m & b_m & \cdots & n_m \end{bmatrix} \qquad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

$$\Rightarrow \theta = (X^T X)^{-1} X^T y \quad (\text{pinv}(X' * X) * X' * y.)$$

- For Normal Equation No need to use Feature Scalling.

* m training examples, n features.

Gradient Descent	Normal Equation
• Need to choose α.	• No need to chose α.
• Needs many iterations.	• Don't need to iterate.
• Work well when n is large.	• Need to compute $(X^TX)^{-1}$ (n × n)
	• Slow if n is very large.

$O(n^3)$

Use GD. $n = 10^6$ ← → $n = 10,000$

* Normal Equation Non Invertibility

$$\theta = (X^TX)^{-1}X^Ty$$

- What if X^TX is non-invertible? (singular/degenerate).

pinv (pseudo-inverse) → always finds an invertible.

Causes:

① Redundant features (linearly dependant).

② Too many features (e.g. $m \leq n$).

- Delete some features, or use regularization.

* LOGISTIC REGRESSION

* Classification

0: "Negative class"
1: "Positive class"

$y \in \{0, 1\}$ → Binary classification

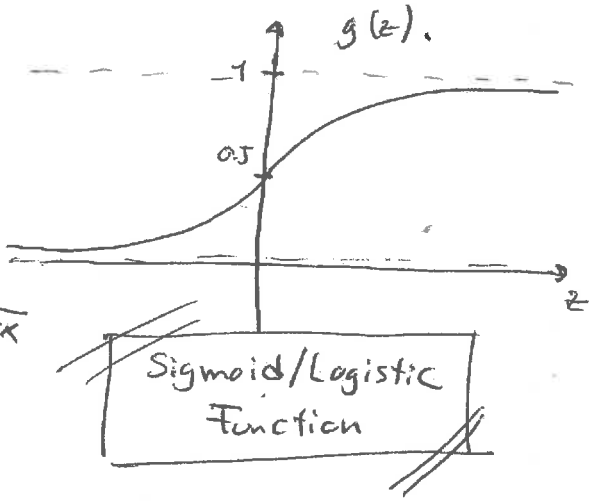
$y \in \{0, 1, 2 \ldots\}$ → Multi-classification

* Hypothesis Representation

- Want $0 \leq h_\theta(x) \leq 1$

⇒ $h_\theta(x) = g(\theta^T x)$

→ $g(z) = \frac{1}{1 + e^{-z}}$ ⇒ $h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$



- Interpretation

$h_\theta(x)$ = estimated probability that $y = 1$ on input x

$h_\theta(x) = p(y = 1 \mid x; \theta)$.

* Decision Boundary

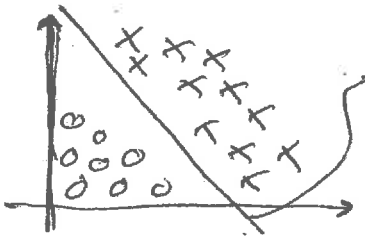
$h_\theta(x) = g(\theta^T x) = p(y = 1 \mid x; \theta)$

* Suppose predict "$y = 1$" if $h_\theta(x) \geq 0.5$ → $g(z) \geq 0.5$

↳ $\theta^T x \geq 0$ ← when $z \geq 0$

predict "$y = 0$" if $h_\theta(x) < 0.5$ → $g(z) < 0.5$

↳ $\theta^T x < 0$ ← when $z < 0$



Decision Boundary ⇒ $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$

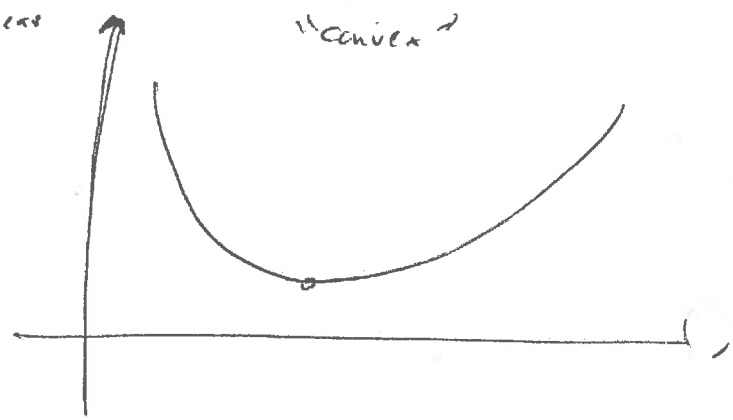
if "$y = 1$" ⇒ $\theta_0 + \theta_1 x + \theta_2 x_2 \geq 0$

* Cost Function

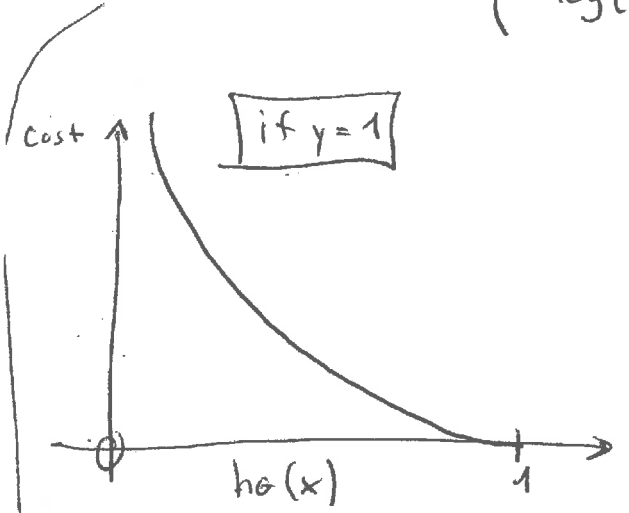
Logistic Regression: $J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$

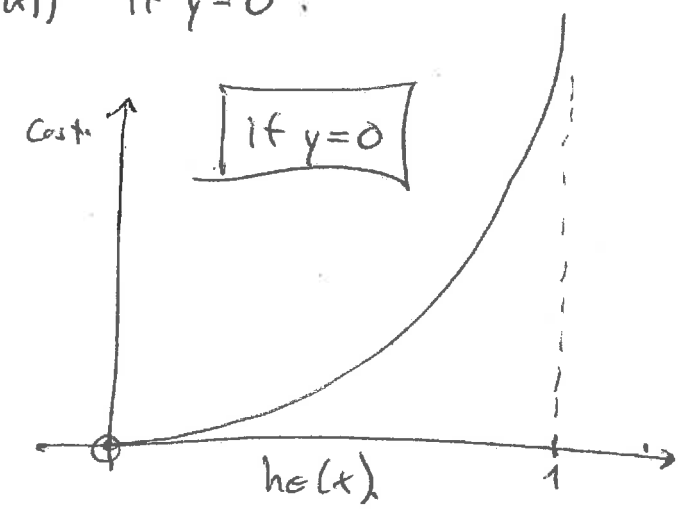
$\frac{1}{2} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 = \text{cost}(h_\theta(x^{(i)}), y)$

non-linear



$$\Rightarrow \text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$





$$\Rightarrow \boxed{\text{Cost}(h_\theta(x), y) = -y \log(h_\theta(x)) - (1-y) \log(1 - h_\theta(x))}$$

$$\Rightarrow \boxed{J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right]}$$

$$-\frac{1}{m} \left[y^{(i)} \frac{x_j^{(i)}}{h_\theta(x^{(i)})} + (1 - y^{(i)}) \frac{-x_j^{(i)}}{1 - h_\theta(z)} \right] \quad x_j^{(i)} \quad y^{(i)} - y h_\theta + h_\theta y$$

Gradient Descent

Want $\min J(\theta)$:

Repeat {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}

$$\Leftrightarrow \frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}.$$

Advanced Optimization

- Given θ, we have code that can compute.
  - $J(\theta)$
  - $\frac{\partial}{\partial \theta_j} J(\theta)$ for $j = 0, 1, \dots, n$.

These are needed by these advance methods

Optimization algorithms:

- Gradient descent
- Conjugate gradient
- BFGS
- L-BFGS

Advantages

- No need to manually pick α.
- Often faster than GD.

Disadvantages

- More complex.

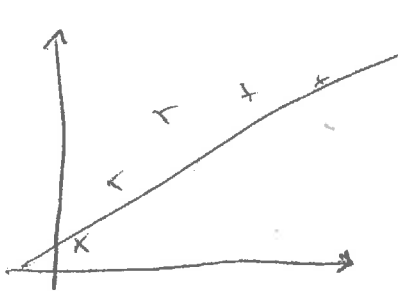
* Multi-class classification: (One vs all).

- Train a logistic regression classifier $h_\theta^{(i)}(x)$ for each class i to predict the probability that $y = i$.
- On a new input x, to make a prediction, pick the class i that maximizes

$$\max_i h_\theta^{(i)}(x).$$

* REGULARIZATION

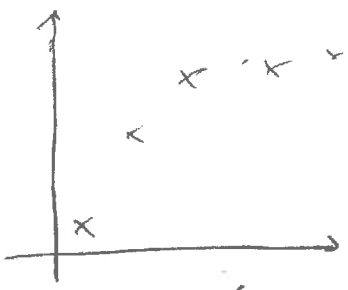
* Problem of Overfitting



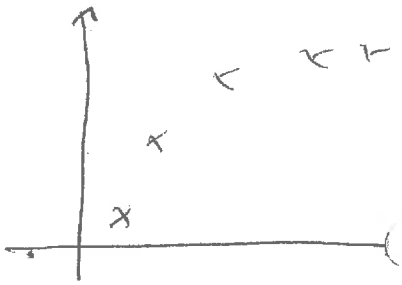
$\theta_0 + \theta_1 x$

"Underfit"

"High bias"



$\theta_0 + \theta_1 x + \theta_2 x^2$



$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4 + \cdots$

"Overfit"

"High variance"

- Addressing overfitting:

(1) Reduce number of features:
- Manually select them.
- Model selection algorithm.

(2) Regularization
- Keep all the features, reduce the magnitude/values of parameters θ_j.
- Works well when we have a lot of features, each of which contributes a bit to predicting y.

Makes values of θ_j small

$\theta_j \simeq 0$

$\Rightarrow \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

Get rid of ~~simp~~ more complex terms.

⊙ Overfitting: If we have too many features, the learned hypothesis may fit the training set very well $J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \simeq 0$, but fail to generalize new examples

* Cost Function

- Regularization

Small values for parameters $\theta_0, \theta_1, \cdots, \theta_n$

↳ Simpler hypothesis

↳ Less prone to overfitting.

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

- $\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$: In charge of making an accurate model.
- λ : Control the tradeoff of importance between predicting the data correctly and keeping θ_n small.
- $\sum_{j=1}^{n}\theta_j^2$: ↳ Regularization parameter.

* Regularized Linear Regression.

- Gradient Descent.

* θ_0 is not penalized by convention.

Repeat {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_0^{(i)}$$

$$\theta_j := \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} - \frac{\lambda}{m}\theta_j\right]$$

}

$$\Rightarrow \theta_j := \theta_j\left(1 - \alpha\frac{\lambda}{m}\right) - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}$$

$1 - \alpha\frac{\lambda}{m} < 1$ → To shrink values of θ_j

- Normal Equation

$$\theta = (X^T X)^{-1} X^T y$$

$$\Rightarrow \theta = \left(X^T X + \lambda \underbrace{\begin{bmatrix} 0 & & & \\ & 1 & & \\ & & 1 & \\ & & & \ddots \end{bmatrix}}_{(n+1)\times(n+1)} \right)^{-1} X^T y$$

Ej. $n=2$ $\begin{bmatrix} 0 & & \\ & 1 & \\ & & 1 \end{bmatrix}$.

⊙ Non-invertibility.

+ Suppose $m \leq n$

* If $\lambda > 0$, $\Rightarrow$ the matrix is invertible.

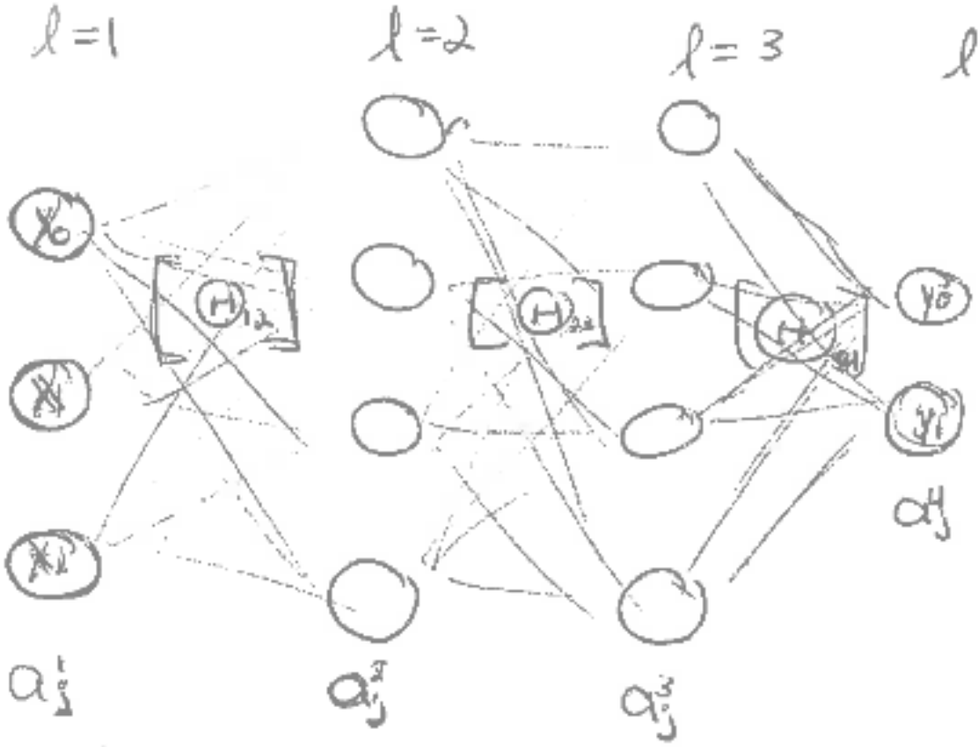
* Regularized Logistic Regression

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

$$+ \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2 \quad | \text{ not } \theta_0$$

*) Gradient Descent the same as for Regularized Linear Regression

Basic FF-NN Implementation



$x_j^{(i)}$ = input from the i-th training example, describing the j-th feature.

$\Theta^{(l)}_{NxS}$ = Weight matrix, mapping inputs from the l layer to the $l+1$ layer. Dims: $S_{l+1} \times S_l$

Note: $S_{l+1} \times S_l+1$, if bias is added.

$a_j^{(l)} := g(\Theta^{(l-1)} \cdot a_j^{(l-1)}) = g(z^{(l)})$ ~ activation function value

L = number of layers

m = number of training examples.

* Feed-forward pass

- Set $a_j^{(1)} = x_j$

→ $z^{(2)} = \Theta^{(1)} a^{(1)}$ ⇝ $g(z^{(2)}) = a^{(2)}$ $s_2 \times 1 / s_{l+1} \times 1$
add $a_0^{(2)}$ (bias term)

$s_2 \times 1$ $s_{l+1} \times 1$ | $s_2 \times s_1$ $s_{l+1} \times s_l$ | $s_1 \times 1$ $s_l \times 1$

→ $z^{(3)} = \Theta^{(2)} a^{(2)}$ ⇝ $g(z^{(3)}) = a^{(3)}$
add $a_0^{(3)}$ (bias term)

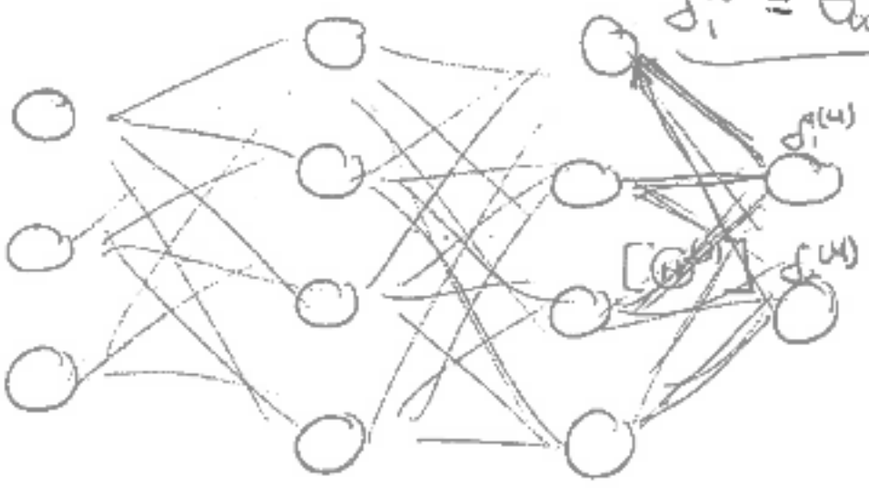
→ $z^{(4)} = \Theta^{(3)} a^{(3)}$ ⇝ $g(z^{(4)}) = a^{(4)}$
↳ No bias since it's last layer.

↳
Set $a_j^{(1)} = x_j$ (include bias)
For $l = 1$ to $l = L-1$:
$z^{(l+1)} = \Theta^{(l)} a^{(l)}$ ⇝ $g(z^{(l+1)}) = a^{(l+1)}$
add $a_0^{(l+1)}$ (bias)
if $l+1 \neq L$

---

* Back Propagation

$\delta_1^{(3)} = \Theta_{10}\delta_1^{(4)} + \Theta_{11}\delta_2^{(4)}$



Weighted sum of the error.

- Roughly we can express the error and derivative as follows:

$$E = \frac{1}{2m} \sum^{m} \sum^{K\text{-classes}} \left(g(\Theta x) - y \right)^2$$

$$\frac{\partial E}{\partial \Theta_j} \approx -\frac{1}{m} \left(y - g(\Theta x) \right) g'(\Theta x) x_j$$

$$\frac{\partial E}{\partial \Theta_j} \simeq (\text{error\_term})\, g'()(\text{input})$$

$g'()$ → activation fcn. derivative.

↓

- Computation of propagated error.

$$\delta^{(4)} = a_j^{(4)} - y_j \rightsquigarrow \delta^{(L)} = \left(a_j^{(L)} - y_j \right) g'\left(a_j^{(L)} \right)$$

↳ assumes there is no act. fcn. in last layer.

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

$$\delta^{(l-1)} = \left(\Theta^{(l-1)} \right)^T \delta^{(l)} .* g'(z^{(l-1)})$$

$s_{l-1} \times 1$; $s_{l-1} \times s_l$; $s_l \times 1$; $s_{l-1} \times 1$

$s_{l-1} \times 1$; $.*$; $s_{l-1} \times 1$

Mathematically it can be proved: $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$

$$\delta^{(l+1)} \cdot \left(a^{(l)} \right)^T = \Delta^{(l)} \rightsquigarrow \text{Update increment for weight matrix.}$$

$s_{l+1} \times 1$; $1 \times s_l$; $s_{l+1} \times s_l$

* BP-Algorithm

- Training set: $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
- Set $\Delta_{ij}^{(l)} = 0 \quad \forall_{i,j,l}$ → Accumulators for partial derivatives

- For $i = 1$ to m:
  - → Set $a^{(1)} = x^{(i)}$ [Input layer]
  - → Forward Propagation → compute all $a^{(l)}$
  - → $\delta^{(L)} = (a^{(L)} - y^{(i)})\, g'(z^{(L)})$ [ERROR OUTPUT LAYER]
  - → Compute: $\delta^{l} \; \forall \{1 < l < L\}$
    - ↳ $\delta^{(l)} = \underbrace{(\Theta^{(l)})^T \delta^{(l+1)} .* g'(z^{(l)})}$
      Weight propagation derivative.
  - → Accummulate:
    - ↳ $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
      $:= \Delta^{(l)} + \delta_i^{(l+1)} (a_j^{(l)})^T$

$$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$$
$$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} \quad \text{(Bias term)}$$
} Final derivative.

- Update weights:

$$\Theta_{ij}^{(l)} := \Theta_{ij}^{(l)} + \alpha D_{ij}^{(l)}$$

↳ learning rate