RECAP ON NOTATION:

He likes to use: $x_j^{(i)}$ → Example $i = 1 \to m$; Feature $j = 1 \to n$

PROPER MATRIX NOTATION:

$\vec{h}_\theta(x) = m \times 1$ vector

$\vec{y} = m \times 1$ vector

$\vec{\theta} = (n+1) \times 1$ vector

$X_{ij} = m \times (n+1)$ matrix = $\begin{bmatrix} | & | & | & | & \\ \vec{x}_1 & \vec{x}_2 & \vec{x}_3 & \cdots & \\ | & | & | & | & \end{bmatrix}$ ↓ examples, → features

i = example, j = features

ASIDE: MINIMIZATION ALGORITHMS $\min J(\vec{\theta})$ ⇒ How? (He covers G.D.)

Basic GD: $\vec{\theta}_{k+1} = \vec{\theta}_k - \alpha \vec{\nabla} J$ (best guess on α)

NEWTONS METHOD: well known, converges quickly in k, computationally expensive per step ($\mathcal{O}(n^3)$) — inverting & calculating H long!

$\vec{\theta}_{k+1} = \vec{\theta}_k - H^{-1} \vec{\nabla} J(\theta)$

$H_{ij} = \frac{\partial^2 J}{\partial \theta_i \partial \theta_j}$ = HESSIAN MATRIX

Option Between GD (uses ∇J) & Newton (using $\nabla J, \nabla^2 J$)? → CONJUGATE GRADIENT

Works extremely well for quadratic optimizations of form: $Qx = b \iff \min \frac{1}{2} x^T \theta x - b^T y$

can show this is exactly linear regression with: $(x^T x)\theta = x^T y \iff \min \frac{1}{2} \theta^T (x^T x)\theta - (x^T y)^T \theta$

Uses conjugate vectors: $d_i^T Q d_j = 0 \quad i \neq j$ ⇒ new orthogonal (conjugate) basis set (lin. ind.)

use these vectors to create an iterative update scheme → ALWAYS CONVERGED EXACTLY WITHIN N STEPS

$x^0 = \text{start}$	$d_0 = -g_0$
$x_{k+1} = x_k + \alpha_k d_k$	$d_{k+1} = -g_{k+1} + \beta_k d_k$
$g_k = Q x_k - b$	
$\alpha_k = -\frac{g_k^T g_k}{d_k^T Q d_k}$	$\beta_k = \frac{g_{k+1}^T g_{k+1}}{g_k^T g_k}$

→ need k to compute N. matrices

How? $Qx^* = b \quad x^* = \alpha_i d_i \Rightarrow d_i^T Q x^* = d_i^T Q \alpha_j d_j = \alpha_i d_i^T Q d_i$

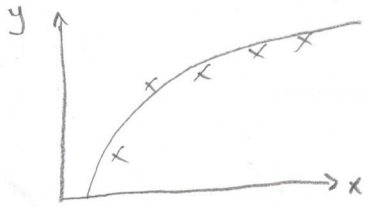
So $\alpha_i = \frac{d_i^T b}{d_i^T Q d_i}$ → Once we know d_i & d_i, can update any $x^0 \to x^*$ by adding $\alpha_k d_k$ n times (provable)

less clear how to implement for non-linear $J(\vec{\theta})$

So use QUASI-NEWTON METHODS → don't compute Hessian matrix, just approximate to cut down on time

example: (L)-BFGS $H_k^{-1} = \left(I - \frac{s y^T}{y^T s}\right) H_{k-1}^{-1} \left(I - \frac{y s^T}{y^T s}\right) + \frac{s s^T}{y^T s}$ $\quad s = x^k - x^{k-1}$ $\quad y = \nabla f(x^k) - \nabla f(x^{k-1})$ $\quad \mathcal{O}(n^2)$

LIN REG



hypothesis: $h_\theta(x)$ → tells most probable value for predicting future outputs

→ $x, \theta, h \in \mathbb{R}$

$\vec{h_\theta(x)} = X\vec{\theta}$ $\quad (h_\theta(x^{(i)}) = \theta_j x_j^{(i)})$

$h_\theta(x)$ directly predicts y

No equivalent to a DB.

COST FUNCTION: $J(\theta) = \frac{1}{m}\sum_{i=1}^{n} \overbrace{\frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2}^{\text{COST}}$

$= \frac{1}{2m}(x\theta - y)^T(x\theta - y)$

$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$ // $\vec{\theta} \to \theta - \alpha\delta$

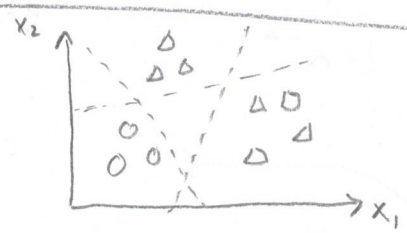
GRADIENT DESCENT $\quad \partial_{\theta_j} J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta^{(i)} - y^{(i)})\, x_j^{(i)}$

Vectorized: $\delta = \frac{1}{m} X^T(\vec{h_\theta} - \vec{y})$

Use feature scaling → $x_j \to \frac{x_j - \langle x_j \rangle}{\sigma_j}$

y = continuous, no need to deal w/ multiple features

LOG REG

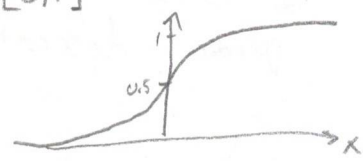


$h_\theta(x)$ → estimated probability that $y=1$

→ $P(y=1 \mid x; \theta)$

$x, \theta \in \mathbb{R}$, $\quad h \in [0,1]$

$\vec{h_\theta}(x) = \underbrace{\frac{1}{1+e^{-x\vec{\theta}}}}_{\text{sigmoid function}}$

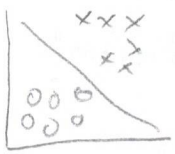


predict $y=1$ for $h_\theta \geq 0.5$

DECISION BOUNDARY: $\boxed{h_\theta = 0.5 \iff \vec{\theta}\cdot\vec{x} = 0}$

$(1, x_1, x_2, x_3 \ldots)$

LINEAR



$h(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$

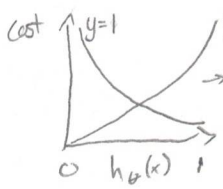
OR

NON LINEAR

$h(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x$

$J(\theta) = \frac{1}{m}\sum \text{costs}$

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



$= -y\log(h_\theta(x)) - (1-y)\log(1 - h_\theta(x))$

$J(\theta) = -\frac{1}{m}\left(\vec{y}\cdot\log(\vec{h}) + (1-\vec{y})\cdot\log(1-\vec{h})\right)$

→ IDENTICAL! (if you work through derivation)

→ STILL USE FEATURE SCALING

MULTICLASS CLASSIFICATION: ONE-VS-ALL

→ do Log. Reg for each class

$h_\theta^{(1)} = P(y=\square \mid x;\theta)$

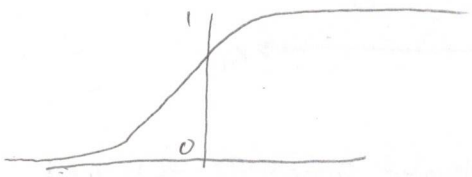
$h_\theta^{(2)} = P(y=\circ \mid x;\theta)$

$h_\theta^{(3)} = P(y=\triangle \mid x;\theta)$

} Predict: $\max_i (h_\theta^{(i)})$ class

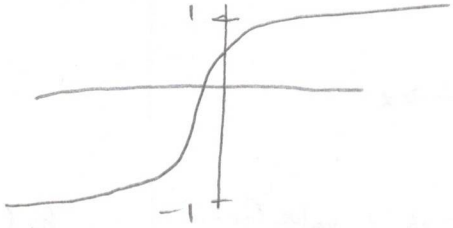
Note on other $h(\theta)$ commonly used:

sigmoid: $f(z) = \frac{1}{1+e^{-z}}$



- historically of interest but slow (like gradient descent)

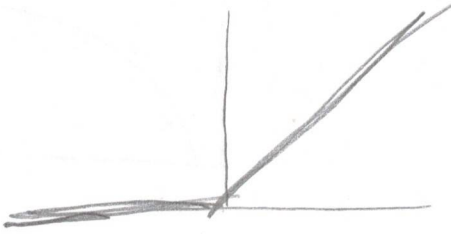
tanh: $f(z) = \tanh(z)$



- much steeper gradient than sigmoid, converges faster

rectified linear unit

ReLU: $f(x) = \max(0, x)$



- helps model interaction effects
- helps (more) w/ nonlinear effects

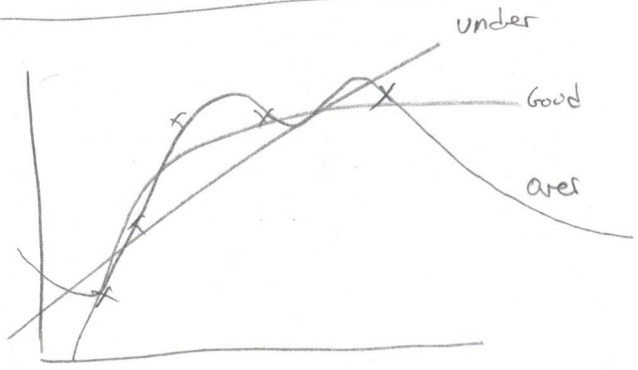
most commonly used today?

problem:

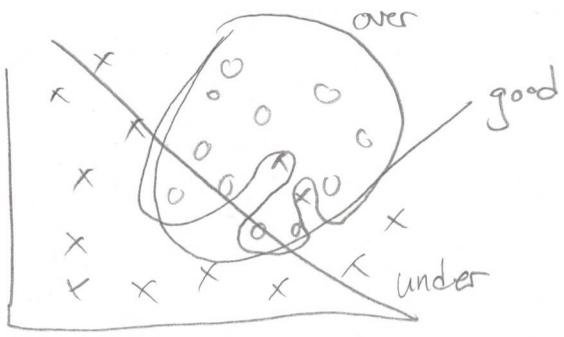
have steep ∇ in small region only, difficult to improve weights in gradient descent

→ "VANISHING GRADIENT PROBLEM"

LIN REG



LOG REG



underfit: High BIAS (user bias as to what features/physics to include)
overfit: High VARIANCE (lots of wiggles)

REGULARIZATION : penalize parameters to keep them small

eg model $h = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

forcing these to be small, will look more like quadratic "good fit"

Implementation:

$$J(\theta) \rightarrow J(\theta) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

→ typically don't include θ_0!

Update becomes: $\theta_j := \theta_j\left(1 - \frac{\alpha\lambda}{m}\right) + \alpha \frac{\partial J}{\partial \theta_j}$ (except for $j=0$, no change)

ie new gradient: $\frac{\partial J}{\partial \theta_j} = \text{old}\ \frac{\partial J}{\partial \theta_j} + \frac{\lambda}{m}\theta_j$

Analytic Solⁿ For Normal Equation (Lin. Reg.)

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

Always Invertible!

PROBLEM: Combining features into various polynomials → quadratic in features, $n \to \Theta(\frac{n^2}{2})$
(w/ Lin/Log Reg) grows n extremely fast.

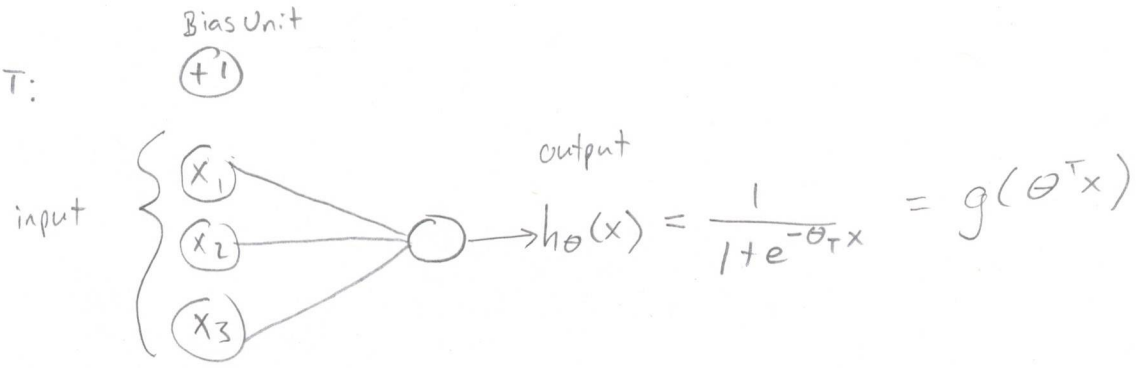
Becomes computationally expensive.

Eg: 50x50 pixel grid, computer vision → over 3 million quadratic terms!

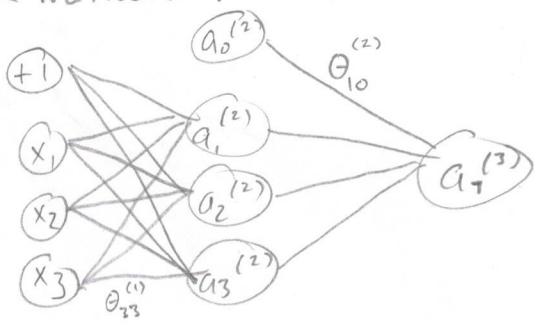
SOLUTION: Use Neural Networks

Bottom Line → hidden layers allow for more simple development of complex features

LOGISTIC UNIT:



NEURAL NETWORK:



(also $a^{(1)}$)
Layer 1 | Layer 2 (Hidden Layer) | Layer 3 (output Layer)

Definitions:
$a_i^{(j)}$ = Activation Unit in Layer j
s_j = # units in layer (j) (not w/ bias)
$\Theta^{(j)} \in \mathbb{R}^{(s_{j+1}) \times (s_j + 1)}$ = parameters/weights maps from $j \to j+1$

$\Theta_{i;j}$
↙ new layer ↘ old feature

So schematically: $a_i^{(j+1)} = g(\underbrace{\Theta_{i0}^{(j)} a_0^{(j)} + \Theta_{i1}^{(j)} a_1^{(j)} + \dots + \Theta_{is}^{(j)} a_{s_j}^{(j)}}_{z_i^{(j+1)}}) = g(z_i^{(j+1)})$

VECTORIZATION:

$$z^{(j)} = \Theta^{(j-1)} a^{(j-1)}$$
$$a^{(j)} = g(z^{(j)}) \quad (+\ a^{(0)} = 1)$$

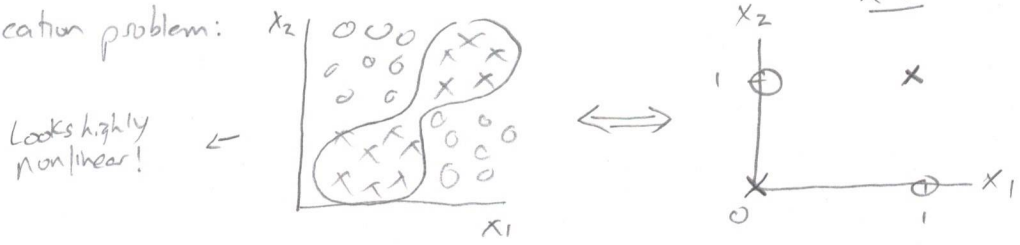
FORWARD PROPAGATION

Common Logical Operators:

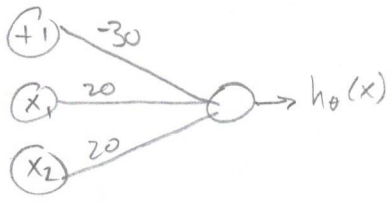
X_1	X_2	OR	AND	NAND	NOR	XOR	XNOR
0	0	0	0	1	1	0	1
0	1	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	1	1	1	0	0	0	1

X	NOT
0	1
1	0

Similar to possible classification problem:



Start Simple: How do we get $y = x_1$ AND x_2?

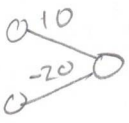


X_1	X_2	$h_\theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

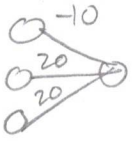
} exactly matches and fxn

Others?

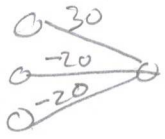
NOT:



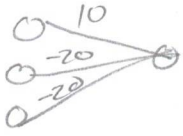
OR:



NAND:



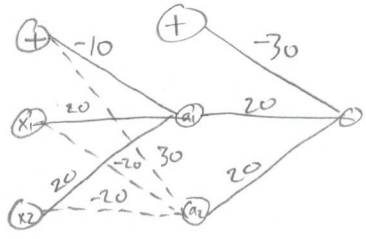
NOR:



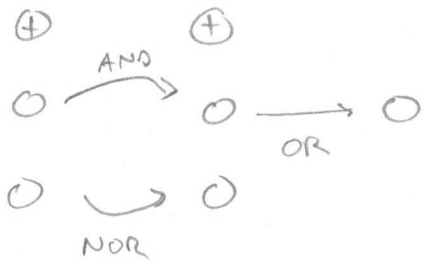
More complicated? XOR = (x_1 OR x_2) AND (x_1 NAND x_2) (same t.t.)

So build out of simpler:

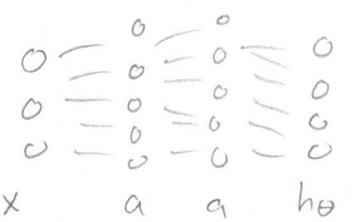
XOR:



XNOR:



MULTICLASS? Again, if y = multiple outputs (0-9, types of cars, etc...) → make h_θ vector



$h_\theta \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ for option 1

INPUT: $\left[(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)}) \right]$

$x^{(i)} \in \mathbb{R}^n$ $\quad y^{(i)} \in \mathbb{R}^{\#\text{ classes}}$

n = # input features $\quad m$ = # examples