→ Perceptron is a linear classifier

→ The w_s will be the weights (θs earlier).
w_0 is the intercept $w_0 = 1$ initially.

A perceptron calculates (2) quantities { (1) a weighted sum of the input features; (2) a threshold on this sum by the (1) function.

The perceptron is a (simple) artificial model of human neurons

Weights → synapses

threshold → neuron firing

Perceptron operation:

$$\hat{f}(x_1, \dots x_n) = \begin{cases} 1 & \text{if } \theta_1 x_1 + \dots + \theta_n x_n > 0 \\ 0 & \text{otherwise (can also be defined as } -1) \end{cases}$$

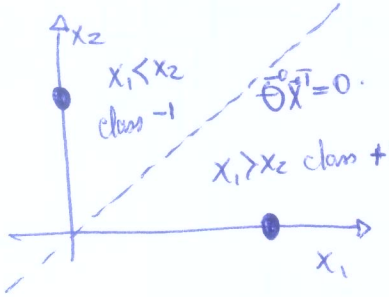
$\vec{\theta}$ → weight vector $\in \mathbb{R}^{n+1}$

$\vec{x}$ → feature vector $\in \mathbb{R}^{n+1}$.

$\theta_0 x_0 + \theta_1 x_1 + \dots \theta_n x_n = \underbrace{\vec{\theta}\vec{x}^T}$
vector inner product

{ The perceptron represents a hyperplane decision surface in n-dimensional space { 2d line, 3d plan
The equation of the hyperplane is $\vec{\theta}\vec{x}^T = 0$
This is the equation for points in X-space that are on the boundary.

Example



Suppose $\vec{\theta} = (\theta_1 = 1, \theta_2 = -1, \theta_0 = 0)$.

$\vec{\theta}\vec{x}^T \Rightarrow 1x_1 + (-1 \cdot x_2) + 0(x_0) = 0$

$x_1 - x_2 = 0$

$\boxed{x_1 = x_2}$ decision boundary equation

* A data set is ~~linearly~~ **separable** by a learner (if) there is some instance of that learner that correctly predicts (all) data points

* The learner is **linearly separable** if:
  - Can separate the two classes using a straight line if $X \in \mathbb{R}^2$ or with a hyperplane if $X \in \mathbb{R}^n$.

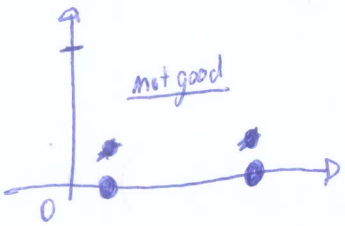
Class overlap ~~Concept~~
- Common in practice
- some observation values possible for both classes (benign/malign cells look similar).

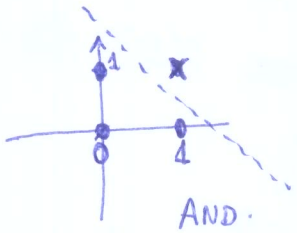
what to do?
- Increase number of features.
- Increase model complexity.

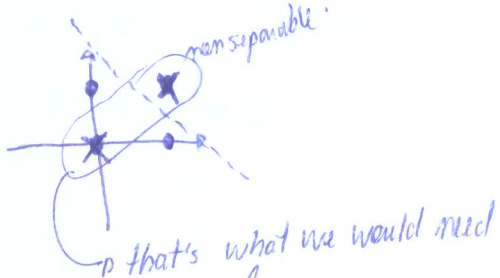
Representational power of perceptrons

↝ what mappings a perceptron can represent perfectly?
well, it is a linear classifiers, so it can represent any mapping that is **linearly separable**

Example: some boolean functions (AND) but no **XOR**







Remember Effects of dimensionality

(1) Data are increasingly separable in high dimension – Is this a good thing?

Good
- Separation is easier in higher dimensions (for fixed (M) Co nbr examples.
- Increase nbr of features, and even a linear classifier would do the trick!

Bad
- training (vs) test error; overfitting; bias (vs) variance
- Increasingly complex decision boundaries can eventually get all training data right but it doesn't mean necessarily good for testing data ⇒ **lacks generalization**

How to update weights?

for each data point $x^{(j)}$:

$$f(x^{(j)}) \leftarrow T(\vec{w} * x^{(j)})$$

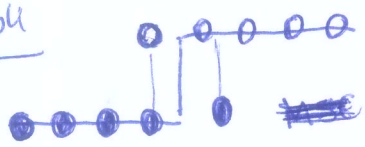
$$\vec{w} \leftarrow w + \alpha \left(f(x^{(j)}) - y^{(j)} \right) \cdot x^{(j)} \quad \text{gradient-like step}$$

- if $f(x^{(j)})$ correctly predicted: no update.
- otherwise: update weights.

Another way of thresholding would be using smooth functions

- if we are far from decision boundary, $|f(.)|$ is large, small error.
- nearby the boundary: $(f(.))$ near $1/2$, large error.

Example



$J(\theta) = \cdot = 2/10$ normal thresholding.

while

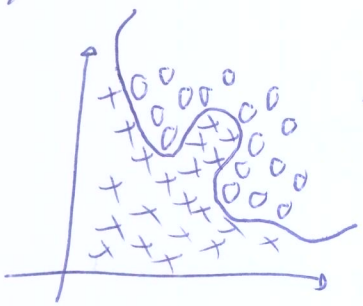
$J(\theta) = (0^2 + 1^2 + .2^2 + .95^2 + \dots)/10.$

Summary
- linear classifier $\Leftrightarrow$ perceptron
- visualizing the decision boundary: useful but not always possible
- measuring quality: sum of squared errors (SSE).
- Learning the weights of a linear classifier reduces to an optimization problem. For (SSE) we can do gradient descent but there are others.

Neural Networks – Non-linear hypotheses

- Quite old idea
- Out of favor for sometime but in the hype again for several recent results such as Deep Belief Networks (Deep learning) and results such as image nets
- Why do we need yet another learning algorithm?

Consider a problem



→ we could use LogReg such as

$$\hat{f}(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1 x_2^2 + \dots)$$

poly terms

→ normally it works OK for a few features (2,3) but not as good for many features (how to derive polynomials then?).

If we have 100 features and want to just include 2nd order polynomials, we would have

$$\hat{f}(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_{100} x_{100} + \theta_{101} x_1^2 + \theta_{102} x_1 x_2, \dots, \theta_K x_1 x_{100} + \theta_{K+1} x_2 x_3 + \dots) \approx 5K \text{ features. } \theta(n^2) \approx \frac{n^2}{2}.$$

orig. feature space

→ If we want 3rd order polys, we would have $\theta(n^3)$.
For 100 features ≈ 170K features!

→ In practice, we have problems with thousands or millions of features in the original feature space.

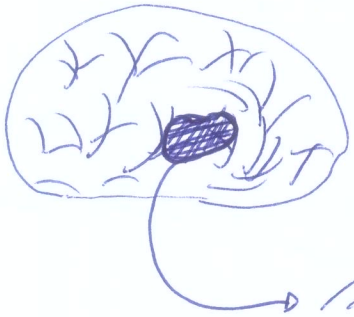
Neurons and the brain

Origins:
- algorithms that try to mimic the brain.
- widely used in the 80's and early 90s. Popularity went down in late 90s.
- Recent hype due to state-of-the-art results in many applications.
- the reason is that now we have enough computational power to design and use large-scale neural networks

expensive to run

The "one learning algorithm" hypothesis

→ Some people believe the brain does all of its amazing things based on one learning algorithm.



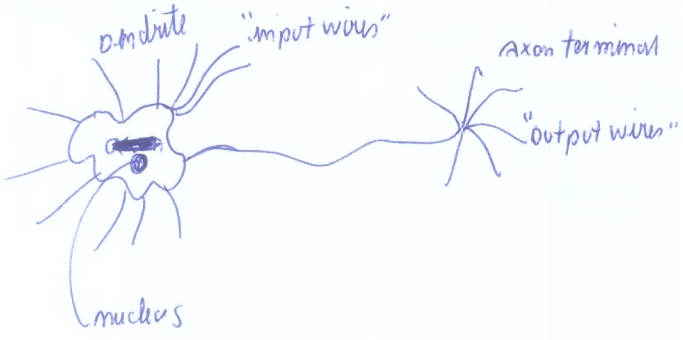
→ Auditory cortex learns to hear/listen
but researchers have shown that cutting the connection of Auditory cortex to the ear and rewiring it to the eye leads to a learning process in which the auditory cortex learns to see!

neuro-rewiring experiments.

Examples: - Brainport experiment: Camera for low-res grayscale image on top of the head and a wire to connect each pixel to our tongue (low voltage → dark pixel, high voltage → bright pixel) and we can "see" through our tongue.

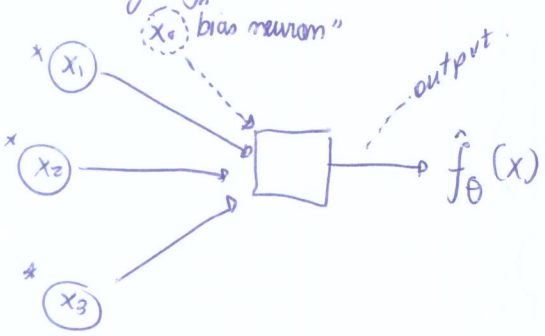
- Human echolocation (sonar): snap fingers. Boy that had his eyeballs removed learned to navigate using this "rewiring" technique.
- Implanting a 3rd eye in a frog will lead him to actually learn how to use it.

Neural Networks - Representation



Computationally speaking, we can think of a neuron as a computational unit that receives a set of inputs, performs some computation and outputs some signals through the axons to other neurons.

We are going to model this as a logistic unit



$$\hat{f}_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

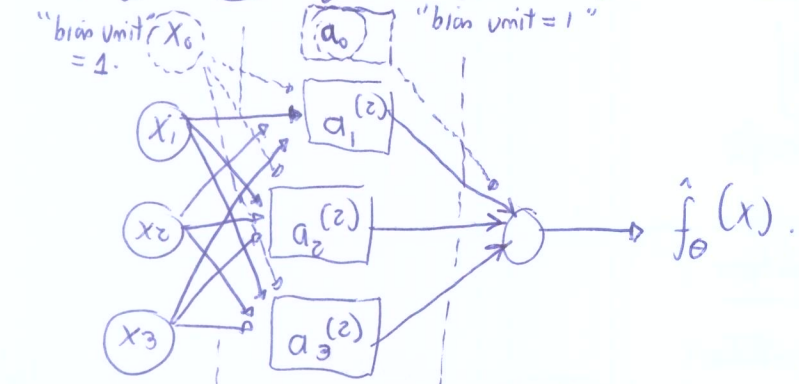
$$\vec{x} = \begin{bmatrix} x_0 \\ \vdots \\ x_m \end{bmatrix} \quad \vec{\theta} = \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_m \end{bmatrix}.$$

$* \Rightarrow$ inputs.

$x_0 \Rightarrow$ bias neuron always equal to (1).

In (NN) terminology the sigmoid/logistic activation function and the $\vec{\theta}$ parameters are known as weights. (activation function represents)

Generalizing, a (NN) is just a group of several neurons.



input layer. hidden layer. output layer.

because it is not (x) or (y) (input/output) therefore the term "hidden".

↝ We may have (NN's) with several hidden layers.

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

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

Here's the computation:

$$a_1^{(2)} = g\left(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3\right)$$

$$a_2^{(2)} = g\left(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3\right).$$

$$a_3^{(3)} = g\left(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3\right).$$

$$h_\Theta(x) = a_1^{(3)} = g\left(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_3^{(2)} a_3^{(2)}\right).$$

$\Theta^{(1)}$ is a matrix mapping layer (1) to (2), so $\Theta^{(1)} \in \mathbb{R}^{3\times 4}$.

↝ If network has (s_j) units in layer (j), and (s_{j+1}) in layer $(j+1)$, then $\Theta^{(j)}$ will have dimensions $(s_{j+1}) \times (s_j + 1)$.

↝ with this, we have $\hat{f}_\Theta(x)$ (h) our classifier.

---

model Representation, intuition and calculations

forward propagation: vectorized implementation

$$a_1^{(2)} = g\left(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3\right)$$

$$a_2^{(2)} = g\left(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3\right)$$

$$a_3^{(2)} = g\left(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3\right)$$

let's call $z_1^{(2)}$, therefore $a_1^{(2)} = g(z_1^{(2)})$.

$z_2^{(2)}$

$z_3^{(2)}$.

$\Theta^{(1)} \cdot \vec{x}$ or $\Theta^{(1)} x$

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_3 \end{bmatrix} \quad ; \quad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix} \quad ;$$

$$z^{(2)} = \vec{\Theta}^{(1)} \vec{x}$$

$$a^{(2)} = g(z^{(2)})$$

$\mathbb{R}^3$ $\quad$ $\mathbb{R}^3$

(g) function applies sigmoid function on each $z_i^{(i)}$ element-wise. Given all elements in $z^{(i)}$ (e.g. in $\mathbb{R}^3$), we apply (g) on each individually.

Let's define $a^{(1)} = \vec{x}$, therefore $z^{(2)} = \vec{\Theta}^{(1)} \cdot \vec{x}$ now becomes $\boxed{z^{(2)} = \vec{\Theta}^{(1)} \cdot a^{(1)}}$

Now we add $\boxed{a_0^{(2)} = 1}$ so that $a^{(2)} \in \mathbb{R}^4$
bias unit

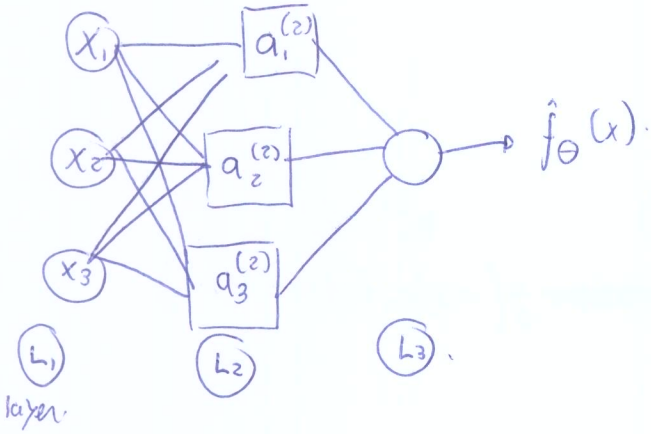
and

$$z^{(3)} = \vec{\Theta}^{(2)} \cdot a^{(2)}$$

$$h_\Theta(x) = \hat{f}_\Theta(x) = a^{(3)} = g(z^{(3)})$$

This algorithm is known as forward propagation.

Neural network learning its own features



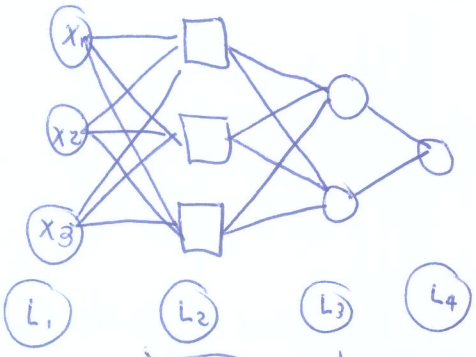
* Suppose we don't see the left part of it. If we do that, we end up having a logistic Regression like situation

$$\hat{f}_\Theta(x) = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

The crucial thing here is the features (a_1) are learned from the input.

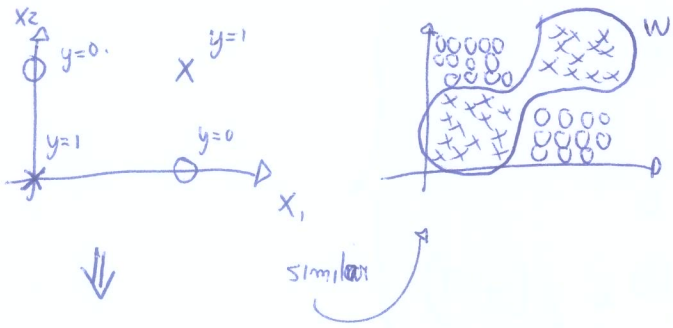
Observation: we can have other (NN) diagrams/architectures

↳ how (NN) units connect to each other.



$\hat{f}_\Theta(x)$.

Let's return to logical operations example in which x_1 and x_2 are binary (0 or 1).



we want to learn a non-linear separator/classifier

$y = x_1$ XOR x_2

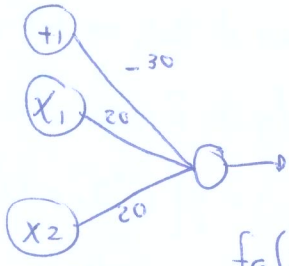
$y =$ XNOR x_2

Not $(x_1$ XOR $x_2)$.

Let's start with a simple example for logical (AND).

$x_1, x_2 \in \{0, 1\}$

$y = x_1$ AND x_2.

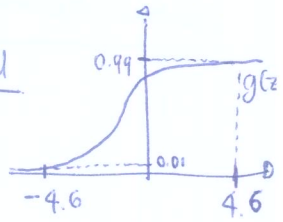


$f_\Theta(x)$.

$\Theta_{10}^{(1)}$ $\Theta_{11}^{(1)}$ $\Theta_{12}^{(1)}$

$f_\Theta(x) =$ ~~sigmoid~~ $g(-30x_0 + 20x_1 + 20x_2)$

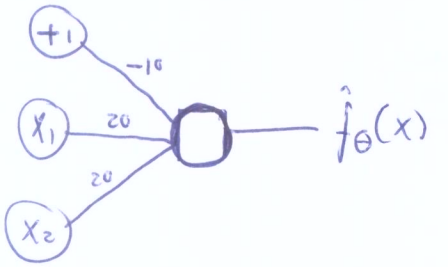
sigmoid



x_1	x_2	$\hat{f}_\Theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$.

$f_\Theta(x) \approx x_1$ AND x_2

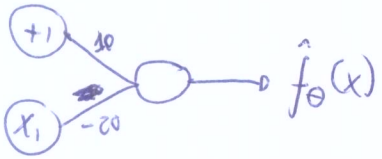
Example for (OR) function:



x_1	x_2	$\hat{f}_\Theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(+10) \approx 1$
1	0	$g(+10) \approx 1$
1	1	$g(30) \approx 1$.

$$\hat{f}_\Theta(x) = g\left(-10x_0 + 20x_1 + 20x_2\right).$$

Example for NOT x_1

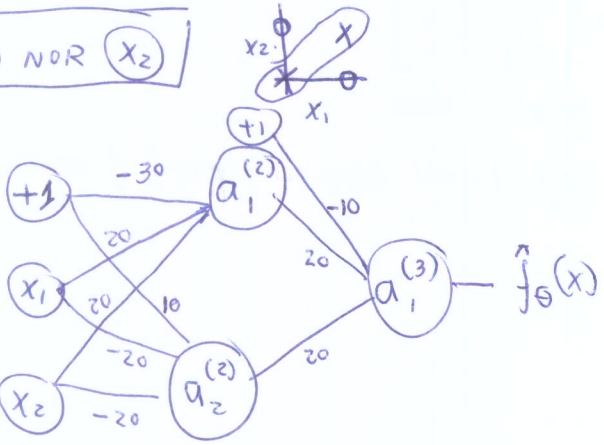


x_1	$f_\Theta(x)$
0	$g(10) \approx 1$.
1	$g(-10) \approx 0$.

$$\hat{f}_\Theta(x) = g(10 - 20x_1)$$

Exercize: how to compute $\sim$ (Not x_1) AND (Not x_2) ?

(x_1) NOR (x_2)



x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$\hat{f}_\Theta(x)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1.

$a_1^{(2)}$ is using x_1 AND x_2 network

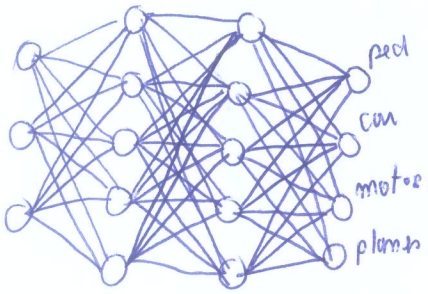
$a_2^{(2)}$ is using $\neg x_1$ and $\neg x_2$ or (not x_1) AND (not x_2).

* Each layer in the sequence can learn more complex functions on top of its predecessors output layers.

Multiclass Classification

Basically, it is an extension of the one (vs) all method.

Pedestrian | Car | motorcycles | Planes



$\hat{f}_\theta(x) \in \mathbb{R}^4$.

We want $\hat{f}_\theta(x) \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ when pedestrian, $\hat{f}_\theta(x) \approx \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$ when car, $\hat{f}_\theta(x) = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$ when motorcycles and $\hat{f}_\theta(x) = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$ when plane.

In the training set, $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}), \dots, (x^{(m)}, y^{(m)})$

$y^{(i)}$ is going to be one of $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

ped car moto plane

↝ previously, we have represented $y \in \{1,2,3,4\}$ (e.g., K-NN). Now we represent it in a binary form. So, for ④ classes,

$$\hat{f}_\theta(x) \approx y^{(i)}$$

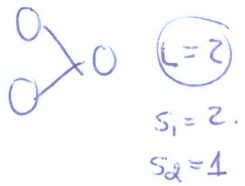
$\hat{f}_\theta(x) \in \mathbb{R}^4$; $y^{(i)} \in \mathbb{R}^4$

x

Cost function for Neural Networks

~ Let's start with the classification problem

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



(L) will be the number of layers in the network.

(s_l) = number of units (excluding the "bias" unit) in layer (l).

In the binary classification, $y=0$ or $y=1$, therefore we will have (1) output unit. For a multiclass classification problem (K classes), $y \in \mathbb{R}^K$, therefore we will have (K) output units.

Let's now define a **cost function**. Recall that for logistic regression, we had:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log \hat{f}_\theta(x^{(i)}) + (1-y^{(i)}) \cdot \log\left(1-\hat{f}_\theta(x^{(i)})\right)\right] + \underbrace{\frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2}_{\text{Regularization } \theta_0 = 1}$$

for a neural network, instead of having just one logistic regression output unit, we will have (K) of them.

$$\hat{f}_\theta(x) \in \mathbb{R}^K \quad (\hat{f}_\theta(x))_i = (i^{th}) \text{ output}.$$

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{f}_\theta(x^{(i)})\right)_k + (1-y_k^{(i)}) \cdot \log\left(1-\hat{f}_\theta(x^{(i)})\right)_k\right] +$$

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

Wow! How can we optimize such cost function to actually find our weights or parameters?

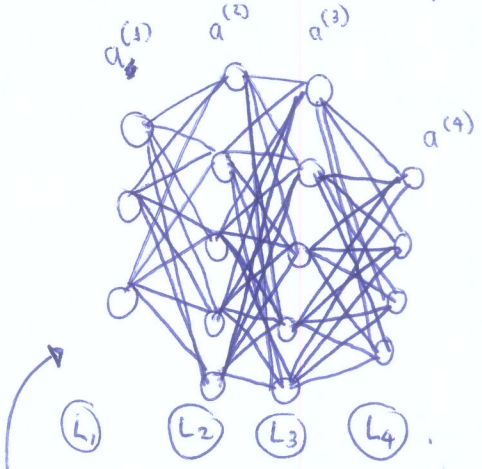
Back propagation Algorithm

Given our $J(\Theta)$, we want to $\min_{\Theta} J(\Theta)$. For that we need to compute:

1. $J(\Theta)$
2. $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$. ; recall $\Theta_{ij}^{(l)} \in \mathbb{R}$.

Suppose we only have one training example (x, y). Using the forward propagation, we have

$$a^{(1)} = x.$$
$$z^{(2)} = \Theta^{(1)} \cdot a^{(1)}$$
$$a^{(2)} = g(z^{(2)}) \quad \text{add } a_0^{(2)}$$
$$z^{(3)} = \Theta^{(2)} \cdot a^{(2)}$$
$$a^{(3)} = g(z^{(3)}) \quad \text{add } a_0^{(3)}$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g(z^{(4)}).$$



Gradient computation: Back propagation algorithm

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

$a_j^{(l)} \Rightarrow$ activation of jth unit in layer l.

Concretly, if we have the NN above, for each output unit (layer $L=4$),

$$\delta_j^{(4)} = \boxed{a_j^{(4)}} - y_j \qquad \left(= h_\Theta(x)_j\right)$$

if we vectorize it, we have $\vec{\delta}^{(4)} = \vec{a}^{(4)} - \vec{y}$.

$\in \mathbb{R}^K$ where K is the number of output units in the NN.

After that, we compute the δ terms for the earlier layers:

$$\delta^{(3)} = \left(\Theta^{(3)}\right)^T \cdot \delta^{(4)} \ .* \ g'(z^{(3)}).$$
$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} \ .* \ g'(z^{(2)}).$$

$.*$ element-wise multiplication.

The derivative $g'(z^{(3)})$ is basically:

$$g'(z^{(3)}) = \underbrace{a^{(3)}}_{\text{vector}} .* \underbrace{(\vec{1} - a^{(3)})}_{\text{vector of ones}}.$$

$$g'(z^{(2)}) = a^{(2)} .* (1 - a^{(2)}).$$

- There is no $\delta^{(1)}$ since they are the features we observe in our training set
- The name **back propagation** comes from the fact we start in the output layer, calculate the error $\delta^{(4)}$ there and go back to the 3rd layer where we compute $\delta^{(3)}$ and go back to layer (2) and calculate $\delta^{(2)}$ in such a way it is propagating the error from layer $(l+1)$ to (l) and from (l) to $(l-1)$ and so on.

Finally $\boxed{\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}}$ when ignoring the regularization term ($\lambda = 0$).

Now, for a larger training set $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, set

$\Delta_{ij}^{(l)} = 0 \quad \forall i, j, l.$

↳ will be used as accumulators for computing $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$.

For $i = 1$ to m { // on each iteration, we work with one example.

- Set $a^{(1)} = x^{(i)}$
- Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \dots, L$.
- Using $y^{(i)}$, compute $\boxed{\delta^{(L)} = a^{(L)} - y^{(i)}}$ // computing error for this example for the output layer.
- Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$ // used to compute the partial derivatives in the end
- $\boxed{\Delta_{ij}^{(l)} \leftarrow \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}}$ → if we vectorize, $\Delta^{(l)} \leftarrow \Delta^{(l)} + \delta^{(l+1)} \cdot (a^{(l)})^T$

}

$D_{ij}^{(l)} \leftarrow \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$

$D_{ij}^{(l)} \leftarrow \frac{1}{m} \cdot \Delta_{ij}^{(l)}$ if $j = 0$. (bias term case)

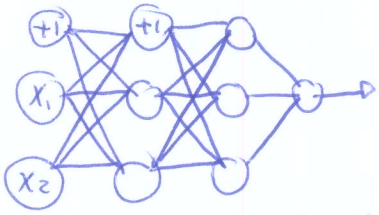
} is exactly $\boxed{\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}}$

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

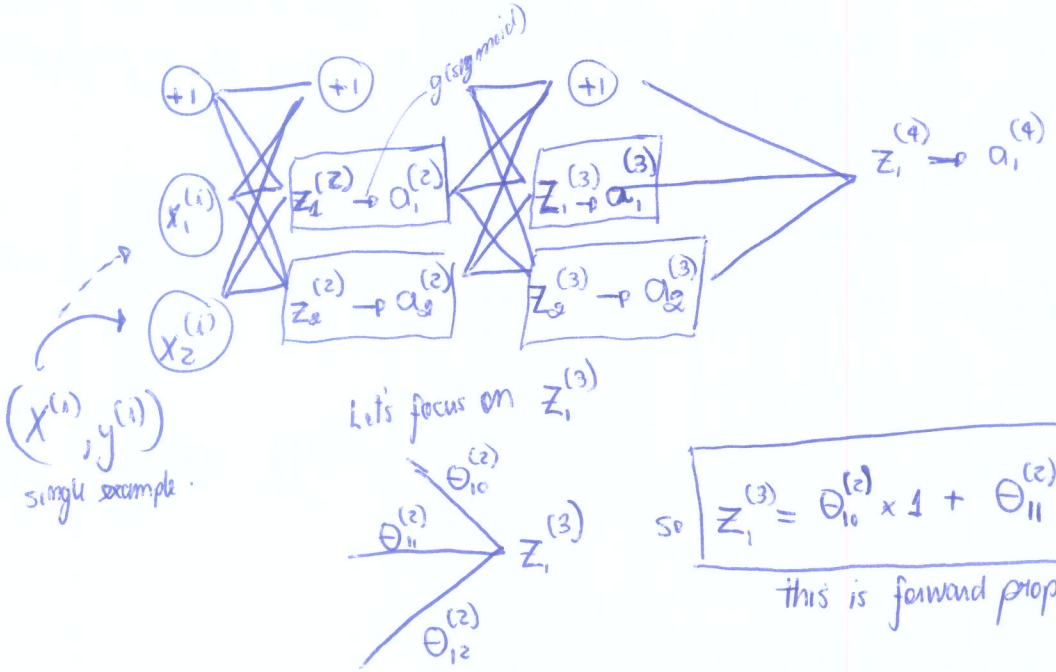
Backpropagation: Intuition

mechanical steps.

Let's see again forward propagation



2 units 2 in 2 hid 1 unit
not counting bias



Let's focus on $z_1^{(3)}$

So $\boxed{z_1^{(3)} = \Theta_{10}^{(2)} \times 1 + \Theta_{11}^{(2)} \cdot a_1^{(2)} + \Theta_{12}^{(2)} a_1^{(2)}}$.

this is forward propagation.

What does Backpropagation do?

Recall its formulation

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log\left(f_\Theta(x^{(i)})\right) + \left(1-y^{(i)}\right) \cdot \log\left(1 - f_\Theta(x^{(i)})\right)\right] + \underbrace{\frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{S_l} \sum_{j=1}^{S_{l+1}} \left(\Theta_{ji}^{(l)}\right)^2}_{\text{Regularization.}}$$

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

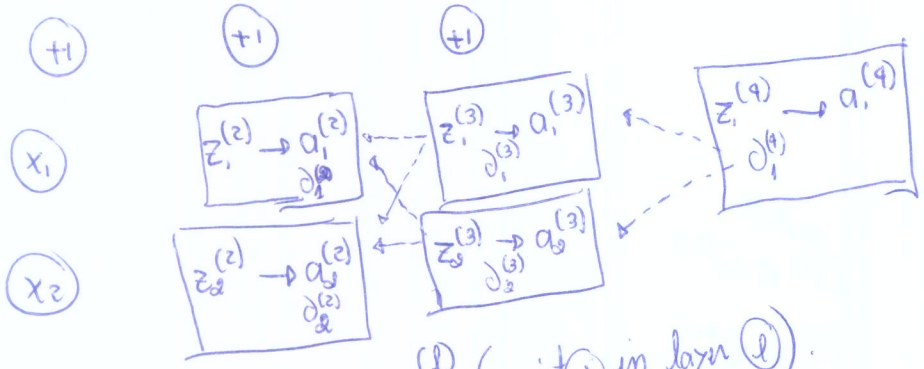
$$\text{Cost}(i) = y^{(i)} \log f_\Theta(x^{(i)}) + (1-y^{(i)}) \log \left(f_\Theta(x^{(i)})\right)^*$$

* $\log\left(f_\Theta(x^{(i)})\right)$. but should be $\log\left(1 - f_\Theta(x^{(i)})\right)$ the correct form is

think of $\text{Cost}(i) \approx \left(f_\Theta(x^{(i)}) - y^{(i)}\right)^2$.

How to the (NN) on example (i)?

Forward Propagation



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

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

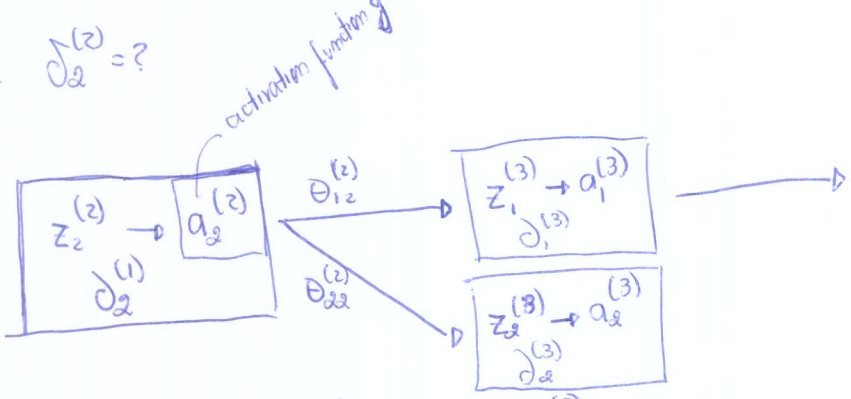
($z_j^{(l)}$ change here affects *)

$$\text{cost}(i) = y^{(i)} \cdot \log \underbrace{f_\theta(x^{(i)})}_{\text{* here}} + \left(1 - y^{(i)}\right) \cdot \log \left(1 - \underbrace{f_\theta(x^{(i)})}_{\text{and * here}}\right).$$

Let's see it:

for $\delta_1^{(4)} = y^{(i)} - a_1^{(4)}$ output.

how to get $\delta_2^{(2)} = ?$



$$\begin{cases} \delta_2^{(2)} = \Theta_{12}^{(2)} \delta_1^{(3)} + \Theta_{22}^{(2)} \cdot \delta_2^{(3)} \\ \text{and} \\ \delta_{(2)}^{(3)} = \Theta_{12}^{(3)} \cdot \delta_1^{(4)} \end{cases}$$

Implementation Note

Example $s_1 = 10$ units

$s_2 = 10$

$s_3 = 1$ output unit

$\Theta^{(1)} \in \mathbb{R}^{10 \times 11}$ $\quad D^{(1)}$ (derivatives) $\in \mathbb{R}^{10 \times 11}$

$\Theta^{(2)} \in \mathbb{R}^{10 \times 11}$ $\quad D^{(2)} \in \mathbb{R}^{10 \times 11}$

$\Theta^{(3)} \in \mathbb{R}^{1 \times 11}$ $\quad D^{(3)} \in \mathbb{R}^{10 \times 11}$.

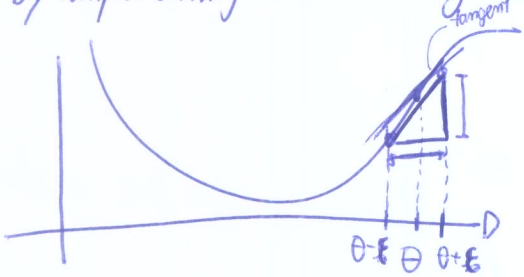
If we want to use this with more advanced ~~#~~ techniques of optimization, we may need to deal with them as ~~matrices~~ vectors instead of as matrices.

E.g., when optimizing in Octave.

Gradient Checking

How can we check the Backpropagation algorithm along with its optimization function for finding the parameters (e.g., gradient descent) are actually working?

~ We start by implementing numerical gradient checking.



$\theta \in \mathbb{R}$.

The slope of the line $\left(J(\theta+\varepsilon), (\theta+\varepsilon); J(\theta-\varepsilon), \theta-\varepsilon\right)$ is an approx for the derivative of $J(\theta)$.

$$\text{slope} = \frac{\text{vertical height}}{\text{horizontal width}} = \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}.$$

Therefore $\frac{\partial}{\partial\theta} J(\theta) \approx \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$ } two sided difference.

$\varepsilon = 10^{-4}$ (the smaller the ε, the closer to the actual derivative.

In a more general case, $\theta \in \mathbb{R}^n$ (e.g., θ is "unrolled" version of $\theta^{(1)}, \theta^{(2)}, \theta^{(3)}$)

unrolled matrix 2×2 $\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \\ a \end{bmatrix}$.

$\theta = [\theta_1, \theta_2, \cdots, \theta_n]^T$.

we can do:

$$\frac{\partial}{\partial\theta_1} J(\theta) \approx \frac{J(\theta_1+\varepsilon, \theta_2, \theta_3, \cdots \theta_n) - J(\theta_1-\varepsilon, \theta_2, \cdots, \theta_n)}{2\varepsilon}.$$

$$\frac{\partial}{\partial\theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2+\varepsilon, \theta_3, \cdots \theta_n) - J(\theta_1, \theta_2-\varepsilon, \theta_3, \cdots \theta_n)}{2\varepsilon}.$$

$$\frac{\partial}{\partial\theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \cdots \theta_n+\varepsilon) - J(\theta_1, \theta_2, \cdots, \theta_n-\varepsilon)}{2\varepsilon}.$$

These equations allow us to numerically estimate the derivatives with respect to any parameter θ_i and then can be used to double check the derivatives calculated by backpropagation.

~ Backpropagation is very efficient compared to gradient numerical checking. Therefore we should use it just in one or a few iterations to ~~check~~ double check backpropagation results.

Random Initialization

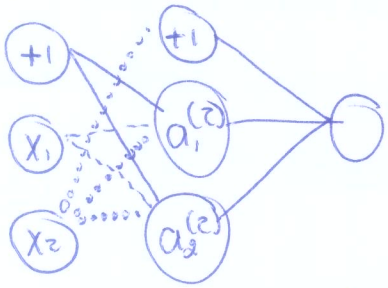
We need to pick some initial value for Θ.

Initializing Θ with zeros worked OK for logistic regression but is doesn't for (NNs).

The reason is that the network will be redundant.

$a_1^{(2)} = a_2^{(2)}$ and $\delta_1^{(2)} = \delta_2^{(2)}$.

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

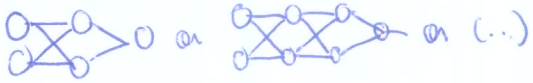


$\boxed{a_1^{(2)} = a_2^{(2)}}$ if $\Theta = 0$ for all elements in the beggining

This is known as the problem of symmetric weights. For breaking the symmetry, we must initialize $\Theta_{ij}^{(l)}$ to a random value in $[-\varepsilon, \varepsilon]$ (this means $-\varepsilon \leq \Theta_{ij}^{(l)} \leq \varepsilon$)

(This is not ε of before)

Wrapping up The first thing we need to do when training a (NN), is to select a (NN) architecture (connectivity pattern)



How to choose this?

1. input units: dimensionality of the problem $x^{(i)}$.
2. output units: number of classes.

③ for the internal (hidden layers), a reasonable default is [One] hidden layer. But if we are using more than one hidden layer; it is interesting to have the same number of units in all layers (the more the hidden layers, the better, normally).

Note: } the more hidden layers, the better at the cost of much more computation.
nbr of units in hidden layer normally is comparable to that in the input layer.

Training a neural network

① Initialize weights randomly.

② Implement forward propagation to obtain $f_\Theta(x^{(i)})$ $\forall\, x^{(i)}$.

③ Compute cost function $J(\Theta)$.

④ Backpropagation to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$.

Example
for $i=1$ to m } nbr examples // $x^{(1)}, y^{(1)}; \dots; x^{(m)}, y^{(m)}$
  Perform forward prop and backprop using $x^{(i)}, y^{(i)}$
  (Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l=2, \dots, L$).
  $\Delta^{(l)} \leftarrow \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$. // accumulators having, in the end, the partial derivatives
  // $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$
}
Compute $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

Batch

⑤ Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed in ④ with a numerical estimate of gradient of $J(\Theta)$.

⑥ Disable gradient checking.

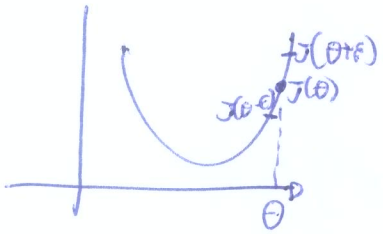
⑦ Use Gradient Descent or other opt method with backprop to minimize $J(\Theta)$ as a function of params Θ.

} ⚠ we have $J(\Theta)$ as a non-convex function, we may converge to local minima.

Backprop → calculates the derivatives (direction of a step) the gradient.
Gradient Descent → take little baby steps downhill.

Last notes on Gradient Checking

- Always check the first results of Gradient Descent using gradient checking.



- Gradient checking is too slow. Do not use it for learning, just to check if it works in the first iteration.
