NEURAL NETWORKS: REPRESENTATION

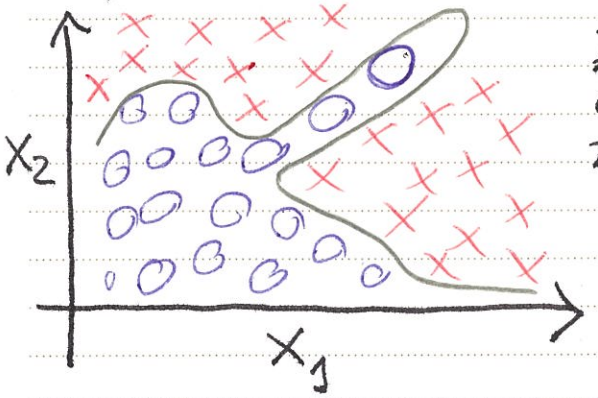
Lecture 44: Non-linear hypotheses

Neural networks are a relatively old idea, but then fell out of favor for a while. Today, however, it's the state of the art technique for many ML problems.

Why do we need yet another learning algorithm though? We already have linear and logistic regression, so why do we need more?

→ Because of complex non-linear hypotheses.

Imagine we have a classification problem as follows:



If we use logistic regression, we would have to include many terms to capture this non-linearity:

$$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^3 x_2 + \theta_6 x_1 x_2^2 + \dots)$$

→ will need to fit for a lot of θs.

This gets much worse if we have more features. Imagine we have 100 features, for instance, for the problem of predicting whether or not a house is likely to sell in the next six months or so:

$$\left.\begin{array}{l} x_1 = \text{size} \\ x_2 = \#\text{ bedrooms} \\ x_3 = \#\text{ floors} \\ x_4 = \text{age} \\ \cdots \\ x_{100} \end{array}\right\} \quad n = 100.$$

In general, if we have n features & we want to keep up to K orders, we will need to fit:

$$\#\text{ of terms} = \sum_{i=0}^{K} \binom{i+n-1}{i} = \binom{n+K}{K}$$

Check: $n = 2$

$K = 1 \longrightarrow 1 + \binom{2}{1} = 3$ ✓

$K = 2 \longrightarrow 1 + \binom{2}{1} + \binom{3}{2} = 6$ ✓

For instance, for $n = 100$, if we want to keep up to 3rd order terms, we'd need 176851 features!

So it seems like when n is large & we're trying to fit a non-linear model, linear & logistic regression won't be feasable. In many ML problems n will be large.

Consider the following problem in image recognition: our classification algorithm determines whether an image contains a car or not. Even if we only look at very small images, say 50×50 pixels, we would have at least 2500 features. (That's if we use gray scale. RGB would need 7500 features.)

$$x = \begin{bmatrix} \text{Pixel 1 intensity} \\ \text{pixel 2 intensity} \\ \vdots \\ \text{pixel 2500 intensity} \end{bmatrix} \rightarrow \text{Every value between } 0\text{–}255$$

If we decide to only keep up to quadratic terms, we would need to fit for 3,128,751 values! This is just far too large to be reasonable.

Lecture 45: Neurons and the Brain

The neural networks algorithm was motivated by the goal of having machines that mimic the brain. They were widely used in 80s and early 90s, but their popularity diminished in the late 90s. They have made a come-back recently, and are the state-of-the-art technique for many applications. Part of the reason for this recent resurgence is that Neural Networks are computationally intensive and only recently computers have gotten fast enough to run large-scale Neural Networks.

The brain does so many remarkable things: it can learn to see, hear, process the sense of touch, do math, etc. It's natural to conclude that the brain has many different "learning algorithms" for each of these tasks. There's a hypothesis, however, which states that the brain does all of this with one generic learning algorithm instead. This is called the "one learning algorithm" hypothesis.

Some evidence for the "One learning algorithm" hypothesis:

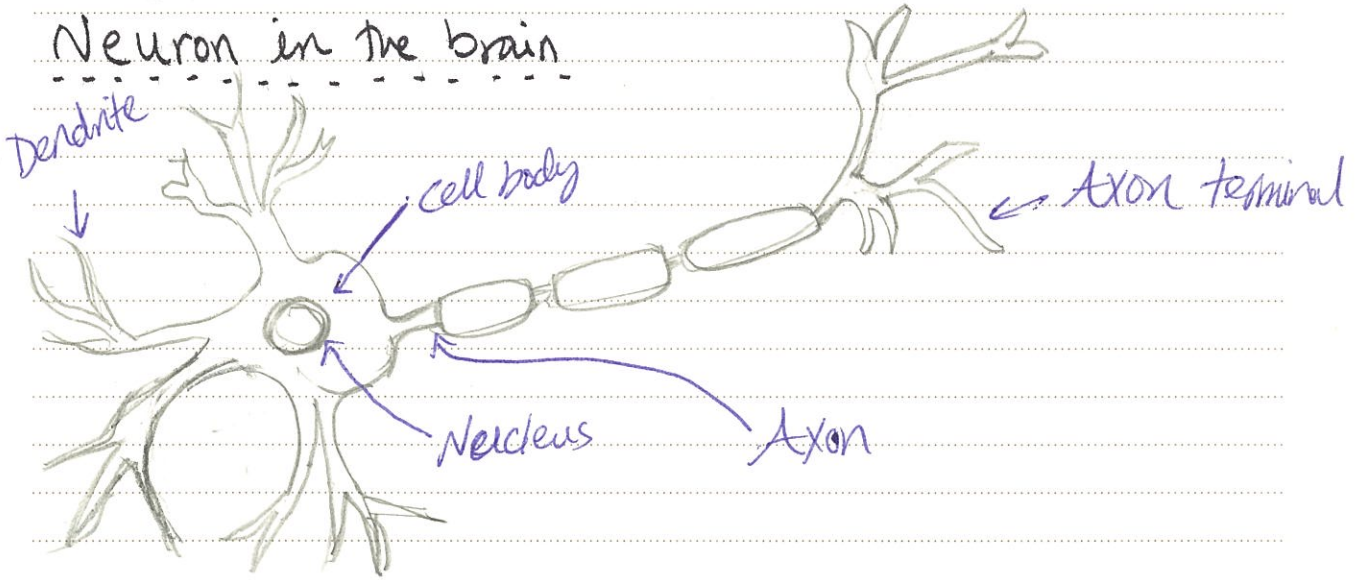
* The Auditory Cortex is responsible for interpreting sound. There have been experiments on animals where the signal from the ears to the Auditory Cortex has been cut, and instead the signal from eyes are sent there. (These types of experiments are called Neural Rewiring experiments.) Apparently, the Auditory Cortex can learn to see once this is done! [Reference: Roe et al, 1992]
* The Somatosensory Cortex is responsible for interpreting the sense of touch. Apparently the same neural rewiring experiment as above has shown that the Somatosensory Cortex can learn to see. [Reference: Metin & Frost, 1989]
* BrainPort: an actual commercial product that helps blind people see using their tongues! It includes a video camera mounted on a pair of sunglasses.

* BrainPort continued: The camera is connected to an array of electrodes, which are placed on the tongue. White pixels from the camera are felt as strong simulation, and black pixels as no simulation. Researchers have found that shortly after using the device, most individuals can begin interpreting spatial information. Within an hour, most users can point to different shapes. After a few hours, they can identify familiar objects and avoid obstacles.

It seems like you can just plug in any kind of sensor into the brain, and the brain will learn to interpret that data and learn from it. Maybe if we can understand this algorithm and implement it, we'll have a chance at achieving the AI dream.

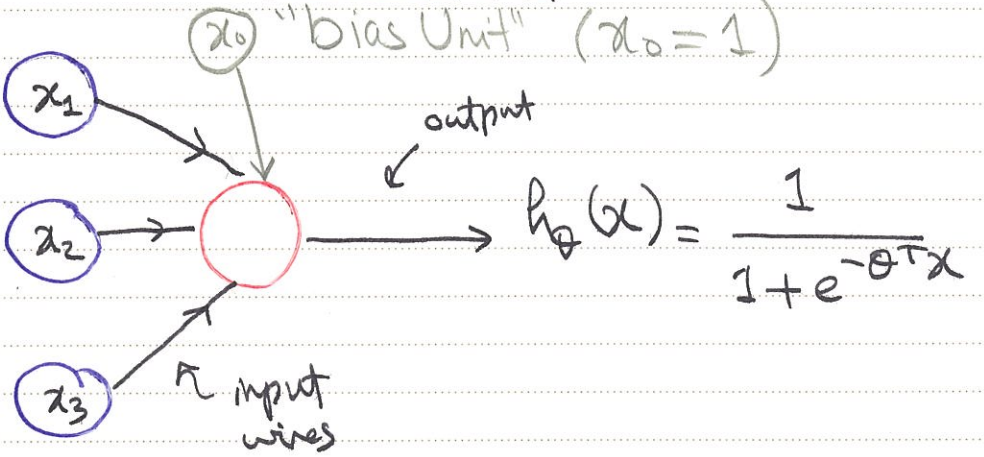
Lecture 46: Model Representation I

Neuron in the brain



A neuron is an electrically excitable cell that receives, processes, and transmits information through electrical and chemical signals. At a very simplistic level, a neuron is a computational unit that gets a number of inputs through its input wires (Dendrites), does some computation, and sends the output via its output wire (Axon) to other neurons. It's basically through these networks of neurons that human thought happens.

Neuron model: Logistic unit

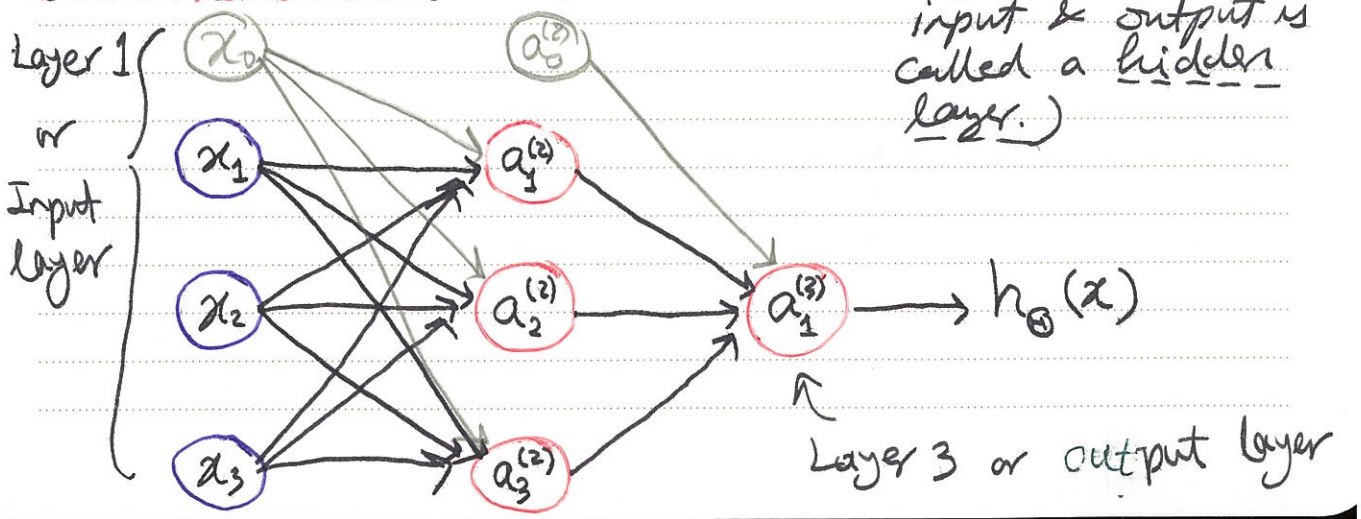


The bias unit, which is a name for $x_0 \equiv 1$, is often not drawn.

Terminology:

* This model is called a neuron (artificial neuron) with a Sigmoid (logistic) activation function.
* θs are often called the "weights" of the model.

Neural Network



Notation:

* $a_i^{(j)} \equiv$ "activation" of unit i in layer j.

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

$\Theta_{ik}^{(j)} \equiv$ Weight of k^{th} feature for unit i in layer $j \to j+1$ mapping.

In the neural network of the previous page, we have:

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

$$a_2^{(2)} = g(\Theta_{20}^{(1)} + \Theta_{11}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$

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

$$a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

$\updownarrow$

$h_\Theta(x)$

As usual: $a_0^{(j)} \equiv 1$

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

For convenience, let $x_i \equiv a_i^{(1)}$

Then we have:

$$a_i^{(j+1)} = g\left(\sum_{k=0}^{s_j} \Theta_{ik}^{(j)} a_k^{(j)}\right)$$

$$= g\left([\Theta^{(j)} a^{(j)}]_i\right)$$

This is why the notation makes sense: we apply the matrix $\Theta^{(j)}$ on the vector of outputs from the j^{th} layer (including $a_0^{(j)} = 1$), and then apply the activation function g, which we've taken to be the Sigmoid function, on every element of the resulting vector. So, we can write the following symbolic equation: (#)

$$a^{(j+1)} = g(\Theta^{(j)} a^{(j)})$$

s_{j+1} dimensional vector

s_{j+1} dimensional vector

applied element wise.

Lecture 47: Model Representation II

Applying equation ⓗ on the previous page to go from input to output of the neural network recursively is called the "vectorized implementation" of "forward propagation". Again, to recap:

Layer 1→2:

$$\begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ \vdots \\ a_{S_2}^{(2)} \end{bmatrix} = g\left(\Theta^{(1)} \begin{bmatrix} 1 \\ a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_{S_1}^{(1)} \end{bmatrix} \right)$$

The activation function g is applied element wise.

Layer 2→3:

$$\begin{bmatrix} a_1^{(3)} \\ a_2^{(3)} \\ \vdots \\ a_{S_3}^{(3)} \end{bmatrix} = g\left(\Theta^{(2)} \begin{bmatrix} 1 \\ a_1^{(2)} \\ a_2^{(2)} \\ \vdots \\ a_{S_2}^{(2)} \end{bmatrix} \right)$$

⋮

where $x_1 \equiv a_1^{(1)}$, $x_2 \equiv a_2^{(1)}$, ..., $x_{S_1} \equiv a_{S_1}^{(1)}$

& $S_1 \equiv$ # of units in layer 1, i.e. # of features.

$S_2 \equiv$ # of units in layer 2

etc.

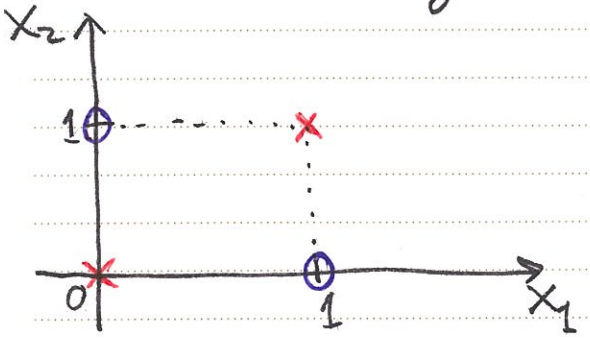
It's interesting to note that the hidden layers can be thought of as a mechanism by which the Neural Network is learning its own features.

The design of a Neural Network is called its architecture.

Lecture 48: Examples and Intuitions I

Let's go through some examples of why Neural Networks can represent complex non-linear functions.

Consider the following non-linear classification example:



We're assuming x_1 & x_2 are binary. This is nothing but an XNOR gate:

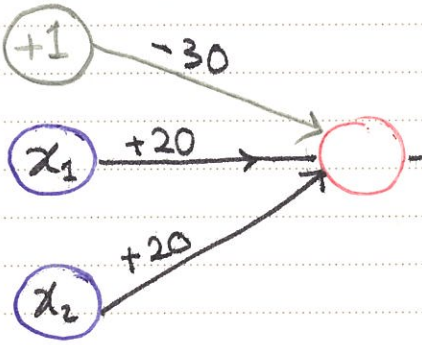
x_1	x_2	y
0	0	1
0	1	0
1	0	0
1	1	1

Very non-linear!

We can think of this problem as a simplification of:

Before tackling XNOR, let's show how we can build AND and OR gates.

AND Gate

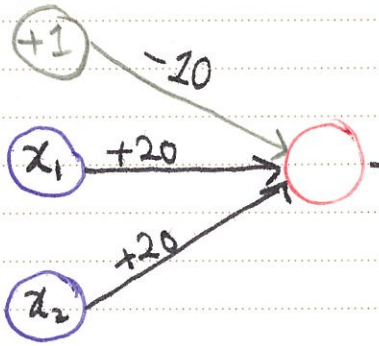


Remember that the Sigmoid function is basically a smoothed-out Heaviside function.

$$h_\theta(x) = g(-30 + 20x_1 + 20x_2)$$

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	$g(+10) \approx 1$

OR Gate

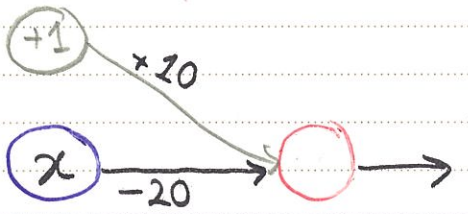


$$h_\theta(x) = g(-10 + 20x_1 + 20x_2)$$

x_1	x_2	$h_\theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(+10) \approx 1$
1	0	$g(+10) \approx 1$
1	1	$g(+30) \approx 1$

Lecture 49: Examples and Intuitions II

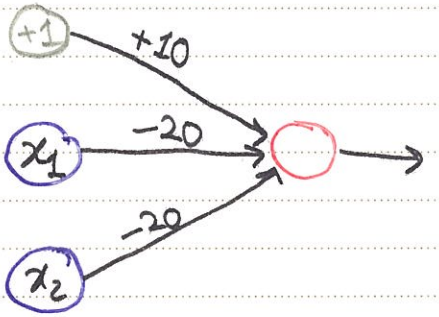
NOT Gate



$$h_\theta(x) = g(+10 - 20x)$$

x	$h_\theta(x)$
0	$g(+10) \approx 1$
1	$g(-10) \approx 0$

(NOT x_1) AND (NOT x_2)



$$h_\theta(x) = g(+10 - 20x_1 - 20x_2)$$

x_1	x_2	$h_\theta(x)$
0	0	$g(+10) \approx 1$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(-30) \approx 0$

And finally, back to the original problem: XNOR Gate.

x_1 XNOR x_2 = (x_1 AND x_2) OR ((NOT x_1) AND (NOT x_2))

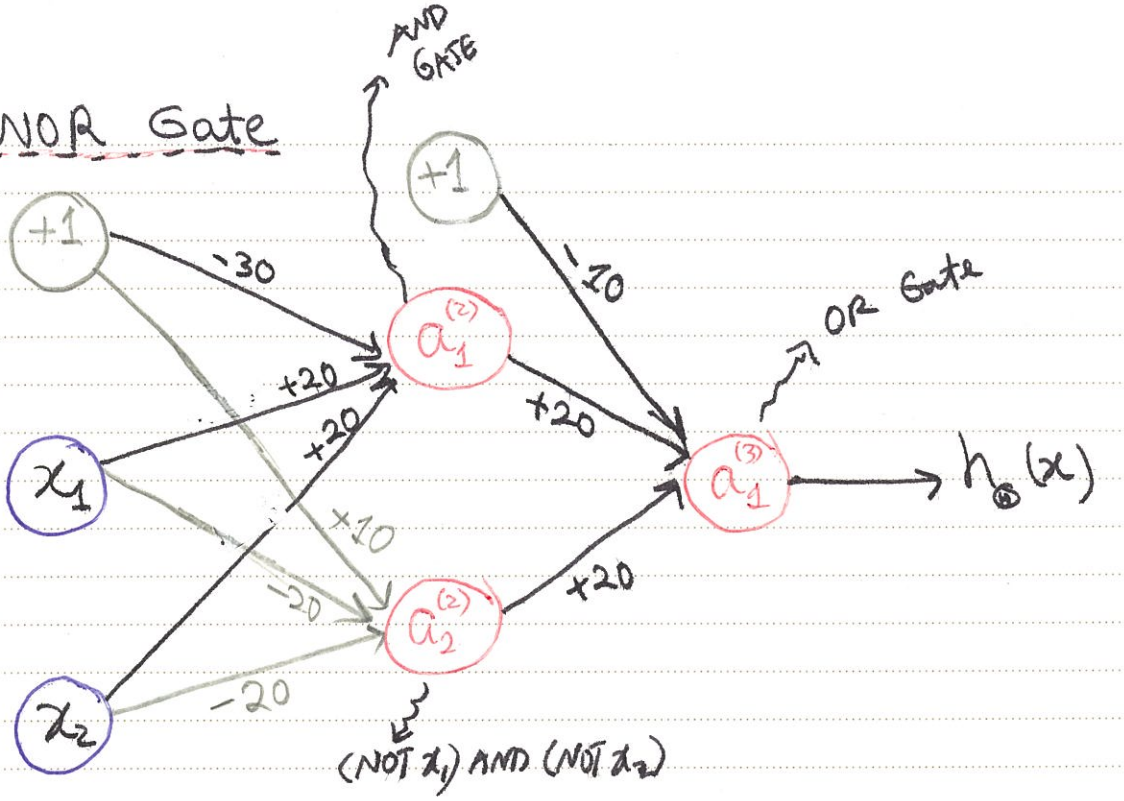
(x_1 AND x_2) $\equiv y$; ((NOT x_1) AND (NOT x_2)) $\equiv z$

Check:

x_1	x_2	y	z	y OR z
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

✓

XNOR Gate



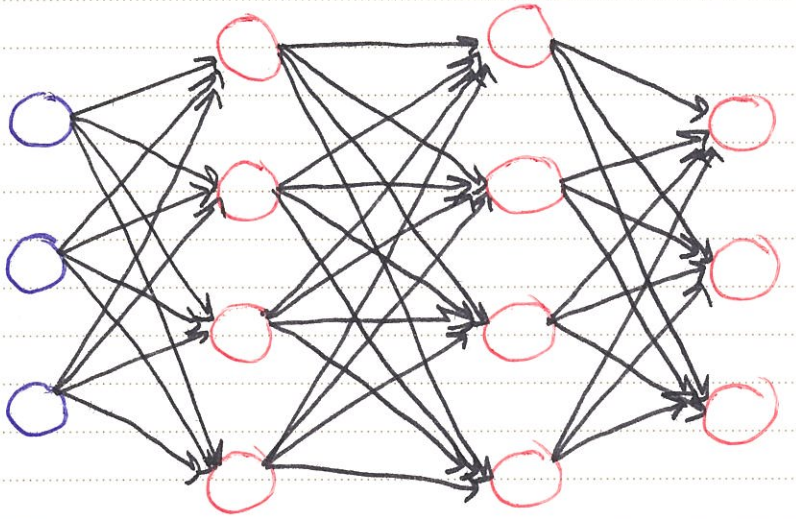
$$\Theta^{(1)} = \begin{bmatrix} -30 & +20 & +20 \\ +10 & -20 & -20 \end{bmatrix} \qquad \Theta^{(2)} = \begin{bmatrix} -10 & +20 & +20 \end{bmatrix}$$

Lecture 50: Multiclass Classification

What if we need to predict not just a binary classification but a multiclass one? For instance, given an image of a hand-written digit, we'd like to classify as $\{0, 1, 2, \ldots, 9\}$.

We'll use an extension of the One-vs-all method, and have multiple output units.

Let's say we have images of cars, motorcycles, and trucks. Given a new image, we'd like to predict whether it's an image of a car, motorcycle, or a truck. We'll have a Neural Network with 3 output units. For example:



$h_\Theta(x) \in \mathbb{R}^3$

We would want: $h_\Theta(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ when input is car

$h_\Theta(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ when input is motorcycle

$h_\Theta(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ when input is truck

The training set $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})$ will be constructed so that

$$y^{(i)} \in \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\}$$

$\updownarrow$ Car $\quad$ $\updownarrow$ Motorcycle $\quad$ $\updownarrow$ Truck

NEURAL NETWORKS: LEARNING

Lecture 51: Cost Function

We will start by looking at classification problems.

Notation:

* Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$
* $L \equiv$ Total number of layers in network
* $s_l \equiv$ Total number of units (not counting bias unit) in layer l
* $K \equiv s_L$ (i.e. # of output units)

Types of classification:

* Binary: $\rightarrow$ $y = 0$ or 1
  $\rightarrow$ 1 output unit (i.e $K = 1$)
  $\rightarrow$ $h_\Theta(x) \in \mathbb{R}$
* Multi-class: $\rightarrow$ $y \in \mathbb{R}^K$ e.g. $\begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\end{bmatrix}$
  $\rightarrow$ K output units ($K \geq 3$)
  $\rightarrow$ $h_\Theta(x) \in \mathbb{R}^K$

The cost function will be a straightforward generalization of that of logistic regression (see page 52):

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)} \ln\left(h_\Theta(x^{(i)})_k\right) + (1-y_k^{(i)}) \ln\left(1-h_\Theta(x^{(i)})_k\right)\right]$$
$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{S_l}\sum_{j=1}^{S_{l+1}}\left(\Theta_{ji}^{(l)}\right)^2$$

Regularization term. Note that just as was the case with logistic regression, the bias weights are not included (i.e. $i=0$ not included in the sum).

Since we could have multiple outputs, we sum over them

Note that $\left[h_\Theta(x^{(i)})\right]_k = a_k^{(L)}$ (when applied to input $x^{(i)}$, of course)

Lecture 52: Backpropagation Algorithm

Now that we have a cost function, we need to minimize it. We already know how to compute $J(\Theta)$ with forward propagation. We now need to compute $\frac{\partial J}{\partial \Theta_{ij}^{(l)}}$, so that

algorithms like Gradient Descent or Conjugate Gradient can use $\{J(\Theta), \frac{\partial J}{\partial \Theta_{ij}^{(l)}}\}$ to find the minimum of $J(\Theta)$.

The algorithm we will come up with below is called Back propagation.

(I will derive this below, but this isn't done in the lecture.)

$$J(\Theta) = +\frac{1}{m}\sum_{i=1}^{m} J^{(i)}(\Theta) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{S_{l+1}}\sum_{j=1}^{S_l}\left[\Theta_{ij}^{(l)}\right]^2 \quad (\#)$$

where

$$J^{(i)}(\Theta) \equiv -\sum_{K=1}^{S_L} y_K^{(i)} \ln\left([h_\Theta(x^{(i)})]_K\right) + (1-y_K^{(i)})\ln\left(1-[h_\Theta(x^{(i)})]_K\right)$$

It's then enough to compute $\nabla_\Theta J^{(i)}$, i.e. the gradients for a single training example, because generalizing to $\nabla_\Theta J$ is trivial via (#).

Backpropagation for a single example

For one training set example (x, y):

$$[h_\Theta(x)]_K = a_K^{(L)}$$

$$a_K^{(l)} = g(z_K^{(l)}) \quad \text{where} \quad z_K^{(l)} = \sum_{K'=0}^{S_{l-1}} \Theta_{KK'}^{(l-1)} a_{K'}^{(l-1)}$$

We will assume g to be the Sigmoid function for layer L

$$g(z) = \frac{1}{1+e^{-z}} \quad ; \quad g'(z) = g(z)\,[1-g(z)]$$

$$\frac{d}{dz} \ln g(z) = 1 - g(z) \quad ; \quad \frac{d}{dz} \ln (1-g(z)) = -g(z)$$

Assuming the activation function is the Sigmoid function for the output layer, let's use the nice properties above:

$$\frac{\partial J}{\partial \Theta^{(L)}_{ij}} = -\sum_{k=1}^{S_L} \left\{ y_k \left[1 - g(z_k^{(L)})\right] - (1-y_k)\, g(z_k^{(L)}) \right\} \frac{\partial z_k^{(L)}}{\partial \Theta^{(L)}_{ij}}$$

$$= \sum_{k=1}^{S_L} \left[a_k^{(L)} - y_k \right] \frac{\partial z_k^{(L)}}{\partial \Theta^{(L)}_{ij}} \qquad (*)$$

$$z_k^{(L)} = \sum_{k_1=0}^{S_{L-1}} \Theta^{(L-1)}_{k k_1}\, a_{k_1}^{(L-1)}$$

$$\frac{\partial z_k^{(L)}}{\partial \Theta^{(L-1)}_{ij}} = a_j^{(L-1)}\, \delta_{ki}$$

$$\frac{\partial z_k^{(L)}}{\partial \Theta^{(L-2)}_{ij}} = \sum_{k_1=1}^{S_{L-1}} \Theta^{(L-1)}_{k k_1}\, g'(z_{k_1}^{(L-1)})\, \frac{\partial z_{k_1}^{(L-1)}}{\partial \Theta^{(L-2)}_{ij}}$$

$$= a_j^{(L-2)}\, \Theta^{(L-1)}_{ki}\, g'(z_i^{(L-1)})$$

Note that k_1 starts from 1 & not 0 because $a_0^{(L-1)} = 1$ & $\frac{\partial a_0^{(L-1)}}{\partial \Theta^{(L-2)}_{ij}} = 0$

$$\frac{\partial Z_K^{(L)}}{\partial \Theta_{ij}^{(L-3)}} = \sum_{k_1=1}^{S_{L-1}} \Theta_{Kk_1}^{(L-1)} g'(Z_{k_1}^{(L-1)}) \sum_{k_2=1}^{S_{L-2}} \Theta_{k_1 k_2}^{(L-2)} g'(Z_{k_2}^{(L-2)}) \frac{\partial Z_{k_2}^{(L-2)}}{\partial \Theta_{ij}^{(L-3)}}$$

$$= \sum_{k_1=1}^{S_{L-1}} \Theta_{Kk_1}^{(L-1)} g'(Z_{k_1}^{(L-1)}) \times \Theta_{k_1 i}^{(L-2)} g'(Z_i^{(L-2)}) \times a_j^{(L-3)}$$

let $\boxed{\Lambda_{KK'}^{(l)} \equiv \Theta_{KK'}^{(l)} g'(Z_{K'}^{(l)})}$ (K' starts from 1)

$l \in \{2, 3, \dots, L-1\}$
$K' \in \{1, \dots, S_l\}$
$K \in \{1, \dots, S_{l+1}\}$

We can then see that:

$$\frac{\partial Z_K^{(L)}}{\partial \Theta_{ij}^{(L-1)}} = a_j^{(L-1)} \delta_{Ki} \qquad \text{Layer } L-1$$

$$\frac{\partial Z_K^{(L)}}{\partial \Theta_{ij}^{(L-2)}} = a_j^{(L-2)} \Lambda_{Ki}^{(L-1)} \qquad \text{Layer } L-2$$

$$\frac{\partial Z_K^{(L)}}{\partial \Theta_{ij}^{(L-3)}} = a_j^{(L-3)} \left[\Lambda^{(L-1)} \Lambda^{(L-2)}\right]_{Ki} \qquad \text{Layer } L-3$$

$\vdots$

Now let's plug these back into ⊛ on pg 73:

let $\boxed{\delta_K^{(L)} \equiv a_K^{(L)} - y_K}$

We then have that

$$\frac{\partial J}{\partial \Theta_{ij}^{(L-1)}} = a_j^{(L-1)} \delta_i^{(L)}$$

Transpose

$$\frac{\partial J}{\partial \Theta_{ij}^{(L-2)}} = \sum_{K=1}^{S_L} \delta_K^{(L)} a_j^{(L-2)} \Lambda_{Ki}^{(L-2)} = a_j^{(L-2)} \left[\Lambda^{(L-1)T} \delta^{(L)}\right]_i$$

$$\frac{\partial J}{\partial \Theta_{ij}^{(L-3)}} = \sum_{K=1}^{S_L} \delta_K^{(L)} a_j^{(L-3)} \left[\Lambda^{(L-1)} \Lambda^{(L-2)}\right]_{Ki}$$

$$= a_j^{(L-3)} \left[\Lambda^{(L-2)T} \Lambda^{(L-1)T} \delta^{(L)}\right]_i$$

& so on. The pattern should be clear now:

Layer L: $\delta^{(L)} \equiv a^{(L)} - y$ S_L-dimensional

Layer L-1: $\delta^{(L-1)} \equiv \Lambda^{(L-1)T} \delta^{(L)}$ S_{L-1}-dimensional

Layer L-2: $\delta^{(L-2)} \equiv \Lambda^{(L-2)T} \delta^{(L-1)}$ S_{L-2}-dimensional

⋮

Layer 2: $\delta^{(2)} \equiv \Lambda^{(2)T} \delta^{(3)}$ S_2-dimensional

After all the δs are computed:

$$\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)} \qquad l \in \{1, 2, \ldots, L-1\}$$

Also see page 79

Let's now generalize to the case of multiple training examples.

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

Set $\Delta_{ij}^{(l)} = 0 \quad \forall i, j, l. \left(l \in \{1, 2, \dots, L-1\}, i \in \{1, \dots, s_{l+1}\}, j \in \{1, \dots, s_l\}\right)$

For $i = 1$ to m

(1) Forward propagate.

* Set $a^{(1)} = x^{(i)}$
* Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \dots, L$

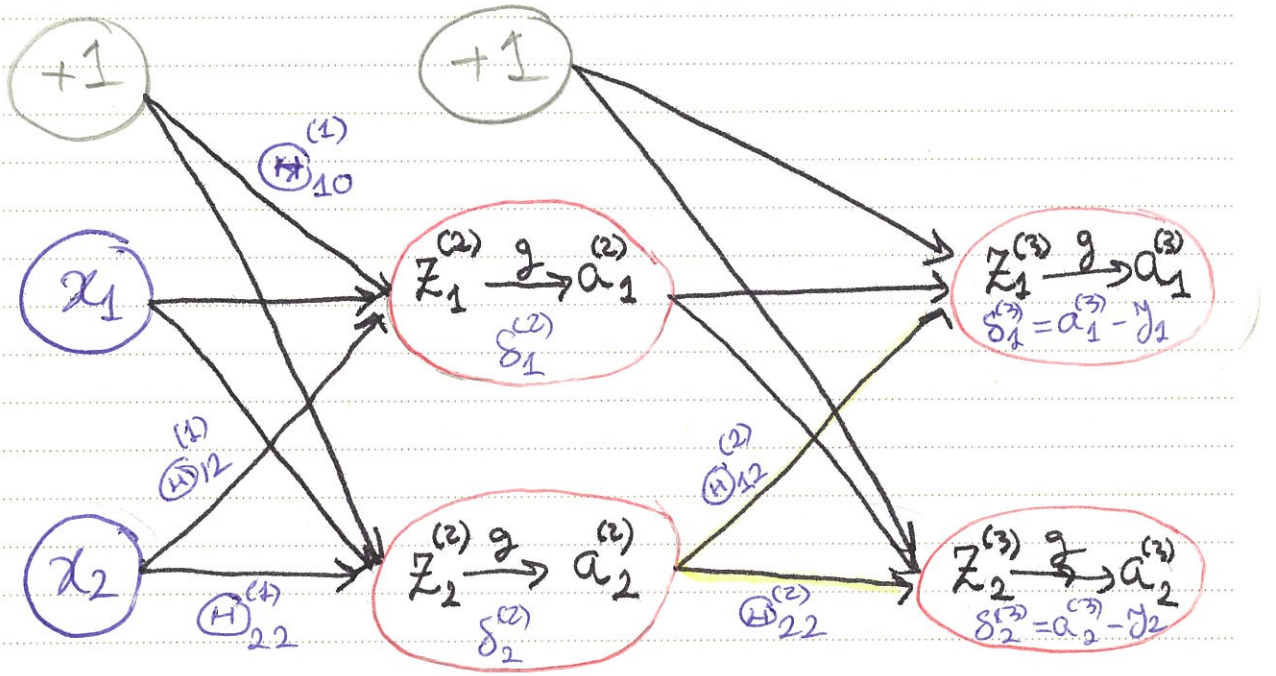
(2) Backpropagate

* Compute $\delta^{(L)}, \delta^{(L-1)}, \dots, \delta^{(2)}$
* $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

$$\frac{\partial J}{\partial \Theta_{ij}^{(l)}} = \frac{1}{m} \Delta_{ij}^{(l)} + \lambda (1 - \delta_{j0}) \Theta_{ij}^{(l)} \equiv D_{ij}^{(l)}$$

This notation is used in the lecture to denote the gradient of J w.r.t $\Theta_{ij}^{(l)}$.

Lecture 53: Backpropagation Intuition



When we compute $\frac{\partial J}{\partial \Theta^{(l)}_{ij}}$ for a given training example,

what we're really interested in is what happens to $J(\Theta)$ if we change $\Theta^{(l)}_{ij}$ a little bit: $\Theta^{(l)}_{ij} \to \Theta^{(l)}_{ij} + \delta \Theta^{(l)}_{ij}$.

Consider the example above, for instance. Say we want to know the impact of changing $\Theta^{(1)}_{12} \to \Theta^{(1)}_{12} + \delta\Theta^{(1)}_{12}$. It should be clear that:

$$\frac{\partial J}{\partial \Theta^{(1)}_{12}} = \frac{\partial J}{\partial z^{(2)}_1} \frac{\partial z^{(2)}_1}{\partial \Theta^{(1)}_{12}} = \frac{\partial J}{\partial z^{(2)}_1} x_2 = \frac{\partial J}{\partial z^{(2)}_1} a^{(1)}_2$$

(By def.)

In other words, the non-trivial part is computing the impact of $z_1^{(2)} \to z_1^{(2)} + \delta z_1^{(2)}$ on the cost function. As a matter of fact, it can be easily seen from the calculations of the previous lecture that:

$$\delta_i^{(l)} = \frac{\partial J}{\partial z_i^{(l)}} \qquad \begin{matrix} l \in \{2, \dots, L\} \\ i \in \{1, \dots, s_l\} \end{matrix}$$

Let's spill out the calculation of $\delta_2^{(2)}$ explicitly in the single example of previous page: (see pages 74-75)

$$\begin{aligned} \delta_2^{(2)} &= \left[\Lambda^{(2)T} \delta^{(3)}\right]_2 = \left[\Lambda^{(2)T}\right]_{21} \delta_1^{(3)} + \left[\Lambda^{(2)T}\right]_{22} \delta_2^{(3)} \\ &= \Lambda_{12}^{(2)} \delta_1^{(3)} + \Lambda_{22}^{(2)} \delta_2^{(3)} \\ &= \textcircled{H}_{12}^{(2)} g'(z_2^{(2)}) \delta_1^{(3)} + \textcircled{H}_{22}^{(2)} g'(z_2^{(2)}) \delta_2^{(3)} \\ &= \left(\textcircled{H}_{12}^{(2)} \delta_1^{(3)} + \textcircled{H}_{22}^{(2)} \delta_2^{(3)} \right) g'(z_2^{(2)}) \end{aligned}$$

This corresponds to a backwards weighting of the highlighted edges in the figure on previous page.

In fact, we can rewrite the δs more explicitly in terms of $\textcircled{H}$s:

Layer L: $\delta_k^{(L)} = a_k^{(L)} - y_k$ $\quad k \in [1, s_L]$

Layer $L-1$: $\delta_k^{(L-1)} = \left[\Theta^{(L-1)^T} \delta^{(L)}\right]_k g'(z_k^{(L-1)})$ $\quad k \in [1, s_{L-1}]$

Layer $L-2$: $\delta_k^{(L-2)} = \left[\Theta^{(L-2)^T} \delta^{(L-1)}\right]_k g'(z_k^{(L-2)})$ $\quad k \in [1, s_{L-2}]$

$\vdots$

Layer 2: $\delta_k^{(2)} = \left[\Theta^{(2)^T} \delta^{(3)}\right]_k g'(z_k^{(2)})$ $\quad k \in [1, s_2]$

Lecture 54 | Implementation Note: Unrolling Parameters

This is a technical note about how we may need to use a vector representation (as opposed to matrix representation) of $\Theta^{(l)}$ and also the gradients $D^{(l)}$. At least in Octave, minimization algorithms, such as fminunc, work with vectors & not matrices.

Example: * functionVal, gradientVec] = costFunction(thetaVec)
(gradientVec → vector; thetaVec → vector)

* fminunc(@costFunction, initialThetaVec, options)
(initialThetaVec → vector)

Suppose we start with some initial guess $\Theta^{(1)}, \Theta^{(2)}, \ldots, \Theta^{(L)}$. We first unroll these into a single vector initialThetaVec to pass to fminunc. Inside of costFunction, we reshape thetaVec into $\Theta^{(1)}, \ldots, \Theta^{(L)}$, use forward prop / back prop to compute $D^{(1)}, \ldots, D^{(L)}$ & $J(\Theta)$ & then unroll $D^{(1)}, \ldots, D^{(L)}$ into a single vector gradientVec.