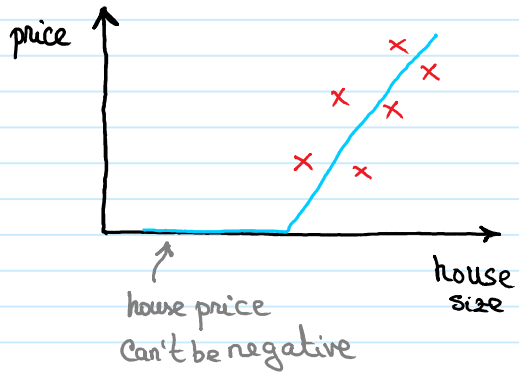
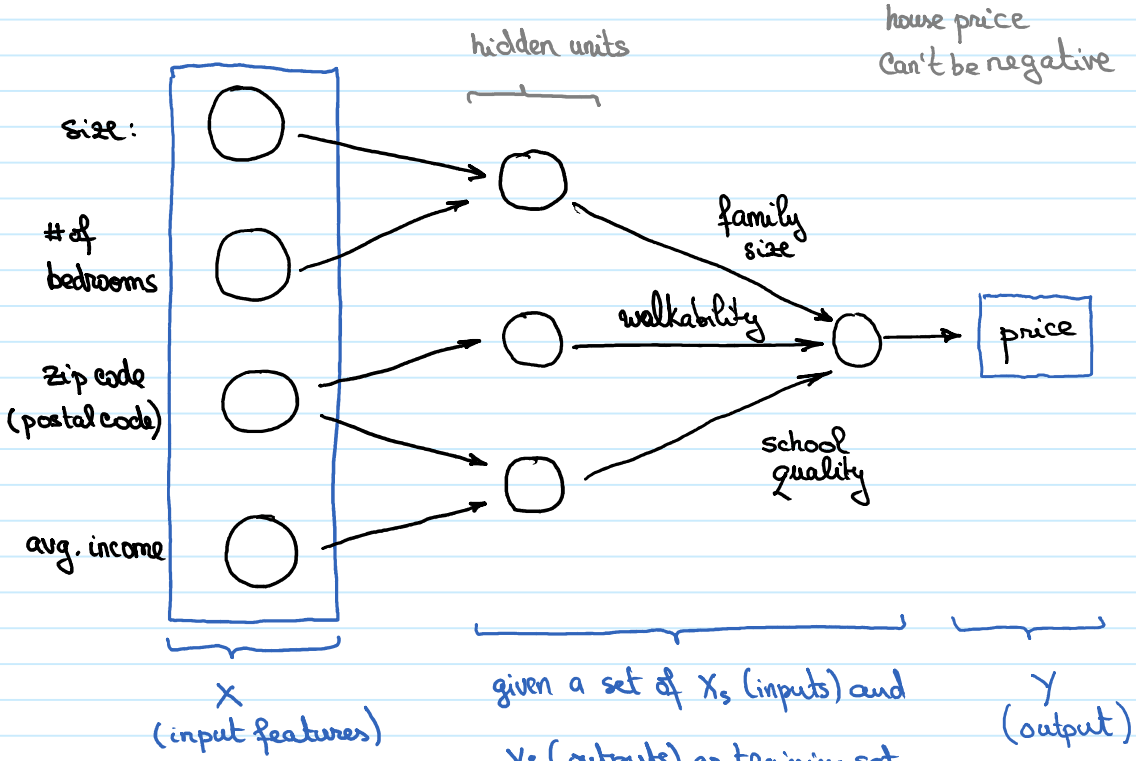
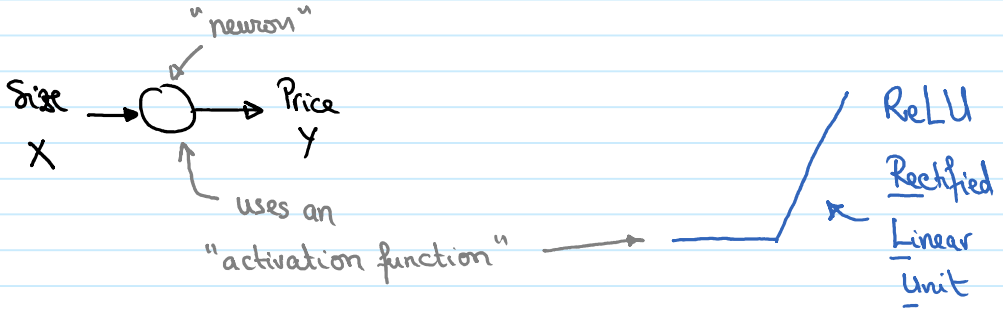
Week 1 Intro to Neural Networks

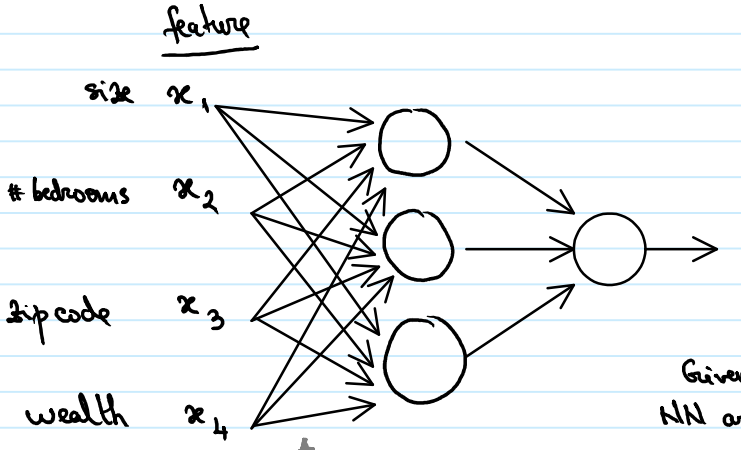
Saturday, 23 June, 2018 20:30

Simple model - housing price prediction









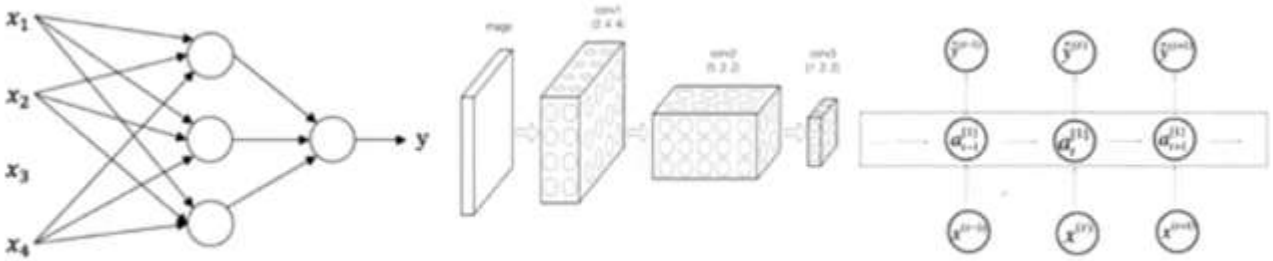
Given enough training data (x,y) NN are remarkably good @ finding the mapping function of $x \rightarrow y$

when all input features are connected to all nodes the network is called densely connected (or fully connected)

SUPERVISED LEARNING EXAMPLES

Input (x)	Output (y)	Application	
Home features	Price	Real Estate	standard NN
Ad, user info	Click on ad? (0/1)	Online Advertising	standard NN
Image	Object (1 ... 1,000)	Photo tagging	Convolution NN (CNN)
Audio	Text transcript	Speech recognition	Recurrent NN (RNN)
English	Chinese	Machine translation	Recurrent NN (RNN)
Image, Radar info	Position of other cars	Autonomous driving	advance NN

TYPES OF NEURAL NETS



Standard NN | Convolutional NN | Recurrent NN

Screen clipping taken: 2018-06-23 21:38

where there are spatial relationships between input features (image processing)

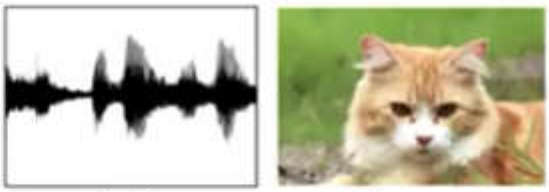
where the input data is a sequence that has information encoded in it's temporal order (language is commonly represented as an RNN)

Structured Data

Size	#bedrooms	...	Price (1000$s)
2104	3		400
1600	3		330
2400	3		369
⋮	⋮		⋮
3000	4		540

User Age	Ad Id	...	Click
41	93242		1
80	93287		0
18	87312		1
⋮	⋮		⋮
27	71244		1

Unstructured Data



Audio | Image

Four scores and seven years ago...

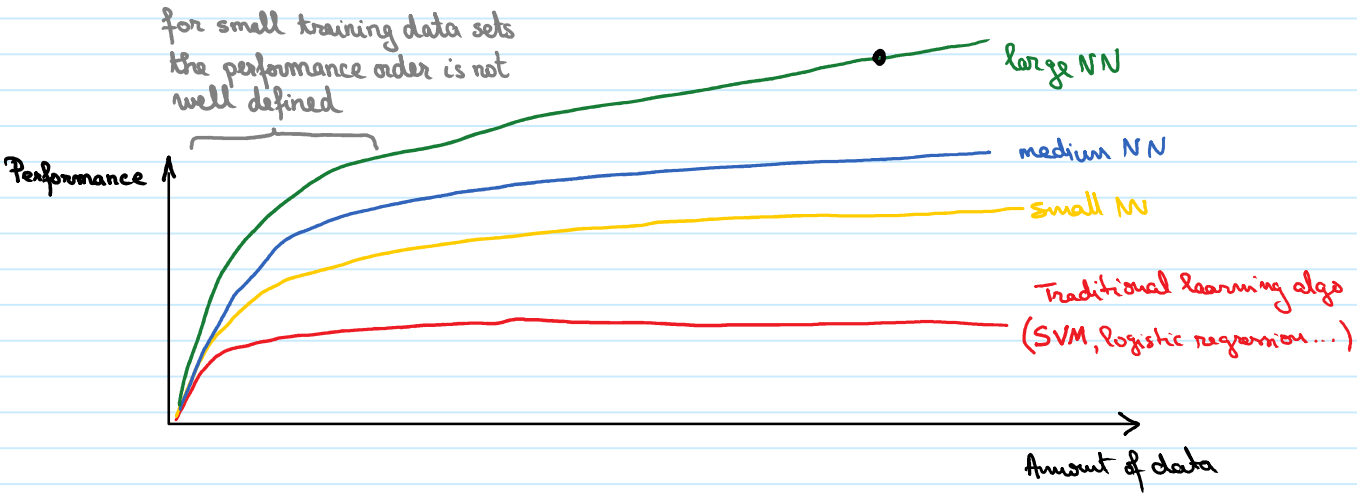
Text

Screen clipping taken: 2018-06-23 21:43

- Each of the input features & output labels has a well defined value / meaning
  - → more economic value (better ads system, product recommendations)

Pictures or raw audio or text

- The input features can be
  - → pixel values in an image
  - → individual words in a text
- Computers has more trouble interpretting unstructured data.
  - → more news coverage



labeled data (x, y)

m : # of input/output examples

Scale of NN (many layers, many nodes, many connection)

+

Scale of input data (large digitized input sets)

+

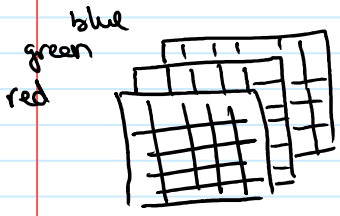
fast hardware

High performance outputs of neural nets.

Week 2 Neural Networks Basics

Sunday, 24 June, 2018 22:56

Input (x) → Output $y = \{1; 0\}$ cat or no cat.
an RGB 64×64 px img.



↓ unroll each channel & concatenate them into one long ass vector

$$x = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ 134 \\ \vdots \end{bmatrix} \begin{matrix} \} \text{ red} \\ \\ \\ \} \text{ grn} \\ \\ \} \text{ blue} \end{matrix}$$

Size of $x = n_x =$
$64 \times 64 \times 3 =$
12,288 values.
↳ this x is the input feature vector

Notation for the course

Thursday, 05 July, 2018 23:02

Binary Classification:

Single training example is represented by a pair:

$$(x, y)$$

where x is an n_x dimensional feature vector

i.e. $x \in \mathbb{R}^{n_x}$

and y (the label) is $\{0, 1\}$

The training set is comprised of m training examples:

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

$m = m_{train}$ vs m_{test} = # of test examples.

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad n_x \text{ features} \qquad Y = \begin{bmatrix} y^{(1)}, y^{(2)} \cdots y^{(m)} \end{bmatrix}$$

m examples

$$Y \in \mathbb{R}^{1 \times m}$$

$y.shape \rightarrow (1, m)$

$$X \in \mathbb{R}^{n_x \times m} \rightarrow X.shape \rightarrow (n_x, m)$$

Logistic Regression

Monday, 25 June, 2018 21:00

Given an input feature vector x, we want to determine $\hat{y}$ (an estimate of y).

$\hat{y}$ is the probability that $y = 1$ given input feature x.

$$\hat{y} = P(y=1 \mid x)$$

i.e. if x is a picture, $\hat{y}$ tells us what is the chance that x is a cat picture

Inputs:

$x \in \mathbb{R}^{n_x}$ (input feature vector)

$w \in \mathbb{R}^{n_x}$ and $b \in \mathbb{R}^1$ (parameters)

↳ corresponds to y intercept

Outputs:

$\hat{y}$

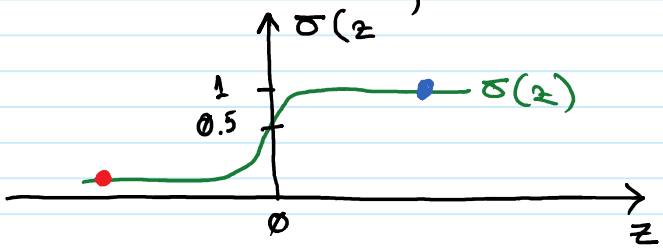
- OPTION 1: $\hat{y} = w^T x + b$ (this is the approach used for linear regression)

This doesn't work because $\hat{y}$ is supposed to be a probability

i.e. $\hat{y} \in [0; 1]$

- OPTION 2: $\hat{y} = \sigma(w^T x + b)$

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



- If $z \to \infty \Rightarrow \sigma(z) \to \frac{1}{1+0} = 1$
- If $z \to -\infty \Rightarrow \sigma(z) \to \frac{1}{1+\infty} = 0$

ASIDE:

In some courses w & b are not kept separate.

→ a new feature $x_0 := 1$ is defined such that $x \in \mathbb{R}^{n_x + 1}$

→ in this case $\hat{y} = \sigma(\theta^T x)$, where $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_{n_x} \end{bmatrix}$ $\left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\}$ $\begin{matrix} b \\ \\ \\ w \end{matrix}$

Logistic Regression Cost Function

Thursday, 05 July, 2018 23:06

Given:

→ a probability $\hat{y} = \sigma(w^T x + b)$, where $\sigma(z) = \frac{1}{1+e^{-z}}$

→ a training set of m examples $\{(x^{(1)}, y^{(1)}), \dots (x^{(m)}, y^{(m)})\}$

we want to find $\hat{y}^{(i)}$ such that $\hat{y}^{(i)} \approx y^{(i)}$, where $i \in \{1 \dots m\}$

predictions we made using training example inputs ($\hat{y}$) match the "ground truth" (training example output y)

i.e. $y^{(i)} = \sigma(w^T x^{(i)} + b)$, where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$

$x^{(i)}, y^{(i)}, z^{(i)}$ are parameters / variables / etc associated with the i^{th} training example

WHAT LOSS (OR ERROR) FUNCTION CAN WE USE?

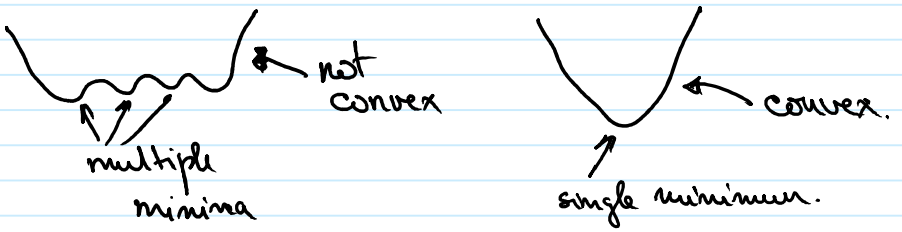
A loss function measures how good the neural network predicted output $\hat{y}$ is when the training value output (ground truth) is y

OPTION 1 AVERAGE SQUARE OF THE ERROR

- $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$

This loss function is not used much because the optimization problem is not convex

i.e. there is no global minimum towards which Gradient Descent can optimize



OPTION 2 LOGISTIC REGRESSION LOSS (AKA CROSS ENTROPY LOSS)

- $\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$

Intuition: we want $\mathcal{L}(\hat{y}, y)$ to be as small as possible when $\hat{y}$ is close to y.

→ If $y = 1 \Rightarrow \mathcal{L}(\hat{y}, y) = -\log \hat{y}$ ← want this to be small
→ want $\log \hat{y}$ large → want $\hat{y}$ large
but $\hat{y} = \sigma(z) \Rightarrow \hat{y} = 1$ at maximum

→ If $y = 0 \Rightarrow \mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$ ← want this to be small
→ want $\log(1-\hat{y})$ large → want $1-\hat{y}$ large
so want $\hat{y}$ small
but $\hat{y} = \sigma(z) \rightarrow \hat{y} = 0$ at minimum

So by using the Logistic Regression loss function $\hat{y}$ will be pushed towards y at the extremes.

COST FUNCTION:

Cost function averages the loss function over all training examples.

$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

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

The COST FUNCTION applies to all examples and it is used to determine the values of w & b.

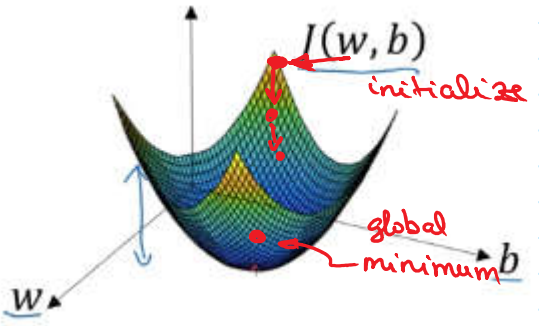
The loss function $\mathcal{L}(\hat{y}, y)$ measures how well we are doing on a single example

Gradient descent

Thursday, 05 July, 2018 23:06

Given a cost function $J(w,b)$ over a given training set $\{(x^{(1)}, y^{(1)}), \ldots (x^{(m)}, y^{(m)})\}$ we want to find parameters w & b such that $J(w,b)$ is as small as possible.

i.e. find w & b that minimize $J(w,b)$



somewhere
(almost any initial value works)

Screen clipping taken: 2018-06-25 22:14

It turns out our $J(w,b)$ is a convex function (like a bowl)

* ignore b for now (makes drawing easier)

Repeat {

$$w := w - \alpha \frac{dJ(w)}{dw}$$

}

learning rate

$$w := w - \alpha \cdot dw$$

(this is the label of the variable used in code)

When taking into account optimizing both w & b we have

$$w := w - \alpha \frac{dJ(w,b)}{dw}$$

$$b := b - \alpha \frac{dJ(w,b)}{db}$$

$\frac{\partial J(w,b)}{\partial w}$ "partial derivative"

$\frac{\partial J(w,b)}{\partial b}$

dw

$$b := b - \alpha \frac{d J(w,b)}{d b}$$

$\frac{\partial J(w,b)}{\partial b}$ → dw / db (when implemented in code)

Reminder:

$$f(a) = \log_e a \Rightarrow \frac{d f(a)}{da} = \frac{1}{a}$$

Computation Graph

Thursday, 05 July, 2018 23:09

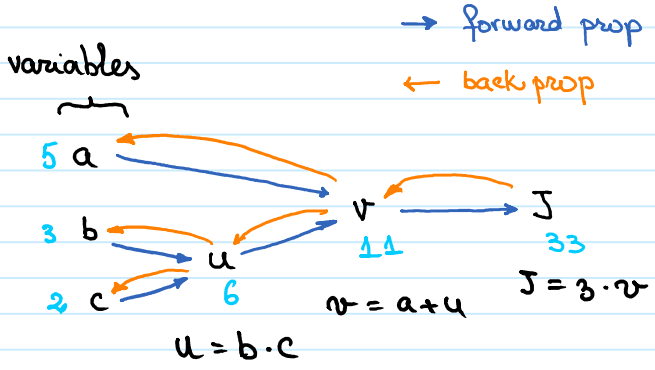
In a Neural Net:

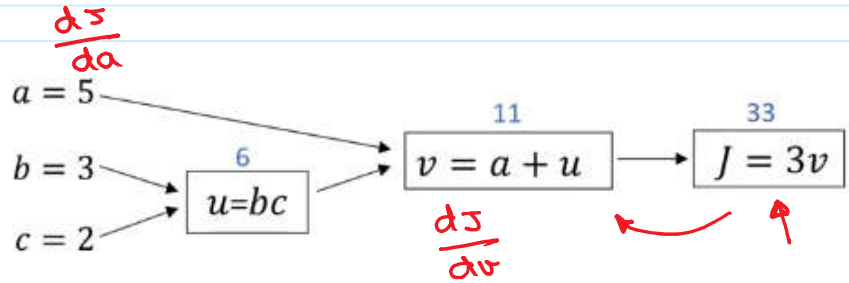
- Forward pass (Forward propagation) computes the output of the NN
- Backward pass (Back propagation) computes gradients (derivatives)

COMPUTATION GRAPH:

$J(a,b,c) = 3(a+bc)$

$u = bc$, $v = a+u$, $J = 3v$





Screen clipping taken: 2018-06-25 23:01

$\frac{dJ}{dv}$ = ? if we change the value of v a little bit how does the value of J change?

$J = 3v$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

$\Rightarrow \frac{dJ}{dv} = 3$

$a = 5 \rightarrow 5.001$

$v = 11 \rightarrow 11.001$

$\frac{dv}{da} = 1$

$J = 33 \rightarrow 33.001$

$$\frac{dJ}{da} = ? = \frac{dJ}{dv} \times \frac{dv}{da}$$

$= 3 \times 1 = 3$

aka "chain rule"

$a \rightarrow v \rightarrow j \Rightarrow \frac{dJ}{da} = \frac{dJ}{dv} \times \frac{dv}{da}$

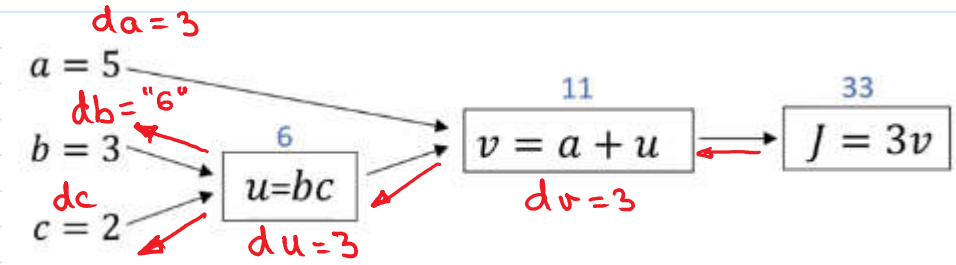
J

$\frac{\partial \text{Final Output Var}}{\partial \text{var}}$ ← in Python we label this as "dvar"

where dvar can be $\{a, b, c, u, v\}$

$\frac{\partial J}{\partial a} \rightarrow$ "da" $= 3$

$\frac{\partial J}{\partial v} \rightarrow$ "dv" $= 3$



Screen clipping taken: 2018-06-25 23:14

$$\frac{dJ}{du} = \underbrace{\frac{dJ}{dv}}_{3} \cdot \underbrace{\frac{dv}{du}}_{1} = 3 \cdot 1 = 3$$

$$\frac{dJ}{db} = \underbrace{\frac{dJ}{dv}}_{3} \underbrace{\frac{dv}{du}}_{1} \cdot \underbrace{\frac{du}{db}}_{c=2} = 3c = 6$$

$$\frac{dJ}{dc} = 3 \cdot b = 9$$

Logistic Regression Gradient Descent

Thursday, 05 July, 2018 23:22

$z = w^T x + b$

For a given example a

$\hat{y} = a = \sigma(z)$

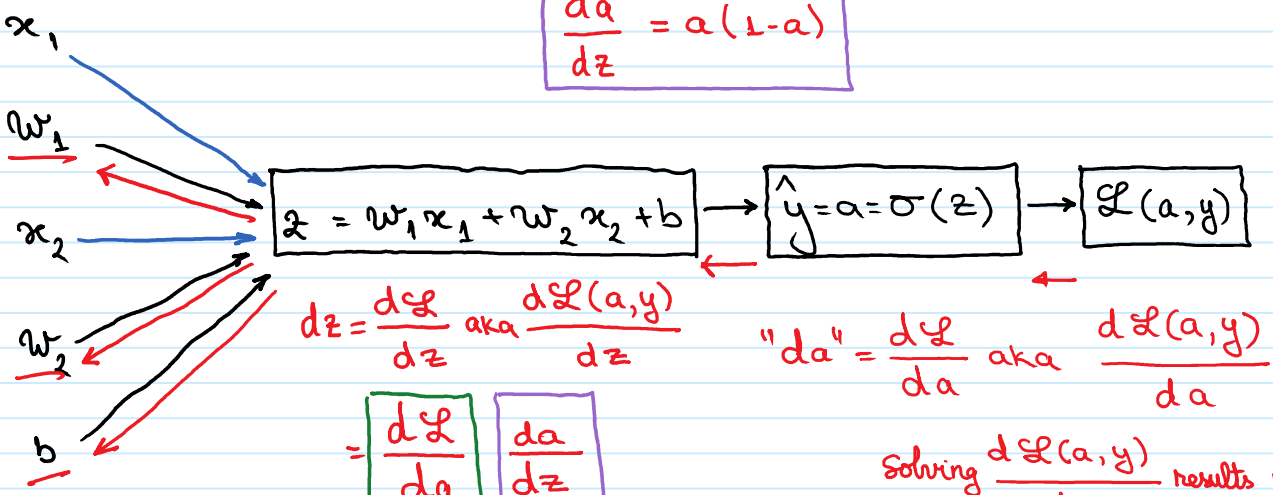
$\mathcal{L}(\hat{y}, y) = -\left(y \log(a) + (1-y) \log(1-a)\right)$

Let's say we only have two features x_1 & x_2

We also have to figure out w and b (want to modify w_1, w_2 & b to minimize $\mathcal{L}$)

Remember: $a = \sigma(z) = \frac{1}{1+e^{-z}} \Rightarrow \frac{da}{dz} = \frac{d}{dz}\frac{1}{1+e^{-z}}$

$$\boxed{\frac{da}{dz} = a(1-a)}$$



$dz = \frac{d\mathcal{L}}{dz}$ aka $\frac{d\mathcal{L}(a,y)}{dz}$

$= \boxed{\frac{d\mathcal{L}}{da}} \boxed{\frac{da}{dz}}$

$= \frac{a-y}{a(1-a)} \cdot a(1-a)$

$dz = a - y$

"da" $= \frac{d\mathcal{L}}{da}$ aka $\frac{d\mathcal{L}(a,y)}{da}$

Solving $\frac{d\mathcal{L}(a,y)}{da}$ results in:

$= -\frac{y}{a} + \frac{1-y}{1-a}$

$= \frac{-y + ya + a - ya}{a(1-a)}$

$$\boxed{\frac{d\mathcal{L}}{da} = \frac{a-y}{a(1-a)}}$$

$\frac{d\mathcal{L}}{dw_1}$ = "dw_1" $= x_1 \cdot dz = x_1(a-y)$

dw_1

$dw_2 = x_2 \cdot dz = x_2(a-y)$

$db = dz = a-y$

Repeat {

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

}

This is for a SINGLE example

For multiple examples we use $J(w,b)$ the cost function.

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$$

where $a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma\left(w^T x^{(i)} + b\right)$

So we know how to compute $dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$

$$\frac{\partial J(w,b)}{\partial w_1} = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial w_1}\mathcal{L}(a^{(i)}, y^{(i)})}_{dw_1^{(i)} \text{ for the training example } (x^{(i)}, y^{(i)})}$$

dw_1 = average dw_1 over all training examples.

Non-vectorized Gradient Descent

Tuesday, 26 June, 2018 19:30

GRADIENT DESCENT ON M EXAMPLE NON-VECTORIZED

Initial values: $J = 0$; $dw_1 = 0$; $dw_2 = 0$; $db = 0$

for loop { For $i := 0$ to m # iterate over all training examples.

$$z^{(i)} := w^T x^{(i)} + b$$

$$a^{(i)} := \sigma(z^{(i)})$$

$$J := J - [y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$$

$$dz^{(i)} := a^{(i)} - y^{(i)}$$

for loop for multiple features {

$$dw_1 := dw_1 + x_1^{(i)} dz^{(i)}$$

$$dw_2 := dw_2 + x_2^{(i)} dz^{(i)}$$

} # assume we only have 2 features otherwise calculate $dw_3, dw^4 \ldots$

$$db := db + dz^{(i)}$$

dw_x is used as an accumulator

$$dw_1 = \frac{\partial J}{\partial w_1}$$

$$J := \frac{J}{m}$$

$$dw_1 := \frac{dw_1}{m}$$

$$dw_2 := \frac{dw_2}{m}$$

$$db := \frac{db}{m}$$

$$w_1 := w_1 - \alpha\, dw_1$$

$$w_2 := w_2 - \alpha\, dw_2$$

$$b := b - \alpha\, db$$

All of the above implements a single step of Gradient Descent.

→ we need to repeat this for every step of Gradient Descent

The Problem with this algorithm is that it uses two for-loops.

→ for-loops are not computationally efficient and slow execution

Vectorization is used to implement this algorithm without using any for-loops

Vectorization

Friday, 06 July, 2018 20:44

The art of removing "for loops" in code.

$z = w^T x + b \qquad w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \quad x = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \qquad w \in \mathbb{R}^{n_x} \quad x \in \mathbb{R}^{n_x}$

Non-vectorized implementation:

```
z = 0
for i in range(n_x):
    z += w[i] * x[i]
z += b
```

Vectorized:

```
z = np.dot(w, x) + b
```

(np.dot(w, x) = $w^T x$)

np stands for numpy

GPU } parallelization using SIMD (single instruction multiple data)
CPU }

Whenever possible, avoid explicit for-loops:

$u = A \cdot v$

$u_i = \sum_j A_{ij} \cdot v_j$

Non-vectorized	Vectorized

`u = np.zeros(n,1)`
`for i= ...`
`  for j= ...`
`  u[i] = A[i][j] * v[j]`
	`u = np.dot(A,v)`

If we need to apply the exponential operation on every element of a matrix / vector.

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \rightarrow u = \begin{bmatrix} e^{v_1} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

Non-vectorized	Vectorized

`u = np.zeros((n,1))`
`for i in range(n):`
`  u[i] = math.exp(v[i])`
	`u = np.exp(v)`

Some Numpy vectorized functions are:

`np.exp(v)`	`np.abs(v)`	`v**2`
`np.log(v)`	`np.maximum(v,0)`	`1/v`

$J = 0$, ~~$dw_1 = 0$, $dw_2 = 0$~~, $db = 0$ $\quad dw = np.zeros((n_x, 1))$

→ for $i = 1$ to m:

$z^{(i)} = w^T x^{(i)} + b$

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

$J \mathrel{+}= -[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$

$dz^{(i)} = a^{(i)}(1 - a^{(i)})$

for $j = 1, \ldots, n_x$: $dw_j \mathrel{+}= \ldots$

~~$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$~~ $\quad n_x = 2$ $\quad dw \mathrel{+}= x^{(i)} dz^{(i)}$

~~$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$~~

$db \mathrel{+}= dz^{(i)}$

$J = J/m$, ~~$dw_1 = dw_1/m$, $dw_2 = dw_2/m$~~, $db = db/m$

$dw /= m$.

Andrew Ng

Screen clipping taken: 2018-06-26 20:29

VECTORIZING LOGISTIC REGRESSION FORWARD PASS

Normally we would have to compute the following for each training example

$$z^{(1)} = w^T x^{(1)} + b \longrightarrow a^{(1)} = \sigma(z^{(1)})$$

$$z^{(2)} = w^T x^{(2)} + b \longrightarrow a^{(2)} = \sigma(z^{(2)})$$

$$z^{(3)} = w^T x^{(3)} + b \longrightarrow a^{(3)} = \sigma(z^{(3)})$$

To vectorize it we will use matrix multiplication:

$$Z = [z^{(1)}, z^{(2)}, \ldots z^{(m)}]$$

$w^T = [w_1, w_2, w_3 \ldots w_{n_x}]$, where n_x is the number of features

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

n_x features

m examples

where $X \in \mathbb{R}^{n_x \times m}$

$$Z = w^T X + \underbrace{[b \dots b]}_{m} = [w^T x^{(1)} + b, w^T x^{(2)} + b, \dots w^T x^{(m)} + b]$$

$Z := np.dot(w.T, X) + b$ ← b will automatically be cast as a $[1 \times m]$ vector for this operation (see "Broadcasting" in Python)

vectorized

To calculate the a (aka $\hat{y}$):

$$A = [a^{(1)}, a^{(2)} \dots a^{(m)}] = \sigma(Z)$$

VECTORIZING LOGISTIC REGRESSION BACKWARD PASS

$$dz^{(1)} = a^{(1)} - y^{(1)}$$
$$dz^{(2)} = a^{(2)} - y^{(2)}$$
$$dz^{(3)} = a^{(3)} - y^{(3)}$$
$$\vdots$$

Let's define:

$$dZ = [\underbrace{dz^{(1)}, dz^{(2)}, \dots dz^{(m)}}_{\text{m examples}}]$$

then:

$$A = [a^{(1)} \dots a^{(m)}]$$

$$Y = [y^{(1)} \dots y^{(m)}]$$

$$dZ = A - Y = [a^{(1)} - y^{(1)}, a^{(2)} - y^{(2)}, \dots]$$

To calculate db

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$$

$$= \frac{1}{m} * np.sum(dZ)$$

vectorized

$$dw = \frac{1}{m} X \cdot dZ^T$$

$$= \frac{1}{m} \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \cdot \begin{bmatrix} dz^{(1)} \\ dz^{(2)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$

$$= \frac{1}{m} \left[x^{(1)} dz^{(1)} + \ldots + x^{(m)} dz^{(m)} \right]$$ an $m \times 1$ vector

Vectorized Gradient Descent

Friday, 06 July, 2018 20:49

GRADIENT DESCENT ON M-EXAMPLES VECTORIZED

for step in range (1000): # of steps to iterate

$$Z_1 := W^T X + b$$
$$:= np.dot(W.T, X) + b$$
$$A := \sigma(Z_1)$$
$$dZ_1 := A - Y$$
$$dW := \frac{1}{m} X dZ_1^T$$
$$db := \frac{1}{m} np.sum(dZ_1)$$
$$W := W - \alpha dW$$
$$b := b - \alpha db$$

Why Sigma as cost function?

Friday, 06 July, 2018 20:53

WHY σ AS COST FNC.?

$\hat{y} = \sigma(w^T x + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$

$\hat{y} = p(y=1 \mid x)$ probability of $y=1$ given a set of input features x

If $y=1$: $p(y|x) = \hat{y}$

$y=0$: $p(y|x) = 1-\hat{y}$

These two equations above can be represented by

$p(y|x) = \hat{y}^y \cdot (1-\hat{y})^{(1-y)}$

→ $y=1 \Rightarrow p(y|x) = \hat{y}^1 \cdot (1-\hat{y})^0 = \hat{y}$

→ $y=0 \Rightarrow p(y|x) = \hat{y}^0 \cdot (1-\hat{y})^1 = 1-\hat{y}$

$\log p(y|x) = \log(\hat{y}^y (1-\hat{y})^{1-y})$

$= y \log(\hat{y}) + (1-y)\log(1-\hat{y})$

$= -\mathcal{L}(\hat{y}, y)$ ↓ Negative of the loss function.

Negative because when training an algorithm we want to make probabilities large, but the loss function should be minimized in logistic regression. So minimizing the loss function corresponds to maximizing the log of the probability.

Cost Function over all training set

Friday, 06 July, 2018 20:54

HOW DOES THE COST FUNCTION OVER ALL M EXAMPLES (ENTIRE TRAINING SET) LOOK LIKE?

$$p(\text{labels in training set}) = \prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)})$$

assuming the training examples are drawn independently (i.e. are drawn identically, are drawn independently distributed)

So if we would like to carry out maximum likelyhood estimation then we want to find the parameters that maximizes the chance the observation is in the training set.

Maximizing $p(\text{labels in training set})$ = maximizing the $\log p(\text{labels in training set})$

$$\log[p(\text{labels in training set})] = \log\left(\prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)})\right)$$

$$= \sum_{i=1}^{m} \underbrace{\log[p(y^{(i)} \mid x^{(i)})]}_{-\mathcal{L}(\hat{y}^{(i)}, y^{(i)})}$$

The principle of maximum likelihood estimation which means to choose the parameters of maximizing $\log(p(\dots))$

$$\Rightarrow \text{maximize} - \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

because we want to minimize the cost we get rid of the "$-$"

because we want to minimize the cost we get rid of the "−" sign and we also divide by m to better scale our quantities (for convenience) and we get the cost function

$$\text{Cost } J(w,b) = \frac{1}{m}\sum \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

So by minimizing $J(w,b)$ we are carrying out maximum likelihood estimation w/ the logistic model under the assumption that our training examples were iid (identically independently distributed).

! Type of Vector Multiplication

Thursday, 28 June, 2018 23:00

Given:

$$W = \begin{bmatrix} | & & | \\ x_{n,1} & \cdots & x_{n,m} \\ | & & | \end{bmatrix} \; ; \; y = [y_1, \ldots y_m] \; ; \quad k = [k_1, \ldots k_m]$$

"MATRIX MULTIPLICATION" (DOT PRODUCT) Inner Product

$$np.dot(W, y^T) = \begin{bmatrix} x_{1,1} \cdot y_1 + \ldots x_{1,m} \cdot y_m \\ \vdots \\ x_{n,1} \cdot y_1 + \ldots x_{n,m} \cdot y_m \end{bmatrix}$$

- dimension wise: $A[n,m] \times B[p,r] = C[n,r]$

OUTER PRODUCT

$$np.outer(y,k) = \begin{bmatrix} y_1 \cdot k_1 & y_1 \cdot k_2 & \cdots & y_1 \cdot k_m \\ \vdots & & & \\ y_m \cdot k_1 & & \cdots & y_m \cdot k_m \end{bmatrix}$$

ELEMENT-WISE MULTIPLICATION

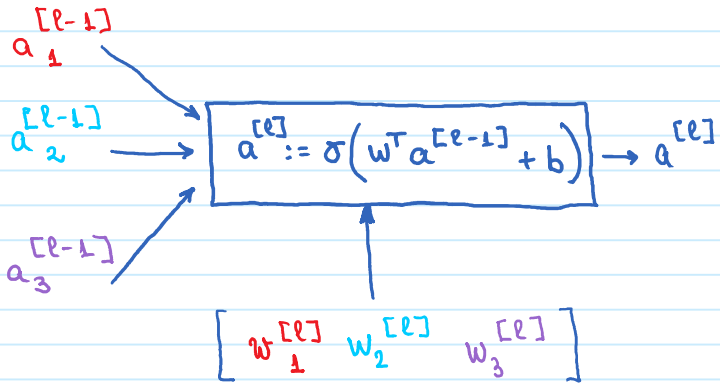
$$y * k = [y_1 \cdot k_1 \quad y_2 k_2 \quad \cdots \quad y_m k_m]$$

! MII Summary

Monday, 20 August, 2018 23:36

Given a Neural Network with L total layers on layer l:

Forward propagation



For the 1st layer ($l=1$) $a^{[l-1]} = x$ (input features from training example)

To the last layer ($l=L$) $a^{[l]} = \hat{y}$ (predicted output)

Given a training set of m examples (x, y) we use Gradient Descent to find the vector w and scalar b such that $J(w, b, x, y)$ is minimized.

$$g\left(m \left\{ \underbrace{\begin{bmatrix} x_1^{(1)} & | & & | \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} \\ \vdots & \vdots & & \vdots \\ x_n^{(1)} & | & & | \end{bmatrix}}_{X} \cdot \underbrace{[w_1 \; w_2 \cdots w_n]^T}_{w} + b \right) = \hat{y}$$

g is known as the activation function and it must be non-linear.

i.e σ, tanh, ReLU

$$J() = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$$ (average loss accross all examples, for a given definition of loss $\mathcal{L}$)

OBSERVATIONS:

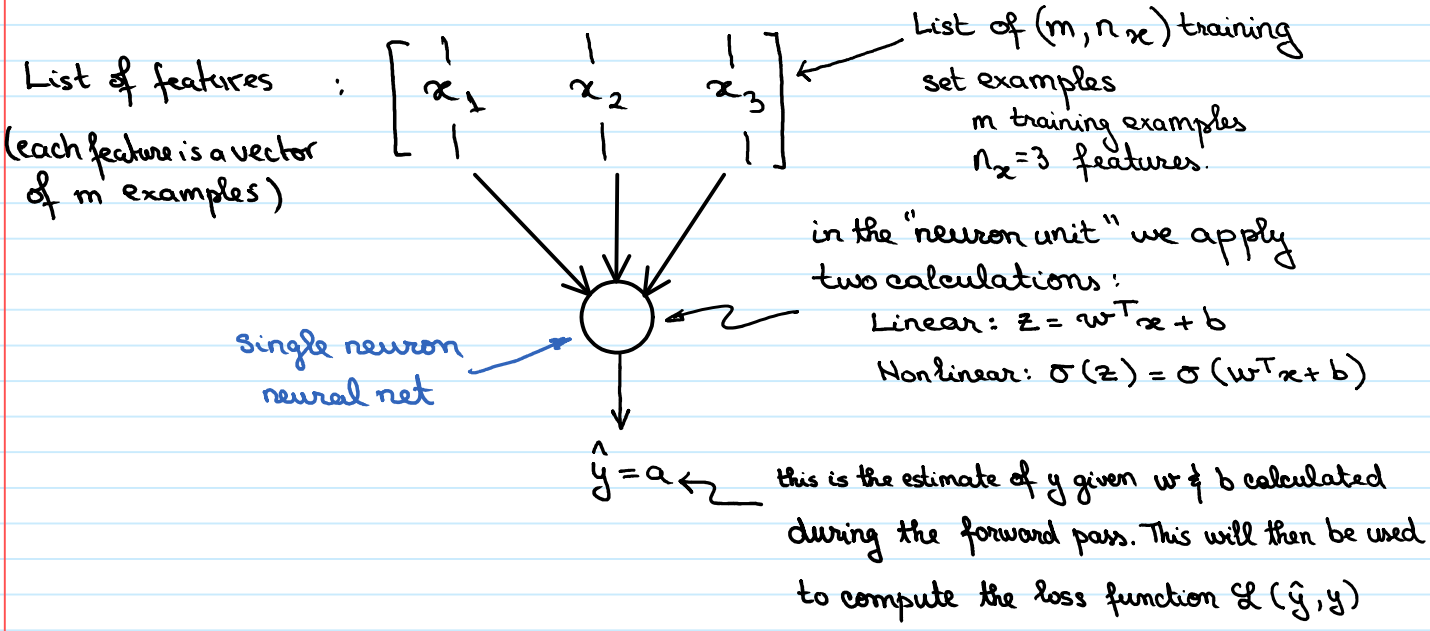
- Each value in vector W is associated with each feature in the training data set
- W_1 is derived by running Gradient Descent on all m examples of feature 1 (similarly for the other weights)
- The same scalar b applies to all m training examples and all features.

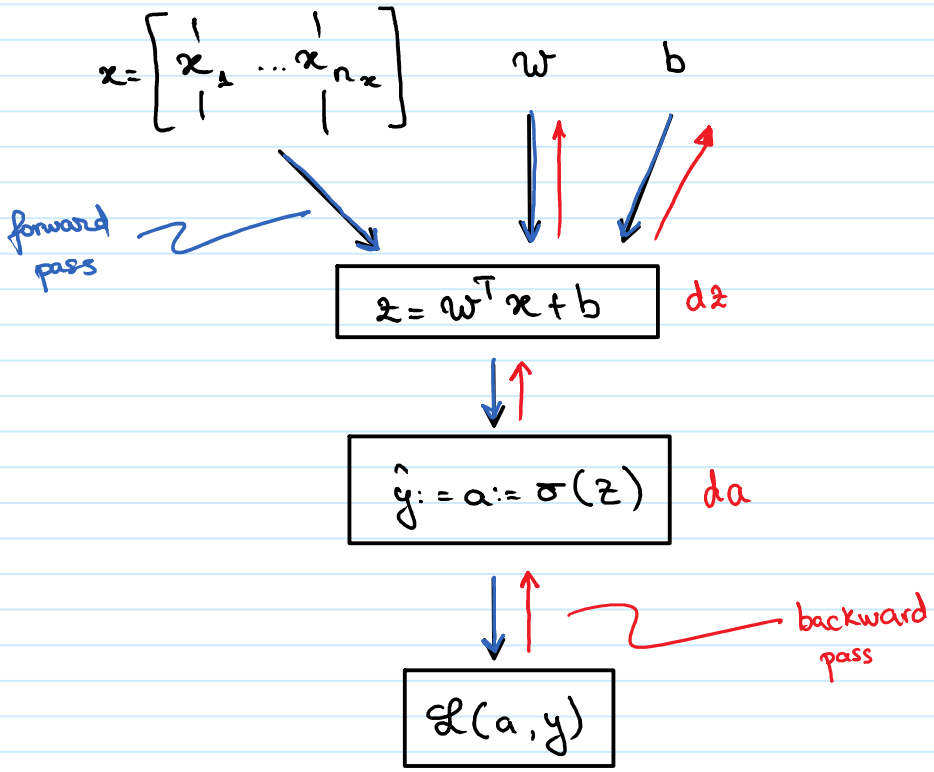
Week 3 Shallow NNs

Sunday, 01 July, 2018 13:05

Overview of NN implementation

Friday, 06 July, 2018 00:02





One forward pass outputs m × $\hat{y}$ predictions which are then used to calculate the cost function:

$$J = \frac{1}{m} \sum_{i}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

$$J = \frac{1}{m} \sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

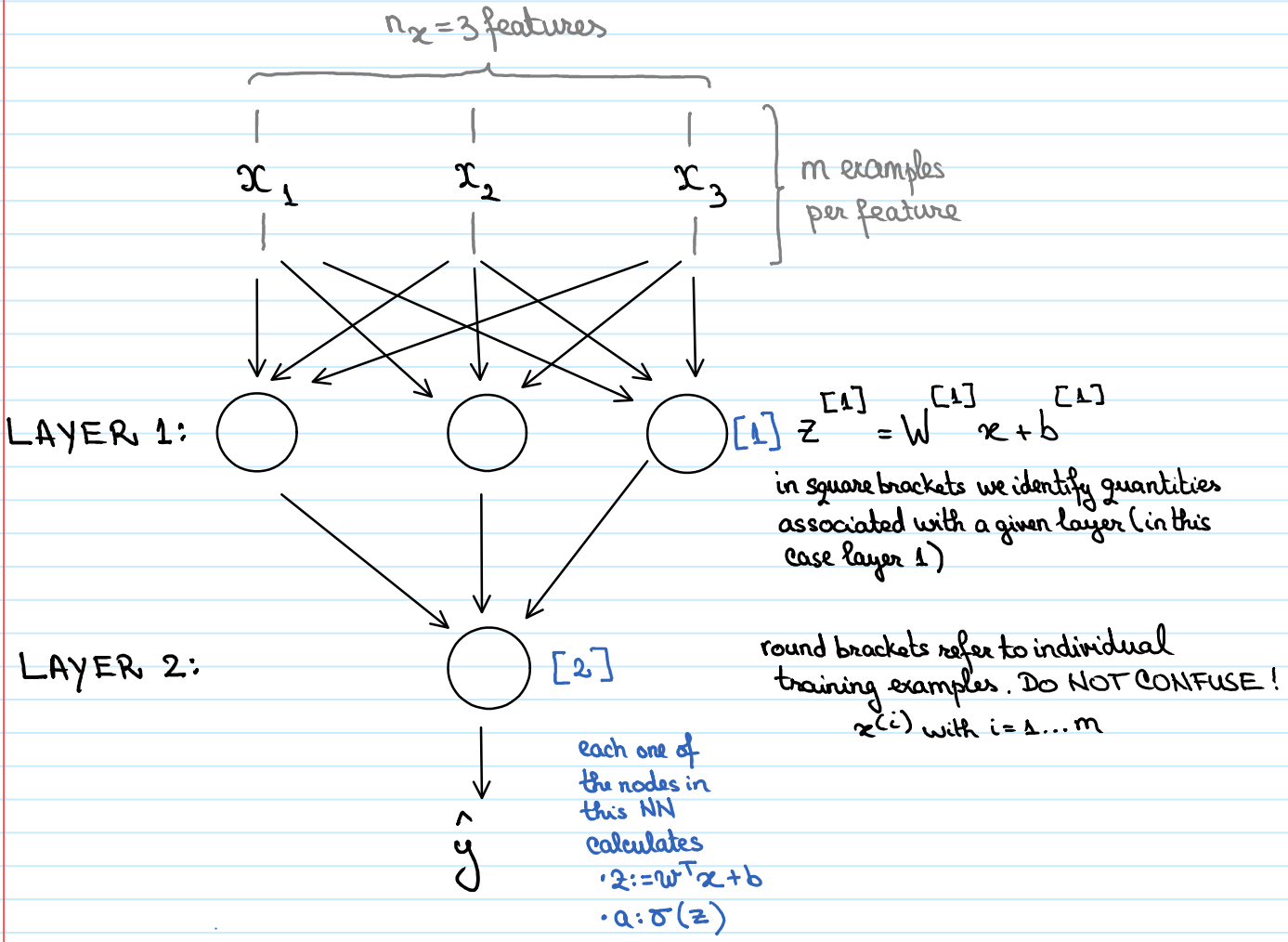
Then in the backward pass the cost function J is used to determine da.

da is a vector of gradients (one value per input feature) which is used to determine dw (the gradient by which we reduce the weights to minimize J).

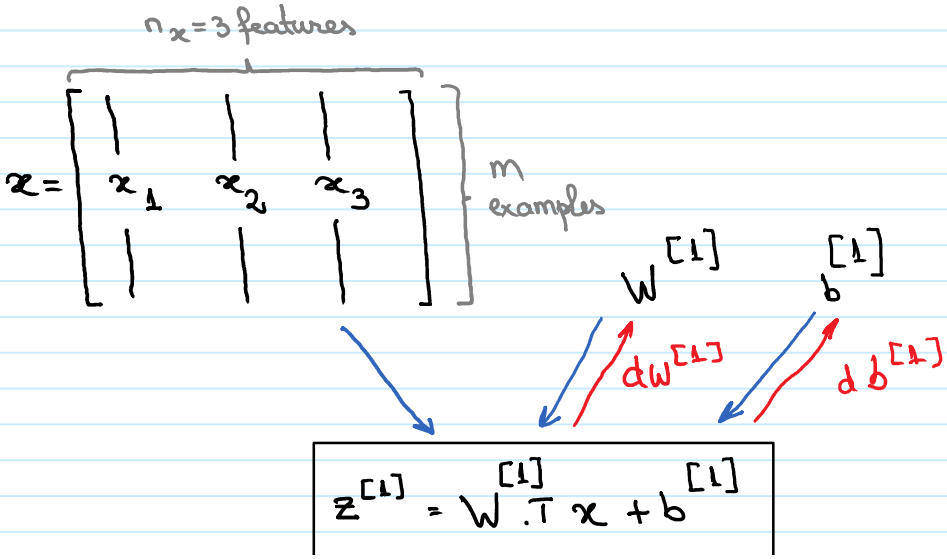
We then repeat the forward & backward pass until we converge on a minimum cost J

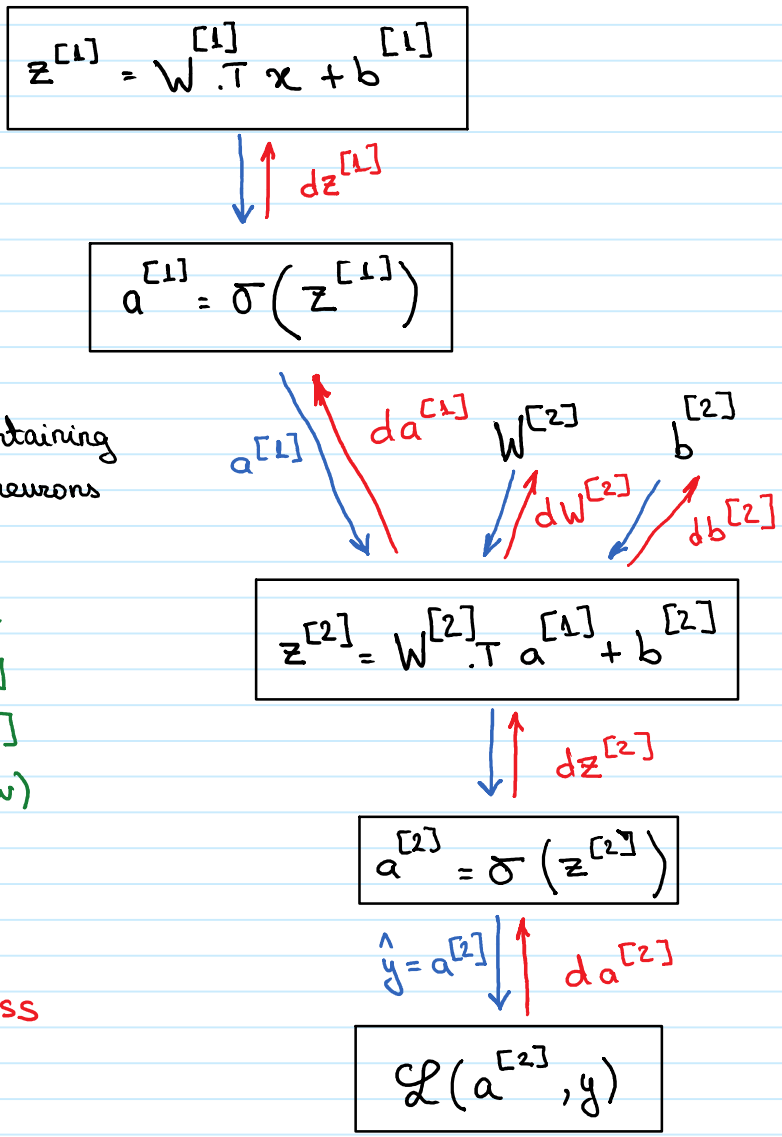
! A more general neural net

Friday, 06 July, 2018 00:02



DETAILED DESRIPTION





Q: Are $W^{[1]}$ & $b^{[1]}$ containing parameters for all neurons in layer 1?

A: Yes. In this case:

W.shape = [3,3]

b.shape = [3,1]

(more details below)

→ FWD PASS

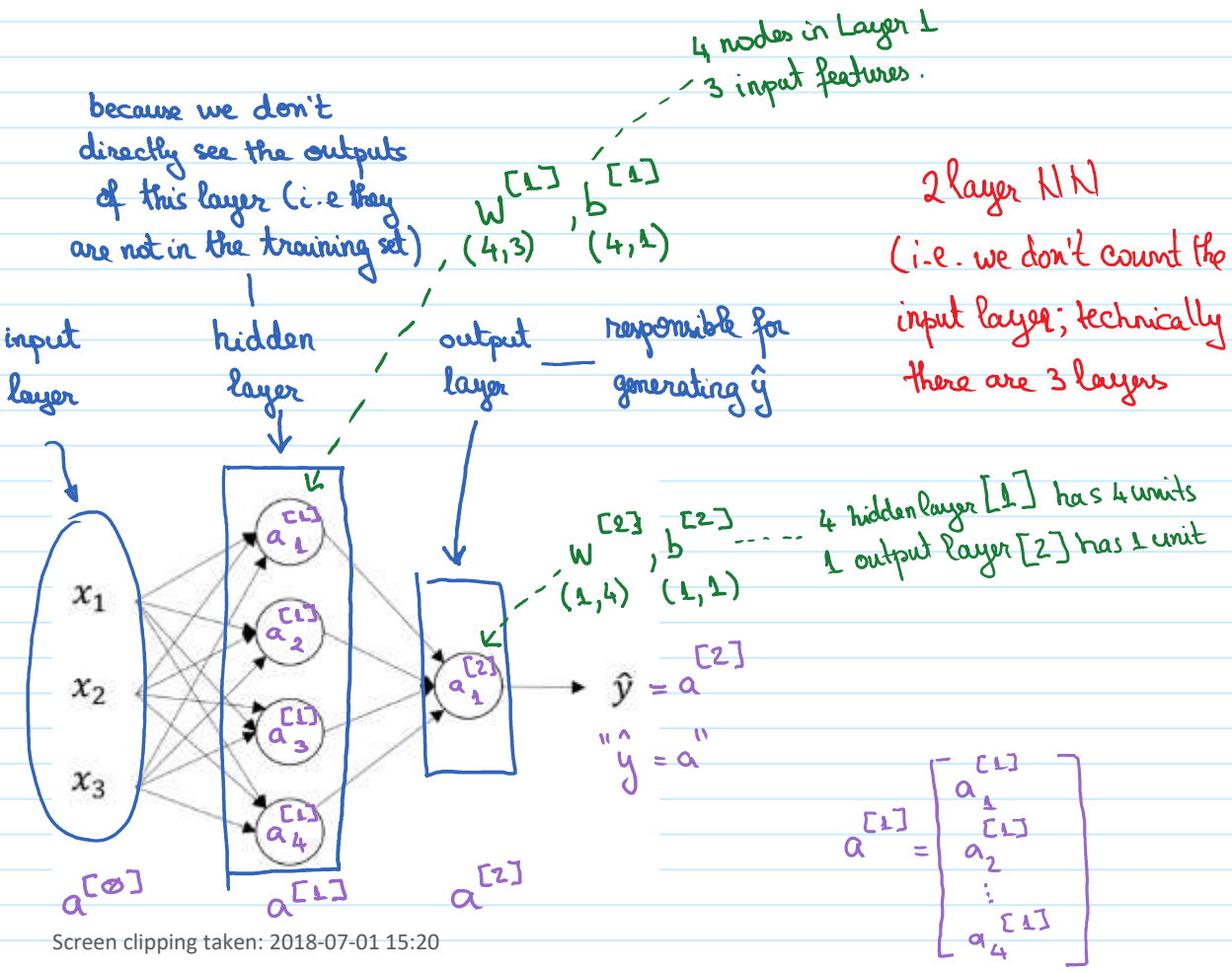
← BACKWARD PASS

Summary:

A neural net w/ 2 layers takes a logistic regression and repeats it twice.

Neural Network Representation

Friday, 06 July, 2018 00:02



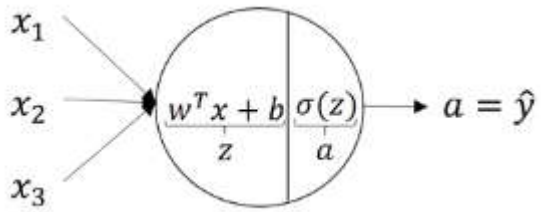
Screen clipping taken: 2018-07-01 15:20

$a^{[0]} := X$ (the input features)

where "a" stands for activations → it refers to the values different layers of the NN are passing on to the next layer

Computing NN output for a single example

Monday, 02 July, 2018 15:00



A node in a NN computes two values

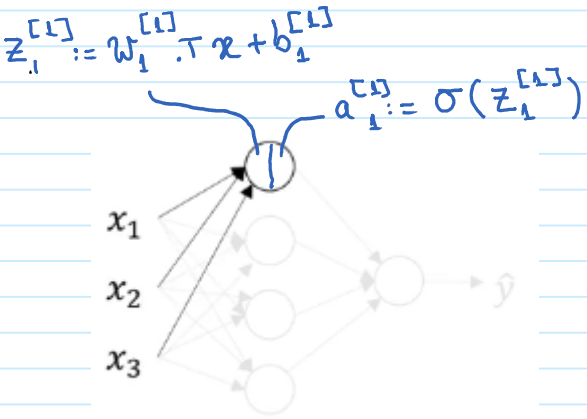
$z := w^T x + b$

$a := \sigma(z)$

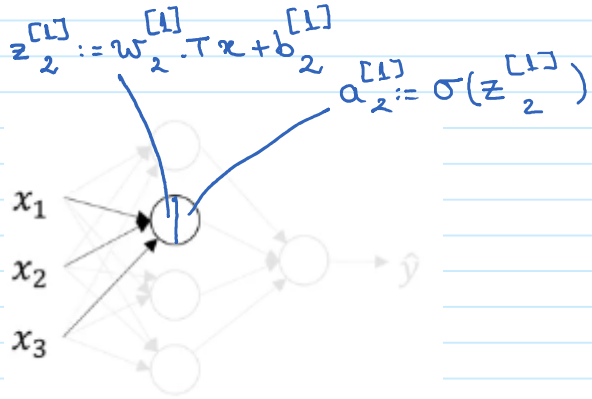
Screen clipping taken: 2018-07-02 14:51

$[l]$ ← layer # in NN

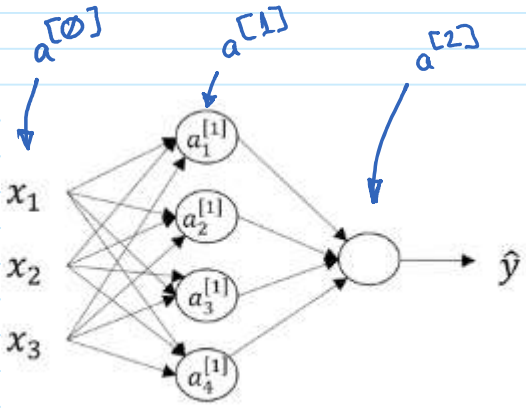
a_i ← node # in layer l



Screen clipping taken: 2018-07-02 15:16



Screen clipping taken: 2018-07-02 15:23



Screen clipping taken: 2018-07-02 15:26

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

Screen clipping taken: 2018-07-02 15:27

We want to compute z in vectorized form.

$$\underbrace{\begin{bmatrix} \text{—} & w_1^{[1]T} & \text{—} \\ \text{—} & w_2^{[1]T} & \text{—} \\ \text{—} & w_3^{[1]T} & \text{—} \\ \text{—} & w_4^{[1]T} & \text{—} \end{bmatrix}}_{W^{[1]}\ (4,3)} \times \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{(3,1)} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}}_{b^{[1]}\ (4,1)} = \underbrace{\begin{bmatrix} w_1^{[1]T}x + b_1^{[1]} \\ w_2^{[1]T}x + b_2^{[1]} \\ w_3^{[1]T}x + b_3^{[1]} \\ w_4^{[1]T}x + b_4^{[1]} \end{bmatrix}}_{(4,1)} = \underbrace{\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}}_{z^{[1]}\ (4,1)}$$

Think about it as having 4 logistic regressions (one per node) for each of the 3 inputs and we stack them together

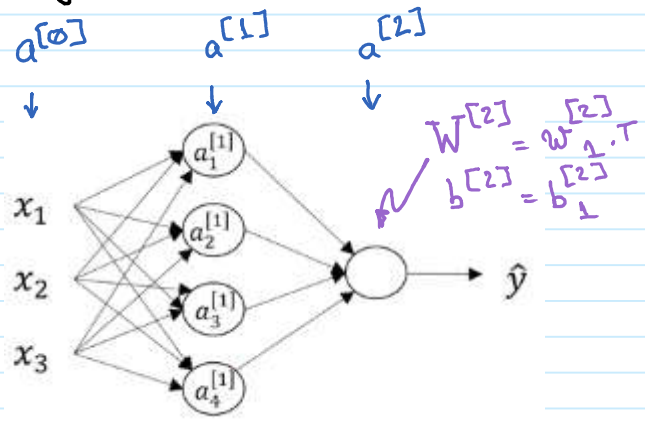
→ we end up with a $(4,3)$ matrix

Rule of thumb:
The vertical dimension corresponds to a layer in a network.

$$a^{[1]} := \begin{bmatrix} a_1^{[1]} \\ \vdots \\ a_4^{[1]} \end{bmatrix} := \sigma(z^{[1]})$$

Now if we replace the input layer (i.e. $[x_1, x_2, x_3]$) w/ $a^{[0]}$ and the output layer (i.e. $\hat{y}$) w/ $a^{[2]}$

We get:



Screen clipping taken: 2018-07-02 15:53

$$\underset{(4,1)}{z^{[1]}} := \underset{(4,3)}{W^{[1]}} \cdot \underset{(3,1)}{a^{[0]}} + \underset{(4,1)}{b^{[1]}}$$

$$\underset{(4,1)}{a^{[1]}} := \sigma(\underset{(4,1)}{z^{[1]}})$$

$$\underset{(1,1)}{z^{[2]}} := \underset{(1,4)}{W^{[2]}} \cdot \underset{(4,1)}{a^{[1]}} + \underset{(1,1)}{b^{[2]}}$$

$$a^{[2]} := \sigma(z^{[2]})$$

(1, 1) (1, 1)

So to calculate $\hat{y}$ (predicted output) in vectorized form for a single example (i.e. $a^{[0]} = [x_1 \dots x_m]$) we need to calculate the following equations:

Logistic Regression 1 layer & 1 node	Neural Net 2 layers w/ 3 & 1 nodes respetively
$z := w^T x + b$ $\hat{y} := a := \sigma(z)$	$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$ $a^{[1]} := \sigma(z^{[1]})$ $z^{[2]} := W^{[2]} a^{[1]} + b^{[2]}$ $\hat{y} := a^{[2]} := \sigma(z^{[2]})$

Computing a NN output for multiple examples

Thursday, 05 July, 2018 23:58

For one example we computed the predicted output as:

$$x \longrightarrow a^{[2]} = \hat{y}$$

For multiple examples we'll calculate the predicted outputs as:

$$x^{(1)} \longrightarrow a^{[2](1)} = \hat{y}^{(1)}$$
$$x^{(2)} \longrightarrow a^{[2](2)} = \hat{y}^{(2)}$$
$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$$
$$x^{(m)} \longrightarrow a^{[2](m)} = \hat{y}^{(m)}$$

in this case $a^{[2](i)}$: layer 2, example set i

where m is the # of examples used.

A non-vectorized form of this (i.e. using a for loop) is:

for $i = 1$ to m:

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

Note: the same weights $W^{[1]}, b^{[1]}$ and $W^{[2]}, b^{[2]}$ are applied to all examples during a forward pass

THIS IS NOT WHAT HAPPENS IN A LIVING ORGANISM!

$$Z^{[1]} := W^{[1]} \underbrace{\begin{bmatrix} | & & | \\ x^{(1)} & \cdots & x^{(m)} \\ | & & | \end{bmatrix}^T}_{A^{[0]}} + b^{[1]}$$

$Z^{[1]}$: $(4, n_x)$; $W^{[1]}$: $(4, n_x)$; $A^{[0]}$: (n_x, m); $b^{[1]}$: $(1, n_x)$

$$A^{[1]} := \sigma(Z^{[1]})$$

$A = \sigma(Z)$

$$Z^{[2]} := W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} := \sigma(Z^{[2]})$$

$$Z^{[1]} := \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$$

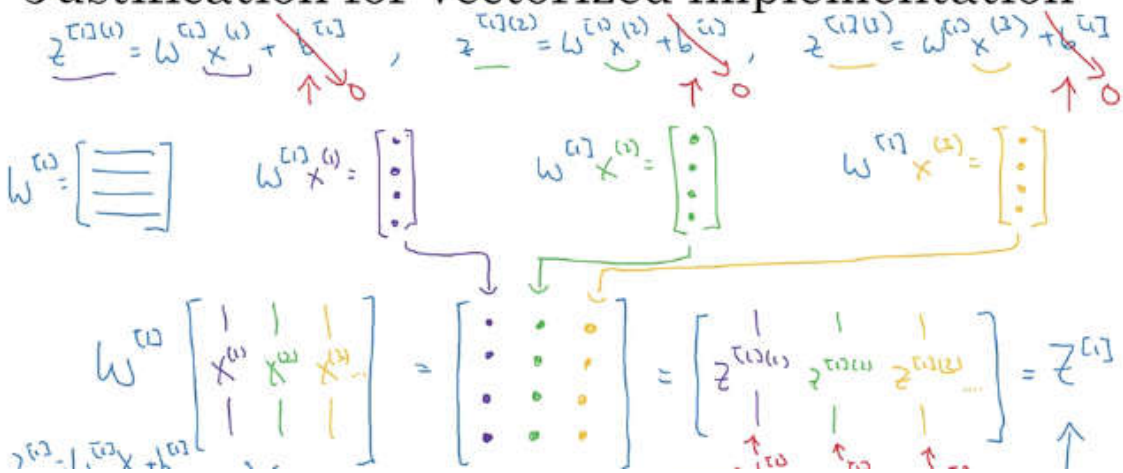
$$A^{[1]} := \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

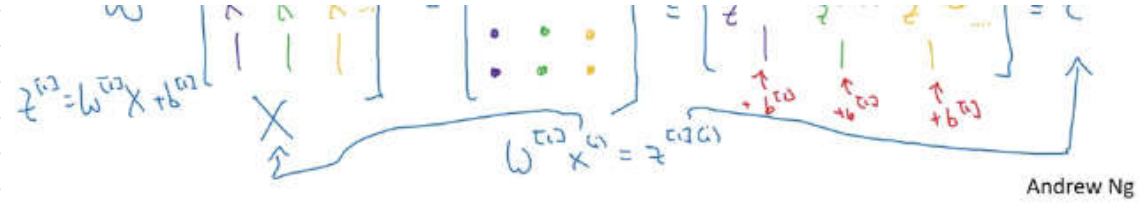
value of the activation of the first training example of the first node of layer 1

index into each node on layer 1 (scan through hidden units)

scan through the training examples

Justification for vectorized implementation



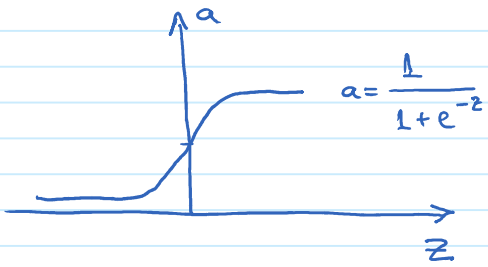


Screen clipping taken: 2018-07-02 22:08

Activation Functions

Thursday, 05 July, 2018 23:57

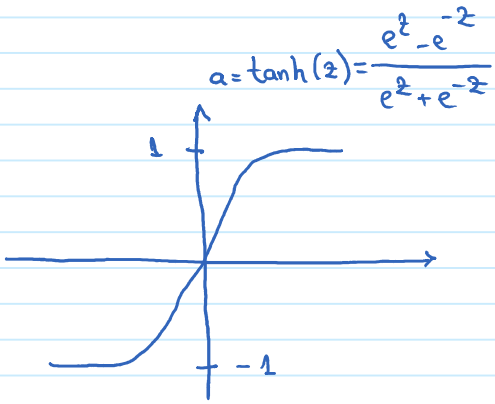
So far we have been using the σ function $\sigma(z) = \frac{1}{1+e^{-z}}$ as the activation function for each node.



Given a more general form of the forward pass is:

$$z^{[1]} = w^{[1]} x + b^{[1]}$$
$$a^{[1]} = g(z^{[1]})$$
$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = g(z^{[2]})$$

Where g can be a non-linear function that is not the σ.



tanh is a shifted and scaled version of the σ function

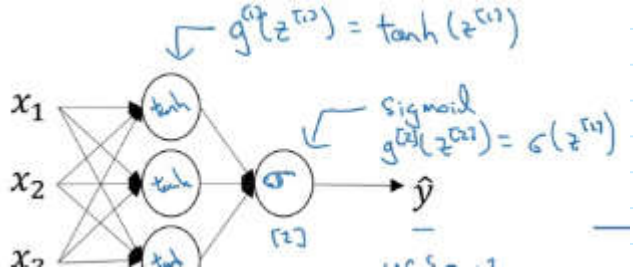
If we use $g(z) = \tanh(z)$ as the activation function it almost always works better than the σ

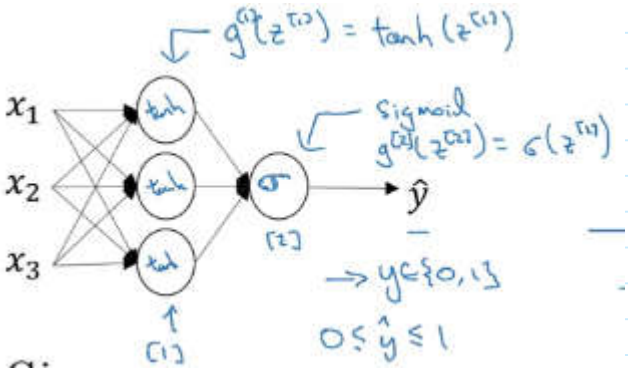
→ that is because the mean activation is closer to 0 as the function takes values between $(-1; 1)$ so it helps "center" your data around 0.

→ centering makes the learning process for the next layer easier.

EXCEPTION: On the output layer of a binary classification problem the σ function is better as the output $\hat{y}$ is between 0 & 1

We can use different activation functions for the different layers of the NN



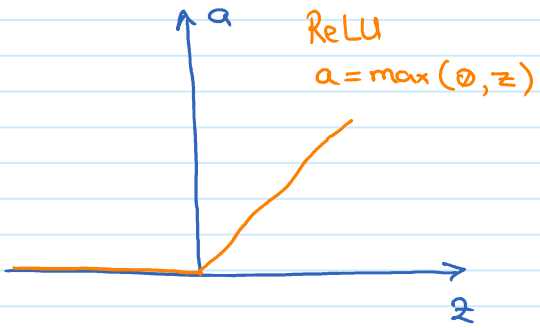


Screen clipping taken: 2018-07-02 22:37

Flat Activation Function Problem

Thursday, 05 July, 2018 23:55

When the derivative of the activation function is close to 0 learning occurs very slowly (slows down gradient descent)



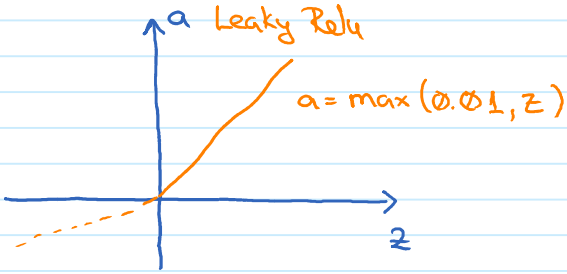
ReLU to the Rescue!

Rectified Linear Unit

Note: ReLU is not differentiable when $z = 0$; but in practice we'll almost never encounter $z = 0.000000\ldots$

One disadvantage of ReLU is that the derivative is 0 for $z < 0$.

Some use the leaky ReLU, though in practice quite few nodes will have positive z so learning will still occur quite fast.



use 0.01 b/c it empirically seems to do well (but this can be modified)

σ : only use it for the output layer

tanh: try it some times

ReLU: use this most of the time

Evaluating the performance of different activation function in practice instead of always choosing the same "best" one ensures that data idiosyncrasies are adequately

dealt with and also hardens the architecture against evolutions of the algorithm.

Also evaluate: network topology, initialization values, etc

Why non-linear activation functions?

Thursday, 05 July, 2018 23:20

Why not set $g(z) = z$ (the linear activation function) ?

In this case:

$$a^{[1]} = z^{[1]} = w^{[1]}x + b^{[1]}$$

$$a^{[2]} = z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$$

$$= w^{[2]}\left(w^{[1]}x + b^{[1]}\right) + b^{[2]}$$

$$= \underbrace{\left(w^{[2]}w^{[1]}\right)}_{w'}x + \underbrace{\left(w^{[2]}b^{[1]} + b^{[2]}\right)}_{b'}$$

$$= w'x + b'$$

So it turns out we are computing a linear function of the inputs

→ there's a theorem showing that if we are computing a linear func. of the inputs a network with many layers can be replaced by a network w/ only a single layer so the hidden layers don't do anything (they are not more expressive than a linear function of the inputs).

ALL B/C the composition of a linear func is itself a linear func.

EXCEPTION: If we do machine learning on a regression problem (i.e. housing prices) so $y \in \mathbb{R}$ i.e. $y = \$0 \ldots \$1,000,000$ the output layer should use a linear activation function (so that the predicted output

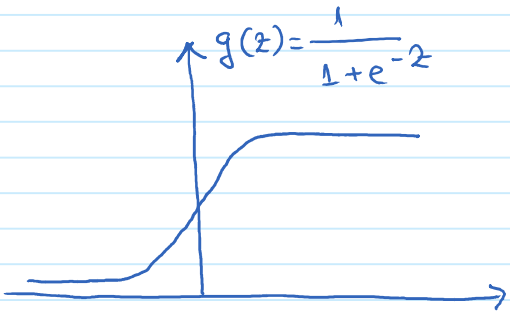
can go from $\emptyset$ to ∞). But the hidden layers should still NOT USE a linear activation function.

Derivatives of Activation Functions

Wednesday, 04 July, 2018 22:51

- Sigmoid activation function

$$a = g(z) = \frac{1}{1+e^{-z}}$$



$$g'(z) = \frac{d}{dz} g(z) = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right)$$

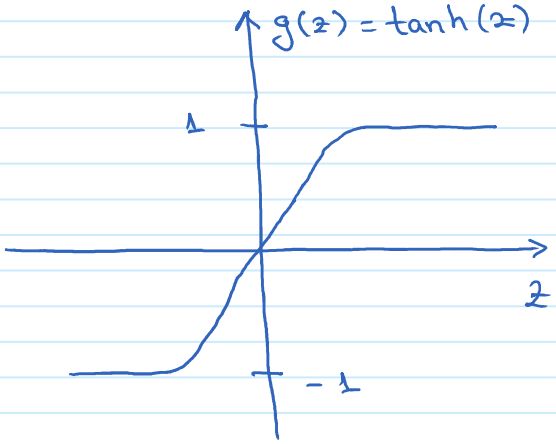
$$= g(z)\left(1 - g(z)\right)$$

$$\boxed{g'(z) = a\,(1-a)}$$

- Tanh activation function

$$a = g(z) = \tanh(z)$$

$$= \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



$$g'(z) = \frac{d}{dz} g(z) = 1 - (\tanh(z))^2$$

$$\boxed{g'(z) = 1 - a^2}$$

- ReLU & Leaky ReLU

$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & z < 0 \end{cases}$$

$$g(z) = \max(0.01, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0.01, & z < 0 \end{cases}$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & z < 0 \\ \text{undefined}, & z = 0 \end{cases}$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0.01, & z < 0 \end{cases}$$

We can set $g'(z) = 1$ for $z = 0$

↳ g' becomes a subgradient of the optimization func.

Gradient descent NN

Thursday, 05 July, 2018 22:41

Goal of this section is to determine what parameter values to use when running gradient descent

Gradient descent consists of repeating 3 stages:

0) INITIALIZE PARAMETERS

REPEAT {
1) FORWARD PROPAGATION (compute predictions)
2) BACKWARD PROPAGATION (compute derivatives)
3) UPDATE WEIGHTS
}

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$

$(n^{[1]}, n^{[0]})$ $(n^{[2]}, 1)$ $(n^{[2]}, n^{[1]})$ $(n^{[2]}, 1)$

$n_x = n^{[0]}, n^{[1]}, n^{[2]} = 1$

$n^{[0]}$: input features; $n^{[1]}$: hidden units; $n^{[2]}$: output units

Cost function (assume binary classification):

$$J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} \sum \mathcal{L}(\hat{y}, y)$$

$\hat{y} = a^{[2]}$

To train the NN we need to do Gradient Descent:

Important to initialize parameters randomly (instead of all 0s)

After we initialize each loop of Gradient Descent

Repeat {

* Compute predictions ($\hat{y}^{(i)}$ for $i = 1 \dots m$)
* Compute derivatives

$$dW^{[1]} = \frac{dJ}{dW^{[1]}}, \quad db^{[1]} = \frac{dJ}{db^{[1]}}, \dots$$

* Update weights:

$$W^{[1]} = W^{[1]} - \alpha\, dW^{[1]}$$

$$b^{[1]} = b^{[1]} - \alpha\, db^{[1]}$$

} until it looks like parameters are converging.

FORWARD PROPAGATION (compute predictions):

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

$$A^{[1]} = g(Z^{[1]})$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = g(Z^{[2]}) \text{ fa binnary classification } A^{[2]} = \sigma(Z^{[2]})$$

BACK PROPAGATION (compute derivatives):

$$dZ^{[2]} = A^{[2]} - Y$$

↑ assume we are doing binary classification so the activation function for the output layer is $\sigma(z) \rightarrow g'(z) = \sigma(z) - Y$

this is all vectorized accross examples

$$Y = [y^{(1)}, y^{(2)} \dots y^{(m)}]$$

$$dW^{[2]} = \frac{1}{\quad} \quad A^{]} .T$$

$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$$

↑ sum horizontally accross matrix

↑ prevents Python from outputting $(n^{[2]},)$ instead it outputs $(n^{[2]}, 1)$

$$dZ^{[1]} = \underbrace{W^{[2]}.T\, dZ^{[2]}}_{(n^{[1]}, m)} * \underbrace{g^{[1]'}(Z^{[1]})}_{(n^{[1]}, m)}$$

element wise product

↑ the derivative of whatever activation function we used fa the hidden layer

$$dW^{[1]} = \frac{1}{\quad} dZ^{[1]} X^T$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$$

$(n^{[1]}, 1)$

↑ we don't want a $(n^{[1]},)$ output

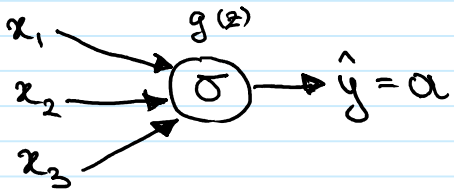
Backpropagation Intuition

Thursday, 05 July, 2018 22:55

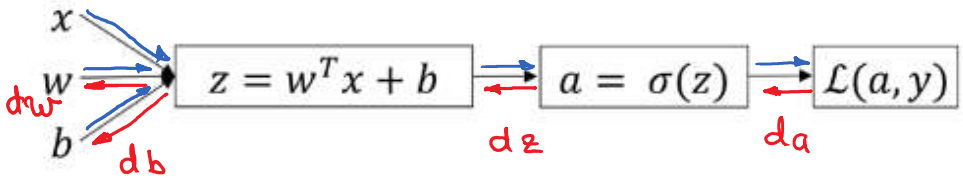
Goal: Develop an intuition of how the parameters used in backpropagation are derived.

We'll start from the (single node, single layer) Logistic Regression and progress to the (multi node, multilayer) Neural Net.

• LOGISTIC REGRESSION



Logistic regression



→ Forward Prop

← Back Prop

Screen clipping taken: 2018-07-05 22:38

$da = -\frac{y}{a} + \frac{1-y}{1-a}$ $\quad dz = a - y$ $\quad dw = dz \cdot x$ $\quad db = dz$

See below for derivations.

NOTE: We are applying a lot of chain rules (see COMPUTATION GRAPH in Week 2)

$$\mathcal{L}(a, y) = -y \log(a) - (1-y) \log(1-a)$$

↓ take derivative of $\mathcal{L}(a, y)$ wrt a

$$da = \frac{d}{da} \mathcal{L}(a, y)$$

$$\| \quad da = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$\| \quad da = -\frac{y}{a} + \frac{1-y}{1-a}$$

Then:

$$dz = \frac{d}{dz}\mathcal{L}(a,y)$$

(apply chain rule)

$$= \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{d}{dz} a = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{d}{dz} g(z) = \frac{\partial \mathcal{L}}{\partial a} \cdot g'(z) = da \cdot g'(z)$$

for logistic regression

$a = g(z) = \sigma(z) \Rightarrow \quad g'(z) = a\,(1-a)$

as per DERIVATIVES OF ACTIVATION FUNCTIONS

So:

$$dz = da \cdot g'(z)$$

$$= da \cdot a(1-a)$$

$$= \left(-\frac{y}{a} + \frac{1-y}{1-a}\right) \cdot a(1-a)$$

$$= -y(1-a) + a(1-y)$$

$$= -y + ya + a - ay$$

$$\| \quad dz = a - y$$

Then we compute

$$dw = \frac{d}{dw}\mathcal{L}(a,y)$$

$$= \underbrace{\frac{d}{da}\mathcal{L}(a,y) \cdot \frac{d}{dz} a}_{a-y} \cdot \frac{d}{dw} z$$

$$= (a-y) \cdot \frac{d}{dw} z$$

$z = w^T x + b$

$\Rightarrow \frac{d}{dw} z = x \quad \{ \frac{d}{db} z = 1$

|| $dw = (a-y)\cdot x$ or $dz \cdot x$

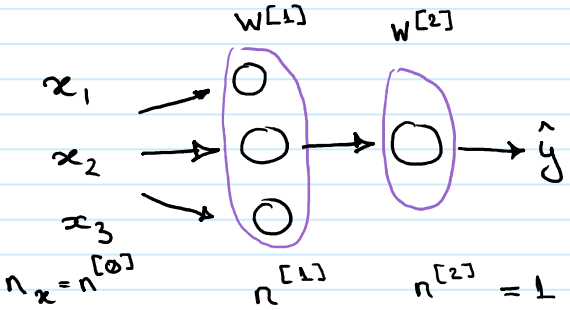
Similarly to dw

|| $db = dz$

2018-07-07 (Sat) @ 10:30

- NEURAL NETWORK 2 LAYER:

Equations for a single example:

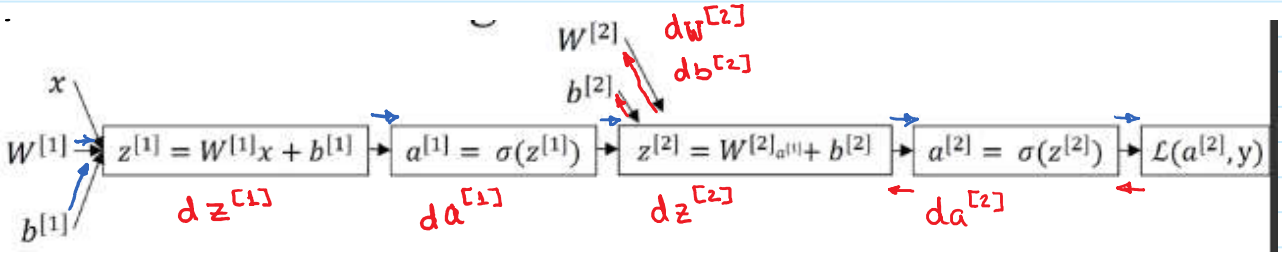


$W^{[2]} : (n^{[2]}, n^{[1]})$

$z^{[2]}, dz^{[2]} : (n^{[2]}, 1)$

$z^{[1]}, dz^{[1]} : (n^{[1]}, 1)$

Instead of having a single computational step (for a single layer) we have two steps one for each layer.



Screen clipping taken: 2018-07-07 10:55

Note: We don't need to take derivatives for the input x b/c for supervised learning the input x is fixed so we are not trying to optimize x.

|| $dz^{[2]} = a^{[2]} - y$

|| $dW^{[2]} = dz^{[2]} a^{[1]\,T}$ ←

In Linear Regression "$dW = dz \cdot x$"

In NN $a^{[1]}$ plays the role of x, and it's transposed b/c W is a row vector, whereas $dW^{[2]}$ is a column vector.

$W^{[2]} = [\_\_\_\_\_]$

$dW^{[2]} = dz^{[2]} a^{[1]T}$

$db^{[2]} = dz^{[2]}$

b/c $W^{[2]}$ is a row vector, whereas $dW^{[2]}$ is a column vector.

$W^{[2]} = [\quad\quad]$

and x was a column vector.

element wise multiplication

$$dz^{[1]} = (W^{[2]T} dz^{[2]}) * g^{[1]'}(z^{[1]})$$

$(n^{[1]}, 1)$ $(n^{[1]}, n^{[2]})$ $(n^{[2]}, 1)$ $(n^{[1]}, 1)$

☆ TIP: Ensuring the dimensions match up resolves many bugs in implementing back propagation

$dW^{[1]} = dz^{[1]} \cdot X^T$ ($X^T = a^{[0]T}$)

$db^{[1]} = dz^{[1]}$

Vectorized equations for multiple examples:

SINGLE EXAMPLE

$$dz^{[2]} = a^{[2]} - y$$

$$dW^{[2]} = dz^{[2]} a^{[1]T}$$

$$db^{[2]} = dz^{[2]}$$

$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$$

$(n^{[1]}, 1)$

$$dW^{[1]} = dz^{[1]} x^T$$

$$db^{[1]} = dz^{[1]}$$

Screen clipping taken: 2018-07-07 11:24

VECTORIZED IMPLEMENTATION

$$dZ^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis=1, keepdims=True)$$

element wise multiplication

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]})$$

$(n^{[1]}, m)$ $(n^{[1]}, m)$ $(n^{[1]}, m)$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis=1, keepdims=True)$$

For single example we have:

$$z^{[1]} = W^{[1]} x + b^{[1]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$

To vectorize this for multiple examples we stack the z in columns (one col. per example)

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1][m]} \\ | & | & & | \end{bmatrix}$$

and then we have

$$Z^{[1]} = W^{[1]} X + b$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

Random initialization of weights

Saturday, 07 July, 2018 11:41

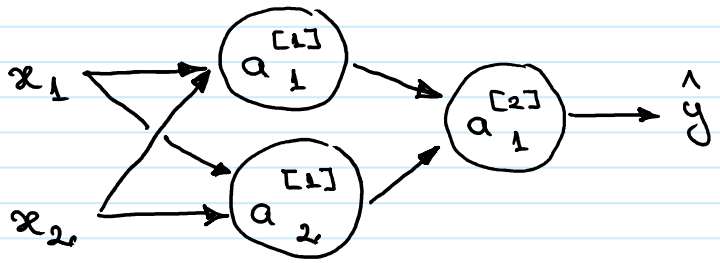
For NN it's important to initialize the weights randomly.

→ for logistic regression it is ok to initialize the weights to 0

→ for NN this won't work w/ gradient descent.

WHAT HAPPENS WHEN INITIALIZING WEIGHTS TO ZERO?

Note: initializing b to 0 is ok but $W = 0$ is a problem



$n^{[0]} = 2$ $n^{[1]} = 2$ $n^{[2]} = 1$

$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad W^{[2]} = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

$$a^{[1]}_1 = a^{[1]}_2 \Rightarrow dz^{[1]}_1 = dz^{[1]}_2$$

these units are identical (aka symmetric → they compute exactly the same function)

↓

eg induction: after every iteration of training they still compute the exact same function (ie both hidden units compute the same function → so there is no point of having more than one hidden

Proof:

Given $dW = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$

$$W^{[1]} = W^{[1]} - \alpha\, dW$$

$$W^{[1]} = \begin{bmatrix} \text{---} \\ \text{---} \end{bmatrix}$$

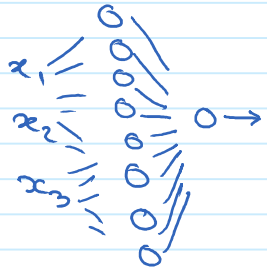
(i.e. the 1st row = 2nd row so still symmetric)

same function - so there is no point of having more than one hidden unit)

$W = \begin{bmatrix} \text{---} \\ \text{---} \end{bmatrix}$

(i.e. the 1st row = 2nd row so still symmetrical)

This argument works the same for networks w/ many hidden units → if they all are initialized to the same value (i.e. 0) they all compute the same function and can be replaced by a single unit.



RANDOM INITIALIZATION

$W^{[1]} = np.random.randn((2,2)) * 0.01$

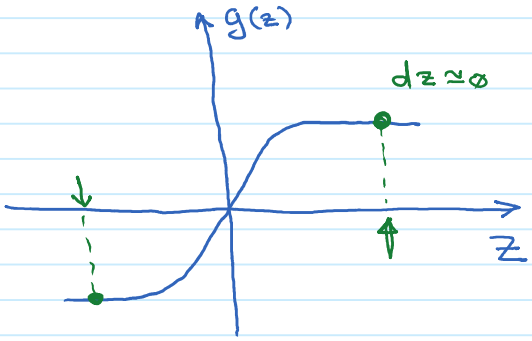
$b^{[1]} = np.zero((2,1))$

$W^{[2]} = np.random.randn((1,2))$

$b^{[2]} = 0$

Why 0.01? Why do we want to initialize the weights to small values? Why not 100 or 1000?

When we use a sigmoid output function (tanh or σ) if the weights are too large z is very large and we end up in the regions where the function is flat so the derivative $\simeq 0$ so gradient descent is slow and the learning is very slow



If we don't use sigmoids this is not as big of an issue.

Sometimes 0.01 is not ideal → for shallow networks it is OK (not many hidden units)

→ for deep networks there is better

Week 4 Deep NNs

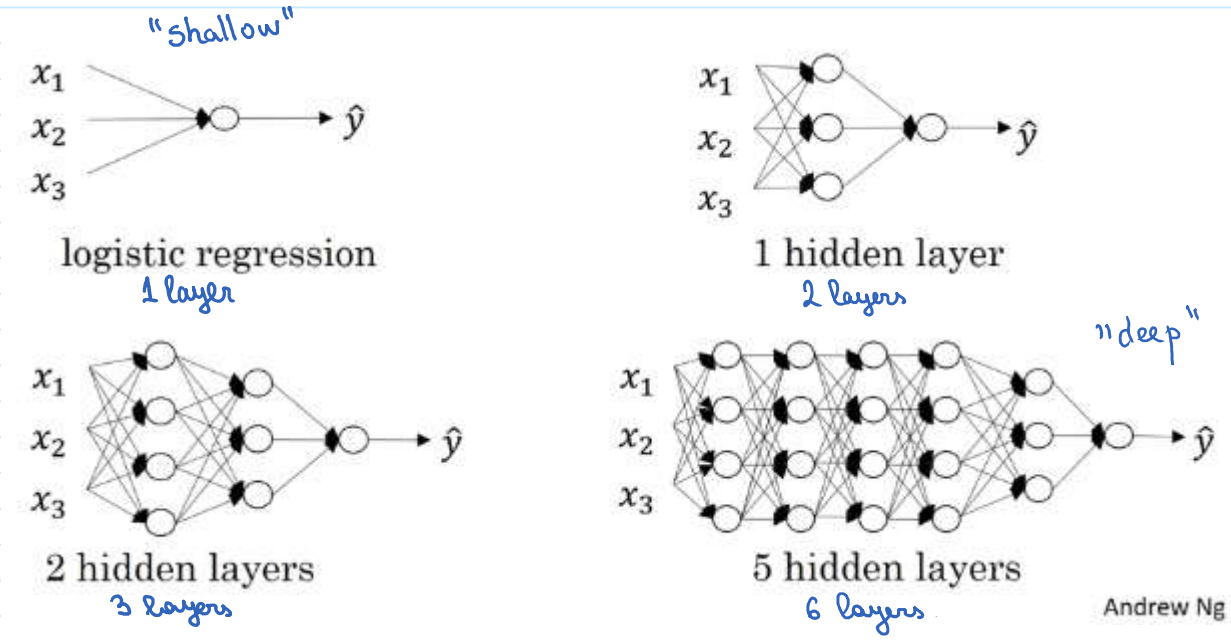
Monday, 09 July, 2018 20:55

DEEP L-LAYER NEURAL NETWORKS

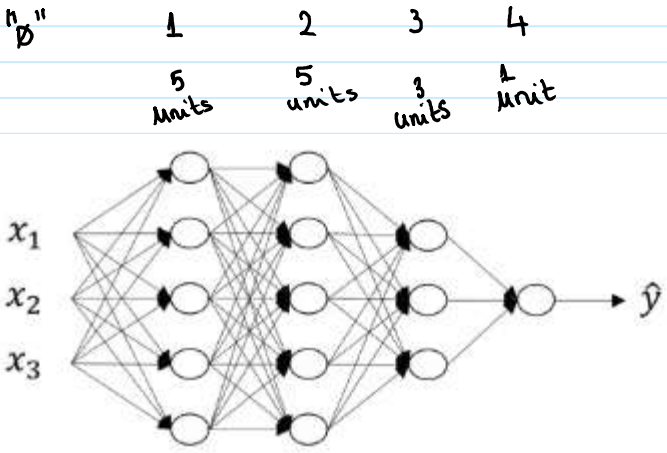
Deep NN nomenclature

Monday, 09 July, 2018 22:21

Nomenclature of Deep neural network



Screen clipping taken: 2018-07-09 22:08



Screen clipping taken: 2018-07-09 22:11

4 layer NN $\Rightarrow L = 4$

L: # of layers in the NN

$n^{[l]}$: # of units (nodes) in a layer l

$n^{[0]} = 3 = n_x$

$n^{[1]} = 5$

$n^{[2]} = 5$

$n^{[3]} = 3$

$n^{[4]} = 1 = n^{[L]}$

$a^{[l]}$: activations in layer l

$a^{[l]} = g^{[l]}(z^{[l]})$

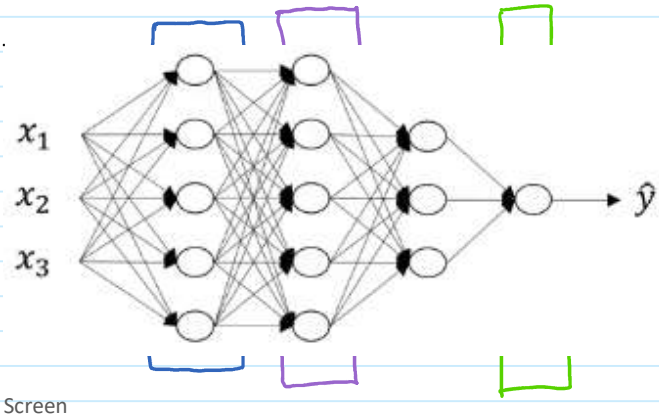
$W^{[l]}$, $b^{[l]}$ } weights used to compute $z^{[l]}$ in layer l

$X = a^{[0]}$ (the input features are the activations for layer 0)

$a^{[L]} = a^{[4]} = \hat{y}$

Forward Prop in deep NNs

Monday, 09 July, 2018 22:23



Screen clipping taken: 2018-07-09 22:24

General Forward Propagation Equation:

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]})$$

Example of FWD Prop on a single example x

$$z^{[1]} = W^{[1]} x + b^{[1]} \quad (a^{[0]} = x)$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$

$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$

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

$$\vdots$$

$$z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}$$

$$a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}$$

Vectorized form (for multiple training examples):

$$Z^{[1]} := W^{[1]} A^{[0]} + b^{[1]} \quad (A^{[0]} = X \text{ = training examples set})$$

$$A^{[1]} := g^{[1]}(Z^{[1]})$$

$$Z^{[2]} := W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} := g^{[2]}(Z^{[2]})$$

$$A^{[4]} := g^{[4]}(Z^{[4]}) = \hat{Y}$$

Layer 1 … Layer 4

for $l = 1$ to 4 there is no way to get rid of (vectorize away) the for loop to iterate through layers to calculate activations

$\hat{Y}$ = prediction for all training examples stacked horizontally

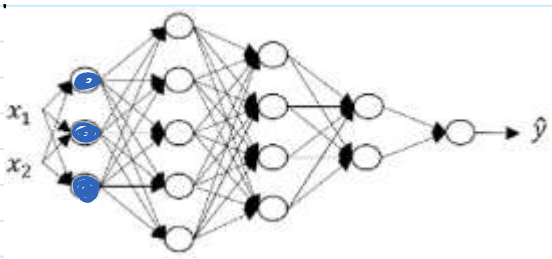
$$Z^{[n]} = \begin{bmatrix} | & | & & | \\ z^{[n](1)} & z^{[n](2)} & \cdots & z^{[n](m)} \\ | & | & & | \end{bmatrix}$$

$$Z^{[n]} = \begin{bmatrix} | & | & & | \\ z^{[n](1)} & z^{[n](2)} & \cdots & z^{[n](m)} \\ | & | & & | \end{bmatrix}$$

Getting matrix dimensions right

Monday, 09 July, 2018 22:49

$L = 5$



Screen clipping taken: 2018-07-09 22:52

$n^{[0]} = 2 = n_x$

$n^{[1]} = 3$

$n^{[2]} = 5$

$n^{[3]} = 4$

$n^{[4]} = 2$

$n^{[5]} = 1$ ← for now we've seen only NN w/ a single output unit. In later courses we'll see NN w/ multiple output units.

FOR SINGLE EXAMPLE IMPLEMENTATION:

$$z^{[1]} = W^{[1]} x + b^{[1]}$$

$z^{[1]}$: vector of activations for the 1st hidden layer

x: vector of input features

$(3,1)$ $(2,1)$

$(n^{[1]},1)$ $(n^{[0]},1)$

So for W's dimensions:

$$\begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix} = \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot \\ \cdot \end{bmatrix}$$

$(3,2)$

$|W| = (n^{[1]}, n^{[0]})$

In General:

$$W^{[l]} : (n^{[l]}, n^{[l-1]})$$

So for $W^{[2]} : (n^{[2]}, n^{[1]}) = (5,3)$

b/c we are going to compute

$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$

$(5,1) \quad (3,1)$

$(5,3)$

$W^{[3]} : (4,5)$

$W^{[4]} : (2,4)$

$W^{[5]} : (1,2)$

For the dimensions of b remember b is the same dimension as z

In General

$b^{[l]} : (n^{[l]}, 1)$

$b^{[1]} : (3, 1)$

$b^{[2]} : (5, 1)$

$|dW^{[l]}| = |W^{[l]}| = (n^{[l]}, n^{[l-1]})$

$|db^{[l]}| = |b^{[l]}| = (n^{[l]}, 1)$

$|a^{[l]}| = |z^{[l]}| = (n^{[l]}, 1)$

FOR MULTIPLE EXAMPLE IMPLEMENTATION (VECTORIZED)

The dimensions of $\{W, dW, b, db$ stay the same as for a single example

The dimensions of z, a & x will change a little.

$|z^{[l]}| : (n^{[l]}, 1)$

$|Z^{[l]}| : (n^{[l]}, m) = |dZ^{[l]}|$

$|x| = |a^{[0]}| : (n_x, 1)$

$|X| : (n_x, m)$ ← entire training set

$|a^{[l]}| : (n^{[l]}, 1)$ → special case $l = 0$

$|a^{[\ell]}| : (n^{[\ell]}, 1)$

$|A^{[\ell]}| : (n^{[\ell]}, m) = |dA^{[\ell]}|$

special case $\ell = 0$

$|A^{[0]}| = |X| = (n^{[0]}, m) = (n_x, m)$

Note for $|b^{[\ell]}|$

$$Z^{[\ell]} = W^{[\ell]} \cdot A^{[\ell-1]} + b^{[\ell]}$$

$(n^{[\ell]}, m)$ $\quad$ $(n^{[\ell]}, n^{[\ell-1]})$ $\quad$ $(n^{[\ell-1]}, m)$ $\quad$ $(n^{[\ell]}, 1)$

$(n^{[\ell]}, m)$

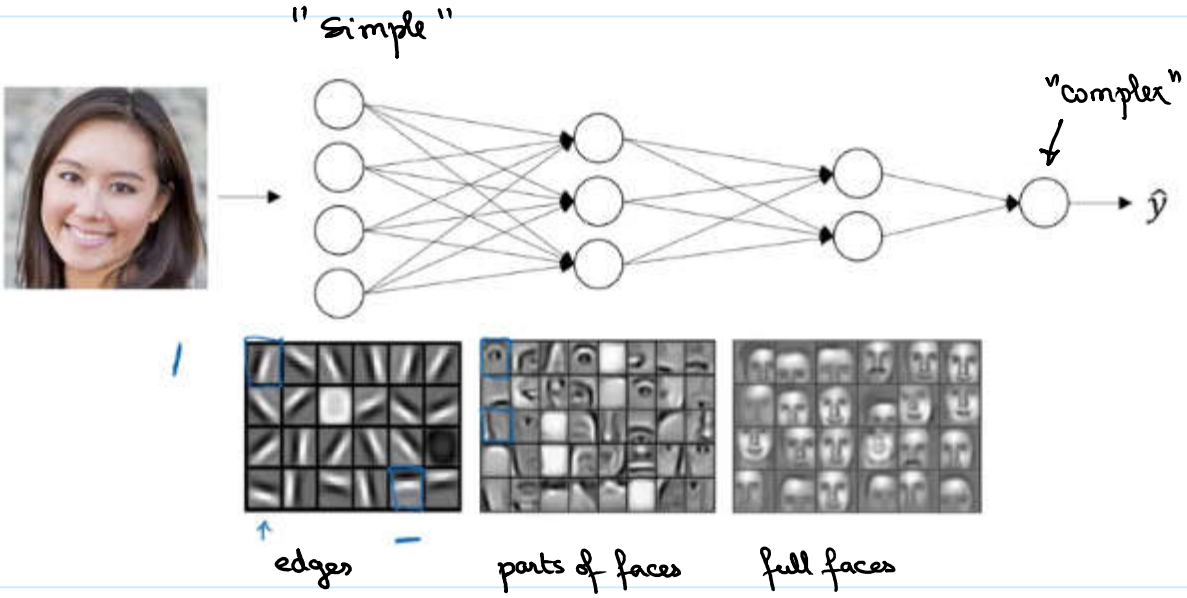
but when the addition occurs $b^{[\ell]}$ is duplicated m times by python broadcasting and added to the $W^{[\ell]} A^{[\ell-1]}$ product.

Also $|dZ^{[\ell]}| = |Z^{[\ell]}| = |A^{[\ell]}| = |dA^{[\ell]}| = (n^{[\ell]}, m)$

Why deep representations?

Tuesday, 10 July, 2018 22:09

Why do deep NN work well at solving many problems?



Screen clipping taken: 2018-07-10 22:12

First layer detects simple patterns (edges)

Second layer detects slightly more complex patterns (parts of faces)

Third layer detects highly complex patterns (faces)

A NN composes simple patterns and builds them up in a hierarchichal representation.

Audio → low level audio wave form features (rising tone, falling tone) → Phonems (basic units of sound) → words → sentences phrases.

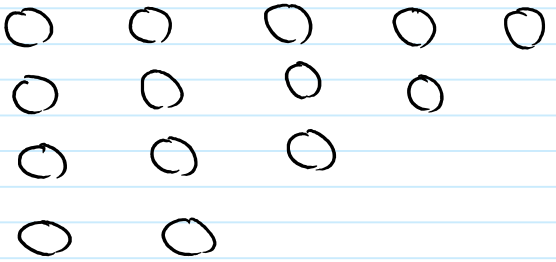
CIRCUIT THEORY & DEEP LEARNING

Circuit theory: describes what kind of functions we can compute using logic gates (AND, NOT, OR ...)

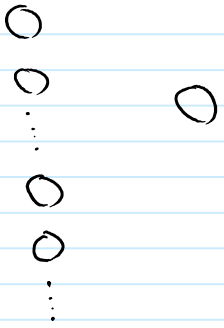
An informal argument as to why "deep" neural nets are good @ solving problems:

There are functions that can be computed with a "small" L-layer deep neural net that shallower networks require exponentially more hidden nodes to compute.

DEEP NN (14 nodes on 5 layers)



SHALLOW NN (200 nodes on 2 layers)

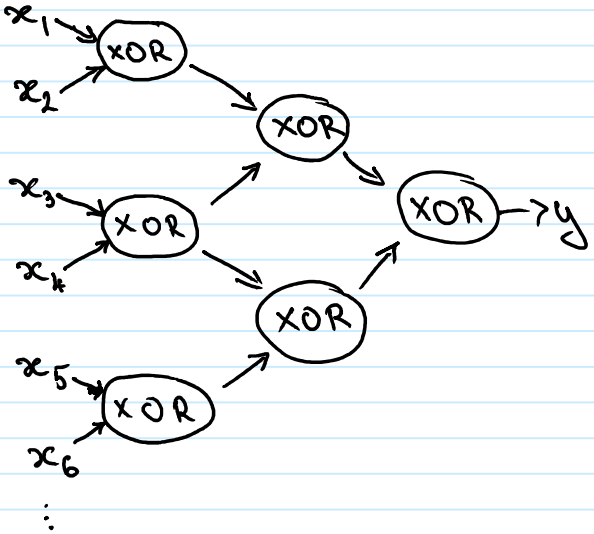


Example:

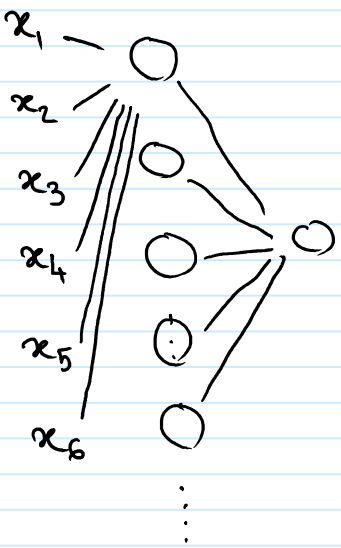
Compute the XOR of many inputs

WHEN MANY HIDDEN LAYERS ARE ALLOWED

← O(log n) layers deep →



WHEN FEW HIDDEN UNITS ARE ALLOWED



The hidden layer needs to be exhaustively large

The hidden layer needs to be exhaustively large to accommodate all the possible combinations $O(2^n)$ → exponentially large.

Very deep NN (with maybe dozens of hidden layers) seem to work on some hard problems.

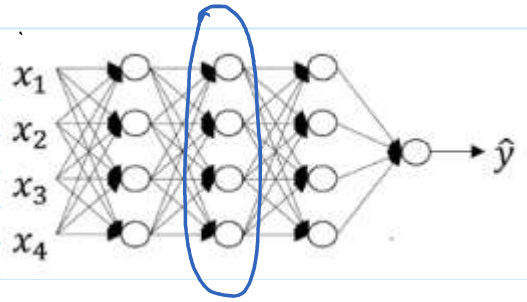
Building blocks of deep NN

Tuesday, 10 July, 2018 22:53

Key Components

Forward Propagation

Backward Propagation



Screen clipping taken: 2018-07-10 22:57

For a given layer l: $W^{[l]}, b^{[l]}$

FWD PROP:

Input: $a^{[l-1]}$

Output: $a^{[l]}$, cache $z^{[l]}$ (we'll use this during Back Prop)

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]})$$

BACK PROP:

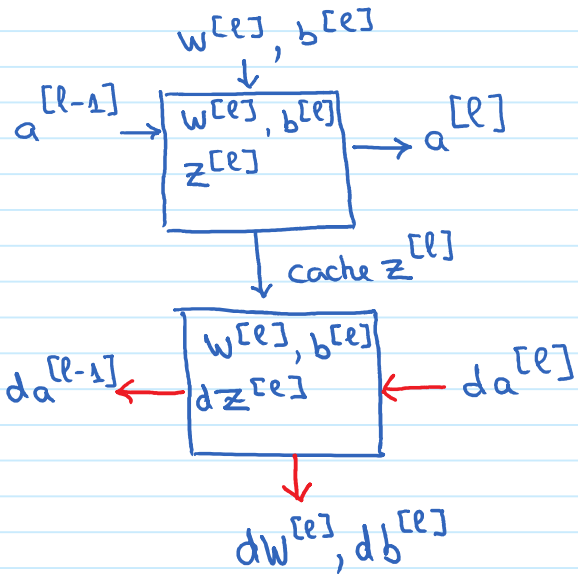
Input: $da^{[l]}$, cache $z^{[l]}$

Output: $da^{[l-1]}$

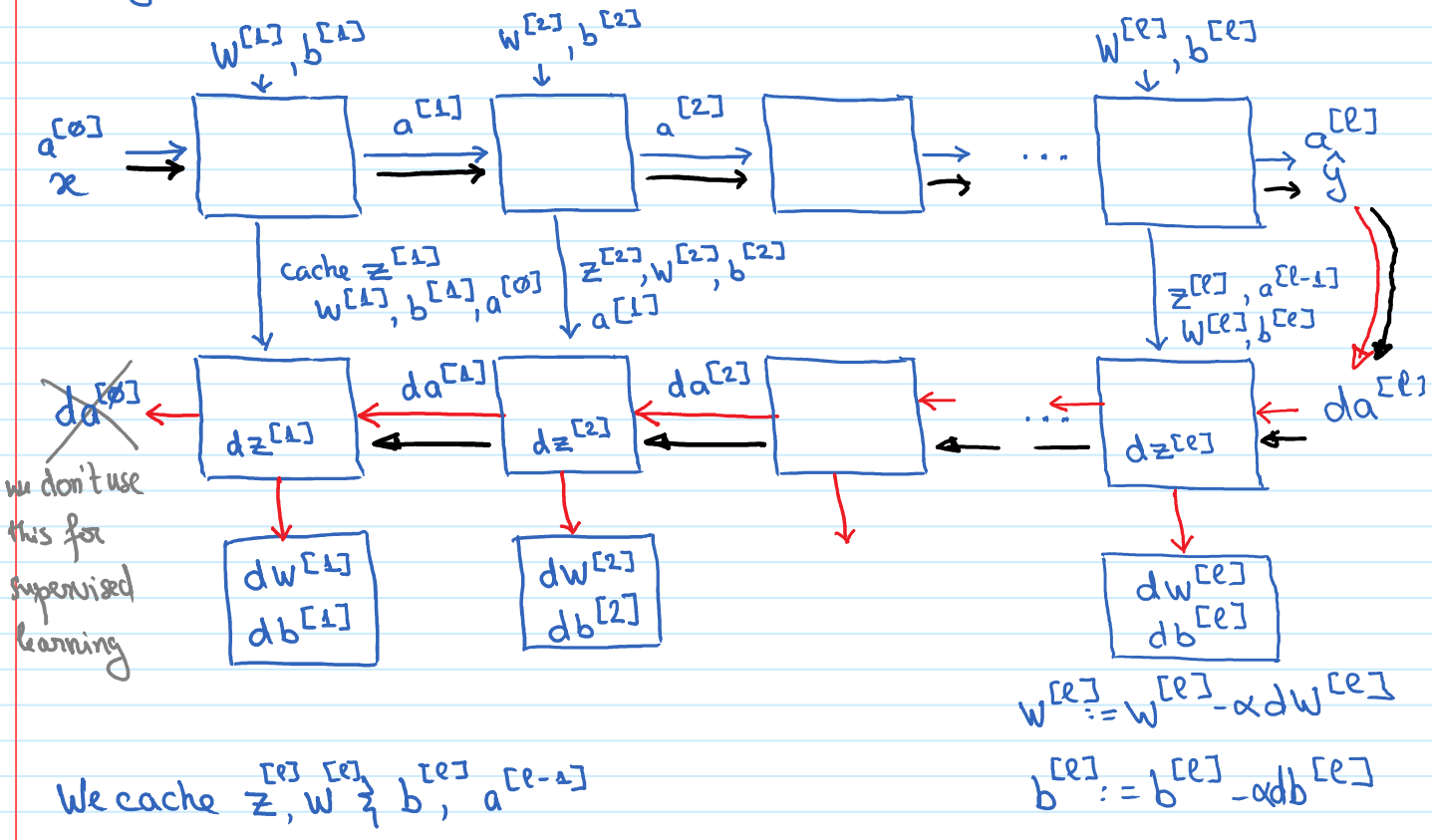
$dW^{[l]}, db^{[l]}$

For layer l:

$W^{[l]}, b^{[l]}$



For a general NN:



We cache $z^{[l]}, w^{[l]}, b^{[l]}, a^{[l-1]}$

$w^{[l]} := w^{[l]} - \alpha dw^{[l]}$

$b^{[l]} := b^{[l]} - \alpha db^{[l]}$

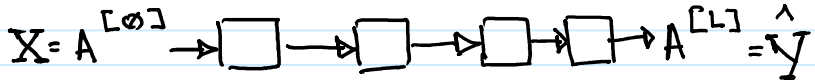
FWD & BACK prop

Tuesday, 10 July, 2018 23:21

FORWARD PROPAGATION (for layer l):

- **Inputs**: $a^{[l-1]}$ ($W^{[l]}, b^{[l]}$ saved in cache)
- **Outputs**: $a^{[l]}$ ($z^{[l]}$ saved in cache)

SINGLE EXAMPLE	VECTORIZED (python broadcast)
$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$	$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$
$a^{[l]} = g^{[l]}(z^{[l]})$	$A^{[l]} = g^{[l]}(Z^{[l]})$



BACWARD PROPAGATION (for layer l)

- **Inputs**: $da^{[l]}$
- **Outputs**: $da^{[l-1]}, dw^{[l]}, db^{[l]}$

SINGLE EXAMPLE

$$dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$$

element wise multiplication

$$dw^{[l]} = dz^{[l]} \cdot a^{[l-1]}$$

$$db^{[l]} = dz^{[l]}$$

$$da^{[l-1]} = W^{[l]T} \cdot dz^{[l]}$$

If we plug the definition of $da^{[l-1]}$ into $da^{[l]}$ and compute $dz^{[l]}$ we get:

$$dz^{[l]} = W^{[l+1]T} dz^{[l+1]} * g^{[l]'}(z^{[l]})$$

See Backprop Intuition in Week 3 for how to understand the derivation of these formulae.

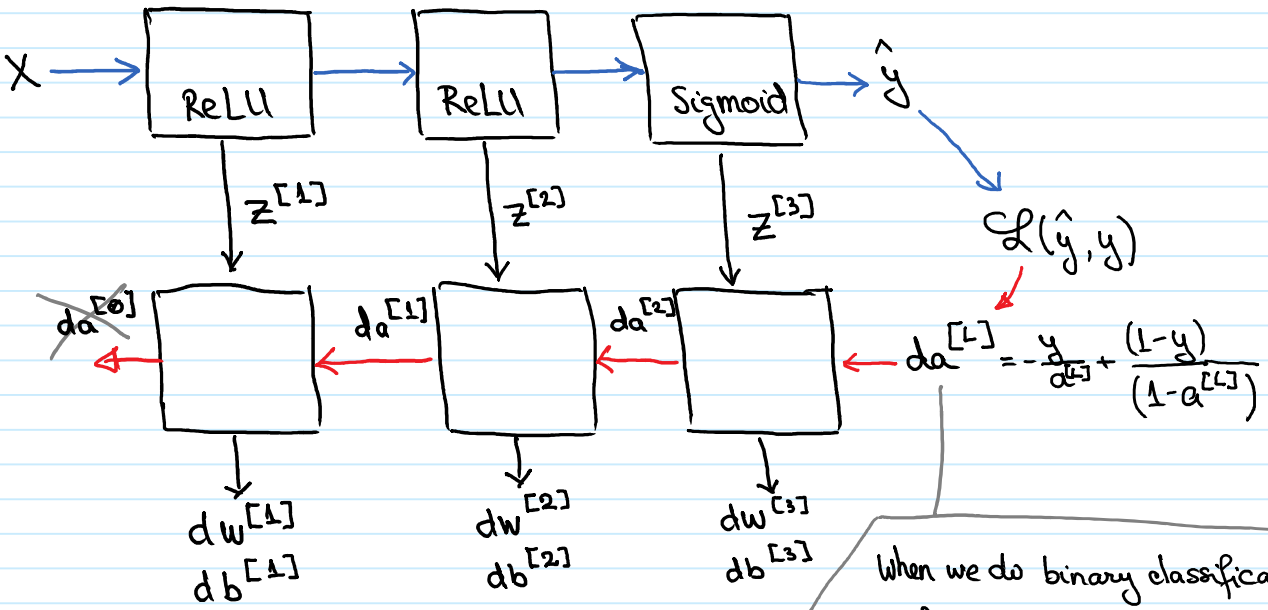
VECTORIZED

$$dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$$

$$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$$

$$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis=1, keepdims=True)$$

$$dA^{[l-1]} = W^{[l]T} dZ^{[l]}$$



When we do binary classification using logistic regression it can be shown:

$$da^{[L]} = \frac{d}{d\hat{y}} \mathcal{L}(\hat{y}, y) = \frac{d}{da^{[L]}} \mathcal{L}(a^{[L]}, y)$$

$$= -\frac{y}{a^{[L]}} + \frac{1-y}{1-a^{[L]}}$$

$$= -\frac{y}{a^{[L]}} + \frac{1-y}{1-a^{[L]}}$$

When we VECTORIZE the scheme above we get:

$$dA^{[L]} = \left[-\frac{y^{(1)}}{a^{[L]}} + \frac{1-y^{(1)}}{1-a^{[L]}} \quad \cdots \quad -\frac{y^{(m)}}{a^{[L]}} + \frac{1-y^{(m)}}{1-a^{[L]}} \right]$$

↑ the activations for all training examples

Parameters vs Hyperparameters

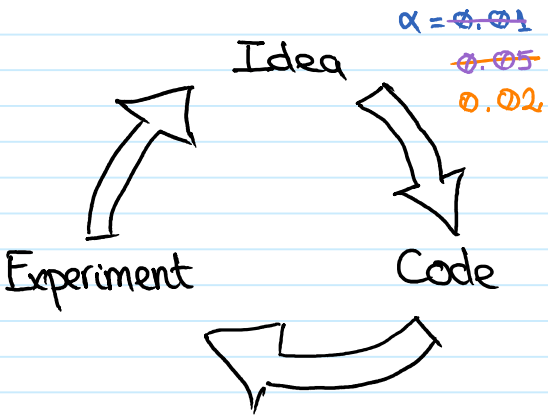
Wednesday, 11 July, 2018 21:38

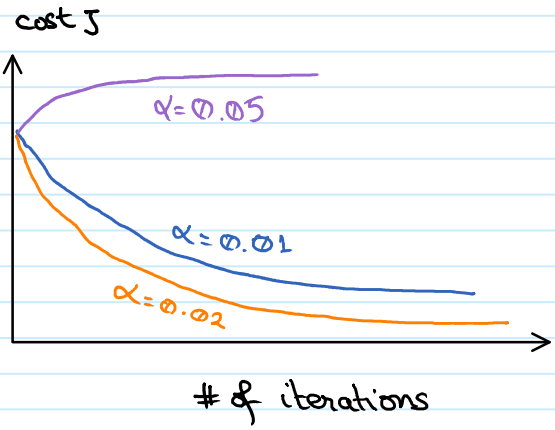
Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]} \ldots$

Hyperparameters:
- Learning rate α
- # of iterations
- # of hidden layers L
- # of hidden units $n^{[1]}, n^{[2]} \ldots$
- choice of activation function
- ⋮

LATER COURSES:
- Momentum term
- Minibatch size
- Regularization parameters

APPLYING DEEP LEARNING IS A VERY EMPIRICAL PROCESS:
(aka need to try different things and see what works)





Deep Learning is applied to many different domains:

Computer vision, speech, natural language processing (NLP), online advertising,

web search, product recommendations etc.

Intuitions from one domain sometimes carry over to other domains sometimes they don't.

→ Just try out a range of values and see what works.

Even when working in one domain hyperparameters best values change as the hardware improves or has changed, data sets have changed.

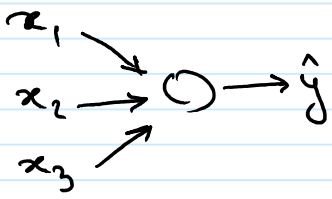
HEURISTIC: Every now and then try different hyperparameters.

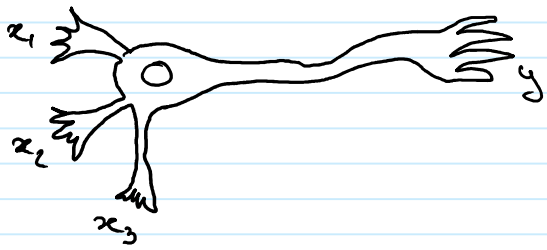
Deep Learning and the human brain

Wednesday, 11 July, 2018 21:56

Punch Line: Not very much in common!

There is a very loose analogy between a logistic regression unit and a biological neuron.





We don't know how a neuron works!