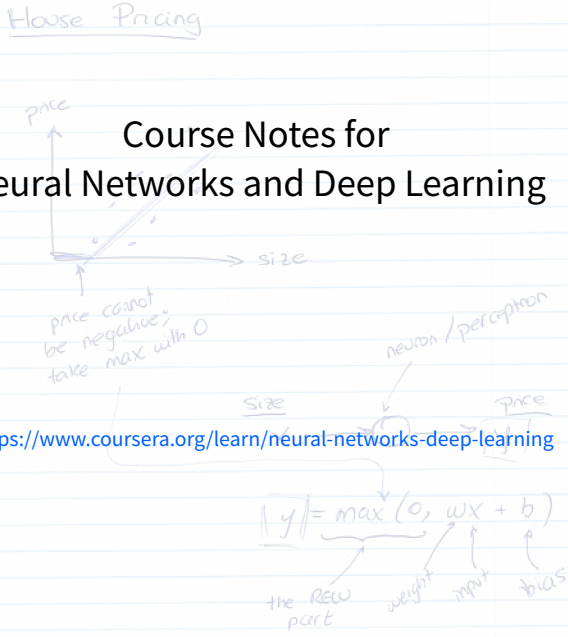
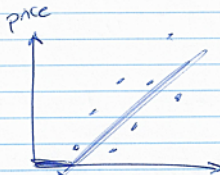


Course Notes for Neural Networks and Deep Learning

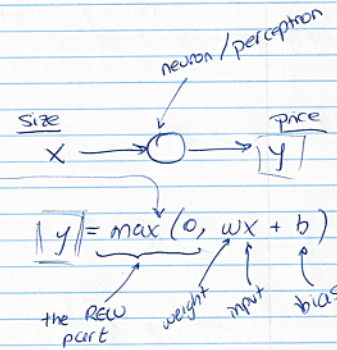
https://www.coursera.org/learn/neural-networks-deep-learning

Note: I am in no way affiliated to Coursera or the course authors.

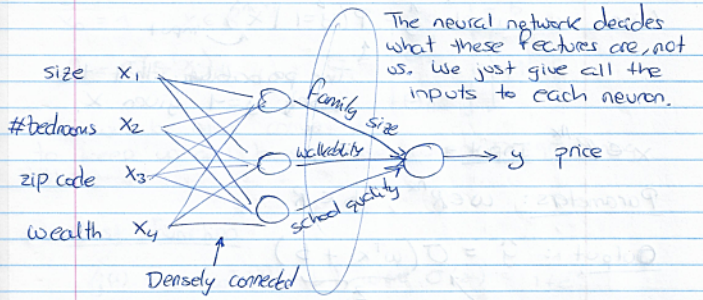
House Pricing



price cannot be negative; take max with 0



Multiple Variables



The neural network decides what these features are, not us. We just give all the inputs to each neuron.

Notation

(x, y) : input/output pair (labelled input) $\quad x \in \mathbb{R}^{n_x}, \; y \in \{0, 1\}$

m training examples : $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad \begin{matrix} \} n_x \\ \underbrace{\qquad}_{m} \end{matrix}$$

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}$$

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

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

Logistic Regression

Given x, want $\hat{y} = P(y=1 \mid x)$ — input

binary labels, $y \in \{0,1\}$

The probability that the label $y=1$ given x.

$x \in \mathbb{R}^{n_x}$ Input

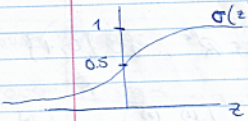
Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

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

z

sigmoid function

Linear regression. Will not produce a value $\in [0,1]$, so unsuitable for probabilities.



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

↳ Large $z \rightarrow \sigma(z) \approx \frac{1}{1+0} \approx 1$

↳ Large negative $z \rightarrow \sigma(z) \approx \frac{1}{1+Big} \approx 0$

Maps the linear regression to a value $\hat{y} \in [0,1]$ that can be interpreted as a probability.

Goal is to learn parameters w, b so that $\hat{y}$ becomes a good estimate of the probability of $y=1$ given an input x.

Alternative notation that merges w and b:

$x_0 = 1, \quad x \in \mathbb{R}^{n_x+1}$

$\hat{y} = \sigma(\theta^T x)$ $\quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{n_x} \end{bmatrix} \begin{matrix} = b \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} w \end{matrix}$

It will be more convenient to keep w and b separate when programming this.

Cost Function

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

Given $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ $\quad \left.\begin{matrix} x^{(i)} \\ y^{(i)} \end{matrix}\right\}$ i-th example

want $\hat{y}^{(i)} \approx y^{(i)}$

Loss (error) function: → Measures how good our prediction ($\hat{y}$) is.

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

This is not very good in practice where we end up with a non-convex problem: the loss function we are trying to optimise has various local optima.

Gradient descent may therefore not find the global optima.

A better loss function that yields a convex optimisation problem:

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

global optima

Intuition behind the loss function in logistic regression:

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

If $y = 1$:

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

Since we are minimising $L(\hat{y}, y)$,
we want $-\log \hat{y}$ to be a large negative number
$\Leftrightarrow$ we want $\hat{y}$ to be large
$\leftrightarrow$ $\hat{y} = 1$, since sigmoid gives a value in $[0,1]$

If $y = 0$:

$$L(\hat{y}, 0) = -\log (1-\hat{y})$$

Since we are minimising $L(\hat{y}, y)$,
we want $-\log (1-\hat{y})$ to be a large negative number
$\Leftrightarrow$ we want $\hat{y}$ to be as small as possible
$\Leftrightarrow$ $\hat{y} = 0$, since sigmoid gives a value in $[0,1]$.

Cost Function

The loss function is defined wrt a single training example.
To measure how well we are doing on the entire traing set, we define the cost function.

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

m training samples

cost function

The parameters we have learned

The average loss across the whole training set.

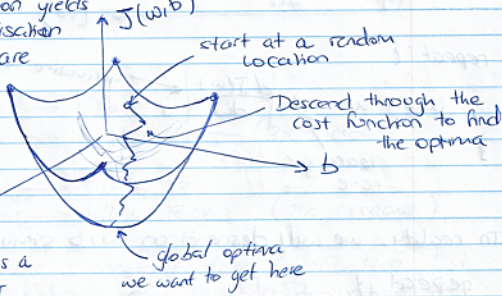
Gradient Descent

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

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

We want to find w, b that minimise $J(w,b)$

Our cost function yields a convex optimisation problem, so we are looking at a bowl with a single global optimum.

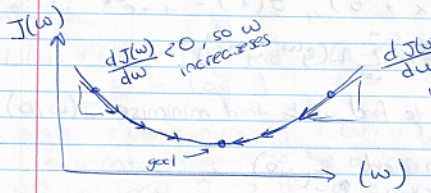


(Assuming this is a scalar just for illustration purposes)

To start the optimisation procedure, w, b are initialised to either 0 or random values. 0 is the more popular choice for logistic regression.

Had we used a different cost function such as the squared difference, the resulting optimisation problem would not be convex (have multiple local optima), and therefore gradient descent would not work really well.

Gradient descent on w (scalar) (for illustration purposes):



repeat {

$$w := w - \alpha \boxed{\frac{dJ(w)}{dw}}$$ ← derivative wrt w

}

learning rate

In reality, we will descend on w, b simultaneously:

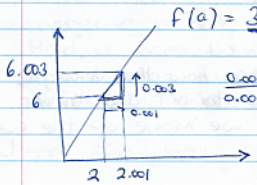
repeat {

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

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

}

Derivatives



$\frac{0.003}{0.001} = \frac{\text{height}}{\text{width}} = \frac{df(a)}{da} = \frac{d}{d(a)} f(a) = 3$

When we increase a by a small amount, $f(a)$ increases by 3x that amount.

$a = 2 \rightarrow f(a) = 6$
$a = 2.001 \rightarrow f(a) = 6.003$ (3x increase)

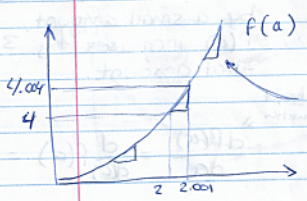
$a = 5 \rightarrow f(a) = 15$
$a = 5.001 \rightarrow f(a) = 15.003$ (3x increase)

A line has a constant slope / derivative, but other functions may have a derivative that changes at every point. So the definition of the derivative must involve a very small increase.

$$\frac{df(a)}{da} = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

This is just the definition of slope (rise over run), but with a run that gets infinitesimally smaller.

A function with a varying derivative:



$a = 2 \longrightarrow f(a) = 4$

$a = 2.001 \longrightarrow f(a) = 4.004001$ (4x increase)

$a = 5 \longrightarrow f(a) = 25$

$a = 5.001 \longrightarrow f(a) = 25.010$ (10x increase)

The increase varies with the value of a.

At $a = 2$, $\frac{d}{da} f(a) = \boxed{4}$ (2·2x)

At $a = 5$, $\frac{d}{da} f(a) = \boxed{10}$ (2·5x)

In general: $\frac{d}{d(a)} f(a) = \frac{d}{da} a^2 = \boxed{2a}$

What this means is that if we increse **a** by a small value ε, then the function $f(a)$ increases by:

$a \longrightarrow \boxed{a} + \boxed{\varepsilon}$

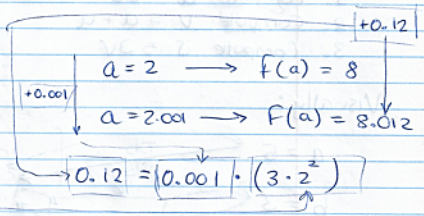
$f(a) \longrightarrow \underbrace{\boxed{f(a)}}_{\text{starting point}} + \underbrace{\boxed{2a(\varepsilon)}}_{\text{increase}}$

Note that when we bumped a from $a=2$, to $a=4$, $f(a)$ increased by 0.004001, which is not exactly $2a = 4$. This is because the increase is not an infinitesimally small value. Had a been bumped by an infinitesimally small value, we would see an increase in $f(a)$ of exactly $2(a)$.

More Examples

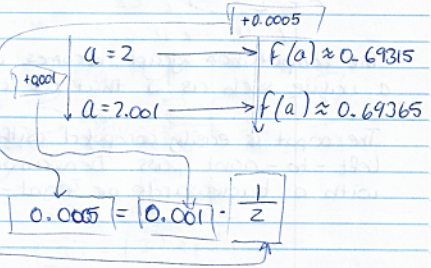
$f(a) = a^3$

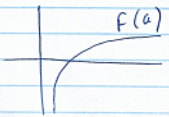
$\frac{d}{da} f(a) = 3a^2$



$f(a) = \log_e(a)$
$= \ln(a)$

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





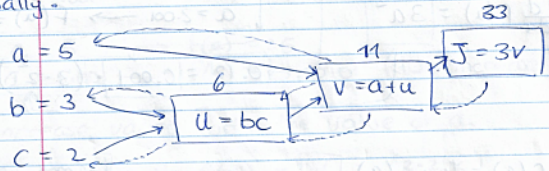
Computation Graph

A representation of a computation that breaks down smaller computations into nodes and where edges represent depencies / ordering.

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

1. Compute $u = bc$
2. Compute $v = a+u$
3. Compute $J = 3v$

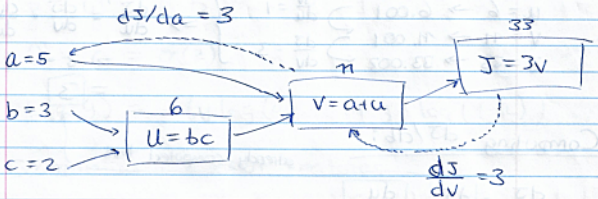
Visually:



The computation graph becomes useful when there is a value such as J that we want to optimize.

The ouput is easily computed with a forward or left-to-right pass. Derivatives can be computed with a backwards or right-to-left pass.

Derivatives with a Computation Graph



Computing dJ/dv:

$J = 3v$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

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

increase v by 0.001

J increases by 3×0.001

Computing dJ/da:

$\frac{dJ}{da}$

$a = 5 \rightarrow 5.001$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

$dv/da = 1$

$dJ/dv = 3$

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

Chain rule: $\frac{dJ}{da} = \frac{dJ}{dv} \cdot \frac{dv}{da}$

$3 = 3 \cdot 1$

Computing dJ/du:

$u = 6 \rightarrow 6.001$ $\quad \frac{dv}{du} = 1$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$ $\quad \frac{dJ}{dv} = 3$

$$\Rightarrow \frac{dJ}{du} = \frac{dJ}{dv} \cdot \frac{dv}{du}$$

$$= 3 \cdot 1$$

$$= \boxed{3}$$

Computing dJ/db:

already computed this

$$\frac{dJ}{db} = \boxed{\frac{dJ}{du}} \cdot \boxed{\frac{du}{db}}$$

$b = 3 \rightarrow 3.001$ $\quad \frac{du}{db} = 2$

$u = 6 \rightarrow 6.002$

$$= 3 \cdot 2$$

$$= 6$$

We can verify the math:

$b = 3 \rightarrow 3.001$ $\quad \frac{du}{db} = 2$

$u = 6 \rightarrow 6.002$ $\quad \frac{dv}{du} = 1$

$v = 11 \rightarrow 11.002$

$J = 33 \rightarrow 33.006$ $\quad \frac{dJ}{dv} = 3$

$$\frac{dJ}{db} = 2 \cdot 1 \cdot 3 = 6$$

$$\frac{dJ}{db} = \frac{dJ}{dv} \cdot \frac{dv}{du} \cdot \frac{du}{db}$$

$$= 3 \cdot 1 \cdot 2$$

$$= 6$$

Computing dJ/dc:

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

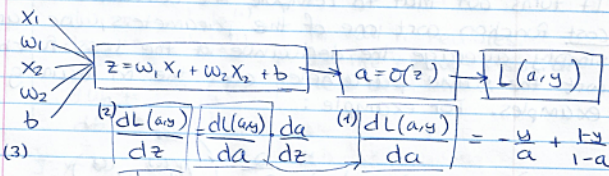
Logistic Regression Gradient Descent

$$z = w^T x + b$$

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

$$L(a, y) = -(y \log(a) + (1-y) \log(1-a))$$

Example with 2 input features



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

$$= a - y$$

x_1, x_2 are the input features.
we want to tune the parameters w_1, w_2, b to minimise the loss function L.

Proceed backwards in the computation graph from the output to the input.

1. Compute dL/da
2. Compute dL/dz
3. Compute dL/dw_1, dL/dw_2, dL/db
4. Update:

$$w_1 := w_1 - \alpha \, dL/dw_1$$
$$w_2 := w_2 - \alpha \, dL/dw_2$$
$$b := b - \alpha \, dL/db$$

α: learning rate

Gradient Descent on m Examples

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

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

Previously, we had seen how to compute $dw_1^{(i)}$, $dw_2^{(i)}$, $db^{(i)}$ for a single training example $(x^{(i)}, y^{(i)})$.

It turns out that to compute the derivative of the cost function wrt one of the parameters, we can simply average the derivative of the loss function wrt the same parameter over the m training examples. For example:

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial w_1} L(a^{(i)}, y^{(i)})$$

- $\frac{\partial}{\partial w_1} J(w,b)$: cost
- $\frac{1}{m}\sum_{i=1}^{m}$: Average on all m training examples
- $\frac{\partial}{\partial w_1} L(a^{(i)}, y^{(i)})$: loss. Already saw how to compute this term $\frac{\partial L(a,y)}{\partial w_1}$

Example code is on the next page.

In practice, to speed up computations, we would use vectorisation instead of using explicit loops.

One single step of gradient descent over m training examples:

$J = 0;\ dw_1 = 0;\ dw_2 = 0;\ db = 0$

for $i = 1 \dots m$:

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

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

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

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

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

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

$\vdots$

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

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

For all n features. In the examples so far we used $n = 2$.

$J \mathrel{/}= m$

$dw_1 \mathrel{/}= m$

$dw_2 \mathrel{/}= m$

$\vdots$

$dw_n \mathrel{/}= m$

$db \mathrel{/}= m$

Dividing everything by m to average out over the m training examples.

// update!

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

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

$\vdots$

$w_n := w_n - \alpha\, dw_n$

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

Vectorisation

The point of vectorisation is to remove explicit 'for' loops and instead process several elements (a vector) at a time using special CPU/GPU instructions.

Derivatives with Vectorisation during Gradient Descent

Instead of:

$dw_1 = 0, dw_2 = 0, \ldots, dw_{nx} = 0$

for $i = 1 \ldots m$:

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

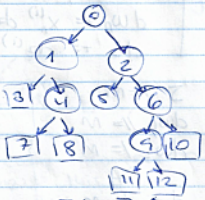
$dw_2 = x_2^{(i)} dz^{(i)}$

$dw_{nx} = x_{nx}^{(i)} dz^{(i)}$

$dw_1 /= m$

$dw_2 /= m$

$dw_{nx} /= m$



Write:

$dw = np.zeros((nx, 1))$

For $i = 1 \ldots m$:

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

$dw /= m$

$\begin{bmatrix} dw_1 \\ dw_2 \\ \vdots \\ dw_{nx} \end{bmatrix} \mathrel{+}= \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_{nx}^{(i)} \end{bmatrix} \begin{bmatrix} dz^{(i)} \\ dz^{(i)} \\ \vdots \\ dz^{(i)} \end{bmatrix}$

These are scalars, but are automatically expanded

Vectorizing Logistic Regression

To perform a forward pass of logistic regression given m training examples, we need to compute:

$$z^{(1)} = w^T x^{(1)} + b \qquad z^{(2)} = w^T x^{(2)} + b \qquad \cdots \qquad z^{(m)} = w^T x^{(m)} + b$$
$$a^{(1)} = \sigma(z^{(1)}) \qquad a^{(2)} = \sigma(z^{(2)}) \qquad\qquad a^{(m)} = \sigma(z^{(m)})$$

Instead of iterating over the m training examples explicitly with a for loop, we define a huge matrix:

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \qquad X \in \mathbb{R}^{n_x, m}$$

Then:

$$\underset{1\times m}{Z} = \begin{bmatrix} z^{(1)} & z^{(2)} & \cdots & z^{(m)} \end{bmatrix} = \underset{1\times n_x}{w^T} \underset{n_x \times m}{X} + \underset{1\times m}{\begin{bmatrix} b & b & \cdots & b \end{bmatrix}}$$

Where w^T is the row vector $w^T = \begin{bmatrix} w^{(1)} & w^{(2)} & \cdots & w^{(m)} \end{bmatrix}$

$$\begin{bmatrix} z^{(1)} & z^{(2)} & \cdots & z^{(m)} \end{bmatrix} = \begin{bmatrix} w^{(1)} & w^{(2)} & \cdots & w^{(m)} \end{bmatrix} \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} + \begin{bmatrix} b & b & \cdots & b \end{bmatrix}$$

In code:

$Z = np.dot(w.T, X) + b$ ← scalar b, automatically broadcasted

Finally, we would compute:

$$A = \begin{bmatrix} a^{(1)} & a^{(2)} & \cdots & a^{(m)} \end{bmatrix} = \sigma(Z)$$

Vectorising the Gradient Computation

1. Instead of computing $dz^{(i)}$ over the m training examples with a loop:

$$dz^{(1)} = a^{(1)} - y^{(1)}, \quad dz^{(2)} = a^{(2)} - y^{(2)}, \quad \dots, \quad dz^{(m)} = a^{(m)} - y^{(m)}$$

define:

$$dZ = A - Y$$

$$\left[dz^{(1)}\ dz^{(2)} \cdots dz^{(m)}\right] = \left[a^{(1)} - y^{(1)},\ a^{(2)} - y^{(2)},\ \dots,\ a^{(m)} - y^{(m)}\right]$$

2. To compute db,

Instead of:

```
db = 0
db += dz(1)
db += dz(2)
   :
db += dz(m)
db /= m
```

Do this:

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

$$= \frac{1}{m} \text{np.sum}(dz)$$

3. To compute dw,

Instead of: (vectors, scalars)

```
dw = 0
dw += x(1) dz(1)
dw += x(2) dz(2)
   :
dw += x(m) dz(m)
dw /= m
```

Do this:

$$dw = \frac{1}{m} X\, dz^{T}$$

To see why the vectorised computation of dw works:

$$dw = \frac{1}{m} X \, dz^T$$

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

$$= \frac{1}{m} \underbrace{\left[x^{(1)} dz^{(1)} + x^{(2)} dz^{(2)} + \cdots + x^{(m)} dz^{(m)} \right]}_{n \times 1}$$

Example:

$$\begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ dz^{(2)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} dz^{(1)} + x_1^{(2)} dz^{(2)} \\ x_2^{(1)} dz^{(1)} + x_2^{(2)} dz^{(2)} \end{bmatrix}$$

$$= \begin{bmatrix} x^{(1)} dz^{(1)} \\ x_2^{(1)} dz^{(1)} \end{bmatrix} + \begin{bmatrix} x_1^{(2)} dz^{(2)} \\ x_2^{(2)} dz^{(2)} \end{bmatrix} = dz^{(1)} \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} + dz^{(2)} \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \end{bmatrix}$$

$$= \underbrace{x^{(1)}}_{\text{vector}} \underbrace{dz^{(1)}}_{\text{scalar}} + \underbrace{x^{(2)}}_{\text{vector}} \underbrace{dz^{(2)}}_{\text{scalar}}$$

Putting it all together to implement Logistic regression:

$$z = w^T X + b = \text{np.dot}(w.T, X) + b$$
$$A = \sigma(z)$$
$$dz = A - Y$$
$$dw = \frac{1}{m} X dz^T$$
$$db = \frac{1}{m} \text{np.sum}(dz)$$

$$w = w - \alpha dw$$
$$b = b - \alpha db$$

This implements one step of forward propagation and backward propagation on all m training samples.

Note that we will still need a for loop over the number of training iterations.

Logistic Regression Cost Function

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

Interpret $\hat{y} = p(y=1|x)$, which in other words is:

$$\left.\begin{array}{l} \text{if } y=1: \; p(y|x) = \hat{y} \\ \text{if } y=0: \; p(y|x) = 1-\hat{y} \end{array}\right\} p(y|x)$$

Next, we summarise these two equations in a single equation:

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

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

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

Finally, we are going to maximise $\log p(y|x)$. Note that since the log function is monotonically increasing, maximising $\log p(y|x)$ is the same as maximising $p(y|x)$.

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

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

$$= -L(\hat{y}, y)$$

Maximise $\log p(y|x)$ = minimise negative loss

This defines the loss function for a single example. Now, let's define the cost function for m examples.

Assuming that the training examples are drawn independently and are identically distributed:

$$p(\text{labels in training set}) \overset{\text{i.i.d}}{=} \prod p(y^{(i)} | x^{(i)})$$

To carry out maximum likelihood estimation, we find the parameters that maximise the probability of the observations in the training set. Again, instead of maximising the probability directly, we maximise its log:

$$\log P(\text{labels}) = \sum_{i=1}^{m} \underbrace{\log P(y^{(i)}, x^{(i)})}_{-L(\hat{y}^{(i)}, y^{(i)})}$$

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

Cost: (minimise)

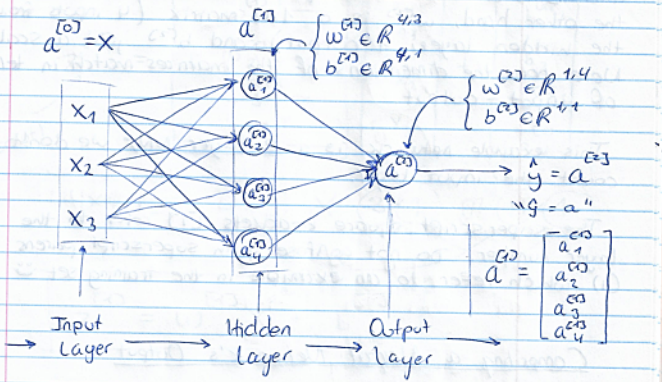
$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

Minimising the cost is maximising the likelihood, so the negative sign $-$ disappears.

The $\frac{1}{m}$ is added for convenience, so that quantities are better scaled. The cost function ends up being the average of the loss function across the m training examples.

SHALLOW Neural Networks

Neural Network Representation



"Hidden" refers to the fact that the values for the hidden layer are not observed in the training set. The training set only contains (input, output) pairs.

"a" stands for activations. They are the output of each layer. The inputs can be thought as the first set of activations.

Note the dimension of things. In logistic regression, "a" is a scalar. Here, $a^{[1]}$ is a 3D vector, $a^{[1]}$ is a 4D vector, and $a^{[2]}$ (the final output) is a scalar.

Similarly, every non-input layer is associated with a set of parameters w, b. w need no longer be a vector, nor "b" a scalar. In the example, $w^{[1]}$ is a 4×3 matrix (the layer has 3 inputs & 4 outputs), and b is a 4×1 matrix (a 4D column vector). On the other hand, $w^{[2]}$ is a 1×4 matrix (4 inputs from the hidden layer, 1 output) and $b^{[2]}$ just a scalar. Note how the dimensions of the matrices match in terms of input - output.

This example network is a 2 layer NN; we don't count the input layer.

The superscript square brackets $[1]$ refer to the layer number. Do not confuse with superscript parens (1) which refer to an example in the training set :)

Computing a Neural Network's Output

The neural network is basically going to perform one logistic regression per node / neuron.

Hidden Layer Computations

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

The notation is: $z_i^{[L]}$ ← L: layer, i: node inside the layer

And of cause, we are going to vectorise the whole computation. This is done by putting all the weight vectors into a matrix (this is done for each layer):

$$z_1^{[1]} = \begin{bmatrix} — & w_1^{[1]T} & — \\ — & w_2^{[1]T} & — \\ — & w_3^{[1]T} & — \\ — & w_4^{[1]T} & — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}$$

No subscript!

$$4\times 1 \;(=)\; 4\times 3 \;\times\; 3\times 1 \;(+)\; 4\times 1$$

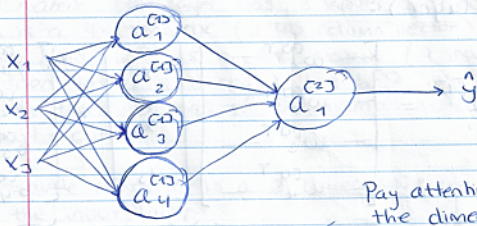
$$z^{[1]} = W^{[1]} \underbrace{X}_{=a^{[0]}} + b^{[1]}$$

Similarly for the activations:

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \begin{bmatrix} \sigma(z_1^{[1]}) \\ \sigma(z_2^{[1]}) \\ \sigma(z_3^{[1]}) \\ \sigma(z_4^{[1]}) \end{bmatrix}$$

$$a^{[1]} = \sigma\left(z^{[1]}\right)$$

Finally, we would go about doing the same thing for the output layer. The computations are all summarised here:



Pay attention to the dimension of things!

Given input x:

$$\underset{4\times1}{z^{[1]}} = \underset{4\times3}{W^{[1]}}\,\underset{3\times1}{a^{[0]}} + \underset{4\times1}{b^{[1]}}$$

$$\underset{4\times1}{a^{[1]}} = \sigma\left(\underset{4\times1}{z^{[1]}}\right)$$

$$\underset{1\times1}{z^{[2]}} = \underset{1\times4}{W^{[2]}}\,\underset{4\times1}{a^{[1]}} + \underset{1\times1}{b^{[2]}}$$

$$\underset{1\times1}{a^{[2]}} = \sigma\left(\underset{1\times1}{z^{[2]}}\right)$$

Vectorising Across Multiple Examples

Now we want to compute the output of the NN, but for all training examples:

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

Without vectorisation, we would just repeat what we did in the previous section but for all examples:

for $i = 1..m$:

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

(first layer; second layer)

To vectorise this, we construct one giant matrix with all training examples stacked horizontally:

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \qquad X \in \mathbb{R}^{(n_x, m)}$$

And then we compute: (matrices)

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

(matrices)

To visualise this:

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

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

$$Z^{[1]} = \begin{bmatrix} — & w_1^{[1]T} & — \\ — & w_2^{[1]T} & — \\ & \vdots & \\ — & w_n^{[1]T} & — \end{bmatrix} \begin{bmatrix} | & | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & | & & | \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_n^{[1]} \end{bmatrix}$$

Here, 'n' is the number of neurons in the layer, and 'm' is the number of training examples.

Finally:

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

Interpretation:

$A^{[1]} \in \mathbb{R}^{n \times m}$

activation → $a_{i,j}$ ← j-th training example

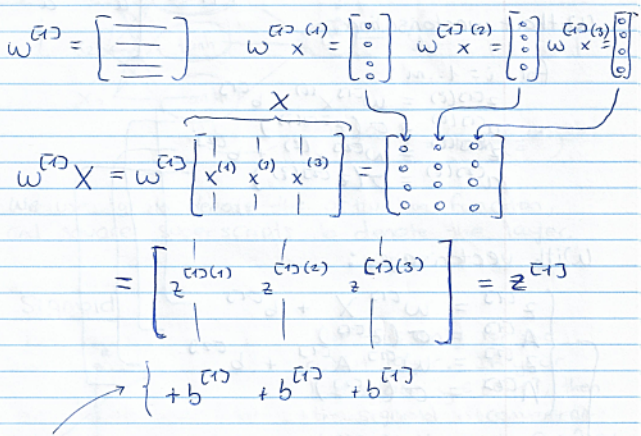
$a_{i,j}$ ← i-th neuron of the layer

$A^{[1]} =$ → example, ↓ neuron

More visualisation follows. Given that we have to vectorise:

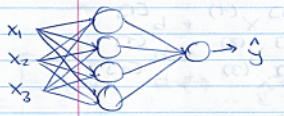
$$z^{[1](1)} = w^{[1]} x^{(1)} + b^{[1]}$$
$$z^{[1](2)} = w^{[1]} x^{(2)} + b^{[1]}$$
$$z^{[1](3)} = w^{[1]} x^{(3)} + b^{[1]}$$

Let's pretend that $b^{[1]} = 0$, to make things easier. Then:



With a non-zero $b^{[1]}$, $b^{[1]}$ is broadcasted across the columns of $Z^{[1]}$.

To summarise:



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

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

Without vectorisation:

for $i = 1 \ldots m$:

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

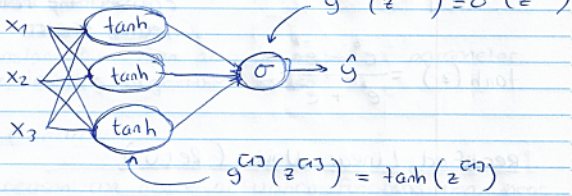
With vectorisation:

$$\begin{cases} Z^{[1]} = W^{[1]} X + b^{[1]} \\ A^{[1]} = \sigma(Z^{[1]}) \\ Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]} \\ A^{[2]} = \sigma(Z^{[2]}) \end{cases}$$

Each of these four lines correspond to the lines above but vectorised across all m training examples.

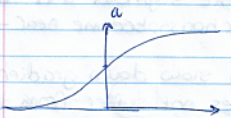
Activation Functions

One can use activation functions other than the sigmoid in the layers of a neural network. The activation function will be the same for all neurons in a given layer, but each layer can use a different one.



We use 'g' to denote the activation function, and square superscripts to denote the layer.

Sigmoid

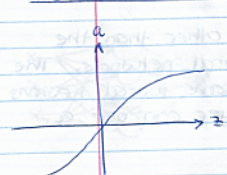


$$a = \frac{1}{1 + e^{-z}}$$

Use this only for the output layer. If your labels are $y \in \{0, 1\}$, then the sigmoid is convenient because it outputs $0 \leq \hat{y} \leq 1$.

For hidden layers, the tanh function is a better choice.

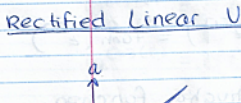
Hyperbolic Tangent



Use this rather than the sigmoid for hidden layers. With tanh, the activations of a layer will have a mean closer to 0, and that centering around 0 causes the network to learn faster.

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

Rectified Linear Unit (ReLU)



This is a good default choice.

When z is very large or very small, the derivatives of the sigmoid and tanh functions become near-zero.

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

These near-zero derivatives then slow down gradient descent. The relu function does not suffer from this problem, because the derivative is >0 for $z>0$. Of course, for $z<0$ the derivative of the relu is 0, but this does not usually cause problems in practice. If it does, try the leaky relu.

Leaky ReLu

$a = \max(0.01 \cdot z, z)$

Also a good default choice, though apparently not as popular as the ReLu.

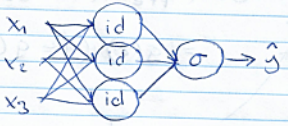
Fixes the problem with the zero derivative for $z < 0$.

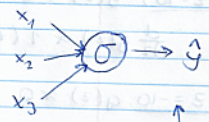
The constant 0.01 can also be made a parameter of the learning algorithm instead.

Non-Linearity

The reason we make activation functions non-linear is to make neural networks more powerful / expressive. If we were to use the identity as the activation function, then all we would be doing is computing a linear combination of the inputs, no matter how many hidden layers we add. This defeats deep networks.

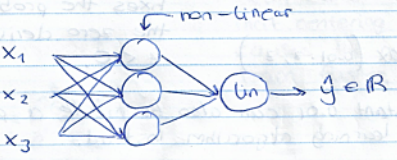
For instance:





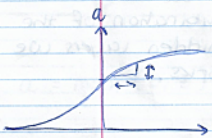
This is not more expressive than the network on the right because the composition of two linear functions is itself a linear function.

One case where we would use a linear activation function like the identity is when computing a regression, that is when $y \in \mathbb{R}$ (and also $\hat{y}$). Even in this case, however, we would use non-linear activation functions in the hidden layers to make the network more expressive.



Derivatives of Activation Functions

Sigmoid



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

$\frac{d}{dz} g(z)$ = slope of $g(x)$ at z

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

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

$z = 10$, $g(z) \approx 1$

$\frac{d}{dz} g(z) \approx 1(1-1) \approx 0$

$z = -10$, $g(z) \approx 0$

$\frac{d}{dz} g(z) \approx 0(1-0) \approx 0$

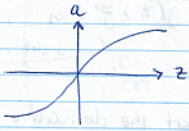
Collapse!

$z = 0$, $g(z) = 1/2$

$$\frac{d}{dz} g(z) = \frac{1}{2}\left(1 - \frac{1}{2}\right) = \frac{1}{4}$$

Notice how the derivative of the sigmoid collapses to near zero for large positive/negative values of z. This is what we talked about when introducing the ReLU.

Tanh



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

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

$$= 1 - g^2(z)$$

$z = 10$, $\tanh(z) \approx 1$

$$\frac{d}{dz} g(z) \approx \boxed{0}$$

$z = -10$, $\tanh(z) \approx -1$

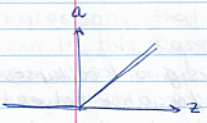
$$\frac{d}{dz} g(z) \approx \boxed{0}$$

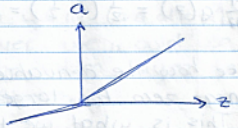
Collapses for large $\pm z$ just like the sigmoid.

$z = 0$, $\tanh(z) = 0$

$$\frac{d}{dz} g(z) = 1$$

ReLU and Leaky ReLU





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

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

$$\frac{d}{dz} g(z) = \begin{cases} 0 & , z < 0 \\ 1 & , z > 1 \\ \text{undef} & , z = 0 \end{cases}$$

$$\frac{d}{dz} g(z) = \begin{cases} 0.01 & , z < 0 \\ 1 & , z > 0 \\ \text{undef} & , z = 0 \end{cases}$$

In software, we'll just pretend that the derivative is just 1 or 0 when $z = 0$. This is technically a subgradient of the activation function, which works just fine for our purposes.

Gradient Descent for Neural Networks

Parameters: $w^{[1]} \in \mathbb{R}^{n^{[1]} \times n^{[0]}}$

$b^{[1]} \in \mathbb{R}^{n^{[1]} \times 1}$

$w^{[2]} \in \mathbb{R}^{n^{[2]} \times n^{[1]}}$

$b^{[2]} \in \mathbb{R}^{n^{[2]} \times 1}$

Cost Function: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]})$

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

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

Gradient Descent:

repeat {

// Forward pass

compute predictions $\{ \hat{y}^{(i)} \mid i = 1..m \}$

// Back prop, compute derivatives

$dw^{[1]} = \frac{dJ}{dw^{[1]}}$, $db^{[1]} = \frac{dJ}{db^{[1]}}$

$dw^{[2]} = \frac{dJ}{dw^{[2]}}$, $db^{[2]} = \frac{dJ}{db^{[2]}}$

// Update parameters

$w^{[1]} := w^{[1]} - \alpha\, dw^{[1]}$

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

$w^{[2]} := w^{[2]} - \alpha\, dw^{[2]}$

$b^{[2]} := b^{[2]} - \alpha\, db^{[2]}$

}

Forward Propagation

$$A^{[1]} = g^{[1]}\left(Z^{[1]} = W^{[1]} X + b^{[1]}\right)$$
$$A^{[2]} = g^{[2]}\left(Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}\right)$$

Back propagation

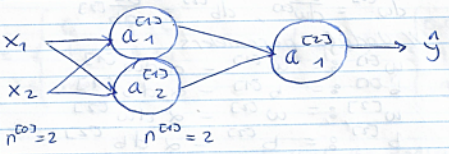
$$dZ^{[2]} = A^{[2]} - Y$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} \text{np.sum}(dZ^{[2]}, \text{axis} = 1, \text{keepdims} = \text{True})$$
$$dZ^{[1]} = W^{[2]T} dZ^{[2]} \,|*|\, g^{[1]\prime}(Z^{[1]})$$
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$$
$$db^{[1]} = \frac{1}{m} \text{np.sum}(dZ^{[1]}, \text{axis} = 1, \text{keepdims} = \text{True})$$

$|*|$ element-wise product

Random Initialization

When training a neural network, we must initialise the weights randomly instead of setting them to 0, otherwise the training will not work. This was ok for logistic regression, but not for networks with more than one neuron. The bias vectors can still be zero.

Consider:



Now initialise:

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

Now, for any given input, the two neurons in the hidden layer will compute the same function, that is:

$$a_1^{[1]} = a_2^{[1]}$$

Consequently, when we compute backpropagation, the derivatives will be the same:

$$dz_1^{[1]} = dz_2^{[1]}$$

It can then be shown by induction that after every single training iteration, the two hidden

units will still be computing the same function.

$$dW = \begin{bmatrix} u & v \\ u & v \end{bmatrix} \qquad W^{[1]} := W^{[1]} - \alpha\, dW$$

$$W^{[1]} = \begin{bmatrix} \\ \\ \\ \end{bmatrix} \left.\right\} \text{identical rows}$$

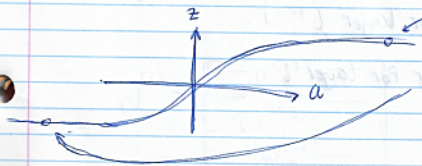
With a zero initialisation, all neurons in a layer are symmetric, and this symmetry never breaks throughout the training.

Instead, initialise weights randomly. Bias vectors can still be zero because they don't suffer from the symmetry problem:

$$W^{[1]} = \text{np.random.randn}((2,2)) \cdot \boxed{0.01}$$
$$b^{[1]} = \text{np.zero}((2,1))$$
$$W^{[2]} = \ldots$$
$$b^{[2]} = \ldots$$

small value!

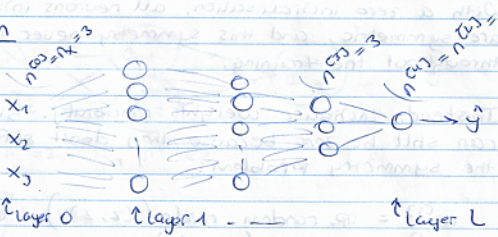
The constant 0.01 is to give initial weights small values. This is to prevent saturation of sigmoid and tanh functions. This matters even if you only use sigmoid and tanh in the output layer, and ReLUs in the hidden ones.



Finally, note that the constant 0.01 will likely work only for shallow networks. As you add more layers to the network, you will want to make this constant smaller.

Deep Neural Network

Notation



L = # layers

$n^{[l]}$ = # units in layer l

$a^{[l]}$ = activations of layer l $\left(a^{[l]} = g^{[l]}(z^{[l]})\right)$

$a^{[0]} = X$ = input

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

$W^{[l]}$ = weights for layer l

$b^{[l]}$ = bias vector for layer l

Forward Propagation

There is nothing particularly new here, just more layers.

Given an input, we need to compute:

for $l = 1 .. L$:

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

Where:

$a^{[0]} = x$ = input
$a^{[L]} = \hat{y}$ = output

$g^{[l]}$ = activation function of layer l
$w^{[l]}, b^{[l]}$ = weights and bias

We can then vectorise over all 'm' training examples like we had done before:

for $l = 1 .. L$: ← Can't get rid of this loop

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

Where

$$X = A^{[0]} = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

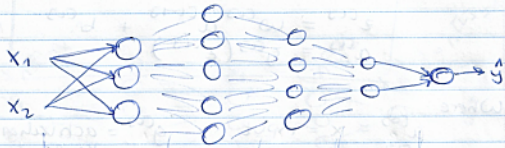
Training examples input

$$W^{[l]} = \begin{bmatrix} — & w_1^{[l]T} & — \\ — & w_2^{[l]T} & — \\ & \vdots & \\ — & w_n^{[l]T} & — \end{bmatrix}$$

Weight matrix for layer l

The explicit for-loop on the layers is still necessary because we need the output of one layer to compute the output of the following layer. There does not appear to be a way to vectorise this.

Matrix Dimensions



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

$$(3,1) = (3,2)(2,1) + (3,1)$$
$$(n^{[1]}, 1) = (n^{[1]}, n^{[0]})(n^{[0]}, 1) + (n^{[1]}, 1)$$

$$w^{[l]} = (n^{[l]}, n^{[l-1]})$$
$$b^{[l]} = (n^{[l]}, 1)$$
$$dw^{[l]} = (n^{[l]}, n^{[l-1]})$$
$$db^{[l]} = (n^{[l]}, 1)$$

A matrix maps inputs to outputs. The number of columns is the number of inputs, and the number of rows is the number of outputs:

$$W^{[l]} \rightarrow \left(n^{[l]}, n^{[l-1]}\right)$$

$n^{[l]}$ outputs; $n^{[l-1]}$ inputs

Vectorised Version

The vectorised version does not change much. Now we have:

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

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

Therefore:

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

This is still a vector

$$\left(n^{[l]}, m\right) = \left(n^{[l]}, n^{[l]}\right)\left(n^{[l]}, m\right) + \left(n^{[l]}, 1\right)$$

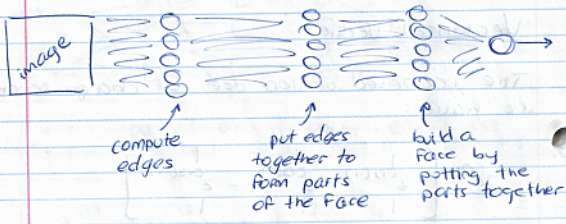
broadcasted

$$\left(n^{[l]}, m\right)$$

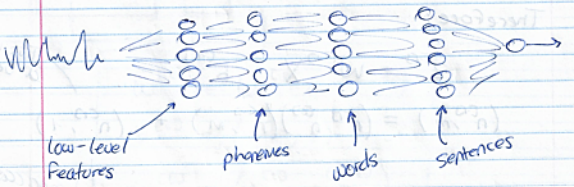
Why Deep Representations?

The idea is that further layers in the neural network can compute more and more complex functions by combining the outputs of the previous layer. Conversely, the first layers in the network compute simple functions.

Example: image recognition



Example: speech recognition

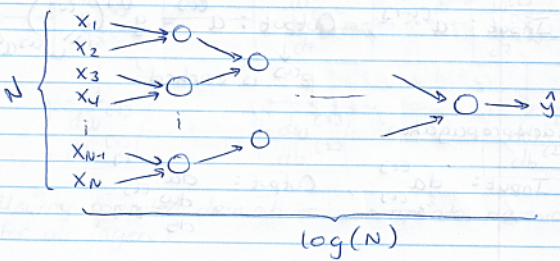


Circuit theory and deep learning

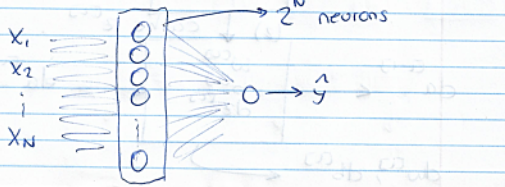
"There are functions you can compute with a 'small' L-layer deep neural network that shallower networks require exponentially more hidden units to compute".

Example: compute $X_1 \text{ XOR } X_2 \text{ XOR } \cdots \text{ XOR } X_N$

This can be done by pairing the inputs and $\log(N)$ layers:



If we attempted to do this with a single hidden layer, we would need an exponentially big number of hidden ~~too~~ units, which would basically exhaustively enumerate all input combinations:



Building Blocks of Deep Neural Networks

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

(1) Forward propagation

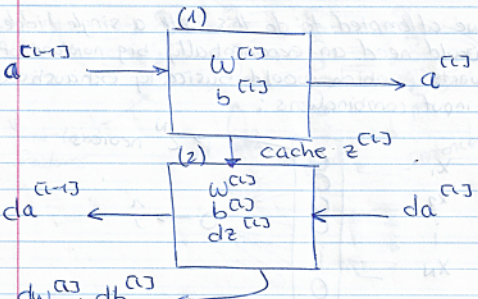
Input: $a^{[l-1]}$, Output: $a^{[l]} = g^{[l]}(z^{[l]})$

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

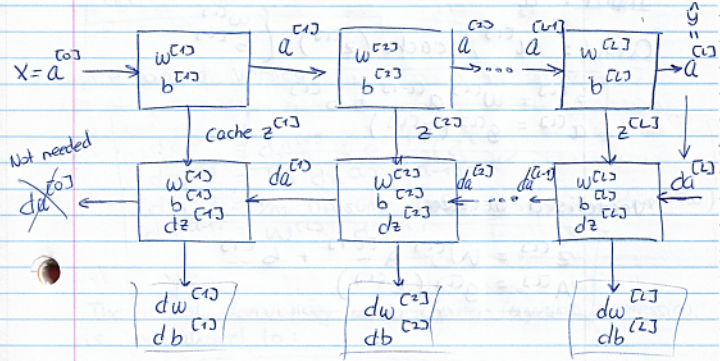
$z^{[l]}$ is cached

(2) Backpropagation

Input: $da^{[l]}$, $z^{[l]}$ Output: $da^{[l-1]}$, $dw^{[l]}$, $db^{[l]}$



That was for a single layer l. To compute forward and backpropagation for the whole network, we just repeat the process for all layers.



Having computed derivatives, we update parameters for all layers:

for $l = 1 .. L$:

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

Forward and Backward Propagation

Forward propagation for layer l

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

Output: $a^{[l]}$, cache $(z^{[l]})$ $\begin{cases} w^{[l]} \\ b^{[l]} \end{cases}$

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

Vectorised version:

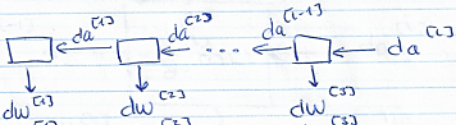
$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

$X = A^{[0]} \longrightarrow \square \rightarrow \square \rightarrow \cdots \rightarrow \square \rightarrow \hat{y}$

Backward propagation for layer l

Input: $da^{[l]}$

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



element-wise

$$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$$
$$dw^{[l]} = dz^{[l]} a^{[l-1]}$$
$$db^{[l]} = dz^{[l]}$$
$$da^{[l-1]} = w^{[l]T} dz^{[l]}$$

Vectorised Version:

$$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$$
$$dW^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$
$$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis=1, keepdims=True)$$
$$dA^{[l-1]} = W^{[l]T} dZ^{[l]}$$

The initial derivative, when doing logistic regression, is initialised to:

$$da^{[l]} = -\frac{y}{a} + \frac{(1-y)}{(1-a)}$$

The vectorised version $dA^{[l]}$ contains this same value but for all training examples:

$$dA^{[l]} = \left[da^{[l](0)} \; da^{[l](1)} \cdots da^{[l](m)} \right]$$

Parameters vs Hyperparameters

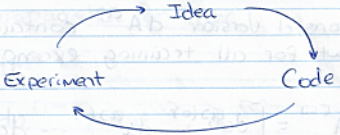
The parameters of our model are weights and biases:

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

Then there are other parameters that affect the learning algorithm, and that ultimately control the above parameters W, b. This last set of parameters we call hyperparameters, which include:

- Learning rate α
- # iterations
- # hidden layers L
- # hidden units $n^{[1]}, n^{[2]}, \ldots, n^{[L]}$
- Choice of activation functions

Determining the hyperparameters is an empirical process. You just need to play around and see what works.



There are more systematic approaches towards determining the hyperparameters that are seen in later courses.