Deep Learning - Backpropagation - Computational Graph

$x, w \in \mathbb{R}^{n_x}$
$b \in \mathbb{R}$
$z = w^T x + b$
$a = \hat{y} = \sigma(z) = \sigma(w^T x + b) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-(w^T x+b)}}$

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

if $\quad n_x = 2 \quad \Rightarrow \quad x, w \in \mathbb{R}^2$

$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$

$$\begin{tikzcd}
x_1 \arrow[rrdd] & & & & \\
w_1 \arrow[rrd] & & & & \\
x_2 \arrow[rr] & & \boxed{z = w_1x_1 + w_2x_2 + b} \arrow[r] & \boxed{a = \sigma(z)} \arrow[r] & \boxed{\mathcal{L}(a, y)} \\
w_2 \arrow[rru] & & & & \\
b \arrow[rruu] & & & &
\end{tikzcd}$$

$$\mathbf{da} = \frac{\partial \mathcal{L}}{\partial a} = \frac{\partial \mathcal{L}(a,y)}{\partial a} = \frac{\mathbf{a}-\mathbf{y}}{\mathbf{a}-\mathbf{a^2}} = -\frac{\mathbf{y}}{\mathbf{a}} + \frac{\mathbf{1}-\mathbf{y}}{\mathbf{1}-\mathbf{a}} = -\left(\frac{\mathbf{y}}{\mathbf{a}} - \frac{\mathbf{1}-\mathbf{y}}{\mathbf{1}-\mathbf{a}}\right) = -\frac{y}{\sigma(z)} + \frac{1-y}{1-\sigma(z)} = -\frac{y}{\frac{1}{1+e^{-(w^Tx+b)}}} + \frac{1-y}{1-\frac{1}{1+e^{-(w^Tx+b)}}}$$

$\mathbf{da} = -(np.divide(Y, A^{[l]}) - np.divide(1-Y, 1-A^{[l]}))$
da denotes the "**derivative** of the **loss** with respect to **a**"

$$\frac{\partial \mathbf{a}}{\partial \mathbf{z}} = \frac{\partial \sigma(z)}{\partial z} = \mathbf{a}(\mathbf{1}-\mathbf{a}) = \sigma(z)(1-\sigma(z)) = \frac{1}{1+e^{-(w^Tx+b)}}\left(1 - \frac{1}{1+e^{-(w^Tx+b)}}\right) = \frac{e^z}{(e^z+1)^2} = \frac{e^{(w^Tx+b)}}{(e^{(w^Tx+b)}+1)^2}$$

$\frac{\partial \mathbf{a}}{\partial \mathbf{z}}$ denotes the "**derivative** of the **activation function a** with respect to **z**"

$$\mathbf{dz} = \frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}(a,y)}{\partial z} = \frac{\partial \mathcal{L}(\sigma(z),y)}{\partial z} = \mathbf{a}-\mathbf{y} = \sigma(z) - y = \frac{1}{1+e^{-(w^Tx+b)}} - y$$

dz denotes the "**derivative** of the **loss** with respect to **z**"

$$= a - y = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} = \left(-\frac{y}{\frac{1}{1+e^{-(w^Tx+b)}}} + \frac{1-y}{1-\frac{1}{1+e^{-(w^Tx+b)}}}\right) \cdot \left(\frac{1}{1+e^{-(w^Tx+b)}}\left(1 - \frac{1}{1+e^{-(w^Tx+b)}}\right)\right)$$

$\mathbf{db} = \mathbf{dz}$

$$\mathbf{dw_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{w_1}} = \frac{\partial \mathcal{L}(a,y)}{\partial w_1} = \frac{\partial \mathcal{L}(\sigma(z),y)}{\partial w_1} = \mathbf{x_1} \cdot \mathbf{dz}$$

$$\mathbf{dw_2} = \frac{\partial \mathcal{L}}{\partial \mathbf{w_2}} = \frac{\partial \mathcal{L}(a,y)}{\partial w_2} = \frac{\partial \mathcal{L}(\sigma(z),y)}{\partial w_2} = \mathbf{x_2} \cdot \mathbf{dz}$$

Updating Params
$\mathbf{w_1} := \mathbf{w_1} - \alpha \mathbf{dw_1}$
$\mathbf{w_2} := \mathbf{w_2} - \alpha \mathbf{dw_2}$
$\mathbf{b} := \mathbf{b} - \alpha \mathbf{db}$

Deep Learning - Backpropagation - Logistic regression on m examples

$X \in \mathbb{R}^{n_x \times m}$ is the Set of training examples $x \in \mathbb{R}^{n_x}$

$i \in \{\ x \mid (x \in \mathbb{N}_+) \wedge (x \leq m)\} = [1, m] \subseteq \mathbb{N}_+$

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

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} \left[\left(y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right) \right) \right]$$

if $x \in \mathbb{R}^2$ and there are m training examples. the i'th training example x has 2 corresponding weigthts w

$\Rightarrow (x^{(i)}, y^{(i)}) = dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$

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

Algorithm - Should be implemented vectorized - No For Loops!

```
$J = 0;\ dw_1 = 0;\ dw_2 = 0;\ db = 0$
For $i$=1 to $m$
    $z^{(i)} = w^T x^{(i)} + b$
    $a^{(i)} = \sigma(z^{(i)})$
    $J \ += -\left[y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})\right]$
    $dz^{(i)} = a^{(i)} - y^{(i)}$

    $\left.\begin{array}{l} dw_1 \ += x_1^{(i)} dz^{(i)} \\ dw_2 \ += x_2^{(i)} dz^{(i)} \end{array}\right\}$   $n = 2 \rightarrow n$ can be any size

    $db \ += dz^{(i)}$
$J = \frac{J}{m}$
$dw_1 = \frac{dw_1}{m}$
$dw_2 = \frac{dw_2}{m}$
$db = \frac{db}{m}$
```

With all of these calculations, it computes the derivatives of the cost function J with respect to each parameters w_1, w_2 (of course up to n) and b. $\mathbf{dw_1}, \mathbf{dw_2}, \mathbf{db}$ are considered as **accumulators**

$\mathbf{dw_1} = \frac{\partial J}{\partial w_1}$
$\mathbf{dw_2} = \frac{\partial J}{\partial w_2}$

Notice that $\mathbf{dw_1}$ and $\mathbf{dw_2}$ do not have a superscript (i), because we're using them in this code as **accumulators** to **sum over the entire training set**.

Implementing 1 Step of Gradient Descent Note: α is the learning rate

$\mathbf{w_1} := \mathbf{w_1} - \alpha \mathbf{dw_1}$
$\mathbf{w_2} := \mathbf{w_2} - \alpha \mathbf{dw_2}$
$\mathbf{b} := \mathbf{b} - \alpha \mathbf{db}$

Everything on this page implements only 1 single step of Gradient Descent. In order to minimize the cost function everything needs to be performed multiple times.

L = number of layers in the network
$z = w^T x + b$
$l \in [1, L]$
$n_h^{[l]}$ = number of hidden units in Layer l
$a_j^{[l]}$=Activation Node in layer l in position j
$X \in \mathbb{R}^{n_x \times m}$
$i \in [1, m]$

$$X = \underbrace{\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nm} \end{bmatrix}}_{n_x \times m} = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$$W^{[l]} = \underbrace{\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n_h^{[l-1]}} \\ w_{21} & w_{22} & \dots & w_{2n_h^{[l-1]}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_h^{[l]}1} & w_{n_h^{[l]}2} & \dots & w_{n_h^{[l]}n_h^{[l-1]}} \end{bmatrix}}_{n_h^{(l)} \times n_h^{(l-1)}} = \text{Weights}$$

The Blue row shows $\left(w_1^{[l]}\right)^T$ = Weights vector corresponding to the first activation node in layer l.

$$B = \underbrace{\begin{bmatrix} b & b & \dots & b \end{bmatrix}}_{1 \times m}$$

$$Z = \underbrace{\begin{bmatrix} z^{(1)} & z^{(2)} & \dots & z^{(m)} \end{bmatrix}}_{1 \times m} = (w^{(l)})^T X + B = \underbrace{\begin{bmatrix} (w^{(l)})^T x^{(1)} + b & (w^{(l)})^T x^{(2)} + b & \dots & (w^{(l)})^T x^{(m)} + b \end{bmatrix}}_{1 \times m}$$

$Z = numpy.dot(w.T, X) + \underbrace{b}_{(1 \times 1) \in \mathbb{R}} \quad \leftarrow$ in Python
$a^{(1)} = \sigma(z^{(1)}) = \frac{1}{1+e^{-z^{(1)}}}$
$a^{(2)} = \sigma(z^{(2)})$
...
$A = \begin{bmatrix} a^{(1)} & a^{(2)} & \dots & a^{(m)} \end{bmatrix}$
$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m)} \end{bmatrix}$
$dZ = \begin{bmatrix} dz^{(1)} & dz^{(2)} & \dots & dz^{(m)} \end{bmatrix} = A - Y = \begin{bmatrix} a^{(1)} - y^{(1)} & a^{(2)} - y^{(2)} & \dots & a^{(m)} - y^{(m)} \end{bmatrix}$

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)} = \frac{1}{m} \cdot numpy.sum(dZ)$$

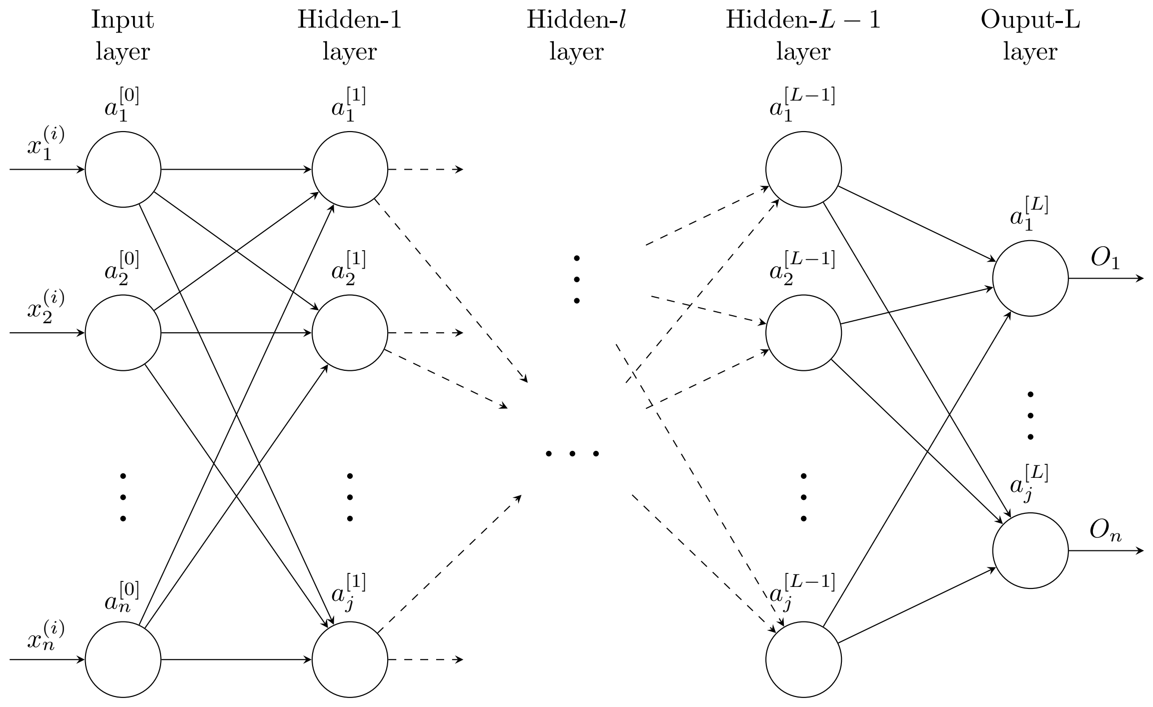
$$dw = \frac{1}{m} \underbrace{\begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}}_{n \times m} \underbrace{\begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}}_{m \times 1} = \begin{bmatrix} x^{(1)} dz^{(1)} & + \dots + & x^{(m)} dz^{(m)} \end{bmatrix} = \underbrace{\frac{1}{m} \cdot X \cdot (dz)^T}_{n \times 1}$$

Algorithm Vectorized - Forward + Backpropagation

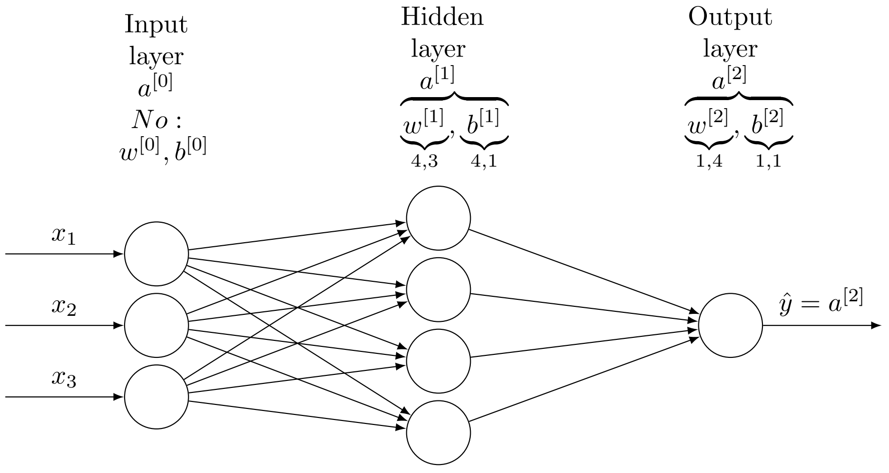
```
For  $i$=1  in  $range(1000)$ :
    $Z = numpy.dot(w.T, X) + b = (w^{(l)})^T X + b$
    $A = \sigma(Z)$
    $dZ = A - Y$
    $dw = \frac{1}{m} \cdot numpy.dot(X, (dZ)^T)$
    $db = \frac{1}{m} \cdot numpy.sum(dZ)$

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

Sum a matrix



Deep Learning - Backpropagation - Vectorization


Input layer $a^{[0]}$ $No:$ $w^{[0]}, b^{[0]}$
Hidden layer $a^{[1]}$ $\underbrace{w^{[1]}}_{4,3}, \underbrace{b^{[1]}}_{4,1}$
Output layer $a^{[2]}$ $\underbrace{w^{[2]}}_{1,4}, \underbrace{b^{[2]}}_{1,1}$
x_1
x_2
x_3
$\hat{y} = a^{[2]}$

Notes:

- The input layer of a Neural network does not count. If there is an input layer, hidden layer, output layer, we say it is a **"2 layer NN"**

$$W^{[l]} = \underbrace{\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n_h^{[l-1]}} \\ w_{21} & w_{22} & \cdots & w_{2n_h^{[l-1]}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_h^{[l]}1} & w_{n_h^{[l]}2} & \cdots & w_{n_h^{[l]}n_h^{[l-1]}} \end{bmatrix}}_{n_h^{[l]} \times n_h^{[l-1]}} = \text{Weights in layer } l \text{ for training example}$$

The blue row shows $\underbrace{\left(w_1^{[l]}\right)^T}_{1 \times n_h^{[l-1]}} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n_h^{[l-1]}} \end{bmatrix}$

Weights vector corresponding to the first activation node in layer l

In essence $W^{[l]}$ is a matrix stacked with $\left(w_j^{[l]}\right)^T$ vectors.

$\underbrace{a^{[l]}}_{n_h^{[l]}\times 1}$ = It is the [l] layer activation nodes $\underbrace{\begin{bmatrix} a_1^{[l]} \\ \vdots \\ a_j^{[l]} \end{bmatrix}}_{n_h^{[l]}\times 1} = \underbrace{\begin{bmatrix} \sigma\left(z_1^{[l]}\right) \\ \vdots \\ \sigma\left(z_j^{[l]}\right) \end{bmatrix}}_{n_h^{[l]}\times 1} = \underbrace{\begin{bmatrix} \sigma\left(\left(w_1^{[l]}\right)^T a^{[l-1]} + b_1^{[l]}\right) \\ \vdots \\ \sigma\left(\left(w_j^{[l]}\right)^T a^{[l-1]} + b_j^{[l]}\right) \end{bmatrix}}_{n_h^{[l]}\times 1}$

If $l = 0$ Corresponds to the input parameter X

j Corresponds to the j'th Node in layer l

$\underbrace{w_j^{[l]}}_{1\times n_h^{[l-1]}}$ Is the associated Weights vector, for this specific activation unit j in layer l

$\underbrace{b_j^{[l]}}_{1\times 1}$ Is the associated Bias vector, for this specific activation unit j in layer l

$\underbrace{W^{[l]}}_{n_h^{[l]}\times n_h^{[l-1]}}$ = Weights matrix, consists out of weight vectors $w_j^{[l]}$ for all activation units in layer l

$\underbrace{z^{[l]}}_{n_h^{[l]}\times 1} = W^{[l]}a^{[l-1]} + b^{[l]}$ Note, if $l = 0 \Rightarrow a^{[0]} = X$

$$z^{[l]} = \underbrace{\begin{bmatrix} z_1^{[l]} \\ \vdots \\ z_j^{[l]} \end{bmatrix}}_{n_h^{[l]}\times 1} = \underbrace{\begin{bmatrix} \underbrace{\left(w_1^{[l]}\right)^T}_{1\times n_h^{[l-1]}} \underbrace{a^{[l-1]}}_{n_h^{[l-1]}\times 1} + b_1^{[l]} \\ \vdots \\ \left(w_j^{[l]}\right)^T a^{[l-1]} + b_j^{[l]} \end{bmatrix}}_{n_h^{[l]}\times 1} = W^{[l]}a^{[l-1]} + b^{[l]}$$

$$\underbrace{b^{[l]}}_{n_h^{[l]}\times 1} = \underbrace{\begin{bmatrix} b_1^{[1]} \\ \vdots \\ b_j^{[1]} \end{bmatrix}}_{n_h^{[l]}\times 1}$$

$$\underbrace{a^{[l]}}_{n_h^{[l]}\times 1} = \sigma(z^{[l]})$$

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

If L is the number of layers in a NN, forwar propagation is calculated as follows:

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

$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma^{[2]}(z^{[2]})$
$\vdots$
$z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]}$
$a^{[L]} = \sigma^{[L]}(z^{[L]})$

For m training examples the following notation is applied:
$a^{[2](i)}$ where the square brackets [ ] denote the Layer and the round brackets ( ) denote the training example i.

ONE $x \in X$ after another Iterating through m training examples is done as follows:

for $i = 1$ to m:

$$\underbrace{z^{[1](i)}}_{n_h^{[1]}\times 1} = \underbrace{W^{[1]}}_{n_h^{[1]}\times n_h^{[0]}} \underbrace{a^{[0](i)}}_{n_h^{[0]}\times 1} + \underbrace{b^{[1]}}_{n_h^{[1]}\times 1}$$

$$\underbrace{a^{[1](i)}}_{n_h^{[1]}\times 1} = \underbrace{\sigma^{[1]}(z^{[1](i)})}_{n_h^{[1]}\times 1}$$

$$\underbrace{z^{[2](i)}}_{n_h^{[2]}\times 1} = \underbrace{W^{[2]}}_{n_h^{[2]}\times n_h^{[1]}} \underbrace{a^{[1](i)}}_{n_h^{[1]}\times 1} + \underbrace{b^{[2]}}_{n_h^{[2]}\times 1}$$

$$\underbrace{a^{[2](i)}}_{n_h^{[2]}\times 1} = \underbrace{\sigma^{[2]}(z^{[2](i)})}_{n_h^{[2]}\times 1}$$

$$\vdots$$
$$z^{[L](i)} = W^{[L]}a^{[L-1](i)} + b^{[L]}$$
$$a^{[L](i)} = \sigma^{[L]}(z^{[L](i)})$$

Vectorized Impl (For all m Training Examples, not for all Layer! - Remember how X is set up)

$$X = \underbrace{\begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}}_{n\times m}, Z^{[l]} = \underbrace{\begin{bmatrix} | & | & & | \\ z^{[l](1)} & z^{[l](2)} & \cdots & z^{[l](m)} \\ | & | & & | \end{bmatrix}}_{n_h^{[l]}\times m}, A^{[l]} = \underbrace{\begin{bmatrix} | & | & & | \\ a^{[l](1)} & a^{[l](2)} & \cdots & a^{[l](m)} \\ | & | & & | \end{bmatrix}}_{n_h^{[l]}\times m}$$

ALL $x \in X$ at the same time is done as follows:

$$\underbrace{Z^{[1]}}_{n_h^{[1]}\times m} = \underbrace{W^{[1]}}_{n_h^{[1]}\times n_h^{[0]}} \underbrace{A^{[0]}}_{n_h^{[0]}\times m} + \underbrace{b^{[1]}}_{n_h^{[1]}\times m}$$

$$\underbrace{A^{[1]}}_{n_h^{[1]}\times m} = \underbrace{\sigma^{[1]}(Z^{[1]})}_{n_h^{[1]}\times m}$$

$$\underbrace{Z^{[2]}}_{n_h^{[2]}\times m} = \underbrace{W^{[2]}}_{n_h^{[2]}\times n_h^{[1]}} \underbrace{A^{[1]}}_{n_h^{[1]}\times m} + \underbrace{b^{[2]}}_{n_h^{[2]}\times m}$$

$$\underbrace{A^{[2]}}_{n_h^{[2]}\times m} = \underbrace{\sigma^{[2]}(Z^{[2]})}_{n_h^{[2]}\times m}$$

$$\vdots$$
$$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$$
$$A^{[L]} = \sigma^{[L]}(Z^{[L]})$$

Replacing $\sigma(Z^{[l]})$ with $g(Z^{[1]})$ in order to make the activation function exchangeable.

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

Tanh activation function:
$g(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
$\frac{d}{dz}g(z) = g'(z) = 1 - (tanh(z))^2 = 1 - a^2 = (1 - numpy.power(a, 2))$

ReLU and **Leaky ReLU**:
ReLU:
$g(z) = max(0, z)$

$$\frac{d}{dz}g(z) = g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

Leaky ReLU:
$g(z) = max(0.01z, z)$

$$\frac{d}{dz}g(z) = g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

Cost Function: $J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}log\left(a^{[l](i)}\right) + \left(1-y^{(i)}\right)log\left(1-a^{[l](i)}\right)\right)$
$-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}log\left(a^{[l](i)}\right)\right) = -np.sum\left(np.multiply(np.log(A), Y)\right)$
$J = cost = -\frac{1}{m} * np.sum\left(np.multiply(np.log(A2), Y) + np.multiply(np.log((1-A2)), (1-Y))\right)$

Forward Propagation: ALL $x \in X$ at the same time is done as follows:

$$\underbrace{Z^{[1]}}_{n_h^{[1]}\times m} = \underbrace{W^{[1]}}_{n_h^{[1]}\times n_h^{[0]}} \underbrace{A^{[0]}}_{n_h^{[0]}\times m} + \underbrace{b^{[1]}}_{n_h^{[1]}\times m}$$

$$\underbrace{A^{[1]}}_{n_h^{[1]}\times m} = \underbrace{g^{[1]}(Z^{[1]})}_{n_h^{[1]}\times m}$$

$$\underbrace{Z^{[2]}}_{n_h^{[2]}\times m} = \underbrace{W^{[2]}}_{n_h^{[2]}\times n_h^{[1]}} \underbrace{A^{[1]}}_{n_h^{[1]}\times m} + \underbrace{b^{[2]}}_{n_h^{[2]}\times m}$$

$$\underbrace{A^{[2]}}_{n_h^{[2]}\times m} = \underbrace{g^{[2]}(Z^{[2]})}_{n_h^{[2]}\times m}$$

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

Backpropagation:
Note: Y must have the same dimensions as the last layer in the NN in order tell the network which of the activation output result nodes in the last layer is the expected one.
***** is the **is the element-wise** multiplication

$$Y = \underbrace{\begin{bmatrix} | & | & & | \\ y^{(1)} & y^{(2)} & \cdots & y^{(m)} \\ | & | & & | \end{bmatrix}}_{n_h^{[L]}\times m}$$

$$\underbrace{dZ^{[L]}}_{n_h^{[L]}\times m} = \underbrace{A^{[L]}}_{n_h^{[L]}\times m} - \underbrace{Y}_{n_h^{[L]}\times m}$$

$$\underbrace{dW^{[L]}}_{n_h^{[L]}\times n_h^{[L-1]}} = \frac{1}{m}\underbrace{dZ^{[L]}}_{n_h^{[L]}\times m}\underbrace{A^{[L-1]^T}}_{m\times n_h^{[L-1]}}$$

$$\underbrace{db^{[L]}}_{(n_h^{[L]}\times m)?} = \frac{1}{m}numpy.sum(dZ^{[L]}, axis=1, keepdims=True)$$

$$\underbrace{dZ^{[L-1]}}_{n_h^{[L-1]}\times m} = \underbrace{W^{[L]^T}}_{n_h^{[L-1]}\times n_h^{[L]}}\underbrace{dZ^{[L]}}_{n_h^{[L]}\times m} * \underbrace{\overbrace{g^{[L-1]'}\left(Z^{[L-1]}\right)}^{=dA^{[L-1]}=-(np.divide(Y,A^{[l]})-np.divide(1-Y,1-A^{[l]}))}}_{n_h^{[L-1]}\times m}$$

$$\underbrace{dW^{[L-1]}}_{n_h^{[L-1]}\times n_h^{[L-2]}} = \frac{1}{m}\underbrace{dZ^{[L-1]}}_{n_h^{[L-1]}\times m}\underbrace{A^{[L-2]^T}}_{m\times n_h^{[L-2]}}$$

$\vdots$
Eventually $W^{[l]}$ and $b^{[l]}$ are updated as follows:
$W^{[l]} := W^{[l]} - \alpha dW^{[l]}$
$b^{[l]} := b^{[l]} - \alpha db^{[l]}$

Cost Function with L2 Regularization:

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

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}log\left(a^{[l](i)}\right)+\left(1-y^{(i)}\right)log\left(1-a^{[l](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{\lambda}{2m}\sum_{l}\sum_{k}\sum_{j}\left(W_{k,j}^{[l]}\right)^2}_{\text{L2 regularization cost}}$$

$$L2_{regularizationCost} = \frac{\lambda}{2m}\sum_{l}\sum_{k}\sum_{j}\left(W_{k,j}^{[l]}\right)^2 =$$

```
$L2_{regularizationCost} = 0$
FOR $l = 1$ to $\#L$ :
    $L2_{regularizationCost} \mathrel{+}= np.sum(np.square("W" + str(l)))$
END FOR
$L2_{regularizationCost} = \frac{\lambda}{2m} * L2_{regularizationCost}$
```

Backpropagation needs also to be adjusted by adding the regularization term's gradient $\left(\frac{d}{dW}(\frac{\lambda}{2m}W^2)\right) = \frac{\lambda}{m}W$

Go From:

$$\underbrace{dW^{[L]}}_{n_h^{[L]}\times n_h^{[L-1]}} = \frac{1}{m}\underbrace{dZ^{[L]}}_{n_h^{[L]}\times m}\underbrace{A^{[L-1]^T}}_{m\times n_h^{[L-1]}}$$

To

$$\underbrace{dW^{[L]}}_{n_h^{[L]}\times n_h^{[L-1]}} = \frac{1}{m}\underbrace{dZ^{[L]}}_{n_h^{[L]}\times m}\underbrace{A^{[L-1]^T}}_{m\times n_h^{[L-1]}} + \frac{\lambda}{m}W^{[L]}$$

What is L2-regularization actually doing?:
L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.

What you should remember – the implications of L2-regularization on:

- The cost computation:
  - A regularization term is added to the cost
- The backpropagation function:
  - There are extra terms in the gradients with respect to weight matrices
- Weights end up smaller ("weight decay"):
  - Weights are pushed to smaller values.

Mini-batch gradient descent:
NOTE: $\|\cdot\|_F$ is the **Frobeniusnorm**, it is simply squares each element in the matrix, adds them up and takes the squareroot.

Example:

$$A = \begin{pmatrix} 1 & 2 & 1 \\ -1 & 2 & -3 \\ 0 & 1 & -2 \end{pmatrix}$$

$$\|A\|_F^2 = \left(\sqrt{\sum_{i=1}^{3} \sum_{j=1}^{3} (a_{ij})^2} \right)^2 = \left(\sqrt{1^2 + 2^2 + 1^2 + (-1)^2 + 2^2 + (-3)^2 + 0^2 + 1^2 + (-2)^2} \right)^2 = \left(\sqrt{25} \right)^2 = (5)^2 = 25$$

Divide X into chunks, in this example 5000 chunks.
Adress each chunk via notation $^{\{t\}}$

```
For $t = 1, ..., 5000$ Do:{

  **Forward prop** on $X^{\{t\}}$

      $\underbrace{Z^{[1]}}_{n_h^{[1]} \times m} = \underbrace{W^{[1]}}_{n_h^{[1]} \times n_h^{[0]}} \underbrace{X^{\{t\}}}_{n_h^{[0]} \times chunksize} + \underbrace{b^{[1]}}_{n_h^{[1]} \times m}$

      $\underbrace{A^{[1]}}_{n_h^{[1]} \times m} = \underbrace{g^{[1]}(Z^{[1]})}_{n_h^{[1]} \times m}$

      $\underbrace{Z^{[2]}}_{n_h^{[2]} \times m} = \underbrace{W^{[2]}}_{n_h^{[2]} \times n_h^{[1]}} \underbrace{A^{[1]}}_{n_h^{[1]} \times m} + \underbrace{b^{[2]}}_{n_h^{[2]} \times m}$

      $\underbrace{A^{[2]}}_{n_h^{[2]} \times m} = \underbrace{g^{[2]}(Z^{[2]})}_{n_h^{[2]} \times m}$
      $\vdots$
      $Z^{[L]} = W^{[L]} A^{[L-1]} + b^{[L]}$
      $A^{[L]} = g^{[L]}(Z^{[L]})$
   Compute cost $J = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2 \cdot 1000} \sum_l \|W^{[2]}\|_F^2$

  **Backprop** to compute gradients with respect to $J^{\{t\}}$ using $(X^{\{t\}}, Y^{\{t\}})$

  $W^{[l]} := W^{[l]} - \alpha dW^{[l]}, b^{[l]} := b^{[l]} - \alpha db^{[l]}$
}
```

The code above states **1 epoch**. 1 Epoche denotes 1 pass with the entire training set through the NN. In a non-batched gradient descent, one epoch is one gradient descent step. In batch gradient descent it corresponds to however the chunksize is gradient descent steps.
In practice the code above is nested within another for loop until the code converges.
Note: With a large training set, **mini gradient descent runs much faster than without chunking**

Neural Network and Convolutional Neural Network Parameter Dimensions

Convolution Result Matrix Formula (NO PADDING):

Matrix A with dimensions $(n \times n)$
  Example: $n = 6 \rightarrow (6 \times 6)$
Filter B with dimensions $(f \times f)$
  Example: $f = 3 \rightarrow (3 \times 3)$
Convolution: $A * B = C$
C dimensions: $((\mathbf{n} - \mathbf{f} + \mathbf{1}) \times (\mathbf{n} - \mathbf{f} + \mathbf{1})) = ((6 - 3 + 1) \times (6 - 3 + 1)) = (4 \times 4)$

Convolution Result Matrix Formula WITH PADDING:

Matrix A with dimensions $(n \times n)$
  Example: $n = 6 \rightarrow (6 \times 6)$

Padding $\mathbf{p = 1}$ for Matrix $A \rightarrow ((n + 2 * p) \times (n + 2 * p)) = ((6 + 2 * 1) \times (6 + 2 * 1)) = (8 \times 8)$

  Example: $f = 3 \rightarrow (3 \times 3)$
Convolution: $A * B = C$

C dimensions:
$(((\mathbf{n} + \mathbf{2} * \mathbf{p}) - \mathbf{f} + \mathbf{1}) \times ((\mathbf{n} + \mathbf{2} * \mathbf{p}) - \mathbf{f} + \mathbf{1}))$
$= (((6 + 2 * 1) - 3 + 1) \times ((6 + 2 * 1) - 3 + 1))$
$= ((8 - 3 + 1) \times (8 - 3 + 1))$
$= (6 \times 6)$

Determining the SAME Convolution:

Goal: Finding a Padding size p in order to preserve the input Matrix A dimensions.

Note:
  f is the Dimension Size from the filter applied, for instance if a Filter has the dimension of (3×3) then $f = 3$
  f is usually an odd number!

$$p = \frac{f-1}{2}$$

Determining Striding dimensions s

The stride **s** determines the number of cells which are jumped over when performing Strided Convolution.
Calculating the output dimension is performed as follows:

$$\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$$

Convolutions over Volume

PREREQUISITE: The value of n_c must be equal in the input and in the filter!

Case:
INPUT: You have an image as input whith dimensions $\mathbf{n} \times \mathbf{n} \times \mathbf{n_c}$

FILTER: There are $\mathbf{n_c^{'}}$ Filters by which the input image is convolved with.
Note $\mathbf{n_c^{'}}$ is simply the number of Filters the input image is being applied to, also called the **depth**. This means $n_c^{'}$ defines the number of features being detected in the input. Each filter handles different detection mechanisms, for instance one filter for vertical edge detection, one filter for horizontal edge detection etc.

Each filter has the dimensions $\mathbf{f} \times \mathbf{f} \times \mathbf{n_c}$

Convolution Output:
Case one: No padding $\rightarrow p = 0$ and Stride $\rightarrow s = 1$:
Output Dimension: $(n - f + 1) \times (n - f + 1) \times n_c^{'}$

Case two: Padding $\rightarrow p = 1$ and Stride $\rightarrow s = 1$:
Output Dimension: $(n + 2p - f + 1) \times (n + 2p - f + 1) \times n_c^{'}$

Case three: Padding $\rightarrow p = 1$ and Stride $\rightarrow s = 2$:
Output Dimension: $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times n_c^{'}$

Summary of notation

$f^{[l]} = filtersize$
$p^{[l]} = padding$
$s^{[l]} = stride$
$n_c^{[l]}$ = number of filters

$f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$ = Volume of each Filter, $n_c^{[l-1]}$ is equal to the third input dimension!

$a^{[l]} = g(Z^{[l]}) = g(a^{[l-1]} * W^{[l]} + b) = ReLU(Z^{[l]}) \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ = Activations Dimension

$A^{[l]} \rightarrow m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ = Activations Dimension Batch Gradient Descent

$n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$ = INPUT VOLUME SIZE OF LAYER l

$n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ = OUTPUT VOLUME SIZE OF LAYER l

$W^{[l]} = f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]} \rightarrow$ Weights / Parameters in layer l

$n_c^{[l]} \rightarrow$ bias dimension

$$n_H^{[l]} = \left\lfloor \frac{n_H^{[l-1]}+2p-f}{s} + 1 \right\rfloor = \text{Height Dimension in Layer } l$$

$$n_W^{[l]} = \left\lfloor \frac{n_W^{[l-1]}+2p-f}{s} + 1 \right\rfloor = \text{Width Dimension in Layer } l$$

0. Determine the number of parameters in $w^{[1]}$ if the Neural Network has an input image with dimensions $1000 \times 1000 \times 3 = 3,000,000$ and the number of hidden units in the fist layer is $n_h^{[1]} = 1000$. Thus the dimensions of the input layer are $(3million, 1)$ and the first hidden layer $(1000, 1)$.
As per definition the dimension of $w^{[1]}$ =(# of elements in $n_h^{[1]}$, # of elements in input layer)

The resulting number of parameters in $w^{[1]} = 3,000,000 * 1,000 = 3$ billion

1. Determine the number of parameters (including the bias parameters) in the first hidden layer of a Neural Network (no convolutional network).

Input layer size $n \times m$ Pixel RGB Image with (c Channels).
First Hidden layer contains l neurons
Each neuron is fully connected to the input

Formula:
$\#NoOfParamsInHiddenLayer = ((n \cdot m \cdot c) + 1) \cdot l$

Example:

Your input is a 300×300 color (RGB) image.
The first hidden layer has 100 neurons.
Each neuron is fully connected to the input, how many parameters does this hidden layer have? (including the bias parameter)

Solution:

$27,000,100 = ((300 \cdot 300 \cdot 3) + 1) \cdot 100.$

2. Determine the number of parameters (including the bias parameters) in the first hidden layer of a Neural Network is a convolution layer with f filters.