# Deep Learning

Saideep

June 2019

## 1 Introduction

This is a tutorial on the basics of Deep Learning based on the Deep Learning Specialization on Coursera. Let us begin with a classic classification problem and keep building on it. Let us also get familiar with the notations that we are going to use throughout.

Classification Problem:

Take the example of image classification where you have to identify the presence of cat in an image. The dimensions of the image being considered are: 64x64x3 (3 channels for R, B, G).

Figure 1: Sample Image

$n_x = 64*64*3 = 12288$
(x,y) where $x \in R^{n_x}$ & $y \in \{0,1\}$
m training examples $\{ (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}) \ldots (x^{(m)}, y^{(m)}) \}$
$m_{train}$ = # of training examples ; $m_{test}$ = # of test examples

$$X = \begin{bmatrix} \vdots & \vdots & \vdots & \cdots & \vdots \\ x^{(1)} & x^{(1)} & x^{(1)} & \cdots & x^{(m)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \end{bmatrix} \quad X.shape = (n_x,m)$$

$Y = [y^{(1)}, y^{(2)}, y^{(3)} \ldots y^{(m)}]$ Y.shape = (1,m)

## 2 Logistic Regression

$\widehat{y} = P(y = 1/x); \quad 0 \leq \widehat{y} \leq 1 \quad x \in R^{n_x}$

parameters: w $\in R^{n_x}$, b $\in$ R
Output = $\sigma$ ($w^T x + b$)

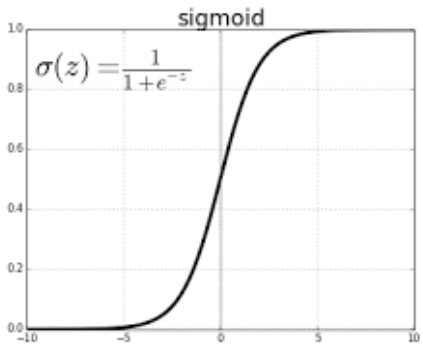



Figure 2: Sigmoid Function

## 2.1 Cost function:

L($\widehat{y}, y$) = $1/2(\widehat{y} - y)^2$ => non-convex
$\widehat{y} = \sigma$ ($w^T x + b$), $\sigma$(z) = $1/(1 + exp(-z))$
Hence, L($\widehat{y}, y$) = $-(ylog(\widehat{y}) + (1-y)log(1-\widehat{y}))$
Cost function over entire training set:

J(w,b) = $1/m(\sum_{i=1}^{m} L(\widehat{y^{(i)}}, y^{(i)}))$
= $-1/m \sum_{i=1}^{m} (y^{(i)} log(\widehat{y^{(i)}}) + (1-y^{(i)}) log(1-\widehat{y^{(i)}}))$

## 2.2 Gradient Descent:

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

**Gradient Descent on one training example:**

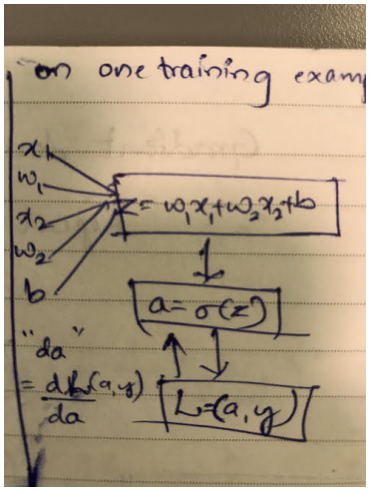



Figure 3:

$da = -y/a + (1-y)/a$
$dz = dL/dz = (dL/da) * (da/dz) = a - y$
$dL/dw_1 = dw_1 = x_1 dz$
$dL/dw_2 = dw_2 = x_2 dz$
$db = dz$
$w_1 := w_1 - \alpha dw_1$; $w_2 := w_2 - \alpha dw_2$; $b := b - \alpha db$

**Gradient Descent on m training examples:**
J(w,b) $= 1/m \sum_{i=1}^{m} L(a^{(i)}, y)$
$a^{(i)} = \widehat{y^{(i)}} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$

**for** i = 1 to m **do**
$z^i = w^T x^{(i)} + b$
$a^{(i)} = \sigma(z^{(i)})$
J += -$[y^{(i)} log(\widehat{y^{(i)}}) + (1 - y^{(i)}) log(1 - \widehat{y^{(i)}})]$
$dz^{(i)} = a^{(i)} - y^{(i)}$
$dw_{(i)} \mathrel{+}= x_1^{(i)} dz^{(i)}$
$dw_{(2)} \mathrel{+}= x_2^{(i)} dz^{(i)}$ ▷ n=2
$db \mathrel{+}= dz^{(i)}$
**end for**
$J/ = m$ $dw_1/ = m$ $dw_2/ = m$
$dw_1 = dJ/dw_1$
$w_1 := w_1 - \alpha dw_1$; $w_2 := w_2 - \alpha dw_2$; $b := b - \alpha db$

## 2.3 Vectorizing Logistic Regression:

$z^{(1)} = w^T x^1 + b \quad z^{(2)} = w^T x^2 + b \quad z^{(3)} = w^T x^3 + b$
$a^{(1)} = \sigma(z^{(1)}) \quad a^{(2)} = \sigma(z^{(2)}) \quad a^{(3)} = \sigma(z^{(3)})$

$$X = \begin{bmatrix} \vdots & \vdots & \vdots & \dots & \vdots \\ x^{(1)} & x^{(1)} & x^{(1)} & \dots & x^{(m)} \\ \vdots & \vdots & \vdots & \dots & \vdots \end{bmatrix} \quad \text{X.shape} = (n_x, m)$$

Z = [$z^{(1)}$, $z^{(2)}$, $z^{(3)}$ ... $z^{(m)}$] Z.shape = (1,m)

$$= w^T X + [b, b, b, ...b]_{1xm}$$

$$= w^T \begin{bmatrix} \vdots & \vdots & \vdots & \dots & \vdots \\ x^{(1)} & x^{(1)} & x^{(1)} & \dots & x^{(m)} \\ \vdots & \vdots & \vdots & \dots & \vdots \end{bmatrix} + b$$

$$= \begin{bmatrix} \vdots & \vdots & \vdots & \dots & \vdots \\ w^T x^{(1)} + b & w^T x^{(1)} + b & w^T x^{(1)} + b & \dots & w^T x^{(m)} + b \\ \vdots & \vdots & \vdots & \dots & \vdots \end{bmatrix}$$

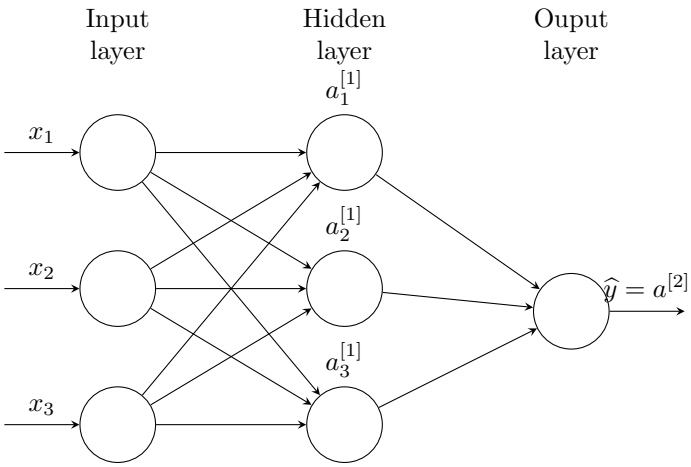
In python,
$Z = np.dot(w.T, X) + b_{(1,1)}$

$dz^{(1)} = a^{(1)} - y^{(1)}$ $dz^{(2)} = a^{(2)} - y^{(2)}$
$dZ = [dz^{(1)}, dz^{(2)}, dz^{(3)} ... dz^{(m)}]_{(1,m)}$
$A = [a^{(1)}, a^{(2)}, a^{(3)} ... a^{(m)}]$ $A.shape = (1, m)$
$Y = [y^{(1)}, y^{(2)}, y^{(3)} ... y^{(m)}]$ $Y.shape = (1, m)$

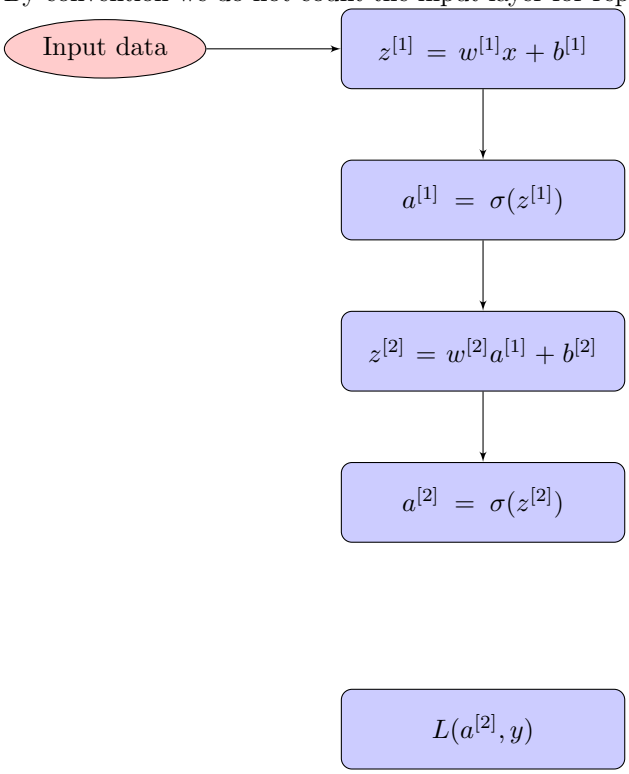
So,
$Z = w^T X + b = np.dot(w.T, X) + b$
$A = \sigma(Z)$
$dZ = A - Y$
$dw = (XdZ^T)/m$
$db = np.sum(dZ)/m$
$w := w - \alpha dw$ $b := b - \alpha db$

# 3 Neural Networks:

Using these ideas, let us now build a simple neural network and learn some notations as well.




superscript $^{[1],[2]}$ to denote the layer of input & subscript $_{1,2}$ to denote the neuron number in the layer.
By convention we do not count the input layer for reporting the number of layers in the neural network.



## 3.1 For one training example:

$z^{[1]} = w_1^{[1]T}x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$
$z^{[2]} = w_2^{[1]T}x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$
$z^{[3]} = w_3^{[1]T}x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$
$z^{[4]} = w_4^{[1]T}x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$

$$\begin{bmatrix} \dots & w_1^{[1]T} & \dots \\ \dots & w_2^{[1]T} & \dots \\ \dots & w_3^{[1]T} & \dots \\ \dots & w_4^{[1]T} & \dots \end{bmatrix}_{4x3} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{3x1} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}_{4x1}$$

$$= \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}_{4x1} = z^{[1]}$$

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}_{4x1} = \sigma(z^{[1]})$$

Given input x,

$Z^{[1]}_{4x1} = W^{[1]}_{4x3}\, x_{3x1} + b^{[1]}_{4x1}$
$a^{[1]} = \sigma(z^{[1]})$
$Z^{[2]}_{1x1} = W^{[2]}_{1x4}\, a^{[1]}_{4x1} + b^{[2]}_{1x1}$
$a^{[2]} = \sigma(z^{[2]})$

## 3.2 Vectorizing across multiple examples:

We denote $x^{(i)}$ for $i^{th}$ training example,

### 3.2.1 Non-vectorized :

**for** i = 1 to m **do**
$z^{[1](i)} = w^{[1]}x^{(i)} + b^{[1]}$
$a^{[1](i)} = \sigma(z^{[1](i)})$
$z^{[2](i)} = w^{[2]}a^{[1](i)} + b^{[2]}$
$a^{[2](i)} = \sigma(z^{[2](i)})$
**end for**

### 3.2.2 Vectorized :

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

$$X = \begin{bmatrix} \vdots & \vdots & \vdots & \dots & \vdots \\ x^{(1)} & x^{(1)} & x^{(1)} & \dots & x^{(m)} \\ \vdots & \vdots & \vdots & \dots & \vdots \end{bmatrix} \quad X.shape = (n_x, m)$$

$$\text{Training examples} \longrightarrow$$

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

$$A = [a^{[1](1)}, a^{[1](2)}, ...a^{[1](m)}] \qquad \downarrow \text{Hidden Units}$$

## 3.3 Activation Functions :

**tanh**

$g(z^{[1]}) = tanh(z^{[1]}); tanh(z) = (e^z - e^{-z})/(e^z + e^{-z})$

**ReLU :** Rectified Linear Unit

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

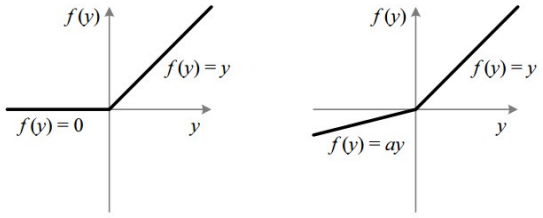



Figure 4: ReLU and Leaky ReLU

## 3.4 Back Propagation

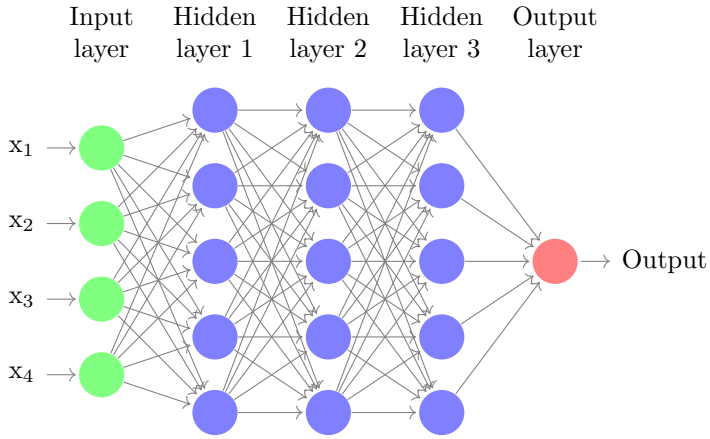
### 3.4.1 Forward Propagation

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

### 3.4.2 Back Propagation

$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dZ^{[2]}A^{[1]T}/m$
$db^{[2]} = np.sum(dZ^{[2]}A^{[1]T}, axis = 1, keepdims = True)/m$
$dZ^{[1]} = W^{[2]T}dZ^{[2]} \overbrace{*}^{elem-wise-product} g^{[1]'}(Z^{[1]})$
$dW^{[1]} = dZ^{[1]}X^T/m$
$db^{[1]} = np.sum(dZ^{[1]}, axis = 1, keepdims = True)/m$

# 4 Deep Neural Networks




$z^{[1]} = w^{[1]}x + b^{[1]}$
$a^{[1]} = g^{[1]}(z^{[1]})$
$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = g^{[2]}(z^{[2]})$
.
.
.
$z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$
$a^{[4]} = g^{[4]}(z^{[4]}) = \widehat{y}$

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

## 4.1 Forward Propagation for all training examples:

**for** i = 1 to 4 **do**
  $Z^{[\mathrm{i}]} = W^{[\mathrm{i}]}A^{[\mathrm{i\text{-}1}]} + b^{[\mathrm{i}]}$ ▷ $X = A^{[0]}$
  $A^{[i]} = g^{[i]}(Z^{(i)})$ ▷ $A^{[4]} = \widehat{Y} = g^{[4]}(Z^{[4]})$
**end for**

Dimensionality check,
For single training example -
$Z^{[l]}_{(n^{[l]},1)} = W^{[l]}_{(n^{[l]},n^{[l-1]})}A^{[l-1]}_{(n^{[l-1]},1)} + b^{[l]}_{(n^{[l]},1)}$ ($W$ and $dW$ will have the same dimensions)
For all training examples -
$A^{[l]}\&Z^{[l]} - (n^{[l]}, m)$
$W^{[l]} - (n^{[l]}, n^{[l+1]})$
$b^{[l]} - (n^{[l]}, 1)$ (python broadcasts hence it will add to each column when added to matrix)

## 4.2 Backward Propagation

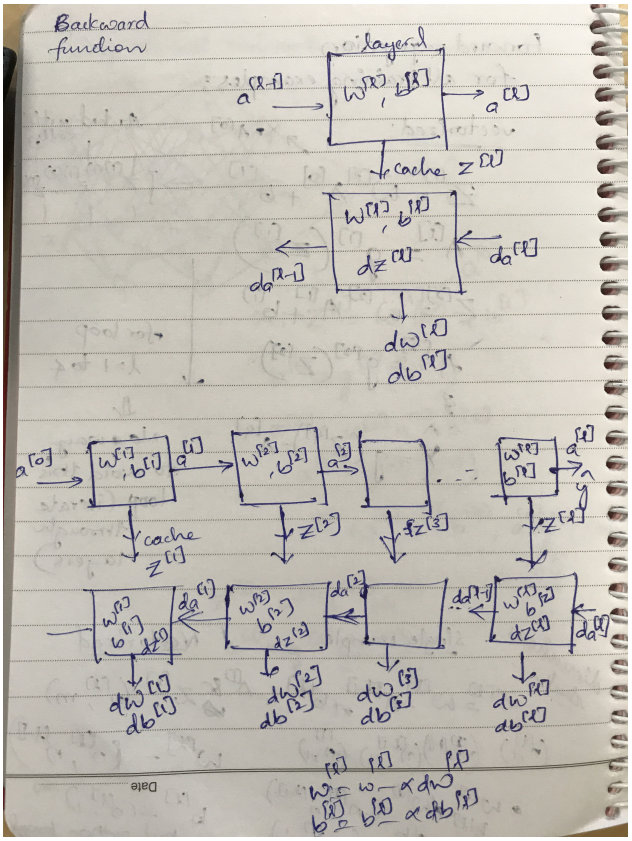



Figure 5:

## 4.3 Backward Propagation for layer l:

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

$dz^{[l]} = da^{[l-1]} * g^{[l]'}(z^{[l]})$
$dw^{[l]} = dz^{[l]} a^{[l-1]}$
$db^{[l]} = dz^{[l]}$
$da^{[l-1]} = W^{[l]T} dz^{[l]}$
$da^{[l]} = (-y)/a + (1-y)/(1-a)$

## 4.4 Vectorized version for layer l (all training examples:

$dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$
$dW^{[l]} = dZ^{[l]} A^{[l-1]T}$
$db^{[l]} = np.sum(dZ^{[l]}, axis = 1, keepdims = True)/m$
$dA^{[l-1]} = W^{[l]T} dZ^{[l]}$

$dA^{[l]} = (\frac{-y^{[(1)]}}{a^{(1)}} + \frac{1-y^{(1)}}{1-a^{(1)}}, \frac{-y^{[(2)]}}{a^{(2)}} + \frac{1-y^{(2)}}{1-a^{(2)}}, ... \frac{-y^{[(m)]}}{a^{(m)}} + \frac{1-y^{(m)}}{1-a^{(m)}})$

# 5 Convolutional Neural Networks

## 5.1 Applications in Computer Vision:

* Classification
* Position Identification
* Edge detection

**Convolution operator:**

| | | |
|---|---|---|
| 3 | 0 | 1 |
| 1 | 5 | 8 |
| 2 | 7 | 2 |

$*$

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 1 | 0 | -1 |
| 1 | 0 | -1 |

$= 3*1+1*1+2*1+0*0+5*0+1*0+1*-1+8*-1+2*-1 =$ -5

**Vertical Edge detection:**

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 1 | 0 | -1 |
| 1 | 0 | -1 |

**Horizontal Edge detection:**

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 0 | 0 |
| -1 | -1 | -1 |

## 5.2 CNNs:

In Deep Learning we try to learn the weights for the filter/kernel:

| | | |
|---|---|---|
| $w_1$ | $w_2$ | $w_3$ |
| $w_4$ | $w_5$ | $w_6$ |
| $w_7$ | $w_8$ | $w_9$ |

If we convolve a n*n image with f*f filter the resulting image has the dimensions (n-f+1)*(n-f+1), i.e. (i) it shrinks the output as well as (ii) some pixels are used less than the others when we convolve (edges).

**Padding**

When we pad the image by p, image size would be (n+2p) hence output of the convolved image is (n+2p-f+1)

**Valid Convolution:**

* No padding

(nxn)*(fxf) -> (n-f+1)x(n-f+1)

**Same Convolution:**

*Pad so that the output size is same as the input size.

=> n+2p-f+1 = n

=> p = (f-1)/2

f is almost always chosen to be odd probably because (i) asymmetry in padding when the filter size is even (ii) positioning of filter (it's nice to have a central pixel)

**Strided Convolution:**

Steps of convolution,

stride(s) =2 means jump the window/filter by 2 steps

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

**Convolutions on RGB images**

n x n x $n_c$ * f x f x $n_c$ –> (n-f+1)x(n-f+1)  $n_c$ = depth/# of channels

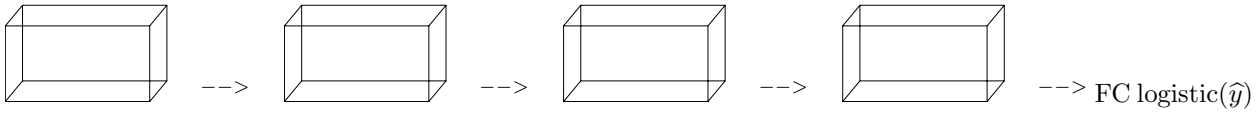
## 5.3 One CNN layer:

$d^{[l]}$ = filter size
$p^{[l]}$ = padding
$s^{[l]}$ = stride
$n_c^{[l]}$ = number of filters
Input : $n_H^{[l-1]}$ x $n_W^{[l-1]}$ x $n_c^{[l-1]}$
Output : $n_H^{[l]}$ x $n_W^{[l]}$ x $n_c^{[l]}$

$$n_{H/W}^{[l]} = \left\lfloor \frac{n_{H/W}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

Each filter is : $f^{[l]}$ x $f^{[l]}$ x $n_c^{[l-1]}$
Activations :
$a^{[l]} -> n_H^{[l]}$ x $n_W^{[l]}$ x $n_c^{[l]}$
$A^{[l]} -> m$ x $n_H^{[l]}$ x $n_W^{[l]}$ x $n_c^{[l]}$
Weights : $f^{[l]}$ x $f^{[l]}$ x $n_c^{[l-1]}$ x $n_c^{[l]}$ - > # of filters in layer l
bias : $n_c^{[l]} - (1, 1, 1, n_c^{[l]})$

## 5.4 Typical CNN example




39x39x3 $\overbrace{10filters}^{f^{[1]}=3, s^{[1]}=1, p^{[1]}=0}$ 37x37x10 $\overbrace{20filters}^{f^{[2]}=5, s^{[2]}=2, p^{[2]}=0}$ 17x17x20 $\overbrace{40filters}^{f^{[3]}=5, s^{[3]}=2, p^{[2]}=0}$ 7x7x40

### 5.4.1 Types of layer in CNN

**Convolution (CONV)**
**Pooling (POOL)**
**Fully Connected (FC)**

**Pooling Layer : Max Pooling**

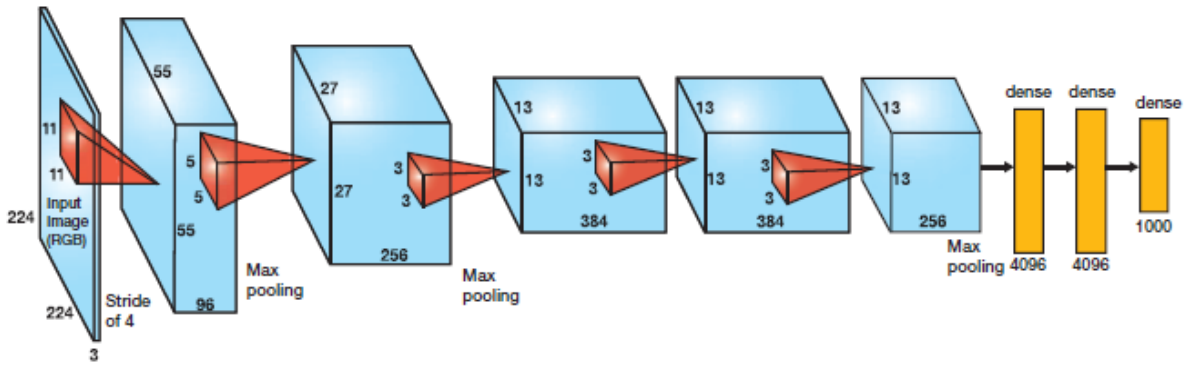
| | | | |
|---|---|---|---|
| 1 | 3 | 2 | 1 |
| 2 | 9 | 1 | 1 |
| 1 | 3 | 2 | 3 |
| 5 | 6 | 1 | 2 |

$\overbrace{-->}^{f=2, s=2}$

| | |
|---|---|
| 9 | 2 |
| 6 | 3 |

**Case Studies:**
Classic Networks:
LeNet - 5
AlexNet
VGG
ResNet (152)

**AlexNet**

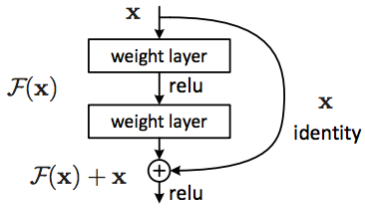


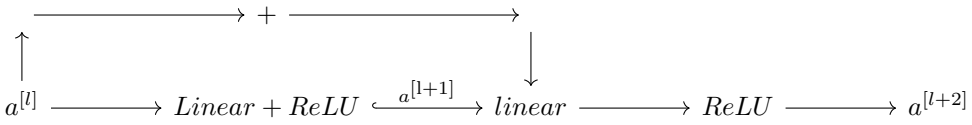

**VGG**

Here, CONV = 3x3 filter, s=1, same & max pool = 2x2, s=2


**ResNet**

Residual Networks



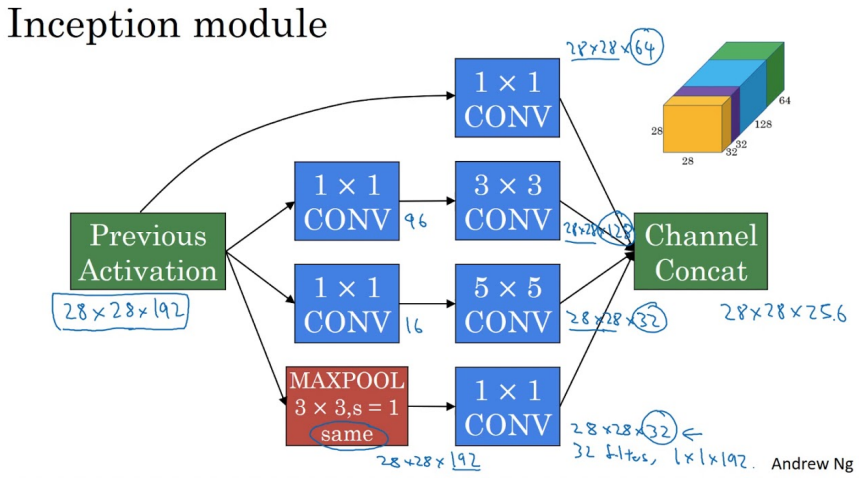

$$z^{[l+1]} = w^{[l+1]}a^{[l]} + b^{[l+1]} \quad a^{[l+1]} = g(z^{[l+1]})$$

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

Why ResNets work?

As we go deeper & deeper it becomes difficult to learn parameters but it's easy to learn identity when residual block is added.

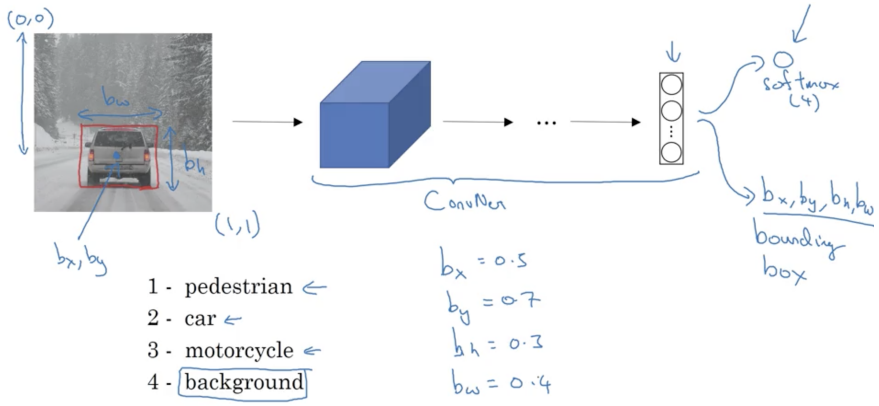
**Inception network**
Instead of choosing one filter size, use all sizes in the network.




# 6 Detection Algorithms:

## 6.1 Object Localization

In Object localization, instead we try to predict the x, y co-ordinates of the bounding box as well as the relative width and height of the box.




$$\text{Target label} = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

$p_c -> 0$ {background}
1 {car, pedestrian or motor-cycle}
$c_1 ->$ pedestrian/not
$c_2 ->$ car/not
$c_3 ->$ motor cycle/not

Loss function:
$L(y, \widehat{y}) = (\widehat{y_1} - y_1)^2 + (\widehat{y_2} - y_2)^2 + ... + (\widehat{y_8} - y_8)^2$

For landmark detection, one needs to annotate all the key points on the face.
Sliding window implementation - high computation cost
Convolutional implementation of sliding windows

YOLO (You Only Look Once) algorithm : You apply the localization algorithm to each of the grids.
Anchor boxes for multiple objects, where each object in the training image is assigned to grid cell that contains the object's mid-point & anchor box for the grid cell with highest IoU.