Convolution Neural Nets
Part 1: Forward Propagation

Convolution:
Simple Convolution: H×W image with h×h filter: One matrix convolved by another matrix. Output is **H×W**

```
def convHWpad(im,f):

    H,W = im.shape
    h,h = f.shape                #filter
    hh = h//2
    im_pad = np.zeros((H+hh*2,W+hh*2))
    im_pad[hh:H+hh,hh:W+hh] = im
    out = np.zeros((H,W))

    for r in range(W):
        for c in range(H):
            prod = im_pad[r:r+h, c:c+h]*f
            out[r,c] = np.sum(prod,axis=(0,1))
    return out
```

Convolution: H×W×C image with h×h×C filter: One image with *multiple* channels convolved with a filter of equal channels. Output is **H×W**

```
def convHWCpad(im,f):

    H,W,C = im.shape
    h,h,C = f.shape                #filter
    hh = h//2
    im_pad = np.zeros((H+hh*2,W+hh*2,C))
    im_pad[hh:H+hh,hh:W+hh,:] = im
    out = np.zeros((H,W))

    for r in range(W):
        for c in range(H):
            prod = im_pad[r:r+h, c:c+h,:]*f
            out[r,c] = np.sum(prod,axis=(0,1,2))
    return out
```

Convolution: H×W×C image with n×h×h×C filter: One image with multiple channels convolved with *multiple* filters of equal channels. Output **H×W×n**

```
def convHWNpad(im,f):

    H,W,C = im.shape
    n,h,h,C = f.shape                #filter
    hh = h//2
    im_pad = np.zeros((H+hh*2,W+hh*2,C))
    im_pad[hh:H+hh,hh:W+hh,:] = im
    out = np.zeros((H,W,n))

    for j in range(n):
        for r in range(W):
            for c in range(H):
                prod = im_pad[r:r+h, c:c+h,:]*f[j]
                out[r,c,j] = np.sum(prod,axis=(0,1,2))
    return out
```

Convolution: N× H × W ×C image with n×h×h×C filter: *Multiple* images (batch/sample size) with multiple channels convolved with *multiple* filters of equal channels. Output is **N×H×W×n**

```
def convNHWNpad(im,f):

    N,H,W,C = im.shape
    n,h,h,C = f.shape                #filter
    hh = h//2
    im_pad = np.zeros((N,H+hh*2,W+hh*2,C))
    im_pad[:,hh:H+hh,hh:W+hh,:] = im
    out = np.zeros((N,H,W,n))

    for i in range (N):
        for j in range(n):
            for r in range(W):
                for c in range(H):
                    prod = im_pad[i,r:r+h, c:c+h,:]*f[j]
                    out[i,r,c,j] = np.sum(prod,axis=(0,1,2))
    return out
```

Max-Pooling:

Simple max-pooling: H×W image with 2×2 filter, stride=2: One matrix, max is determined in 2×2, stride = 2. Output is W/2 ×H/2.

```
def maxpoolHW(im):
    H,W = im.shape
    out = np.zeros((H//2,W//2))
    for r in range(0,H,2):
        for c in range(0,W,2):
            out[r//2,c//2] = np.max(im[r:r+2,c:c+2])
    return out
```

max-pooling: H×W×C image with 2×2 filter, stride=2: multiple channels, max is determined in 2×2 per channel, stride = 2. Output is W/2×H/2×C.

```
def maxpoolHWC(im):
    H,W,C = im.shape
    out = np.zeros((H//2,W//2,C))
    for r in range(0,H,2):
        for c in range(0,W,2):
            for j in range(?):
                out[r//2,c//2,j] = np.max(im[r:r+2,c:c+2,j])
    return out
```

max-pooling: N× $\mathbf{H} \times \mathbf{W}$ ×C image with 2×2 filter, stride=2: multiple channels, max is determined in 2×2 per channel, stride = 2. Output is **N**×W/2×H/2×C.

```
def maxpoolNHWC(im):
    N,H,W,C = im.shape
    out = np.zeros((N,H//2,W//2,C))
    for i in range(N):
        for r in range(0,H,2):
            for c in range(0,W,2):
                for j in range(C):
                    out[i,r//2,c//2,j] = np.max(im[i,r:r+2,c:c+2,j])
    return out
```

ReLU Activation:

Simple ReLU on a $\mathbf{H} \times \mathbf{W}$ image: blocks negative values of the image. Output is $\mathrm{H} \times \mathrm{W}$.

```
def ReLU(im):
    return (img>0)*img
```

ReLU on a N× $\mathbf{H} \times \mathbf{W} \times \mathbf{C}$ image: blocks negative values of the image. Output is $N \times \mathrm{H} \times \mathrm{W} \times \ \mathrm{C}$.

```
def ReLU(im):
    return (img>0)*img
```

Flatten:

Simple flattening of $\mathbf{H} \times \mathbf{W}$ image: Output is $1 \times \mathrm{HW}$.

```
def flatten(im):
    H,W = im.shape
    return img.reshape(H*W)
```

Flattening of $\mathbf{H} \times \mathbf{W} \times \mathbf{C}$ image: Output is $1 \times \mathrm{HWC}$.

```
def flatten(im):
    H,W,C = im.shape
    return img.reshape(H*W*C)
```

Simple flattening of $\mathbf{N} \times \mathbf{H} \times \mathbf{W} \times \mathbf{C}$ image: Output is $\mathrm{N} \times \mathrm{HWC}$.

```
def flatten(im):
    N,H,W,C = im.shape
    return img.reshape(N,H*W*C)
```

Part 2: Backward Propagation

Backprop Theory

Forward Layers: $\text{Input} \to \underbrace{\text{Convolution} \to \text{ReLU} \to \text{Max-pooling}}_{\text{repeats}} \to \text{Flatten} \to \underbrace{\text{Sigmoid}}_{\text{repeats}} \to \text{Softmax}$

$\mathcal{L}$:	loss function
l:	layer number (L = last layer)
W^l:	Weight from layer l
a^l:	activation function in layer l
Z^l:	Input to activation functions
$\frac{\partial a^l}{\partial Z^l} = g^l$:	derivative of the activation function

$$\frac{\partial \mathcal{L}}{\partial W^{L-2}} = \frac{\partial \mathcal{L}}{\partial a^L}\left(\frac{\partial a^L}{\partial Z^L}\right)\frac{\partial Z^L}{\partial a^{L-1}}\left(\frac{\partial a^{L-1}}{\partial Z^{L-1}}\right)\frac{\partial Z^{L-1}}{\partial a^{L-2}}\left(\frac{\partial a^{L-2}}{\partial Z^{L-2}}\right)\frac{\partial Z^{L-2}}{\partial W^{L-2}}$$

$$= \frac{\partial \mathcal{L}}{\partial a^L} \quad (g^L) \quad W^{L-1} \quad (g^{L-1}) \quad W^{L-2} \quad (g^{L-2}) \quad a^{L-2}$$

$$\Rightarrow \quad \frac{\partial \mathcal{L}}{\partial a^l} = \frac{\partial \mathcal{L}}{\partial a^{l+1}}(g^l)W^l, \qquad \frac{\partial \mathcal{L}}{\partial W^l} = \frac{\partial \mathcal{L}}{\partial a^l}\ a^L, \qquad \frac{\partial \mathcal{L}}{\partial b^l} = \frac{\partial \mathcal{L}}{\partial a^l}$$

Summary:
For each layer calculate the gradient of activation function g^{l+1}, then the partial derivative of the loss wrt the activation function, $\frac{\partial \mathcal{L}}{\partial a^l}$, then partial derivative of loss wrt the weights, $\frac{\partial \mathcal{L}}{\partial W^l}$.

Backprop Layer:

```
def backproplayer(a_l, da_lp1, g_l, W_l):
    da_l = np.dot(da_lp1 , W_l.T) * g_l
    dW_l = np.dot(a_l.T , da_l)
    db_l = np.sum(da_l, axis=0, keepdims=True)
    return da_l, dW_l, db_l
```

Backward Softmax (L):

$$g_{\text{softmax}} = \frac{\partial a_{\text{softmax}}}{\partial Z} = 1$$

$$\frac{\partial \mathcal{L}}{\partial a^L} = \frac{\partial \mathcal{L}}{\partial a_{\text{softmax}}} = \delta \left(\frac{\partial a_{\text{softmax}}}{\partial Z}\right) = \delta$$

$$\frac{\partial \mathcal{L}}{\partial W^{L-1}} = \delta \left(a^{L-1}\right)$$

Backprop Layer:

```
da_Lm1, dW_L, db_L = backproplayer(softmax(a_Lm1,W_L,b_L), delta, 1, W_L):
```

Backward Sigmoid ($L-1$):

$$g^{L-1} = \frac{\partial a_{\text{sigmoid}}^{L-1}}{\partial Z^{L-1}} = a_{\text{sigmoid}}^{L-1}\left(1 - a_{\text{sigmoid}}^{L-1}\right)$$

$$\frac{\partial \mathcal{L}}{\partial a^{L-1}} = \frac{\partial \mathcal{L}}{\partial a^L}\left(g^{L-1}\right)W^L = \delta\left(g^{L-1}\right)$$

$$\frac{\partial \mathcal{L}}{\partial W^{L-1}} = \frac{\partial \mathcal{L}}{\partial a^{L-1}}\ a_{\text{sigmoid}}^{L-2}$$

Backward Sigmoid (l):

$$g^l = \frac{\partial a_{\text{sigmoid}}^{l}}{\partial Z^{l}} = a_{\text{sigmoid}}^{l}\left(1 - a_{\text{sigmoid}}^{l}\right)$$

$$\frac{\partial \mathcal{L}}{\partial a^{l}} = \frac{\partial \mathcal{L}}{\partial a^{l+1}}\left(g^{l}\right)W^{l+1}$$

$$\frac{\partial \mathcal{L}}{\partial W^{l}} = \frac{\partial \mathcal{L}}{\partial a^{l}}\, a^{l-1}$$

Backward Flatten (l):

Flattening is a re-organization of the feature maps from the previous layer (max-pooling layer). It doesn't change the values in the previous layer, i.e. $\frac{\partial a_{\text{flatten}}}{\partial Z} = 1$. There are no weights connecting the previous layer to the flattened layer.

$$g^l = 1$$

$$\frac{\partial \mathcal{L}}{\partial a^{l}} = \frac{\partial \mathcal{L}}{\partial a^{l+1}} W^l$$

Simple unflattening of N$\times$ HWC image: Output is $\mathrm{N} \times \mathrm{H} \times \mathrm{W} \times \mathrm{C}$.

```
def unflatten(a,N,H,W,C):
    return a.reshape(N,H,W,C)
```

Backward Max-pool (l):

Max-pooling selects the max in a 2x2. So for the values that match the max, $g = 1$, otherwise it is 0. There are no weights connecting the previous layer (convolution layer) to the max-pooling.

1	2	3	4
8	7	5	6
9	11	12	10
10	7	13	9

Z^l

8	6
11	13

a^l

0	0	0	0
1	0	0	1
0	1	0	0
0	0	1	0

g^l

$$g^l = \begin{cases} 1 & Z^l = a^l \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial \mathcal{L}}{\partial a^l} = \frac{\partial \mathcal{L}}{\partial a^{l+1}} W^{l+1}$$

Simple Gradient of Max-pooling: Assumes input is HxW and max pooling was 2×2, stride = 2

```
def g_maxpool(z_l, a_mp):            #a_mp is the output of maxpooling
    g = np.repeat(a_mp,2, axis=1)    #duplicate the columns
    g = np.repeat(g,2, axis=0)       #duplicate the rows
    g = (g==z_l)*1.                  #z_l is the input to max-pooling
    return g
```

Backward Convolution:

$$\frac{dL}{da^l} = convolution\left(W, \frac{dL}{da^{l+1}}\right)$$

$$\frac{dL}{dW^l} = convolution\left(a^{l-1}, \frac{dL}{da^l}\right)$$

See https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/

Note that convolution used in convolutional NNet is not a mathematical convolution. In mathematical convolution you flip one of the inputs about its axis (in 2D you flip horizontally, then vertically, hence it's a rotation by 180 degrees).