AI Project

Robin MENEUST

August 2023

Table of contents

Introduction

1 Softmax derivative

Softmax :

$$s_{z_i}(z_1, z_2, ...z_n) = \frac{e^{z_i}}{\sum\limits_{j=1}^{n} e^{z_j}} \tag{1}$$

Derivative :

$$
\begin{aligned}
\frac{\partial s_{z_i}}{\partial z_i} &= \frac{\partial \left(\frac{e^{z_i}}{\sum\limits_{j=1}^{n} e^{z_j}} \right)}{\partial z_i} \\
&= \frac{\partial \left(\frac{e^{z_i}}{k + e^{z_i}} \right)}{\partial z_i}, \quad \text{where } k = \sum_{j=1,\ j \neq i}^{n} e^{z_j} \\
&= \frac{\partial \left(1 - \frac{k}{k + e^{z_i}} \right)}{\partial z_i} \\
&= \frac{k e^{z_i}}{(k + e^{z_i})^2} \\
&= \left(\sum_{j=1,\ j \neq i}^{n} e^{z_j} \right) \frac{e^{z_i}}{\left(\left(\sum\limits_{j=1,\ j \neq i}^{n} e^{z_j} \right) + e^{z_i} \right)^2} \\
&= \left(\sum_{j=1,\ j \neq i}^{n} e^{z_j} \right) \frac{e^{z_i}}{\left(\sum\limits_{j=1}^{n} e^{z_j} \right)^2} \\
&= \left(\left(\sum_{j=1}^{n} e^{z_j} \right) - e^{z_i} \right) \frac{e^{z_i}}{\left(\sum\limits_{j=1}^{n} e^{z_j} \right)^2} \\
&= e^{z_i} \left(\frac{1}{\sum\limits_{j=1}^{n} e^{z_j}} - \frac{s_{z_i}^2}{e^{z_i}} \right) \\
&= s_{z_i} - s_{z_i}^2 \\
&= s_{z_i}(1 - s_{z_i})
\end{aligned} \tag{2}
$$

And:

$$
\begin{aligned}
\frac{\partial s_{z_i}}{\partial z_{k\neq i}} &= \frac{\partial\left(\frac{e^{z_i}}{\sum\limits_{j=1}^{n} e^{z_j}}\right)}{\partial z_{k\neq i}} \\
&= e^{z_i}\frac{\partial\left(\frac{1}{c+e^{z_k}}\right)}{\partial z_{k\neq i}}, \quad \text{where } c = \sum_{j=1,\ j\neq k}^{n} e^{z_j} \\
&= -e^{z_i}\frac{e^{z_k}}{(c+e^{z_k})^2} \\
&= -\frac{e^{z_i}e^{z_k}}{\left(\left(\sum\limits_{j=1,\ j\neq k}^{n} e^{z_j}\right)+e^{z_k}\right)^2} \\
&= -\frac{e^{z_i}e^{z_k}}{\left(\sum\limits_{j=1}^{n} e^{z_j}\right)^2} \\
&- s_{z_i}s_{z_k}
\end{aligned} \tag{3}
$$

So we have :

$$
\begin{aligned}
\frac{\partial s_{z_i}}{\partial z_k} &= \begin{cases} s_{z_i}(1-s_{z_i}) & \text{if } i=k \\ -s_{z_i}s_{z_k} & \text{else} \end{cases} \\
&= s_{z_i}\left(\delta_{ik}-s_{z_k}\right)
\end{aligned} \tag{4}
$$

2 Definitions and standard functions derivatives

Let :

1. C be the total cost function
2. y_i be the output (prediction) i
3. $\hat{y}_i$ be the expected output i
4. C_i be the cost for output i (e.g. MSE: $\frac{1}{2}\left(\hat{y}_i-y_i\right)^2$). In this document we use MSE, but for MNIST CCE (categorical cross-entropy) might be better (and we will use it in a future update)

5. $w_{i,j}^{(l)}$ be the weight of the neuron j of the layer $l-1$ for the neuron i of the layer l
6. $z_i^{(l)}$ be the weighted sum for the neuron i of the layer l (activation function input)
7. $a_i^{(l)} = g^{(l)}(z_i^{(l)})$ be the output of the neuron i of the layer l (activation function output)
8. $b_i^{(l)}$ be the bias of the neuron i of the layer l
9. L be the number of layers and the index of the output layer (layers index goes from 1 to L)
10. n_l be the number of neurons in the layer l
11. The derivative of Sigmoid σ is $\sigma(1-\sigma)$
12. The derivative of Softmax $s_{z_i}(z_i)$ is $s_{z_i}(z_i)\,(1-s_{z_i}(z_i))$

Back-propagation

1 Output layer L

Here we consider that the activation function of the layer L is Softmax s.

$$\frac{\partial C}{\partial w_{i,j}^{(L)}} = \sum_{p=1}^{n_L} \frac{\partial C}{\partial a_p^{(L)}} \frac{\partial a_p^{(L)}}{\partial z_i^{(L)}} \frac{\partial z_i^{(L)}}{\partial w_{i,j}^{(L)}}$$

Where

$$\frac{\partial C}{\partial a_p^{(L)}} = \frac{\partial \frac{1}{n_l} \sum_{k=1}^{n_L} C_k}{\partial a_p^{(L)}} = \frac{1}{n_L} \sum_{k=1}^{n_l} \frac{\partial C_k}{\partial a_p^{(L)}} = \frac{1}{n_L} \frac{\partial C_p}{\partial a_p^{(L)}}$$

$$\frac{\partial a_p^{(L)}}{\partial z_i^{(L)}} = \frac{\partial s_{z_p^{(L)}}}{\partial z_i^{(L)}}$$

$$\frac{\partial z_i^{(L)}}{\partial w_{i,j}^{(L)}} = \frac{\partial \left(\sum_{k=1}^{n_{L-1}} \left(w_{i,k}^{(L)} a_i^{(L-1)} \right) + b_i^{(L)} \right)}{\partial w_{i,j}^{(L)}} = \sum_{k=1}^{n_{L-1}} \left(\frac{\partial w_{i,k}^{(L)} a_i^{(L-1)}}{\partial w_{i,j}^{(L)}} \right) + \frac{\partial b_i^{(L)}}{\partial w_{i,j}^{(L)}} = a_j^{(L-1)}$$

For the bias it's almost the same equation:

$$\frac{\partial C}{\partial b_i^{(L)}} = \sum_{p=1}^{n_L} \frac{\partial C}{\partial a_p^{(L)}} \frac{\partial a_p^{(L)}}{\partial z_i^{(L)}} \frac{\partial z_i^{(L)}}{\partial b_i^{(L)}}$$

Where

$$\frac{\partial z_i^{(L)}}{\partial b_i^{(L)}} = \frac{\partial \left(\left(\sum_{k=1}^{n_{L-1}} w_{i,k}^{(L)} a_i^{(L-1)} \right) + b_i^{(L)} \right)}{\partial b_i^{(L)}} = \sum_{k=1}^{n_{L-1}} \left(\frac{\partial w_{i,k}^{(L)} a_i^{(L-1)}}{\partial b_i^{(L)}} \right) + \frac{\partial b_i^{(L)}}{\partial b_i^{(L)}} = 1$$

2 Layer L-1

Here we consider that the activation function of the layer $L-1$ and the other ones except L is sigmoid σ.

$$\frac{\partial C}{\partial w_{i,j}^{(L-1)}} = \frac{\partial C}{\partial a_i^{(L-1)}} \frac{\partial a_i^{(L-1)}}{\partial z_i^{(L-1)}} \frac{\partial z_i^{(L-1)}}{\partial w_{i,j}^{(L-1)}}$$

Where

$$\frac{\partial a_i^{(L-1)}}{\partial z_i^{(L-1)}} = \frac{\partial \sigma}{\partial z_i^{(L-1)}} = g'^{(L-1)}(z_i^{(L-1)})$$

$$\frac{\partial z_i^{(L-1)}}{\partial w_{i,j}^{(L-1)}} = \frac{\partial \left(\sum_{k=1}^{n_{L-2}} \left(w_{i,k}^{(L-1)} a_i^{(L-2)} \right) + b_i^{(L-1)} \right)}{\partial w_{i,j}^{(L-1)}} = \sum_{k=1}^{n_{L-2}} \left(\frac{\partial w_{i,k}^{(L-1)} a_i^{(L-2)}}{\partial w_{i,j}^{(L-1)}} \right) + \frac{\partial b_i^{(L-1)}}{\partial w_{i,j}^{(L-1)}} = a_j^{(L-2)}$$

$$\frac{\partial C}{\partial a_i^{(L-1)}} = \sum_{k=1}^{n_L} \sum_{p=1}^{n_L} \frac{\partial C}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial z_p^{(L)}} \frac{\partial z_p^{(L)}}{\partial a_i^{(L-1)}}$$

We already calculated the 2 first derivatives in the previous subsection, and for the last one:

$$\frac{\partial z_k^{(L)}}{\partial a_i^{(L-1)}} = \frac{\partial \left(\sum_{p=1}^{n_{L-1}} \left(w_{k,p}^{(L)} a_k^{(L-1)} \right) + b_k^{(L)} \right)}{\partial a_i^{(L-1)}} = \sum_{p=1}^{n_{L-1}} \left(\frac{\partial w_{k,p}^{(L)} a_k^{(L-1)}}{\partial a_i^{(L-1)}} \right) + \frac{\partial b_k^{(L)}}{\partial a_i^{(L-1)}} = w_{k,i}^{(L)}$$

Also, we should notice here that for many activation functions (Sigmoid, ReLU...) $\frac{\partial a_k^{(l)}}{\partial a_p^{(l)}}$ will be equal to 0 if $p \neq k$, but not for functions such as Softmax.

3 Layer $l < L$

Here we have the general case, where the activation function can be softmax, sigmoid...

$$\frac{\partial C}{\partial w_{i,j}^{(l)}} = \left(\sum_{k=1}^{n_l} \frac{\partial C}{\partial a_k^{(l)}} \frac{\partial a_k^{(l)}}{\partial z_i^{(l)}} \right) \frac{\partial z_i^{(l)}}{\partial w_{i,j}^{(l)}} = \left(\sum_{k=1}^{n_l} \frac{\partial C}{\partial a_k^{(l)}} \frac{\partial a_k^{(l)}}{\partial z_i^{(l)}} \right) a_j^{(l-1)}$$

Where if $l < L$:

$$\frac{\partial C}{\partial a_i^{(l)}} = \sum_{k=1}^{n_{l+1}} \sum_{p=1}^{n_{l+1}} \frac{\partial C}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial z_p^{(l+1)}} \frac{\partial z_p^{(l+1)}}{\partial a_i^{(l)}} = \sum_{k=1}^{n_{l+1}} \sum_{p=1}^{n_{l+1}} \frac{\partial C}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial z_p^{(l+1)}} w_{p,i}^{(l+1)}$$

Otherwise if $l = L$:

$$\frac{\partial C}{\partial a_i^{(L)}} = \frac{1}{n_L} \frac{\partial C_i}{\partial a_i^{(L)}}$$

4 Algorithm

Here we will calculate in each layer the following derivative (to propagate to the next layer):

$$\begin{aligned}
\frac{\partial C}{\partial z_i^{(l)}} &= \sum_{k=1}^{n_{l+1}} \sum_{p=1}^{n_{l+1}} \sum_{j=1}^{n_l} \frac{\partial C}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial z_p^{(l+1)}} \frac{\partial z_p^{(l+1)}}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_i^{(l)}} \\
&= \sum_{k=1}^{n_{l+1}} \sum_{p=1}^{n_{l+1}} \sum_{j=1}^{n_l} \frac{\partial C}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial z_p^{(l+1)}} w_{p,j}^{(l+1)} \frac{\partial a_j^{(l)}}{\partial z_i^{(l)}} \\
&= \sum_{k=1}^{n_{l+1}} \sum_{p=1}^{n_{l+1}} \left(\frac{\partial C}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial z_p^{(l+1)}} \sum_{j=1}^{n_l} \left(w_{p,j}^{(l+1)} \frac{\partial a_j^{(l)}}{\partial z_i^{(l)}} \right) \right) \\
&= \sum_{p=1}^{n_l} \sum_{j=1}^{n_l} \left(prevDerivatives \times w_{p,j}^{(l+1)} \frac{\partial a_j^{(l)}}{\partial z_i^{(l)}} \right)
\end{aligned}$$

Note: This algorithm is not optimized, we should use the fact that we are using matrices.

Step 1: feed-forward and store values

```
# input: input vector, fed to this network
# getWeightedSum(layerIndex, prevLayerValues) (z_i)
# activationFunction(layerIndex, input) (a_i)

outputs = []
weightedSums = []

outputs[0] = getWeightedSum(0, input)
```

```
weightedSums[0] = activationFunction(i, outputs[0])

for(i in range(len(layers)):
    weightedSums[i] = getWeightedSum(i, outputs[i-1])
    outputs[i] = activationFunction(i, weightedSums[i])
```

Step 2: Back-propagation

```
# for all p,i dC/da_p * da_p/dz_i

tempCostDerivatives = getCostDerivatives(outputs[len(layers)
    -1], expectedOutput) # vector of dC/da_j for all j

activationDerivatives = getActivationDerivatives(len(layers)-1,
     weightedSums[getNbLayers()-1]) # matrix of da_j/dz_i for
    all j and all i

for(i in range(getLayerSize(len(layers) - 1)):
    if(isActivationFunctionMultidimensional(len(layers) - 1)):
        currentCostDerivatives[i] = 0
        for(j in range(getLayerSize(len(layers) - 1)):
            currentCostDerivatives[i] += tempCostDerivatives[j]
     * (1.0f/getLayerSize(len(layers) - 1) *
    activationDerivatives[j][i]
    else:
        currentCostDerivatives[i] = tempCostDerivatives[i] *
    (1.0f/getLayerSize(len(layers) - 1) * activationDerivatives
    [0][i]


for(l in range(len(layers)-1,-1,-1):
    # Next cost derivatives computation
    if l>0:
        nextCostDerivatives = []
        for(i in range(getLayerSize(l-1))):
            nextCostDerivatives[i] = 0 # sum for all j,p,j dC/
    da_k * da_k/dz_p * dz_p/da_j * da_j/dz_i

            for(j in range(getLayerSize(l))):
                activationDerivatives =
    getActivationDerivatives(l-1, weightedSums[l-1]) # matrix
    of da_j/dz_i for all j and all i

                if(isActivationFunctionMultidimensional(l-1)):
                    for(p in range(getLayerSize(l))):
                        nextCostDerivatives[i] +=
    currentCostDerivatives[p] * getWeight(l,p,j) *
    activationDerivatives[j][i]
                else:
                    nextCostDerivatives[i] +=
    currentCostDerivatives[j] * getWeight(l,p,i) * activationDerivatives[0][i]


    # Adjust the weights and biases of the current layer

    prevLayerOutput = outputs[l-1] if l>0 else input

    for(i in range(getLayerSize(l))):
        for(j in range(len(prevLayerOutput)):
            setWeight(l,i,j) -= lr * currentCostDerivatives[i] * prevLayerOutput[j] # lr = learning rate and we have dC/dz_k * dz_i/dw_i,j
            setBias(l,i) -= lr * currentCostDerivatives[i] # dC/dz_k

    currentCostDerivatives = nextCostDerivatives
```

5 Improvement: mini-batch

Instead of training only once all the examples one by one, we will randomly create batches of examples. So for all epoch and for all batch in the epoch, we feed-forward all the examples of the batch and we calculate the error for each. Then we modify the previous algorithm to propagate all the batch derivatives vector and when we adjust a parameter we use the mean derivative of all the batch derivatives. We will now send several examples (of the same batch) to our function for thee back-propagation (batch with several inputs instead of one input).
Here we will choose to only train on complete batches, so if there are left-overs after grouping items in batch of the same size, we won't use them.

Step 1: feed-forward and store values

```
# batch: list of inputs vector, fed to this network
# getWeightedSum(layerIndex, prevLayerValues, batchSize) (list of vectors z_i of the layer layerIndex of all items in the batch: it's a matrix)
# activationFunction(layerIndex, input) (list of vectors a_i of the layer layerIndex of all items in the batch: it's a matrix)

outputs = []
weightedSums = []

outputs[0] = getWeightedSum(0, batch, len(batch))
weightedSums[0] = activationFunction(i, outputs[0], len(batch))

for(i in range(len(layers)):
    weightedSums[i] = getWeightedSum(i, outputs[i-1], len(batch))
    outputs[i] = activationFunction(i, weightedSums[i], len(batch))
```

Step 2: Back-propagation

```
# dC/da_k * da_k/dz_k
tempCostDerivatives = getCostDerivatives(outputs[len(layers)
    -1], expectedOutput, len(batch)) # Matrix of dC/da_j for
    all batch item and for all j
activationDerivatives = getActivationDerivatives(len(layers)-1,
     weightedSums[getNbLayers()-1], len(batch)) # Tensor of
    order 3 of da_j/dz_i for all batch item and for all j,i

for(b in range(len(batch))):
    for(i in range(getLayerSize(len(layers) - 1)):
        if(isActivationFunctionMultidimensional(len(layers) -
    1)):
            currentCostDerivatives[b][i] = 0
            for(j in range(getLayerSize(len(layers) - 1)):
                currentCostDerivatives[i] +=
    tempCostDerivatives[j] * (1.0f/getLayerSize(len(layers) -
    1) * activationDerivatives[b][j][i]
        else:
            currentCostDerivatives[b][i] = tempCostDerivatives[
    i] * (1.0f/getLayerSize(len(layers) - 1) *
    activationDerivatives[b][0][i]

for(l in range(len(layers)-1,-1,-1):
    # Next cost derivatives computation
    if l>0:
        nextCostDerivatives = []
        for(b in range(len(batch)):
            nextCostDerivatives[b] = []
            for(i in range(getLayerSize(l-1))):
                nextCostDerivatives[b][i] = 0 # sum for all j,p
    ,j dC/da_k * da_k/dz_p * dz_p/da_j * da_j/dz_i

                for(j in range(getLayerSize(l))):
                    activationDerivatives =
    getActivationDerivatives(l-1, weightedSums[l-1]) # matrix
    of da_j/dz_i for all j and all i

                    if(isActivationFunctionMultidimensional(l
    -1)):
                        for(p in range(getLayerSize(l))):
                            nextCostDerivatives[b][i] +=
    currentCostDerivatives[b][p] * getWeight(l,p,j) *
```

```
activationDerivatives[j][i]
                    else:
                        nextCostDerivatives[b][i] += currentCostDerivatives[b][j] * getWeight(l,p,i) * activationDerivatives[0][i]


    # Adjust the weights and biases of the current layer

    prevLayerOutput = outputs[l-1] if l>0 else batch

    for(i in range(getLayerSize(l))):
        for(j in range(len(prevLayerOutput)):
            # Mean of the derivatives
            deltaWeight = 0
            deltaBias = 0
            for(b in range(len(batch))):
                deltaWeight += currentCostDerivatives[b][i] * prevLayerOutput[b][j]
                deltaBias += currentCostDerivatives[b][i]
            deltaWeight /= len(batch)
            deltaBias /= len(batch)

            # Adjust the parameters
            setWeight(l,i,j) -= lr * deltaWeight # lr = learning rate and we have dC/da_k * da_k/dz_k * dz_k/dw_i,j
            setBias(l,i) -= lr * deltaBias # dC/da_k * da_k/dz_k

    currentCostDerivatives = nextCostDerivatives
```

Conv2D, Max-pooling and flatten layers

1 Conv2D Layer

Here we will look at the following example :

1. Input: 3x3 matrix $w_I = 3$
2. Filter: 2x2 matrix $w_F = 2$
3. Number of output matrices: 3
4. Padding p (Number of lines and columns of zeros added). We want in this example to get an output with the same dimensions as the input. Additionally, if the input matrix length and width are equal we need $w_I + p - w_F + 1$ iterations to read a full row, this number will be output matrix width. Hence we have: $w_I + p - w_F + 1 = w_I \Leftrightarrow p = w_F - 1$. Here $p = 1$
5. Activation function = ReLU : We use it to keep positive values for the next layers

Feed-forward

For each filter Fn :

$$Fn = \begin{pmatrix} Fn_{1,1} & Fn_{1,2} \\ Fn_{2,1} & Fn_{2,2} \end{pmatrix} \tag{5}$$

$$input = I = \begin{pmatrix} I_{1,1} & I_{1,2} & I_{1,3} \\ I_{2,1} & I_{2,2} & I_{2,3} \\ I_{3,1} & I_{3,2} & I_{3,3} \end{pmatrix} \tag{6}$$

We add padding, if p is odd then we add more one more row and column of zero on the last row and column, otherwise there is the same padding on every sides. In this example $p = 1$ is odd and $p/2 = 0$, thus we don't add zeros on the upper and left sides (because $p/2 = 0$), but we add one row and one columns on the lower and right sides.

$$input = I_{pad} = \begin{pmatrix} I_{1,1} & I_{1,2} & I_{1,3} & 0 \\ I_{2,1} & I_{2,2} & I_{2,3} & 0 \\ I_{3,1} & I_{3,2} & I_{3,3} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{7}$$

$$output = On = Conv\left(\begin{pmatrix} I_{1,1} & I_{1,2} & I_{1,3} & 0 \\ \boxed{\begin{matrix} I_{2,1} & I_{2,2} \\ I_{3,1} & I_{3,2} \end{matrix}} & \begin{matrix} I_{2,3} \\ I_{3,3} \end{matrix} & \begin{matrix} 0 \\ 0 \end{matrix} \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} \boxed{\begin{matrix} Fn_{1,1} & Fn_{1,2} \\ Fn_{2,1} & Fn_{2,2} \end{matrix}} \end{pmatrix}\right) \tag{8}$$

$$= \begin{pmatrix} On_{1,1} & On_{1,2} & On_{1,3} \\ \boxed{On_{2,1}} & On_{2,2} & On_{2,3} \\ On_{3,1} & On_{3,2} & On_{3,3} \end{pmatrix}$$

Here, for instance, $On_{2,1} = ReLU(I_{2,1} \times Fn_{1,1} + I_{2,2} \times Fn_{1,2} + I_{3,1} \times Fn_{2,1} + I_{3,2} \times Fn_{2,2})$

Back-propagation

We need to calculate two derivatives for each input matrices :

1. One to continue the back-propagation in the previous layers
2. Another one to adjust the filters values (similarly to the weights of the Dense layers)

Let's begin with the first one:

$$\begin{aligned}
&\forall a,b \in [\![1, w_O]\!]^2, \forall i,j \in [\![a, a+w_F]\!] \times [\![b, b+w_F]\!],\\
&\frac{\partial z_{a,b}}{\partial I_{i,j}} = \frac{\partial \sum\limits_{k,l \in [\![0,w_F-1]\!] \times [\![0,w_F-1]\!]} h(a+k,b+l) \times F_{k+1,l+1}}{\partial I_{i,j}}\\
&\text{where } h(k,l) = \begin{cases} I_{k,l} & \text{if } k,l \in [\![1+\frac{p}{2}, w_I]\!]^2 \\ 0 & \text{else} \end{cases}
\end{aligned} \tag{9}$$

So we get

$$\frac{\partial z_{a,b}}{\partial I_{i,j}} = \begin{cases} F_{i-a+1,j-b+1} & \text{if } (i-a),(j-b) \in [\![1, w_O]\!]^2 \\ 0 & \text{else} \end{cases} \tag{10}$$

And

$$\frac{\partial O_{a,b}}{\partial z_{a,b}} = \frac{\partial ReLU}{\partial z_{a,b}} = \begin{cases} 0 & \text{if } z_{a,b} < 0 \\ 1 & \text{if } z_{a,b} > 0 \end{cases} \tag{11}$$

Therefore

$$\begin{aligned}
\frac{\partial O_{a,b}}{\partial I_{i,j}} &= \frac{\partial O_{a,b}}{\partial z_{a,b}} \frac{\partial z_{a,b}}{\partial I_{i,j}} = \begin{cases} 0 & \text{if } z_{a,b} < 0 \\ 1 & \text{if } z_{a,b} > 0 \end{cases} \times \begin{cases} F_{i-a+1,j-b+1} & \text{if } (i-a),(j-b) \in [\![1, w_O]\!]^2 \\ 0 & \text{else} \end{cases}\\
&= \begin{cases} F_{i-a+1,j-b+1} & \text{if } (i-a),(j-b) \in [\![1, w_O]\!]^2 \text{ and } z_{a,b} > 0 \\ 0 & \text{else} \end{cases}
\end{aligned} \tag{12}$$

Then we calculate the second one:

$$\begin{aligned}
&\forall a,b \in [\![1, w_O]\!]^2, \forall i,j \in [\![1, w_F]\!]^2,\\
&\frac{\partial z_{a,b}}{\partial F_{i,j}} = \frac{\partial \sum\limits_{k,l \in [\![0,w_F-1]\!] \times [\![0,w_F-1]\!]} h(a+k,b+l) \times F_{k+1,l+1}}{\partial F_{i,j}}\\
&\frac{\partial z_{a,b}}{\partial F_{i,j}} = h(a+i-1, b+j-1)\\
&\frac{\partial z_{a,b}}{\partial F_{i,j}} = \begin{cases} I_{a+i-1,b+j-1} & \text{if } (a+i-1),(b+j-1) \in [\![1+\frac{p}{2}, w_I]\!]^2 \\ 0 & \text{else} \end{cases}
\end{aligned} \tag{13}$$

Therefore

$$\begin{aligned}\frac{\partial O_{a,b}}{\partial F_{i,j}} &= \frac{\partial O_{a,b}}{\partial z_{a,b}}\frac{\partial z_{a,b}}{\partial F_{i,j}} = \begin{cases}0 & \text{if } z_{a,b} < 0\\ 1 & \text{if } z_{a,b} > 0\end{cases} \times \begin{cases}I_{a+i-1,b+j-1} & \text{if } (a+i-1),(b+j-1) \in [\![1+\frac{p}{2}, w_I]\!]^2\\ 0 & \text{else}\end{cases}\\ &= \begin{cases}I_{a+i-1,b+j-1} & \text{if } (a+i-1),(b+j-1) \in [\![1+\frac{p}{2}, w_I]\!]^2 \text{ and } z_{a,b} > 0\\ 0 & \text{else}\end{cases}\end{aligned} \tag{14}$$

2 Max-pooling Layer

Feed-forward

Here we will look at the following example :

1. Input: 3x3 matrix $w_I = 3$
2. Filter: 2x2 matrix $w_F = 2$ because we want to divide the resolution by 2 here
3. Output: 2x2 matrix $w_O = 2$
4. Stride: $s = w_F = 2$ because we don't want to re-read values here. (s is the width of the step between 2 iterations when we move the filter (e.g. here we move the filter matrix of 2 cells to the right for each iteration, and of 2 cells to the bottom when we reached the last cell of the row)).
5. Padding: p (Number of lines and columns of zeros added). Here, we want to satisfy the previous conditions while minimizing the padding, so we get: $w_I \equiv p \mod w_F$ (i.e. in *C++*: p = w_I % w_F). Here we get $p = 1$.
6. We don't need an activation function here, since the max of a positive value is always positive, there are no weights (that could be negative for Conv2D filters or Dense layers) involved here.

If we have multiple input matrices, we just apply it to each one, and we get as many output matrices as we got in the input.

We add padding the same way as before.

$$input = I_{pad} = \begin{pmatrix} I_{1,1} & I_{1,2} & I_{1,3} & 0 \\ I_{2,1} & I_{2,2} & I_{2,3} & 0 \\ I_{3,1} & I_{3,2} & I_{3,3} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{15}$$

$$MaxPool_{2x2}\left(\begin{pmatrix} \begin{array}{|cc|cc} \hline I_{1,1} & I_{1,2} & I_{1,3} & 0 \\ I_{2,1} & I_{2,2} & I_{2,3} & 0 \\ \hline I_{3,1} & I_{3,2} & I_{3,3} & 0 \\ 0 & 0 & 0 & 0 \end{array} \end{pmatrix}\right) = \begin{pmatrix} \boxed{O_{1,1}} & O_{1,2} \\ O_{2,1} & \boxed{O_{2,2}} \end{pmatrix} \tag{16}$$

For example, $O_{1,1} = \max(I_{1,1}, I_{1,2}, I_{2,1}, I_{2,2})$

Back-propagation

Here, there is no parameter to adjust, so we just compute the derivative used to continue the back-propagation for each input matrices. It's similar to the section with the Dense layers and it will be used in the for loop of the back-propagation. We have $O^{(l)}_{a,b} = a^{(l)}_p = a^{(l)}_k$, it's just that we have 2 dimensions instead of one, but if we consider it flatten then it's the same.

$$\forall a, b \in [\![1, w_O]\!]^2, \forall i, j \in [\![(a-1) \times s + 1, a \times s]\!] \times [\![(b-1) \times s + 1, b \times s]\!],\ \frac{\partial O_{a,b}}{\partial I_{i,j}}$$
$$= \begin{cases} 1 & \text{if } i, j = \underset{k,l \in [\![(a-1)\times s+1, a\times s]\!] \times [\![(b-1)\times s+1, b\times s]\!]}{\arg\max} (I_{k,l}) \\ 0 & \text{else} \end{cases} \tag{17}$$

Thus, we have defined above $\frac{\partial z^{(l)}_k}{\partial a^{(l-1)}_i}$ and we have $\frac{\partial a^{(l)}_k}{\partial z^{(l)}_k} = 1$

3 Flatten Layer

This layer take one or multiple input matrices and flatten and concatenate them to get only one vector (dim = 1)

Feed-forward

$$flatten(I) = flatten\left(\begin{pmatrix} I1_{1,1} & I1_{1,2} & I1_{1,3} \\ I1_{2,1} & I1_{2,2} & I1_{2,3} \\ I1_{3,1} & I1_{3,2} & I1_{3,3} \end{pmatrix}, \begin{pmatrix} I2_{1,1} & I2_{1,2} & I2_{1,3} \\ I2_{2,1} & I2_{2,2} & I2_{2,3} \\ I2_{3,1} & I2_{3,2} & I2_{3,3} \end{pmatrix}\right) = \tag{18}$$

$(I1_{1,1},\ I1_{1,2},\ I1_{1,3},\ I1_{2,1},\ I1_{2,2},\ I1_{2,3},\ I1_{3,1},\ I1_{3,2},\ I1_{3,3},\ I2_{1,1},\ I2_{1,2},\ I2_{1,3},$
$I2_{2,1},\ I2_{2,2},\ I2_{2,3},\ I2_{3,1},\ I2_{3,2},\ I2_{3,3})$

Back-propagation

There is no derivative involved, we simply revert the flatten transformation, so we need to store the dimensions before the transformation:

$(I1_{1,1}, I1_{1,2}, I1_{1,3}, I1_{2,1}, I1_{2,2}, I1_{2,3}, I1_{3,1}, I1_{3,2}, I1_{3,3}, I2_{1,1}, I2_{1,2}, I2_{1,3},$
$I2_{2,1}, I2_{2,2}, I2_{2,3}, I2_{3,1}, I2_{3,2}, I2_{3,3})$

$$\mapsto \left(\begin{pmatrix} I1_{1,1} & I1_{1,2} & I1_{1,3} \\ I1_{2,1} & I1_{2,2} & I1_{2,3} \\ I1_{3,1} & I1_{3,2} & I1_{3,3} \end{pmatrix}, \begin{pmatrix} I2_{1,1} & I2_{1,2} & I2_{1,3} \\ I2_{2,1} & I2_{2,2} & I2_{2,3} \\ I2_{3,1} & I2_{3,2} & I2_{3,3} \end{pmatrix}\right) \tag{19}$$