Implementing Convolutional Layer

Victor BUSA victor.busa@gmail.com

April 13, 2017

In this paper, I will describe how a to compute naively the forward and the backward pass in a convolutional neural network. I based my work on the 2nd assignment of CS231n gave at Stanford in 2016. In this assignment we are asked to compute the forward and backward pass in a convolutional neural network. We're also asked to compute the forward pass and backward pass on the Max-pool layer. I won't detail how to implement it and how to compute it. it is actually straightforward and looks like the ReLu activation layer. I will instead focus on the forward/backward pass of the convolutional layer. I will also derive an intuition on how to compute easily the forward pass and the backward pass of the spatial batch normalization. So let's go.

Chapter 1

Convolutional Layer

1.1 Forward pass

Although the forward pass isn't difficult (all the work reside in computing the backward pass), I think we still need to take our time to understand precisely what quantity is computed during the forward pass of a convolutional layer. To understand this I made several pictures. But First of all let's define the problem.

Problem definition I will use the same dimension as I Stanford class But I will use different notations. So let's define our notations:

- Input x of shape (N, C, H, W)
- Weights w of shape (F, C, $H1$, $W1$)
- Output a (N, F, $H2$, $W2$)

Where :

- N corresponds to the number of images
  - (H, W) are respectively the height and the width of the images (they have same height and width)
  - C is the number of channel (here $C = 3$ corresponding to RGB)
- F is the number of filters
  - ($H1$, $W1$) are respectively the height and the width of the filters (all filters have same height and width within the same convolutional layer)
  - C is the number of channel (here we convolve across all the channel, so the sum (see after) will include all the channels)
- ($H2$, $W2$) are the height and width of the activation map (see how $H2$ and $W2$ are computed later)

Computing the forward pass So now that we set our notations, let's foucs on the forward pass. For simplicity we will first consider only one input image and only one channel of the input image. In such case, during the forward pass of the convolutional layer we have to overlap a filter over our

input image (matrix x). Each time we overlap our filter over x it gives us a number that will be put at the corresponding place in the activation map. We repeat this processus for all filters. If we have f filters we will hence have f activation maps. This procedure is detailed in Figure 1.1

Know that we understand what will be computed during the forward pass let's extend it to c channels and f filters. If we have c channels, we need to "overlap" a window (our window is our filter) over our input x and we need to do this for **each channel**. we then compute the sum of the element-wise product of my f^{th} filter with each of my c layers. we write this newly computed quantity $a_{f,h,w}$. Indeed this quantity doesn't depend on c as we sum up over all c_{th} channels. $a_{f,h,w}$ refer to one number of my activation maps. To better understand this principle you can refer to the Figure 1.2 So at the end, using the example of the Figure 1.2, we can write (mathematically):

$$a_{f,2,1} = \sum_{c=1}^{C} \sum_{i=1}^{H1} \sum_{j=1}^{W1} w_{f,c,i,j} x_{c,2+(i-1),1+(j-1)} + b_f \tag{1.1}$$

each of this newly computed number $a_{f,h,w}$, will be placed on what we call an "activation map". And we have F (number of filters) activation map of size:

$$\begin{aligned} H2 &= (H - H1 + 2pad)/stride + 1 \\ W2 &= (W - W1 + 2pad)/stride + 1 \end{aligned} \tag{1.2}$$

This formula comes from the course. I won't detail it as it is quite straightforward and intuitive. So actually at the end we will have a stack of F activation maps having in each cells $a_{f,h,w}$. See Figure 1.3.

I omitted to mention it but if we are using the formulas given by relation (1.2), that means that in relation (1.1) x refers actually to the padded version of x (see course to understand what is the padded version of x). That in mind we might think that the generalize formula for $a_{f,h,w}$ might look like:

$$a_{f,h,w} = \sum_{c=1}^{C} \sum_{i=1}^{H1} \sum_{j=1}^{W1} w_{f,c,i,j} x^{pad}_{c,h+(i-1),w+(j-1)} + b_f \tag{1.3}$$

But there is a drawback in this formula. Indeed it works when we are dealing with $S = 1$ (stride 1) but what if we are using a stride S of 2, 3, ... ? If we are using such stride we will need to translate our window (filter matrix) by S in vertical and horizontal position to get the next $a_{f,h,w}$, hence instead of having to convolve $w_{f,i,j}$ with $x^{pad}_{c,h+(i-1),w+(j-1)}$ we will have to convolve $w_{f,i,j}$ with $x^{pad}_{c,hS+(i-1),wS+(j-1)}$ and our general formula is just :

$$a_{f,h,w} = \sum_{c=1}^{C} \sum_{i=1}^{H1} \sum_{j=1}^{W1} w_{f,c,i,j} x^{pad}_{c,hS+(i-1),wS+(j-1)} + b_f \tag{1.4}$$

Nice ! So now we have our general formula to compute the forward pass. Well, not exactly. In the assignment they are dealing with N images and so x has shape (N, C, H, W). In our formula x has shape (C, H, W) so it doesn't fit. Actually our formula works just fine if we are dealing with 1 image. If we are dealing with n images we can still use our formula but we have to take care of our indexes. Indeed we need to convolve our n^{th} image (x) with the same w (filter matrix) and we will get $a_{n,f,h,w}$ (the activation map w.r.t to the image and to the filter) instead of just $a_{f,h,w}$, so finally the general formula is:

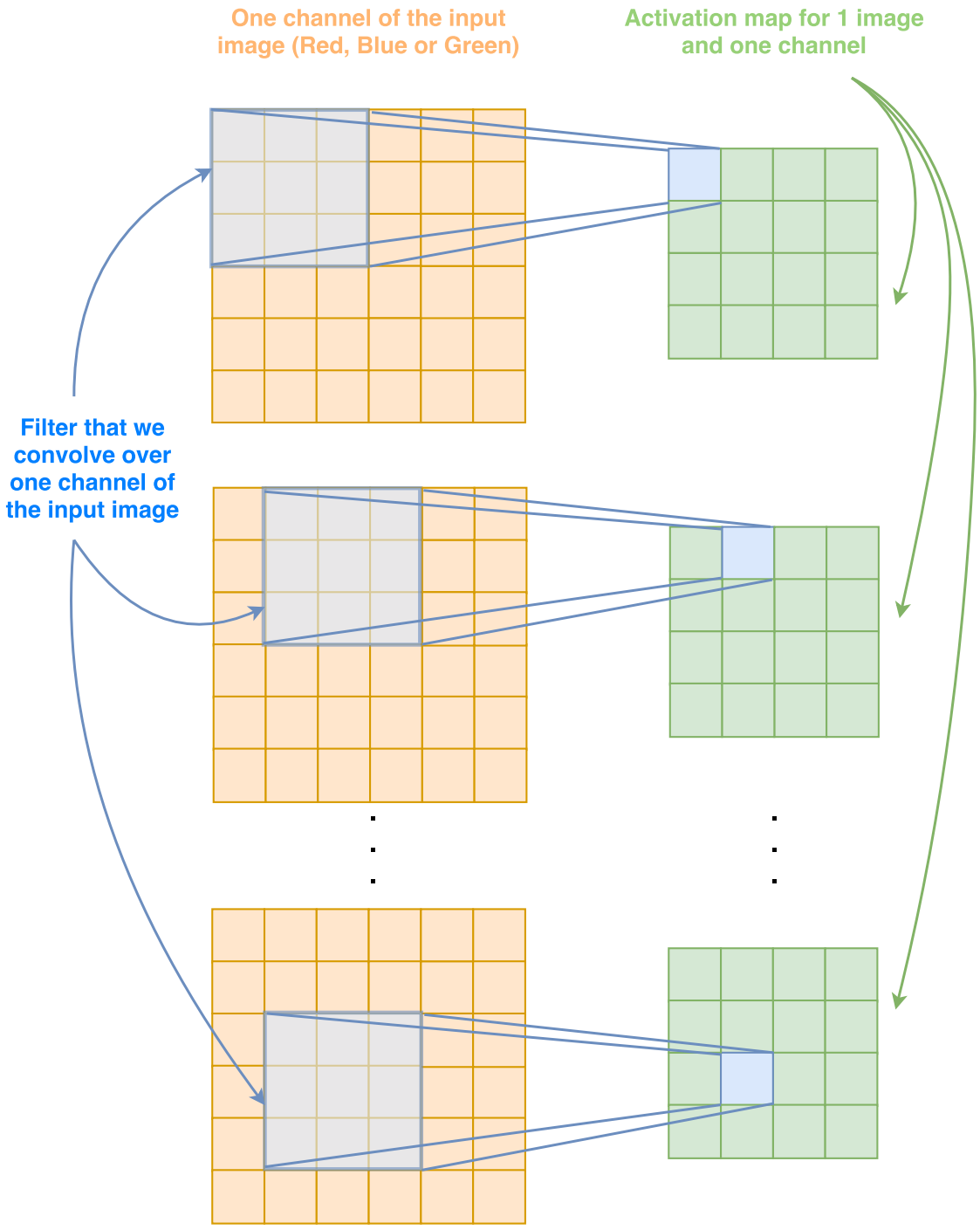


Figure 1.1: Forward pass in a Convolutional Layer for one image, one filter and one channel

$$a_{n,f,h,w} = \sum_{c=1}^{C} \sum_{i=1}^{H1} \sum_{j=1}^{W1} w_{f,c,i,j} x^{pad}_{n,c,hS+(i-1),wS+(j-1)} + b_f \tag{1.5}$$

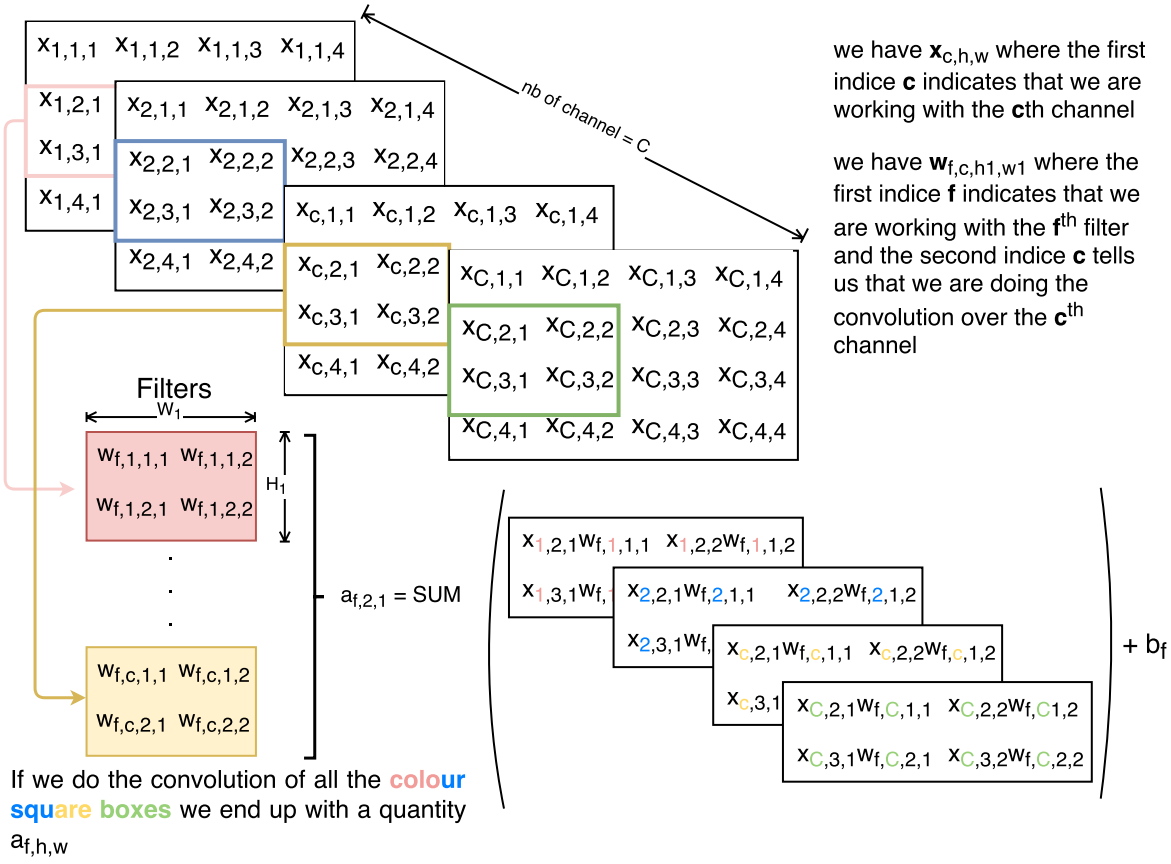


Figure 1.2: Forward pass in a Convolutional Layer

So at the end we will have F activation maps for each image. See Figure 1.4 for a better understanding.

So, now that we understand what quantity is computed during the forward pass, we can implement it in Python (see code in layers.py)

1.2 Backward pass

So we detailed what quantity was computed during the forward pass but what we really want is to compute the backward pass. As we know it is always the difficult point. If we have some previous experiences on convolutional neural network we can do it quite intuitively, or we can used result share on the internet, but what if we really want to do it ourselves ? In this part I will use the relation (1.5) to compute the backward pass in a convolutional layer.

Problem Our goal is to compute the backward pass. Supposing we already have access to the quantity $\frac{\partial L}{\partial a}$ using backpropagation of the layers following the convolutional layer, we want to compute the **gradient** of the loss L w.r.t to the inputs of the convolutional layer. here the inputs are: x (the images as a tensor), b (the bias shared among filters) and w (the filtering tensor composed of

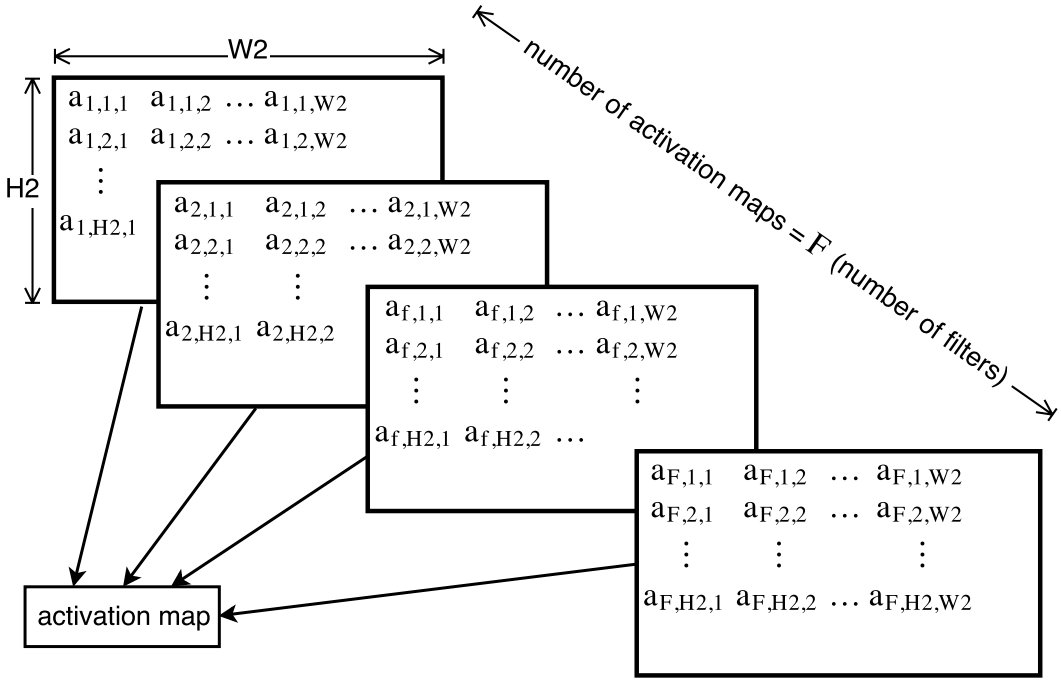


Figure 1.3: Activation map obtains after convolution

our weights). **To make it simple** a tensor is a matrix in higher dimension (matrix is a 2-D array, tensor is a N-D array with N integer)

Computing the backward pass

Gradient of L w.r.t b Let's start by computing the easiest quantity: $\frac{\partial L}{\partial b}$ As b is a bias vector of shape (F, 1), so do $\frac{L}{b}$. To compute this quantity as we suppose we have access to $\frac{\partial L}{\partial a}$, we will use the chain rule in higher dimension. Hence we have:

$$\begin{aligned}
\frac{dL}{db_u} &= \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} \frac{da_{n,f,h,w}}{db_u} \\
&= \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} \frac{d}{db_u} \left(\sum_{c=1}^{C} \sum_{i=1}^{H1} \sum_{j=1}^{W1} w_{f,c,i,j} x^{pad}_{n,c,hS+(i-1),wS+(j-1)} + b_f \right) \\
&= \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} \frac{db_f}{db_u} = \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} 1\{f = u\} = \sum_{n,h,w} \frac{dL}{da_{n,u,h,w}}
\end{aligned} \tag{1.6}$$

Hence the column vector $\frac{\partial L}{\partial b}$ (db in python notation) is the sum of $\frac{\partial L}{\partial a}$ over all axis beside the second axis (f here). See code in layers.py

Gradient of L w.r.t w The second easiest quantity to compute is $\frac{\partial L}{\partial w}$. As before we will compute it using the chain rule. But here we know that w is a (F,C,$H1$,$W1$) tensor so we will have:

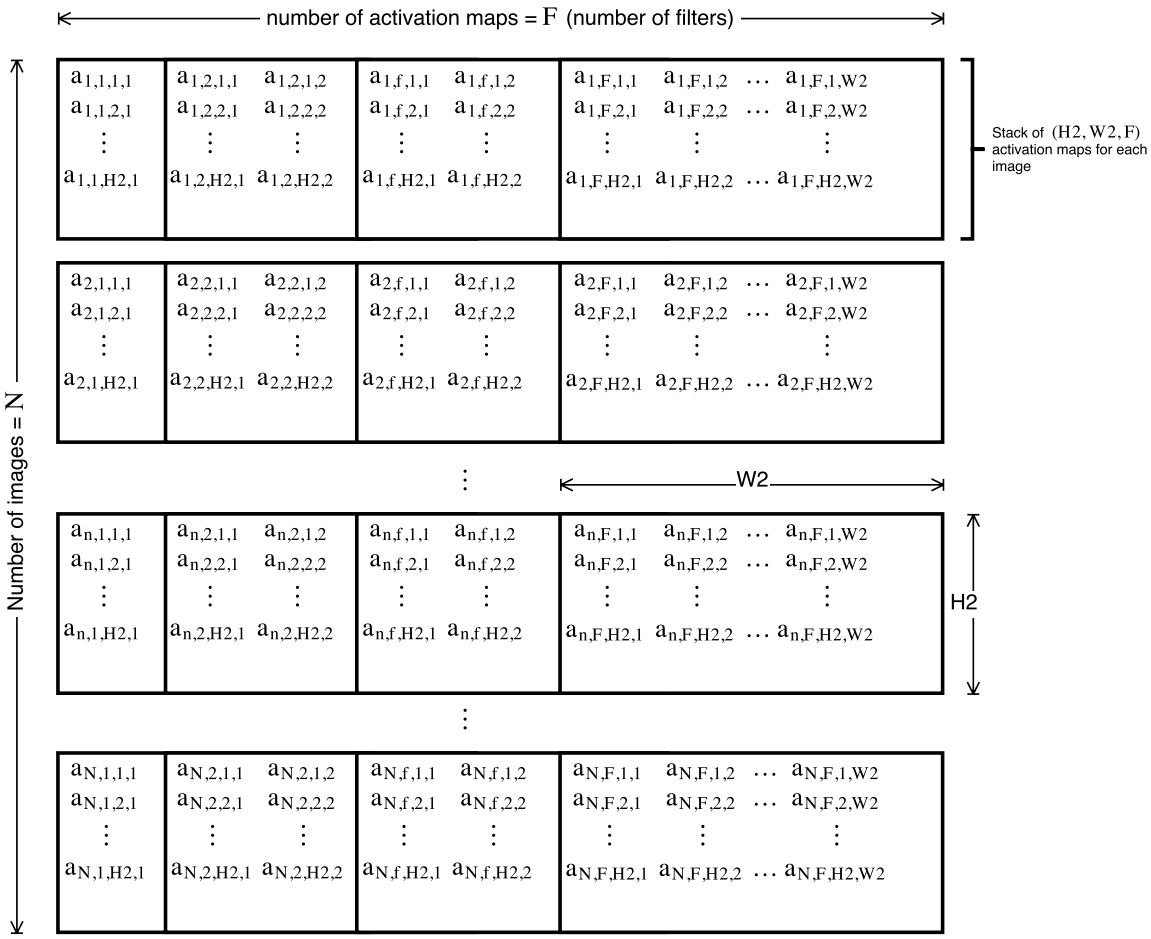


Figure 1.4: Stack of activation maps for each image in a convolution layer

$$\frac{dL}{dw_{f1,c1,h1,w1}} = \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} \frac{da_{n,f,h,w}}{dw_{f1,c1,h1,w1}}$$

$$= \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} \sum_{c=1}^{C} \sum_{i=1}^{H1} \sum_{j=1}^{W1} \frac{dw_{f,c,i,j}}{dw_{f1,c1,h1,w1}} x^{pad}_{n,c,hS+(i-1),wS+(j-1)}$$

$$= \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} \sum_{c=1}^{C} \sum_{i=1}^{H1} \sum_{j=1}^{W1} 1\{f=f1\}1\{c=c1\}1\{i=h1\}1\{j=w1\} x^{pad}_{n,c,hS+(i-1),wS+(j-1)} \quad (1.7)$$

$$= \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} 1\{f=f1\} x^{pad}_{n,c1,hS+(h1-1),wS+(w1-1)}$$

$$= \sum_{n,h,w} \frac{dL}{da_{n,f1,h,w}} x^{pad}_{n,c1,hS+(h1-1),wS+(w1-1)}$$

Here we computed $\frac{dL}{dw_{f1,c1,h1,w1}}$. So naively to implement the code in Python we will have to compute this quantity for each $f1$, $c1$, $h1$, $w1$, so the **naive implementation** will have **at least** 4 loops. The naive implementation might have more loops if the quantity we need to compute that is to say $\sum_{n,h,w} \frac{dL}{da_{n,f1,h,w}} x^{pad}_{n,c1,hS+(h1-1),wS+(w1-1)}$ here need inner loops. The naive implementation of the code is in layers.py

Gradient of L w.r.t x Finally we will deal with the computation of $\frac{\partial L}{\partial x}$. As before we will compute it using the chain rule. So we will have to compute:

$$\begin{aligned}\frac{dL}{dx_{n1,c1,h1,w1}} &= \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} \frac{da_{n,f,h,w}}{dx_{n1,c1,h1,w1}} \\ &= \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} \frac{d}{dx_{n1,c1,h1,w1}} \left(\sum_{c=1}^{C} \sum_{i=1}^{H1} \sum_{j=1}^{W1} w_{f,c,i,j} x^{pad}_{n,c,hS+(i-1),wS+(j-1)} + b_f \right)\end{aligned} \tag{1.8}$$

Here we see that the previous relation use both x^{pad} and x, so to compute the derivative of x^{pad} w.r.t x we will have to find the relationship between x^{pad} and x. So, as we want $\frac{dx^{pad}_{n,c,h,w}}{x_{n1,c1,h1,w1}}$ we will need to find a relationship between $x^{pad}_{n,c,h,w}$ and $x_{n,c,h,w}$. Refer to Figure 1.5 to see this relationship. So, now that we have this relationship we can easily compute :

$$\frac{dx^{pad}_{n,c,h,w}}{x_{n1,c1,h1,w1}} = 1\{n = n1\}1\{c = c1\}1\{h - pad = h1\}1\{w - pad = w1\} \tag{1.9}$$

Using (1.9) in (1.8) we finally have:

$$\begin{aligned}\frac{dL}{dx_{n1,c1,h1,w1}} &= \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} \frac{da_{n,f,h,w}}{dx_{n1,c1,h1,w1}} \\ &= \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} \left(\sum_{c=1}^{C} \sum_{i=1}^{H1} \sum_{j=1}^{W1} w_{f,c,i,j} \frac{dx^{pad}_{n,c,hS+(i-1),wS+(j-1)}}{dx_{n1,c1,h1,w1}} \right) \\ &= \sum_{n,f,h,w} \frac{dL}{da_{n,f,h,w}} \left(\sum_{c=1}^{C} \sum_{i=1}^{H1} \sum_{j=1}^{W1} w_{f,c,i,j} 1\{n = n1\}1\{c = c1\}1\{hS + (i-1) - pad = h1\}1\{wS + (j-1) - pad = w1\} \right) \\ &= \sum_{f,h,w} \frac{dL}{da_{n1,f,h,w}} \left(\sum_{i=1}^{H1} \sum_{j=1}^{W1} w_{f,c1,i,j} 1\{hS + (i-1) - pad = h1\}1\{wS + (j-1) - pad = w1\} \right) \\ &= \sum_{f,h,w} \frac{dL}{da_{n1,f,h,w}} w_{f,c1,pad+h1-hS+1,pad+w1-wS+1} 1\{1 \leq pad + h1 - hS + 1 \leq H1\}1\{1 \leq pad + w1 - wS + 1 \leq W1\}\end{aligned}$$

I implemented this in python using the second to last relation in the previous formula. The last formula is a bit trickier to implement. Finally 2 things I'd like to emphasize. Firstly we have $1\{1 \leq pad+h1-hS+1 \leq H1\}$ that appear in the last formula because we replace i by $pad+h1-hS+1$ but $i \in [1, H1]$ (see the sum over i). It is the same thing for j. Secondly I like to point out the fact that I'm using 1 indexing in my math while in python the index start at 0. That is why in my implementation of dx in python you don't see the -1 in $hS + (i-1) - pad = h1$.

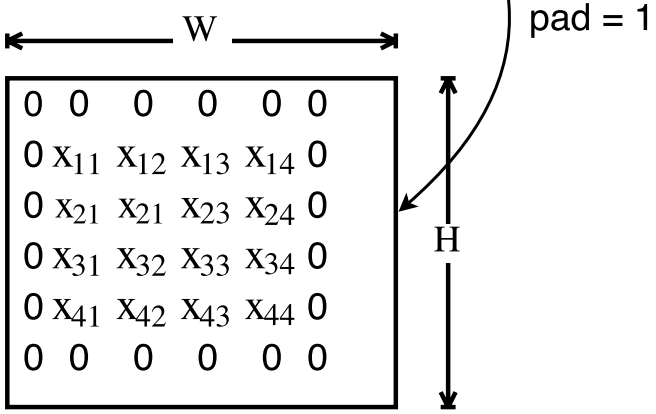
We want to compute:

$$\frac{dx^{pad}_{n,c,h,w}}{dx_{n1,c1,h1,w1}}$$

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{21} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix}$$

According to the figure we see that **if we are in non padding area** of X^{pad} we have:

$$x^{pad}_{n,c,h+1,w+1} = x_{n,c,h,w}$$



So generally **if we are in non padding area** of X^{pad} we have:

$$x^{pad}_{n,c,h+pad,w+pad} = x_{n,c,h,w} \tag{1}$$

We can also rewrite the previous equation:

$$x^{pad}_{n,c,h,w} = x_{n,c,h-pad,w-pad} \tag{2}$$

The previous relationship works fine when we aren't in the padding area. But **if we are in the padding area**, that is to say, if $(h - pad < 1)$ or $(w - pad < 1)$ or $(h - pad > H - 2pad)$ or $(w - pad > W - 2pad)$ we have:

$$x^{pad}_{n,c,h,w} = 0$$

Finally Combining (1) and (2) we have:

$$x^{pad}_{n,c,h,w} = x_{n,c,h-pad,w-pad}\ 1\{pad - 1 < h < H - pad\}1\{pad - 1 < w < W - pad\}$$

Figure 1.5: Relationship between x_{pad} and x

Chapter 2

Spatial Batch Normalization

2.1 Forward pass

As it is stated in the assignment : "If the feature map was produced using convolutions, then we expect the statistics of each feature channel to be relatively consistent both between different images and different locations within the same image. Therefore spatial batch normalization computes a mean and variance for each of the C feature channels by computing statistics over both the minibatch dimension N and the spatial dimensions H and W".
That means that for the forward pass if we want to reuse our batch norm implementation that takes an input of size (N, D) with N being the minibatch dimension we will need to pass an input of size ($N \times H \times W$, C). Having that in mind the forward implementation is straightforward:

```
# wrong code
N, C, H, W = x.shape
# -1 is use to complete with the right dimension
# I could have used x.reshape(N*H*W, C)
x_tmp = x.reshape(N*H*W, -1)
gamma_c = np.resize(gamma, (C, 1))
beta_c = np.resize(beta, (C, 1))
out, cache = batchnorm_forward(x_tmp, gamma_c, beta_c, bn_param)
out = out.reshape((N, C, H, W))
```

Well actually the previous code **doesn't work** ! I decided to put it here because I made the mistake myself. What's wrong with this code ? It's quite simple. Of course we reshape x to have shape ($N \times H \times W$, C), but the thing is that x has shape (N,C,H,W), so if we apply reshape() on our data without swaping our axis such that we have x of shape (N, H, W, C), the reshape function will reshape our input x with the wrong data. So we need to swap the axis before reshaping the data (at the end we need to reshape our data and then re-swap the axis) so *out* is of the same shape as x was in the being, that is to say: (N,C,H,W). So actually we can come up with the following code (I used **swapaxes** but they are better ways do it):

```
N, C, H, W = x.shape

# (N, W, H, C) size after swaping axes C and W
x_tmp = np.swapaxes(x, 1, 3)

# then we can reshape correctly
x_tmp = x_tmp.reshape(N*W*H, -1)

out_tmp, cache = batchnorm_forward(x_tmp, gamma, beta, bn_param)

# we do the reverse to have the right shape
out = out_tmp.reshape((N, W, H, C))
out = np.swapaxes(out, 1, 3)
```

2.2 Backward pass

I won't detail the backward pass, if you didn't do the same mistake that I made during the forward pass then the backward pass is straightforward. I won't put the code here. You can see it in layers.py

Conclusion Here we've learned a lot. We actually see how a Convolutional Layer works. We apply forward and backward pass and we saw how we can easily reuse previous function to implement new function in higher dimension.