CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS SUPERIORES DEL IPN

A beginner tutorial for CNN

Edwin Efraín Jiménez Lepe
eejimenez@gdl.cinvestav.mx
and
Andrés Méndez Vázquez
amendez@gdl.cinvestav.mx

November 8, 2016

Contents

1 INTRODUCTION

This document is intended to provide a clear and brief introduction to Convolutional Neural Networks, it should make simple to understand how layer works, how to train the whole network, how to implement it using Theano to take advantage of GPU's power, and also gives you ideas to propose your own architecture.

For that, the tutorial is going to be split in three sections: First one is a brief reminder of perceptron, multilayer perceptron and backpropagation as training algorithm, Second one explain layers details, the last one shows how to join the layers to create a full model and use it for classification tasks.

The notation and some definitions are taken just like the author proppose in the correspondence reference, this is to facilitate the reader to check external information.

A prior knowledge of artificial neural networks is expected, also you can read [12] to see the first application of CNN.

We hope that this tutorial is useful for you, also we highly recommend you to read this alternatives notes [2, 18, 16, 8, 9, 20] because CNN its a large topic and read external resources may help you to clarify you understanding.

If you have any comment we wait your feedback in the mail addresses mentioned in the cover.

May the math be with you.

2 Neural Networks

This whole section is based on the book "Neural Networks" from Haykin[5]. We are going to talk about multilayer perceptron, so it's highly recomend to read the chapters 3 and 4 of Haykin's book for better understanding.

First of all we define a *neuron* as the basic unit of the neural networks (and for consequence for Convolutional Neural Networks) and detail the three elements of it:

- Synapses or connecting links: The conection between an input and a specific neuron. Is characterized by a *weight*. The notation for this values is w_{kj} where k represents the neuron and j represents the input signal x_j.
- Adder: To sum all the input signals according to the corresponding weight.
- Activation function: To limit the range of neuron's output.

Another element is the *bias* which help us to increase or decrease the output of a neuron before we apply the activation function, it's represented by b_k. You can see an example in Figure 2.1

Mathematically we can represent the model of a neuron as

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{2.1}$$

and

$$y_k = \varphi(u_u + b_k) \tag{2.2}$$

where y_k represents the output signal of the neuron.

We can have a lot of neurons sharing the same inputs but with different synapses and we call it a *layer*. When we have one layer or more we call it *multilayer perceptron* which consist of a *input layer* where every unit is a input signal, one or more *hidden layers* where every unit is a neuron, and an *output layer* where the network give us the result of processing the input signal, which traverses the network layer-by-layer in a forward direction.

The *error back-propagation algorithm* is widely used to train this kind of network. It consists of two phases: a *forward pass* when we receive the input signal and it traverses the network until the output layer, and *backward pass* where the synaptic weights are adjusted accoding to an error-correction rule.

That's why you're going to find in the subsections of each layer *feed forward (forward pass)* and *back propagation (backward pass)*.

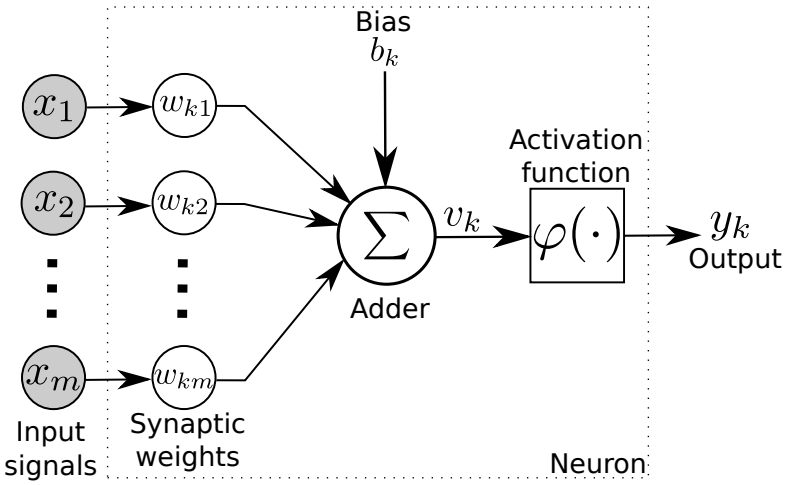

Figure 2.1: Example of a neuron

2.1 FeedForward and Backpropagation

Figure 2.2 show a multilayer perceptron with a input layer, two hidden layers and an output layer. This network is fully connected, it means that all outputs of a layer are received as input by all units in the next layer. The input signal is propagate through the network from left to right, layer-by-layer until it reach the output of the network, we call it *function signal*. This signal is calculated as a function of the inputs and associated weights applied to that neuron.

On the other hand, an error signal originates at an output neuron, and it's recalculated for every neuron to correct the synapses weights according to the expected output, Figure 2.3 shows the error signal flow during backpropagation on the same architecture.

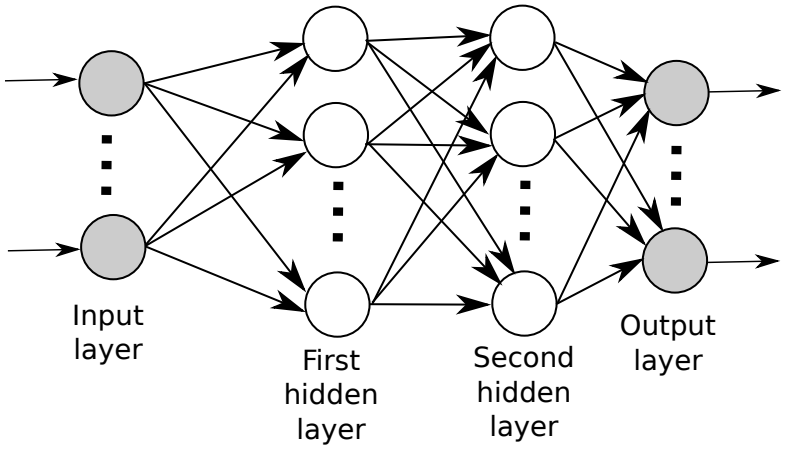

Figure 2.2: Example of feed forward in MLP

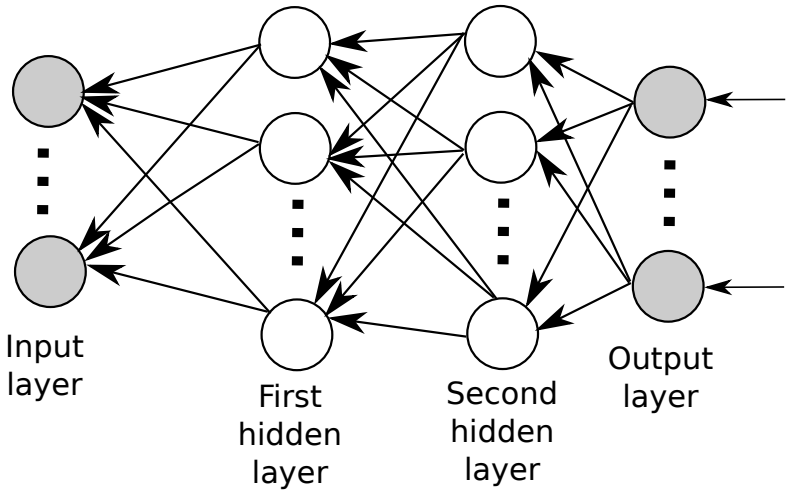

Figure 2.3: Example of backpropagation in MLP

2.2 Summary of the back-propagation algorithm

Haykin defines five steps to apply the training algorithm:

1. Initialization: Usually bias is initialized in 0 and weights are picked from a uniform distribution with zero mean, sometimes in the range [-1,1].

2. Presentations of Training Examples: When you complete an iteration over al training examples it's called *epoch*. For each element in the training set of examples you should apply steps 3 and 4.[1]

3. Forward Computation: We can denote the current training example as $(x(n), d(n))$, where $x(n)$ is a vector of features that we use in the input layer and $d(n)$ is the desired output of the network for that example. "Compute the induced local fields and function signals of the network by proceeding forward through the network, layer by layer. The induced local field $v_j^l(n)$ for neuron j in layer l is

$$v_j^l(n) = \sum_{i=0}^{m_0} w_{ji}^{(l)}(n) y_i^{l-1}(n) \tag{2.3}$$

where $y_i^{l-1}(n)$ is the output (function) signal of neuron i in the previous layer $l-1$ at iteration n, and $w_{ji}^{(l)}(n)$ is the synaptic weight of neuron j in layer l that is fed from neuron i in layer $l-1$. For $i=0$, we have $y_0^{l-1}(n) = +1$ and $w_{j0}^l(n) = b_j^{(l)}(n)$ is the bias applied to neuron j in layer l. Assuming the use of a sigmoid function, the output signal of neuron j in layer l is

$$y_j^{(l)} = \varphi_j(v_j(n)) \tag{2.4}$$

[1] You can divide the examples of the epoch in *mini-batches* of the same size and compute steps 3 and 4 with it.

If neuron j is in the first hidden layer (i.e., $l = 1$), set

$$y_j^{(0)} = x_j(n) \tag{2.5}$$

where $x_j(n)$ is the jth element of the input vector $x(n)$. If neuron j is in the output layer (i.e., $l = L$, where L is referred to as the depth of the network), set

$$y_j^{(L)} = o_j(n) \tag{2.6}$$

Compute the error signal

$$e_j(n) = d_j(n) - o_j(n) \tag{2.7}$$

where $d_j(n)$ is the jth element of the desired response vector $d(n)$."

4. Backward Computation: "Compute the δs (i.e., local gradients) of the network, defined by

$$\delta_j^{(l)}(n) = \begin{cases} e_j^{(L)}(n)\varphi'_j\left(v_j^{(L)}(n)\right), & \text{for neuron } j \text{ in output layer L} \\ \varphi'_j\left(v_j^{(L)}(n)\right)\sum_k \delta_k^{(l+1)}(n)\, w_{kj}^{(l+1)}(n), & \text{for neuron } j \text{ in hidden layer l} \end{cases} \tag{2.8}$$

where the prime in $\varphi'_j(\cdot)$ denotes differentiation with respect to the argument. Adjust the synaptic weights of the network in layer l according to the generalized delta rule:

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha\left[w_{ji}^{(l)}(n-1)\right] + \eta\delta_j^{(l)}(n)\, y_i^{l-1}(n) \tag{2.9}$$

where η is the learning-rate parameter and α is the momentum constant."

5. Iteration: Repeat steps 3 and 4 until the error is in an aceptation range or when the rate of change in the error is under a valid treshold.

3 LAYERS

3.1 CONVOLUTION

The convolution in 2D is equivalent to apply a filter in a image. It help us to learn the features that distinguished the images classes, instead of use hand-crafted features we trust the CNN. It is the main layer of CNNs, the use of this kind of layer help in the last years to obtain the best state-of-the-art results on image classification for small (mnist, cifar-10) and really challenging datasets (ImageNet) as well.

3.1.1 BACKGROUND

This subsection is extracted from [14].
"The convolution integral is expressed in one dimension by the relationship

$$g(t) = \int_{-\infty}^{\infty} f(\tau) h(t-\tau) d\tau \tag{3.1}$$

This represents the convolution of two time functions, $f(t)$ and h(t); commonly $f(t)$ is a time varying signal, e.g. speech, and $h(t)$ is the impulse (time) response of a particular filter. τ is a dummy variable which represents the shift of one function with respect to the other, as illustrated in Figure 3.1

The importance of the convolution integral is based to a large extent on the **convolution theorem** which relates multiplication in the time domain to convolution in the frequency domain and vice versa. Representing the convolution of two functions by the symbol $*$, then

$$f(t)h(t) \equiv F(w) * H(w) \tag{3.2}$$

and

$$F(w)H(w) \equiv f(t) * h(t) \tag{3.3}$$

In 2-dimensional image processing terms, the continuous convolution integral may be expressed

$$g(x,y) = f(x,y) * h(x,y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\tau_u, \tau_v)\, h\left(x-\tau_u, y-\tau_v\right) d\tau_u d\tau_v \tag{3.4}$$

In a manner analogous to one-dimensional convolution, the function $h(0-\tau_u, 0-\tau_v)$ is simply the image function $h(\tau_u, \tau_v)$ rotated by 180 degrees about the origin. The function $h\left(x-\tau_u, y-\tau_v\right)$ is the function further translated to move the origin of the image function g to the point (x,y) in the (m,n) plane. The functions are then point-wise multiplied and the product function is integrated over 2 dimensions. Convolution of digital sampled images is analogous to that for continuous images, except that the integral is transformed to a discrete summations over the image dimensions, m and n.

$$g(x,y) = \sum_{\tau_u} \sum_{\tau_v} f(\tau_u, \tau_v)\, h\left(x-\tau_u, y-\tau_v\right) \tag{3.5}$$

Since both $f(x,y)$ and $h(x,y)$ are non-zero over a finite domain, the field of view of the imaging system, the summation is necessary only over the area of non-zero overlap. The number of required multiply and add operations is equal to the number of pixels in $h(x,y)$ times the number of pixels in $f(x,y)$.

Hence an algorithm for digital convolution of an image $f(x,y)$ of pixel dimensions p by q with a mask, $h(x,y)$ of dimensions 2m+1 by 2n+1, is expressed:"

Algorithm 3.1: PSEUDOCODING OF CONVOLUTION(image, mask)

1. For each row, $i = 1$ to p
2. For each column, $j = 1$ to q
3. Process pixel (i, j)
4. Set output image pixel, $g(i, j) = 0$
5. For each mask row, $k = -m$ to m
6. For each mask column, $l = -n$ to n
7. $g(i, j) = g(i, j) + f(i + k, j + l)h(k, l)$

You can see an example of convolution in 2 dimensions in Figure 3.2, extracted from [6]

On the other hand **correlation** can be defined in a similar way that convolution, but without the rotation of one of the functions.
From [17]
The cross-correlation of two complex functions $f(t)$ and $g(t)$ of a real variable t, denoted $f \star g$ is defined by

$$f \star g \equiv \bar{f}(-t) * g(t) \tag{3.6}$$

where $*$ denotes convolution and $\bar{f}(t)$ is the complex conjugate of $f(t)$. Since convolution is defined by

$$f * g = \int_{-\infty}^{\infty} f(\tau) g(t-\tau) d\tau \tag{3.7}$$

it follows that

$$[f \star g](t) = \int_{-\infty}^{\infty} \bar{f}(-\tau) g(t-\tau) d\tau \tag{3.8}$$

Letting $\tau' \equiv -\tau$, $d\tau' = -d\tau$, so Equation 3.8 is equivalent to

$$f \star g = \int_{-\infty}^{\infty} \bar{f}(\tau') g(t+\tau')(-d\tau') = \int_{-\infty}^{\infty} \bar{f}(\tau) g(t+\tau) d\tau \tag{3.9}$$

Also, the 2D correlation can be expressed as Equation 3.10 you can compare that with Equation 3.5

$$g(x,y) = \sum_{\tau_u} \sum_{\tau_v} f(\tau_u, \tau_v)\, h\left(x + \tau_u, y + \tau_v\right) \tag{3.10}$$

On the Table 3.1 you can see and example of convolution vs. correlation

Table 3.1: Convolution vs Correlation

Input

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

w

1	2	3
4	5	6
7	8	9

input $*$w

1	2	3
4	5	6
7	8	9

input $\star$ w

9	8	7
6	5	4
3	2	1

Input2

8	7	2	0	4
7	6	1	1	1
5	2	8	5	6
4	0	3	3	8
8	8	3	0	7

w2

9	5	1
8	0	5
2	7	5

input2 $*$w2

223	174	118
199	131	192
174	123	232

input2 $\star$ w2

234	211	116
197	137	176
197	115	208

Input3

5	3	7
8	1	7
2	4	6

w3

5	7
4	1

input3 $*$w3

78	73
46	87

input3 $\star$ w3

79	75
59	76

Input4

3	2	4
2	1	2
4	2	3

w4

1	2
2	1

input4 $*$w4

12	14
14	12

input4 $\star$ w4

12	14
14	12

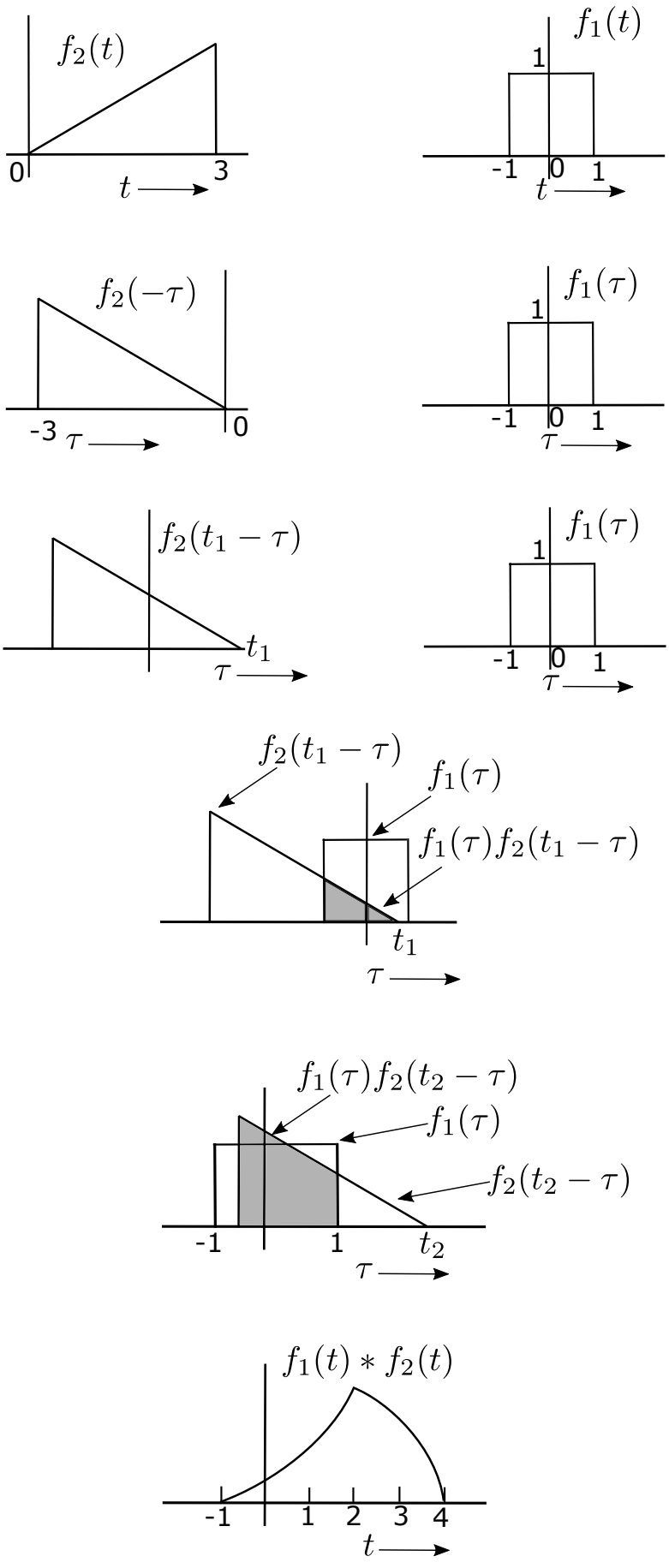

Figure 3.1: Convolution in one dimension, extracted from[14]

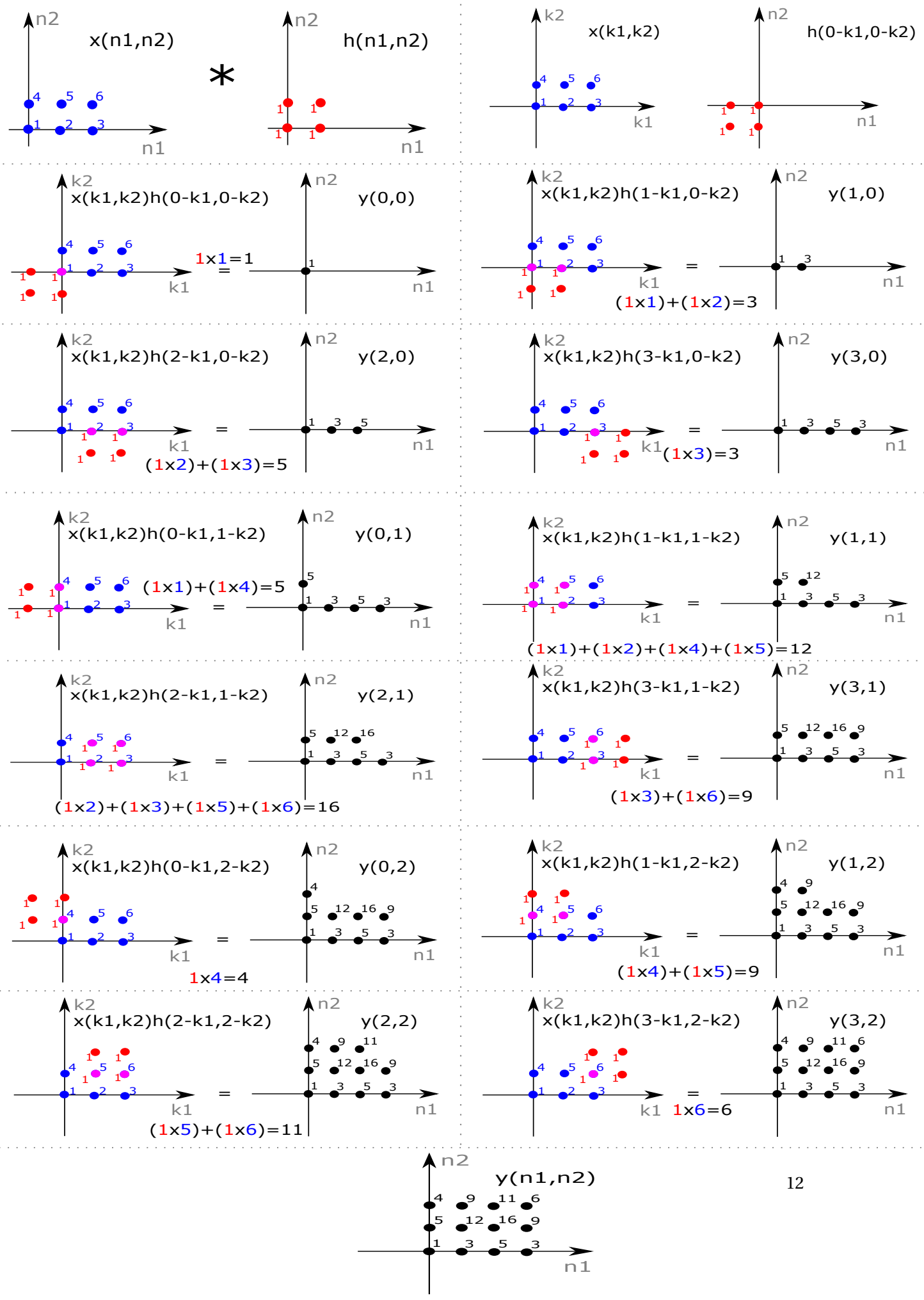

Figure 3.2: Convolution in two dimensions where $y(n_1, n_2) = x(n_1, n_2) * h(n_1, n_2) = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} x(k_1, k_2) h(n_1 - k_1, n_2 - k_2)$

3.1.2 Feedforward

It's pretty simple, but computationally expensive. We denote the convolution operation as $*$. It is like moving a window over a subregion of the input image to make a kind of Hadamard product, because in the subregion we do an element-wise multiplication to sum all the results of the window and (sometimes) add a bias, the same process is repeated until all the possible subregions are covered.

Before define convolution there are two important concepts:

- **Stride**: It is the number of spaces that we move to the right before reach the end of the image, and the spaces we move to below before we reach the end of the image.

Table 3.2: Subregions of the image with stride = 1

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

Table 3.3: Subregions of the image with stride = 2

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

23	-12	16	90
12	32	12	45
-1	7	8	9
-2	-12	14	56

- **Padding**: Sometimes we want to take advantage of all the pixels in the image, so a padding just indicates how many columns and rows of zeros we are going to add in

the border of the image. Also, if you want to apply a 'full' convolution you need to add a (wf-1) padding, where wf is the width of the filter.

Table 3.4: Padding to a 4x4 image

X

-12	16	90	34
32	12	45	-8
7	8	9	12
-12	14	56	18

X with 1-padding of zeros

0	0	0	0	0	0
0	-12	16	90	34	0
0	32	12	45	-8	0
0	7	8	9	12	0
0	-12	14	56	18	0
0	0	0	0	0	0

X with 2-padding of zeros

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	-12	16	90	34	0	0
0	0	32	12	45	-8	0	0
0	0	7	8	9	12	0	0
0	0	-12	14	56	18	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

According to [18] 'for a convolution, if the input is $H^l \times W^l \times D^l$ (where H, W, D represents height, width and channels of the image) and the kernel size is $H \times W \times D^l \times D$, the convolution result has size $(H^l - H + 1) \times (W^l - W + 1) \times D$.

If we consider a simple case, when the stride is 1 and no padding is used. Hence, we have y (or x^{l+1}) in $\mathbb{R}^{H^{l+1} \times W^{l+1} \times D^{l+1}}$, with $H^{l+1} = H^l - H + 1, W^{l+1} = W^l - W + 1$, and $D^{l+1} = D$

In precise mathematics, the convolution procedure can be expressed as an equation:

$$y_{i^{l+1},j^{l+1},d} = \sum_{i=0}^{H} \sum_{j=0}^{W} \sum_{d^l=0}^{D^l} f_{i,j,d^l,d} \times x^l_{i^{l+1}+i,j^{l+1}+j,d^l} \tag{3.11}$$

The Equation 3.11 is repeated for all $0 \leq d \leq D = D^{l+1}$, and for any spatial location $\left(i^{l+1}, j^{l+1}\right)$ satisfying $0 \leq i^{l+1} < H^l - H + 1 = H^{l+1}$,$0 \leq j^{l+1} < W^l - W + 1 = W^{l+1}$. In this equation $x^l_{i^{l+1}+i,j^{l+1}+j}, d^l$ refers to the element of x^l indexed by the triplet $\left(i^{l+1}+i, j^{l+1}+j, d^l\right)$.
A bias term b_d is usually added to $y_{i^{l+1},j^{l+1},d}$.'

Once we have defined the convolution for images in the format of height, width and channels we can formulate some examples to clarify the idea. Also it's important to know that we can manage the input dimensions in the form $D^l \times H^l \times W^l$, rearranging kernel dimensions $D \times D^l \times H \times W$. A little parentheses, when we apply the convolution we have a kernel with rotation of 180°. For the examples in this tutorial we are going to assume that the kernel is already rotated and use as input for convolution instead of the original kernel.

3.1.2.1 Examples of Convolution

- Input 3x3x1 image and 2x2x1x1 kernel, stride = 1, padding=0. The expected output dimension is 2x2x1. See Table 3.5

Table 3.5: Convolution to a 3x3 image

Input

16	24	32
47	18	26
68	12	9

Kernel

0	-1
1	0

Output of Convolution

23	-14
50	-14

But how is the step by step in convolution?, See Table 3.6

Table 3.6: Convolution operation

16 * 0	24*-1	32
47*1	18*0	26
68	12	9

23	0
0	0

16	24	32
47*0	18*-1	26
68*1	12*0	9

23	0
50	0

16	24*0	32*-1
47	18*1	26*0
68	12	9

23	-14
50	0

16	24	32
47	18*0	26*-1
68	12*1	9*0

23	-14
50	-14

- The same example, but using a bias associated with the kernel. We apply the same operation but in the end we just add the bias to any position of the output is shown in Table 3.7.

Table 3.7: Convolution to a 3x3 image with bias

Input

16	24	32
47	18	26
68	12	9

Kernel

0	-1
1	0

Bias

1

Output of Convolution

24	-13
51	-13

- Input 3x3x1 image and 2x2x1x2 kernel, stride = 1, padding=0. The expected output dimension is 2x2x2. It means that we have two kernels of 1 channel of 2x2. So like the first example we apply any kernel to get the correspondent output Table 3.8.

Table 3.8: Convolution to a 3x3 image with two kernels

Input

16	24	32
47	18	26
68	12	9

Kernel

0	-1
1	0

5	4
3	2

Output of Convolution

23	-14
50	-14

353	354
535	248

- Input 3x3x3 image and 2x2x3x2 kernel, stride = 1, padding=0. The expected output dimension is 2x2x2. It means that we have two kernels of 3 channels of 2x2 Table 3.9. In this case we apply the convolution for every kernel, for each case we have a 3 channel input and a 3 channel kernel so we do the multiplication and add for every channel of the kernel with the corresponding subregion in the input, finally we add the correspondent output into one single channel, we have 2 kernels, so the output consist on 2 channels.

Table 3.9: Convolution to a 3x3x3 image with two kernels

Input

16	24	32
47	18	26
68	12	9

26	57	43
24	21	12
2	11	19

18	47	21
4	6	12
81	22	13

Kernel 1

0	-1
1	0

5	4
3	2

16	24
68	-2

Kernel 2

60	22
32	18

35	46
7	23

78	81
20	42

Output of Convolution

2171	2170
5954	2064

13042	13575
11023	6425

3.1.2.2 Convolution as Vector Operation

Sometimes (depending on the dimensions of the input and the kernel) we can apply convolution in a faster way. But we need extra memory to generate another representation for the input and the kernel, also sometimes we need to rearrange the output for the upper layers.

To accomplish that we need another operation (besides matrix multiplications of course) **im2col**, you can find and use this operation in **octave** if you want to goo deeper and also read [15]. Here we are going to explain in a superficial way how it works and how to perform the convolution as matrix multiplication.

For the im2col operation, we generate a matrix based on the input, where any column is a subregion where the filter will be applied, it is useful when we want to apply multiple kernels to the same input.

An example of im2col for a 3x3 input.

Table 3.10: im2col of a 3x3 image to apply a 2x2 kernel

Input		
16	24	32
47	18	26
68	12	9

Output			
16			
47			
24			
18			

16	24	32
47	18	26
68	12	9

16	47		
47	68		
24	18		
18	12		

16	24	32
47	18	26
68	12	9

16	47	24	
47	68	18	
24	18	32	
18	12	26	

16	24	32
47	18	26
68	12	9

16	47	24	18
47	68	18	12
24	18	32	26
18	12	26	9

If we want to apply the operation im2col to a kernel is similar, also we can save steps joining both outputs for the matrix multiplication, Table 3.11.

So, we have a new matrix based on the input and a new matrix based on the kernel, to make convolution we just multiply both matrices, Table 3.12. As you can see we have the same output as if we apply the normal convolution see Table 3.8.

Now, we can have another example for multiple channels kernels, see Table 3.13.

Table 3.11: im2col of 2x2x1x2 kernel

Input

0	-1
1	0

5	4
3	2

Output

0
1
-1
0

5
3
4
2

Output together

0	5
1	3
-1	4
0	2

Table 3.12: Convolution as matrix multiplication

Input

16	47	24	18
47	68	18	12
24	18	32	26
18	12	26	9

Kernel

0	5
1	3
-1	4
0	2

Output

23	353
50	535
-14	354
-14	248

Rearrange

23	-14
50	-14

353	354
535	248

Table 3.13: Convolution to a 3x3x3 image with two kernels

Input

16	24	32
47	18	26
68	12	9

26	57	43
24	21	12
2	11	19

18	47	21
4	6	12
81	22	13

im2col(input).T

16	47	24	18
47	68	18	12
24	18	32	26
18	12	26	9
26	24	57	21
24	2	21	11
57	21	43	12
21	11	12	19
18	4	47	6
4	81	6	22
47	6	21	12
6	22	12	13

Kernel 1

0	-1
1	0

5	4
3	2

16	24
68	-2

Kernel 2

60	22
32	18

35	46
7	23

78	81
20	42

im2col(kernels)

0	60
1	32
-1	22
0	18
5	35
3	7
4	46
2	23
16	78
68	20
24	81
-2	42

output =
im2col(input) × im2col(kernels)

2171	13042
5954	11023
2170	13575
2064	6425

Rearrange

2171	2170
5954	2064

13042	13575
11023	6425

3.1.3 BACKPROPAGATION

To backpropagate the error in a Convolutional Layer we need to calculate the gradient w.r.t three related parameters: W, B and the Input.

3.1.3.1 BIAS

We need to think about the relation between the bias and the result of a convolution. The bias add some specific value to the result in every channel, so for the error that we receive every value of the bias needs to change according to the error that any channel have. It means, we have to sum all the error for each channel, see Equation 3.12, where $\frac{\partial L}{\partial b_{l-1,k}}$ is the error respect to bias at convolutional layer in channel k, and $\frac{\partial L}{\partial y_{l,k}}$ is the error propagated by the upper layer at channel k, also in the sums 'm' is the number of rows and 'n' is the number of columns at k channel in the upper layer.

$$\delta b_{l-1} = \frac{\partial L}{\partial b_{l-1,k}} = \sum_{i}^{m}\sum_{j}^{n}\frac{\partial L}{\partial y_{l,k,i,j}} \tag{3.12}$$

As an example let's imagine we receive an upper layer error of two channels with dimensions 3x3, it have 2 channels so we can surely assume that the correspondent convolution implies two filters and every filter have an associated value as bias. So the expected dimension of δb should be 2×1 (or 1×2, depends how you manage it).

Table 3.14: $\frac{\partial L}{\partial b_{l-1}}$

$\frac{\partial L}{\partial y_l}$
3.31
2.61

0.5	0.3	0.13
0.2	0.65	0.18
0.7	0.23	0.42

0.75	0.12	0.68
0.08	0.04	0.15
0.21	0.24	0.34

3.1.3.2 WEIGHTS

There are two really nice tutorials that you can check to understand the backpropagation for the weights and the input [4, 10].

For this subsection we are going to use δ given by [10], see Equation 3.13 where x is the input to the convolutional layer and $\delta^{(y_l)} = \frac{\partial L}{\partial y_l}$.

$$\delta w_{l-1} = \frac{\partial L}{\partial w_{l-1}} = \delta^{(y_l)} * x = x * \delta^{(y_l)} \text{(performing valid convolution)} \tag{3.13}$$

As an example (this values also will be used in the example of backpropagation of the input) Table 3.15.

Table 3.15: $\frac{\partial L}{\partial w_{l-1}}$

$\frac{\partial L}{\partial y_l}$

0	0
-2.94504954e-05	0

0	0
6.39539432e-06	0

x

16	24	32
47	18	26
68	12	9

$\frac{\partial L}{\partial w_{l-1}} = \delta^{(y_l)} * x$

-1.38417328e-03	-5.30108917e-04
-2.00263369e-03	3.53405945e-04

-1.38417328e-03	-5.30108917e-04
-2.00263369e-03	3.53405945e-04

3.1.3.3 Input

To backpropagate the error in for the down layers, we need to cosnider if we apply kernel rotation. In that case we simply do Equation 3.14, if don't just apply the same without the flip of w (which means a rotation of 180°).

$$\delta y_{l-1} = \delta y_l * \text{flip}(w) \text{(performing full convolution)} \tag{3.14}$$

As an example see Table 3.16.

Table 3.16: δy_{l-1}

$\frac{\partial L}{\partial y_l}$ with 1 zero padding
to perform full convolution

0	0	0	0
0	0	0	0
0	-2.94504954e-05	0	0
0	0	0	0

0	0	0	0
0	0	0	0
0	6.39539432e-06	0	0
0	0	0	0

w

0	1
-1	0

2	3
4	5

$\delta y_{l-1} = \delta y_l * w$

0	0	0
0.3198e-04	0.5503e-04	0
-0.1026e-04	0.1279e-04	0

3.1.3.4 Backpropagation as Vector Operation

As you imagine we can also perform the backpropagation operations as matrix multiplications, here we are going to see examples of how it works.

Table 3.17: Vectorial example of $\frac{\partial L}{\partial w_{l-1}} = \delta y_l * x$

x

16	24	32
47	18	26
68	12	9

im2col(x)

16	47	24	18
47	68	18	12
24	18	32	26
18	12	26	9

δy_l

0	0
-2.94504954e-05	0

0	0
6.39539432e-06	0

im2col(δy_l)

0	0
-2.94504954e-05	6.39539432e-06
0	0
0	0

im2col(x) × im2col(δy_l)

-1.38417328e-03	3.00583533e-04
-2.00263369e-03	4.34886814e-04
-5.30108917e-04	1.15117098e-04
-3.53405945e-04	7.67447318e-05

rearrange of im2col(x) × im2col(δy_l)

-1.38417328e-03	-5.30108917e-04
-2.00263369e-03	-3.53405945e-04

3.00583533e-04	1.15117098e-04
4.34886814e-04	7.67447318e-05

Table 3.18: Vectorial example of $\delta y_{l-1} = \delta y_l * w$ (full convolution)

$\frac{\partial L}{\partial y_l}$ with 1 zero padding

0	0	0	0
0	0	0	0
0	-2.94504954e-05	0	0
0	0	0	0

w

0	1
-1	0

0	0	0	0
0	0	0	0
0	6.39539432e-06	0	0
0	0	0	0

2	3
4	5

im2col($\frac{\partial L}{\partial y_l}$)

0	0	0	0	0	-2.94e-05	0	0	0
0	0	0	0	-2.94e-05	0	0	0	0
0	0	-2.94e-05	0	0	0	0	0	0
0	-2.94e-05	0	0	0	0	0	0	0
0	0	0	0	0	6.395e-06	0	0	0
0	0	0	0	6.395e-06	0	0	0	0
0	0	6.395e-06	0	0	0	0	0	0
0	6.395e-06	0	0	0	0	0	0	0

im2col(w)

0
-1
1
0
2
4
3
5

im2col($\frac{\partial L}{\partial y_l}$).T × im2col(w)

0
0.3198e-04
-0.1026e-04
0
0.5503e-04
0.1279e-04
0
0

rearrange of im2col($\frac{\partial L}{\partial y_l}$).T × im2col(w)

0	0	0
0.3198e-04	0.5503e-04	0
-0.1026e-04	0.1279e-04	0

Table 3.19: A vectorial example of backpropagation, assuming a input of 3×3×3, 2 kernels of 3×2×2. Calculating δy_{l-1}

$\frac{\partial L}{\partial y_l}$

.1678	.098
.002	.246

.5	.67
.21	.487

w1

0	1
-1	0

2	3
4	5

-2	68
24	16

w2

18	32
22	60

23	7
46	35

42	20
81	78

input

16	24	32
47	18	26
68	12	9

26	57	43
24	21	12
02	11	19

18	47	21
4	6	12
81	22	13

$\frac{\partial L}{\partial y_l}$ with 1 zero padding

0	0	0	0
0	.1678	.098	0
0	.002	.246	0
0	0	0	0

0	0	0	0
0	.5	.67	0
0	.21	.487	0
0	0	0	0

im2col($\frac{\partial L}{\partial y_l}$) with 1 zero padding

0	0	0	0	.1678	.002	0	.098	.246
0	0	0	.1678	.002	0	.098	.246	0
0	.1678	.002	0	.098	.246	0	0	0
.1678	.002	0	.098	.246	0	0	0	0
0	0	0	0	.5	.21	0	.67	.487
0	0	0	.5	.21	0	.67	.487	0
0	.5	.21	0	.67	.487	0	0	0
.5	.21	0	.67	.487	0	0	0	0

im2col(w1,w2)

0	2	-2
-1	4	24
1	3	68
0	5	16
18	23	42
22	46	81
32	7	20
60	35	78

im2col($\frac{\partial L}{\partial y_l}$).T × im2col(w1,w2)

30	18.339	41.6848
28.7678	11.3634	37.8224
6.722	1.476	4.336
51.0322	47.6112	98.3552
64.376	44.7626	99.7084
19.61	8.981	35.284
14.642	31.212	56.622
22.528	38.992	73.295
8.766	11.693	19.962

δy_{l-1}=rearrange(im2col($\frac{\partial L}{\partial y_l}$).T × im2col(w1,w2))

30	51.0322	14.642
28.7678	64.376	22.528
6.722	19.61	8.766

18.339	47.6112	31.212
11.3634	44.7626	38.992
1.476	8.981	11.693

41.6848	98.3552	56.622
37.8224	99.7084	73.295
4.336	35.284	19.962

Table 3.20: A vectorial example of backpropagation, assuming a input of 3×3×3, 2 kernels of 3×2×2 (cont.). Calculating $\delta w_{l-1} = \delta y_l * \text{input}$. But, input have dimensions 3x3x3 and δy_l have dimensions 2x2x2, so they don't match, it is telling us that we need to apply both channels of δy_l to every channel of the input.

im2col(input)

16	47	24	18
47	68	18	12
24	18	32	26
18	12	26	9
26	24	57	21
24	2	21	11
57	21	43	12
21	11	12	19
18	4	47	6
4	81	6	22
47	6	21	12
6	22	12	13

im2col(δy_l)

.1678	.5
.002	.21
.098	.67
.246	.487
.1678	.5
.002	.21
.098	.67
.246	.487
.1678	.5
.002	.21
.098	.67
.246	.487

im2col(input) × im2col(δy_l)

33.832	153.425
28.367	121.275
43.2764	162.12
22.627	85.417

δw_{l-1}=rearrange(im2col(input) × im2col(δy_l))

33.832	43.2764
28.367	22.627

153.425	162.12
121.275	85.417

3.2 ReLU

In CNNs the activation function most widely used is Rectified Linear Unit (ReLU), which have a good training speed and superior results compare to others (tanh Figure 3.5, sigmoid Figure 3.4).

3.2.1 Forward

Basically the function returns 0 when it receives a negative value and the value itself in any other case. Given x, the output of a ReLU is:

$$y = max(x, 0) \tag{3.15}$$

A plot of that:

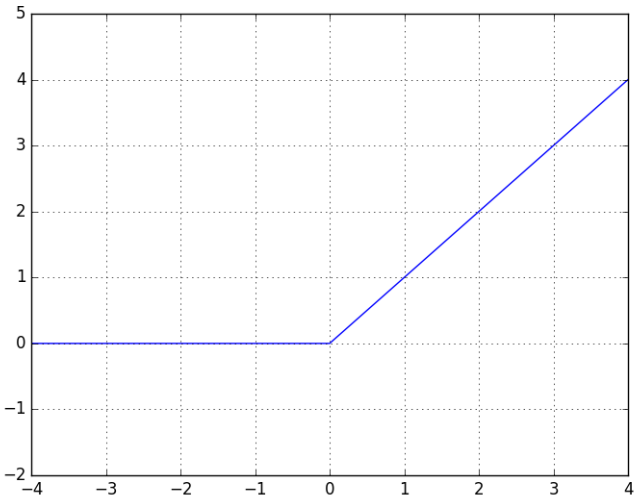

Figure 3.3: ReLU

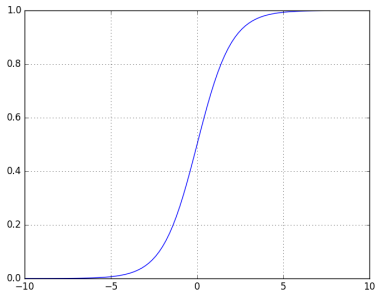

Figure 3.4: Sigmoid

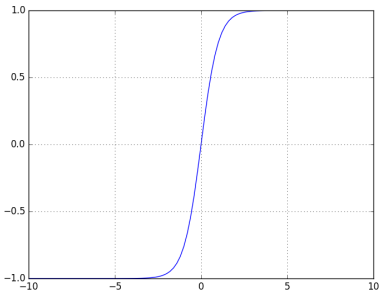

Figure 3.5: Tanh

As example, we assume that ReLU input is a 1-channel 4x5 image and the output is another 1-channel 4x5 image but without negative values.

Table 3.21: Feedforward ReLU

Table 3.22: Input to ReLU (x)

45	-12	16	90	34
-18	32	12	45	-8
-1	7	8	9	12
-2	-12	14	56	18

Table 3.23: Output of ReLU (y(x))

45	0	16	90	34
0	32	12	45	0
0	7	8	9	12
0	0	14	56	18

3.2.2 BACKPROPAGATION

The derivative of ReLU is the next:

$$y'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.16}$$

You can read an explanation of the derivative in the site [7]. Nevertheless, ReLU layers are connected to other layer so we can rewrite the derivative to make it easier to understand in error backpropagation according to [1].

$$\frac{\partial L}{\partial y_{l-1}} = \begin{cases} \frac{\partial L}{\partial y_l}, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.17}$$

Where $\frac{\partial L}{\partial y_{l-1}}$ is the error of the ReLU layer respect to the Loss of the network, $\frac{\partial L}{\partial y_l}$ is the back-propagated error from the subsequent layer and x is the original input to ReLU. As an example, using the same x as the forward example:

Table 3.24: Backpropagation ReLU

Table 3.25: $\frac{\partial L}{\partial y_l}$

.45	.75	.36	.42	.34
.32	.28	.21	.18	.47
.1	.17	.2	.32	.28
.1	.3	.36	.23	.12

Table 3.26: $\frac{\partial L}{\partial y_{l-1}}$

.45	0	.36	.42	.34
0	.28	.21	.18	0
0	.17	.2	.32	.28
0	0	.36	.23	.12

3.3 MAX-POOLING

"Is a form of non-linear down-sampling. Max-pooling partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum value"[11]. It can be seen similar to a convolution operation but instead of apply a filter to any sub-region we just take the max value, also we can use an average pooling (the mean of the subregion) or a L^p-pooling [3], but most architectures uses max-pooling so we focus on this one.

3.3.1 FEEDFORWARD

A math representation of this operation:

$$y_{kij} = \max_{(p,q)\in R_{ij}} x_{kpq} \tag{3.18}$$

"where y_{kij} is the output of the pooling operator related to the kth feature map, x_{kpq} is the element at (p,q) within the pooling region R_{ij} which represents a local neighborhood around the position (i,j)."[19]

As example Table 3.30, in this case we have 1-channel 4x4 image (if we have more channels we just apply the same operation for all), the output if we apply a 2x2 max pooling is an 2x2 image. If we use non-overlapping max-pooling the output will be of dimensions (Number of channels, height of image/height of pooling, width of image/width of pooling).

Table 3.27: Feedforward max-pooling 2x2

Table 3.28: Max-pooling input

23	4	16	90
12	32	12	45
5	7	8	9
2	12	14	56

Table 3.29: Max-pooling output

32	90
12	56

In the example every subregion was of size 2x2, it means that we take 4 elements and compare them to obtain the max value. First we have {23,4,12,32} so the max is 32, we iterate: max(16,90,12,45)=90, max(5,7,2,12)=12 and max(8,9,14,56)=56

3.3.2 BACKPROPAGATION

To backpropagate the error we need the derivative of the max-pooling operation according to [10].
Let be $g(x) = max(x)$:

$$\frac{\partial g}{\partial x_j} = \begin{cases} 1, & \text{if } x_j = max(x) \\ 0, & \text{otherwise} \end{cases} \tag{3.19}$$

So, we can generalize the derivative if we receive the error from an upper layer [1]:

$$\frac{\partial L}{\partial y_{l-1}}(x+p, y+q) = \begin{cases} 0, & \text{if } \left(y_l(x,y) != y_{l-1}(x+p, y+q)\right) \\ \frac{\partial L}{\partial y_l}, & \text{otherwise} \end{cases} \tag{3.20}$$

It's similar to ReLU backpropagation, in the positions that are the max value of the input we use the asociated value of $\frac{\partial L}{\partial y_l}$, for example considering the same input as Table 3.30

Table 3.30: Backpropagation max-pooling 2x2

Table 3.31: $\frac{\partial L}{\partial y_l}$

.22	.02
.48	.16

Table 3.32: $\frac{\partial L}{\partial y_{l-1}}$

0	0	0	.02
0	.22	0	0
0	0	0	0
0	.48	0	.16

If you have properly installed Theano you can just copy and run the next code:

```
1       import theano.tensor as T
2       import numpy as np
3       from theano.tensor.nnet.conv import conv2d
4       from theano.tensor.signal.downsample import max_pool_2d
#Inputs and weigths for convolution
5       a= np.random.randint(-100,100, size=[1,3,4,4])
6       print 'a',a
7       w= np.random.rand(2,3,2,2)
8       print 'w',w
#Convolution
9       l1 = conv2d(a.astype(float), w.astype(float))
10      print 'convolution',l1.eval()
#ReLU
11      l2 = T.maximum(0,l1)
12      print 'ReLU',l2.eval()
#Max pool
13      l3 = max_pool_2d(l2, (2,2))
14      print 'max pool',l3.eval()
```

You can have an output like the Table 3.33. As you can see we have an input a (line 5) of 4 dimensions, it is because Theano is oriented to use convolution in a mini-batch fashion, it means, that the first dimension represents the number of images per mini-batch, the second indicates how many channels every image have, the third and fourth are dimensions of every channel (width, height). In this case a is a single image of 3-channels (RGB) with size 4x4.

w is a matrix for the weights related to the convolution operation (kernels or filters) it have also 4 dimentions like the input, but in this case the first dimension indicates how many filters we have. As you can see in line 7 we have 2 filters of 3 channels, you need to be careful because channels of filter and input must be equals.

In line 9 we apply the convolution that Theano facilitates. It have some parameters but in this case we use the convolution just with input and kernel, it means a 'valid' convolution with no padding and stride=1. So the expected output should be of dimensions (1,2,3,3).

After the convolution, in lie 11 we apply the ReLU activation, and in line 13 we apply a max pooling with a window of size (2,2).

Table 3.33: Convolution->ReLU->Max pooling using Theano

a = input

56	-1	-7	-96
90	-57	-7	-63
20	-70	65	56
70	89	-21	62

-45	-14	-37	33
2	-54	-5	42
59	35	-29	0
-22	-38	7	33

-22	41	0	47
85	-9	21	46
38	-65	79	-36
-84	-45	34	46

w1

0.33802288	0.50899564
0.14104141	0.69301409

0.31737283	0.24737643
0.46818751	0.58417318

0.78298214	0.77431359
0.12114484	0.48877456

w2

0.29372102	0.61100424
0.12561727	0.0250721

0.86594409	0.7906012
0.8071996	0.94855461

0.72748357	0.99974341
0.62388789	0.94621146

output of convolution

68.70796	-43.9922	20.65064
61.42530	-78.90895	96.66672
13.4739	-31.87385	113.1576

23.55476	-82.88188	72.72308
87.53456	-81.61252	155.66603
-26.77457	7.96008	103.971

output of ReLU

68.70796	0	20.65064
61.42530	0	96.66672
13.4739	0	113.1576

23.55476	0	72.72308
87.53456	0	155.66603
0	7.96008	103.97100

output of max pool

68.70796	96.66672
13.4739	113.1576

87.53456	155.66603
7.96008	103.97100

4 PUT IT ALL TOGETHER

We are going to use a minimal example applying the concepts of the tutorial, with a CNN with the next layers: input -> convolution -> ReLU -> max pooling -> affine -> ReLU -> affine -> softmax. For the sake of simplicity, we are going to represent every layer with the output dimension, and for a first general overview of the model we are going to ommit the ReLU.

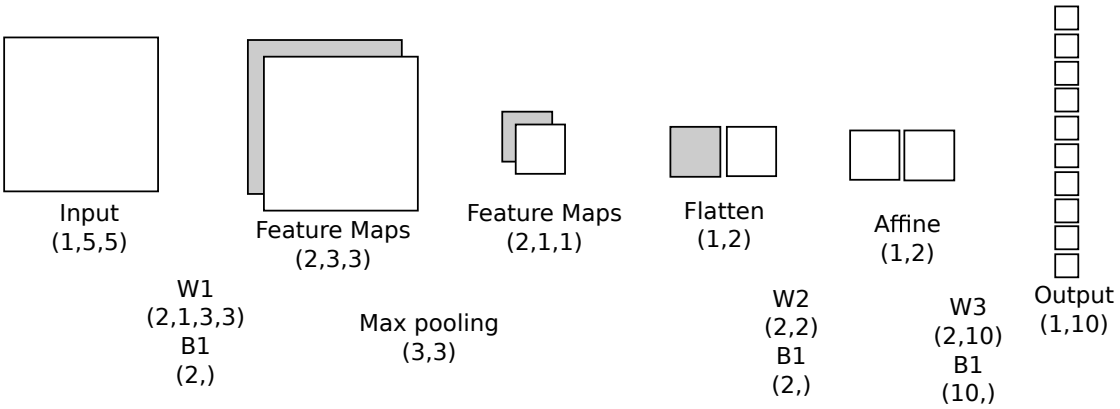

Figure 4.1: Toy Example of a CNN

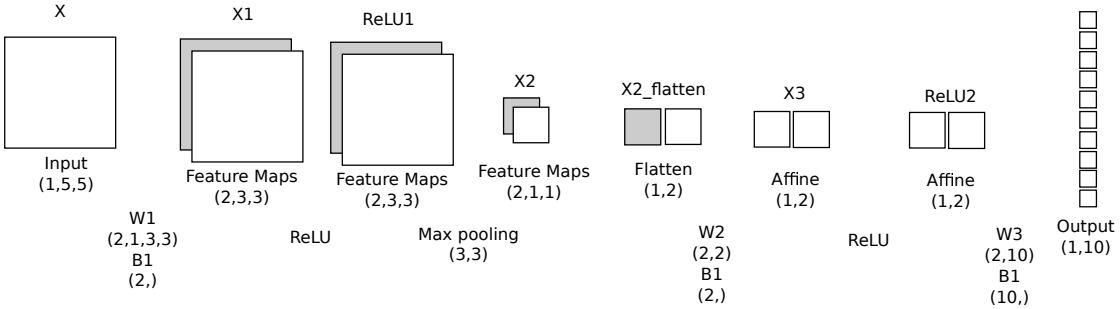

Figure 4.2: Detailing Model of the Toy Example of a CNN

Now we are going to apply the feedforward and the backpropagation using the model represented in Figure 4.2[2].

[2] this example was verified using part of the code of the class CS231n Convolutional Neural Networks for Visual Recognition

Table 4.1: X1= X∗W1 + B1

X

4	1	3	5	3
2	1	1	2	2
5	5	1	2	3
2	2	4	3	2
5	1	3	4	5

W1

1	2	3
4	7	5
3	-32	25

12	18	12
18	-74	45
-92	45	-18

B1

0
0

X1

-85	76	64
109	-1	10
118	71	67

-144	-291	66
-347	102	-192
-239	-52	-162

Table 4.2: ReLU1 = ReLU(X1)

X1

-85	76	64
109	-1	10
118	71	67

-144	-291	66
-347	102	-192
-239	-52	-162

ReLU1

0	76	64
109	0	10
118	71	67

0	0	66
0	102	0
0	0	0

Table 4.3: X2 = 3x3 max pooling (ReLU1)

ReLU1

0	76	64
109	0	10
118	71	67

0	0	66
0	102	0
0	0	0

X2

118

102

Table 4.4: ReLU2 = ReLU(X3) where X3 = X2_flatten × W2 + B2

X2_flatten

118	102

W2

1	2
3	-4

B2

0
0

X3

424	-172

ReLU2

424	0

Table 4.5: Output = softmax(x4) where x4= ReLU2 × W3 + B3

ReLU2.T

424
0

W3.T

0.09	0.02
0.08	0.03
0.07	0.03
0.06	0.02
0.05	0.01
0.04	0.02
0.03	0.07
0.04	0.08
0.05	0.05
0.01	0.01

B3.T

0
0
0
0
0
0
0
0
0
0

X4.T

38.16
33.91
29.68
25.43
21.20
16.95
12.71
16.95
21.2
4.24

Output.T

9.855923e-01
1.419998e-02
2.045877e-04
2.947614e-06
4.246805e-08
6.118620e-10
8.815459e-12
6.118620e-10
4.246805e-08
1.829901e-15

Table 4.6: Calculate Loss (the cost) using Negative Log-Likelihood, assuming that correct label was 9 (0-based index). $Loss = -\log(1.829901e-15) = 33.9345$
Also to calculate de derivative of the error we subtract in the correspondent output position the expected value (we subtract 1 to the position 9 of the output)

δoutput.T

9.855923e-01
1.419998e-02
2.045877e-04
2.947614e-06
4.246805e-08
6.118620e-10
8.815459e-12
6.118620e-10
4.246805e-08
-1

δW3.T (ReLU2.T × δoutput).T

4.178911e+02	0
6.020791e+00	0
8.674518e-02	0
1.249788e-03	0
1.800645e-05	0
2.594295e-07	0
3.737754e-09	0
2.594295e-07	0
1.800645e-05	0
-4.24e+02	0

δB3.T (δoutput.T)

9.855923e-01
1.419998e-02
2.045877e-04
2.947614e-06
4.246805e-08
6.118620e-10
8.815459e-12
6.118620e-10
4.246805e-08
-1

δReLU2.T (δoutput × W3.T).T

0.07985382
0.01014405

Table 4.7: Backpropagating error

δX3.T, see Equation 3.17

0.07985382
0

δW2.T (X2_flatten.T × δX3).T

9.42275079	0
8.145089	0

δB2.T (δX3.T)

0.07985382
0

δX2_flatten.T (δX3 × W2.T).T

0.07985382
0.23956146

Table 4.8: Backpropagating error (cont.) δX2 is equal to δX2_flatten.T because of rearranging of dimensions. The error that receives ReLU1 from X2 need to apply the backpropagation formula of max pooling, see Equation 3.20. Also, δX1 in this case is equal to δReLU1 because of Equation 3.17

δReLU1, δX1

0	0	0
0	0	0
0.07985382	0	0

0	0	0
0	0.23956146	0
0	0	0

Table 4.9: Final step, backpropagating into X

δB1
sum of the error for
channel in δX1

0.07985382	0.23956146

δW1
(δX1 $*$ X), valid convolution

0.3992691	0.3992691	0.07985382
0.15970764	0.15970764	0.31941528
0.3992691	0.07985382	0.23956146

0.23956146	0.23956146	0.47912292
1.1978073	0.23956146	0.47912292
0.47912292	0.95824584	0.71868438

δX
(W1 $*$ δX1), full convolution

0	0	0	0	0
0	2.87473753	4.31210629	2.87473753	0
0.07985382	4.47181394	-17.48798664	10.78026574	0
0.31941528	-21.48067765	11.17953484	-4.31210629	0
0.23956146	-2.55532225	1.99634551	0	0

After that toy example, maybe you want to run code on some real dataset and see the results. The code is based on http://www.dbs.ifi.lmu.de/Lehre/MaschLernen/SS2016/Uebung/u05/Exercise5-2.html with a similar model than Figure 4.2. We are going to use MNIST [13] with 60,000 images of ten classes for training, and 10,000 images for testing. You can find the code in https://github.com/lepe92/examples_cnn/blob/master/ejemplo_mnist_theano.py, you can see the model in Figure 4.3, if you run it you can get an output like this:

```
epoch 0 | loss 1.5646   | accuracy: 0.89800
epoch 1 | loss 0.780637 | accuracy: 0.93850
epoch 2 | loss 0.472072 | accuracy: 0.95190
epoch 3 | loss 0.328828 | accuracy: 0.96360
epoch 4 | loss 0.25307  | accuracy: 0.96820
epoch 5 | loss 0.207475 | accuracy: 0.97180
epoch 6 | loss 0.177256 | accuracy: 0.97370
epoch 7 | loss 0.155574 | accuracy: 0.97610
epoch 8 | loss 0.13908  | accuracy: 0.97830
epoch 9 | loss 0.126003 | accuracy: 0.97870
```

You can observe that in just 10 epochs we obtain 97.87% of accuracy. In Figure 4.4 and Figure 4.5 you can observe the change of the accuracy and the cost (respectively) with every epoch.

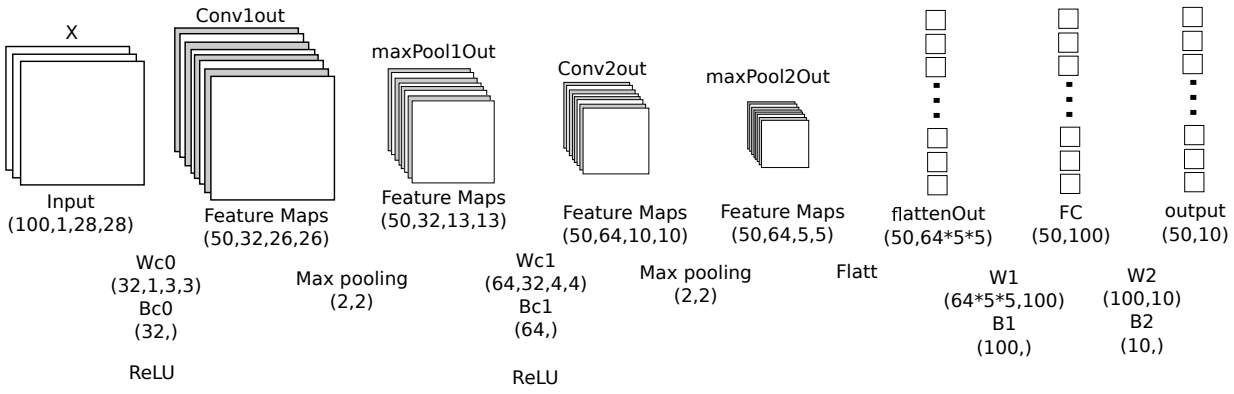

Figure 4.3: CNN model for MNIST

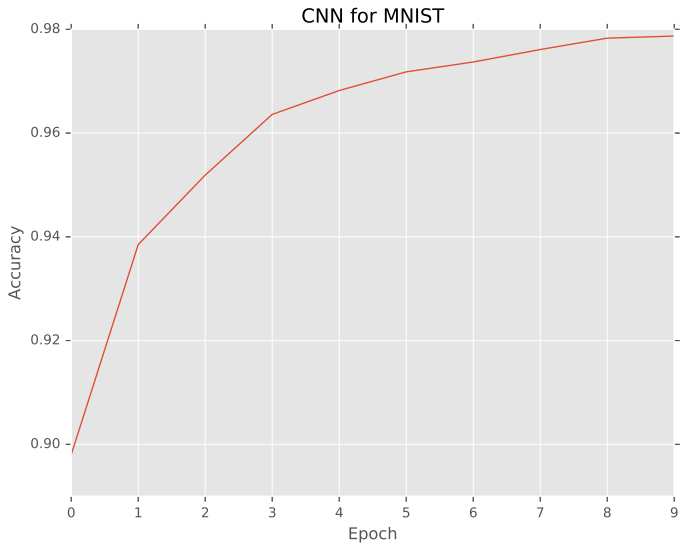

Figure 4.4: Change of accuracy respect to epochs

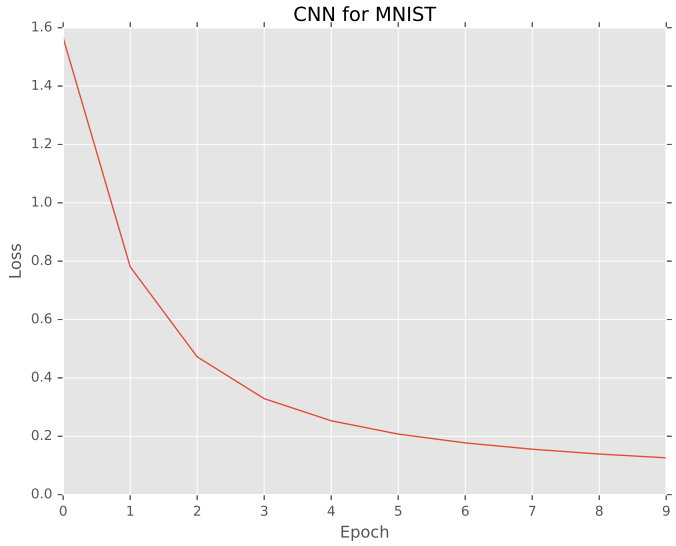

Figure 4.5: Change of loss respect to epochs

REFERENCES

[1] boris.ginzburg@intel.com. Lecture 3:cnn: Back-propagation. Slides. Available at http://courses.cs.tau.ac.il/Caffe_workshop/Bootcamp/pdf_lectures/Lecture%203%20CNN%20-%20backpropagation.pdf.

[2] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[3] Caglar Gulcehre, Kyunghyun Cho, Razvan Pascanu, and Yoshua Bengio. Learned-norm pooling for deep feedforward and recurrent neural networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 530–546. Springer, 2014.

[4] Grzegorz Gwardys. Convolutional neural networks backpropagation: from intuition to derivation, apr 2016. Available at https://grzegorzgwardys.wordpress.com/2016/04/22/8/.

[5] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[6] Aggelos K. Katsaggelos. Fundamentals of digital image and video processing. Video lesson "2D convolution", Available at https://www.coursera.org/learn/digital/lecture/xHtUV/2d-convolution.

[7] J Kawahara. What is the derivative of relu?, may 2016. Available at http://es.slideshare.net/kuwajima/cnnbp.

[8] Jayanth Koushik. Understanding convolutional neural networks. *arXiv preprint arXiv:1605.09081*, 2016.

[9] C-C Jay Kuo. Understanding convolutional neural networks with a mathematical model. *arXiv preprint arXiv:1609.04112*, 2016.

[10] Hiroshi Kuwajima. Memo: Backpropagation in convolutional neural network, aug 2014. Available at http://kawahara.ca/what-is-the-derivative-of-relu/.

[11] LISA lab. Convolutional neural networks (lenet). Documentation. Available at http://deeplearning.net/tutorial/lenet.html.

[12] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[13] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[14] Sarah Price. Convolution in two dimensions, jul 1996. Available at http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MARBLE/low/space/convol.htm.

[15] Jimmy SJ Ren and Li Xu. On vectorization of deep convolutional neural networks for vision tasks. *arXiv preprint arXiv:1501.07338*, 2015.

[16] David Stutz. Understanding convolutional neural networks. 2014.

[17] Eric W Weisstein. Cross-correlation theorem. *From MathWorld A Wolfram Web Resource. http://mathworld. wolfram. com/Cross-CorrelationTheorem. htm*, 2009.

[18] Jianxin Wu. Introduction to convolutional neural networks, apr 2016. Available at http://cs.nju.edu.cn/wujx/paper/CNN.pdf.

[19] Dingjun Yu, Hanli Wang, Peiqiu Chen, and Zhihua Wei. Mixed pooling for convolutional neural networks. In *International Conference on Rough Sets and Knowledge Technology*, pages 364–375. Springer, 2014.

[20] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision*, pages 818–833. Springer, 2014.