MNIST from absolute scratch in C

Emperor Nintri (Dimitri Condoris)

February 4, 2025

1 Introduction

...

2 Functions

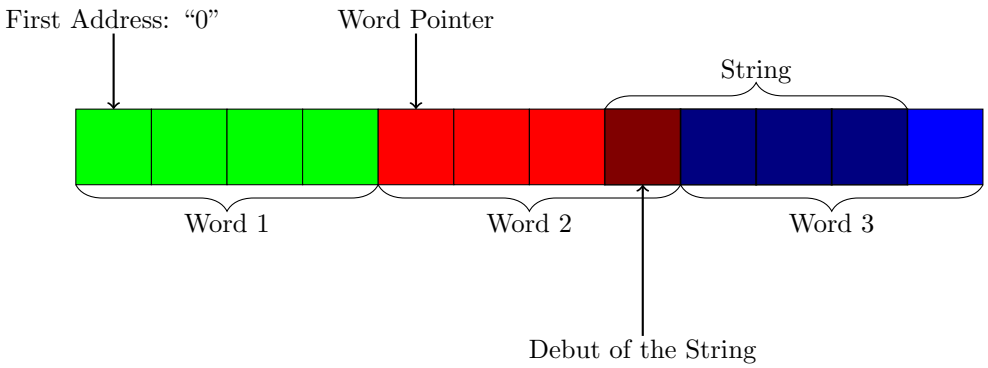
In this section, we explain every trick and algorithm used to code the basic functions needed for this project and others.

2.1 String Length Function

In this subsection we detail the algorithm used in order to compute the length of a string.

2.1.1 Core Idea

Based on some implementation I found, the goal is here to optimize the computation of the string length by checking one (aligned) word at a time for a stopping character (`\0`). The first step is to store the address of the string as an int to perform some bits manipulation. Then we extract the nearby word, in memory, we will have something like this:



Here, each square corresponds to a byte. We therefore apply a first function that will identify all the null bytes in a word. This function uses a mask which is the repetition of the byte:

$$01111111$$

To apply this mask we use the following instruction:

$\sim$ `(((word & mask) + mask) | word | mask)`.

To see what this does. Consider that we only have one byte and that this byte is null, then we have as a result of the previous formula $\sim$ `mask` which is equal to 10000000. Now let word be $b_1b_2b_3b_4b_5b_6b_7b_8$ with at least one b_i equals to 1, then `word & mask` will be equal to $0b_2b_3b_4b_5b_6b_7b_8$. Because there is at least one b_i (not b_0) equal to 1, it is very easy to show that `(word & mask) + mask` will be equal to 1??????? and hence the result will be a 0 byte (inverse of 1??????? | 01111111 = 11111111). Now if only b_0 is equal to 1 then `word | mask` will be equal to 1 and hence we would have the same result as before.

From the previous computation we can deduce that applying the instruction will simply mark as 1 all 0 bytes (on the first bit) and 0 all other bytes. Now because we are not interested in all the bytes of the word we shift it from as many characters as needed to land on the first character of the string. In our drawing for example, only the fourth word belongs to the string so we will shift the word by 3 bytes and verify if the result is 0 or none. The shifted byte belonging to the string, if it is 0 it means that the string is already over, else we have to check the next string bytes (by checking the next word). No problem here if the string is smaller than the word because if it ends it ends with the first 0.

2.1.2 Finding the Zeroes

Now that we have the bytes of interest correctly formatted, we want to verify if there is a zero and if yes at which byte. To do so, we are going to attempt to find the first byte, if we do not find it then

we will return 0 and we will know that no end character is belonging to the current word. So pretty much let's imagine we have the following bytes:

$$0000000\ 10110101\ 00000000\ 01010111$$

Before we wanted to find all the zeros, now we only want to find the first one (because the next ones could be belonging to another element). So here the result we want to have is the following (we read from the right to the left):

$$0000000\ 00000000\ 10000000\ 00000000$$

For that we wil use specific bytes:

$$\texttt{lsb} = 0000001\ 00000001\ 00000001\ 00000001$$
$$\texttt{msb} = 1000000\ 10000000\ 10000000\ 10000000$$

Here we will use the following formula:

`(word - lsb) & ~ word & msb`.

We have `- lsb` = 11111110 11111110 11111110 11111111. It is pretty straight forward that if we add `word` to `- lsb` the first zero byte will be full of one except maybe the first bit (if the zero byte is not the first one) and hence, because ones will be carried, combining with the opposite of the word assures that only the zeros that did not get cleared are kept and that the additional ones are canceled out. Finally combining with `msb` keeps only the first zero byte, other bytes being kind of random. Now the only step missing is to compute the number of trailing zeros to know which byte is zero and hence what size is the string.

Let's give a concrete proof. Arbitrarily let's say that the first byte is zero, then the first byte of `word - lsb` will be full of ones and combining this with the opposite of `word` will also be full of ones (because the opposite of zero is one and one and one is one). Finally combining the result with the `msb` will result in only the eighth bit being 1 and the others zero so we correctly identify the zero byte. Now if the first zero byte is the second, third or fourth one then there exists at least one bit which is one in the previous bytes. Because of that $b_8b_7b_6b_5b_4b_3b_2b_1 + 11111111$ will carry exactly one bit to the next byte by passing by the last bit b_8 so this bit will 1 if by default it was 1 or else it will be 0. If it was 1 by default then combining with the inverse of `word` will yield 0 and hence will not identify a zero byte. If it was 0 then it is 0 already. Now, because we did carry a one (potentially if the next byte is not 0) we will have 11111111 to which we will add at least one one if it does not correspond to the zero byte so the previous logic apply again zeroing the last bit of this byte. Finally, only for the zero byte will the last bit be one because of the carry not being propagated anymore.

2.1.3 Counting Trailing Zeros

As we saw in the last sub-subsection, the only remaining function we have to implement is one that, given a binary representation, outputs the number of trailing zeros. The trailing zeros are the bits of value zero before the first bit of value one. To give a concrete example, consider the following binary representation:

$$1100101010111010101010000111010101100110101100101101010001\color{red}{000000}$$

Then the red zeros represent the trailing zeros and hence here we have 6 trailing zeros. Formally speaking we can define this quantity as:

$$T(x) = \max\{k \in \mathbb{N} \mid 2^k \text{ divides } x\}.$$

Here we are only interest in binary representation of size 64. The first observation we can make is that we do not need all the information of x. Mainly, what we do not need is the information regarding all the ones at the left of the first one from the right. To get rid of this information we can apply the formula:

```
long y = (x & -x);
```

Formally, consider that x is equal to $b_{63} \dots b_{T(x)} 0 \dots 0$ with $0 \leq T(x) \leq 63$ and by definition $b_{T(x)} = 1$. Then we have $-x = \text{not}(x) + 1$ which gives us:

$$\begin{aligned} -x &= (1 - b_{63}) \dots (1 - b_{T(x)}) 1 \dots 1 + 0 \dots 00 \dots 1 \\ &= (1 - b_{63}) \dots b_{T(x)} 0 \dots 0. \end{aligned}$$

because the first 1 will carry to the first 0 which for $\text{not}(x)$ will have the same position as $T(x)$ for x and hence for $-x$ this position will have value 1. The positions before (from the right) 0 and the ones after will have the same values as $\text{not}(x)$ hence the result that:

$$y = 0 \dots b_{T(x)} 0 \dots 0 \text{ with } b_{T(x)} = 1.$$

So now how do we extract the position of this value? Well, we are going to use a property of a *de Bruijn sequence*. A *de Bruijn sequence* in our case will be a cyclic sequence of bits for which whatever the subsequence you take it will be unique. Such a sequence is denoted by $B(k, n)$ where k is the size of the vocabulary we treat (in our case 2 because we only have 0 and 1) and n is the size of the sub-string we want to be unique in the string. Let's say we fix n to 3, then the unique combinations we can make with 0 and 1 of size 3 will be:

$$000, 001, 010, 011, 100, 101, 110 \text{ and } 111$$

One (because there are many different ones) of the $B(2, 3)$ is:

$$00010111$$

Because as you can see the first 3 characters are 000 then the next 3 are 001 etc. We find every combination possible and only one time. If you wonder where is 110, it is composed of the last 2 characters and then the first one because we can cycle through the sequence.
To sum it up really easily, whatever the 3 successive characters that you extract from this sequence; it will be unique. The size of all $B(k, n)$ is k^n. Here we have potentially 64 different values for $T(x)$ and because k is 2, we can use $n = 6$.

We construct one example following a simple algorithm where we try to complete every combination possible:

000000000000000000000000000000000000000000000000000000000000000000
000000100000000000000000000000000000000000000000000000000000000000
000000110000000000000000000000000000000000000000000000000000000000
000001111000000000000000000000000000000000000000000000000000000000

Pretty much we just have to slide adding 1 or 0 and doing all the 64 possible combinations.

Of course doing so randomly will not yield a working result, instead we follow the *prefer-largest greedy construction* which consists of starting with $0^{n-1} = 00000$ and add the largest symbol (here 1) if it does not repeat a previous substring of size 6 and else add 0. Once you reach 64 additions you get rid of the first 5 zeros:

000001
000011
000111
001111
011111
111111
111110
111101

I am not going to write everything but the result we get at the end is:

1111110111100111010111000110110100110010110000101010001001000000

Or in reverse (with the least significant byte on the right):

0000001001000101010000110100110010110110001110101110011110111111

Now the interesting question come into play, all that for what? Well, if you multiply this binary number with the representation of x with only one significant byte, then you'll get as the first 6 bits a unique sequence of bits (because every sub-string is unique). For example, if the significant byte is at the first position then the first 6 bits of the multiplication will yield 000000. If it is the second byte it will yield 000001, etc. Because of that we can pre-compute a table that will associate the correct integer to the correct $T(x)$, for example if $T(x)$ equals 0 then the first 6 bits of the multiplication will be 000000 which is 0 in base 2 so in the table at position 0 we will put the result 0.

You can compute the table yourself if you want but here are the coefficients:

0,	1,	2,	18,	3,	10,	19,	39,
7,	4,	11,	30,	26,	20,	40,	51,
16,	8,	5,	24,	14,	12,	31,	45,
36,	27,	21,	33,	47,	41,	52,	57,
63,	17,	9,	38,	6,	29,	25,	50,
15,	23,	13,	44,	35,	32,	46,	56,
62,	37,	28,	49,	22,	43,	34,	55,
61,	48,	42,	54,	60,	53,	59,	58,

So finally to get the number of trailing zeros you simply have to look at this table at the index returned by the previous operation. We have everything we need to build the function returning the length of a string!

2.2 Print Function

In this subsection we focus on the implementation of the print function.

2.2.1 Printing a String

To print a string we are going to make a system call. Here in C we can do this simply by making the call in assembly directly. To do this we only need the pointer of the first character of the string as well as it's length (hence the previous function). The call then looks like this:

```
size_type string_length = getStringLength (string);
asm volatile
(
"movq $1, %%rax\n"
"movq $1, %%rdi\n"
"movq %0, %%rsi\n"
"movq %1, %%rdx\n"
"syscall       \n"
:
: "r"(string), "r"(string_length)
: "rax", "rdi", "rsi", "rdx"
);
```

The first `$1` means that we want to write to the second `$1` which corresponds to the standard output. We then pass as a first parameter `%0` a pointer to our string and as a second parameter it's length `%1`.

2.2.2 Printing an Integer

To print an integer we are going to apply a very basic algorithm. First of all we are going to successively divide our number by 10 until it reaches 0 to know how many digits we have in our number. Once we know that we will get the rest of the initial number from the division by 10, this will be our first digit, divide the number by 10 again and then rinse and repeat until we reach the end of our integer again.

To give you a concrete example, here is what we are going to do:

```
int x = 10005;
int rest = x % 0; // rest = 5
x /= 10; // x = 1000
rest = x % 0; // rest = 0
x /= 10; // x = 100
rest = x % 0; // rest = 0
x /= 10; // x = 10
rest = x % 0; // rest = 0
x /= 10; // x = 1
rest = x % 0; // rest = 1
```

And hence the method we use to construct the decimal representation.

2.2.3 Printing a Float

The idea for extracting each digit of the number is to first separate it into the integer part and the fractional part. Then, we divide the integer part by 10, floor the result, multiply it by 10, and subtract the product from the original integer part. This will give us the first digit. Before moving to the fractional part handling, we will focus on this first step.

Here is an example of what we want to do:

```
double x = 10005.0505;
unsigned long integer_part = truncate(x); // integer_part = 10005.000000
printUnsignedInteger(integer_part);
```

Here because we use unsigned long to store the integer part it will not be possible to represent numbers that are above 2^{64}. Is this dramatic? No because we do not really care in the scope of this project. If we wanted to print every single digits for integer parts which are bigger than 2^{64} we could use a multitude of techniques but I would implement arbitrary-precision arithmetic as I don't care about speed here. Now how are we going to do the truncating? Let's first recall what is the binary form of a double (of size 64 bits) on a computer:

$$b_{63}b_{62:52}b_{51:0}$$

Here, for $i \in \{0, \dots, 63\}$, b_i represents a bit for which the value is either 0 or 1. If $\{b_i, i \in \{0, \dots, 63\}\}$ represents x then to convert x back to decimal format you apply this formula:

$$\begin{aligned} x &= (-1)^{b_{63}} \times 2^{\sum_{i=0}^{10} b_{52+i} \times 2^i - 1023} \times \left(1 + \left(\sum_{i=1}^{52} b_{52-i} \times 2^{-i}\right)\right) \\ &= (-1)^{b_{63}} \times 2^{e-1023} \times (1+m), \end{aligned}$$

$$\text{with } e := \left(\sum_{i=0}^{10} b_{52+i} \times 2^i\right) \text{ and } m := \left(\sum_{i=1}^{52} b_{52-i} \times 2^{-i}\right).$$

For the moment, let's not consider special cases such as subnormal numbers, zeros and infinities.

What we can remark is that decimal places are entirely determined by the product $2^{e-1023} \times m$, moreover:

$$2^{e-1023} \times m = \left(\sum_{i=1}^{52} b_{52-i} \times 2^{e-i-1023}\right)$$

More precisely, decimal places are determined by every coefficients b_{52-i} such that $e - i - 1023$ is less than 0 (or $i > e - 1023$). From that we can conclude that to truncate the integer we can simply mask all the bits $\{b_i, i < 1075 - e\}$.

For the fractional part, we are going to truncate the other way by keeping only the bits for which $i \geq 1075 - e$. Then we will multiply this value by 10000, round back to an integer (the difference between the rounding and the truncatating will be that in the rounding we will verify if the 2^{-1} bit is 1 or not) and output the correspond characters. We limit our scope here to a precision of 10^{-5} (which totally destroy subnormal numbers) because again in our context, more is not really necessary. I did some particular optimization to make sure that the rounding is correct when ever possible. If you want details regarding that you can check in the code or ask me directly.

Some thought: This subject is a particularly complex topic if you want to have a full precise display of your double, if you want particular details regarding that you can take a look at different ideas such as Grisu (from Florian Loitsch), Grisu-Exact (from Junekey Jeon) and my particular favorite: Dragonbox (also from Junekey Jeon).

2.3 Mathematical Functions

In this subsection we focus on the different algorithms used to compute mathematical functions.

2.3.1 Square Root

We do not really need the square root in this project but it may be useful in further expansion so I decided to still implement it. We are going to use the *Henron's method* which takes a first estimate of the root square of x, x_0 and iterates on it:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{x}{x_i}\right).$$

Let's say this sequence converges (easy to prove), then the limit respects the equality $l = (l + x/l)/2$ which gives you $l = x^2$. Now which initial estimate are we going to pick? Well we are going to follow some recommendation found on Wikipedia:

If $x = a \times 2^{2n}$ with $0.5 \leq a \leq 2$ and $n \in \mathbb{Z}$ then $x_0 = (0.5 + 0.5 \cdot a) \cdot 2^n$. We start by considering that x is a positive (negative numbers and ±Infinity/NaN are easy to handle) normal number with exponent $e - 1023$, with $1 \leq e \leq 2046$ an integer and $0 \leq m \leq 1$ such that:

$$x = (1 + m) \cdot 2^{e-1023}.$$

Now if e is odd then there exists m an integer such that $e = 2r + 1$ and hence:

$$x = (1 + m) \cdot 2^{2r-1022} = (1 + m)2^{2(r-511)}.$$

And because we have:

$$0.5 < 1 \leq (1 + m) \leq 2.$$

We can define $a = (1 + m)$ and $n = r - 511$ which leads to:

$$x_0 = (0.5 + 0.5 \cdot (1 + m)) \cdot 2^{r-511} = (1 + 0.5 \cdot m) \cdot 2^{(r+512)-1023}.$$

This is very easy to apply using the double-precision floating-point format. Now let's treat the case where $e = 2r$ then:

$$x = (1 + m) \cdot 2^{2r-1023} = (0.5 + 0.5 \cdot m) \cdot 2^{2(r-511)}.$$

And because we have:

$$0.5 \leq 0.5 + 0.5 \cdot m \leq 1 < 2.$$

We can define $a = (0.5 + 0.5 \cdot m)$ and $n = r - 511$ which leads to:

$$\begin{aligned} x_0 &= (0.5 + 0.5 \cdot (0.5 + 0.5 \cdot m)) \cdot 2^{r-511} = (0.75 + 0.25 \cdot m) \cdot 2^{(r+512)-1023} \\ &= (1.5 + 0.5 \cdot m) \cdot 2^{(r+511)-1023} \\ &= (1 + (0.5 + 0.5 \cdot m)) \cdot 2^{(r+511)-1023}. \end{aligned}$$

Which is again very easy to compute.

Finally let's say x is subnormal then $x = m \cdot 2^{-1023}$ with $0 \leq m \leq 1$. Then $m \neq 0$ so there must be a first bit for which $b_{52-l} = 1$ so that:

$$m = \sum_{i=l}^{52} b_{52-i} \cdot 2^{-i} \text{ with } b_{52-l} = 1.$$

To normalize m back to a correct interval. If $l = 2r + 1$ then we are going to multiply by 2^l which gives:

$$x = (2^l \cdot m) \cdot 2^{-1023-l} = (2^l \cdot m) \cdot 2^{-1022-2r} = (2^l \cdot m) \cdot 2^{2(-(511+r))}.$$

And because we have:

$$1 \leq 2^l \cdot m \leq 2.$$

We can define $a = 2^l \cdot m$ and $n = -511 - r$ and hence:

$$x_o = (0.5 + 2^{l-1} \cdot m) \cdot 2^{-511-r} = (1 + \sum_{i=l+1}^{52} b_{52-i} \cdot 2^{l-1-i}) \cdot 2^{(512-r)-1023}.$$

Now bear with me for the last case. If $l = 2r$ then we multiply by 2^{l-1} which gives:

$$x = (2^{l-1} \cdot m) \cdot 2^{-1023-l+1} = (2^{l-1} \cdot m) \cdot 2^{-1022-2r} = (2^{l-1} \cdot m) \cdot 2^{2(-(511+r))}.$$

And because we have:

$$0.5 \leq 2^{l-1} \cdot m \leq 1.$$

We can define $a = 2^{l-1} \cdot m$ and $n = -511 - r$ and finally:

$$x_0 = (0.5 + 2^{l-2} \cdot m) \cdot 2^{-511-r} = (1 + 2^{l-1} \cdot m) \cdot 2^{-512-r} = (1 + 2^{l-1} \cdot m) \cdot 2^{(511-r)-1023}.$$

Which is easy to compute. We defined for every scenario possible the exact value we should use for the initialization.

To get l we count the leading zeros: by repeating the most significant bit, if the result is full of 1, we return 0 else we add 1, shift the bits to the right for 1 place and apply the *Counting Trailing Zeros* that we substract from 63.

3 Layers

In this section we detail the layers' structure and results for the forward pass and backpropagation. Here we deal with a classification using cross-entropy as a loss function.

3.1 Output Layer

In this subsection we derive the appropriate formulas for a dense layer.

3.1.1 Notations

First of all, we define the notations used in this subsection:

L	loss function
C	number of classes
O	last layer
n	number of samples
$x_{(O)}$	number of features of layer O output (or loss input)
$X_{(O)}$	layer O activated output of size $(n, x_{(O)})$
Y	true one-hot-encoded labels of size (n, C)
$\hat{Y}$	predicted one-hot-encoded labels of size (n, C)

By definition, we have:

$$L := -\sum_{k=1}^{N}\sum_{x=1}^{x_{(O)}} Y(k,x)\log\Big(\hat{Y}(k,x)\Big) \text{ and } \hat{Y}(k,x) := \frac{\exp X_{(O)}(k,x)}{\sum_{y=1}^{x_{(O)}} \exp X_{(O)}(k,y)}.$$

3.1.2 First gradients

Now we can compute the first gradients:

$$\frac{\partial L}{\partial X_{(O)}(k,u)} \text{ with } 1 \leq k \leq n \text{ and } 1 \leq u \leq x_{(O)}.$$

We have:

$$\frac{\partial L}{\partial X_{(O)}(k,u)} = -\sum_{x=1}^{x_{(O)}} Y(k,x)\frac{\partial \hat{Y}(k,x)}{\partial X_{(O)}(k,u)}\frac{1}{\hat{Y}(k,x)}.$$

So considering that for $1 \leq x \leq x_{(O)}$:

$$\frac{\partial \hat{Y}(k,x)}{\partial X_{(O)}(k,u)} = \frac{\exp X_{(O)}(k,x)\Big(\sum_{y=1}^{x_{(O)}} \exp X_{(O)}(k,y)\Big)1_{x=u} - \exp X_{(O)}(k,x)\exp X_{(O)}(k,u)}{\Big(\sum_{y=1}^{x_{(O)}} \exp X_{(O)}(k,y)\Big)^2}.$$

We get:

$$\frac{\partial L}{\partial X_{(O)}(k,u)} = -\sum_{x=1}^{x_{(O)}} Y(k,x)(1_{x=u} - \hat{Y}(k,u)).$$

Now keep in mind that there exists only one $1 \leq z \leq x_{(O)}$ such that $Y(k,z) = 1$, the rest of the values

being 0. So we have two cases, if $z = u$ then we have:

$$\begin{aligned}\frac{\partial L}{\partial X_{(O)}(k,u)} &= -(1 - \hat{Y}(k,u)) \\ &= \hat{Y}(k,u) - 1 \\ &= \hat{Y}(k,u) - Y(k,u).\end{aligned}$$

Because $Y(k,u) = Y(k,z) = 1$ in this case. Now if $z \neq u$ then we have:

$$\begin{aligned}\frac{\partial L}{\partial X_{(O)}(k,u)} &= -(-\hat{Y}(k,u)) \\ &= \hat{Y}(k,u) \\ &= \hat{Y}(k,u) - Y(k,u).\end{aligned}$$

Because $Y(k,u) = 1 - Y(k,z) = 0$ in this case. We can sum those two results in one stating that:

$$\delta_O(k,x) := \frac{\partial L}{\partial X_{(O)}(k,x)} = \hat{Y}(k,x) - Y(k,x), \text{ for } 1 \leq x \leq x_{(O)}.$$

3.2 Dense Layer

In this subsection we derive the appropriate formulas for a dense layer.

3.2.1 Notations

First of all, we define the notations used in this subsection:

l	current layer
$l-1$	previous layer
n	number of samples
$x_{(l-1)}$	number of features of layer l input (or layer $l-1$ output)
$X_{(l-1)}$	layer l input (or layer $l-1$ output) of size $(n, x_{(l-1)})$
$x_{(l)}$	number of features of layer l output
$W_{(l)}$	layer l weight of size $(x_{(l)}, x_{(l-1)})$
$b_{(l)}$	layer l bias of size $x_{(l)}$
$f_{(l)}$	layer l activation function
$Z_{(l)}$	layer l unactivated output of size $(n, x_{(l)})$
$X_{(l)}$	layer l activated output of size $(n, x_{(l)})$

By definition, we have:

$$Z_{(l)} := W_{(l)} X_{(l-1)}^T + b_{(l)} \text{ and } X_{(l)} := f_{(l)}(Z_{(l)}).$$

3.2.2 Derivatives of Weight and Bias

Let's compute the derivative of the loss relatively to the weight of layer l. For that, we assume that we already know for every $1 \leq k \leq n$ and $1 \leq x \leq x_{(l)}$ the value of:

$$\delta_{(l)}(k,x) := \frac{\partial L}{\partial X_{(l)}(k,x)}.$$

Let $1 \leq u \leq x_{(l)}$ and $1 \leq v \leq x_{(l-1)}$. We have:

$$\begin{aligned}\frac{\partial L}{\partial W_{(l)}(u,v)} &= \sum_{k=1}^{n}\sum_{x=1}^{x_{(l)}} \frac{\partial L}{\partial X_{(l)}(k,x)} \frac{\partial X_{(l)}(k,x)}{\partial W_{(l)}(u,v)} \\ &= \sum_{k=1}^{n}\sum_{x=1}^{x_{(l)}} \delta_{(l)}(k,x) \frac{\partial X_{(l)}(k,x)}{\partial W_{(l)}(u,v)}.\end{aligned}$$

We only have to compute the second term, let $1 \leq k \leq n$ and $1 \leq x \leq x_{(l)}$:

$$\begin{aligned}\frac{\partial X_{(l)}(k,x)}{\partial W_{(l)}(u,v)} &= \frac{\partial X_{(l)}(k,x)}{\partial Z_{(l)}(k,x)} \frac{\partial Z_{(l)}(k,x)}{\partial W_{(l)}(u,v)} \\ &= f'_{(l)}(Z_{(l)}(k,x)) \frac{\partial Z_{(l)}(k,x)}{\partial W_{(l)}(u,v)} \\ &= f'_{(l)}(Z_{(l)}(k,x)) X_{(l-1)}(k,v) 1_{x=u}.\end{aligned}$$

This simplifies to:

$$\frac{\partial L}{\partial W_{(l)}(u,v)} = \sum_{k=1}^{n} \delta_{(l)}(k,u) f'_{(l)}(Z_{(l)}(k,u)) X_{(l-1)}(k,v).$$

Similarly we have:

$$\begin{aligned}\frac{\partial X_{(l)}(k,x)}{\partial b_{(l)}(u)} &= \frac{\partial X_{(l)}(k,x)}{\partial Z_{(l)}(k,x))} \frac{\partial Z_{(l)}(k,x)}{\partial b_{(l)}(u)} \\ &= f'_{(l)}(Z_{(l)}(k,x)) \frac{\partial Z_{(l)}(k,x)}{\partial b_{(l)}(u)} \\ &= f'_{(l)}(Z_{(l)}(k,x)) 1_{x=u}.\end{aligned}$$

And hence:

$$\frac{\partial L}{\partial b_{(l)}(u)} = \sum_{k=1}^{n} \delta_{(l)}(k,u) f'_{(l)}(Z_{(l)}(k,u)).$$

3.2.3 Computing Next Gradients

Now we can compute the next gradients:

$$\frac{\partial L}{\partial X_{(l-1)}(k,y)} \text{ with } 1 \leq k \leq n \text{ and } 1 \leq y \leq x_{(l-1)}.$$

We have:

$$\begin{aligned}\frac{\partial L}{\partial X_{(l-1)}(k,y)} &= \sum_{x=1}^{x_{(l)}} \frac{\partial L}{\partial X_{(l)}(k,x)} \frac{\partial X_{(l)}(k,x)}{\partial Z_{(l)}(k,x)} \frac{\partial Z_{(l)}(k,x)}{\partial X_{(l-1)}(k,y)} \\ &= \sum_{x=1}^{x_{(l)}} \delta_{(l)}(k,x) f'_{(l)}(Z_{(l)}(k,x)) W_{(l)}(x,y).\end{aligned}$$

3.3 Convolutional Layer

In this subsection we derive the appropriate formulas for a convolutional layer.

3.3.1 Notations

First of all, we define the notations used in this subsection:

l	current layer
$l-1$	previous layer
n	number of samples
$h_{(l-1)}$	number of rows (height) of layer l input (or layer $l-1$ output)
$w_{(l-1)}$	number of columns (width) of layer l input (or layer $l-1$ output)
$c_{(l-1)}$	number of channels of layer l input (or layer $l-1$ output)
$F_{(l)}$	filter size for both height and width
$P_{(l)}$	padding
$S_{(l)}$	stride
$A_{(l-1)}$	layer l input (or layer $l-1$ output) of size $(n, h_{(l-1)}, w_{(l-1)}, c_{(l-1)})$
$B_{(l-1)}$	padded version of $A_{(l-1)}$ of size $(n, h_{(l-1)} + 2P_{(l)}, w_{(l-1)} + 2P_{(l)}, c_{(l-1)})$
$h_{(l)}$	height of layer l output
$w_{(l)}$	width of layer l output
$c_{(l)}$	number of channels of layer l output
$W_{(l)}$	layer l weight of size $(F_{(l)}, F_{(l)}, c_{(l)}, c_{(l-1)})$
$b_{(l)}$	layer l bias of size $c_{(l)}$
$f_{(l)}$	layer l activation function
$Z_{(l)}$	layer l unactivated output of size $(n, h_{(l)}, w_{(l)}, c_{(l)})$
$A_{(l)}$	layer l activated output of size $(n, h_{(l)}, w_{(l)}, c_{(l)})$

By definition, we have:

$$h_{(l)} := \left\lfloor \frac{h_{(l-1)} + 2P_{(l)} - F_{(l)}}{S_{(l)}} \right\rfloor + 1 \text{ and } w_{(l)} := \left\lfloor \frac{w_{(l-1)} + 2P_{(l)} - F_{(l)}}{S_{(l)}} \right\rfloor + 1.$$

Let $1 \leq k \leq n$, $1 \leq i \leq h_{(l)}$, $1 \leq j \leq w_{(l)}$ and $1 \leq c_o \leq c_{(l)}$. We also have:

$$Z_{(l)}(k,i,j,c_o) := \sum_{c_i=1}^{c_{(l)}} \sum_{u=1}^{F_{(l)}} \sum_{v=1}^{F_{(l)}} W_{(l)}(u,v,c_i,c_o) B_{(l-1)}(k,(i-1)S_{(l)}+u,(j-1)S_{(l)}+v,c_i) + b_{(l)}(c_o),$$

$$A_{(l)}(k,i,j,c_o) := f_{(l)}(Z_{(l)}(k,i,j,c_o)).$$

3.3.2 Derivatives of Weight and Bias

Let's compute the derivative of the loss relatively to the weight of layer l. For that, we assume that we already know for every $1 \leq k \leq n$, $1 \leq i \leq h_{(l)}$, $1 \leq j \leq w_{(l)}$ and $1 \leq c_o \leq c_{(l)}$ the value of:

$$\delta_{(l)}(k,i,j,c_o) := \frac{\partial L}{\partial A_{(l)}(k,i,j,c_o)}.$$

Let $1 \leq u \leq F_{(l)}$, $1 \leq v \leq F_{(l)}$, $1 \leq c_i \leq c_{(l-1)}$ and $1 \leq c_o \leq c_{(l)}$. We have:

$$\frac{\partial L}{\partial W_{(l)}(u,v,c_i,c_o)} = \sum_{k=1}^{n}\sum_{i=1}^{h_{(l)}}\sum_{j=1}^{w_{(l)}} \frac{\partial L}{\partial A_{(l)}(k,i,j,c_o)} \frac{\partial A_{(l)}(k,i,j,c_o)}{\partial W_{(l)}(u,v,c_i,c_o)}$$
$$= \sum_{k=1}^{n}\sum_{i=1}^{h_{(l)}}\sum_{j=1}^{w_{(l)}} \delta_{(l)}(k,i,j,c_o) \frac{\partial A_{(l)}(k,i,j,c_o)}{\partial W_{(l)}(u,v,c_i,c_o)}.$$

We only have to compute the second term, let $1 \leq k \leq n$, $1 \leq i \leq h_{(l)}$, $1 \leq j \leq w_{(l)}$ and $1 \leq c_o \leq c_{(l)}$:

$$\frac{\partial A_{(l)}(k,i,j,c_o)}{\partial W_{(l)}(u,v,c_i,c_o)} = \frac{\partial A_{(l)}(k,i,j,c_o)}{\partial Z_{(l)}(k,i,j,c_o))} \frac{\partial Z_{(l)}(k,i,j,c_o)}{\partial W_{(l)}(u,v,c_i,c_o)}$$
$$= f'_{(l)}(Z_{(l)}(k,i,j,c_o)) \frac{\partial Z_{(l)}(k,i,j,c_o)}{\partial W_{(l)}(u,v,c_i,c_o)}$$
$$= f'_{(l)}(Z_{(l)}(k,i,j,c_o)) B_{(l-1)}(k,(i-1)S_{(l)}+u,(j-1)S_{(l)}+v,c_i).$$

Similarly we have:

$$\frac{\partial A_{(l)}(k,i,j,c_o)}{\partial b_{(l)}(c_o)} = \frac{\partial A_{(l)}(k,i,j,c_o)}{\partial Z_{(l)}(k,i,j,c_o))} \frac{\partial Z_{(l)}(k,i,j,c_o)}{\partial b_{(l)}(c_o)}$$
$$= f'_{(l)}(Z_{(l)}(k,i,j,c_o)) \frac{\partial Z_{(l)}(k,i,j,c_o)}{\partial b_{(l)}(c_o)}$$
$$= f'_{(l)}(Z_{(l)}(k,i,j,c_o)).$$

And hence:

$$\frac{\partial L}{\partial b_{(l)}(c_o)} = \sum_{k=1}^{n}\sum_{i=1}^{h_{(l)}}\sum_{j=1}^{w_{(l)}} \frac{\partial L}{\partial A_{(l)}(k,i,j,c_o)} \frac{\partial A_{(l)}(k,i,j,c_o)}{\partial b_{(l)}(c_o)}$$
$$= \sum_{k=1}^{n}\sum_{i=1}^{h_{(l)}}\sum_{j=1}^{w_{(l)}} \delta_{(l)}(k,i,j,c_o) f'_{(l)}(Z_{(l)}(k,i,j,c_o)).$$

3.3.3 Computing Next Gradients

Now we can compute the next gradients:

$$\frac{\partial L}{\partial A_{(l-1)}(k,a,b,c_i)} \text{ with } 1 \leq k \leq n,\ 1 \leq a \leq h_{(l-1)},\ 1 \leq b \leq w_{(l-1)} \text{ and } 1 \leq c_i \leq c_{(l-1)}.$$

We have:

$$\frac{\partial L}{\partial A_{(l-1)}(k,a,b,c_i)} = \sum_{i=1}^{h_{(l)}}\sum_{j=1}^{w_{(l)}}\sum_{c_o=1}^{c_{(l)}} \frac{\partial L}{\partial A_{(l)}(k,i,j,c_o)} \frac{\partial A_{(l)}(k,i,j,c_o)}{\partial Z_{(l)}(k,i,j,c_o)} \frac{\partial Z_{(l)}(k,i,j,c_o)}{\partial A_{(l-1)}(k,a,b,c_i)}$$
$$= \sum_{i=1}^{h_{(l)}}\sum_{j=1}^{w_{(l)}}\sum_{c_o=1}^{c_{(l)}} \delta_{(l)}(k,i,j,c_o) f'_{(l)}(Z_{(l)}(k,i,j,c_o)) \frac{\partial Z_{(l)}(k,i,j,c_o)}{\partial A_{(l-1)}(k,a,b,c_i)}.$$

Let $1 \leq i \leq h_{(l)}$, $1 \leq j \leq w_{(l)}$ and $1 \leq c_o \leq c_{(l)}$. Let's focus on the last term of the sum:

$$\frac{\partial Z_{(l)}(k,i,j,c_o)}{\partial A_{(l-1)}(k,a,b,c_i)} = \frac{\partial Z_{(l)}(k,i,j,c_o)}{\partial B_{(l-1)}(k,a+P_{(l)},b+P_{(l)},c_i)} \frac{\partial B_{(l-1)}(k,a+P_{(l)},b+P_{(l)},c_i)}{\partial A_{(l-1)}(k,a,b,c_i)}.$$

Remember that B is pretty much a padded version of A, we conclude that the second term will here be 1. So we are left computing the first term. For that, recall the following formula:

$$Z_{(l)}(k,i,j,c_o) = \sum_{c_i=1}^{c_{(l)}} \sum_{u=1}^{F_{(l)}} \sum_{v=1}^{F_{(l)}} W_{(l)}(u,v,c_i,c_o) B_{(l-1)}(k,(i-1)S_{(l)}+u,(j-1)S_{(l)}+v,c_i) + b_{(l)}(c_o).$$

From this formula it is pretty obvious that the derivative will be equal to $W_{(l)}(u,v,c_i,c_o)$ if there exists $1 \leq u \leq F_{(l)}$ and $1 \leq v \leq F_{(l)}$ such that:

$$a + P_{(l)} = (i-1)S_{(l)} + u \text{ and } b + P_{(l)} = (j-1)S_{(l)} + v.$$

If those do not exist, then the derivative will be 0. Now we can compute for which i and j this is the case (to avoid testing every combination in practice). Let's focus on the "rows" for the moment (the "columns" being symmetrical in reasoning). Such an u exists if we have:

$$\begin{aligned} 1 \leq a + P_{(l)} - (i-1)S_{(l)} \leq F_{(l)} &\iff 1 - a - P_{(l)} \leq -(i-1)S_{(l)} \leq F_{(l)} - a - P_{(l)} \\ &\iff \frac{a + P_{(l)} - 1}{S_{(l)}} + 1 \leq i \leq \frac{a + P_{(l)} - F_{(l)}}{S_{(l)}} + 1. \end{aligned}$$

Considering the initial domain of i, we can deduce that this exists if:

$$i \in \left\{ \max\left(1, \left\lceil \frac{a + P_{(l)} - 1}{S_{(l)}} + 1 \right\rceil\right), \ldots, \min\left(h_{(l)}, \left\lfloor \frac{a + P_{(l)} - F_{(l)}}{S_{(l)}} + 1 \right\rfloor\right) \right\}.$$

Similarly, we have:

$$j \in \left\{ \max\left(1, \left\lceil \frac{b + P_{(l)} - 1}{S_{(l)}} + 1 \right\rceil\right), \ldots, \min\left(w_{(l)}, \left\lfloor \frac{b + P_{(l)} - F_{(l)}}{S_{(l)}} + 1 \right\rfloor\right) \right\}.$$

Finally for those values of i and j the derivative will be equal to:

$$W_{(l)}(a + P_{(l)} - (i-1)S_{(l)}, b + P_{(l)} - (j-1)S_{(l)}, c_i, c_o).$$

3.4 Pooling Layer

In this subsection we derive the appropriate formulas for a pooling layer.

3.4.1 Notations

First of all, we define the notations used in this subsection:

l	current layer
$l-1$	previous layer
n	number of samples
$h_{(l-1)}$	number of rows (height) of layer l input (or layer $l-1$ output)
$w_{(l-1)}$	number of columns (width) of layer l input (or layer $l-1$ output)
$c_{(l-1)}$	number of channels of layer l input (or layer $l-1$ output)
$F_{(l)}$	pooling size for both height and width
$P_{(l)}$	padding
$S_{(l)}$	stride
$A_{(l-1)}$	layer l input (or layer $l-1$ output) of size $(n, h_{(l-1)}, w_{(l-1)}, c_{(l-1)})$
$B_{(l-1)}$	padded version of $A_{(l-1)}$ of size $(n, h_{(l-1)} + 2P_{(l)}, w_{(l-1)} + 2P_{(l)}, c_{(l-1)})$
$h_{(l)}$	height of layer l output
$w_{(l)}$	width of layer l output
$f_{(l)}$	average function or maximum function
$A_{(l)}$	layer l output of size $(n, h_{(l)}, w_{(l)}, c_{(l-1)})$

By definition, we have:

$$h_{(l)} := \left\lfloor \frac{h_{(l-1)} + 2P_{(l)} - F_{(l)}}{S_{(l)}} \right\rfloor + 1 \text{ and } w_{(l)} := \left\lfloor \frac{w_{(l-1)} + 2P_{(l)} - F_{(l)}}{S_{(l)}} \right\rfloor + 1.$$

Let $1 \leq k \leq n$, $1 \leq i \leq h_{(l)}$, $1 \leq j \leq w_{(l)}$ and $1 \leq c \leq c_{(l-1)}$. We also have:

$$A_{(l)}(k, i, j, c) := f_{(l)}(\{B_{(l-1)}(k, (i-1)S_{(l)} + u, (j-1)S_{(l)} + v, c) | 1 \leq u \leq F_{(l)} \text{ and } 1 \leq v \leq F_{(l)}\}).$$

3.4.2 Computing Next Gradients

Now we can compute the next gradients. For that, we assume that we already know for every $1 \leq k \leq n$, $1 \leq i \leq h_{(l)}$, $1 \leq j \leq w_{(l)}$ and $1 \leq c \leq c_{(l-1)}$ the value of:

$$\delta_{(l)}(k, i, j, c) := \frac{\partial L}{\partial A_{(l)}(k, i, j, c)}.$$

We are interested in:

$$\frac{\partial L}{\partial A_{(l-1)}(k, a, b, c)} \text{ with } 1 \leq k \leq n,\ 1 \leq a \leq h_{(l-1)},\ 1 \leq b \leq w_{(l-1)} \text{ and } 1 \leq c \leq c_{(l-1)}.$$

We have:

$$\begin{aligned} \frac{\partial L}{\partial A_{(l-1)}(k, a, b, c)} &= \sum_{i=1}^{h_{(l)}} \sum_{j=1}^{w_{(l)}} \frac{\partial L}{\partial A_{(l)}(k, i, j, c)} \frac{\partial A_{(l)}(k, i, j, c)}{\partial A_{(l-1)}(k, a, b, c)} \\ &= \sum_{i=1}^{h_{(l)}} \sum_{j=1}^{w_{(l)}} \delta_{(l)}(k, i, j, c) \frac{\partial A_{(l)}(k, i, j, c)}{\partial A_{(l-1)}(k, a, b, c)}. \end{aligned}$$

Let $1 \leq i \leq h_{(l)}$, $1 \leq j \leq w_{(l)}$ and $1 \leq c \leq c_{(l-1)}$. Let's focus on the last term of the sum:

$$\frac{\partial A_{(l)}(k, i, j, c)}{\partial A_{(l-1)}(k, a, b, c)} = \frac{\partial A_{(l)}(k, i, j, c)}{\partial B_{(l-1)}(k, a + P_{(l)}, b + P_{(l)}, c)} \frac{\partial B_{(l-1)}(k, a + P_{(l)}, b + P_{(l)}, c)}{\partial A_{(l-1)}(k, a, b, c)}.$$

Remember that B is pretty much a padded version of A, we conclude that the second term will here be 1. So we are left computing the first term. For that, recall the following formula:

$$A_{(l)}(k,i,j,c) := f_{(l)}(\{B_{(l-1)}(k,(i-1)S_{(l)}+u,(j-1)S_{(l)}+v,c) | 1 \leq u \leq F_{(l)} \text{ and } 1 \leq v \leq F_{(l)}\}).$$

From this formula it is pretty obvious that the derivative will be equal to $f'_{(l)}$ with respect to $B_{(l-1)}(k,a+P_{(l)},b+P_{(l)},c)$ evaluated at the given set if there exists $1 \leq u \leq F_{(l)}$ and $1 \leq v \leq F_{(l)}$ such that:

$$a + P_{(l)} = (i-1)S_{(l)} + u \text{ and } b + P_{(l)} = (j-1)S_{(l)} + v.$$

If those do not exist, then the derivative will be 0. Now we can compute for which i and j this is the case (to avoid testing every combination in practice). Let's focus on the "rows" for the moment (the "columns" being symmetrical in reasoning). Such an u exists if we have:

$$\begin{aligned} 1 \leq a + P_{(l)} - (i-1)S_{(l)} \leq F_{(l)} &\iff 1 - a - P_{(l)} \leq -(i-1)S_{(l)} \leq F_{(l)} - a - P_{(l)} \\ &\iff \frac{a + P_{(l)} - 1}{S_{(l)}} + 1 \leq i \leq \frac{a + P_{(l)} - F_{(l)}}{S_{(l)}} + 1. \end{aligned}$$

Considering the initial domain of i, we can deduce that this exists if:

$$i \in \left\{ \max\left(1, \left\lceil \frac{a + P_{(l)} - 1}{S_{(l)}} + 1 \right\rceil\right), \ldots, \min\left(h_{(l)}, \left\lfloor \frac{a + P_{(l)} - F_{(l)}}{S_{(l)}} + 1 \right\rfloor\right) \right\}.$$

Similarly, we have:

$$j \in \left\{ \max\left(1, \left\lceil \frac{b + P_{(l)} - 1}{S_{(l)}} + 1 \right\rceil\right), \ldots, \min\left(w_{(l)}, \left\lfloor \frac{b + P_{(l)} - F_{(l)}}{S_{(l)}} + 1 \right\rfloor\right) \right\}.$$

Finally for those values of i and j the derivative will be equal to:

- 1 if we are using max pooling and the max is reached in $B_{(l-1)}(k,a+P_{(l)},b+P_{(l)},c)$,

- 0 if we are using max pooling and the max is not reached in $B_{(l-1)}(k,a+P_{(l)},b+P_{(l)},c)$,

- $\frac{1}{F_l^2}$ if we are using average pooling.

For the max pooling case, instead of recomputing the maximum we can verify for the valid i and j if $A_{(l)}(k,i,j,c) = A_{(l-1)}(k,a,b,c)$. Computationally speaking it will be way lighter.