Research Project

Iniciação Científica

Deep Learning applied to facial recognition

Student: Kaique Nunes de Oliveira
Bachelor in Computer Science

Advisor: Nina S. T. Hirata
Department of Computer Science

Institute of Mathematics and Statistics
University of São Paulo

Abstract

The idea behind this document is to keep track of my studying process to understand Deep Learning and its use in facial recognition. I believe that writing it down will be a good way to improve my english and will help me write the reports.

São Paulo, 28 de setembro de 2023

1 Theory

1.1 The Perceptron Model

To understand deep learning we need to begin somewhere, this 'somewhere' is the Perceptron Model. This model was developed by the scientist Frank Rosenblat, in the 1950s and 1960s and it is meant to simulate a single neuron. The idea is simple: you have $\mathbf{w}$, which is a column vector of weights, and you have $\mathbf{x_n}$, which is a vector of features. If $\mathbf{w^T x_n}$ is greater than or equal to some threshold, then you label $\mathbf{x_n}$ as some class. We can write it as follows:

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^T \mathbf{x})$$

This model has some limitations. It is only capable of doing classification in linearly separable data.

1.2 The Multi-layer Perceptron

To be able to separate non-linearly a dataset we developed the Multi-layer Perceptron (MLP) model. We can easily notice that the perceptron model cannot solve a significant amount of classification problems, the XOR problem is an example.

You can see that we need at least two perceptrons to be able to solve the XOR properly. The MLP model is capable of using these two perceptrons to make classification.

1.3 Neural Networks

The MLP is a parent of the Neural Networks. In the MLP we use the threshold as an activation function. But it is not a good function for many problems due to its linear nature and its problems with derivation.

Neural Networks are a computational model composed by layers of artificial neurons, also known as processing units, that are interconnected with the goal to learn complex tasks concerning information processing.

The neurons are responsible to do mathematical operations by receiving an input and then creating an output. Each neuron is associated with several weights from the previous layer and the next layer, they are also associated with a bias. The weights and biases are tuned in the learning process.

1.3.1 Summary

Each neuron receives as input the weighted sum of its bias and the outputs of all neurons of the previous layer. The result of this sum is then applied to an activation function. The output of this function is the output of the neuron.

The activation functions are used in the neurons to introduce non-linearity in the outputs of the networks. Some of the most common functions include the sigmoid, ReLU, and the hyperbolic tangent. The choice of the activation function may affect the performance of the network.

The forward propagation is the first step of the process. The data flows from the input layer to the last layer by passing through the hidden layers. Each of the neurons of the first layer receives as input the weighted sum of the input data and its bias, then the output of each of these neurons are passed to the next layer. This process is repeated through the intermediate layers until the last layer is reached.

After forward propagation, the backward propagation is the algorithm responsible to tune the weights and biases of the neural network in a way that it minimizes the difference between the model's output and the labels of the train data-set. The method is based on the chain derivation rule and uses dynamic programming to calculate the gradients of the parameters in relation to the cost function.

1.3.2 Notation

To really understand the theory behind Neural Networks there is the need to introduce a notation. We will use the following graphic representation:

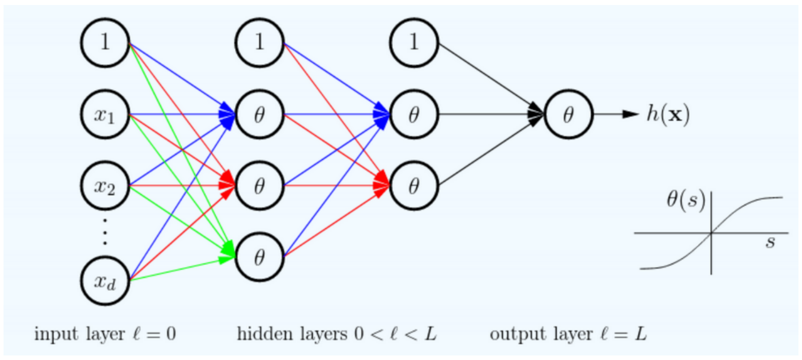


Figura 1: Example of Neural Network (Source: Abu-Mostafa et al. (2012))

The layers are labeled by $l = 0, 1, 2, ...L$. Actually, layer $l = 0$ is meant to be the input layer and is often not seen really as a layer. However, the $l = L$ is the output layer and is seen as a proper layer. Also, the layers $l = 1, 2, ..., L - 1$ are called hidden layers. The superscript$^{(l)}$ will be used to refer to a particular layer. Furthermore, layers also have 'dimensions' $d^{(l)}$: if a layer $l = 3$ has dimension $d^{(l)}$, then it means that the layer $l = 3$ has $d^{(l)} + 1$ nodes labeled $0, 1, ..., d^{(l)}$. Notice that the 0 node represents the bias and it is set to always have 1 as output and it does't have any inputs.

To extend our notation we will now take a look at the relation between two nodes.

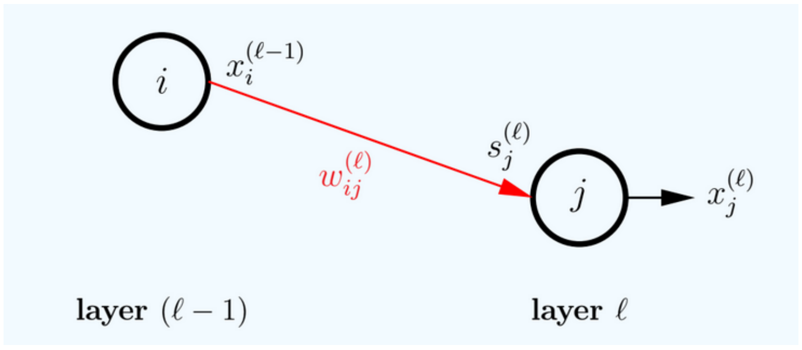


Figura 2: Relationship between two nodes (Fonte: Abu-Mostafa et al. (2012))

Note that a node has an incoming signal s and an output x, based on that we will create two vector. The vector $\mathbf{s}^{(l)}$ will be the signal vector, it represents the input signals received by the $1, 2, ..., d^{(l)}$ nodes in the layer l, remember that the node 0 doesn't have any signal incoming. In addition, the vector $\mathbf{x}^{(l)}$ represents the outputs of the $0, 1, 2, ..., d^{(l)}$ nodes in the layer l. So, the $s_j^{(l)}$ entry is the signal incoming the node j in layer l, and the $x_j^{(l)}$ is the output of the node j in layer l.

We also need a representation for the weights. Given that there are links connecting the outputs of all nodes in the layer $l-1$ to the inputs of the layer l, we can say that we have a $(d^{(l-1)} + 1) \times d^{(l)}$ matrix of weights $W^{(l)}$. Furthermore, each entry $w_{ij}^{(l)}$ of the matrix $W^{(l)}$ is the weight that links the node i of the layer $l-1$ to the node j of the layer l.

Therefore, the set of matrices $\mathbf{w} = \{W^{(1)}, W^{(2)}, ..., W^{(L)}\}$ is our weight parameter.

1.3.3 Forward Propagation

The hypothesis $h(\mathbf{x})$ is calculated by the forward propagation algorithm, we will dive into that now.

First observe that to get the input vector into layer l, we compute the weighted sum of the outputs from the previous layer. That is, with weights given by $W^{(l)}$, $s_j^{(l)} = \sum_{i=0}^{d(l-1)} w_{ij}^{(l)} x_i^{(l-1)}$. This process can be represented by the matrix equation

$$\mathbf{s}^{(l)} = (W^{(l)})^T \mathbf{x}^{(l-1)}$$

Computed the $\mathbf{s}^{(l)}$ vector, we can now get the $\mathbf{x}^{(l)}$ by doing the following step:

$$\mathbf{x}^{(l)} = \begin{bmatrix} 1 \\ \theta(\mathbf{s}^{(l)}) \end{bmatrix}$$

So, the forward propagation algorithm can be represented as the following chain of events:

$$\mathbf{x} = \mathbf{x}^{(0)} \xrightarrow{W^{(1)}} \mathbf{s}^{(1)} \xrightarrow{\theta} \mathbf{x}^{(1)} ... \rightarrow \mathbf{s}^{(L)} \xrightarrow{\theta} \mathbf{x}^{(L)} = h(\mathbf{x})$$

1.3.4 Backpropagation

The backpropagation algorithm is an efficient way to compute the gradient $\nabla \mathcal{L}(\mathbf{w})$ o an error function that a neural network has. We often use the sum of squares,

$$\begin{aligned} E_{in}(\mathbf{w}) &= \frac{1}{N} \sum_{n=1}^{N} (h(x_n; \mathbf{w}) - y_n)^2 \\ &= \frac{1}{N} \sum_{n=1}^{N} (x_n^{(L)} - y_n)^2 \\ &= \frac{1}{N} \sum_{n=1}^{N} e_n \end{aligned}$$

To compute the gradient of E_{in}, we need its derivatives with respect to each weight matrix:

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial W^l} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial e_n}{\partial W^{(l)}}$$

These are the partial derivatives that we want to find. To obtain them the idea is to compute the partial derivatives of the layer l from the partial derivatives of the layer $l+1$. To do that, we will need to introduce the so called 'sensibility' vector $\delta^{(l)}$.

$$\delta^{(l)} = \frac{\partial e}{\partial \mathbf{s}^{(l)}} = \begin{bmatrix} \frac{\partial e}{\partial s_1^{(l)}} \\ \frac{\partial e}{\partial s_2^{(l)}} \\ \dots \\ \frac{\partial e}{\partial s_{d^{(l)}}^{(l)}} \end{bmatrix}$$

The sensitivity tells us how e changes with $\mathbf{s}^{(l)}$. We can write:

$$\frac{\partial e}{\partial W^{(l)}} = \mathbf{x}^{(l-1)}(\delta^{(l)})^T$$

The reason for that will be explained now. Remember the chain of events already introduced in forward propagation:

$$\mathbf{x} = \mathbf{x}^{(0)} \xrightarrow{W^{(1)}} \mathbf{s}^{(1)} \xrightarrow{\theta} \mathbf{x}^{(1)} \dots \rightarrow \mathbf{s}^{(L)} \xrightarrow{\theta} \mathbf{x}^{(L)} = h(\mathbf{x})$$

The error depends on $\mathbf{s}^l$, which in turn depends on $W^{(l)}$. Note that, because of the chain rule, the following statement is true:

$$\frac{\partial e}{\partial w_{ij}^{(l)}} = \frac{\partial e}{\partial \mathbf{s}_j^{(l)}} . \frac{\partial \mathbf{s}_j^{(l)}}{\partial w_{ij}^{(l)}}$$

Also, because of the definition of the vector $\mathbf{s}^{(l)}$:

$$\mathbf{s}_j^{(l)} = \sum_{n=1}^{d^{(l-1)}} w_{nj}^{(l)} \mathbf{x}_n^{(l-1)}$$

Thus, we have:

$$\frac{\partial \mathbf{s}_j^{(l)}}{\partial w_{ij}^{(l)}} = \frac{\partial \sum_{n=1}^{d^{(l-1)}} w_{nj}^{(l)} \mathbf{x}_n^{(l-1)}}{\partial w_{ij}^{(l)}} = \mathbf{x}_i^{(l-1)}$$

Remember that $\frac{\partial e}{\partial \mathbf{s}_j^{(l)}} = \delta_j^{(l)}$ and $\frac{\partial e}{\partial w_{ij}^{(l)}} = \frac{\partial e}{\partial \mathbf{s}_j^{(l)}} . \frac{\partial \mathbf{s}_j^{(l)}}{\partial w_{ij}^{(l)}}$, so:

$$\frac{\partial e}{\partial w_{ij}^{(l)}} = \mathbf{x}_i^{(l-1)} . \delta_j^{(l)}$$

Then, finally:

$$\frac{\partial e}{\partial W^{(l)}} = \mathbf{x}^{(l-1)}(\delta^{(l)})^T$$

The formula for the sensibility vectors used in backpropagation is

$$\delta^{(l)} = \sigma'(\mathbf{s}^{(l)}) \otimes [W^{(l+1)}\delta^{(l+1)}]_1^{d^{(l)}}$$

The σ' is the derivative of the activation function, the vector $[W^{(l+1)}\delta^{(l+1)}]_1^{d^{(l)}}$ contains the components of the vector $W^{(l+1)}\delta^{(l+1)}$ (we need to exclude the bias, which has index 0). The $\otimes$ is the notation for the component-wise multiplication, also known as the Hadamard product.

To understand the equation we must first remind that

$$e = e(\mathbf{x}^{(L)}, y) = e(\sigma(\mathbf{s}^{(L)}), y)$$

Because of the chain rule, we can write:

$$\frac{\partial e}{\partial \mathbf{s}^{(l)}} = \frac{\partial e}{\partial \mathbf{x}^{(l)}}.\frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{s}^{(l)}}$$

$$\delta_j^{(l)} = \frac{\partial e}{\partial \mathbf{s}_j^{(l)}} = \frac{\partial e}{\partial \mathbf{x}_j^{(l)}}.\frac{\partial \mathbf{x}_j^{(l)}}{\partial \mathbf{s}_j^{(l)}}$$

Since $\mathbf{x}^{(l)}$ is a function of $\mathbf{s}^{(l)}$:

$$\frac{\partial \mathbf{x}_j^{(l)}}{\partial \mathbf{s}_j^{(l)}} = \sigma'(\mathbf{s}^{(l)}) \Longrightarrow \delta_j^{(l)} = \frac{\partial e}{\partial \mathbf{x}_j^{(l)}}.\sigma'(\mathbf{s}^{(l)})$$

It makes sense to say that each element of $\mathbf{x}^{(l)}$ influences all the elements of $\mathbf{s}^{l+1}$. Therefore, to obtain the derivation that is lacking in the previous equation, we must sum all of these influences:

$$\frac{\partial e}{\partial \mathbf{x}_j^{(l)}} = \sum_{k=1}^{d^{(l+1)}} \frac{\partial e}{\partial \mathbf{s}_k^{(l+1)}}.\frac{\partial \mathbf{s}_k^{(l+1)}}{\partial \mathbf{x}_j^{(l)}} = \sum_{k=1}^{d^{(l+1)}} \delta_k^{(l+1)}.w_{jk}^{(l+1)}$$

Finally, now we can find the formula for the sensitivities used in backpropagation:

$$\delta_j^{(l)} = \sigma'(\mathbf{s}^{(l)}).\sum_{k=1}^{d^{(l+1)}} \delta_k^{(l+1)}.w_{jk}^{(l+1)}$$

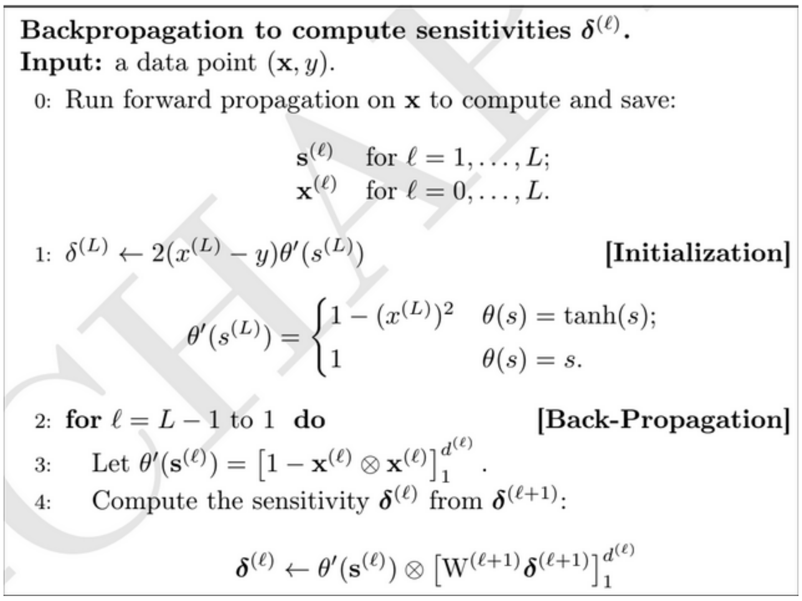
We now have found a way to compute $\delta^{(l)}$ from $\delta^{(L)}$. This means that, to find all the sensitivity vectors, we first need to find $\delta^{(L)}$:

$$\delta^{(L)} = \frac{\partial e}{\partial \mathbf{s}^{(L)}} = \frac{e(\mathbf{x}^{(L)}, y)}{\partial \mathbf{s}^{(L)}}$$

This means that $\delta^{(L)}$ is dependent on the error function that the neural network wants to minimize. Take as an example the sum of squares as the error function, that is $e = (\mathbf{x}^{(L)}) = (\sigma(\mathbf{s}^{(L)}) - y)^2$. Therefore:

$$\delta^{(L)} = \frac{\partial e}{\partial \mathbf{s}^{(L)}} = \frac{\partial(\mathbf{x}^{(L)} - y)^2}{\partial \mathbf{s}^{(L)}} = 2(\mathbf{x}^{(L)} - y)\frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} = 2(\mathbf{x}^{(L)} - y)\sigma'(\mathbf{s}^{(L)})$$

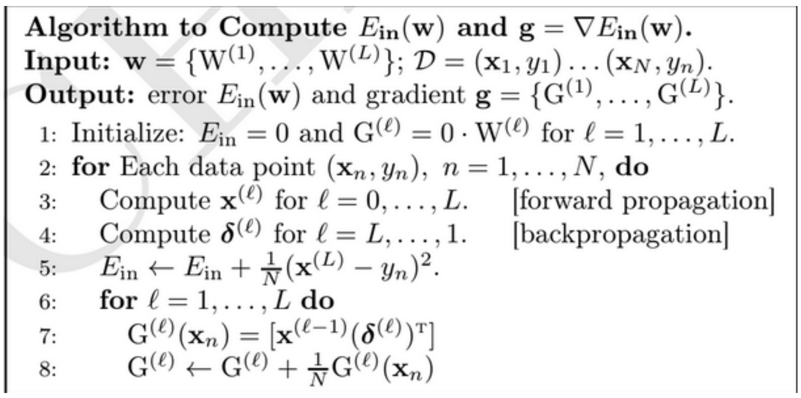
In summary, backpropagation is an algorithm that computes the sensitivity vectors after forward propagation is done with a point from the data-set. You can see an example os the algorithm in the folowing image, which uses the sum of squares as the error function and the tanh(s) as the activation function:



```
Backpropagation to compute sensitivities $\boldsymbol{\delta}^{(\ell)}$.
Input: a data point $(\mathbf{x}, y)$.
0: Run forward propagation on $\mathbf{x}$ to compute and save:
       $\mathbf{s}^{(\ell)}$   for $\ell = 1, \ldots, L$;
       $\mathbf{x}^{(\ell)}$   for $\ell = 0, \ldots, L$.
1: $\delta^{(L)} \leftarrow 2(x^{(L)} - y)\theta'(s^{(L)})$        [Initialization]
       $\theta'(s^{(L)}) = \begin{cases} 1 - (x^{(L)})^2 & \theta(s) = \tanh(s); \\ 1 & \theta(s) = s. \end{cases}$
2: for $\ell = L - 1$ to $1$ do        [Back-Propagation]
3:     Let $\theta'(\mathbf{s}^{(\ell)}) = \left[1 - \mathbf{x}^{(\ell)} \otimes \mathbf{x}^{(\ell)}\right]_1^{d^{(\ell)}}$.
4:     Compute the sensitivity $\boldsymbol{\delta}^{(\ell)}$ from $\boldsymbol{\delta}^{(\ell+1)}$:
           $\boldsymbol{\delta}^{(\ell)} \leftarrow \theta'(\mathbf{s}^{(\ell)}) \otimes \left[\mathrm{W}^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}\right]_1^{d^{(\ell)}}$
```

Figura 3: Backpropagation Algorithm (Source: Abu-Mostafa et al. (2012))

Now, with forward propagation and backpropagation, the neural network is able to find the gradients:



```
Algorithm to Compute $E_{\text{in}}(\mathbf{w})$ and $\mathbf{g} = \nabla E_{\text{in}}(\mathbf{w})$.
Input: $\mathbf{w} = \{\mathrm{W}^{(1)}, \ldots, \mathrm{W}^{(L)}\}$; $\mathcal{D} = (\mathbf{x}_1, y_1) \ldots (\mathbf{x}_N, y_n)$.
Output: error $E_{\text{in}}(\mathbf{w})$ and gradient $\mathbf{g} = \{\mathrm{G}^{(1)}, \ldots, \mathrm{G}^{(L)}\}$.
1: Initialize: $E_{\text{in}} = 0$ and $\mathrm{G}^{(\ell)} = 0 \cdot \mathrm{W}^{(\ell)}$ for $\ell = 1, \ldots, L$.
2: for Each data point $(\mathbf{x}_n, y_n)$, $n = 1, \ldots, N$, do
3:     Compute $\mathbf{x}^{(\ell)}$ for $\ell = 0, \ldots, L$.    [forward propagation]
4:     Compute $\boldsymbol{\delta}^{(\ell)}$ for $\ell = L, \ldots, 1$.    [backpropagation]
5:     $E_{\text{in}} \leftarrow E_{\text{in}} + \frac{1}{N}(\mathbf{x}^{(L)} - y_n)^2$.
6:     for $\ell = 1, \ldots, L$ do
7:         $\mathrm{G}^{(\ell)}(\mathbf{x}_n) = [\mathbf{x}^{(\ell-1)}(\boldsymbol{\delta}^{(\ell)})^{\mathrm{T}}]$
8:         $\mathrm{G}^{(\ell)} \leftarrow \mathrm{G}^{(\ell)} + \frac{1}{N}\mathrm{G}^{(\ell)}(\mathbf{x}_n)$
```

Figura 4: Gradients Algorithm (Source: Abu-Mostafa et al. (2012))

To update the weights for a single iteration of fixed learning rate gradient descent we just need to do $W^{(l)} \leftarrow W^{(l)} - \eta G^{(l)}$, for $l = 1, ..., L$

1.3.5 Practice with Neural Networks

There is a very well known data-set called MNIST (Modified National Institute of Standards and Technology database). This data-set is composed of 60,000 images for training and 10,000 images for testing. The MNIST is often used as an introduction problem by the beginers in Machine Learning and Deep Learning.

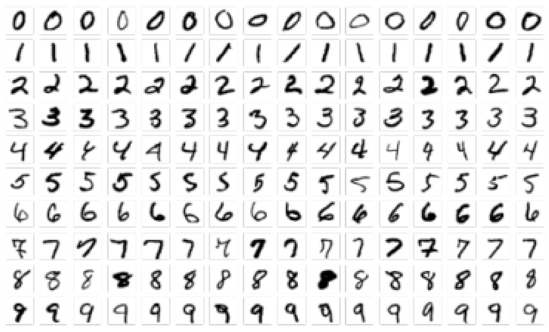


Figura 5: Exemplos de imagens do MNIST

For this data-set I decided to implement a neural network from scratch, it means I only used Numpy. Also, My implementation follows the theory exposed here so far, it took a long time to implement because of some bugs that I needed to sort out to make it actually work. Building my own Neural Network helped me to really understand the theory behind it and made me more aware of the time needed to train a model.

With a Neural Network composed of an input layer of 764 nodes, and two hidden layers composed of 128 the first and 64 the second, and an output layer of 10 nodes, I was able to reach an accuracy of 94,27% for the test data-set of the MNIST after 100 epochs, a mini batch of size 1, and a learning rate of 0.1. The code created is available on my git hub repository.

1.4 Convolutional Neural Networks

CNNs are a specialized kind of Neural Network for processing data that has a know grid topology and they have been very successful in practical applications. This kind of network is based on an operation called convolution, which has '$*$' as its sign. Also, CNNs became really famous in the area of computer vision in the last decade, specially in image classification.

1.4.1 The limits of fully connected layers

Say that we have a data-set composed of one-megapixel images of cats and dogs, and our goal is to build a Neural Network capable of stating if a given image has a dog or a cat in it. This means that each input to the model has one million dimensions and, if we wanted to reduce the dimensions to one thousand, it would require a fully connected hidden layer with $10^6 \times 10^3 = 10^9$ parameters. It's not hard to see that the task of learning from images would become easily infeasible with fully connected layers only. This is why we created the CNNs.

1.4.2 Convolution Operation in CNN

The convolution operation can be easily understood by looking at a simple example:

Every convolution is composed by an input, a kernel (also known as filter), and its output. To find the 19 we just needed to do the output computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$. To find the other elements of the output tensor we just need to slide the kernel across the input tensor, both from the left to right and top to bottom. In addition, note that the output size is smaller than the input size, that is the case because the kernel used has width and height greater than 1, and, to do the computation properly, the filter

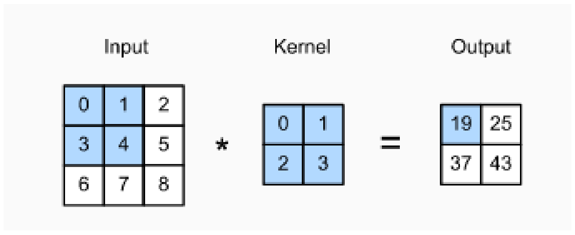


Figura 6: Convolution operation example

needs to fit wholly within the image. The output size is given by the formula:

$$(n_h - k_h + 1) \times (n_w - k_w + 1)$$

where the input size is $n_h \times n_w$ and the kernel size is $k_h \times k_w$.

CNNs use this operation because it can extract important features from images, like edges. To detect a vertical edge in an image we can do this, for example:

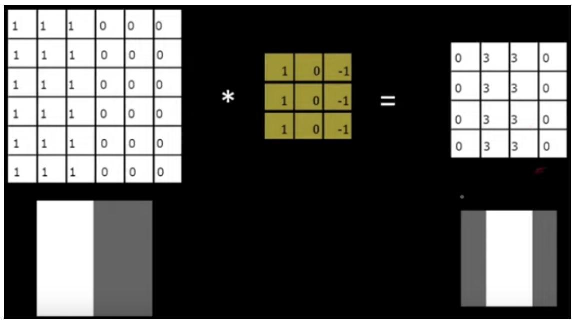


Figura 7

We can see that, where once was the vertical edge in the middle of the input, the output image highlighted its position. In addition, we could use more filters for the same input if we wanted to extract more features, but each filter has its own image as output, this means that, if we use c kernels in one input, then we will have c 'outputs'.

In the examples used here so far, the tensor input has only one channel, which means its dimensions was $n \times n \times 1$. However, it is often not the case since images frequently follow the RGB format, which is a 3-channel approach to represent color. Here we need to learn another rule: the number of channels in the Kernel must always meet the number of channels of the input. This means that, for the RGB format, convolutions would be like:

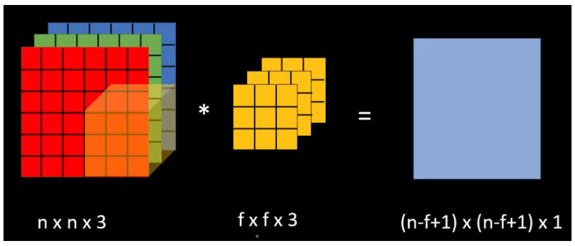


Figura 8

Note that, even though the channels of both the Input and the Kernel is 3, the number of channels in the output remains one.

Also, another thing must be said: a convolutional layer cross-correlates the input and kernel and adds a scalar bias to produce an output. So, the two parameters of a convolution layer are the kernel and the bias. Just like in a fully-connected layer, a convolutional layer often has its parameters initialized randomly in the begining of the learning process.

1.4.3 Padding

As we can see, when convolution is applied, we oftentimes end up with an output of dimension sizes smaller then the original input. This can be a problem because, after each convolution, we are loosing information in the boundaries. In addition, when the kernel is sliding across the input, it tends to do operation more frequently with elements that are far from the perimeter of the inputs, this also contributes to the loss of information from the boundaries of the input.

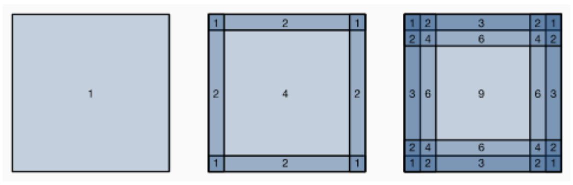


Figura 9: Pixel utilization for convolutions of size 1×1, 2×2, and 3×3, respectively

To deal with this issue we created padding, which basically adding zeros in the boundaries of the input:

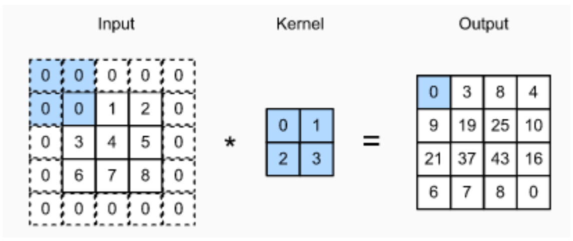


Figura 10: Convolution with padding

In general, if we add p_h rows of padding, often half on top and half on bottom, and a total of p_w columns of padding, often half on top and half on top and half on bottom, the output shape will be

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$$

In many cases we will want to use $p_h = k_h - 1$ and $p_w = k_w - 1$ to give the achieve an output with same height and width as the input. CNNs commonly use filters with odd height and width, this makes it easier to do padding and preserve dimensionality.

1.4.4 Stride

Until now we have been sliding the kernel over the input just one element at a time. However, sometimes, bacause of computational cost or because we want to downsample, we may want use a larger stride.

In general, if the height stride is s_h and the stride for the width is s_w, the output shape will be

$$\lfloor(n_h - k_h + p_h + s_h)/s_h)\rfloor \times \lfloor(n_w - k_w + p_w + s_w)/s_w\rfloor$$

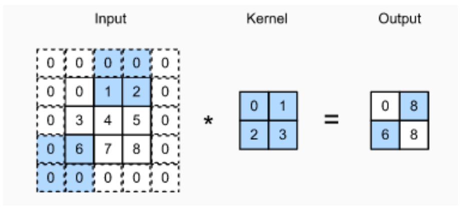


Figura 11: Cross-correlation with strides of 3 and 2 for height and width, respectively

1.4.5 Pooling

The pooling layers serve the dual purposes of mitigating the sensitivity of convolutional layers to location and of spatially downsampling representations.

Like convolutional layers, pooling operators consist of fixed-shape window that slides across the input in the pace of their strides. It is a deterministic operation, there are no parameters to be learned, typically calculating either the maximum of the avarege value of the elements in the pooling window:

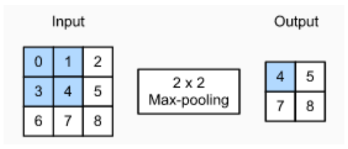


Figura 12

For multi-channel input data, differently from convolutional layers, we pool each input channel separately. This means that the number of output channels is the same as the number of input channels.

Referências

Abu-Mostafa, Y. S., Lin, H.-T., and Magdon-Ismail, M. (2012). *Learning From Data*. AMLBook.