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 report the work done in my research project in context of MAC0215 subject.

São Paulo, December 9, 2023

1 Theory

This section is intended to explain the theoretical work I have studied. It covers the theory behind neural networks and convolutional neural networks. I do not put it here, but another part of the theoretical work done was to study the Tensorflow library so that I managed to create my own models.

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:

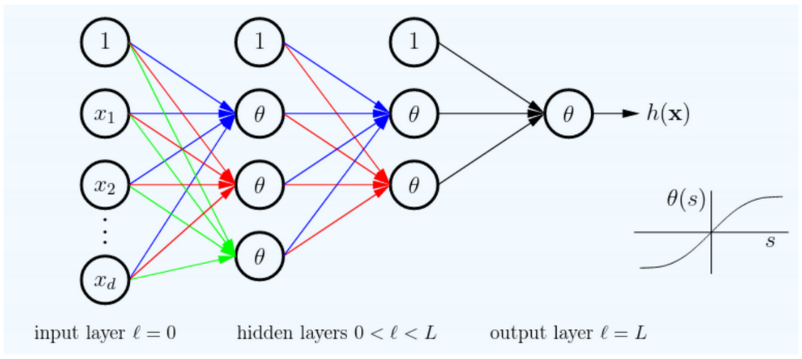


Figure 1: Example of Neural Network (Source: Abu-Mostafa et al. (2012))

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.

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.

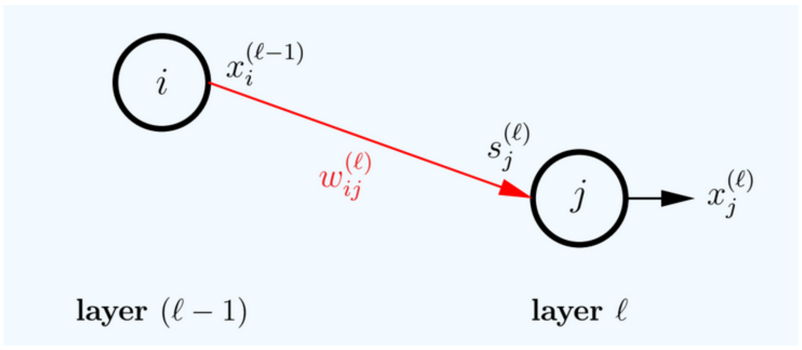


Figure 2: Relationship between two nodes (Fonte: Abu-Mostafa et al. (2012))

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,

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

$$= \frac{1}{N}\sum_{n=1}^{N} e_n$$

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 $\delta^{(l)}$.

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

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:

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

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:

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

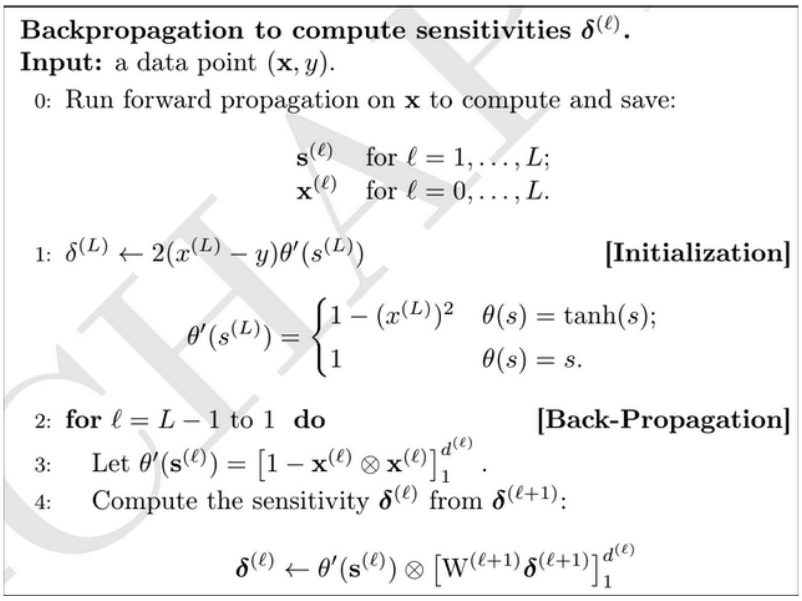
We now have found a way to compute $\delta^{(l)}$ from $\delta^{(L)}$. This means that, to find all the sensitivity vectors, we first need to find $\delta^{(L)}$:

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

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 $\boldsymbol{\delta}^{(\ell+1)}$:
           $\boldsymbol{\delta}^{(\ell)} \leftarrow \theta'(\mathbf{s}^{(\ell)}) \otimes \left[\mathrm{W}^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}\right]_1^{d^{(\ell)}}$
```

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

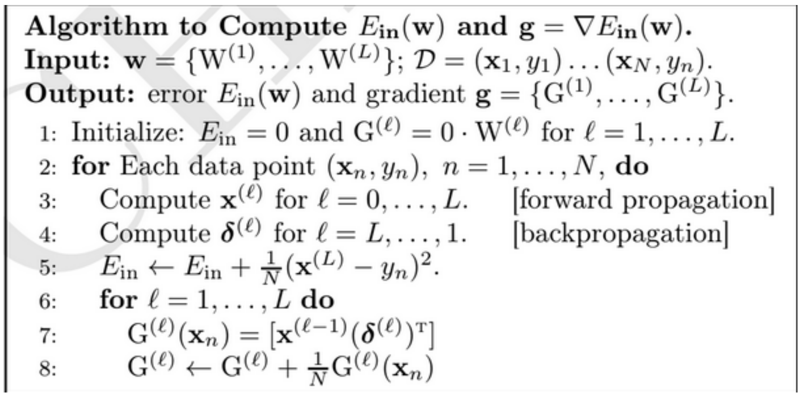
Now, with forward propagation and backpropagation, the neural network is able to find the gradients:

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.

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 it because of some bugs that I needed to sort out to make



```
Algorithm to Compute $E_{\text{in}}(\mathbf{w})$ and $\mathbf{g} = \nabla E_{\text{in}}(\mathbf{w})$.
Input: $\mathbf{w} = \{W^{(1)}, \ldots, W^{(L)}\}$; $\mathcal{D} = (\mathbf{x}_1, y_1) \ldots (\mathbf{x}_N, y_N)$.
Output: error $E_{\text{in}}(\mathbf{w})$ and gradient $\mathbf{g} = \{G^{(1)}, \ldots, G^{(L)}\}$.
1: Initialize: $E_{\text{in}} = 0$ and $G^{(\ell)} = 0 \cdot W^{(\ell)}$ 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:         $G^{(\ell)}(\mathbf{x}_n) = [\mathbf{x}^{(\ell-1)}(\boldsymbol{\delta}^{(\ell)})^{\mathrm{T}}]$
8:         $G^{(\ell)} \leftarrow G^{(\ell)} + \frac{1}{N} G^{(\ell)}(\mathbf{x}_n)$
```

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

Figure 5: Exemplos de imagens do MNIST

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:

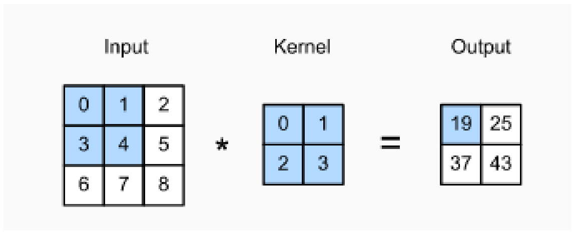


Figure 6: Convolution operation example

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 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:

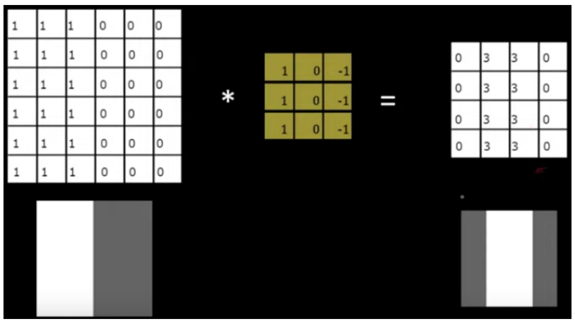


Figure 7

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:

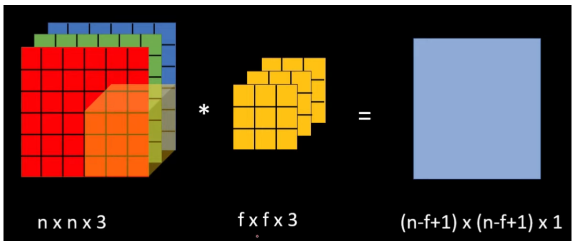


Figure 8

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.

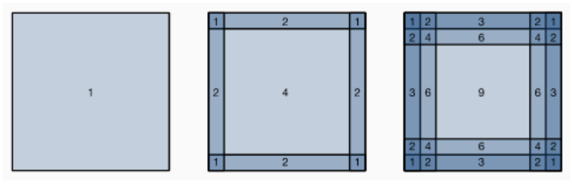


Figure 9: Pixel utilization for convolutions of size 1×1, 2×2, and 3×3, respectively

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

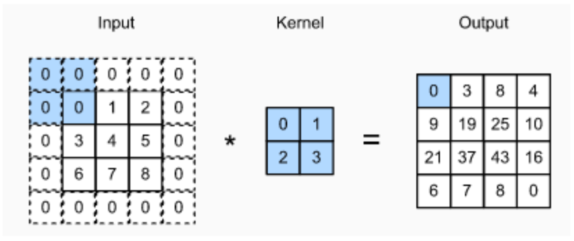


Figure 10: Convolution with padding

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

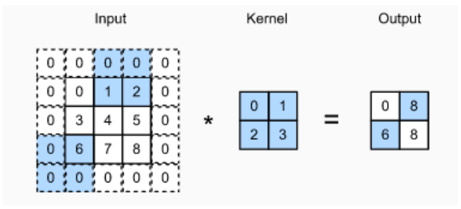


Figure 11: Cross-correlation with strides of 3 and 2 for height and width, respectively

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

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:

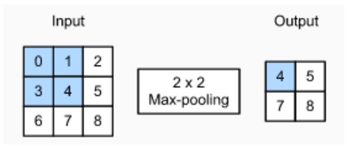


Figure 12

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.

2 Literature Review

The idea behind a literature review is to understand how people in the same area have done things over time. Since this work is about Face Recognition using Deep Learning, I focused on papers that cover this topic. In this section I describe in a succinct way what I understood and my line of thoughts while I read each one of them.

2.1 DeepFace: Closing the Gap to Human-Level Performance in Face Verification

This paper is particularly important because, back in 2015, it achieved a high levels of accuracy and explored new approaches when it comes to facial recognition

2.1.1 Abstract

This paper goes through the creation of a face recognition model that was able to reach an accuracy level of 97.35% on the Labeled Faces in the Wild (LFW) data-set. The deep network used has nine layers and involves more than 120 million parameters. The training happened based on an identity data-set composed of 4 million face images. Also, it goes through the process of face alignment and representation, they used a 3D face modeling in order to apply a piecewise affine transformation.

2.1.2 Introduction

The model created is heavily dependent on a very rapid 3D alignment step, the network architecture is based on the assumption that once the alignment is completed, the location of each facial region is fixed at the pixel level. This approach makes the net work without the need for more convolution layers as is done in many other networks.

In summary, the contributions stated in this article are: (i) the development of a deep neural net (DNN) architecture and learning method that leverage a very large labeled data-set of faces in order to obtain a face representation that generalizes well to other data-sets, (ii) an effective facial alignment system based on 3D modelling.

2.1.3 Face Alignment

The alignment here is based on using fiducial point detectors to direct the alignment process. The process of detecting the fiducial points is done by using a relatively simple detector, but it is applied in several iterations so that the output can be refined. At each iteration, fiducial points are extracted by a Support Vector Regressor (SVR) trained to predict point configurations from an image descriptor. The image descriptor used is based on the LBP Histograms. By transforming the image using the induced similarity matrix T to a new image, the fiducial detector can be used again on the new feature space an then refine the localization.

For the 3D alignment and frontalization, a similar technique is used, but it is more complex. I wasn't able to understand it.

But this paper makes it clear that, to achieve good results for facial recognition problems, I will need to find a away to align the faces well and, if possible, apply a frontalization method.

2.1.4 Reprе́sentation

The overall architecture is presented in this part of the paper. It is extremely worth reading this part because it explains the logic behind the choice of each layer. I will definitely come back to this later, I won't put it in the diary because it would be better to just come back to the original source to remember things. Also, this part of the paper was my first time getting in touch with the Locally Connected Layers, they seem to be important for facial recognition problems.

The face representation learned by this model in a data-set so large like the SFC enables it to be used in other data-sets. That is also my goal in this work, create an app that is able to recognize people in the wild, not only those described in a tiny data-set. To achieve this, I'll need to learn a good representation for faces that enables another model to evaluate if two representations belong to the same person.

2.1.5 Verification Metric

They used the weighted differences and Siamese network to be able to state if two instances belong to the same class. This is the first time I read about Siamese Networks, I will read about them later because that is probably what I'm going to use in my project. Also, I'll need to do my research and see if it is the best that we can use for this task

2.1.6 Experiments

The training of the representation was done on the SFC as a multi-class classification problem. They talk about some hyperparameters and the learning process, but it shocks me that it took 3 days to reach the end of the learning process.

Another important thing is that they did test different models and showed how the train/test proportion affects the end result. Also, they showed how a simplification of the model affected the performance. In addition, they showed how the frontalization and alignment are both important to reach high accuracy on the LWF data-set.

2.2 Learning a Similarity Metric Discriminatively, with Application to Face Verification

This paper was mentioned in the previous one and talks about the use of a siamese architecture in the field of Face Recognition. It is an old work, it was published back in 2005.

2.2.1 Abstract

The idea is to learn a function that maps input patterns into a target space such that the L1 norm in the target space approximates the "semantic" distance in the input space. That is, the learning process minimizes a discriminative loss function that drives the similarity metric to be small for pairs of faces from the same person, and large for pairs from different persons.

2.2.2 Introduction

The approach used in this paper, that is distance-based methods, is intended to be used to solve classification problems that have a lot of classes but not so many examples for each class in the data-set. The solution presented in this paper is to learn a similarity metric from data. This similarity metric can later be used to compare or match new samples from previously-unseen categories (e.g. faces from people not seen during training).

2.2.3 The General Framework

In this part it is explained the origin of the loss-function that they wanted to minimize. Some proofs are made at this part too. The main idea is: we use a convolutional network to generate a representation of an image, than we need to train the model to make this distance between two representations become shorter if the representations belong to the same pair, and become longer if the representations belong to different classes.

2.2.4 Experiments

In this part they explained how from where they got the data-set and how they split it and combined the images to create pairs for training and testing. Also, they showed in this part the architecture of the model, including the convolutional network used. This part is good to understand more of siamese networks, probably I am going to do more research on these, one bad thing about this architecture is that it takes longer to be trained, since we need to do the forward propagation and the backpropagation two times, one for each image within the pair.

2.3 Using Siamese Networks with Transfer Learning for Face Recognition on Small-Samples Datasets

I find this paper particularly important because it uses transferlearning as a way to achieve good results for face recognition. In the the original propose for this research project, we brought up the possibility to use transferlearning.

2.3.1 Abstract

The idea behind this paper was to use a siamese architecture to create features for a pair of images and then state if a given pair of two images belong to the same person. They use transfer learning here and claim that the model has a performance comparable to those models that were trained using larger data-sets.

2.3.2 Introduction

The introduction begins by stating the differences between face verification and face identification, the first is the task to state if two photos belong to the same person, the second is the task to state what is the person in a given photo. In this paper, as said before, they implemented a model capable of addressing the first problem.

2.3.3 Face recognition using siamese architecture and transfer learning

Transfer learning is a popular approach in machine learning, specially in deep learning, in which pre-trained models are used as the beginning point of solving several computer vision problems. This method is well suited to problems that are constrained to the small availability of data for training. In this work, the pre-trained model used was the VGG-16.

They had quite a good idea. They froze the first 4 convolutional blocks, deleted the fully connected layers and put theirs in place, and then did the fine tuning of the model. The loss function used is the contrastive loss function, which the idea behind it is to make the distance between representations of the same class shorter.

2.3.4 Experimental results and evaluation

They mention an algorithm suggested by another paper, which I will definitely look later, that creates the pairs used in the training and testing process. They reached a performance of 95.62%, which was the best for models trained using small data-sets. It was not able to outperform the CosFace model, which was trained using a huge data-set.

2.4 Recent Advances in Deep Learning Techniques for Face Recognition

2.4.1 Abstract

This paper covers recent techniques and make a comprehensive analysis of various FR systems that use Deep Learning. It summarizes more than 171 recent contributions from this area. Seems like it will give me a good insight for the problem I'm trying to adress in this Reasearch Project.

2.4.2 Introduction

We can divide in the task of FR into different subtasks, which are the detection of the face, the normalization of the face (like alignment and frontalization), feature extraction, and then feature matching to identify the person. Nowadays, CNNs based methods are primarily used for feature extraction. Taking advantage of these feature extraction methods, we can use traditional machine learning methods to do the face recognition.

2.5 FaceNet: A Unified Embedding for Face Recognition and Clustering

This is an important work because it shows a way to learn a good face representation for classification problems. Definitely will use it later.

2.5.1 Abstract

In this paper it is presented a system, called FaceNet, that directly learns a mapping from directly correspond to a measure of face similarity. Therefore, the feature vectors created could be used to solve face recognition problems. To train the model they used triplets of images. They achieved an accuracy of 99.63% on the LFW data-set.

2.5.2 Introduction

Previous works based on CNNs' embeddings use a classification layer trained over a set of known face identities and then take an intermediate bottleneck layer as a representation of used to generalize FR beyond the set of identities used in training. The problem here is that we need to hope that the intermediate bottleneck generalizes well for other problems, which might not always be the case. One interesting thing about this paper is that they didn't do some form of 2D or 3D alignment, they only did changes in scale and translation.

2.5.3 Method

This work is based on the triplet loss function, which has the goal to make embeddings of the same person closer in distance and embeddings of different persons to be far in distance. Recommend the reader to see the triplet loss showed on the paper, it is important. Also, to find good triplets for training is important, since it makes the learning process faster. It is highly recommended to read the paper to understand how these triples of images are formed.

2.5.4 Experiments

This part is particularly important to read because it shows how different architectures and dataset sizes can affect the model's performance. It is in fact impressive the amount of face images we need to achieve such high performance. Also, they explored other metrics of the model that I can use as an inspiration for my research project.

2.6 Transfer learning on convolutional neural networks for face recognition

This is a small paper that may give me some hints for the first implementations.

2.6.1 Abstract

In this work, an automated face recognition method using CNNs and transfer learning is proposed. The pre-trained model used was VGG-16, which was trained on huge ImageNet database. They used two publicly available databases of face images, the Yale and the AT&T.

2.6.2 Proposed Method

It is said that VGG-16 was trained on more than 1 million images of different classes, I wonder if it doesn't disturb the learning process for facial classification. The final architecture uses 46 layers in total, with ReLU as the activation function and softmax as the final layer. I read a paper before that used a similar approach, but it froze the first layers to keep the early representations of the first model.

2.6.3 Experiments and results

First thing to notice is that those are very tiny datasets, both with less than 500 images. This might explain why they managed to achieve such good accuracy in both datasets. For the Yale, the accuracy was 98.7%, and, for the AT&T, 100%.

2.6.4 Conclusion

At this point of literature review, I think I should try to use transfer learning for the first experiments. I won't be able to use big databases in the beginning, so that might be the best approach for now. My advisor told me that using transfer learning has already become a trend in the area of machine learning and deep learning, so it is not unusual and it can achieve good results.

2.7 Improving performance in small datasets via pre-trained architectures based on VGGFace and VGGFace2 datasets

This is a special work because it was written by people of University of São Paulo. The idea here is to use transfer learning to do facial recognition with a low number of sample examples. They explored two versions of the ResNet architecture and the VGG-16 architecture, all models pre-trained on the VGGFace data base.

2.7.1 Experiments and Conclusions

The first experiments were done on the Labeled Faces in the Wild data-set. They used this data base because it has too few images to train a state of the art facial recognition model, so it is needed to use transfer learning to achieve good accuracy on this data-set. They also explored the fact that the number of images available for each class in the LFW is very inconsistent, with the majority of the identities having only one image for training.

Some pre-processing was done, but it was not difficult. The researchers only changed the mean and the standard deviation. Good accuracy measures were achieved, specially for those models that were pre-trained on face data sets. I believe that I will probably use transfer learning in my personal work, intend to talk to my advisor soon.

3 Practical Work and Results

At this point I have already obtained a substantial amount of theoretical knowledge in the deep learning field. I have improved my abilities to read and do research. Also as a result, I created my first deep learning model trained to address the face recognition problem. Not only this model, but also other versions of problems and codes are available at my github repository: https://github.com/Kaiqoliver/IC

3.1 SiameseVGGFace 1.0

This is my first model for face recognition, it was done with the use of transferlearning. More precisely, this model: https://github.com/ma7555/keras-vggface. It is the pretrained model used in the VGGFace paper of 2015. It used millions of face images in the learning process, which makes it a good model to use since I am at the first part of the research project.

About the architecture, the idea was to build a siamese neural network that is able to infer if two face images belong to the same person or not. In the code you are able to take a closer look at the model, I added two more fully connected layers, the first with 512 nodes and the second with 128 nodes, to the original model and froze the original layers (to keep the original feature extractor). After the feature vector is generated for each image, the model measures the distance and then uses the cross entropy loss function to do the learning process. O optimizer used was Adam.

3.1.1 Metrics

About metrics, with 10 epochs I was able to achieve 69% of accuracy in the LFW pairs data set that is available at the sckitlearn library. With 50 epochs, I was able to reach reached 74%. It is in fact a good beginning, but I was expecting it to do better. Maybe my model isn't complex enough, or I need to run it for more epochs, like a hundred. But my main worry is that, maybe, the problem is the data set training size, it is composed of only 2200 pairs of images, that might not be enough.

3.1.2 Difficulties

My main difficulty here was to be find a way to load the pretrained model. The original repository has a code that was working only for previous Keras and Tensorflow versions, which was a problem since I wanted to use more recent version. Also, to run the experiments and to do the learning I used Google Collab, which has limitations in terms of RAM and, therefore, often collapsed during the running of the program.

4 Future Steps

The next steps for the project will be trying to improve the SiameseVGGFace and see how much power it actually has. To do so, I have already asked my advisor to give me access to more powerful machines available at the Vision Lab. After this, exploring new architectures, design patterns, and the work pipeline used in data science will be the new goal.

5 Hours Report

1. Theory (September - October)

Neural Networks, CNNs, Tensorflow.

Total: 38h

2. Literature Review (October)

Studying and reading the papers of the area.

Total: 18h

3. Practice With Deep Learning (October - November - December)

Neural Network from scratch, SiameseVGGFace1.0

Total: 44h

Final Amount: 100h

References

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