Implementing from scratch an easy-to-use Deep Learning library and experimenting with the internal neural networks pipeline

Abel García

Abstract — Deep Learning (DL) is acquiring more and more applications in the industry, but not all professionals who need it have knowledge on the topic. For this reason, I have developed my own DL library for Python that includes multilayer perceptrons and convolutional neural networks, completely from scratch and with the aim of being easy to use, as I have shown when it has been used by some students from my old high school. Furthermore, keeping this objective in mind, I have also developed an evolutionary meta-learning algorithm to automatically find the optimal combination of hyperparameters for the input data, which achieves better results than the *RandomSearch* and *GridSearch* techniques. On the other hand, taking advantage of the fact that I implement neural networks from scratch, this article explains its foundations at a theoretical and mathematical level, as well as some interesting comments about its real implementation. Finally, some internal experiments are included that could not be easily performed with high-level libraries, such as the technique that I propose for standardization of values between layers, which allows using high learning rate values without the inconveniences that this usually implies.

Index Terms *— Artificial neural networks, Automated machine learning, AutoML, Computer vision, Convolutional neural network, Deep learning, Image classification, Machine learning, Meta learning, Multilayer perceptron, Python*

Resumen — El *Deep Learning* (*DL*) está adquiriendo cada vez más aplicaciones en la industria, pero no todos los profesionales que lo necesitan tienen conocimientos sobre el tema. Por ello, he desarrollado mi propia librería de *DL* para *Python* que incluye perceptrones multicapa y redes neuronales convolucionales, completamente desde cero y con el objetivo de que sea fácil de usar, como he demostrado cuando la han utilizado algunos alumnos de secundaria de mi antiguo instituto. Además, manteniendo en mente este objetivo, he desarrollado también un algoritmo evolutivo de meta-aprendizaje para encontrar automáticamente la combinación de hiperparámetros óptima para los datos de entrada, y que consigue resultados mejores que las técnicas de *RandomSearch* y *GridSearch* analizadas. Por otro lado, aprovechando que implemento las redes neuronales desde cero, este artículo repasa sus fundamentos a nivel teórico y matemático, así como algunos comentarios interesantes sobre su implementación real. Finalmente, se incluyen algunos experimentos internos que no podrían ser fácilmente realizados con librerías de alto nivel, como la técnica que propongo de estandarización de valores entre capas, que permite utilizar tasas de aprendizaje elevadas sin los inconvenientes que esto suele implicar.

Palabras clave *— Aprendizaje automático, Aprendizaje computacional, Aprendizaje profundo, AutoML, Clasificación de imágenes, Metaaprendizaje, Perceptrón multicapa, Python, Red neuronal artificial, Red neuronal convolucional, Visión por computador*

Repository of the project: *www.github.com/abel-gr/AbelNN*

– – – – – – – – – – ◆ – – – – – – – – – –

1. INTRODUCTION

The field of Deep Learning (DL) is on the rise due to the high value it is acquiring in our society and even in the future that is already beginning to form. It has many applications in medicine, industry, space exploration, and countless other relevant fields. Specifically, in recent years neural networks are being used a lot for computer vision problems with really impressive results. Practically all the people who solve these problems with images use libraries like *TensorFlow* [1] or *Pytorch* [2] that already have neural networks implemented. They are frameworks that allow you to create DL models quickly if you have experience.

However, neural networks are beginning to be used more and more in different sectors of the industry, by people who have little or even no knowledge of artificial intelligence. This supposes slowness on the part of the workers to use the previously mentioned frameworks.

Moreover, there are companies that want to use these technologies and do not have the resources to hire an engineer with this knowledge, owing to the high demand for computer engineers nowadays. Consequently, my objetive is to implement a Deep Learning library from scratch that allows customizing the models in just a single line of code, without the need to specify complex values such as layer sizes and not even the network architecture if you do not want to. In other words, internally, my library is in charge of calculating all the forms of the matrices and all the values so that the user has a customized network in a single line of code. To that end, I have also implemented an AutoML module that is responsible for obtaining the best model so that even unexperienced users do not have to specify a single parameter of the networks. They only need to pass the data they want to use. To be able to achieve it, I have to get into all the theoretical and mathematical concepts on which neural networks are based. Furthermore, I will take advantage of the fact that I am implementing everything from scratch to perform various internal experiments on networks, that cannot be done if you work with current libraries.

- *E-mail: abel.gr@outlook.com*
- *Specialization: Computational Science*
- *Tutor: Dr. Felipe Lumbreras (Dept. Ciències de la Computació)*
- *Course: 2021/22*

2. OBJECTIVES

The main objective of the project is to implement my own *Python* library for Deep Learning from scratch, relying only on *Numpy* for array calculations. Specifically, so as to achieve that my module will be useful to users, I will implement a multilayer perceptron (MLP) and a convolutional neural network (CNN) in my library from scratch, and I will also prepare image classification usage examples with both.

Secondly, one of the main milestones is also to make my library easy to use even by inexperienced users. To that end, another critical objective is to implement an AutoML module inspired by some state-of-the-art meta-learning techniques to automatically find the best model for the data specified by the users, so that they do not need to specify any hyperparameters.

Thirdly, to prove that my library is easy to use, I will write detailed documentation about it, and then I will go to my old high school to let the students consult it to be able to classify images with my module.

Finally, another reason to implement everything from scratch is that it allows me to modify the internal neural network pipeline and procedures, which could not be done with other libraries. As a consequence, another important objective is to perform internal experiments with the MLP and ConvNet of my library. These experiments will consist of the internal manipulation of the networks to plot and analyze how these changes affect their performance when training and classifying images.

3. STATE OF THE ART

3.1. Deep learning frameworks

Before implementing my own deep learning library, I need to check which ones are the most used currently, and then what are their strengths and weaknesses to take everything into account in my implementation.

1. **Tensorflow** [1]
   - GPU support.
   - Large documentation.
   - Efficient low-level tensor operations
   - Complex for inexperienced users.
2. **Pytorch** [2]
   - GPU support.
   - Community support.
   - Rapid prototyping.
   - Complex for inexperienced users.
3. **Caffe** [3]
   - Fast executions.
   - Little customization compared to other frameworks.
4. **scikit-learn** [4]
   - The easiest to use among the 4.
   - Does not implement ConvNets.

Although it is true that the first 3 frameworks are easy to use if you are experienced, they have a steep learning curve, which slows down people who are not in the computer science sector but still need to make use of DL. That is where my library will highlight. The fourth library is quite easy to use, but it does not implement CNNs. The first 3 frameworks allow you to define your own models that include fully connected networks and convolutional neural networks, so I allow to implement those architectures in my library as well. Besides, they allow regularization techniques that I have also implemented from scratch in my library, such as dropout, batch, and several initializations of weights such as *He* and *Xavier* [5]. Apart from all this, I will also include predefined models in my library to simplify the task for inexperienced users, which these first 3 modules do not do by default. In this way, my library will be much easier to use than the frameworks that I have mentioned in this section.

3.2. Meta-learning

Meta-learning consists of observing the different learning outcomes of a model, in order to improve this learning. That is usually called *learning to learn*. To this end, state-of-the-art meta-learning algorithms try to minimize a specific learning metric even with a small amount of data [6]. Accordingly, automated machine learning or AutoML consists of using meta-learning techniques to automatically determine the best model for specific data, without the need to specify any hyperparameters.

Some models (e.g., Learning to learn by gradient descent [7]) use a gradient descent with a meta-learning rate that takes small steps to improve results, being a process similar to the learning that we are used to in neural networks. It is also common to perform meta-learning with Hierarchical Bayesian Inference, but it can become highly computationally complex when there are many hyperparameters [6].

Alternatively, Zoph and Le propose in their paper [8] the use of a Recurrent Neural Network (RNN) to generate model descriptions and train the RNN with reinforcement learning to maximize accuracy. In fact, the approach of using reinforcement learning in meta-learning is highly common.

Finally, a very interesting technique to perform meta-learning was proposed in the article *Large-Scale Evolution of Image Classifiers* [9] based on a population of evolving models: At each step, two models are randomly selected from the population and are compared. The worst of the pair *dies*, and the best is selected as the *parent* and reproduces, applying a mutation, that is, modifying one of its hyperparameters at random.

4. METHODOLOGY AND DEVELOPMENT PROCESS

With the aim of successfully completing all the tasks of the project, it has been developed following an agile methodology through 17 weekly sprints, which allowed the project to be finished iteratively throughout all its phases.

4.1. Multilayer perceptron

To start the development of my library, the first step is to have one of the most known elements of deep learning, and it is the multilayer perceptron. As it can be deduced, a multilayer perceptron or MLP consists of multiple layers, each of which contains a number of perceptrons. In Fig. 1 you can see the scheme of a 3-layer MLP. Neurons of the input layer are red, those of the hidden layer are blue and the only one in the output layer is green.

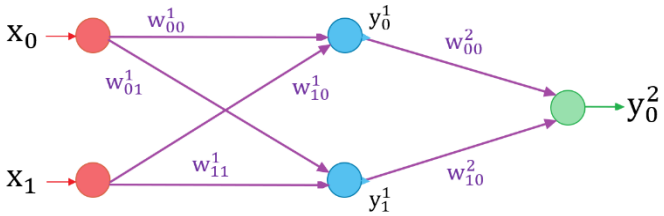

Fig. 1. Scheme of a multilayer perceptron

The values of the train subset of our dataset enter through the neurons of the input layer. These neurons produce an output, which will be the input of the neurons of the next layer, and so on until the end. This process is known as **feedforward**. Once the end is reached, the error made is calculated and it is propagated backwards to try to minimize it. This second process is known as **backpropagation**.

4.1.1. Feedforward

Each neuron calculates its output from the input of all neurons of the previous layer. That is calculated in two steps. The first step is called net input (net_j^m) of the neuron j in layer m. Net input consists of the weighted sum between the neuron inputs x_i and a weight w_{ij}^m assigned to each input to the neuron j (from neuron i). A mathematical function f_s (activation function) must be applied to this net input in order to generate the output y_j^m for that neuron (equation 1). This is done to introduce non-linearities in the network. Otherwise, it would be like having a single layer. This activation function must be differentiable for the reason that we will see during the backpropagation. n^{m-1} refers to the number of neurons in layer $m-1$.

$$y_j^m = f_s(net_j^m) = f_s\left(\sum_{i=0}^{n^{m-1}} w_{ij}^m x_i\right) \tag{1}$$

4.1.2. Backpropagation

Since we are doing supervised learning, we can calculate the error that the network has made in the output layer by comparing it with the groundtruth (GT). Calculating this error allows us to modify the weights, so that we will end up obtaining the combination of weights that give the best approximation to the function that models our training data.

If we only had a single layer, we could do this by applying a process known as Gradient Descent. To understand this process, we must remember that the derivative of a function can be used to measure the speed with which the output value changes as a function of an input. If we calculate the partial derivative of the cost function based on its inputs, we obtain a vector called Gradient. We must imagine the cost function as a mountain in which we want to descend to its minimum and the gradient as the vector that points to the steepest direction, consequently, we will adjust the weights in the opposite direction. The problem is that by having multiple layers, if we update the weights of the last layer, this will cause a conflict since we would be changing weights that depend on previous layers. That is, there is a functional dependence between the values of the different layers. To solve this, the backpropagation process consists of a variant of the Gradient Descent technique that solves this problem using the derivative chain rule. The first step to update that weights w is to know the error E_p our neural network made for the p-th element of the training set according to each weight, that is, we must calculate the derivative of the error with respect to these weights like in equation 2.

$$\frac{\partial E_p}{\partial w_{ij}^{m-1}} \tag{2}$$

As we have seen, this is the gradient, which multiplied by a learning rate α that has to be tuned, will be the value that will allow us to adjust that weight in the correct direction. That adjustment is done with the equation 3.

$$w_{ij}^m(t+1) = w_{ij}^m(t) + \alpha \cdot \frac{\partial E_p}{\partial w_{ij}^{m-1}} \tag{3}$$

In the previous equation 3 we find a small obstacle and that is that we cannot directly relate the error made by a neuron and its weights, since the error E_p (eq. 4) can only be calculated from the output y and the groundtruth d. m_o is the output layer.

$$E_p = \frac{1}{2}\sum_{j=1}^{n^m}\left(d_j^p - y_j^{m_o}\right)^2 \tag{4}$$

We must go to a formula in which the weights appear, so we use the formula of the net input that we had already seen in equation 1. Consequently, in order to calculate the derivative of the error of the output layer neuron with respect to the weights, we must apply the chain rule, as in equation 5.

$$\frac{\partial E_p}{\partial w_{ij}^{m-1}} = \frac{\partial E_p}{\partial y_j^{m-1}} \cdot \frac{\partial y_j^{m-1}}{\partial net_j^{m-1}} \cdot \frac{\partial net_j^{m-1}}{\partial w_j^{m-1}} \tag{5}$$

The result of deriving and applying the chain rule to the formula 5 is shown in equation 6. Notice that f_s' is the derivative of the activation function.

$$\frac{\partial E_p}{\partial w_{ij}^{m-1}} = -\left(d_j - y_j^m\right) \cdot f_s'\left(net_j^m\right) \cdot y_j^{m-1} \tag{6}$$

To simplify the calculations, we will convert a part of the previous equation into what we will call the error term δ resulting in equation 7.

$$\frac{\partial E_p}{\partial w_{ij}^{m-1}} = \delta_j^m \cdot y_j^{m-1} \tag{7}$$

Now we can substitute the partial derivative that we have calculated into the weight update formula, and we obtain equation 8, that allows us to modify the values of the weights of each neuron of the last layer to the right direction.

$$w_{ij}^m(t+1) = w_{ij}^m(t) + \alpha \cdot \delta_j^m \cdot y_i^{m-1} \tag{8}$$

The same chain rule process is applied for hidden layers so that their weights can also be updated, the only change is that we cannot obtain the derivative of the error with respect to the output because we are no longer in the output layer. Hence, we must apply the chain rule again, as shown in formula 9.

$$\frac{\partial E_p}{\partial y_j^{m-1}} = \sum_{i=1}^{n^m} \frac{\partial E_p}{\partial net_i^m} \cdot \frac{\partial net_i^m}{\partial y_j^{m-1}} = -\sum_{i=1}^{n^m} \delta_i^m \, w_{ij}^m \tag{9}$$

If we substitute again in eq. 5 the values of the derivatives, we can calculate our error term δ_j^{m-1} (eq. 10) that will be used in the formula 8 again to also update the weights of the hidden layers.

$$\delta_j^{m-1} = \left(\sum_{i=1}^{n^m} \delta_i^m \, w_{ij}^m\right) \cdot \left(f_s'\left(net_j^{m-1}\right)\right) \tag{10}$$

I have used the matrices from the *Numpy* library to implement my MLP, so that I have been able to calculate the previously described mathematical operations in a direct and vectorized way, thus allowing fast execution. The implementation of the different activation functions and regularizations have also been implemented through *Numpy* object operations.

4.2. Convolutional Neural Network

Current libraries also allow defining ConvNets that are widely used for image classification due to their good results. They have a first part of convolutions that are responsible for the automatic extraction of features, and a second part that is fully connected to perform the classification or regression. Accordingly, I need to implement the feedforward and backpropagation of the convolutional layers from scratch, to also allow its use in my module.

4.2.1. Feedforward

The key to convolutional neural networks is, as their name says, convolutions. The convolution output O for a point in the image X at position (i, j) is a mathematical operation defined by the following formula 11:

$$O_{i,j} = \sum_{k,l=0}^{k,l=K} X_{k,l} \cdot F_{k,l} \tag{11}$$

To put it more simply, convolving an image X is to calculate a dot product between a subregion of our image and a kernel (or mask) matrix [10] for each point of the image. If we use what is called $stride = 2$, we will move our subregion every 2 positions of the image, resulting in an output half the size of the input. Must be mentioned that before convolving we must rotate the kernel 180 degrees since, if not, we will be doing a correlation. In CNN this is something that does not matter since it is the network itself that will be in charge of finding the appropriate values of the mask matrix through its learning.

In the case of a convolutional neural network, this process is done with multiple kernels, which are called filters. The ReLu function is commonly applied to the result of each convolution to rectify the negative numbers and thus introduce non-linearities. These convolutions with their activation function are the feedforward of the convolutional layers. In the case of a ConvNet, the result of the last convolutional layer passes to a fully connected network like that of an MLP (see Fig. 2).

4.2.2. Backpropagation

As we have seen, convolution is a mathematical operation that allows filtering an image with values from a matrix called kernel. The key to why it is such a powerful operation is based on two aspects. Firstly, when we convolution the input images with multiple filters, we obtain new filtered images that describe certain features that the original image has. The second key aspect is that this process occurs in multiple layers, so the first convolutional layers detect simple features, and the last ones are capable of describing more complex shapes [10]. But initially all the kernel matrices in the network are randomly initialized, so the features described do not provide us with information. To solve this, learning must not only occur in the fully connected layers but also in the convolutional layers. As a consequence, it is necessary to calculate the error that the network output produces with respect to the value of the filters applying the chain rule (equation 12), since the error is committed in the next layer, and we cannot relate it directly to the filters.

∂: Derivative E_p: Error of layer p F_p: Filters of layer p O_p: Output of layer p	$\frac{\partial E_p}{\partial F_p} = \frac{\partial E_p}{\partial O_p} \cdot \frac{\partial O_p}{\partial F_p}$ (12)

What interests us firstly is to obtain the derivative of the output O regarding the filter F to be able to substitute it in the equation 12. To exemplify this, we want to calculate the convolution value of the position (1,1) of our image with a kernel of size 2x2. If from the equation 11 we expand the sum for our example, we obtain the following equation 13:

$$O_{1,1} = X_{0,0} \cdot F_{0,0} + X_{0,1} \cdot F_{0,1} + \cdots + X_{1,1} \cdot F_{1,1} \tag{13}$$

From equation 13 we can already calculate the derivative of the output O from the filter F in eq. 14, because the only variable is the filter and everything else is constant.

$$\frac{\partial O_{1,1}}{\partial F_{0,0}} = X_{0,0} \quad \cdots \quad \frac{\partial O_{1,1}}{\partial F_{1,1}} = X_{1,1} \tag{14}$$

Now we can substitute this derivative in the equation 12, and result is shown in formula 15:

$$\frac{\partial E_p}{\partial F_{i,j}^p} = \sum_{k,l=0}^{k,l=K} \frac{\partial E_p}{\partial O_{k,l}^p} \cdot X_{(i-k),(j-l)}^p \tag{15}$$

If we expand the sum of equation 15, as a result we get formula 16 which corresponds to a convolution between the image X and the derivative of the error with respect to the output [11].

$$\frac{\partial E_p}{\partial F_{1,1}^p} = \frac{\partial E_p}{\partial O_{0,0}^p} \cdot X_{0,0}^p + \cdots + \frac{\partial E_p}{\partial O_{1,1}^p} \cdot X_{1,1}^p = conv\left(X, \frac{\partial E_p}{\partial O_p}\right) \tag{16}$$

What we need now is to obtain the value of the error regarding the output. For the last convolutional layer, it is simply the error that has propagated from the first fully connected layer.

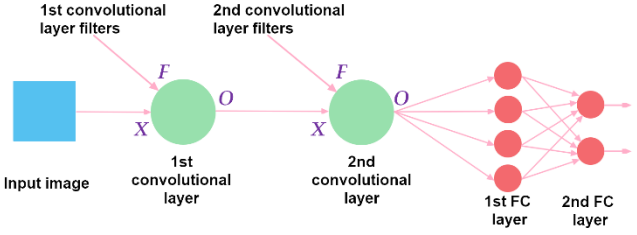

Fig. 2. Scheme of a CNN with 2 conv layers and 2 FC layers

But for the rest of the convolutional layers, it is not so simple. As we can see in Fig. 2, the output O of an intermediate convolutional layer corresponds to the input of its next layer. That means that we can calculate the derivative of the error with respect to the output O of a layer p, from the input X of the next layer, expressed in the equation 17 in which layer $p+1$ is the one to the right of layer p.

$$\frac{\partial E_p}{\partial O_p} = \frac{\partial E_{p+1}}{\partial X_{p+1}} \tag{17}$$

Consequently, we apply the chain rule again to obtain the derivative of the error with respect to X, as in the equation 18.

$$\frac{\partial E_p}{\partial X_p} = \sum_{k,l=0}^{k,l=K} \frac{\partial E_p}{\partial O_{k,l}^p} \cdot \frac{\partial O_{k,l}^p}{\partial X_{k,l}^p} \tag{18}$$

In eq. 18 we already have the error regarding the output O, because it is from the same layer. We only need to calculate the derivative of the output O with respect to X by deriving eq. 13 again but this time with respect to X (eq. 19). Then, after substituting in eq. 18 we obtain equation 20. That operation is a full

convolution between the filter rotated 180 degrees and the propagated error [11]. Finally we update the value of the filters with equation 21.

$$\frac{\partial O_{1,1}}{\partial X_{0,0}} = F_{0,0} \quad \cdots \quad \frac{\partial O_{1,1}}{\partial X_{1,1}} = F_{1,1} \tag{19}$$

$$\frac{\partial E_p}{\partial X_p} = \sum_{k,l=0}^{k,l=K} \frac{\partial E_p}{\partial O_{k,l}^p} \cdot F_{k,l}^p = FullConv\left(rotate_{180}(F_p), \frac{\partial E_p}{\partial O_p}\right) \tag{20}$$

$$F_{t+1}^p = F_t^p - \alpha \cdot \frac{\partial E_p}{\partial F_t^p} \tag{21}$$

Having seen the mathematical and theoretical part of backpropagation, we can already mention some interesting details about its real implementation. First of all, as I use the *stride* technique to reduce the size of the output matrices of the convolutions during the feedforward process and thus reduce computational cost in the subsequent layers, I must add extra code to backpropagation, to adapt it to this technique. Simply what we must do is to increase the size of the error matrix with respect to the output so that it has the same shape that it would have had if the *stride* had been equal to 1. To do this, a very common solution is to add columns and rows of zeros to the matrix, as illustrated by the example in Fig. 3. This technique is called zero-interweave [12].

```
[[1 2]        [[1 0 2 0]
 [3 4]]        [0 0 0 0]
               [3 0 4 0]
               [0 0 0 0]]
```

Fig. 3. Left: Example of error with respect to the output for stride equal to 2. Right: said matrix after applying zero-interweave.

After mentioning that detail, I can already comment that my first approach to implement the mathematical process of backpropagation of the convolutional layers from scratch was to call the convolution function that I implemented for the feedforward, modifying the input parameters to follow the previously explained theory, as can be seen in Fig. 4.

```
f_rotated = np.flip(self.filtersValues[i], 0)
f_rotated = np.flip(f_rotated, 1)

# dE/dF
error_by_filter = self.conv_filters(X, error_by_output,
                                    relu=False, stride=1,
                                    mode='valid')
# dE/dX
error_by_x = self.conv_filters(f_rotated, error_by_output,
                               relu=False, stride=1,
                               mode='full')
```

Fig. 4. First convolutional backpropagation approach

However, this means performing two n-dimensional convolutions, one after the other, for each convolutional layer, which is very slow. So I decided to try merging both convolutions into a single nested double *for* loop, which can be seen in Fig. 5. In order to perform this optimization, I had to implement the convolutions in a way that is quite different from the usual one, but that it continue to perform the same mathematical calculation that I already defined in this section. I carried out this implementation in reverse of my approach for feedforward. While that code was iterating over the input image, during the backpropagation it is now iterating over the kernel, which is, for both convolutions, the error with respect to the output ($\partial E_p / \partial O_p$). This requires making a different selection of the area to be calculated on the two inputs, and in this case the result is being accumulated in each iteration until the final result is obtained, as you may notice in Fig. 5. It is a much less intuitive process, but it allowed me to reduce the total processing time so that each epoch was 10 times faster than using the common convolution implementation, and of course getting exactly the same results.

```
for posY in range(0, err_out_shape0):
    for posX in range(0, err_out_shape1):

        # valid convolution (dE/dF)
        subxpad = X_pad[posYf:posYf+fil_shape0, posXf:posXf+fil_shape1]
        error_by_filter += subxpad * error_by_output[posY, posX]

        # full convolution (dE/dX)
        incx = layer_filters * error_by_output[posY, posX]
        error_by_x[posYf:posYf+ex_shape0, posXf:posXf+ex_shape1] += incx

        posXf = posXf + 1

    posYf = posYf + 1
    posXf = eS1

error_by_x = np.flip(error_by_x, 0)
error_by_x = np.flip(error_by_x, 1)
```

Fig. 5. Final solution for my convolutional backpropagation

4.3. Auto machine learning

To achieve the objective that my library can be used by people without knowledge of neural networks, I have decided to add an optional AutoML function in which absolutely no hyperparameters need to be specified. In this way, the library is responsible for obtaining the model that generates the best results in terms of metrics, by determining **all** hyperparameters automatically.

In my case I have been inspired by the method proposed in the article *Large-Scale Evolution of Image Classifiers* [9] to create my own algorithm from scratch. As I explained in the state-of-the-art section, in that article a population of models is maintained, and pairs are chosen at random to *kill* the worst of the two models. This can cause models that are worse than others to *die*, but those models could end up improving more than the rest. For that reason, I will not use a structure in pairs, but a structure in the shape of a tree.

My tree will start from a model with the default hyperparameter values. Each model will have 3 children by default, with mutations that involve one modification of one randomly chosen hyperparameter, so that it acquires a new value chosen by a uniform distribution. Subsequently, each child is trained with a shuffled subset of the training set, and then the accuracy of classifying a shuffled subset of the test set is evaluated. No node *dies* here, unlike the approach of the aforementioned article. Again, in my algorithm, each child has 3 more children, thus building the tree. To avoid exponential growth, this is done up to a maximum of a certain depth, which can be modified from my constructor. In this way, the models are given the opportunity to improve with the following mutations if their performance worsens with one mutation, just as a human expert would do manually, rather than completely discarding all bad new models. Nonetheless, once the maximum number of children and the maximum depth are reached, the models with the worst accuracy are *killed*, since they have not *survived* evolution. In this first iteration I have made that only leaves (nodes of level *n*) can *die*.

Afterwards, this whole process is performed again, but only the nodes of level *n-1* whose children have been eliminated will have children, and they will have as many children as they have

lost, so that all nodes always have the same number of descendants. Again, a pruning is carried out, but this time nodes of level *n-1* and higher (the leaves) can be pruned. And so on during the *m* iterations that the user wants to perform. The rate at which the pruning level increases can be modified from the constructor. For instance, it can be done that during the first 5 iterations only leaves can be removed, for the next 5 iterations leaves and nodes at level *(n-1)* are removable, and during the last 5 iterations depths *(n-2)* and higher can be pruned. The ratio of pruned nodes, the maximum depth and the children per node can also be modified from the constructor.

5. LIBRARY RESULTS

5.1. Image classification

This subsection shows the result of using my library to classify images from the Digits MNIST and Fashion MNIST datasets and a comparison using other libraries. I have done all of these runs on standard Google Colaboratory machines. All networks in Table 1 and Table 2 have been executed 4 times, and the numbers included are the mean ± the standard deviation across all runs. All of the MLPs in Table 1 and Table 2 have a single hidden layer of 25 neurons, architecture that has given good results in this problem.

Digits MNIST

Model	Test accuracy	Training time (minutes)
AbelNN MLP	0.9515 ± 0.0011	3.19 ± 0.01
Scikit-learn MLP	0.9409 ± 0.0017	3.06 ± 0.43
Keras MLP	0.9527 ± 0.0010	5.17 ± 0.34
AbelNN CNN	0.9627 ± 0.0016	11.61 ± 0.01
Keras CNN	0.9769 ± 0.0003	9.31 ± 0.11

Table 1: Comparison between several neural networks from different libraries when using the Digits MNIST dataset.

As can be seen in Table 1, my multilayer perceptron is capable of obtaining better results as that of the *scikit-learn* library [4], with similar times, when classifying images from the MNIST digits dataset. The efficient vectorized implementation that I have made for my MLP is appreciated in these training times. The *scikit-learn* MLP used to classify the Digits MNIST images runs until convergence. A batch size equal to 1 is used because the three libraries use different batch implementations.

Fashion MNIST

Model	Test accuracy	Training time (minutes)
AbelNN MLP	0.8688 ± 0.0064	3.20 ± 0.01
Scikit-learn MLP	0.8722 ± 0.0033	8.00 ± 0.33
Keras MLP	0.8764 ± 0.0029	10.30 ± 0.31
AbelNN CNN	0.8642 ± 0.0009	31.85 ± 0.02
Keras CNN	0.9106 ± 0.0002	29.37 ± 3e-05

Table 2: Comparison between several neural networks from different libraries when using the Fashion MNIST dataset.

On the other hand, in the case of the Fashion MNIST dataset, I get worse results because many pieces of clothing are very similar, like those of the classes **Sandal**, **Ankle boot** and **Sneaker**, causing my model to make more mistakes when predicting these classes. Sometimes it also mispredicts some images of the **T-shirt** and **Shirt** classes, which are extremely similar garments. However, as I have mentioned, the overall accuracy is quite high. For more detailed information on the models in my library used in the comparisons in this section, such as its hyperparameters, training *loss* plots, and test confusion matrices, see Appendix 1.

The *scikit-learn* MLP used with the Fashion MNIST images after 400 iterations has not yet converged and also learns very little from iteration 200 onwards. That said, for the time comparison with this library to be fair, the execution of the *scikit-learn* MLP with Fashion MNIST images has been limited to 200 iterations. Even so, as it can be seen in Table 2, it runs slower than my MLP. However, in this case *scikit-learn* has generalized better than with the Digits MNIST, in which my MLP obtained a greater accuracy. As a consequence, both libraries get fairly similar performance and metric results overall.

In the case of ConvNets, Keras clearly beats me both in accuracy and time, because it has GPU support and they have been optimizing it for years. In particular, Keras gets better accuracies than AbelNN because they use even more advanced and complex gradient descent algorithms, which would have taken me longer to implement and that was not part of the scope of the project to cover other topics. They also obtain better results because they apply pooling techniques that have an extra computational cost, unlike stride, which reduces computational cost in exchange for slightly worse results. Conversely, my goal was neither to obtain perfect metrics nor a highly optimized library, but rather a module with good results with decent times, but that can be used by anyone. This first objective is fulfilled in this section since it is true that I am very close to the accuracies of the other modules when using my networks.

As I have already shown that my library is versatile and can be used with different datasets with good results, in some of the following experiments in this article I will only show results with Digits MNIST because I have run hundreds of experiments with a total of hundreds of hours of execution.

5.2. Auto machine learning

In this section we are going to see a comparison between my AutoML module and the well-known Random Search and Grid Search algorithms, to obtain the best hyperparameters to classify the images of the Digits MNIST dataset, first using my multilayer perceptron, and later using my convolutional neural network.

5.2.1. Comparison using AbelNN MLP

In Fig. 6 we can observe a plot with the best accuracy obtained so far after each iteration of the three algorithms. That is why the plot is staggered, because if a much better model is found in the next iteration, the accuracy will be much higher, and a sudden large increase will be seen in the plot.

The 3 methods randomly select, in each iteration, 50 images from the shuffled training subset for training and 50 images after shuffling the test subset for the test, to be able to run thousands of iterations in minutes. As is common, Grid Search takes longer than the other methods to improve accuracy considerably, since it tries all possible hyperparameter combinations. It is also true that this means that, after testing more than 2500 models, it seems that it gets better results than my algorithm, when in fact, when we generalize (see Table 3), my algorithm completely beats the Grid Search in both time and accuracy.

What is also remarkable is how my AutoML module is able to find better accuracies even before Random Search. Moreover, it even ends up finding an optimal hyperparameter setting that produces a higher accuracy than the best model found by Random Search.

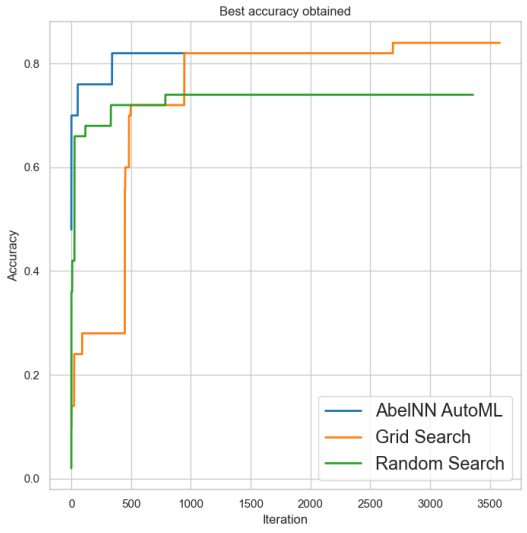

Fig. 6. Best accuracy found on each iteration of the algorithms of Grid Search, Random Search and my AutoML module when using my MLP with images from Digits MNIST dataset.

In the following Table 3 there are some more details about the comparison between the 3 algorithms. In the first column we can see how many models each algorithm has trained and tested, and in the second column how long it took to perform the search for the best hyperparameter configuration. Finally, last column shows the accuracy obtained using the best model found by each algorithm, but now increasing the number of epochs of training to 20 and using all images of training subset to train and all images of test subset to test, since the best model found will only run one time per algorithm for the final comparison.

Algorithm	Tested models	Hyperparameter search time	Final test accuracy
AbelNN AutoML	960	93.05 seconds	0.9548
Random Search	3360	182.54 seconds	0.9085
Grid Search	3584	183.46 seconds	0.9337

Table 3: Comparison between AbelNN AutoML, Random Search and Grid Search to find the MLP optimal model when classifying images of MNIST digits.

As can be seen in Table 3, the accuracy obtained with my AutoML is higher than that obtained with both Grid Search and Random Search, even considering that Random Search is one of the most used methods today for searching hyperparameters. The reason my method works better is that while those other two methods are testing isolated models, my meta-learning algorithm builds a tree of models that evolve, and prunes those that do not survive. This ends up producing multiple positive mutations accumulating in the same model, which turns out to be the one that gives the best results in the classification. This also allows my algorithm to have to test approximately 3.5 times fewer models than the other two methods. As a result, my algorithm runs in half the time of the Random Search and Grid Search.

As you may have noticed, an execution of just 93 seconds of my AutoML module is able to find the best model of my MLP, which obtains an accuracy equal to that of the model that I manually tuned for a couple of hours in section 5.1.

5.2.2. Comparison using AbelNN ConvNet

Using a Grid Search with a convolutional neural network is completely infeasible since there are tens of thousands of possible combinations of hyperparameters after choosing a reasonable search space. Reducing the range of the hyperparameters further would allow the Grid Search to be used, but it would no longer be useful because the values it would test would be too narrow. Therefore, in the case of my CNN, the comparison will be between my AutoML module and a Random Search. This comparison uses these two algorithms to find the best configuration of hyperparameters of the ConvNet of my library when classifying the Digits MNIST images again.

In this case, to compare both algorithms, the Random Search has been executed with a number of models that make its execution time similar to that of my AutoML. At that time, Random Search is capable of testing 1.5 more models than my AutoML, which requires pruning the worst models, a task that requires extra computation. We can see that this extra computation rewards us again with a better model than Random Search, since as we can see in the third column of the following Table 4, when we train the best model found by each algorithm with all the train data (instead of a different shuffled subset of training data in each iteration), the best model that finds my AutoML obtains the highest test accuracy of the two algorithms. It achieves almost the same accuracy as the model that I manually tuned for several hours in section 5.1.

Algorithm	Tested models	Hyperparameter search time	Final test accuracy
AbelNN AutoML	533	15.6 minutes	0.951
Random Search	777	15.1 minutes	0.947

Table 4: Comparison between AbelNN AutoML and Random Search to find the CNN optimal model when classifying images of MNIST digits.

Similarly, when my AutoML module is used with the Fashion MNIST images, once again it is able in less than 1 minute for the MLP and in less than 20 minutes for the CNN, to find a model that achieves the same accuracy as the models that I manually tuned for section 5.1.

As we have seen in this section, my AutoML is able to find the optimal configuration of hyperparameters for the input data, regardless of how and what this data is. My algorithm is able to do it in much less time than it takes a human to manually perform the *babysitting* strategy, widely used even by experts and researchers, freeing people from this tedious task. Furthermore, when a network is tuned manually, there is often the question of whether we have already found the best hyperparameter configuration or the best architecture for our problem. Nonetheless, my AutoML module automatically performs multiple tests with multiple models, keeping them all alive until it prunes the worst models. This is repeated in several iterations, testing thousands of models in a matter of a few minutes, ensuring that the model it finds will be one of the best that exists for our problem.

A researcher could automate the task of testing thousands of models, but that would be like a Random Search, Grid Search or some similar strategy, and therefore the results, as I have shown in this section, would not be as fast or as good as with my evolutionary algorithm. Nevertheless, as my library uses common parameter names, it is compatible with other already existing methods and algorithms for tuning models as we have also seen.

5.2.3. *Comparison using Tensorflow ConvNet*

Continuing with the previous experiments, another run of my AutoML module and Random Search has been done to determine the best model to classify the images of the MNIST dataset, but this time using a *Tensorflow* convolutional neural network architecture. The two algorithms determine the number of convolutional layers, their filters, the kernel size, the batch size, the learning rate, the activation function, the dropout rate, the kernel initializer and whether pooling is used or not. A search space has been established that does not require more than 6 minutes for the Random Search to run, and then this same search space has also been applied to my AutoML so that they are in the same conditions. The hyperparameter space that both methods work with has 1728 possible combinations in total.

Both algorithms search for the optimal model in that space, with a subset of 100 images from the train subset and 100 entries from the test subset, different in each iteration, always shuffled between them before splitting, and never mixing train and test. After opting for the optimal model found by each algorithm, it is trained with the total of train data, and then the total of test data is predicted. This entire process is repeated 5 completely independent times to ensure the validity of the experiment. The plot in Fig. 7 shows the accuracy obtained in each of these 5 runs, by classifying all the test data with the best model found by each algorithm.

Fig. 7. Accuracy plot of the 5 runs of the comparison between AbelNN-AutoML and RandomSearch with Tensorflow models.

It can be seen that my meta-learning algorithm achieves practically the same accuracy in the 5 executions, despite the randomness of the selection of mutations, because it builds the best model evolutionarily. On the contrary, Random Search, due to its extreme randomness, in the first execution it manages to surpass my AutoML, but in the second it obtains a disastrous accuracy. In other words, my algorithm, despite introducing certain elements of random selection, produces consistent and repeatable results for the same data. It should also be noted that the accuracy results obtained with my AutoML are excellent. In Table 5 the accuracies can be seen again, but numerically.

Algorithm	Test accuracy		
	Minimum	Mean	Maximum
AbelNN AutoML	0.95	0.96	0.97
Random Search	0.20	0.78	0.98

Table 5: Test accuracies of the best models found among the 5 runs of the comparison between AbelNN-AutoML and RandomSearch with Tensorflow models (small search space).

Another key point, as you can see in Fig. 8, is that Random Search requires significantly higher execution times than my method. In Table 6 you can see these times as mean, maximum and minimum values among the 5 runs of the experiment.

Notably, on average my AutoML takes 3.27 minutes to run, while Random Search takes 5 minutes. This is because my AutoML performs random mutations, within the search space, but it always tests an exact number of models determined by the limits of the tree that have been defined in the constructor (or that have been left by default). Specifically, it always takes the same time because my algorithm has 2 restrictions designed for this purpose. The first is the number of children that each node (each model trained and tested) has, and the second restriction is the maximum depth of the tree. In this way, its complexity and execution time is completely independent of the size of the search space.

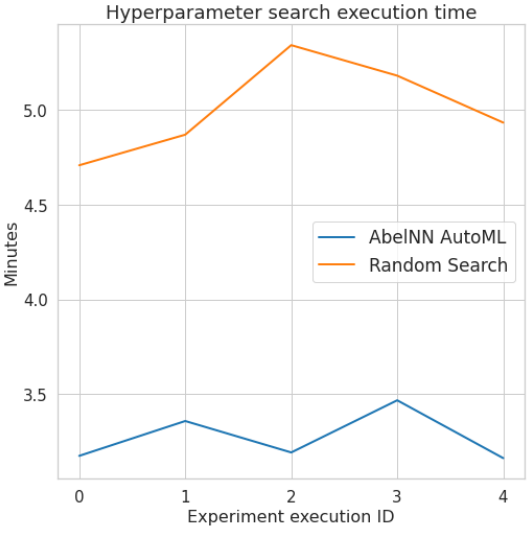

Fig. 8. Execution times plot of the 5 runs of the comparison between AbelNN-AutoML and RandomSearch with Tensorflow models.

Algorithm	Execution time (minutes)		
	Minimum	Mean	Maximum
AbelNN AutoML	3.16	3.27	3.47
Random Search	4.71	5.01	5.34

Table 6: Hyperparameter search execution times among the 5 runs of the comparison between AbelNN-AutoML and RandomSearch with Tensorflow models (small search space).

In fact, my approach performs so well in terms of time because I have previously made Trial & Error with my AutoML algorithm to find its "hyper-hyper-parameters" that allow a more efficient use of its tree structure for almost any input data. In other words, I have taken a first step towards "meta-meta-learning". This means that, as I have already mentioned, these "hyper-hyper-parameters" of my AutoML do not need to be modified any more, allowing users to run it by specifying only the data to use. However, users who want to customize my algorithm can easily modify its parameters from the constructor. For instance, if you want the algorithm to test a specific number of models, you can set accordingly the value of the 2 restrictions mentioned above, which are calculated as in any tree.

As I mentioned, if the data is the same, my algorithm will always take the same time to execute even if the hyperparameter space increases. To demonstrate this, and see how good my algorithm is, I increase the space of hyperparameters that both methods work with so that it goes from 1728 possible combinations as before to a total of 19200 possible combinations. Fig. 9 shows the time plot for the 5 runs of the new experiment.

As can be seen in Fig. 9, by increasing the size of the search space 10 times, Random Search takes 10 times longer, since this type of Random Search searches for a percentage of random combinations, and therefore its execution depends on the size of said space. In contrast, my AutoML takes exactly the same time as when the search space was 10 times smaller, for the reasons I explained previously. Table 7 shows these times numerically, where the Random Search now takes an average of 55 minutes to run and my AutoML still takes only 3 minutes.

Fig. 9. Execution times plot of the 5 runs of the comparison between AbelNN-AutoML and RandomSearch with Tensorflow models.

Algorithm	Execution time (minutes)		
	Minimum	Mean	Maximum
AbelNN AutoML	3.00	3.11	3.18
Random Search	54.60	55.81	56.77

Table 7: Hyperparameter search execution times among the 5 runs of the comparison between AbelNN-AutoML and RandomSearch with Tensorflow models (big search space).

Another significant aspect is that my AutoML, as it happened with the smaller search space, is able to give practically the same accuracy in the 5 executions despite the randomness of the selection of the mutations, in contrast to the extreme randomness of Random Search that, as seen in Fig. 10, the results of its best model fluctuate greatly between each run of the experiment, due to such randomness.

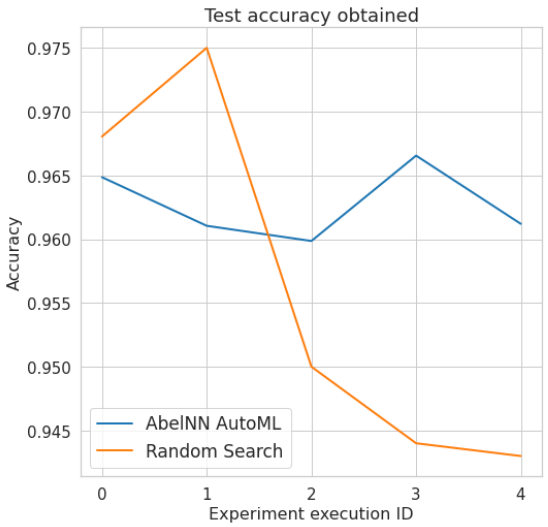

Fig. 10. Accuracy plot of the 5 runs of the comparison between AbelNN-AutoML and RandomSearch with Tensorflow models.

Furthermore, in Table 8 it is clearly appreciated how, despite the fast execution of my algorithm, both techniques obtain the same accuracy on average when classifying all the data of the test subset with the best model found by each algorithm.

Algorithm	Test accuracy		
	Minimum	Mean	Maximum
AbelNN AutoML	0.96	0.96	0.97
Random Search	0.94	0.96	0.98

Table 8: Test accuracies of the best models found among the 5 runs of the comparison between AbelNN-AutoML and RandomSearch with Tensorflow models (big search space).

If we compare Table 5 and Table 8 we will see that with a small search space my AutoML was already able to obtain an average accuracy of 0.96 and in the worst case one of 0.95. However, Random Search achieved a mean accuracy of 0.78 across all experiments, and a much larger hyperparameter search space has been needed (Table 8) to find models that match what my AutoML already found with the small space.

If we join the means of the four tables (Table 5, Table 6, Table 7 and Table 8) we obtain Table 9 in which we can see quickly the comparison between both algorithms, with the same execution time of my algorithm in both the small and large search spaces, and on the contrary, the execution time of Random Search is multiplied by 10 when going from the small to the large space.

Algorithm	Small search space		Big search space	
	Mean time (minutes)	Mean accuracy	Mean time (minutes)	Mean accuracy
AbelNN AutoML	3.27	0.96	3.11	0.96
Random Search	5.01	0.78	55.81	0.96

Table 9: Summary of the comparison between AbelNN-AutoML and RandomSearch using Tensorflow network models, for the 5 runs with the small search space and the 5 runs with the large search space.

As you may have noticed, running this experiment with Grid Search would have required more than 90 hours to complete. For this reason, as I already indicated in section 5.2, Grid Search has been discarded due to its infeasibility.

All things considered show us that my algorithm is capable of obtaining the best model for our data much faster than Random Search (18 times faster in the case of a large search space like the one in this last example). In addition, we have seen how the complexity of my AutoML does not depend at all on the size of said search space, so we can expand it without the fear of slowing down the execution.

Secondly, the accuracy obtained with my algorithm in the case of large search spaces is the same as that obtained with Random Search, and on average much higher in the case of small search spaces, despite the fast execution of my method both in small and large spaces. Moreover, this accuracy is excellent and equals that obtained by an expert with a manual tuning (which takes hours).

Finally, it has also been shown that the results obtained with my meta-learning algorithm are always almost the same provided that the input data is the same, being quite deterministic even though the mutations are random, thus avoiding the need to execute it more than once.

6. USING MY LIBRARY WITH USERS WHO HAVE NEVER TRIED DEEP LEARNING

In this subsection we are going to see the result of another of the objectives that I wanted to achieve with my library, and that is making people without deep learning knowledge able to use my module. Of course, advanced users can use my internal procedures to define their own architectures or models, but in this section, we will look at two ways that inexperienced users can use my library. To demonstrate that my library is indeed easy to use, I went to my old high school to ask 30 students to use my module to classify MNIST images without giving them any clue. Absolutely all of them managed to classify the images correctly using my AbelNN, and in the two following subsections you will find more specific information on how they did it.

6.1. Using my AutoML class

The easiest way to use my library is simply choosing which network you want to use and letting my AutoML class take care of finding the best hyperparameters for your data, as seen in the Fig. 11 example.

```
1  import AbelNN
2
3  # Instance a default ConvNet model:
4  c = AbelNN.ConvNet()
5
6  # Pass the ConvNet to the AutoML.
7  # It returns the best model for your data:
8  clf = AbelNN.AutoML(c).fit(x_train, y_train, x_test, y_test)
9
10 # Train the auto tuned model:
11 clf.fit(x_train, y_train)
12
13 # Use the model to predict your test data:
14 clf.predict(x_test, y_test)
```

Fig. 11. Example of using my AutoML to customize my ConvNet.

As we can see, only with 1 line (line 8 of Fig. 11) the best model can be easily obtained without specifying any hyperparameters, only indicating the data we want to use.

Using my AutoML module, all the high school students I mentioned above got the maximum accuracy that I also achieved, thereby demonstrating that it can be used easily.

6.2. Using my customizable predefined models

Users who want to be able to modify the hyperparameters of neural networks can use the classes that I have predefined, such as MLP and CNN. The constructor of that predefined classes that I have created allows you to fully define and customize the model, following the philosophy that the *scikit-learn* library also follows [4], to make it simpler for the user. In this way, in a single line of code you can have your model with the architecture and hyperparameters that you want.

Then, users can call the *fit* function, that automatically determines the number of neurons needed for the input layer and the output layer from the train data, thereby hiding actions that could overwhelm inexperienced users. For this reason, I have also defined, among other procedures, a convolution function that adapts perfectly to the input data, automatically calculating the necessary size of the filters and of each of the outputs of the convolutional layers, to abstract even more users who do not want to go into these concepts. In other words, with 3 lines, 1 to define the model and its hyperparameters (constructor), another to train (*fit* procedure) and another to predict (*predict* or *predict_proba* functions), a classification or regression problem can be solved with my library, as seen in Fig. 12.

```
1  from ConvNetAbel import *
2
3  clf = ConvNetAbel(convFilters=[64,64], convStride=[2,1],
4                    convFilterSizes=[5,3], hidden=[30,20],
5                    learningRate=0.1)
6
7  clf.fit(x_train, y_train)
8
9  probabs = clf.predict_proba(x_test)
```

Fig. 12. Example of customizing and using my predefined CNN model.

The example of Fig. 12 instantiates a convolutional neural network with 2 convolutional layers of 64 filters each, the first with stride equal to 2 and the second equal to 1, and filters of size 5 in all dimensions of the data in the first layer and size 3 in the second layer. After the convolutional layers it has 2 fully-connected hidden layers, one with 30 neurons and the other with 20. The output layer is always generated automatically. The learning rate is 0.1.

After asking the high school students to use my predefined MLP and CNN constructors, absolutely all 30 students achieved, in less than 1 hour, accuracies greater than 0.9 by performing Trial & Error only by reading my library documentation. Therefore, my Deep Learning module is extremely easy to use even by users who have never had contact with artificial intelligence, thus fulfilling the main objective of this project.

7. NEURAL NETWORK INTERNAL EXPERIMENTS

7.1. Standardization between layers

During the implementation of my neural networks, when dealing with numbers and low-level math calculations, I used to constantly have the exploding gradient problem. To solve it I implemented several initializations of the weights and filters, but having that problem made me think of another extra solution: normalize the values of the neurons in the hidden layers, to

avoid that they grow a lot but to be able to continue using functions like ReLU that had given me such good results.

7.1.1. Standardization between layers in MLPs

In deep learning there is a widely used technique which consists of performing a normalization of the standardization type, that is, subtract the mean from our data so that it has zero-mean and then divide the result by the standard deviation of our data so that it has unit-variance. This is usually done (and I do it) with the data before entering it into the neural network. However, my solution is to apply this standardization between the hidden layers as well. I do this with the values resulting from multiplying the input of a layer and the weights, before entering the result in the activation function. In addition, in this calculation of the mean and standard deviation only those values participate, neither the values of the full dataset nor those of other layers, so all the standardizations that I carry out are completely independent.

In the following Fig. 13 we can see the error made during training with a subset of the Fashion MNIST dataset with my normalization between layers activated (orange) or deactivated (blue), which I have named *pre_norm*. The first model, colored blue, has a learning rate of 0.1 and normalization disabled. Instead, the orange model has a learning rate of 1.5 and my normalization turned on. I have done this to show that my *pre_norm* allows me a smooth and stable learning even at very high learning rates. In fact, convergence is vastly faster and more stable than with a smaller learning rate without using normalization (blue). The consequence is that my algorithm now requires to run half the epochs. As activation function, I have used ReLU in both cases, and all the other hyperparameters of the network have been exactly the same for the two models. The first model (blue) gave an accuracy of 0.76, while the second one an accuracy of 0.8, both for the test data. This suggests that this normalization has not caused overfitting since I am classifying the test data (never seen during training) even better than without normalization. This standardization is always carried out with the data in that layer, that is, during the train phase a normalization is carried out in each layer, and during the test phase another normalization is carried out in each layer, so standardization during training does **not** use test data.

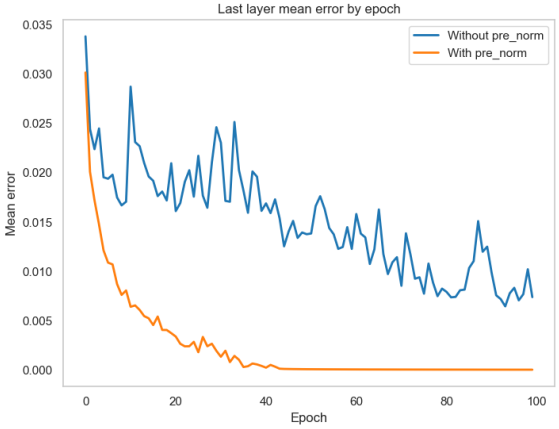

Fig. 13. Mean training error in the network output for the first *pre_norm* experiment.

After seeing this excellent results, I launched a second experiment in which I performed 4 executions with the hyperparameter differences described in Table 10.

Model ID	Learning rate	*pre_norm* enabled
1	0.001	False
2	0.001	True
3	1.5	False
4	1.5	True

Table 10: Hyperparameters in which the 4 models of the second *pre_norm* experiment differ.

Again I will use ReLU as the activation function in all cases. The result is shown in Fig. 14.

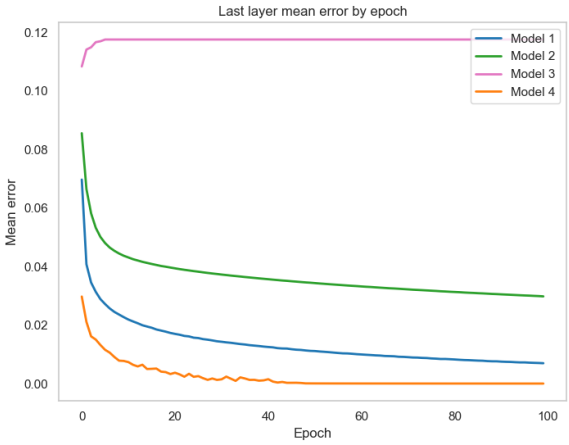

Fig. 14. Mean training error in the network output for the second *pre_norm* experiment.

It is obvious in Fig. 14 that model number 3 suffers from overflow owing to too large updates in the weights due to an excessively large value of learning rate. By contrast, model 4 has the same learning rate value as model 3, but when applying my normalization between layers it does not suffer this problem and converges without problems. Model 2 also has this normalization activated, but it has a lower learning rate. We can see that this results in slightly more stable learning than model 4, but it also slows down learning. Finally, model 1 has a low learning rate which causes a more stable but slow learning.

After predicting with the test data, model 4 is able to generalize and obtain the same accuracy as model 1, but model 4 requires only half as many epochs as model 1. All things considered show us that using my normalization between layers allows to use learning rate values even up to 3 orders of magnitude higher than without the normalization, allowing a much faster convergence but that is still stable, and it generalizes perfectly.

My approach works so well mainly because this standardization between layers allows the network to adjust the weights in the right direction and the right proportion, but never by increasing the values too much and therefore avoids that in the next steps of the backpropagation they have to be decreased again, which commonly occurs with high learning rates, thereby avoiding this instability in learning. Furthermore, usually during backpropagation, if weights are much higher than 1 we can have the exploding gradient problem, and if they are much lower than 1, the vanishing gradient problem. However, if we use my normalization technique, neither of those 2 problems occurs over a very wide range of learning rate with both ReLU and sigmoid.

7.1.2. Standardization between layers in ConvNets

I decided to implement the same *pre_norm* strategy on my CNN as well. Contrary to before, when using high learning rate values as with my MLP I started to get overflow. So, I also applied the standardization on the convolutional layers, and as a consequence, I was able to use learning rate values up to an order of magnitude higher. To be able to increase it as much as in my MLP, it occurred to me to also standardize the value of the derivative of the error with respect to the filters in the same way, to update the kernels in the direction of the gradient, but with normalized values. And indeed, this allowed me to use, with my *pre_norm* enabled, learning rate values even close to 3, when previously I had to use values close to 1e-4 as is normally done with other libraries. Using learning rates greater than 1 does not ensure convergence unless my *pre_norm* technique is used, as we have already seen. This way, learning on my ConvNet is also sped up a lot. Nevertheless, such high learning rate values can cause very unstable learning and even carry out weight updates in which the minimum value is always skipped. However, with the appropriate value, which can now be higher thanks to this technique, I can avoid enough epochs to be able to execute my algorithm in times comparable to those of other neural network frameworks like *Tensorflow* and *Pytorch*. These creative solutions allowed me not to need GPU support with my library, which is not one of the objectives of the project.

8. Conclusions

I have managed to create a library that reaches accuracies comparable to those of the state of the art, and that is also extremely easy to use as I have demonstrated with the high school students who have used it successfully. This project has helped me to learn a lot about how neural networks work, not only at a mathematical level, but also which are the problems that occur when implementing them in practice, such as overflow. Facing these issues has given me the opportunity to offer my own creative solutions, such as my standardization between layers, which allows the use of high values of learning rate, something that is not usually considered, but that in this case accelerates the convergence without its inconveniences of possible overflow and instability. I wanted to implement everything from scratch precisely so that I could come up with solutions like that. In fact, this low level has let me change things that are not commonly modified, thereby giving me knowledge that is not usually acquired when working with high-level frameworks. In this way, in the future I will be able to continue working with low-level networks, but also apply what I have learned here in current high-level libraries such as *Tensorflow* or *Pytorch*, or any other, because now I understand the fundamentals.

On the other hand, I have also entered the world of meta-learning to fulfill the objective that my library is easy to use, creating my AutoML module that is able to find the best model for our data in a matter of minutes, with its optimal architecture and hyperparameters, even surpassing the traditional techniques of Random Search and Grid Search in time and accuracy. Importantly, it is also faster than humans tuning networks manually. Actually, it is able to find in a few minutes models that achieve the same accuracy as those that I have manually tuned for several hours.

Finally, despite the fact that throughout the article I have focused on using my networks to classify images in order to also be able to compare the results with my CNN, I must mention that I have also carried out tests with my MLP for classification and regression of non-image datasets and I also got excellent results. I have also implemented procedures to plot the loss during training, to visually draw the network architecture and to import and export models from my module. To conclude, I believe I can be proud of my library, although in the future, with more resources, I would like to improve it further by performing more internal experiments that allow me to come up with new ideas such as my *pre_norm*, as well as improving my AutoML with reinforcement learning to generate smarter mutations that prioritize the hyperparameters that most influence results, and even work on creating new meta-learning algorithms.

9. References

[1] Keras, "About Keras," [Online]. Available: https://keras.io/about/. [Accessed September 2021].

[2] Pytorch, "Pytorch documentation," [Online]. Available: https://pytorch.org/docs/stable/index.html. [Accessed September 2021].

[3] Berkeley AI Research, "Caffe," [Online]. Available: https://caffe.berkeleyvision.org/. [Accessed September 2021].

[4] Pedregosa et al., "Scikit-learn: Machine Learning in Python, pp. 2825-2830," 2011. [Online]. Available: https://scikit-learn.org/. [Accessed September 2021].

[5] TensorFlow, "Glorot normal initializer," [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/initializers/GlorotNormal. [Accessed September 2021].

[6] Erin Grant, Chelsea Finn, Sergey Levine, Trevor Darrell, Thomas Griffiths, "RECASTING GRADIENT-BASED META-LEARNING AS HIERARCHICAL BAYES," 26 January 2018. [Online]. Available: https://arxiv.org/pdf/1801.08930.pdf. [Accessed October 2021].

[7] Marcin Andrychowicz, Misha Denil, Sergio Gómez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan hillingford, Nando de Freitas, "Learning to learn by gradient descent," 30 November 2016. [Online]. Available: https://arxiv.org/pdf/1606.04474.pdf. [Accessed October 2021].

[8] Barret Zoph, Quoc V. Le, "Neural architecture search with reinforcement learning," 15 February 2017. [Online]. Available: https://arxiv.org/pdf/1611.01578.pdf. [Accessed October 2021].

[9] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V. Le, Alexey Kurakin, "Large-Scale Evolution of Image Classifiers," 11 June 2017. [Online]. Available: https://arxiv.org/pdf/1703.01041.pdf. [Accessed October 2021].

[10] Na8, "How do the Convolutional Neural Networks work?," 29 November 2018. [Online]. Available: https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/. [Accessed September 2021].

[11] P. Solai, "Convolutions and Backpropagations," 19 March 2018. [Online]. Available: https://medium.com/@pavisj/convolutions-and-backpropagations-46026a8f5d2c. [Accessed September 2021].

[12] H. Inada, "Calculate CNN backprop with padding and stride," [Online]. Available: https://hideyukiinada.github.io/cnn_backprop_strides2.html. [Accessed October 2021].

[13] "Fashion MNIST classification," [Online]. Available: https://www.kaggle.com/zalando-research/fashionmnist/code. [Accessed October 2021].

[14] "Image Classification on MNIST," [Online]. Available: https://paperswithcode.com/sota/image-classification-on-mnist. [Accessed October 2021].

[15] GIMA_OKI, "DigitRecognizer with keras," August 2021. [Online]. Available: https://www.kaggle.com/gimaoki/digitrecognizer-keras. [Accessed October 2021].

APPENDIX

Appendix 1

This appendix shows in more detail the models from my library used in section 5.1 comparisons. The error appears to reach zero in some of the *loss* plots, but in none of the cases does it reach completely zero, so the network continues to learn. To display the plots seen in this appendix, it is as simple as calling the procedure *plot_mean_error_last_layer* of my library, which I have implemented so that inexperienced users can easily display them.

As you will see in this section, with only the constructor it is possible to fully customize my multilayer perceptrons and my convolutional neural networks.

My multi-layer perceptron

Digits MNIST

Table 11 shows the hyperparameters of the MLP used to classify Digits MNIST images in section 5.1.

Hyperparameter	Value
Number of hidden layers	1
Hidden layers neurons	[25]
Number of epochs	30
Learning rate	0.3
Softmax	False
Activation function	ReLU
Batch size	1
Pre-norm	True

Table 11. Model of my MLP for the Digits MNIST.

The code to initialize the multilayer perceptron described in Table 11 is as simple as the following constructor shown in Fig. 15.

```
MLP_Abel(hidden=[25], nEpochs=30, learningRate=0.3, debugLevel=2,
        rangeRandomWeight=None, softmax=False, activationFunction='relu',
        verbose=True, showLogs=False, batch_size=1, pre_norm=True)
```

Fig. 15. Model of my MLP for the Digits MNIST

In Fig. 16 it can be seen a plot with the mean error made by the network in the output layer during training for each epoch, for the 10 classes (the 10 values of the legend correspond to the 10 digits of the MNIST).

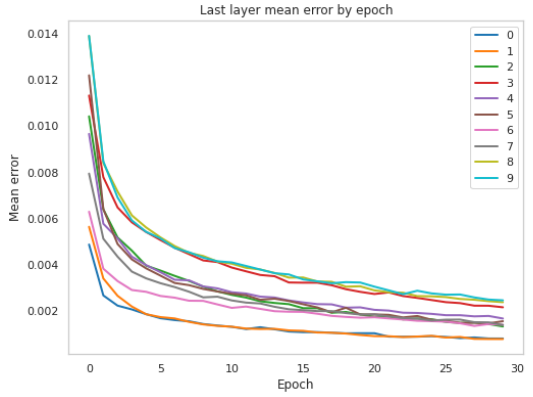

Fig. 16. Mean error per epoch during Digits MNIST MLP training

Finally, the confusion matrix of the classification of the test subset of the images is shown in Fig. 17.

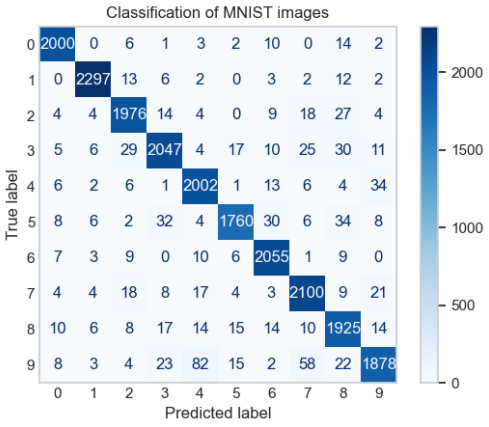

Fig. 17. Confusion matrix of the classification of the Digits MNIST test images with my MLP

Fashion MNIST

Again, my model of AbelNN MLP is shown in Table 12 but this time tuned to classify the Fashion MNIST images in section 5.1.

Hyperparameter	Value
Number of hidden layers	1
Hidden layers neurons	[25]
Number of epochs	20
Learning rate	0.5
Softmax	False
Activation function	ReLU
Batch size	2
Pre-norm	True

Table 12. Model of my MLP for the Fashion MNIST.

Again, in the constructor, as you can see in Fig. 18, you can configure all the hyperparameters of the model:

```
MLP_Abel(hidden=[25], nEpochs=20, learningRate=0.5, debugLevel=2,
        activationFunction='relu', verbose=True,
        batch_size=2, pre_norm=True)
```

Fig. 18. Model of my MLP for the Fashion MNIST

We can see in Fig. 19 a plot with the error during training. It has been verified that model has not been overtrained, carrying out different experiments modifying the number of epochs, and adding dropout, as well as obtaining test accuracies afterwards (Fig. 20).

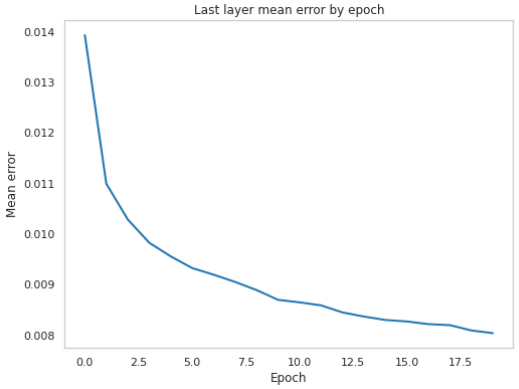

Fig. 19. Mean error per epoch during Fashion MNIST MLP training

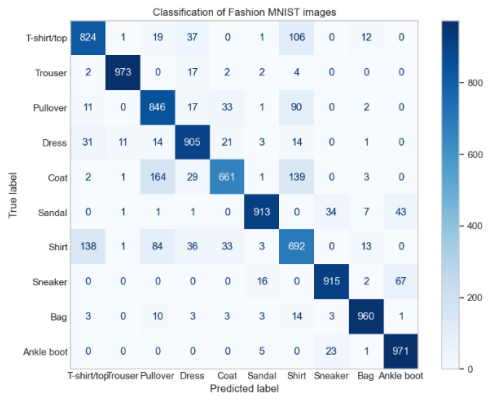

Fig. 20. Confusion matrix of the classification of the Fashion MNIST test images with my MLP

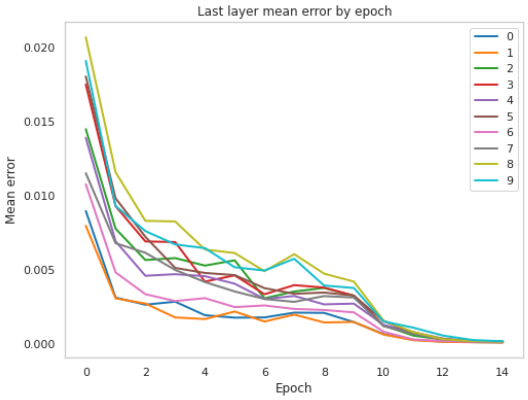

Fig. 22. Mean error per epoch during Digits MNIST CNN training

My Convolutional Neural Network

Digits MNIST

When using my CNN to classify the images from Digits MNIST in section 5.1, the optimal model, described in Table 13, uses one convolutional layer of 16 filters. Here, the use of my *pre_norm* regularization has been key to increase the learning rate without destabilizing the learning, thus being able to reduce the total number of epochs and therefore the training time, which as we saw in section 5.1, is quite competitive.

Hyperparameter	Value
Number of epochs	15
Number of convolutional layers	1
Number of filters	[16]
Stride per layer	[2]
Filter sizes	[3]
Kernel initializer	Xavier normal
Number of hidden layers	1
Hidden layers neurons	[25]
Learning rate (fully-connected part)	0.9
Learning rate (convolutional part)	0.1
Activation function	ReLU
Iteration drop	0.8
Batch size	1
Pre-norm	True

Table 13. Model of my CNN for the Digits MNIST.

With my ConvNet we can also comfortably configure all the hyperparameters in the constructor as in the following Fig. 21:

```
ConvNetAbel(nEpochs=15, convFilters=[16], convStride=[2], convFilterSizes=3,
            hidden=[25], learningRate=0.9, debugLevel=2,
            showLogs=False, activationFunction='relu', learningRateConv=0.1,
            verbose=False, pre_norm=True, iterationDrop=0.8,
            kernel_initializer='xavier_normal', batch_size=1)
```

Fig. 21. Model of my CNN for the Digits MNIST

After ensuring that the model does not suffer from overfitting as could apparently be deduced by looking at Fig. 22, I got the confusion matrix of Fig. 23 from test data classification.

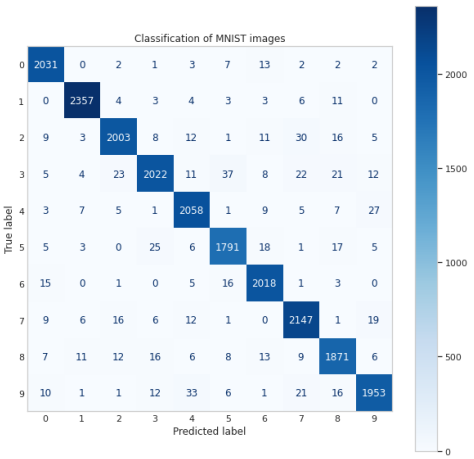

Fig. 23. Confusion matrix of the classification of the Digits MNIST test images with my CNN

Fashion MNIST

In Table 14 the model of my CNN used to classify the Fashion MNIST images of section 5.1 is described.

Hyperparameter	Value
Number of epochs	15
Number of convolutional layers	1
Number of filters	[16]
Stride per layer	[2]
Filter sizes	3
Kernel initializer	He uniform
Number of hidden layers	1
Hidden layers neurons	[25]
Learning rate (fully-connected part)	0.9
Learning rate (convolutional part)	0.1
Activation function	ReLU
Iteration drop	0.75
Batch size	1
Pre-norm	True

Table 14. Model of my CNN for the Fashion MNIST.

Once again I show in Fig. 24 how with the constructor of my convolutional neural network class we can define the values of all the hyperparameters.

```
ConvNetAbel(nEpochs=15, convFilters=[16], convStride=[2], convFilterSizes=3,
            hidden=[25], learningRate=0.9, debugLevel=2, convEpochs=7,
            showLogs=False, activationFunction='relu', learningRateConv=0.1,
            verbose=False, pre_norm=True, iterationDrop=0.75,
            kernel_initializer='he_uniform', batch_size=1)
```

Fig. 24. Model of my CNN for the Fashion MNIST

Fig. 25 shows the overall mean error of the entire network output during training instead of splitting it by classes as before, because the decrease in this way is clearer in this case. Nonetheless, the accuracy results of classifying the test subset are quite competitive and generalizes really well, as can also be seen in the test confusion matrix of Fig. 26.

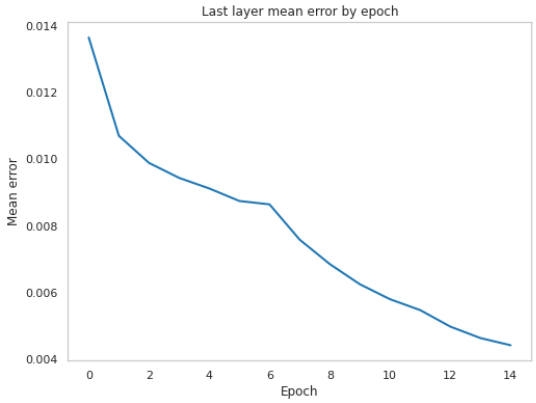

Fig. 25. Mean error per epoch during Fashion MNIST CNN training

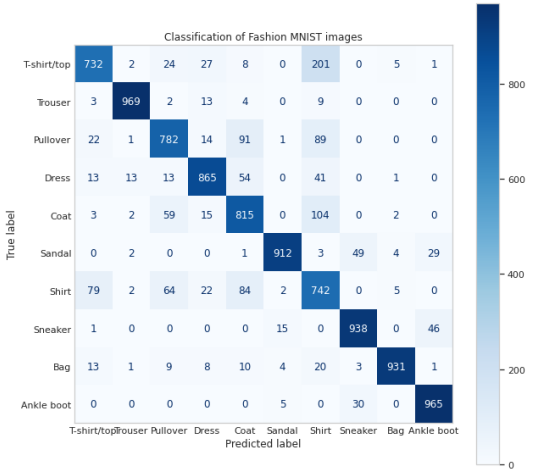

Fig. 26. Confusion matrix of the classification of the Fashion MNIST test images with my CNN