Université Libre de Bruxelles

Study of the algorithm of deep learning and the optimization of those networks (genetic algorithms)

Master in computer sciences

Cédric De Muelenare
000459031

Coordinator : Hugues Bersini
Advisor : Antonio García Díaz

August 2022

Abstract

Convolutional Neural Networks have known a gain in popularity, especially in image classification. One problem related to this type of neural networks, is the research of an architecture for such networks. Knowing basic operations that will serve as base building blocks, the objective is to build an architecture leading to good performance. The research of an architecture can be automated, and several previous works have led to good results.

One way of automating the creation of an architecture is to use Genetic Algorithms, which mimics the evolution process in the nature: as the generations pass, the individuals evolve, and get better results. In this case, better results simply means "better accuracy in image classification".

Specifically, a Cartesian Genetic Programming approach will be used, which uses a grid to represent the architectures. This grid will be modified by the evolution, increasing its accuracy, as the generations pass. Finally, the resulting structures will be analyzed, and compared to other approaches.

Using the cifar10 dataset, composed by colored images, the hyper-parameters (parameters defining the execution) of the evolution, such as the number of generations, are fine-tuned. A set of hyparameters concerning the neural network itself are also fine-tuned. Then, the performance and architectures obtained by the evolution done on the cifar10 dataset are compared to the results obtained with the cifar100 (also containing colored images), MNIST (composed by handwritten digits), and Fashion-MNIST (containing illustrations of clothes) datasets.

Table of contents

1 Introduction

Convolutional Neural Networks have proven their ability to have good performance in visual tasks such as image classification, which goal is to give a prediction about what is represented on a given image. Such networks are made of a sequence of operations on the input image, in order to give an output prediction. One important step in the process of developing a Convolutional Neural Network is the research of an architecture of operations that will give good results.

Network Architecture Search is a field of research dedicated to the automatization of the research of such architectures, and contains different approaches. Among the approaches in Network Architecture Search, Genetic Algorithms are often cited, and we will use such algorithm in order to find architectures.

Genetic Algorithms mimics the evolution theory that is in the nature; "the survival of the fittest". In such algorithms, a population, composed by individuals, will evolve and get better performance, as the generations pass. The meaning of the terms "individuals" and "performance" will depend on the context. In this case, The individuals will be architectures, and the performance will refer to the reliability of the prediction of those architectures.

Specifically, the type of Genetic Algorithm that is going to be used here is Cartesian Genetic Programming, which uses a grid to represent a sequence of operations. Each slot in the grid may contain an operation, and each operation will take, as input, the output value of previous operations. In this case, the operations of the grid will be the common operations found in Convolutional Neural Networks.

Random mutation will modify this grid of operations, which will impact (either positively or negatively) the accuracy of the prediction. In the end, the "positive" mutations will probably be found in the new elements of the population, and the "negative" mutations, will probably be "forgotten".

Given a specific collection of data (in this case, a collection of images) we will use a Cartesian Genetic Approach to find architectures. It is important to remember that all solutions are specific to a given problem, so an architecture may have a high accuracy on a given dataset, but may have a low accuracy on another dataset.

In total, 4 datasets will be used; cifar10, cifar100, MNIST and Fashion-MNIST. Those datasets are various, in terms of image sizes, and types of images (cifar10 and cifar100 contains pictures, MNIST contains handwritten digits and Fashion-MNIST contains illustration of clothes)

The program that we are planning to make is defined by a certain number of parameters ; that could be the size of the architecture, for instance. One of the aim of this project is to examine the impact of some of those parameters, called hyperparameters, on the final performance.

Another objective of this work is to examine the resulting architectures, what are their specificity and how do they compare to other approaches in Network Architecture Search. Also, the evolutionary aspect will be analyzed; how the architectures evolve, how much does the performance change.

2 State of the art

2.1 Genetic algorithms

Genetic algorithms, GA, mimics the evolution process found in the animal world (Figure 1); trough generations, individuals will get better results/fit better to the environment.

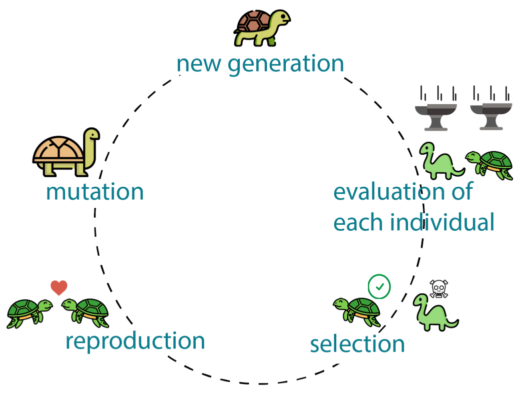


Figure 1: Evolution process, from [1]

Genetic algorithm are often used in optimization problems, and their uses are various [2] [3].

2.1.1 Biology inspiration

GA are inspired by the Darwinian evolution, in which the natural selection makes the best fitted individuals live longer, have more children and see their genes more spread. Thus the "best fit" aspect will also be seen in the future population. Figure 2 shows this idea.

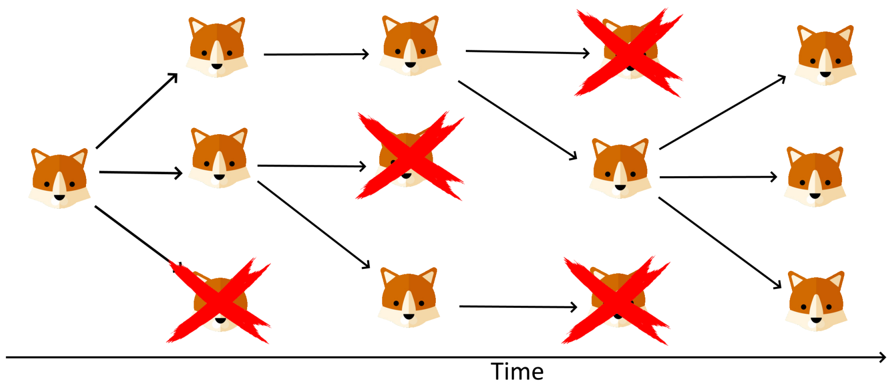


Figure 2: Evolution as found in the nature

2.1.2 Mechanisms

In a GA, the individuals are obviously not animals, but can be roughly anything that can be optimized (hyperparameters, etc.). For the sake of simplicity, the term "individuals" will still be used.

A given algorithm will use a data type to represent the genes of an individual, for example, using a matrix.

Some individuals will be better at some tasks, and all the individuals will be attributed a "fitness score", indicating how good they behave.

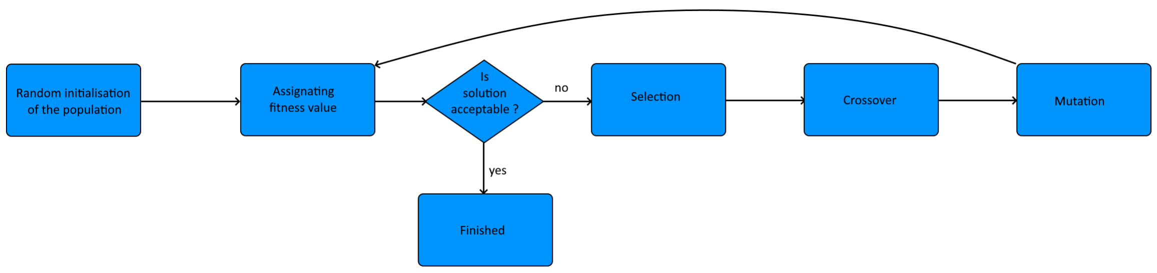
The initial generation is often randomly generated. From that point, each new generation is based on the previous one. The term "based" refers to the inheriting of the parent's genes.

Mutations can occur in a randomly, adding diversity to the genes. Some mutations will be beneficial, some may not. As the generations pass, beneficial mutations will be more likely to be found in the new members of the populations, while "negative" mutations will tend to be "forgotten", since they decrease performance.

Inheriting traits from parents adds the aspect of taking what have been found in the past. Some GA uses a two parents reproduction, but other implementations may use more/less parents. As in the nature, the best individuals will be more likely to have several children. Some genetic algorithms use a process called **crossover** which consists of "mixing" the genes of the parents, in order to create new individuals.

Natural selection allows the best fitting individuals to have their genes more spread, therefore their "best fitting" traits more spread.

The working idea of a GA is shown in the figure 3



Random initialisation of the population
Assignating fitness value
Is solution acceptable ?
no
yes
Selection
Crossover
Mutation
Finished

Figure 3: Genetic algorithm

When designing a GA, several choices have to be made;

- **Fitness function**; how to determine the fitness of an individual, what makes an individual "better" than another. This function will also depends on the problem itself
- **Presence/absence of crossover**; some approaches can use crossover functions, meaning mixing the genes of several parents, in order to produce a new individual [4]. When using a crossover approach, the number of parents also needs to be defined. A common number of parents is two [5]
- **How many individuals are kept from the previous generation**, in some cases, only one (the most performant) individual remain from each generation [6], but it is not always the case.

2.2 Artificial intelligence

2.2.1 Emergence

The first definition of Artificial Intelligence, AI, can be attributed to Marvin Minsky and John McCarthy, who first defined this concept in 1955 [7]. The initial idea behind the concept of "Artificial Intelligence" is "Any attempt, from a computer, to replicate human intelligence" [8].

Nowadays, AI will mostly be used to learn a behavior, as explained in the following section.

2.2.2 Idea

An AI can be seen as a black box, taking input, and producing an output, as shown in figure 4. The objective of an AI is to learn to the black box how to replicate a behavior.

Input(s)
Black box
Output(s)

Figure 4: Simple representation of an AI

The behavior, replicated by an AI, can be various. This could be health diagnosis [9], with the inputs being all the information about a patient, and an output being whether this patient has a certain disease or not.

Another example of behavior replicating is self-driving cars [10], where the inputs are all the data about the environment of a car (speed, direction, other cars nearby, etc.) and the output is the actions of the cars (accelerate, brake, turn, etc.)

AI is a vast term, composed by a number of sub-categories, some of which are described in the following sections.

2.3 Machine learning

Machine learning, ML, is a special type of AI, in which an AI goes through the process of learning[11]. This is, by far, the most used subset of AI, and contributed to the explosion in popularity of AI.

2.3.1 Emergence

This term first appeared in 1959, defined by Arthur Samuel; *"the programming of a digital computer to behave in a way which, if done by human beings or animals, would be described as involving the process of learning"*[11]

2.3.2 Idea

A ML procedure will require a training phase, in which the behavior itself is learned. Using that black box analogy previously used, that phase will correspond to the black box adapting itself, in order to improve its performance, and give an output that is closer to what the output is expected to be. The term "prediction" is often used, when referring to the output.

Often, the training phase is followed by a testing phase, which will not modify the AI anymore, but will just return an indication of the performance of the AI.

Concretely, once the program of a ML based AI is written, the same code could be reused to do other tasks, because what this code does is explain to a computer the process of learning.

A ML may use a **dataset** (but it is not mandatory [12]), in which case the AI will be asked to generalize a concept included in this dataset.

2.3.3 Training phase

This phase corresponds to the learning part itself. The goal of this phase is to learn how to generalize, how to predict an output on a given set of input.

Usually, at the start of the procedure, the AI will predict an output in a near random way. Then, using the data, the AI can adapt itself.

This process is usually done by using the input data that is at our disposal (either in a dataset, or not), and depending on the output of the AI and the "real output" (what should have been predicted); the AI should adapt itself.

Often, the input data are not given one after the other, but they are combined into what is called a **batch**, in which case the **batch size** hyper-parameter have to be defined. Batch size can have a consequent impact on the performances of the AI [13].

2.3.4 Testing phase

This phase does not impact the AI anymore, but it allows obtaining an indication of the performance of the AI. This process is done by iteratively taking new input data, and present those data to the AI. Then, by computing the proportion of "good predictions", we can have an idea on how well the AI learned the asked behavior. Another indicator of the performance is the loss function, indicating how far the prediction is from the "good" answer.

This testing phase should be done on data that have never been presented to the AI before, in order to make sure that the AI correctly learned to generalize a concept.

2.3.5 Using the dataset

As previously mentioned, in order to measure the performance of an AI, it should be given data that has never been seen before. That way, a dataset is generally split into two parts;

- **Training set** is used to train the AI
- **Validation set** is used to evaluate the performance of the AI

The operation of feeding once the training set, to train the network, is called an **epoch**. In some cases, it may be beneficial to use more than one epoch. In other words; in some case, it is beneficial to "show" the same dataset multiple times. The number of epochs is a hyperparameter.

2.3.6 Fitting tradeoff

The goal of an AI is to learn how to generalize a concept. If the AI did not succeed to have good performance, this can be due to 2 problems (those notions does not only apply to AI, but are used more generally in statistics);

- **Overfitting** is the case in which an AI is sticking too closely to the data that has been provided to it; it did not learn to generalize
- **Underfitting** is the opposite; the AI did not learn enough of the dataset.

A comparison between these two cases, as well the "real answer" are shown in the figure 5

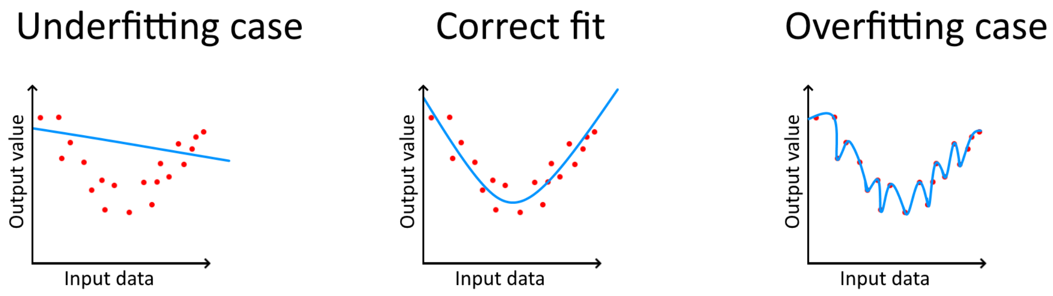


Figure 5: Fitting dilemma

2.4 Artificial Neural Networks

Artificial Neural Networks, ANN (also called Neural networks, NN) is a type of ML. The concept of NN were first described by Warren McCulloch and Walter Pitts in 1943 [14], and are mostly inspired by the animal brain.

2.4.1 Idea

The idea of a NN is similar to the animal brain; neurons will communicate impulsion to other neurons, and so-on.

NN are generally organized in multiple layers, and each layer of neurons will receive the impulsion of the previous layer. Impulsion, in a NN, will be represented by numbers, and those numbers will be multiplied by coefficients, called weights, before reaching the next layers of neurons. Those weights are the result of the training phase, and represent the "strength" of each connection between neurons.

A NN generally contains

- **1 Input layer**, the input data
- **0 or more hidden layers**, corresponding to the black-box itself. Improving the number of hidden layers will increase the abstraction level between the input and the output, which could improve the "intelligence" of the network, but it is not always the case [15].
- **1 output layer** representing the output prediction.

Figure 6 shows a graphical representation of a NN, having one hidden layer.

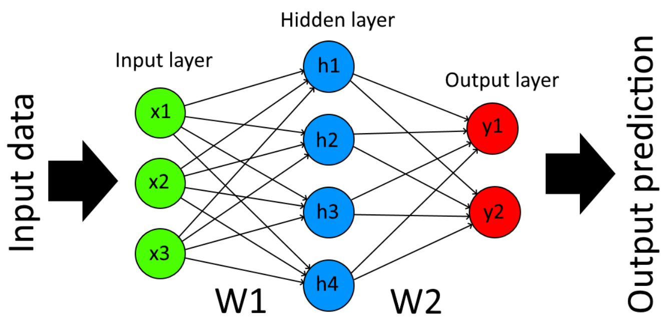


Figure 6: Simple neural network

Several variations of a classical NN can exists such as recurrent neural networks, RNN, where loops are allowed [16].

2.4.2 Representing data

As previously said, the data flowing through the NN will be represented as real numbers, either positive or negative. Each layer may contain several neurons, thus several numbers, so the state of each layer will be represented using a matrix. The layers of the network will be multiplied by a weight matrix, and the result of this operation will define the next layer.

Figure 7 shows a graphical representation of a simple NN, with one layer and one neuron.

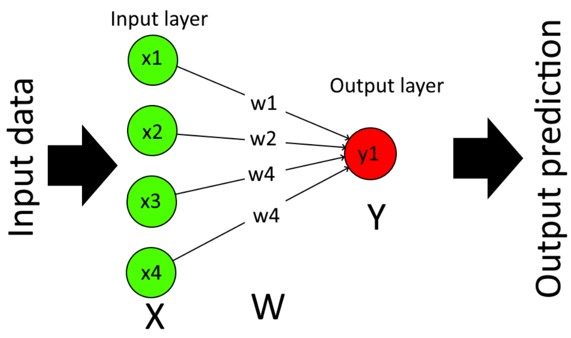


Figure 7: Simple neural network

Initially, the weights are randomly generated. As the data are successively shown to the NN, the weights should converge towards an optimum.

A forward pass 2.4.3 operation consist of obtaining the output prediction in function of an input, and the backpropagation 2.4.5 is the action of adapting the weights.

2.4.3 Forward pass

A forward pass consists in the action of feeding the NN with one set of input data, and progressing through the neural network, until the final layer is reached. Considering the example shown in the figure 6, the value of the hidden layer will be computed as

$$H = X * W1$$

where H is the value of the hidden layer, X represent the input data and $W1$ represent the first weight matrix. Then, the output can be computed as

$$Y = H * W2$$

where Y defines the output, H defines the hidden layer and $W2$ is the second weight matrix.

2.4.4 Additional features

Activation function . In some cases, an activation function may be added to a given layer. This activation function takes as input the data given by the previous layer, and modifies this impulsion before following the procedure. Two well known activation functions are mentioned here;

- **ReLU**, Rectified Linear Unit; $f(x) = max(0, x)$
- **Sigmoïd**; $\frac{1}{1+e^(-x)}$

Whose evolution are shown in the figure 8. The choice of an activation may not be easy and could have a considerable impact on the performance [17]

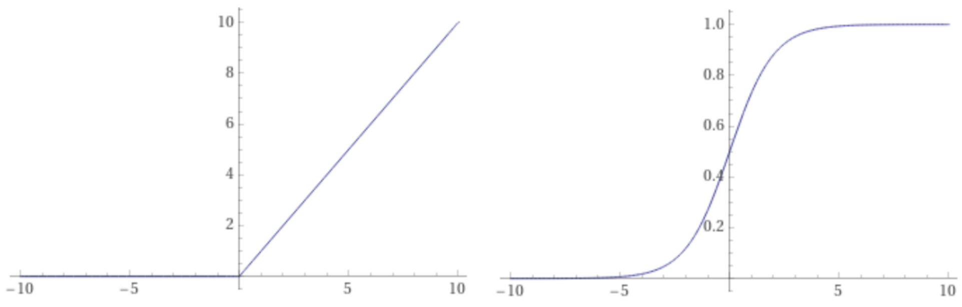


Figure 8: ReLu (left) and sigmoid (right) functions

If A is the following layer, W is a weight matrix and B is the previous layer, using an activation function gives a computation of the next layer as such;

$$A = f(W * B)$$

Batch normalization is a technique that consist of re-centering the layer's value (considering the entire batch), improving stability [18].

2.4.5 Backpropagation

The backpropagation algorithm consist of going from the end of the NN, knowing both the prediction output and the "real value", to the beginning of the NN, while adapting the weights. The goal is to adapt weights, improving the quality of the prediction [19]. This process also takes in account the presence/absence of an activation function.

2.4.6 Deep learning

Deep learning is a certain type of NN in which the number of hidden layers is high. A high number of layers means a lot of abstraction, between the input and the output, and as mentioned in the figure 2.4.1, adding more layers may improve the intelligence of the system.

Figure 9 shows a graphical representation of a deep learning network.

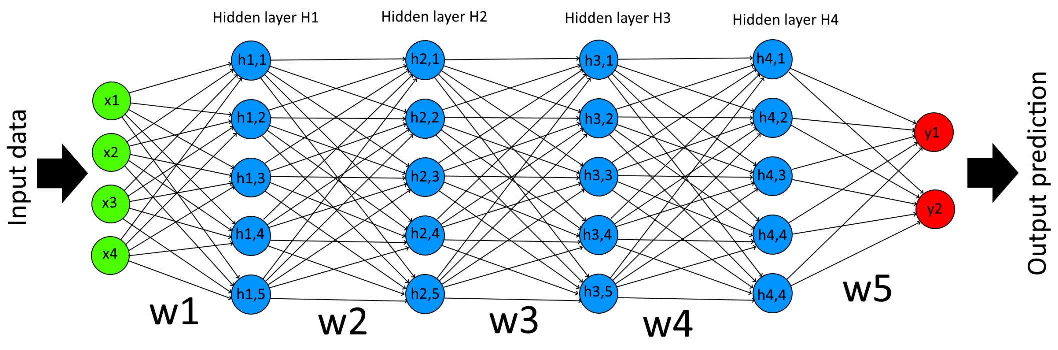


Figure 9: Deep learning

2.5 Convolutional neural networks

Convolutional neural networks, CNN, are a certain type of deep learning network in which a feature extraction part is added. The features are then given to a classical NN. They are particularly efficient in image classification [20]. An example of CNN is shown in the figure 10

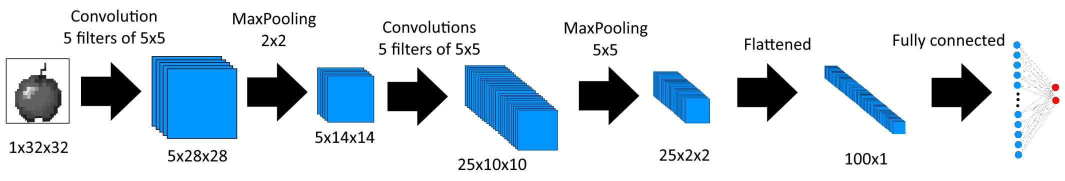


Figure 10: Example of CNN

2.5.1 Emergence of CNN

The history of the CNN started when D.H.Hubel and T.N.Wiesel made important discoveries about the working of the visual cortex of animals, specially cats [21]

2.5.2 Representing the data

Usually, CNN are used on images, so the data considered will be images, and those images will be represented using matrices.

A monochrome 2 dimensions image can be represented using a 2 dimensions matrix, each pixel being represented by a number. The higher this number is, the closer it is to white, a lower value is closer to black.

Similarly, colored 2 dimensions image can be represented using 3 classical 2 dimensions matrices. Each pixel is thus represented by 3 numbers, one for each primary color (red, green and blue). Usually, those numbers are between 0 and 255 (mainly because modern screens can display 256 colors levels per primary color per pixel)

2.5.3 Feature extraction

The feature extraction part will take as input the image (high size matrix) and summarize it into a lower size vector, representing the features of the image. The goal is to lose as little information as possible during this process. This lower size vector will then be used by the classical NN to make a prediction.

2.5.4 Producing output

The feature vector is then given, using a classical NN taking, as input, the features. This part follows exactly the "rules" of a NN, and can, for example, contain hidden layers.

2.5.5 Base building blocks

The entire network that will, in the end, be used to predict the class of a given input image, is composed of base building blocks, which are usually quite simple operations. These blocks will take matrix as input, and return matrix as output. The dimension of the input and output data can (and often will) change in dimensions. As explained previously, those building blocks will generally decrease the dimension of the data, in order to give few input to the NN.

Several blocks are often used, and this section will refer to the most commonly used.

Input block correspond to the input data, and is placed at the start of the network. In the case of a colored image, this block will produce three 2D arrays, each one representing a primary color.

Convolution blocks are used to extract some patterns of a given image. This block uses (usually multiple) filters, also called kernels, which describes exactly the pattern that is looked for.

Figure 11 shows an image going through a convolution block composed by 4 filters.

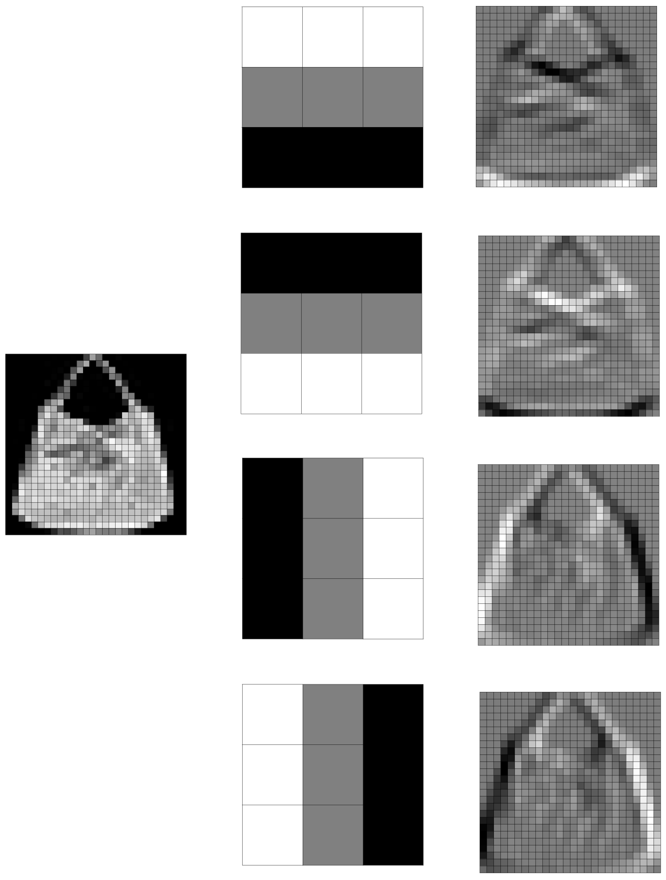


Figure 11: Example of image, going through a convolution composed by 4 filters

The first step in a convolution phase is, for each pixel, to compute the multiplication between every pixel of the filter and the surrounding of this pixel in the original image. Then the sum of every element obtained will be the value of the new pixel in the resulting image, as shown in figure 12.

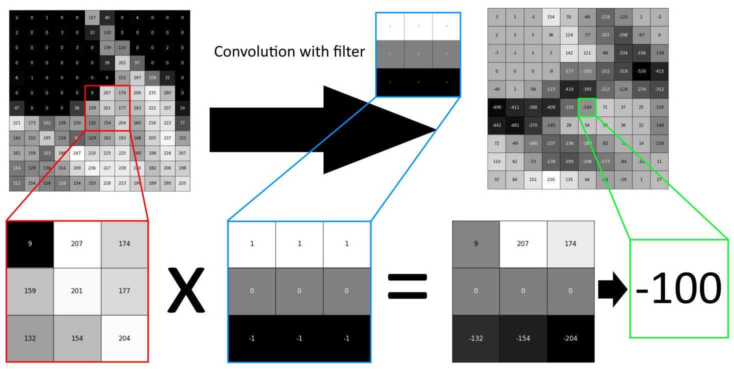


Figure 12: Example of image going through a convolution containing one filter

The pixels at the borders of the image do not possess a neighborhood, therefore the resulting image will be slightly smaller than the original one. In order to keep the same image size, the convolution blocks are often coupled with a **padding block**.

The number of output images will be defined by the number of filters.

After a convolutional block, it is common to add a **Batch normalization block** in order to keep the same order of magnitude in the values. Also, convolutional blocks are often followed by **activation function block**.

Pooling are a type of block that reduce the size of each image. Each pixel and its neighborhood in the original image will be "summarized" by one pixel in the output image. Several types of pooling can be used, such as the **MaxPool** which consist of taking the maximum value of a neighborhood of the input value to define the output value. Similarly, the **Average pooling** consist of taking the average value of the input value in order to define the output value. The size of the reduction is a parameter of this block. An example of MaxPool and Average pool are shown in the figure 13.

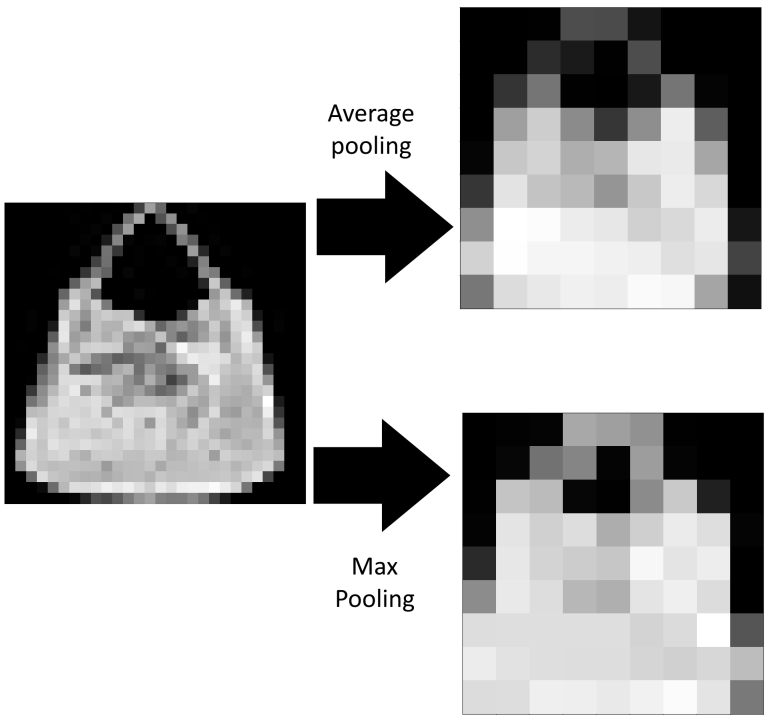


Figure 13: Example of pooling operation

Activation function does not change the size/number of the images, but change directly their values. The previously mentioned **ReLu** (cf 2.4.4) will remove all the negative values, and replace them by 0. An example of ReLu application on an image is shown in the figure 14.

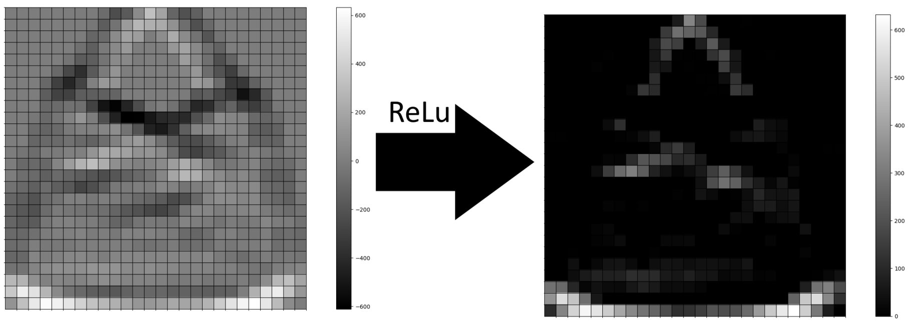


Figure 14: Example of ReLU operation

Padding is a block that will slightly increase the size of each image, in order to aim for a specific size. This process is done by adding values to the border of the image. The new values themselves can be 0, in which case we refer to **zero padding** [22]. Another common padding block is the **nearest neighbor padding**, where every added pixel has the same value as the closest pixel in the original image. Figure 15 shows an example of those two padding techniques.

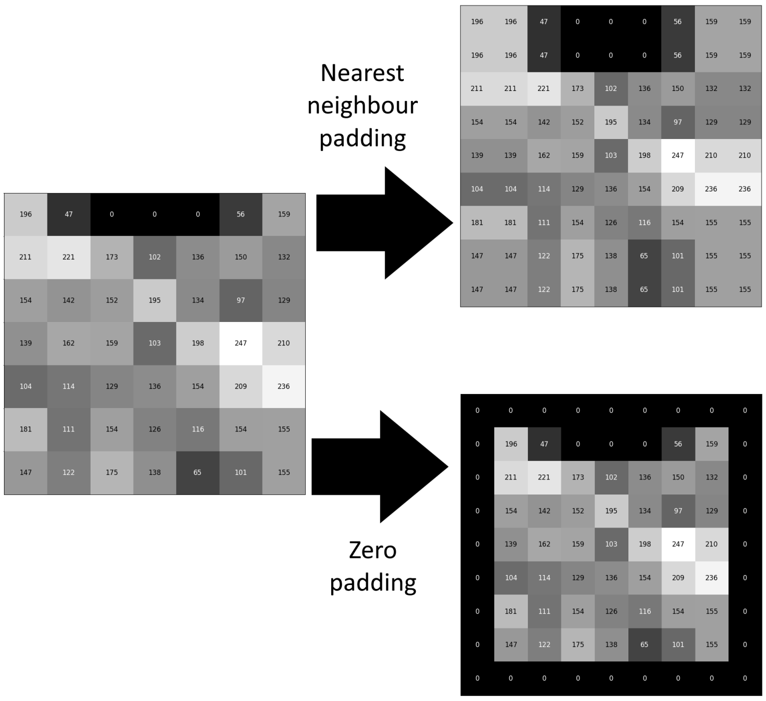


Figure 15: Example of padding operation

Concatenation block will concatenate the output of multiple blocks into one matrix. It requires that all the input images have the same dimensions.

Summation block adds the content of multiple images. This block also require that all the input images have the same size (because matrix addition implies it)

Res block are mainly found in residual networks, consists of keeping connection with blocks that are located closer to the start of the network. Those blocks allow having some advantages of the abstraction of a high number of hidden layers, without having the drawback of a phenomenon called gradient vanishing [23]. As **Convolution blocks**, they are often grouped with **padding, batch normalization** and **an activation function**

Batch normalization block will normalize data, taking in account the entire batch, so that they are all in the same order of magnitude.

Fully connected layer is the classical NN, usually placed at the end of the structure.

2.5.6 Well Known Architectures

One architecture is not intrinsically better than the others, because it always depends on the problem itself. Different types of images (drawings, writing, photos, etc.) could require different architectures, because different types of patterns will be found on the images.

Previous works have found architectures leading to good performance in image classification

A first example of known architecture can be found in a journal published in 2020 by Mukherjee, S et al. [24], where it was used to classify handwritten characters. Its architecture is shown in the figure 16.

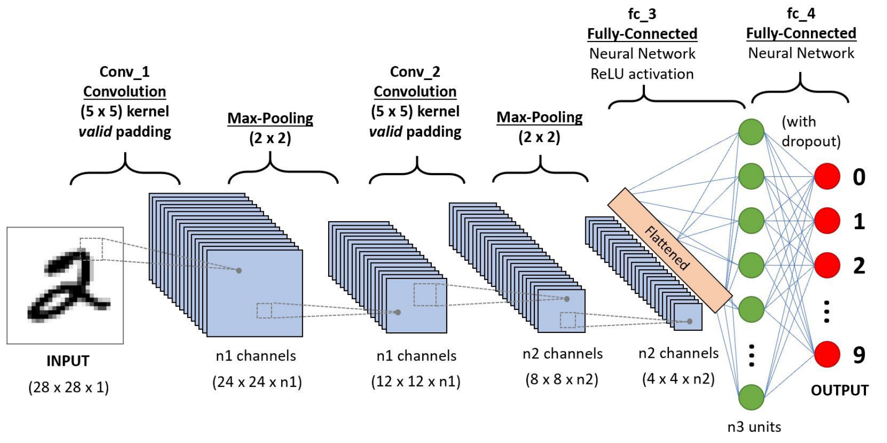


Figure 16: First example of known architecture for a CNN, taken from [24]

VGG Among the most famous architectures, **VGG** is often cited. VGG was initially designed to make image classification between 1000 classes. VGG exists under 2 variations; VGG16 17 and VGG19.

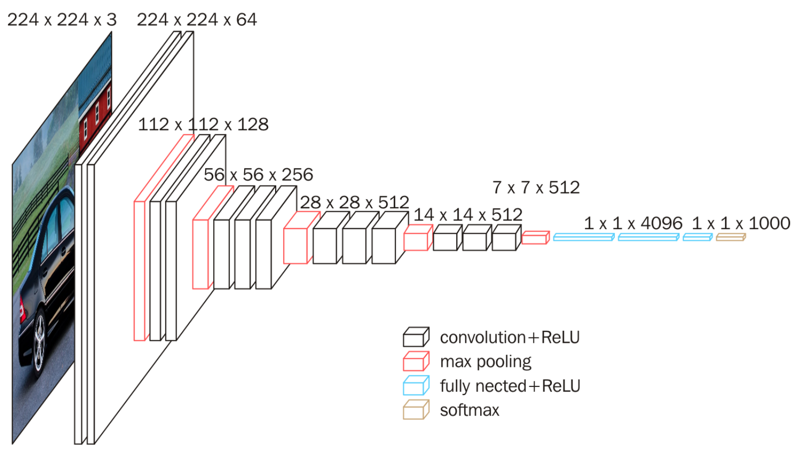


Figure 17: VGG16, taken from [25]

ResNet is a family of architecture that uses the residual block, described earlier. Figure 18 shows ResNet18, one of the variations of ResNet

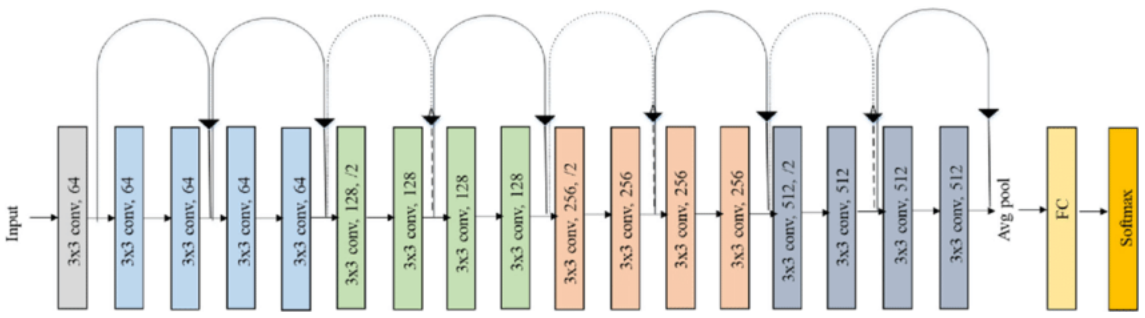


Figure 18: ResNet18 [26]

2.5.7 Summarizing the place of CNNs in AI

Figure 19 shows the place of CNNs in AI

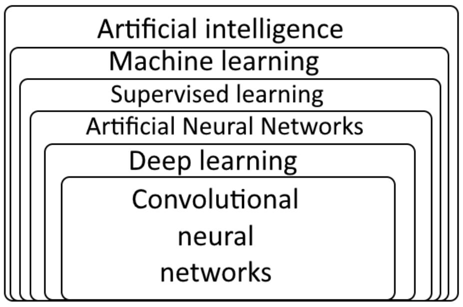


Figure 19: The zoo of AI

2.6 Network Architecture Search

CNN architectures can be found automatically. This field is called NAS (Neural Architecture Search) and contains number of different approaches.

NAS is a difficult task for a human, for several reasons;

- Humans can be biased
- NAS is a time-consuming task

For those reasons, NAS is often done by a computer. NAS is an optimization problem, thus, the approaches may correspond to optimization-solving algorithms.

The usual objective is to obtain an architecture that maximize only the accuracy of the prediction of an architecture, but it is not always the case; **Multi-objective search** is a subfield of NAS in which the goal is not only to maximize the accuracy, but also potentially minimize the size of the resulting architecture in order to save, for example, memory and/or power consumption [27]

2.6.1 Approaches

This section is a non-exhaustive list of the approaches that exists in the field of NAS

Reinforcement learning in which the usual procedure is to reward the best architectures [28]. This approach can also be combined with multi-objective search [29]. Another way Reinforcement Learning has been used is via a Reccurrent Neural Network that produce an architecture [30]

Genetic algorithms approaches, some uses crossover between parents [31] [32] other approaches does not use crossover, and thus relies only on mutations for the evolutionary aspect [6] [33]. Some approaches also combine Genetic algorithms with the multi-objective search mentioned earlier [27]

Gradient descent approaches will search for a local maxima (in terms of accuracy) or a local minima (in terms of loss) [34] [35].

Growing techniques build an architecture from scratch, layer by layer [36] [37]

2.6.2 Ways to accelerate the search

NAS is computationally expensive, and in some cases, "tools" can help to diminish the execution time

Performance predictors allows having an approximation of the accuracy of an architecture (instead of proceeding to a complete train-test procedure) [38][39][40]

Re-use previous knowledge diminish the train time, as there exist similarities between the new and the old architectures [41]

Dense-sparse-Dense is a technique that train a subset of the network, instead of the whole network, in order to save computation time [42]

2.7 Cartesian genetic programming

Cartesian genetic programming, CGP, is a type of GA that use a grid to represent the individuals of the population.

Each slot of the grid can contain one operation, and as the generations pass, the grid of the best individuals should, improve the fitness score of the individuals. Each operation, that will also be called nodes, take as input the result of one or more other operations.

The mutations will randomly add, remove or change elements of the grid.

Figure 20 shows an illustration of a CGP grid

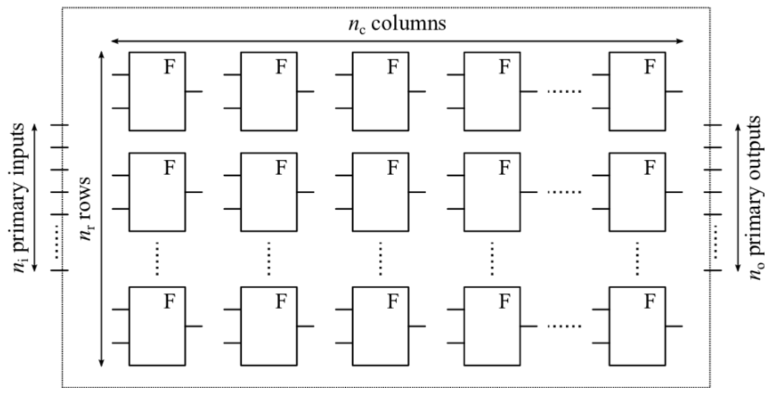


Figure 20: CGP grid, taken from [43]

CGP has been used mainly in the guessing of complex functions [44].

In addition to the selected settings for the GA, other choices have to be made;

- **Levels back** is the number of layers that a connection can go back in order to reach the result of a previous node
- **Dimension of the grid**; height and width
- **The set of functions available** which will depends on the problem itself
- **Minimum and maximum number of nodes** will constraint the number of nodes, i.e. the size of the function

- **Number of input taken by each node**

2.7.1 Initialization

The initial grid of operation is randomly generated. Each slot in the grid is generated randomly, inlcuding

- The type of node
- Potentially the settings of the nodes
- Each input taken by this block

2.7.2 Evolution process

Once the inital grid have been randomly generated, the evolution process takes place.

During that process, each generation is composed by several **offsprings**, which are potentially generated by applying a crossover between the parents, in addition to the mutations. The evolutions process goes as follows; the fitness score of each offspring (grid) is computed, then the future generations will take in account the individuals having the best fitness score

2.7.3 Suganuma implementation

One example of work that uses CGP was done by suganuma [6] . In this case, the CGP aspect is used to evolve the architecture of a CNN; each element of the grid is one of the operations mentioned in the figure 2.5.5.

This implementation uses;

- A fitness function that is the score of an architecture
- No crossover, each individual is generated from one parent. The evolution process relies thus only on random mutation.
- One individual is kept from the best generation; the one having the best fitness function

- In this implementation, each node can theoretically take the input of 2 other nodes, but some nodes will only use the first of the two inputs (maxPool, fully connected etc.).

Parameters of the CGP part can be modified, and the set of available functions is described here;

- **Input layer**, at the start of the architecture
- **Convolution** block, and those block can have 3 sizes of kernels; 1, 3 and 5. A convolution block applies padding before applying the convolution. Then, the result goes through a batch normalization, and finally a ReLU function in applied to the result. The number of outputs per convolution, referred to as output channel, can take the value 32, 64 or 128.
- **ResBlock** consists of two convolution blocks, also followed by a batch normalization and a ReLu, adding a connection that "skips" the block. Similarly to the ConvBlock, this block can have a kernel size of 1, 3 or 5 and its output size, may also be 32, 64 or 128.
- **Concat** block concatenate the output of two previous blocks
- **Sum** adds the result of two previous blocks
- **MaxPool** does a max pool operation on the result of one block, always taking a kernel size (size of the reduction) of 2.
- **AveragePool** does the average pool operation on the result of one block, also taking a kernel size of two.
- **Fully connected** at the end of the network, and no hidden layers

In total, including the variations of each block, there are 22 blocks in total.

Note: original paper [6] splits the executions using the convolution blocks and the executions with the res blocks. Here, both will be used

The rest of the CGP settings are hyperparameters, that can be defined by the user.

Working idea The input node, corresponding to the image itself, can be used by the "hidden nodes". Finally, the output node, the fully connected layer, will produce the output.

Each slot in the grid always contains one operation, but part of the nodes will not be used to produce the output, in which case they become useless. Such nodes are deleted, in order to save computation time.

The remained nodes are known as **active nodes**.

For the rest, a GA is applied to the grid

Representing the genes is done using a list of the nodes, each node being represented by

- Type of operations and settings of that node
- Two inputs nodes, with the second one being possibly ignored if not used; Concat blocks and Sum blocks are the only blocks using both input, every other type of nodes uses only one input (ConvBlock, etc.)

Mutations As explained, the mutations will modify randomly an individual. The mutations can be the addition, deletion, or the modification of one block. The fitness score of each individual (which is the accuracy) is first computed, with a train-test procedure, the fitness being indicated by the accuracy of an individual.

3 Methodology

3.1 Implementation

The implementation that will be used is based on a previous work[6] in which CGP is used in order to find a CNN architecture.

The implementation[45] was done using python3, and most of the network related operations are done using *pytorch*. No real modifications were brought to the code, beside the added ability to use the 4 datasets; cifar100, MNIST and Fashion-MNIST datasets.

In the following sections, we will use those two terms;

Re-train an architecture is the action of starting from an architecture, initializing the weights randomly, and feeding it with the train and validation dataset. This process will be followed by a test procedure with the test dataset.

Re-evolute means re-starting the evolution, without taking any architecture as a basis. This process will use the train test in order to train each architecture, and the validation set to assert its performance.

3.2 Computer used for the benchmarks

All the experiments have been done on a desktop computer having the following specifications;

- Intel core i5-3470 @3.2GHz
- NVIDIA GTX 950
- 16GB RAM

Concerning the software part, all the Cuda drivers were installed, and the operating system is Linux Ubuntu 20.04.3 LTS

3.3 Dataset

3.3.1 Datasets used

We chose to use 4 datasets;

cifar10 is a dataset containing 10 different classes, and contains colored 32x32 pictures.

cifar100 is a dataset containing 100 different classes, and contains colored 32x32 pictures.

MNIST is a dataset containing 10 classes of hand drawn digits, and are 28x28 monochrome images.

Fashion-MNIST is a dataset containing 10 classes of clothes, and are 28x28 monochrome illustrations.

A subset of each dataset is shown in the figure 21

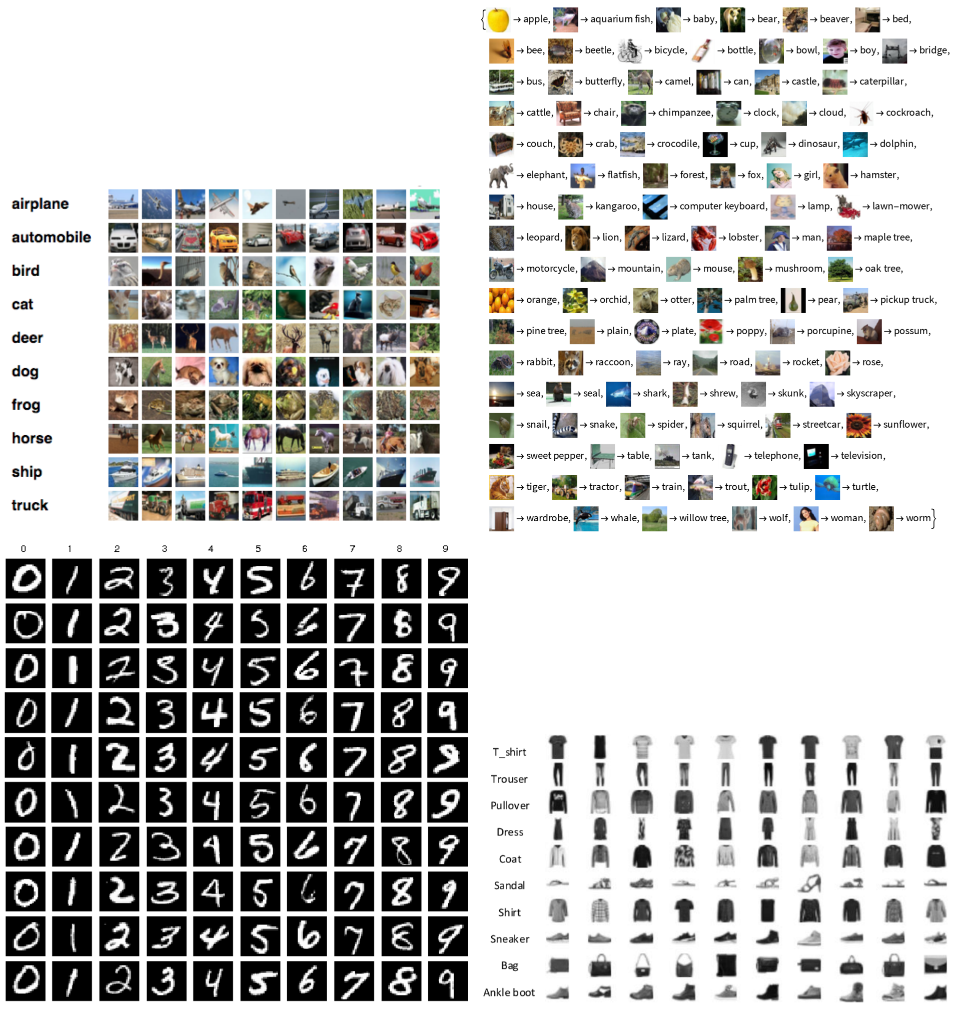

{→ apple, → aquarium fish, → baby, → bear, → beaver, → bed,
→ bee, → beetle, → bicycle, → bottle, → bowl, → boy, → bridge,
→ bus, → butterfly, → camel, → can, → castle, → caterpillar,
→ cattle, → chair, → chimpanzee, → clock, → cloud, → cockroach,
→ couch, → crab, → crocodile, → cup, → dinosaur, → dolphin,
→ elephant, → flatfish, → forest, → fox, → girl, → hamster,
→ house, → kangaroo, → computer keyboard, → lamp, → lawn−mower,
→ leopard, → lion, → lizard, → lobster, → man, → maple tree,
→ motorcycle, → mountain, → mouse, → mushroom, → oak tree,
→ orange, → orchid, → otter, → palm tree, → pear, → pickup truck,
→ pine tree, → plain, → plate, → poppy, → porcupine, → possum,
→ rabbit, → raccoon, → ray, → road, → rocket, → rose,
→ sea, → seal, → shark, → shrew, → skunk, → skyscraper,
→ snail, → snake, → spider, → squirrel, → streetcar, → sunflower,
→ sweet pepper, → table, → tank, → telephone, → television,
→ tiger, → tractor, → train, → trout, → tulip, → turtle,
→ wardrobe, → whale, → willow tree, → wolf, → woman, → worm}
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck
0
1
2
3
4
5
6
7
8
9
T_shirt
Trouser
Pullover
Dress
Coat
Sandal
Shirt
Sneaker
Bag
Ankle boot

Figure 21: Sample of the used datasets, taken from [46] [47] [48] [49]

3.3.2 Dataset separation

The dataset will be split into 3 parts;

- **Train set** to train an architecture
- **Validation set** to assert its performance (but also used to train on the retrain procedure, in addition to the train set)
- **Test set** to, at the end of the procedure, using the best architecture found, test the performance of the CNN on data that has never been seen previously.

The 4 datasets used are already split into a train set and a test set, which sizes are shown in the table 22

Dataset	Size of the training set	Size of the testing set
cifar10	50 000	10 000
cifar100	50 000	10 000
MNIST	60 000	10 000
Fashion-MNIST	60 000	10 000

Figure 22: Size of the mentioned datasets

Additionally, the training set will be shuffled, in order to remove part of the deterministic aspect.

As explained, in this case, we need to split the train dataset into two parts; training set and validation set. This separation will be done by removing part of the training set, and use the removed data in the validation set. The proportion of the training set assigned to the validation set will be a fixed percentage, and should neither be too high (in which case, the number of images used to train the network will be too low) neither too low (in which case the accuracy may not be relevant, as it was computed on too few examples)

3.4 Proceeding to tests

In the experiments, we will mention the score of an architecture, referring to the median score (of a certain number of re-train procedures), using that architecture, trained on the train and validation dataset and tested on the test dataset. We chose to proceed to several retrains because the retrain procedure contains a shuffle part, which is subject to randomness.

Several hyperparameters will be let constant, and not analyzed;

- **Mutation probability** will be set to 10%
- **Max levels back of the CGP grid** will be set to 15

- **CGP grid size** will be 15x5
- **Individuals per generation** will be 20
- **Number of generations** will be 30
- **Minimum and maximum number of active nodes** will be set to 1 and 10

3.4.1 First set of experiments: impact of the parameters

In that first set of experiments, we will change some hyperparameters, in order to examine the impact on the final results. After that, we will set those parameters to the values that gave the best accuracy.

The drawback of this strategy is that we could reach "local maxima", but it would probably not be far from the best achievable score.

Each of the following parameters will be fine-tuned;

- **Quantity of retrains needed**, in order to have a high confidence about the median score
- **Batch size** is a tradeoff, should be neither too high, neither too low
- **Number of epochs of training, during the "architecture training phase"**, which will only affect the architecture selection phase.
- **Number of training epochs, during "retrain phase"**, which will affect the final training, once an architecture have been selected.

3.4.2 Second set of experiments: insights on the selected architectures

In this series of experiment, some parameters will be the set of default values, as shown previously, and the rest of the parameters will be the one found in the previous set of experiments.

The goal of this series of experiment is to learn more about the evolution part AND the architecture produced by such evolution.

To do so, a certain number of re-evolution will be made on each dataset.

Experiments on the evolutionary aspect. Some indicators will be noted at each generation, in order to analyze their evolution at the end. The analyzed indicators are the following;

- Probability of finding a better architecture (than the architecture previously used)
- Size of the architecture, in number of parameters
- Validation score
- Time taken by the evolution

Experiments on final architectures In this batch of experiment, each of the architecture obtained during the evolutionary aspect experiments, will be analyzed on those indicators;

- Time taken, for the evolution and the retrain procedure
- Size of the architecture, in number of parameters
- Test score
- Time taken by the retrain procedure
- Benchmark on other datasets; During this experiment, each of the obtained architectures (obtained during this set of experiments) as well as one "control" architecture (from the well known architectures, shown in the figure 16) will be subject to a benchmark on each dataset. Then, a correlation between the dataset used in the evolution phase and the accuracy on each dataset will potentially be found.

- Analyzing the correlation between the architecture size and the accuracy first using a graph, presenting each variable on one axis, and also by using the coefficient of correlation, in order to determine whether the size of an architecture will be related to its final performance.

Analyzing the best found architectures; from each dataset, the best architecture will be shown.

4 Results

4.1 First set of experiments : impact of the parameters

Experiments in the 3 first experiments were done on one architecture, obtained by evolving on the cifar10 dataset.

4.1.1 Quantity of retrains needed

For that first experiment, one architecture was selected, and 100 retrains were made. A box plot of the results is shown in the figure 23. On those 100 retrains, the sample standard deviation was 0.70%

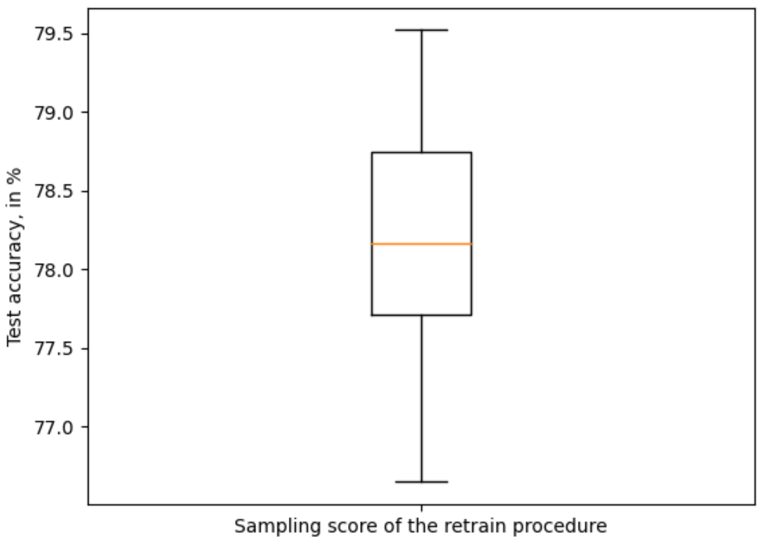


Figure 23: Sample variability of the retrain procedure

As we can see, even with 100 retrains, the variability is not high, the difference between the best score and the worst score is not consequent. Therefore, using few retrains to have an idea of the performance of an architecture will still be reliable. We arbitrarily set this parameter to 5.

4.1.2 Batch size

In that second experiment, one architecture was also selected and 100 retrains were made in each batch size setting. The result is shown in the figure 24.

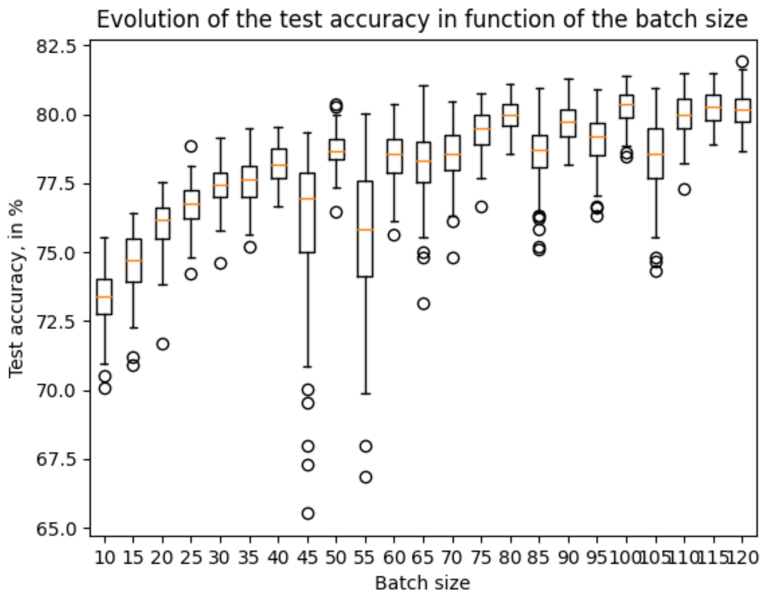


Figure 24: Evolution of the validation score with the batch size

The correlation between the batch size and the score is not consequent. It seems that the score increase slightly until a batch size of approximately 30. For that reason, we use a batch size of 30.

4.1.3 Number of epochs for training phase

In, that section, 10 re-trainings are made for each of the selected epochs number during the training phase (after the evolution process). The result is shown in the figure 25.

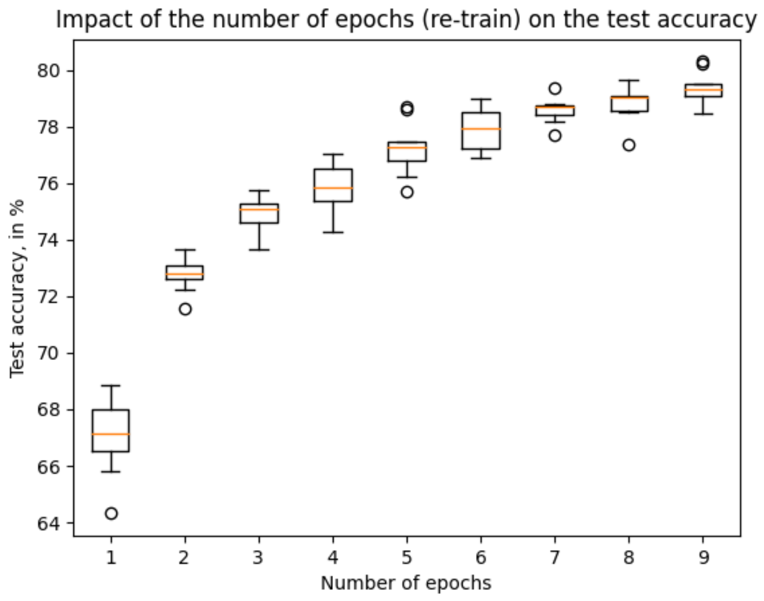


Figure 25: Evolution of the validation score and test score with the number of epochs during training phase

As expected, the number of epochs have a considerable impact on the score. We choose to use a number of epochs of 8 (we did not use more in order to avoid overfitting)

4.1.4 Number of epochs for evolution phase

In, that section, 15 re-evolutions are made for 2 different number of epochs during the architecture finding.

Accuracy is shown under a boxplot form in the figure 26 and some statistics are shown in the table 27

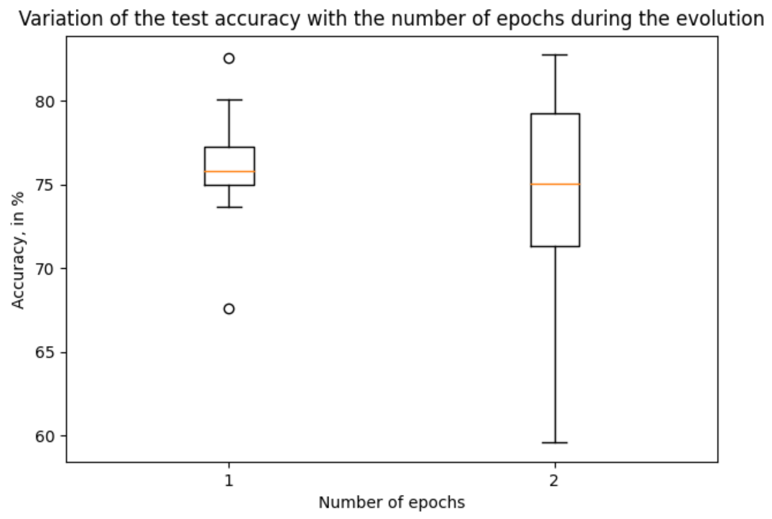


Figure 26: Evolution of the test accuracy with the number of epochs in the evolution phase

Epoch number	Median accuracy	Sample standard deviation
1	75.8%	3.32%
2	75.06%	6.032%

Figure 27: Statistics about the accuracy with the number of epochs in the evolution phase

Those results show that, for such number of epochs, the difference between the median scores is not significant. However, the standard deviation seems to be higher with 2 epochs during the evolution phase. This is probably due to the fact that a higher number of epochs means potentially explore bigger architectures, leading to more diversity, and thus, more standard deviation. We can expect bigger differences in standard deviation with higher number of epochs. The difference between the number of epochs here is not great, but we can expect that with much higher number of epochs, the median score will be higher.

Architecture size is shown under a boxplot form in the figure 28, and some statistics are shown in the table 29

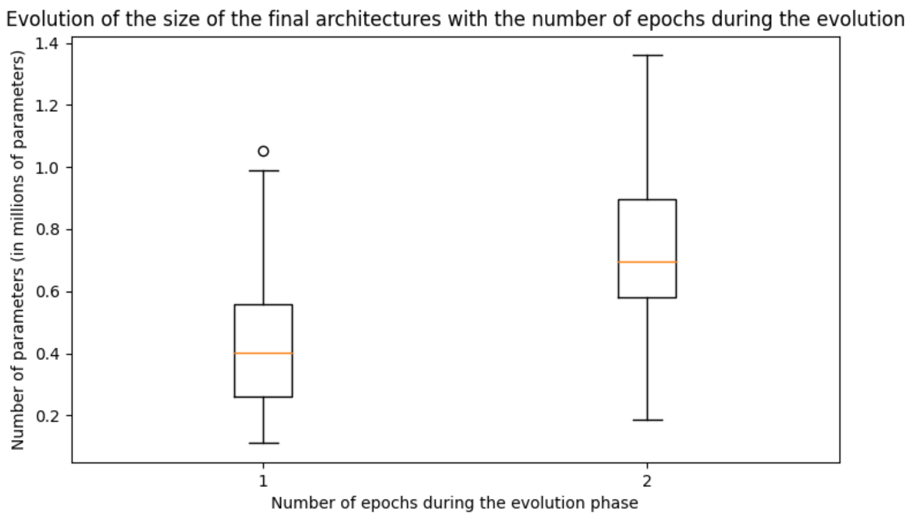


Figure 28: Evolution of the size of the architectures with the number of epochs in the evolution phase

Epoch number	Median number of parameters	Sample standard deviation
1	402 506	271 836.84
2	694 218	314 350.09

Figure 29: Statistics about the architecture sizes with the number of epochs in the evolution phase

Those results are compatible with our last hypothesis; A low number of epochs will promote the selection of smaller architectures, because the bigger architectures will not have the knowledge to have sufficient performance.

Also, a higher number of epochs will induce a higher standard deviation in the size of the architectures.

Considering the benefit added by a high number of epochs (nearly no improvement in accuracy, but bigger architectures), and the cost of added epochs (multiplication of the execution time), for the following experiment, we will use a number of epochs during the evolution of 1.

4.2 Second set of experiments : Insights on the selected architectures

4.2.1 Evolutionary aspect

For the following results, the evolution process were executed on each dataset for 15 times, and the evolution of some indicators are analyzed.

Frequency of discovering new architectures is shown in the figure 30, using mean and standard deviation.

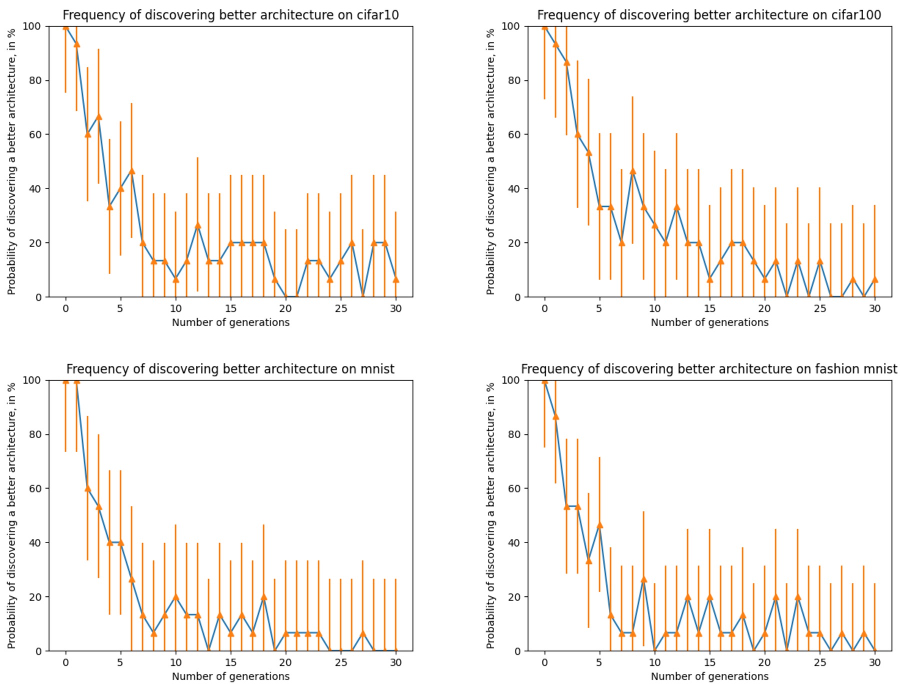


Figure 30: Probability of finding a new/better architecture, at each generation

It is clear that the probability of discovering a new architecture gets smaller as the generations goes on. The accuracy increase, and the way of improving the architecture gets rarer. The evolution on MNIST and

Fashion-MNIST seems to converge faster, since it is an easier dataset. The standard deviation seems to be high, so different executions could have led to different results.

Size of the architectures are shown in the figure 31, using the mean and standard deviation

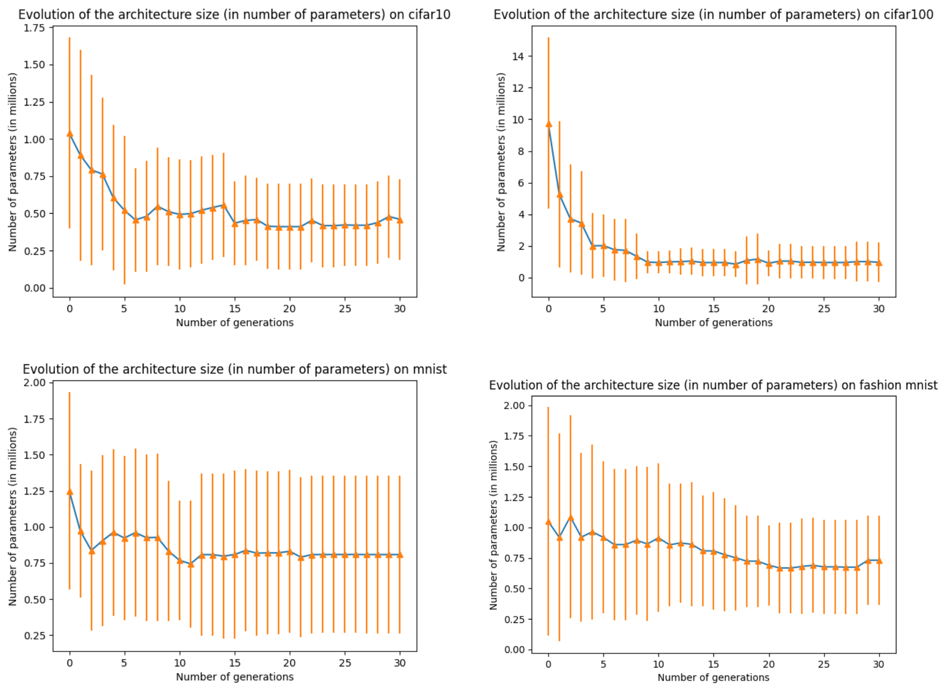


Figure 31: Size of the architecture, in number of parameters

Those results show that the architecture size tends to stabilize at some point. The evolution on cifar100 is bigger than the other datasets at the start because this dataset contains 100 classes, thus the number of tunable parameters is higher than the other datasets (mostly due to the fact that the fully connected layer has 100 outputs). Then, the number of parameters, on cifar100, seems to converge to much lower values. This is probably due to the number of epochs during the evolution phase (1), which could have caused a lower chance of seeing a "bigger" architecture (The final number of parameters will be reviewed later)

Validation score is shown under a boxplot form in the figure 32

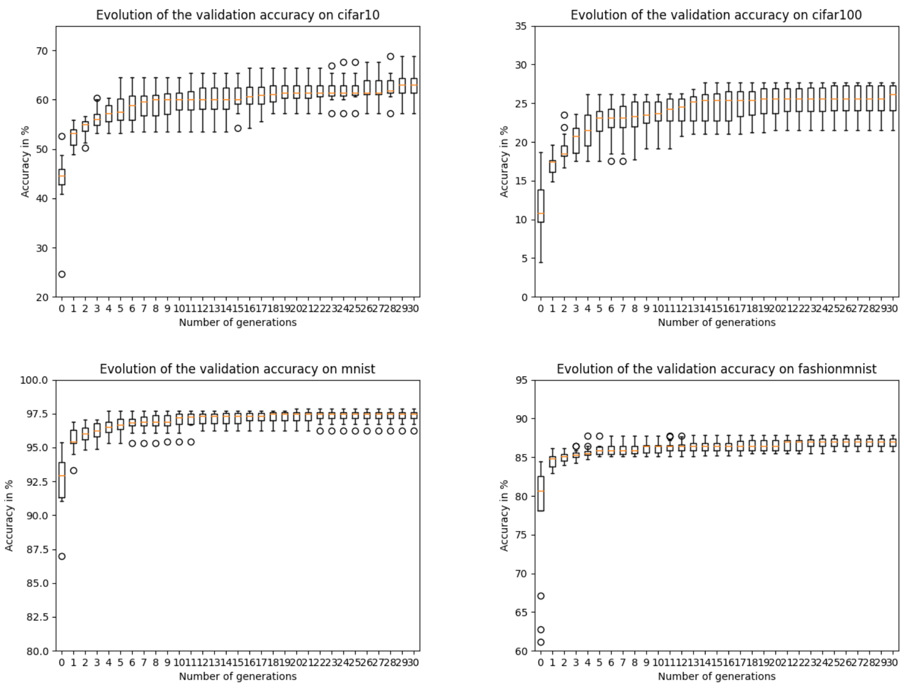


Figure 32: Evolution of the validation score trough generations

Those results show that the validation accuracy increase, and seems to converge at some point. The convergence on MNIST and Fashion-MNIST seems to be quicker, this is probably due to the fact that those datasets are "easier", and thus the room for improvement becomes quickly reduced.

Time for the evolution is shown under a boxplot form in the figure 33, and some statistics are shown in the table 34

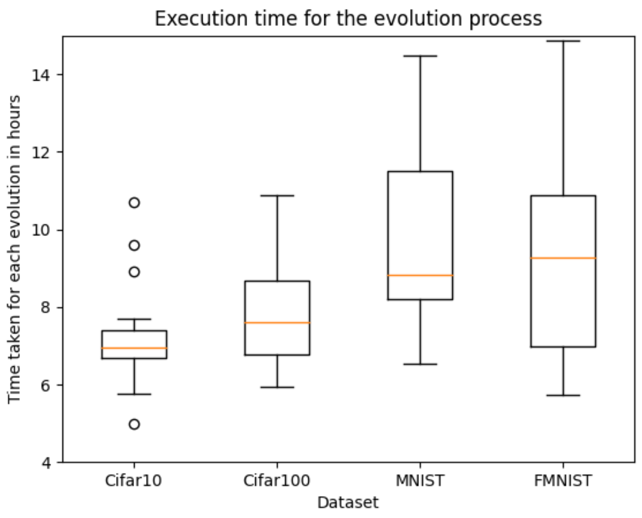


Figure 33: Time taken to proceed the evolution

Dataset	Median execution time	Sample standard deviation
cifar10	6h56min	1h26min
cifar100	7h35min	1h37min
MNIST	8h49min	2h32min
fashion-MNIST	9h15min	2h37min

Figure 34: Statistics about the evolution time depending on the dataset

We can observe that the time taken is roughly the same between each dataset. However, the execution time seems to be higher on MNIST and Fashion-MNIST, which is probably due to effects of randomness. On those datasets, the standard deviation is also higher.

4.2.2 Final architectures

In this sections, the 15 resulting architectures (for each dataset) are analyzed under many factors.

Test score is shown under a boxplot form in the figure 35, and the statistics are shown in a table 36

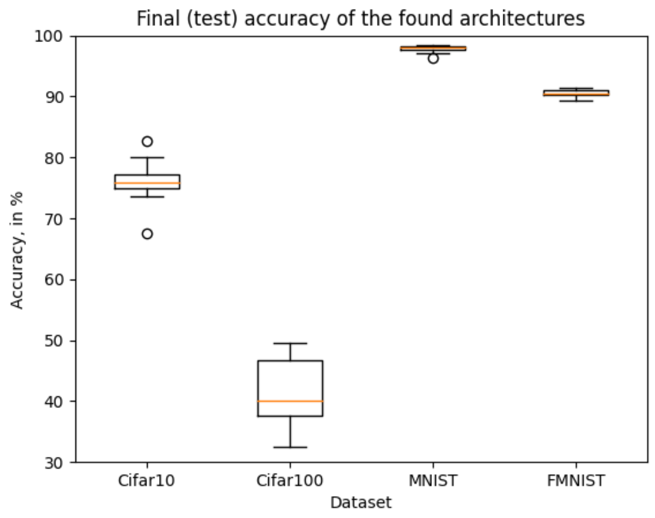


Figure 35: Accuracy of the final architectures (test set)

Dataset	Median accuracy	Sample standard deviation
cifar10	75.8%	3.32%
cifar100	40.08%	5.86%
MNIST	98.09%	0.58%
Fashion-MNIST	90.52%	0.64%

Figure 36: Statistics about the accuracy depending on the dataset

We can observe that the evolution on MNIST and Fashion-MNIST will have a low variance; this is probably due to the fact that these are "easier" dataset; in that case the worst architectures found have accuracy comparable to the best architecture (not the case for cifar10 and cifar100)

In fact, on MNIST and Fashion-MNIST, using only a fully connected could already give results that are far above a random guess (which would probably not be the case for cifar10 and cifar100)

Size of the architectures is shown in a boxplot in the figure 37 and some statistics can be found in the table 38

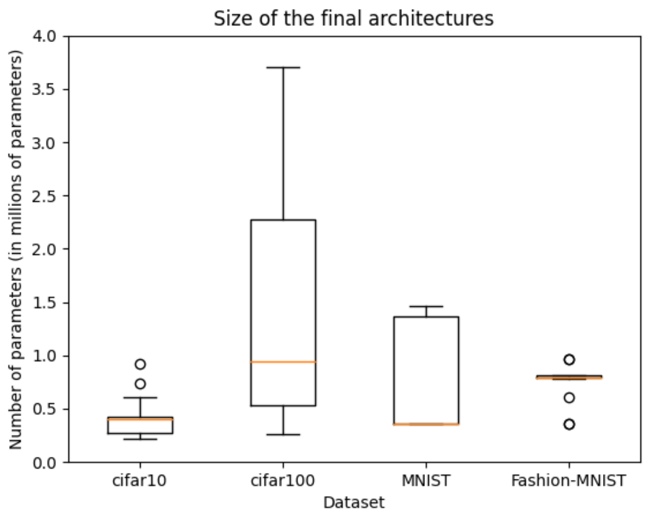


Figure 37: Size of the final architecture, in number of parameters

Dataset	Median size	Sample standard deviation
cifar10	397 322	155 392.684
cifar100	941 732	2 487 814.415
MNIST	361 674	465 150.432
Fashion-MNIST	794 570	125 687.292

Figure 38: Statistics about the architecture size depending on the dataset

As explained previously, the choice of using 1 epoch during the evolution phase may have impacted the results, so we can expect that, with a higher number of epochs, the number of parameters for the final architectures could have been higher, because bigger architectures will not be "avoided".

Concerning the standard deviation, it seems to be higher on cifar100 than on the other datasets, because cifar100 would require big architectures, and our implementation may have a tendency to prioritize small architectures. The evolution process may find some "good and small" architectures occasionally, but it does not happened frequently.

Benchmark on resulting structures ; In this last experiment, each of the found architectures (as well as one known architecture, shown in the figure 16) are tested with each of the 4 datasets. The median scores are shown in the table 54, standard deviation is shown in the table 40.

Structure from \ train and test dataset	cifar10	cifar100	MNIST	F-MNIST
Obtained for cifar10	75.71%	42.12%	98.05%	91.36%
Obtained for cifar100	70.76%	40.08%	97.0%	90.37%
Obtained for MNIST	74.47%	37.98%	98.09%	90.92%
Obtained for Fashion-MNIST	70.45%	35.02%	97.55%	90.52%
Source architecture	69.49%	36.37%	97.71%	89.44%

Figure 39: Median accuracy

Structure from \ train and test dataset	cifar10	cifar100	MNIST	F-MNIST
Obtained for cifar10	3.32%	3.74%	0.44%	0.84%
Obtained for cifar100	4.62%	5.86%	0.86%	0.99%
Obtained for MNIST	5.83%	5.87%	0.58%	1.05%
Obtained for Fashion-MNIST	4.51%	6.70%	0.83%	0.64%

Figure 40: standard deviation of the accuracy

We can observe that among all the datasets, the evolution on cifar10 and MNIST seems to produce "better" architectures, leading to higher accuracy, but those results were obtained from "only" 15 executions, so those results may not be a consequence of a huge difference in the evolutionary aspect on each dataset.

Also, on cifar10 and cifar100, the standard deviation seems to be quite high, because the images are more complex, so finding a structure able to recognize patterns in those images may be harder/rarer.

One interesting result is the fact that cifar100 is not the dataset that led to the "best" architectures, but specifically on the cifar100 dataset, the accuracy is the second best, which can lead us to the idea that the performance of an architecture is closely related to the dataset.

Time for the retrain is shown in a boxplot form in the figure 41, and some statistics are shown in the table 42

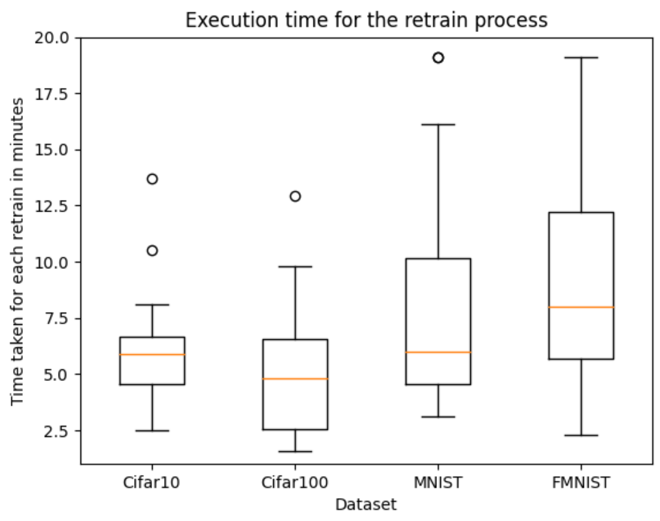


Figure 41: Time taken to proceed the retrain

Dataset	Median execution time	Sample standard deviation
cifar10	5min54sec	2min55sec
cifar100	4min49sec	3min19sec
MNIST	6min0sec	5min30sec
Fashion-MNIST	7min59sec	4min31sec

Figure 42: Statistics about the retrain time depending on the dataset

We can conclude the same thing about the evolution time (both are related, since a training time will imply a high evolution time); a high standard deviation on MNIST and Fashion-MNIST that is probably due to the effects of randomness. Also, the standard deviation seems high compared to the median time itself, which is a consequence of the fact that we did not use a multi-objective search approach, the time taken was not taken in account in the evolution process.

Correlation between number of parameters and accuracy of final architectures is shown in the figure 43, with the coefficient of correlation in the table in the figure 44

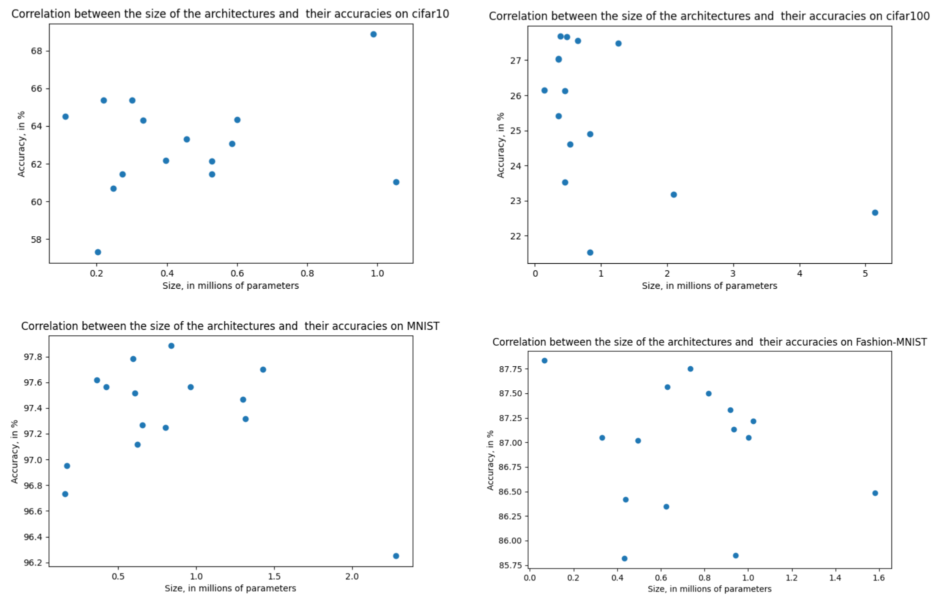


Figure 43: Correlation between the architecture size and accuracy

Dataset	Correlation coefficient between size and accuracy
cifar10	0.256
cifar100	-0.491
MNIST	-0.272
Fashion-MNIST	-0.156

Figure 44: Correlation between the architecture size and accuracy

We could have expected a coefficient of correlation between 0 and 1, meaning that the accuracy increase with the size of the architectures. But surprisingly, the coefficient does not take high magnitude, meaning no real correlation between the 2 variables. Also, the variance is still high, so those result may be biased. This "lack" of correlation may be explained by the fact that there are "good and small" architectures, our implementation just did not find many of them. In any case, our implementation did not produce "big" architectures

4.2.3 Best found architectures

In this section, for each dataset, the best architecture (out of 15 re-evolutions) is shown. Figure 45 shows the best obtained structure for the cifar10 dataset.

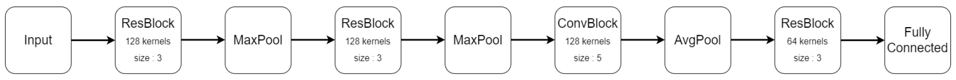

Input
ResBlock 128 kernels size : 3
MaxPool
ResBlock 128 kernels size : 3
MaxPool
ConvBlock 128 kernels size : 5
AvgPool
ResBlock 64 kernels size : 3
Fully Connected

Figure 45: Resulting structure for the cifar10 dataset

Figure 46 shows the best obtained structure for the cifar100 dataset.

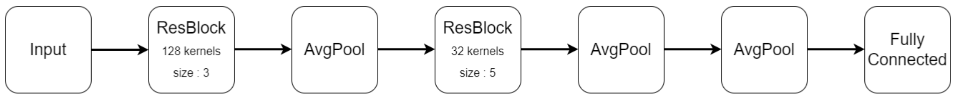


Figure 46: Resulting structure for the cifar100 dataset

Figure 47 shows the best obtained structure for the MNIST dataset.

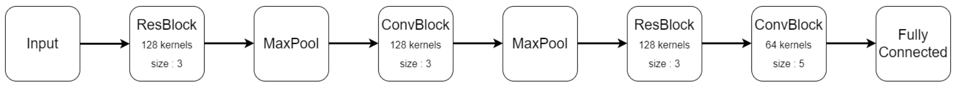

Input
ResBlock 128 kernels size : 3
MaxPool
ConvBlock 128 kernels size : 3
MaxPool
ResBlock 128 kernels size : 3
ConvBlock 64 kernels size : 5
Fully Connected

Figure 47: Resulting structure for the MNIST dataset

Figure 48 shows the best obtained structure for the Fashion-MNIST dataset.

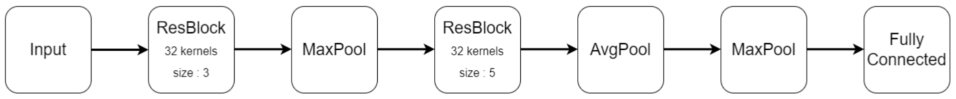


Figure 48: Resulting structure for the Fashion-MNIST dataset

Usually, the well known structures are composed by a succession of the following pattern (multiple times);

- 1 or more ConvBlock/Resblock
- one Pooling layer

Whose patterns can often be found in our architectures

4.3 Comparison with other well known structures

In the following, we compare our architectures to well known structures on multiple datasets. They are compared to the accuracy on this dataset, and the number of parameters.

The comparison on cifar10 is shown in the table 49.

Architecture name	Accuracy	Parameters
Median result for cifar10	75.8 %	397 322
Our best for cifar10	82.60%	988 042
VGG16	93.15% [50]	14 991 946 [50]
ResNet-20	91.52% [51]	269 722 [51]
ResNet-32	92.53% [51]	464 154 [51]
ResNet-44	93.16% [51]	658 586 [51]
ResNet-56	93.21% [51]	853 018 [51]
ResNet-110	93.90% [51]	1 727 965 [51]

Figure 49: comparison of structures on cifar10

The comparison on cifar100 is shown in the table 50.

Architecture name	Accuracy	Parameters
Median result for cifar100	40.08%	941 732
Our best for cifar100	49.52%	484 452
mobilenet	65.98% [52]	3.3M [52]
VGG11	68.64% [52]	28.5M[52]
VGG19	72.33% [52]	39M[52]
resnet18	75.61% [52]	11.2M [52]
resnet152	77.69% [52]	58.3M [52]
densenet121	77.01% [52]	7.0M [52]
densenet201	78.54% [52]	18M [52]
googlenet	78.03% [52]	6.2M [52]

Figure 50: comparison of structures on cifar100

The comparison on MNIST is shown in the table 51.

Architecture name	Accuracy	Parameters
Median result for MNIST	98.09%	361 674
Our best for MNIST	98.43%	838 154
LeNet	98.69% [53]	60 000 [54]

Figure 51: comparison of structures on MNIST

The comparison on Fashion-MNIST is shown in the table 52.

Architecture name	Accuracy	Parameters
Median result for Fashion-MNIST	90.52 %	794 570
Best for Fashion-MNIST	91.38%	66 090
LeNet	88.9% [55]	60 000 [54]

Figure 52: comparison of structures on Fashion-MNIST

We can see that our architecture were generally lower to other architectures in terms of size and accuracy. Again, this is probably due to the choice of using 1 epoch for the evolution phase.

4.4 Comparison with other approaches in NAS using GA

In the following, we compare our results to other works on NAS that uses GA (On cifar10)

Results of	Accuracy	Parameters
Our median for cifar10	75.8 %	397 322
Our max for cifar10	82.60%	988 042
Classical GA [32]	86%	0.62M
Suganuma original paper (ConvSet)	93.66%	1.75M
Suganuma original paper (ResSet)	93.95%	2.64M
Large-scale Evolution [33]	94.6%	5.4M
NSGA-Net + macro search space [27]	96.15%	3.3M

Figure 53: comparison with other approaches in NAS using GA

Again, the global tendency is that we obtained smaller architectures, and lower accuracy.

Compared to the original paper [6], we obtained lower results, probably because, as explained, we used a few number of epochs during the evolution phase.

4.5 Comparison with other approaches in NAS

In the following, we compare our architectures with other works on NAS (also on cifar10)

Results of	Accuracy	Parameters
Median for cifar10	75.8 %	397 322
Max for cifar10	82.60%	988 042
DensEMANN (Average of 5 executions) [36]	71.61%	11 800
Grid Search [32]	83%	0.84M
NAS with RL (using RNN) [30]	94.5%	4.2M

Figure 54: comparison with other approaches in NAS

5 Conclusion

In conclusion, using Cartesian genetic programming to obtain an architecture for a CNN seems to be a viable solution.

The variance of the performance may be high, which means that the randomness may have a great impact on the performance. We can expect that with much higher settings (offsprings per generation, etc.) the variance may be slightly lower. Additionally, the variance in performance seems to be highly correlated to the dataset.

Some discovered architectures were good on certain dataset but below the average on other datasets, confirming us that there is not one architecture that is intrinsically better than the others, it depends on the dataset.

5.1 Choice of parameters

Some choices were made, and can have impacted the results. The biggest limiting factor was probably the number of epochs during the evolution phase, which can have limited the size of the found architectures.

The number of generations did not seem to be a limiting factor, since there is a clear convergence on the plots that showed the evolution of the accuracy. Increasing this setting could have slightly improved the final accuracy, but not have a drastic impact.

Compared to the original implementation, we used more offspring per generation, but lower number of generations. The original setting may be more "efficient" in the sense that the "positive" mutations are directly taken in account, not waiting for a high number of offspring to be tested until the next generation. The advantage of our choice is that we probably avoided some "local maxima".

6 Future work

- Add one or more hidden layers to fully connected layer may improve the overall performance.
- Use a number of epochs during the evolution phase that is much higher, in order to confirm that our resulting architectures are "small" due to that parameter. We did not see a big difference between the experimented sized, but with a bigger difference in number of epochs, the results may be different.
- Use a performance predictor to accelerate the process [38][39][40], which means that in the same time, the evolution could "go further" (more epochs, more offsprings per generation, etc.).
- Some experiments in the fine-tuning section were done on one architecture, and this choice may have influenced the results, so those experiments could be reproduced using multiple architectures.
- Add experiments taking in accounts the parameters that were not experimented (mutation rate, number of offsprings, etc.)
- Use a different fitness indicator, taking the size of the architecture in account, for example.

References

[1] https://www.google.com/search?q=%5Dhttps%3A%2F%2F2900157524-files.gitbook.io%2F~%2Ffiles%2Fv0%2Fb%2Fgitbook-legacy-files%2Fo%2Fassets%252F-LZMLRvaju5sqPs7pYTX%252F-Ltv4uFdHxY7yt_MdVwX%252F-LrPm3TI57kjTUXtuCts%252Fwhatisgenetic3.png%3Fgeneration%3D1574023185419913%26alt%3Dmedia&rlz=1C1CHBF_frBE856BE856&oq=%5Dhttps%3A%2F%2F2900157524-files.gitbook.io%2F~%2Ffiles%2Fv0%2Fb%2Fgitbook-legacy-files%2Fo%2Fassets%252F-LZMLRvaju5sqPs7pYTX%252F-Ltv4uFdHxY7yt_MdVwX%252F-LrPm3TI57kjTUXtuCts%252Fwhatisgenetic3.png%3Fgeneration%3D1574023185419913%26alt%3Dmedia&aqs=chrome..69i57.401j0j7&sourceid=chrome&ie=UTF-8.

[2] John Biles et al. "GenJam: A genetic algorithm for generating jazz solos". In: *ICMC*. Vol. 94. Ann Arbor, MI. 1994, pp. 131–137.

[3] Timo Vanhatupa, Marko Hannikainen, and Timo D Hamalainen. "Genetic algorithm to optimize node placement and configuration for WLAN planning". In: *2007 4th International Symposium on Wireless Communication Systems*. IEEE. 2007, pp. 612–616.

[4] Janet Clegg, James Alfred Walker, and Julian Frances Miller. "A new crossover technique for cartesian genetic programming". In: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. 2007, pp. 1580–1587.

[5] Julian Francis Miller. "Cartesian genetic programming: its status and future". In: *Genetic Programming and Evolvable Machines* 21.1 (2020), pp. 129–168.

[6] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. "A genetic programming approach to designing convolutional neural network architectures". In: *Proceedings of the genetic and evolutionary computation conference*. 2017, pp. 497–504.

[7] John McCarthy et al. "A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955". In: *AI magazine* 27.4 (2006), pp. 12–12.

[8] Pei Wang. "What Do You Mean by" AI"?" In: *AGI*. Vol. 171. 2008, pp. 362–373.

[9] Shigao Huang et al. "Artificial intelligence in cancer diagnosis and prognosis: Opportunities and challenges". In: *Cancer letters* 471 (2020), pp. 61–71.

[10] Sahil Gupta, Divya Upadhyay, and Ashwani Kumar Dubey. "Self-Driving Car Using Artificial Intelligence". In: *Advances in Interdisciplinary Engineering*. Springer, 2019, pp. 521–533.

[11] Arthur L Samuel. "Machine learning". In: *The Technology Review* 62.1 (1959), pp. 42–45.

[12] Simon Rentzmann and Mario V Wuthrich. "Unsupervised Learning: What is a Sports Car?" In: *Available at SSRN 3439358* (2019).

[13] Pavlo M Radiuk. "Impact of training set batch size on the performance of convolutional neural networks for diverse datasets". In: (2017).

[14] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[15] Md Abu Bakr Siddique et al. "Study and observation of the variations of accuracies for handwritten digits recognition with various hidden layers and epochs using neural network algorithm". In: *2018 4th International Conference on Electrical Engineering and Information & Communication Technology (iCEEiCT)*. IEEE. 2018, pp. 118–123.

[16] Larry R Medsker and LC Jain. "Recurrent neural networks". In: *Design and Applications* 5 (2001), pp. 64–67.

[17] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. “Activation functions in neural networks”. In: *towards data science* 6.12 (2017), pp. 310–316.

[18] Shibani Santurkar et al. “How does batch normalization help optimization?” In: *Advances in neural information processing systems* 31 (2018).

[19] Robert Hecht-Nielsen. “Theory of the backpropagation neural network”. In: *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[20] Claus Nebauer. “Evaluation of convolutional neural networks for visual recognition”. In: *IEEE transactions on neural networks* 9.4 (1998), pp. 685–696.

[21] David H Hubel and Torsten N Wiesel. “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex”. In: *The Journal of physiology* 160.1 (1962), p. 106.

[22] Henry Jen-Hao Cheng. *Empirical Study on the Effect of Zero-Padding in Text Classification with CNN*. University of California, Los Angeles, 2020.

[23] `https://medium.com/ai%C2%B3-theory-practice-business/resblock-a-trick-to-impove-the-model-8ba11891c52a`.

[24] `http://ijetemr.org/2020/06/19/handwritten-character-recognition-using-deep-learning/`.

[25] `http://deanhan.com/2018/07/26/vgg16/`.

[26] `https://www.researchgate.net/figure/Original-ResNet-18-Architecture_fig1_336642248`.

[27] Zhichao Lu et al. “Nsga-net: neural architecture search using multi-objective genetic algorithm”. In: *Proceedings of the genetic and evolutionary computation conference*. 2019, pp. 419–427.

[28] Arash Vahdat et al. “Unas: Differentiable architecture search meets reinforcement learning”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 11266–11275.

[29] Bo Lyu et al. “Multiobjective reinforcement learning-based neural architecture search for efficient portrait parsing”. In: *IEEE Transactions on Cybernetics* (2021).

[30] Barret Zoph and Quoc V Le. “Neural architecture search with reinforcement learning”. In: *arXiv preprint arXiv:1611.01578* (2016).

[31] Yanan Sun et al. “Automatically designing CNN architectures using the genetic algorithm for image classification”. In: *IEEE transactions on cybernetics* 50.9 (2020), pp. 3840–3854.

[32] Petro Liashchynskyi and Pavlo Liashchynskyi. “Grid search, random search, genetic algorithm: a big comparison for NAS”. In: *arXiv preprint arXiv:1912.06059* (2019).

[33] Esteban Real et al. “Large-scale evolution of image classifiers”. In: *International Conference on Machine Learning*. PMLR. 2017, pp. 2902–2911.

[34] Santanu Santra, Jun-Wei Hsieh, and Chi-Fang Lin. “Gradient descent effects on differential neural architecture search: A survey”. In: *IEEE Access* 9 (2021), pp. 89602–89618.

[35] Mingwei Zhang et al. “NAS-HRIS: automatic design and architecture search of neural network for semantic segmentation in remote sensing images”. In: *Sensors* 20.18 (2020), p. 5292.

[36] Antonio Garcia-Diaz and Hugues Bersini. “DensEMANN: Building A DenseNet From Scratch, Layer by Layer and Kernel by Kernel”. In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2021, pp. 1–10.

[37] Utku Evci et al. "Gradmax: Growing neural networks using gradient information". In: *arXiv preprint arXiv:2201.05125* (2022).

[38] Lukas Hahn et al. "Fast and Reliable Architecture Selection for Convolutional Neural Networks". In: *arXiv preprint arXiv:1905.01924* (2019).

[39] Boyang Deng, Junjie Yan, and Dahua Lin. "Peephole: Predicting network performance before training". In: *arXiv preprint arXiv:1712.03351* (2017).

[40] Yanan Sun et al. "Surrogate-assisted evolutionary deep learning using an end-to-end random forest-based performance predictor". In: *IEEE Transactions on Evolutionary Computation* 24.2 (2019), pp. 350–364.

[41] T Chen, I Goodfellow, J Shlens, et al. "Accelerating learning via knowledge transfer". In: (2016).

[42] Song Han et al. "Dsd: Dense-sparse-dense training for deep neural networks". In: *arXiv preprint arXiv:1607.04381* (2016).

[43] `https://www.researchgate.net/figure/Cartesian-genetic-programming-scheme_fig1_265642724`.

[44] Julian F Miller et al. "An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach". In: *Proceedings of the genetic and evolutionary computation conference*. Vol. 2. 1999, pp. 1135–1142.

[45] `https://github.com/sg-nm/cgp-cnn`.

[46] `https://www.cs.toronto.edu/~kriz/cifar.html`.

[47] `https://datarepository.wolframcloud.com/resources/CIFAR-100`.

[48] `https://www.researchgate.net/figure/Example-images-from-the-MNIST-dataset_fig1_306056875`.

[49] `https://www.researchgate.net/figure/Sample-images-from-Fashion-MNIST-dataset_fig2_342801790`.

[50] URL: https://github.com/SeHwanJoo/cifar10-vgg16.

[51] https://github.com/SeHwanJoo/cifar10-ResNet-tensorflow..

[52] https://github.com/weiaicunzai/pytorch-cifar100.

[53] https://medium.com/mlearning-ai/lenet-and-mnist-handwritten-digit-classification-354f5646c590.

[54] https://www.analyticsvidhya.com/blog/2021/03/the-architecture-of-lenet-5/#:~:text=It%20has%203%20convolution%20layers,of%20trainable%20parameters%20is%2060000..

[55] http://ijetemr.org/2020/06/19/handwritten-character-recognition-using-deep-learning/.