IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 19 January 2023, accepted 16 February 2023, date of publication 22 February 2023, date of current version 27 February 2023.
Digital Object Identifier 10.1109/ACCESS.2023.3247773

RESEARCH ARTICLE

Generating A-Star Algorithm Admissible Heuristics Using a Dynamic Dataloader on Neural Networks, Enhanced With Genetic Algorithms, on a Distributed Architecture

OUARDI AMINE AND MESTARI MOHAMMED
Signals, Distributed Systems and Artificial Intelligence Laboratory, École Normale Supérieure de l'Enseignement Technique (ENSET), Mohammedia 28830, Morocco
Department of Mathematics and Computer Science, Hassan II University of Casablanca, Casablanca 20000, Morocco
Corresponding author: Ouardi Amine (amine.ouardi22@gmail.com)

ABSTRACT Heuristic search algorithms are informed search strategies that use heuristics to estimate the minimal cost of the path from the current state to the goal. Using this additional knowledge, this type of algorithms can distinguish non-goal states, and then directs its search towards those that look more promising. This makes these algorithms faster than uninformed search algorithms. The A* algorithm is a well-known example of heuristic-based algorithms that is guaranteed to find the least-cost path to a goal state if the heuristic used is admissible, which means that it never overestimates the real cost from the current state to the goal. In this article, we are going to present a neural network architecture that makes it possible to predict admissible heuristics for a given graph and destination node, then we are going to refine our models using genetic algorithms in order to obtain better results.

INDEX TERMS Neural networks, genetic algorithms, A star algorithm, graphs, heuristics, path finding, PyTorch, Azure databricks, distributed computing.

I. INTRODUCTION

In many video games, more specifically the Massively Multiplayer Online Role-Playing Games (MMORPGs), in order to simulate a real human experience, the characters will bypass boxes of trees, avoid crushing into mountains or lakes, as to get to the selected destination. In order to achieve this kind of human behavior, one of the commonly used pathfinding algorithms is A-star (or A∗) search algorithm [1]: it is an informed search algorithm as it combines path cost information and heuristics to get the shortest path between graph nodes.

A* returns the path that minimizes the function:

$$f(n) = g(n) + h(n) \tag{1}$$

where n is the next node to be visited, $g(n)$ is the real cost from the start node to the current node n, and $h(n)$ is a heuristic that is used to estimate the remaining cost of the path from n to the goal. A* can be optimal and complete, optimal if the heuristic used is consistent and complete if the graph is finite.

A heuristic is consistent if for every node N and for each of its successors, the estimated cost of reaching the set goal from the node N is always lower than or equal to the cost between N and any of its successors plus the estimated cost of reaching the same set goal from that successor:

$$h(N) \leq c(N, S) + h(S) \tag{2}$$

where h is the heuristic that is considered consistent, N represents a graph node, S is one of its successors, and $c(N,S)$ is the cost of the path between N and S.

However, A* algorithm may be implemented in such a way that the returned path is optimal even though the heuristic is not consistent, provided that it is admissible: once a node is reached by a path, it is consequently removed from the set of discovered nodes that might need to be expanded, and if this same node is latterly reached by a cheaper path, it will be added again to the discovered nodes set.

The associate editor coordinating the review of this manuscript and approving it for publication was Diego Oliva.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

A heuristic is considered admissible if, for every graph node, it never overestimates the lowest possible cost to get to the goal node:

$$h(n) \leq d(n) \quad (3)$$

where $h(n)$ is the cost returned by the heuristic h to reach the goal from the node n, and $d(n)$ is the optimal cost to reach the goal from n. As a result, if the used heuristic is not consistent, but admissible, it is possible for a node to be expanded many times by the A* algorithm.

Throughout this paper, a PyTorch [2] based neural network is going to be developed with a dynamic Dataloader approach that enables to provide an admissible heuristic for any given graph, by setting up the goal node. We will set a hyperparameters dictionary and proceed to train different models with different hyperparameter values, which will allow us to obtain different models. These models will be considered as the initial population for the genetic algorithm, tackled in this article, as a mean to improve the models' scores. In order to accelerate this process, we will use a Microsoft Azure Databricks [3] cluster to launch this process on a distributed computing system. This new approach of a distributed architecture is considered among the new features brought up in this paper, compared to our previous work entitled Predicting A* search algorithm heuristics using neural networks'' [4] in which the training phase was performed on a local machine.

Another feature that can be considered as a distinct characteristic will be noticed at the level of the neural network implementation, in which the predicted heuristic value for the destination node will henceforth be fixed on zero (on the forward method), which differs from our earlier publication where this value was approaching zero drastically without ever reaching it. On the other hand, the dynamic Dataloader technique will be preserved and reused in the training phase.

Finally, the genetic algorithm is a totally new perspective presented throughout this paper as it was not implemented in the previous one. It aims to enhance the obtained models' performance by considering each one of them as an individual that will be represented as a Python dataclass object.

PyTorch is an open-source machine learning framework based on the Python programming language and the Torch library. This framework combines both usability and speed: it's well known for its tensor computation, which is a high-level array-based programming model allowing a faster training time. That is why it is considered as one of the most commonly used and recommended frameworks for deep learning research. The reasons behind this choice will be highlighted throughout this paper.

PyTorch is well supported on major cloud platforms, including Microsoft Azure Databricks which is a fast and optimized cloud-scale platform for machine learning. We will be using Azure Databricks clusters to run our jobs in a distributed way, on preconfigured virtual machines with preinstalled deep learning environments, to economize time and to debug our code quickly.

II. USING DIJKSTRA ALGORITHM TO GENERATE TRAINING DATA

In order to train our model, a training dataset should be prepared, describing graphs alongside their appropriate heuristics. As long as our first approach consists on working with undirected graphs with positive weights (representing the distances between each graph nodes), Dijkstra algorithm seems to be a suitable algorithm that can be used to initialize our dataset: we consider Dijkstra's start node as the A-star's goal node.

Dijkstra algorithm [5] is a pathfinding algorithm mostly used on weighted graphs to create a tree of shortest paths from the designated starting node to all other graph nodes. By reversing this tree of shortest distances, we will generate a heuristic that describes the shortest path from each node to the Dijkstra starting node, that we will then consider as a goal node for A-star algorithm.

In this article, we had considered working with graphs with an order of 10. Thus, we started by generating random distance matrices of size 10*10, which represent the weights of the graph, then we randomly selected a start node index. We apply Dijkstra algorithm on each distance matrix and its starting node to finally create a vector of the shortest distances from the start node to the other graph nodes.

We then concatenated the flatten distance matrix and its start node in a PyTorch tensor, to finally create a tuple with the vector of minimal distances also saved as a PyTorch tensor and which represents our initialized heuristic:

$$\begin{aligned} tuple = (&concat \\ &\times (start\ node,\ flatten\ distance\ matrix), \\ &heuristic) \end{aligned} \quad (4)$$

This process will be repeated until finally getting a list of tuples that will be considered as our training dataset. Once created, this final list had been saved in an Azure Data Lake Storage, that will be mounted on the training cluster. This last operation will allow to read the training data and save our models and some other files that will be listed later. Data Lake Storage is a Microsoft Azure storage service optimized for storing massive amounts of unstructured data for distributed access, which corresponds to our needs.

III. AN ADAPTED ACTIVATION FUNCTION AND A DYNAMICALLY REGENERATED DATALOADER

A. THE MODEL'S ARCHITECTURE AND THE ADAPTED ACTIVATION FUNCTION

As mentioned above, only undirected graphs will be processed, resulting in symmetric distance matrices. It is also important to mention that the generated graphs on the training dataset have no loops or parallel edges, but may be incomplete: two or more nodes may not be connected.

Therefore, in the purpose of avoiding redundant data, we can preprocess the generated training data by eliminating the strictly upper or lower triangular matrix as $a_{ij} = a_{ji}$, for every i, j in [0, 9]:

~~a00~~	~~a01~~	~~a02~~	~~a03~~	~~a04~~	~~a05~~	~~a06~~	~~a07~~	~~a08~~	~~a09~~
a10	~~a11~~	~~a12~~	~~a13~~	~~a14~~	~~a15~~	~~a16~~	~~a17~~	~~a18~~	~~a19~~
a20	a21	~~a22~~	~~a23~~	~~a24~~	~~a25~~	~~a26~~	~~a27~~	~~a28~~	~~a29~~
a30	a31	a32	~~a33~~	~~a34~~	~~a35~~	~~a36~~	~~a37~~	~~a38~~	~~a39~~
a40	a41	a42	a43	~~a44~~	~~a45~~	~~a46~~	~~a47~~	~~a48~~	~~a49~~
a50	a51	a52	a53	a54	~~a55~~	~~a56~~	~~a57~~	~~a58~~	~~a59~~
a60	a61	a62	a63	a64	a65	~~a66~~	~~a67~~	~~a68~~	~~a69~~
a70	a71	a72	a73	a74	a75	a76	~~a77~~	~~a78~~	~~a79~~
a80	a81	a82	a83	a84	a85	a86	a87	~~a88~~	~~a89~~
a90	a91	a92	a93	a94	a95	a96	a97	a98	~~a99~~

FIGURE 1. **Retrieving data from the distance matrix without considering the upper triangular matrix.**

Actually, eliminating the redundant data could be done on the training dataset creation step, but it was agreed that it would be easier for the user to only provide a graph distance matrix and its destination node, and then apply this data preprocessing operation from the model's side.

The 45 nodes retrieved from the distance matrix are representing the same information: their values are the distances between the graph nodes, whereas the destination node index, considered as the 46$^{\text{th}}$ input, does not store the same feature. Following this distinction, it had been decided not to mix the 45 distance nodes with the destination node on the input layer, but rather add the latter on the first hidden layer:

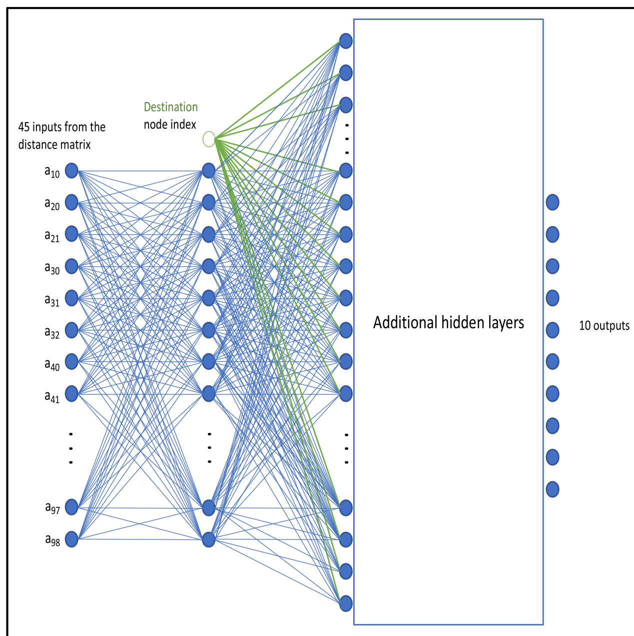


FIGURE 2. **The neural network architecture: the destination node index added on the first hidden layer.**

The purpose of this approach is to share distance data between the totality of the graph nodes first: on the first hidden layer, each neuron will be connected to all the inputs, which allows aggregating this information from all the graph nodes. Then the destination node index is added on the first hidden layer as a new input, and this input is going to be connected to all the neurons on the following layer.

Achieving this approach, 45 elements are obtained on the input layer and 46 elements on the first hidden layer. Then, after carrying out some tests on many architectures, we decided to add two more hidden layers, with 100 neurons each, another one with 50 neurons, before the final layer which contains 10 outputs representing the heuristic that we are looking for (the order of all the used graphs in the training dataset is 10).

We have opted for a fully-connected architecture, with the use of the *Parametric ReLU activation function* [6] for all the hidden layers, except for the last one where the *Soft Exponential activation function* [7] had been applied, combined with the absolute value function as all the predicted heuristics must be positive.

For starters, the *ReLU activation function* had been used on the first four hidden layers, however, it had been found that, after some epochs, most of the neurons remain inactive independently of the training sample passed as input, which was leading to a no gradient flow in a large number of neurons: this is called the *dying ReLU problem*. Thus, to avoid affecting the model's performance, the *ReLU activation function* was replaced by the *Parametric ReLU activation function*.

Using *Parametric ReLU (PReLU)* allows to avoid zeroing out the negative neurons' inputs: instead, it multiplies them by a slope parameter that is optimized during the training, which extends the range of *ReLU* and compensates *the dying ReLU problem*.

Optimizing the slop parameter through the backpropagation process causes a negligible increase in the cost of the training: each layer tries to optimize a single slope parameter. Hence, using PReLU on the hidden layers results in adding four trainable parameters, which is negligible compared to the number of weights and biases to be learned.

As mentioned before, we have used the composition of *the soft exponential activation function* and the absolute value function on the output layer. This function is a great example of activation functions with trainable parameters that are meant to be optimized while training the model. Adding the absolute value function aims to get positive outputs, as the heuristics that the model is trying to predict must be positive, thus we called this function *the absolute soft exponential activation function*. That being the case, this function remains a simple and differentiable function that can consistently interpolate between logarithmic, linear and exponential functions:

$$f(\alpha, x) = \begin{cases} \left| \dfrac{log_e(1 - \alpha(x + \alpha))}{\alpha} \right|, & \alpha < 0 \\ |x|, & \alpha = 0 \\ \left| \dfrac{e^{\alpha x} - 1}{\alpha} + \alpha \right|, & \alpha > 0 \end{cases} \tag{5}$$

Compared to the soft exponential activation function, the composition of the latter with the absolute value function saves the same characteristics, with some slight differences:

- Characteristic 1:

$$f(-1, x) = |log_e(x)|' \tag{6}$$

- Characteristic 2:

$$f(0, x) = |x| \tag{7}$$

- Characteristic 3:

$$f(1, x) = e^x \tag{8}$$

- Characteristic 4: it is continuously differentiable with respect to x:

$$\frac{\partial f(\alpha, x)}{\partial x} = \begin{cases} \frac{\alpha.log_e(1-\alpha(x+\alpha))}{\left|\alpha.log_e(1-\alpha(x+\alpha))\right|.(\alpha(x+\alpha)-1)}, & \alpha < 0 \\ \frac{x}{|x|}, & \alpha = 0 \\ \frac{e^{\alpha x}.(e^{\alpha x}-1+\alpha^2)}{\alpha.\left|\frac{e^{\alpha x}-1}{\alpha}+\alpha\right|}, & \alpha > 0 \end{cases} \tag{9}$$

because

$$\lim_{\alpha\to 0^+} \frac{\partial f(\alpha, x)}{\partial x} = \lim_{\alpha\to 0^-} \frac{\partial f(\alpha, x)}{\partial x} = sgn(x) \tag{10}$$

where

$$sgn(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases} \tag{11}$$

- Characteristic 5: it is continuously differentiable with respect to α:

$$\frac{\partial f(\alpha, x)}{\partial \alpha} = \begin{cases} \frac{(2\alpha+x).log_e(1-\alpha(x+\alpha))}{\left|\alpha.log_e(1-\alpha(x+\alpha))\right|.(\alpha(x+\alpha)-1)} - \frac{\alpha.\left|log_e(1-\alpha(x+\alpha))\right|}{|\alpha|^3}, & \alpha < 0 \\ 0, & \alpha = 0 \\ \frac{(e^{\alpha x}-1+\alpha^2).(e^{\alpha x}(\alpha x-1)+\alpha^2+1)}{\alpha^3.\left|\frac{e^{\alpha x}-1}{\alpha}+\alpha\right|}, & \alpha > 0 \end{cases} \tag{12}$$

because

$$\lim_{\alpha\to 0^+} \frac{\partial f(\alpha, x)}{\partial \alpha} = \lim_{\alpha\to 0^-} \frac{\partial f(\alpha, x)}{\partial \alpha} = \frac{x.(x^2+2)}{2.|x|} \tag{13}$$

- Characteristic 6: the continuum of operations between addition and multiplication of soft exponential activation function is also saved in our absolute version of this function, with some changes:

$$h(\beta, p, q) = f(\beta, f(-\beta, p) + f(-\beta, q)) \tag{14}$$

If $\beta = 0$, the function h adds |p| and |q|, rather than p and q in the case of the soft exponential function. If $\beta = 1$, it multiplies $p^{sgn(p)}$ and $q^{sgn(q)}$ for the absolute soft exponential function, rather than p and q for the soft exponential function.

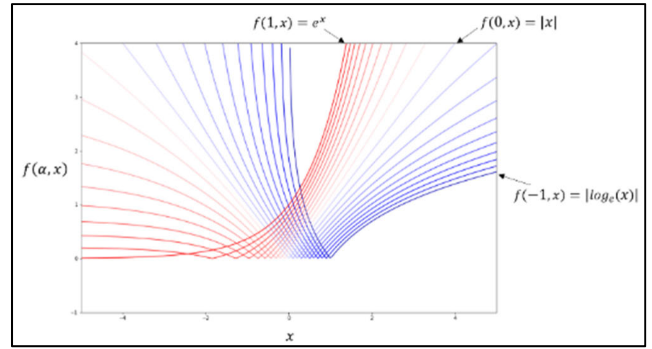


FIGURE 3. The plot of f(α, x) for α = {−1, −0.9, −0.8, . . . , 0.8, 0.9, 1} from blue to red.

Thus, as the absolute soft exponential function is differentiable, the model can be trained using the gradient descent. We first initialize the alpha parameter (α) by zero value, and then it is continuously updated comparably to the weights, by stepping toward the gradient direction leading to the reduction of the output error (the loss value). In the PyTorch framework, the optimization of this parameter is done by setting the attribute 'requiresGrad' on true:

```
alpha.requiresGrad = True
```

B. THE CROSS-VALIDATION

One of the most well-known data resampling methods is the cross-validation [8] which aims to enhance the generalization ability of the predictive models.

The way this method had been exploited on our model was by applying it on a batch level: the neural network is fed the whole training dataset on each epoch, but on different batches with an already fixed size. On each batch, the examples are split into two sets: training and validation sets (their sizes depend on another fixed parameter). The validation dataset is constructed by randomly selecting samples indices from the batch training dataset, which makes it possible for a data sample to be on the training dataset on a given epoch, and on the validation dataset on another epoch. Therefore, the main purpose of using the cross-validation method is to calculate the training loss and the validation loss on each batch, which will allow the use of the early stopping technique to avoid the overfitting: this will be thoroughly discussed afterwards.

C. GUIDING THE TRAINING TOWARD THE ADMISSIBILITY

As explained on the early sections of this article, a heuristic may be qualified as admissible if this one does not overestimate the cost of reaching the goal from any node of the graph.

Our input training dataset is in the form of pairs of destination nodes and distance matrices that were obtained using Dijkstra algorithm, consequently, each distance is guaranteed to be the least cost (the shortest path) to the corresponding goal node. So, in order to lead the training phase towards

the prediction of heuristics respecting the admissibility condition, we have created a new method to exploit the dataloader, which is a PyTorch class that is used to feed the training data to the model, serving up batches of fixed size. We have implemented a dataloader that dynamically self-increments, reloading itself with every single training example that does not provide heuristic values lower than or equal to the least possible cost that is contained within the distance matrix.

Algorithm 1 Recharging the Dataloader Dynamically

1: **for** epoch **in** range(0, nb_epochs):
2: Load trainset previously created via the Dijkstra algorithm
3: Create a dataset Python object as a GraphDataset
4: Generate the dataloader from the GraphDataset object
5: i_iter ← 0
6: **while** i_iter < len(dataloader)**:**
7: i_iter + =1
8: Iterate over the dataloader to the (i_iter)th batch
9: Cross-validation: split the data into training set and validation set
10: Train the model (calculate the train-loss)
11: Validate the model (calculate the valid-loss)
12: Concatenate the training and validation sets
13: Compare the predicted heuristics to the real costs
14: Create liste_bad, a list with all the batch examples which generated heuristics greater than the actual real costs
15: Extend the trainset list with list_bad values
16: Generate a new GraphDataset object from the extended trainset list
17: Create a dataloader from the GraphDataset previously generated: replace the old one by this one
18: **end while**
19: **end for**

Following this approach, the training dataset length at the end of a certain epoch may be way bigger than its length at the start of the epoch: we can start an epoch with a training dataset containing 100 samples (pairs of destination nodes and distance matrices), with a batch size of 5 per example, which gives us a dataloader length of 20, and end up with a dataloader's length of 153, generated by a training dataset of 765 samples, obtained after 133 iteration of the "while" loop at the end of the epoch, as that was the case on the logs of one of our experiments.

The GraphDataset object, mentioned in the algorithm above, is an overriding of the Dataset class from the PyTorch framework, that was customized to handle our training dataset in the form of pairs of distance matrices and destination nodes, with their respective heuristics (obtained via the Dijkstra algorithm). Afterwards, it is used to create the PyTorch Dataloader class, to recreate the dataloader instance.

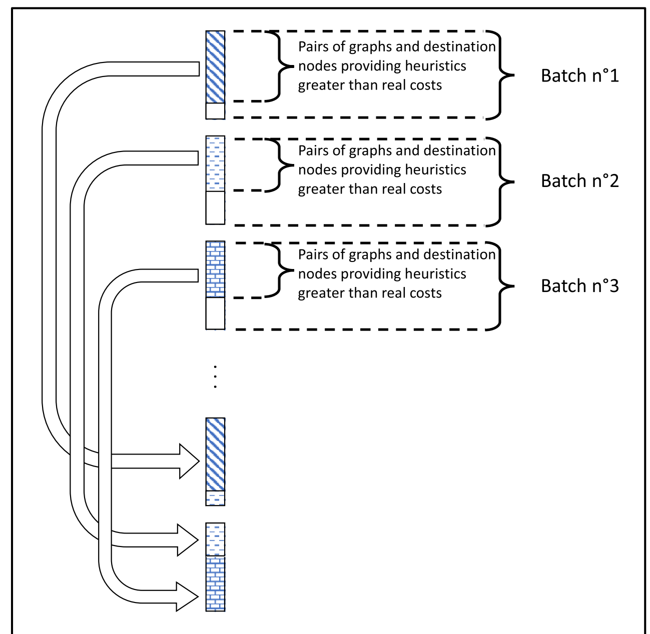


FIGURE 4. Dynamically incrementing the dataloader in one epoch.

With this approach, the last batch of a certain epoch, containing examples of graphs and destination nodes that have failed to provide admissible heuristics earlier in the training phase, may be with a length lower than the size of the previous batches. In order to avoid dropping this last non-full batch, the PyTorch Dataloader class provides the 'drop_last' argument that we have set on False.

It is also important to note that we have observed, after many experiments, that the value of the predicted heuristics for the destination nodes approaches drastically to zero, without ever reaching it. Thus, we decided to implement the forward method of the network PyTotch class, describing our architecture, to always keep this value at zero: it is a postprocessing step added once the training sample is fed to the neural network (through all the layers) and the heuristic is predicted.

D. CALLING THE EARLY STOPPING TWICE

On the previous algorithm, describing the dynamically regenerated dataloader approach, we nested a "while" loop within a "for" loop: the first loop purpose is to verify the admissibility condition, expanding the dataloader size with samples that have provided heuristics greater than the real cost, while the second loop aims to iterate over the chosen number of epochs. To avoid the overfitting, we decided to interrupt each loop execution using the early stopping technique.

In the training process, the early stopping technique [9] is a reputable and inexpensive regularization method allowing to stop the training at a suitable moment, to avoid the overfitting dilemma.

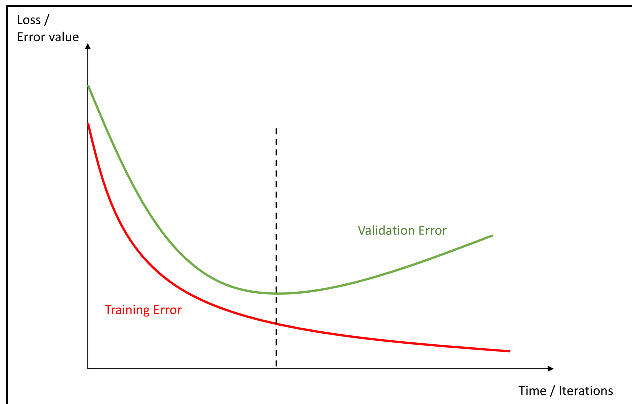


FIGURE 5. Evolution of the validation and the training error over time.

After splitting the batch samples into training set and validation set, using the cross-validation, we train the model over the training set, then proceed to its evaluation on the validation set, to get the training error and the validation error. The training is then stopped once the validation set starts to increase, meaning that the performance of the model on the validation set starts to degrade. This allows to avoid the overfitting as the model at this stage has low variance, and will generalize well: continuing the training will increase its variance and lead to overfitting.

In the ''while'' loop, each time a decline in value is detected on the validation error, a checkpoint is created on the Data Lake storage, saving the model's *state_dict* object, which is a Python dictionary that maps each layer to its learnable parameters' tensor (weights, biases, the slope parameter of PReLU for the hidden layers and the α parameter of the absolute soft exponential activation function for the output layer), and the optimizer's *state_dict* as well, which is another Python dictionary that contains information about the optimizer's state. Once we detect that the validation error starts increasing, the early stopping function is triggered: it will be pending for a certain period of time before interrupting the training, to make sure that the validation error does not decrease in the following iterations. This delay is called '*patience*', it is actually a predefined parameter of the early stopping function.

This being the case, the overfitting problem is avoided on each epoch, and the early stopping method is also used as the breaking criteria that will cease incrementing our dataloader with samples that failed to provide admissible heuristics. This process is applied on an epoch level; however, the overfitting still needs to be avoided through the remaining epochs: that's why another call of the early stopping method is considered necessary.

Once the early stopping is executed on an epoch level, the ''while'' loop is then exited passing the last validation error to another instance of the early stopping function, that we have named 'EPOCH_early_stopping', which compares the last validation error obtained on each epoch to the one from the previous epoch, which corresponds to the previous iteration of the ''for'' loop (as shown on the figure bellow, it compares VE_n to VE_{n-1}), and then stops this loop's execution once the validation error value starts increasing, after another delay as well (the *patience* parameter of this instance).

Thus, the ''for'' loop, iterating over the predefined number of epochs, might as well be stopped to avoid the model's overfitting.

IV. VANISHING AND EXPLODING GRADIENTS

As explained earlier, the cross-validation method and the early stopping technique are good solutions to prevent the overfitting problem, which allows the model to generalize well on new and unseen data.

However, the overfitting obstacle is not the only problematic that might be faced. During the backpropagation, the model goes through continuous matrices multiplications from the final to the initial layer, to calculate the derivatives used to update the weights. At a certain point of the training process, these derivatives may get large, which leads the gradient values to increase exponentially as the backpropagation is flowing down the model, until they eventually explode: this is known as the *exploding gradient problem*. By contrast, if the derivatives get small, the gradient values will decrease exponentially with the backpropagation flow, until eventually vanishing: this is called the *vanishing gradient problem*.

If either occurs, the model's performance degrades as it becomes unable to learn from the training data: large updates in the model weights produce a very unstable network. In some of our experiments, we have even encountered some overflows due to rapidly increasing gradients, which has led to some NaN weights values, meaning that they can no longer be updated. Alternatively, the trainable parameters of the first layer will not be functionally updated because of accumulated small gradients, inducing the model to be unable to learn the core features of the input data.

Many approaches exist to prevent these exploding and vanishing gradients problems. In the next section, we will be presenting two methods that we had adopted to deal with such issues, namely the layers' weights initialization and the gradient clipping.

A. INITIALIZING THE WEIGHTS

Weight initialization can have a great impact on the model's performance, but requires the right method. If we initialize all the weights to the same value, such as zero or one for example, the neurons will get the same signal, which makes them learn the same identical features during the training. Therefore, breaking the symmetry is decisive, but it is not the only criteria: initializing the weights with too small values produces to a slowness of the learning process and may cause the vanishing gradients problem, whereas initializing them with too large values will lead to divergence and ends in exploding the gradients.

In order to prevent this from happening, the most accurate initializing methods are the ones where the mean of the

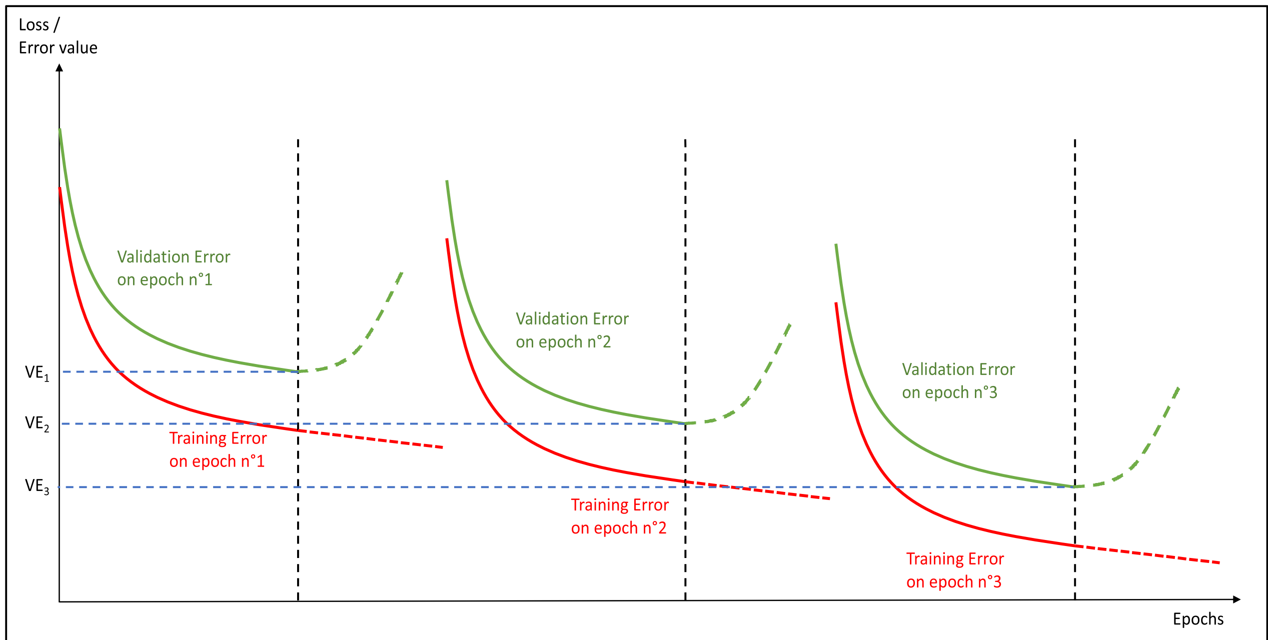


FIGURE 6. **Early stopping applied over the epochs.**

activations is zero, with a variance that remains the same across every layer. It guarantees that the gradient signal will not be multiplied by too small or too large values in any layer during its backpropagation.

Thus, we implemented two weights initialization approaches:

1) UNIFORM INITIALIZATION

As the name suggests, this method relies on the uniform distribution to randomly initialize the weights. The probability to select every weight value in the uniform distribution U is:

$$\omega_{ij} \sim U(\frac{-1}{\sqrt{n}^{[l-1]}}, \frac{1}{\sqrt{n}^{[l-1]}}) \tag{15}$$

where $n^{[l-1]}$ is the number of neurons in the layer $l-1$, biases are initialized with zeros.

2) XAVIER INITIALIZATION

This approach is a Gaussian initialization heuristic that keeps the same variance on the input and the output of the layer.

Thus, Xavier initialization method [10] ensures that the variance remains unchanged throughout the network.

It sets layer's weights to values chosen from a random uniform distribution, bounded between

$$\omega_{ij} \sim U(-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}) \tag{16}$$

where n_i represents the number of incoming network connections to the layer, also known as 'fan-in', on the other hand, n_{i+1} represents the number of outgoing network connections from that layer, also known as the 'fan-out'.

It had been observed, throughout our experiments, that Xavier initialization method allowed the network to maintain near identical variances of its weights across its layers:

It is important to note that, in the table above, the two models used have the same hyperparameters values, only the initialization method is different. Through the tests that have been carried out on these models, and that will be discussed later, we have observed that maintaining a nearly identical variance on every layer, through the use of Xavier initialization approach, allowed us to get better results.

Tensorboard is a measurement and visualization tool used during the training workflow to track the specified metrics and then project them to a lower dimensional space. From PyTorch version 1.1.0, Tensorboard has been added as a utility package that enables its exploitation easily. The Tensorboard histograms and the distribution graphs, presented in the table above, display how the distribution of the gradients has changed over time: their distribution over each epoch.

B. GRADIENT CLIPPING

It is possible to avoid gradients exploding problems by modifying the derivatives of the error before propagating it backwards into the model's layers: before using it to update the model's weights. This is achieved by recalling the gradients given a chosen vector norm, and then clipping gradient values that exceed a selected range. This approach is called 'gradient clipping' [11].

There are two variants of the gradient clipping method:

1) GRADIENT CLIPPING BY VALUE

Using this approach, we had started by defining a minimum and a maximum clip value. Once the gradient exceeds the

TABLE 1. Distributions and histograms of the backpropagated gradients on 500 epochs, generated via Tensorboard.

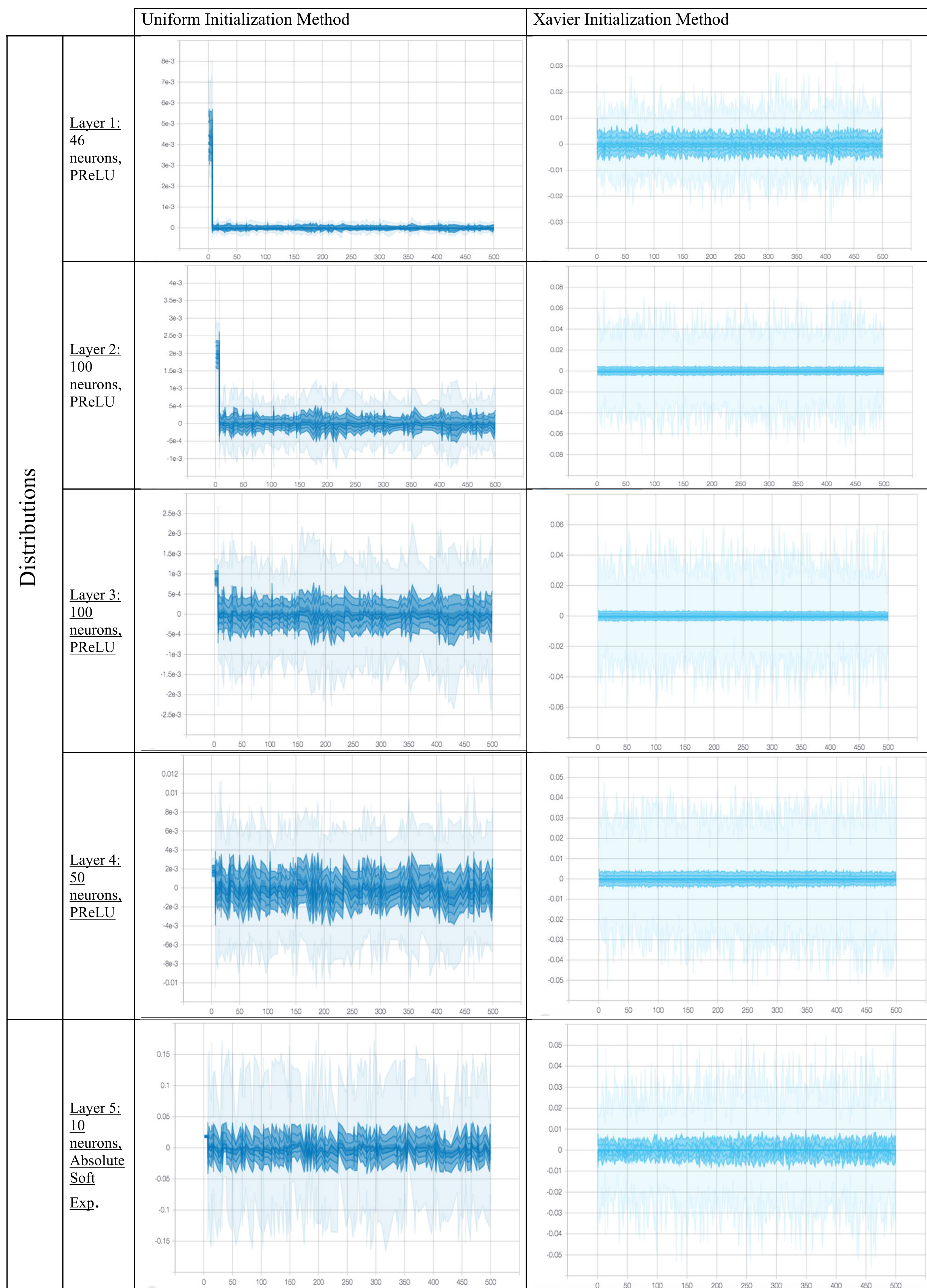


TABLE 1. *(Continued.)* **Distributions and histograms of the backpropagated gradients on 500 epochs, generated via Tensorboard.**

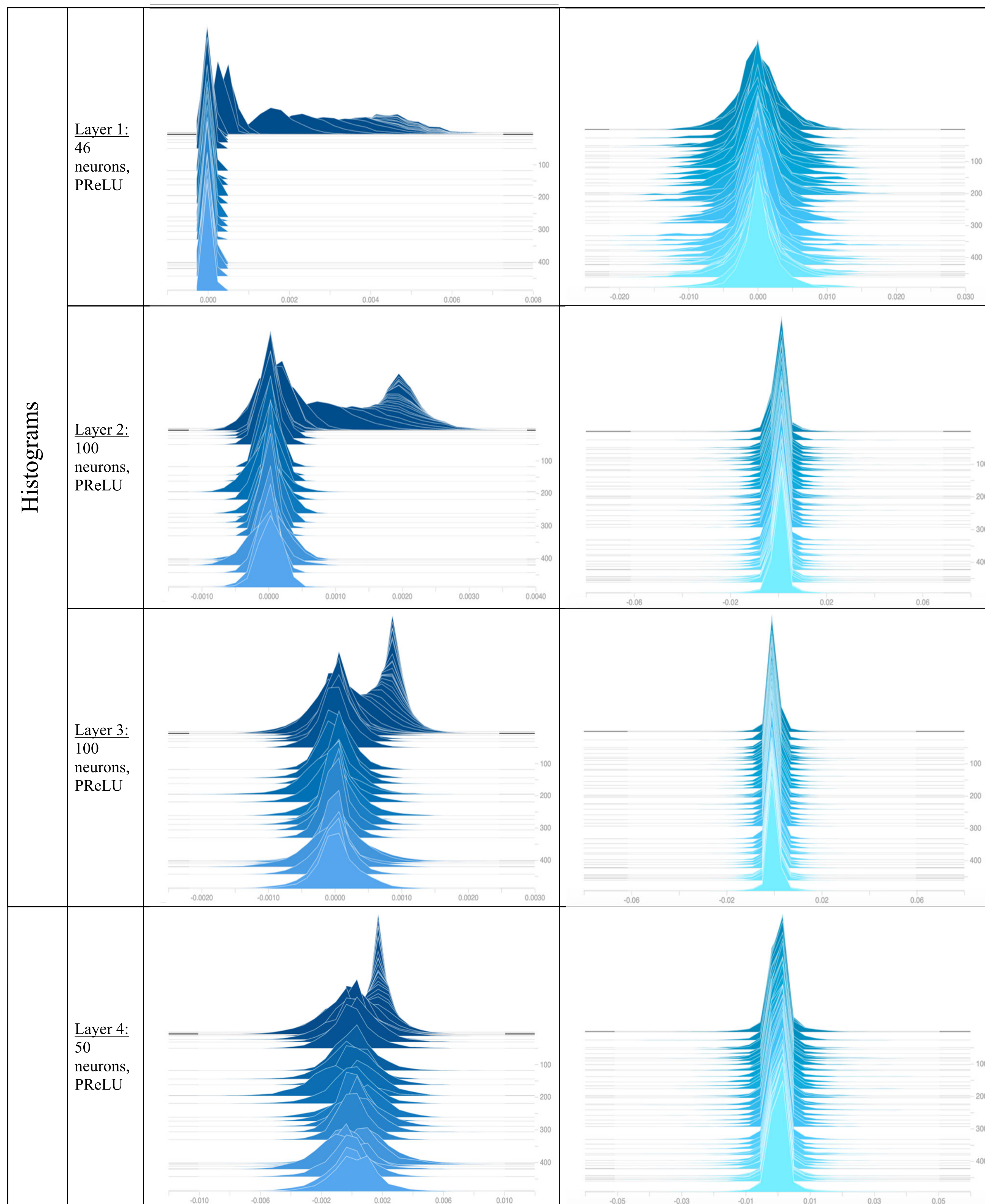
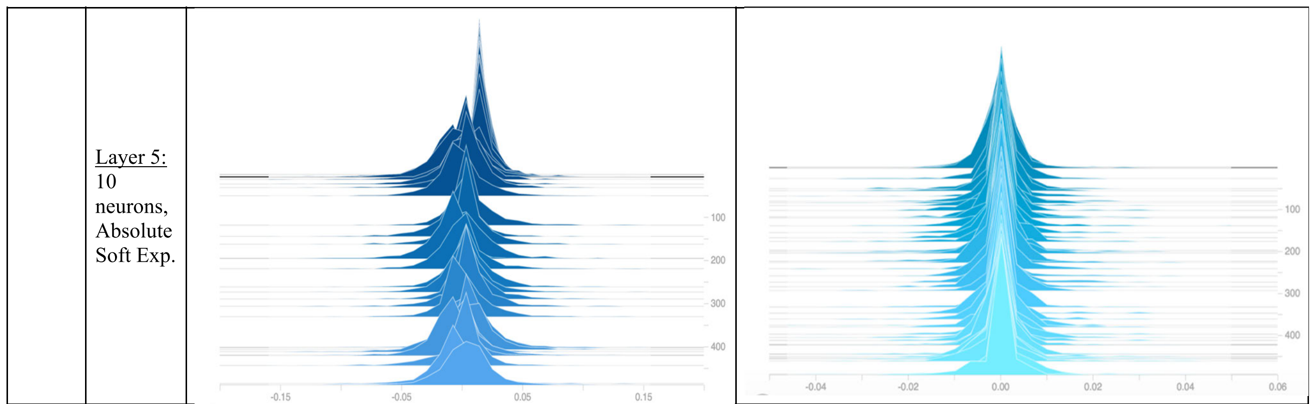


TABLE 1. *(Continued.)* **Distributions and histograms of the backpropagated gradients on 500 epochs, generated via Tensorboard.**

	Layer 5: 10 neurons, Absolute Soft Exp.		

maximum clip value, it is clipped to that threshold. Respectively, the gradient is clipped to the minimum threshold if its value is less than the defined lower limit.

if $||g|| \geq max_threshold$ ***or*** $||g|| \leq min_threshold$ ***then***
    $g \leftarrow threshold$ (respectively)
end if

The interval in which lie the values that gradient can take is defined between *min_threshold* and *max_threshold*, that are predefined. The gradient is represented by g, and $||g||$ is its norm.

2) GRADIENT CLIPPING BY NORM

This second gradient clipping approach is similar to the previous one, the main difference is that in this variant, the gradients are clipped by multiplying their unit vectors by the threshold.

if $||g|| \geq threshold$ ***then***
    $g \leftarrow threshold * g/||g||$
end if

where the *threshold* is a hyperparameter (a predefined value), g represents the gradient and $||g||$ is its norm. Given that $g/||g||$ is a unit vector, resizing the new value of g will make it equal to the *threshold*.

Thus, whether the gradient is clipped by value or by norm, the gradient clipping technique guarantees that the gradient vector g will always have a norm at most equal to the clip value (the *threshold*), which leads the gradient descent to have a correct conduct.

As the clip value or the threshold is considered as a hyperparameter, we had conducted different experiments to look for its best suited value.

In the PyTorch framework, the syntax is different for these two techniques: the syntax to clip gradients by value is:

```
torch.nn.utils.clip_grad_value_(
   model.parameters(),
   clip_value
)
```

And for clipping the gradient by norm, the syntax is:

```
torch.nn.utils.clip_grad_norm_(
   model.parameters(),
   clip_value
)
```

with model.parameters() is an argument that holds all the learnable parameters, i.e. weights, biases and the trainable parameters of the activation functions PReLU and the absolute soft exponential function.

V. CHOSING THE LOSS FUNCTION AND TUNING THE HYPERPARAMETERS

To optimize the hyperparameter values, we have defined a Python dictionary as a search space, holding the hyperparameters names as keys and their values as lists. To scroll through the values of each hyperparameter, we have used the Grid Search algorithm [12] which is an exhaustive approach that allows to divide the search space (the domain of the hyperparameters) into discrete grid, which leads to try every combination of values of this grid. In our case, each combination of the hyperparameters values is considered as an independent run of our training algorithm, that produces a new model.

As per the choice of the loss function, many experiments had been carried out on the following functions:

- Binary Cross-Entropy with Logits Loss (BCEWithLogitsLoss)
- Hinge Embedding Loss
- Cosine Embedding Loss
- Kullback-Leibler Divergence Loss (KLDivLoss)
- L1 Loss

- Smooth L1 Loss
- Mean Square Error (MSELoss, also known as L2 loss)
- Multi Label Soft Margin Loss
- Poisson Negative Log Likelihood Loss (PoissonNLL-Loss)

We have also been wondering whether the choice of the optimizer would impact the performance of the model, hence, we have tried the following optimizers through several experiments:

- Adam
- AdamW
- Adamax
- Adadelta
- Adagrad
- ASGD
- RMSprop
- Rprop
- SGD

After several months of experimenting each combination of these loss functions and optimizers, we have concluded that the best results were achieved using Hinge Embedding Loss, and that changing the optimizer does not really improve the results, so we have opted for the Adam optimizer which has led us to good results. This choice was made based on the score of different models, combining different loss functions and optimizers: the testing method will be discussed in the following section. Throughout the same approach, we have noticed that better results are achieved when small learning rates are used. Taking into consideration these notes, we reduced the hyperparameter search space in order to get less combinations, resulting in less runs (or less models). Thus, we represented the hyperparameters values tested on the following Python dictionary:

```
HyperParams = OrderedDict(
   lr = [0.001, 0.005, 0.0001],
   batch_size = [10, 20],
   num_workers = [0, 4, 16],
   shuffle = [False],
   criterion = [ HingeEmbeddingLoss],
   weights_init = [weights_init_uniform,
weights_init_xavier],
   clip_mthd = ["value", "norm"],
   clip_value = [1,  0.5]
)
```

As the dataloader is extended with every training sample, which predicted heuristic is not admissible, we decided not to shuffle the dataloader to avoid having the same sample twice in a given batch, consequently, the shuffle parameter had been set on 'False'.

The Hinge Embedding Loss function [13] is a loss function that is widely used for classification problems involving non-linear embeddings and semi-supervised learning. It evaluates the similarities between two inputs by pulling together those that are similar and pushing away those that are dissimilar:

$$l_{node} = \begin{cases} y_{node}, & if\ y_{node} > 0 \\ \max\{0, \Delta - \hat{y}_{node}\}, & if\ y_{node} = 0 \end{cases} \quad (17)$$

where y_{node} is the real heuristic value (fed to the model in the training dataset), and $\hat{y}_{node}$ is the predicted heuristic value. This means that, for a given heuristic, we first calculate the mean of its distances, then add a difference between a predefined margin (the latter's value was set to one) and the predicted heuristic value for the distances that are equal to zero on the given heuristic, if this difference is negative, it is replaced by zero (it is not added to the mean of the given heuristic distances). Note that, on the given heuristic, the distance that is equal to zero represents the destination node.

Then, we calculate the total loss function over all the nodes of the graph:

$$l_{graph} = \frac{\sum l_{node}}{N} \quad (18)$$

where N is the graph's order.

In the PyTorch framework, calculating the Hinge Embedding loss over all samples of a batch depends on the 'reduction' argument:

$$l_{batch} = \begin{cases} mean\,(L)\,, & if\ reduction = \text{'}mean\text{'} \\ \text{sum}\,(\text{L})\,, & if\ reduction = \text{'}sum\text{'} \end{cases} \quad (19)$$

where $\text{L} = \{l_{graph1}, l_{graph2}, \ldots, l_{graphbs}\}^{\text{T}}$, and bs is the batch size.

VI. THE CLUSTER'S ARCHITECTURE AND THE FILES HIERARCHY

The choice of relying on a distributed architecture to train our models, generated from the hyperparameters dictionary, was mainly based on the run execution of our program. On a local MacOS system, with 16 GB 1600MHz DDR3 of RAM and 2.2 GHz Quad-Core Intel Core i7 processor, the learning phase took more than one week, and more than 3 weeks when the hyperparameter values on the dictionary were larger. Moving to a cluster on the cloud reduced the execution time to 2-3 days, as we have chosen to rely on Azure F-series based cluster, which is made of virtual machines that are featuring a higher CPU-to-memory ratio: VMs that are optimized for compute intensive workloads, which meets our requirements. Thus, our cluster is constituted of a master node and three workers, each one is a 'Standard_F8' virtual machine with 8 cores and 16 GB memory.

Once every component is coded and fully tested, we have imported the training program, as a Python notebook, from our local machine to the Databricks workspace. We started by creating a storage account to which a Data Lake is connected, to mount this Data Lake afterwards on the cluster to ensure the models' persistence. These steps had allowed to start the migration operation on our code, for it to adapt with the Databricks environment, and to point to the Data Lake storage, on which had been created the following file hierarchy:

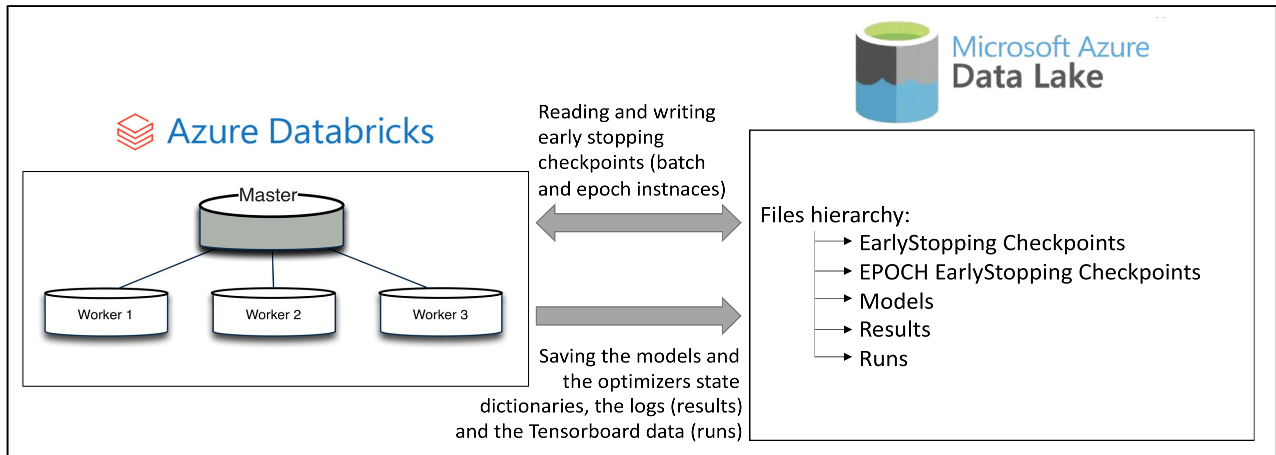


FIGURE 7. Cluster architecture used to train our models.

- *EarlyStopping Checkpoints*: directory used to save the model's state once the early stopping is executed on a batch level
- *EPOCH EarlyStopping Checkpoints*: directory used to save the model's state once the early stopping is executed on an epoch level
- *Models*: directory used to save the model's and the optimizer's state dictionary once the training of each single combination of hyperparameters is completed (also called 'run')
- *Results*: logs of the execution of each run, describing the hyperparameters used and the loss value evolution over epochs (in.csv and.json formats)
- *Runs*: directory used to save the TensorBoard files storing the records of the metrics' evolution

The 'EarlyStopping Checkpoints' directory is constantly accessed during the training: on every batch iteration, if the loss value decreases, a new checkpoint is created replacing the old one. If the loss value stops its decrease pattern, after a certain delay, the early stopping is executed, on a batch level, interrupting the training to create a new epoch checkpoint on 'EPOCH EarlyStopping Checkpoints' directory, which will compare the final loss value of the corresponding epoch to the one from the previous epoch, in order to decide whether to continue the training on the remaining number of epochs or not.

At the end of each run, the final model's state dictionary is saved in the 'Models' directory, as well as the optimizer's state dictionary. A log file is saved too on the 'Results' directory, and a TensorBoard event file, recording all the metrics evolution, is saved on the 'Runs' directory.

VII. TESTING OUR MODELS

Applying the grid search algorithm on the hyperparameter dictionary previously described allowed us to obtain 144 model in total.

To test these models, a new test set of 100 graphs was generated, with a destination node and a heuristic h for each graph (similar to what has been done for the training dataset).

Algorithm 2 Testing the Models

1: Generate 100 new graph, with each a destination node and an appropriate heuristic
2: Create a set of distance matrices and their destination node (*graph*, *endNodePosition*) and another set of heuristics *heur*
3: **for** model **in** trained models
4: score ← 0
5: realPathList ← list()
6: predPathList ← list()
7: **for** i **in** range(100):
8: **for** node **in** *graph*[i] **and** node **!=** *endNodePosition*:
9: realPath ← A*(*graph*[i], node, *endNodePosition*, *h*[i])
10: predHeur ← model(*graph*[i], *endNodePosition*)
11: predPath ← A*(*graph*[i], node, *endNodePosition*, $\hat{h}$[i])
12: Append realPath to the list realPathList
13: Append predPath to the list predPathList
14: **endfor**
15: **if** realPathList ≡ predPathList:
16: score + =1
17: **endif**
18: **endfor**
19: **end for**

Then our models were used to predict a heuristic $\hat{h}$ for each graph of this test set.

To get the score of a model, the A* algorithm had been applied on each test set graph using both the heuristic h and the predicted heuristic $\hat{h}$, from each graph's node the destination node: 9 paths in total, since the graphs used are of order 10. If the nine predicted paths (represented by 'realPathList' in the following algorithm) are identical in steps and costs to the ones obtained with the real heuristic (corresponding to 'predPathList'), we increment the score by one and go to the next graph until the model is tested on all the test set graphs.

TABLE 2. 20 Best results after applying 'Algorithm 2' on the 144 trained models. Models were trained using the hing embedding loss function and the 'Shuffle' attribute of the data loader module set to false.

Learning Rate	Batch Size	Num Workers	Weights Init Method	Clipping Method	Clipping Value	Score	MSE Loss	Hinge Embedding Loss
0.005	10	0	Xavier	Value	0.5	83	34.099209	0.95285
0.005	20	4	Xavier	Value	0.5	82	31.846237	0.986197
0.005	10	0	Xavier	Norm	1	82	34.632912	0.945398
0.001	20	4	Xavier	Norm	1	82	31.047319	1.001722
0.001	20	16	Xavier	Value	1	81	28.860783	1.12479
0.005	10	16	Xavier	Value	0.5	81	32.223289	0.975539
0.001	10	4	Xavier	Value	0.5	81	30.387152	1.047759
0.005	20	16	Xavier	Value	1	80	34.394279	0.94724
0.001	10	16	Xavier	Norm	1	80	31.214827	1.000597
0.001	10	4	Xavier	Norm	1	80	31.018332	0.994154
0.005	20	0	Uniform	Norm	0.5	80	34.79731	0.957661
0.005	10	0	Xavier	Norm	0.5	79	35.028664	0.941573
0.001	10	16	Xavier	Norm	0.5	79	34.086586	0.951123
0.005	10	4	Xavier	Norm	0.5	79	35.073624	0.942628
0.005	10	4	Xavier	Norm	1	79	34.582188	0.943193
0.005	20	0	Xavier	Norm	0.5	79	34.879494	0.943439
0.005	20	16	Uniform	Norm	0.5	79	33.711723	1.112743
0.005	20	16	Xavier	Norm	1	79	34.351665	0.95209
0.005	20	4	Uniform	Norm	1	79	33.972534	1.022105
0.005	20	4	Xavier	Norm	0.5	79	35.204861	0.94047

Therefore, the 'score' variable is the number of graphs on which the model was able to correctly predict the shortest paths from any node to the destination node. It is necessary to mention that the number of epochs had been fixed at 500. Displayed below the obtained results for the best 20 model:

Most of our models reached a score that exceeds 75%, meaning that they were able to correctly predict heuristics that allowed us to find the shortest path from each graph node to the destination node on more than 75 graphs, applying the A* algorithm, up to 83 graphs from the 100 graphs of the test set.

About the admissibility of these predicted heuristics, once tested, they have been checked as to make sure that their values are lower than the real cost: we had observed that the predicted heuristics are much lower than the desired output (the provided heuristic, which is already admissible). For example, let us use the best model obtained, of which the score is 83%, to predict a heuristic for the following graph, with the destination node index 3 (node indexes are in [0,9]):

$$\begin{pmatrix} 0 & 7 & 8 & 13 & 0 & 0 & 2 & 12 & 1 & 7 \\ 7 & 0 & 9 & 2 & 10 & 12 & 4 & 0 & 1 & 14 \\ 8 & 9 & 0 & 3 & 4 & 3 & 9 & 12 & 10 & 4 \\ 13 & 2 & 3 & 0 & 0 & 7 & 15 & 11 & 0 & 2 \\ 0 & 10 & 4 & 0 & 0 & 12 & 5 & 13 & 14 & 3 \\ 0 & 12 & 3 & 7 & 12 & 0 & 9 & 12 & 10 & 5 \\ 2 & 4 & 9 & 15 & 5 & 9 & 0 & 11 & 11 & 3 \\ 12 & 0 & 12 & 11 & 13 & 12 & 11 & 0 & 11 & 10 \\ 1 & 1 & 10 & 0 & 14 & 10 & 11 & 11 & 0 & 4 \\ 7 & 14 & 4 & 2 & 3 & 5 & 3 & 10 & 4 & 0 \end{pmatrix}$$

The provided heuristic is: [4, 2, 3, 0, 5, 6, 5, 11, 3, 2]. And the predicted heuristic is: [0.11479, 0.51802, 0.08716, 0.00000, 0.28879, 0.85889, 0.35404, 0.31780, 0.23488, 0.50746].

Using A* search algorithm with the provided heuristic, these are the paths obtained from each node of the graph to the destination node, with their respective costs:

Using the provided heuristic
Node 0 → Node 8 → Node 1 → Node 3 : ***Cost = 4***
Node 1 → Node 3 : ***Cost = 2***
Node 2 → Node 3 : ***Cost = 3***
Node 3 : ***Cost 0***
Node 4 → Node 9 → Node 3 : ***Cost = 5***
Node 5 → Node 2 → Node 3 : ***Cost = 6***
Node 6 → Node 9 → Node 3 : ***Cost = 5***
Node 7 → Node 3 : ***Cost = 11***
Node 8 → Node 1 → Node 3 : ***Cost = 3***
Node 9 → Node 3 : ***Cost = 2***

And when applying A* with the predicted heuristic:

Using the predicted heuristic
Node 0 → Node 8 → Node 1 → Node 3 : ***Cost = 4***
Node 1 → Node 3 : ***Cost = 2***
Node 2 → Node 3 : ***Cost = 3***
Node 3 : ***Cost 0***
Node 4 → Node 9 → Node 3 : ***Cost = 5***
Node 5 → Node 2 → Node 3 : ***Cost = 6***
Node 6 → Node 9 → Node 3 : ***Cost = 5***
Node 7 → Node 3 : ***Cost = 11***
Node 8 → Node 1 → Node 3 : ***Cost = 3***
Node 9 → Node 3 : ***Cost = 2***

It can be observed that the predicted heuristic is admissible, and that the predicted paths correspond in cost and in steps to the ones obtained using the provided heuristic (generated with the Dijkstra algorithm).

VIII. IMPLEMENTING THE GENETIC ALGORITHM

Genetic algorithms have shown their efficiency when dealing with optimization problems, as they are generally used to find optimal or near-optimal solutions. This concept has been developed by John H. Holland [14] when investigating the theory and practice of algorithmic evolution.

Inspired by Charles Darwin's theory of evolution, this algorithm is based on the ideas of natural selection and genetics: it simulates the principle of survival of the fittest among individuals of consecutive generations for solving a problem.

The process of a genetic algorithm starts by initializing a pool or population of potential solutions to the chosen problem. These solutions are then recombined and mutated in order to produce new children, as it is the case in natural genetics. Based on the objective function, we assign to each eventual solution, commonly referred to as an individual, a fitness value, while giving a higher selection probability to fitter individuals, which is in line with the Darwinian principle of survival of the fittest. This leads to a better individuals' evolution over generations.

Since some specific terminology might be encountered in genetic algorithms, below are the definitions of the key words that are used in this article:

- Population: represents a subset of candidate solutions to the given problem.
- Chromosomes: represent a solution or an individual
- Genes: represent a one element position of a chromosome

In our approach, the initial population had been created by selecting the best 20 models, previously trained, based on their scores. Thus, a chromosome or an individual in our case is represented by a neural network model, and the values of its trainable parameters are considered as genes.

The main objective of using genetic algorithms is to verify if this approach can enhance our models' performance to obtain better scores.

A. INITIALIZING THE POPULATION WITH INDIVIDUALS AS DATACLASS INSTANCES

Python dataclasses are mainly aimed at helping to conceive and handle data-oriented classes, by adding a couple of convenient mechanisms such as defining the data structure, adding an easy initialization to it, representing the object and comparing it to other dataclass objects. This makes it different from behavior-oriented classes that exposes a number of methods that are being called in order to process actions.

Consequently, our population's individuals (our models) were handled as dataclass instances, since they store all of the trainable parameters' values as tensors.

A 'GA_individual' dataclass was then created, with the following attributes:

- *sort_index*: with the type 'int', helping to sort the population's individuals by their scores
- *model*: with the type 'Network()' as our models' class
- *optimizer*: with the type 'torch.optim' as the optimizer's class
- *GA_run*: instantiating a new class created upstream to store the model's hyperparameters (learning rate, batch size, clip method, weight initialization method...)
- *score*: with the type 'int' as our model's test score

Once the dataclass decorator is implemented, the Python's dunder methods such as **'__init__'** and **'__repr__'** are automatically generated from the above attributes:

```
1  '''
2  Create an individual class: dataclass
3
4  Attributes:  -sort_index
5               -model
6               -optimizer
7               -GA_run
8               -score
9  '''
10
11 @dataclass(order=True)
12 class GA_individual:
13     sort_index: int = field(init=False, repr=False)
14     model: Network()
15     optimizer: torch.optim
16     GA_run: GA_run
17     score: int
18
19     def __post_init__(self):
20         self.sort_index = self.score
21
22     def __str__(self):
23         return f'score >> {self.score}\n {self.GA_run.__str__()}'
```

FIGURE 8. Implementing the individuals as a Python dataclass.

The main purpose of the *sort_index* attribute is to allow the comparison of individuals by their score, as the **'order'** flag had been set on true on the dataclass decorator, then we have defined the ordering rule of these dataclass objects into the **'__post_init__'** method: adding the condition **'init = False'** with the **'field'** function means that this attribute will not be part of the initializer, thus it doesn't need to be mentioned when creating a new individual. As for the **'repr = False'** condition, it means that the *sort_index* attribute will not be part of the string representation of these objects.

The **'__str__'** dunder method is defined to describe in a string format the score and the hyperparameters of the individual (the model).

Accordingly, at the very beginning of our genetic algorithm approach, we have created a 'GA_individual' instance for each model, then we have loaded the model and the optimizer state dictionaries into the *model* and the *optimizer* attributes, and stored the hyperparameters on an instance of the *GA_run* class, to finally assign the scores obtained on the testing phase to these instances via the *score* attribute.

Once the population is created, the individuals were sorted by their scores: from the model with the highest to the lowest score.

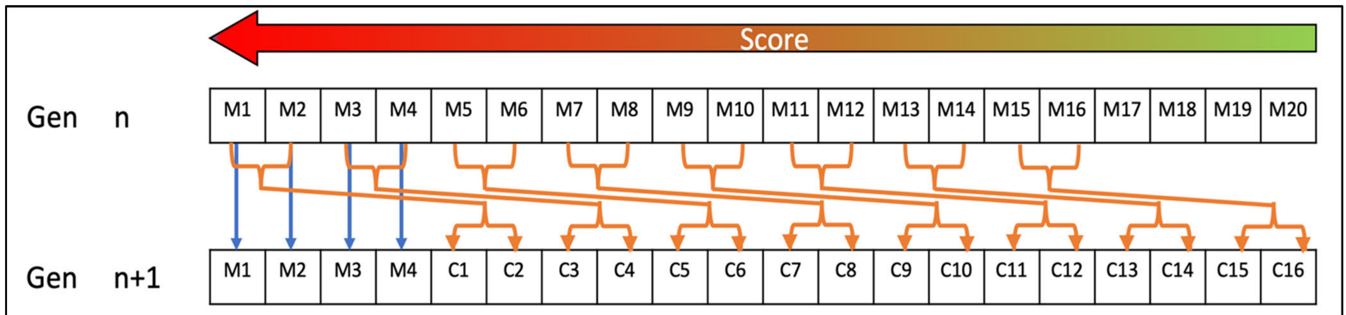


FIGURE 9. Selection with an elitism rate of 20%.

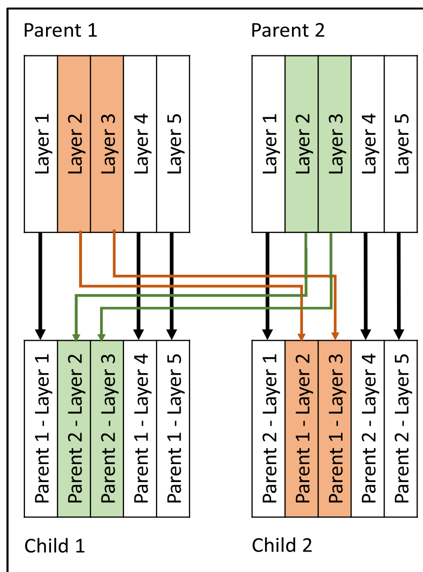


FIGURE 10. Crossover example over two neural networks.

B. THE SELECTION

The selection part of the genetic algorithm consists on picking the fittest individuals first, to get better children (with the most likely high scores). While experimenting the genetic algorithms approach, we have concluded that applying the elitism [15] principle conducted us to obtain better and more significant results.

Elitism is a strategy that helps reduce the genetic drift by guaranteeing a place in the next generation to the fittest individuals: to the models with the higher scores, which means that they will still get to be selected as parents. The elites are therefore present with their children in the next generation. In the evolution theory, such a technique can push the selection towards improving the convergence speed.

We have then decided to use an elitism rate of 20%, keeping the best four models as elites. To keep a constant population's length, we drop the last four models with the lowest score, as in the below figure ('M' for model, 'C' for child and 'Gen n' and 'Gen n+1' are two successive generations).

C. THE CROSSOVER

In this step, new models are going to be created, inheriting some characteristics from both of their parents. Our approach to this step is to cross some predefined hidden layers from the two models that are considered as parents, to obtain new children in the next generation with a new combination of hidden layers:

When creating the new children, we are actually transferring the values of all the weights related to their parents' trained models, through the models' state dictionaries: the child 1 for instance will have the weights of layer 1, 4 and 5 from parent 1 and those of layer 2 and 3 from parent 2.

However, each child inherits the hyperparameters' values and the optimizer's state dictionary from one parent: child 1 from parent 1 and child 2 from parent 2. These genes will be used later to train the children on a new generated dataset.

It can be noted that this crossover's approach on neural networks is similar to the two-point crossover, which consists on choosing two points on the parent chromosomes, and then swipe the genes in between with another parent.

D. TRAINIG THE CHILDREN AND UPDATING THEIR SCORES

After the crossover step, a new training dataset is generated, as it was done in the dataset creation section (using Dijkstra algorithm). The new generated individuals are then trained on this new training dataset, using the hyperparameters stored in their *GA_run* attribute, and with the last state of their respective optimizer, loaded as a new instance from the Python state dictionary stored in the *optimizer* attribute.

E. THE MUTATION

The mutation step is a technique that is used to maintain some amount of randomness and diversity in genetic algorithms, from one generation to the next. Thereby, it increases the likelihood of generating individuals with better fitness values. In our case, the mutation function had been defined as a change on the weights of a random neuron on the fourth layer, updating it with the same neuron weights from a random parent.

However, it is generally advised that the mutation rate should be low [16], as high mutation rates tend to convert the algorithm towards a random search. We have then adopted a

TABLE 3. Score's evolution over a sample of generations.

Input Models		Gen 23		Gen 76	
Score	**Nb of models**	**Score**	**Nb of models**	**Score**	**Nb of models**
77	0	77	0	77	1
78	0	78	1	78	1
79	9	79	0	79	1
80	4	80	6	80	5
81	3	81	3	81	0
82	3	82	2	82	8
83	1	83	2	83	2
84	0	84	5	84	0
85	0	85	1	85	2
86	0	86	0	86	0
Mean	80.15	**Mean**	81.8	**Mean**	81.3
Median	80	**Median**	81.5	**Median**	82
Mode	79	**Mode**	80	**Mode**	82
Maximum	83	**Maximum**	85	**Maximum**	85
Gen 107		**Gen 113**		**Gen 174**	
Score	**Nb of models**	**Score**	**Nb of models**	**Score**	**Nb of models**
77	0	77	0	77	0
78	0	78	1	78	0
79	0	79	0	79	1
80	3	80	0	80	2
81	4	81	3	81	4
82	2	82	3	82	3
83	5	83	5	83	4
84	5	84	6	84	4
85	1	85	1	85	2
86	0	86	1	86	0
Mean	82.4	**Mean**	82.85	**Mean**	82.35
Median	83	**Median**	83	**Median**	82.5
Mode	83, 84	**Mode**	84	**Mode**	81, 83, 84
Maximum	85	**Maximum**	86	**Maximum**	85

fitness-based adaptive mutation probability [17]:

$$p = p_{max} * (1 - \frac{f}{f_{max}}) \tag{20}$$

where p is the mutation probability of the chromosome, p_{max} is the maximum mutation probability (that we have defined as 0.1, it is kept constant throughout all the generations), f is the fitness of the chromosome (its updated score in our case) and f_{max} is the fitness of the best chromosome in the population.

This approach to define the mutation rate ensures that the best individual will always have zero mutation probability, whereas the poorest one will have the highest probability.

IX. DISCUSSING THE GENETIC ALGORITHM RESULTS

It is important to mention that the same cluster's architecture was used to launch the genetic algorithm program, creating the same file hierarchy on each generation.

We have observed, using the genetic algorithm approach, that the performance of our models may be different from one generation to another: some individuals were found to reach higher scores than the ones on the next generation. It means that the scores did not keep increasing constantly through the generations' evolution, as our models went through a training process on a new generated dataset in each generation.

By testing our genetic algorithm approach on 200 generations, we have been able to reach a score of 86%. As it can be observed on the following table, the values of both the mean and the median are relatively increasing throughout the generations, which reflects the improvement of our models' performance. It is also the case for the mode of our populations, which describes the most frequent value of the score variable: starting with a mode of 79 in our initial population, we have been able to reach a mode of 84 after the 100th generation, up to 6 models with this score value in the 113th generation.

Our objective from using the genetic algorithm approach was to enhance the performances of our models, however this approach still needs to be optimized for execution time, crossover techniques and mutation rate which are research projects of their own.

X. CONCLUSION

In this paper, we have conceived a PyTorch based neural network to predict admissible heuristic, for a given graph and destination node, that can be exploited by the A* pathfinding algorithm to find the shortest path from any graph node to the destination node. During the training phase, we have used the Dijkstra algorithm to produce a training data set, by assigning a heuristic to each graph represented by a randomly created distance matrix: the start node in the Dijkstra algorithm is considered as the destination node on the A* search algorithm.

Our predicted heuristics should meet the admissibility condition, which was addressed by implementing a dynamic dataloader: samples of which the predicted heuristics are greater than the real costs are reinserted at the end of the dataset that is used to generate a new dataloader, then the learning process is resumed at the iteration following the batch we were on, at the current epoch.

This dynamic dataloader regenerating process is interrupted by calling the early stopping method to avoid the overfitting problem: at each iteration on the same epoch, the training and the validation errors are compared and a checkpoint is created to save the model's state. When the gap between the training and the validation loss starts to grow, as the validation loss starts increasing, and after a predefined number of iterations called *patience* (a delay to make sure that the validation loss will not decrease), the epoch is stopped and the last checkpoint will be saved as the model's state: the dataloader is no more regenerated. After triggering the early stopping method to stop each epoch, the last validation error value will be saved and passed to another instance of this method, that will compare it to the one from the previous epoch and then stop the whole training process when its value starts increasing (as explained on figure 6).

Reloading the dataloader with samples that do not provide admissible heuristics was our proposed solution to implement a condition-oriented training process, it may be useful or exploited when facing a problem where the output values should meet a certain condition or requirement.

Once the training phase completed, we tested the resulting models by applying the A* search algorithm on a new test dataset made of a new set of 100 graphs, with a destination node and a heuristic assigned to each graph (similar to what had been done to create the training dataset). Then, we attributed a score to each model by comparing the resulting paths from each node to the destination node, using the test set heuristic and the predicted one on all the graphs from the test dataset. Our best models have reached a score that exceeds 80%, up to 83%. However, we have been wondering whether the use of an optimization algorithm will enhance our models' performance.

We have then opted for a genetic algorithm approach, to check if better scores can be obtained. Thus, we initialized the population with the best 20 models (based on their scores). Then, we have proceeded to the implementation of the steps constituting this approach: selecting the fittest individuals or models, identifying the elites to extend their impact on the next generation by keeping them unmodified, executing the crossover to obtain new individuals, then mutating some of these new individuals to add a bit of diversity, which increases the likelihood of generating individuals with better scores. Therefore, we were able, throughout the predefined number of generations, to enhance the performance of our initial models by increasing their scores, as shown in the previous table.

However, the dynamic dataloader approach along with improving the models' scores through a genetic algorithm turned out to be time-consuming and adds more complexity to our programs, which explains our choice to execute them on a distributed architecture. Thus, we have created a Databricks

cluster of virtual machines adapted to intensive workloads, using the Microsoft Azure cloud provider, and have linked it to an Azure Data Lake to store our models and other related files (logs, checkpoints, state dictionaries, etc.).

REFERENCES

[1] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Jul. 1968.

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, and G. Chanan, "PyTorch: An impressive style, high-performance deep learning library," in *Proc. Conf. Neural Inf. Process. Syst.*, Vancouver, BC, Canada, Dec. 2019, pp. 1–12.

[3] Microsoft. *What is Azure Databricks?—Azure Databricks—Microsoft Learn*. Accessed: Dec. 8, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/databricks/introduction/

[4] O. Amine and M. Mohammed, "Predicting a search algorithm heuristics using neural networks," in *Proc. Int. Conf. Electr., Comput. Energy Technol. (ICECET)*, Cape Town, South Africa, Dec. 2021, pp. 1–12.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1026–1034.

[7] L. B. Godfrey and M. S. Gashler, "A continuum among logarithmic, linear, and exponential functions, and its potential to improve generalization in neural networks," in *Proc. 7th Int. Joint Conf. Knowl. Discovery, Knowl. Eng. Knowl. Manage.*, 2015, pp. 481–486.

[8] D. Berrar, "Cross-validation," in *Reference Module in Life Sciences*. Amsterdam, The Netherlands: Elsevier, Jan. 2018.

[9] L. Prechelt, "Automatic early stopping using cross validation: Quantifying the criteria," *Neural Netw.*, vol. 11, no. 4, pp. 761–767, 1998.

[10] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *J. Mach. Learn. Res.*, vol. 9, Jan. 2010, pp. 249–256.

[11] J. Qian, Y. Wu, B. Zhuang, S. Wang, and J. Xiao, "Understanding gradient clipping in incremental gradient methods," in *Proc. 24th Int. Conf. Artif. Intell. Statist.*, San Diego, CA, USA, vol. 130, 2021, pp. 1504–1512.

[12] P. Liashchynskyi and P. Liashchynskyi, "Grid search, random search, genetic algorithm: A big comparison for NAS," 2019, *arXiv:1912.06059*.

[13] PyTorch Contributors. *HingeEmbeddingLoss—PyTorch 1.13 Documentation*. Accessed: Dec. 8, 2022. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.HingeEmbeddingLoss.html

[14] J. H. Holland, "Genetic algorithms," *Sci. Amer.*, vol. 267, no. 1, pp. 66–73, 1992.

[15] D. Bhandari, C. A. Murthy, and S. K. Pal, "Genetic algorithm with elitist model and its convergence," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 10, no. 6, pp. 731–747, Sep. 1996.

[16] R. N. Greenwell, J. E. Angus, and M. Finck, "Optimal mutation probability for genetic algorithms," *Math. Comput. Model.*, vol. 21, no. 8, pp. 1–11, Apr. 1995.

[17] A. Basak, "A rank based adaptive mutation in genetic algorithm," *Int. J. Comput. Appl.*, vol. 175, no. 10, pp. 49–55, Aug. 2020.

OUARDI AMINE was born in Marrakesh, Morocco, in 1992. He received the M.S. degree in business intelligence from the Faculty of Science Dhar El Mahraz, Sidi Mohammed Ben Abdellah, Fez, Morocco, in 2018. He is currently pursuing the Ph.D. degree in artificial intelligence with the Laboratory of Signals, Distributed Systems and Artificial Intelligence, École Normale Supérieure de l'Enseignement Technique (ENSET), Mohammedia, Morocco. His research interests include neural networks and heuristic-based pathfinding algorithms.

MESTARI MOHAMMED received the M.A. degree from the École Normale Supérieure de l'Enseignement Technique (ENSET), Mohammedia, Morocco, in 1991, and the Ph.D. degrees in applied mathematics and artificial intelligence from the Faculty of Science Ben M'Sick, Hessan II University, Casablanca, Morocco, in 1997 and 2000, respectively. He is currently a Professor in applied mathematics with ENSET, and the Head of the Artificial Intelligence Research Team with the Laboratory of Signals, Distributed Systems and Artificial Intelligence. His current research interests include neural networks for signal processing, neural networks hardware implementation, high-speed and/or low-power techniques and systems for neural networks, and theoretical issues directly related to hardware implementation.