DOI: 10.15439/2022F192
ISSN 2300-5963 ACSIS, Vol. 30

NiaNet: A framework for constructing Autoencoder architectures using nature-inspired algorithms

Sašo Pavlič
Faculty of Electrical Engineering and Computer Science
University of Maribor
Koroška cesta 46, 2000 Maribor
Slovenia
Email: saso.pavlic@student.um.si

Sašo Karakatič
Faculty of Electrical Engineering and Computer Science
University of Maribor
Koroška cesta 46, 2000 Maribor
Slovenia
Email: saso.karakatic@um.si

Iztok Fister Jr.
Faculty of Electrical Engineering and Computer Science
University of Maribor
Koroška cesta 46, 2000 Maribor
Slovenia
Email: iztok.fister1@um.si

***Abstract*—Autoencoder, an hourly glass-shaped deep neural network capable of learning data representation in a lower dimension, has performed well in various applications. However, developing a high-quality AE system for a specific task heavily relies on human expertise, limiting its widespread application. On the other hand, there has been a gradual increase in automated machine learning for developing deep learning systems without human intervention. However, there is a shortage of automatically designing particular deep neural networks such as AE. This study presents the NiaNet method and corresponding software framework for designing AE topology and hyper-parameter settings. Our findings show that it is possible to discover the optimal AE architecture for a specific dataset without the requirement for human expert assistance. The future potential of the proposed method is also discussed in this paper.**

***Index Terms*—AutoML, autoencoder, deep learning, nature-inspired algorithms, optimization**

I. Introduction

DEEP neural networks (DNN)s have seen a surge in popularity in recent years, with applications in various domains. Their potential began with better-than-human performance in tasks including image recognition, natural language processing, and product recommendation [1]–[3] and progressed to sophisticated tasks such as protein-folding, self-driving cars, and weather forecasting [4]–[6]. Even if DNN-based systems are not yet intelligent, we may employ them wisely to tackle complex problems. It is projected that with the current and future rise of computational resources and the large availability of data, DNN will benefit. Greater computer capabilities enable us to build more sophisticated and complex DNN topologies, while larger datasets will improve training performance.

Despite all the outstanding achievements and expectations of employing DNNs, data scientists and researchers are still dealing with DNN construction. The DNN construction phase can be a resourcefully expensive process and contributes to the global carbon footprint [7]. The construction phase specifies DNN topology, which includes defining layers, neurons, and connections. After the DNN topology has been designed, the optimum hyper-parameters must be chosen. Variables such as topology and hyper-parameters influence the final performance of DNN, and those variables are often limited by the researcher's prior knowledge and experience. The most significant disadvantage is the time spent by a human expert manually attempting to determine relevant variables for a specific search space.

This research was funded by the Slovenian Research Agency (research core funding No. P2-0057)

In the literature, we can find many studies tackling the previously mentioned problem. The techniques presented in the studies attempt to automatically optimize certain aspects of the entire DNN creation process for a given search space [8]. One efficient method is to use an technique that employs population-based nature-inspired algorithms (NIA)s [9], [10]. Mostly because such a method is good at optimizing highly computationally expensive problems. When comparing the scale of studies, we can see that they are focusing on either the topology construction (without the weights) or topology with the weights simultaneously [11]. This is to keep the search space as small as possible. More details on the related work will follow in the following sections.

Motivated by these methods, we propose NiaNet (Nature-Inspired Algorithms for Deep Neural NeEtwork creaTion), a method capable of auto-designing the novel autoencoder (AE) model with only the dataset as input. The NiaNet is simultaneously constructing the AE topology and setting the optimal hyper-parameters. This process aims to optimally explore the search space by utilizing nature-inspired algorithms. The success of our proposed method, NiaNet, is determined by evaluating the best performing AE model with the fitness function, where reconstruction error, training duration, and topological simplicity are calculated. We find an advantage in our proposed method's ease of use and adaptation to varied datasets while achieving promising results.

Altogether, the main contributions of this paper can be summarized as follows:

- We propose a method NiaNet for constructing the autoencoder topology and hyper-parameter setting.
- We present the automated machine learning (AutoML) framework capable of applying the NiaNet method on a given dataset.

- We test the NiaNet method on a well-known diabetes dataset.

The remaining structure of this paper is as follows. Section II briefly describes the related work. Section III and Section IV presents the used framework and proposed method NiaNet. Obtained results of the experiment are presented in Section V. The last Section VI contains the conclusion.

II. Related work

In this section, we review relevant strategies for building DNN models without the need for human intervention. Since the construction of DNN is a highly complex problem that can also be represented as an optimization problem, scientists are looking for the potential of applying nature-inspired algorithms to cope with this problem. These algorithms are highly efficient in finding the solutions to multi-dimensional problems such as DNN construction. This section looks at target optimization problems that have received considerable attention in the literature. All of these strategies aim to discover the DNN topology efficiently in multiple dimensions. They vary in that they may either search simply the DNN topology (neurons and connections) or DNN topology and weights. The time axis or computational resources can represent efficiency before converging to the optimal solution.

A. Neuroevolution (NE)

A population of genetic encoding of artificial neural networks (ANN)s is evolved in neuroevolution to identify a network that solves a given task. Each encoding in the population (a genotype) is chosen one at a time and decoded into the neural network corresponding to it (a phenotype). The performance of this network in the task is then measured over time, yielding a fitness value for the relevant genotype. As a result, the process resembles an intelligent parallel search for superior genotypes, and it continues until a specific fitness threshold value is found or evolution reaches a specific generation limit [12]. Neuroevolution methods differ by type of encoding genotype to phenotype:

- A direct encoding will explicitly specify the direct connection between phenotype and genotype.
- An indirect encoding specifies the rules or parameters on how the phenotype is built from the genotype.

The following evolutionary algorithms, i.e., genetic algorithm (GA) [13], memetic algorithm (MA) [14], and particle swarm optimization (PSO) [15], showed promising results when tackling this problem.

Readers are invited to read a recent comprehensive study that presents current trends and future challenges in neuroevolution, as well as various types of neuroevolution and their strengths, and limitations [16].

B. Evolutionary neural architecture search

Neural Architecture Search (NAS) is a technique that automatically designs artificial neural networks. One of the many modifications of this technique is Evolutionary NAS (ENAS). It is a bio-inspired automated neural network architecture design technique that follows the core principles of biological evolution [17]. Its goal is to identify a network topology that will give the best result on a given task. The three main components of the NAS method are as follows [18]:

a) Search space: It defines the boundaries inside which the search is allowed. This can be a set of rules for topology, layer number, layer type, and optimizers. Its size represents the set of all possibilities.

b) Search strategy: It defines the method for exploring the search space. The majority of the work on the NAS approach was focused on addressing this aspect. Since it is always challenging to determine which optimization methods work best and how to adapt or change them to yield better results. ENAS technique is using evolving ANN (EANN) as a search strategy. The EANN strategy is used to evolve ANN's connection weights, topology, and learning rules [19].

c) Evaluation strategy: Alternatively, sometimes called performance estimation strategy, evaluates the ANN offspring. Such evaluation is done prior to construction and training phase. This method primarily depends on many factors, such as search space size, datasets size, depth of topology, and others. To accurately measure the ANN offspring performance [18] many new methods have been proposed to reduce the time, and computation resources [20].

C. Structure learning

The method of utilizing data to train the linkages of a Bayesian network is known as structural learning. The method's goal is to represent the data in a graph format, providing a good balance of expressive power and querying performance. Bayesian networks are a type of structured knowledge representation in which domain variables are represented as nodes in a graph whose structure encodes their relationships [21]. However, these techniques need a lot of computing power, making the solution unsuitable for most applications with limited computing power and time.

III. Automated machine learning

As mentioned in the introduction, deep learning has been applied in various fields to solve challenging artificial intelligence (AI) tasks in recent years. Such diversification often leads to specific cases where even field experts operate on trial-and-error. This substantially increases the resources and time needed to create well-performing DNN models [22]. To reduce the development cost and automate the entire machine learning pipeline, an AutoML methodology was introduced. Its pipeline consists of data processing, feature engineering, model generation, and model evaluation. The goal is to be able to automate the complicated process of selecting pipeline components so that a user only needs to specify a dataset and an appropriate pipeline will be built automatically [23]. This frees up a human specialist to concentrate on other areas of the process.

This section introduces the AutoML framework, which utilizes our proposed method, NiaNet, in a model generation stage. In symbiosis, both the framework and method construct a deep autoencoder topology and their hyper-parameters to discover the best possible ML pipeline for an input dataset. The framework (see. Fig. 1) is built using a layer-style layout architecture with multiple components.

A. Data Ingestion

Collecting and importing data into the ML pipeline is known as data ingestion. Acquiring data can be a complex component and one of the most challenging tasks because we need to have solid business and data understanding abilities [24]. The ML pipeline components will be influenced by the dataset used. The user performs this operation before the pipeline begins.

B. Dataset processing

Data processing is the first stage in the AutoML pipeline. There are numerous approaches for processing data to be used to build models. Real-world data is commonly skewed; there is missing data, which is often noisy. As a result, processing the data is required to make it clean and processed so that it may be run through the ML algorithms. The yellow section in Fig. 1 illustrates the data processing process. As authors in paper [22] explained, it may be divided into three processes: data collecting, data cleaning, and data augmentation. Data collection is an important stage in creating a new dataset or expanding an old one. The data cleaning process filters noisy data so that subsequent model training is not affected. Data augmentation is critical for increasing model robustness and improving model performance. The three aspects will be discussed in further depth in the following subsections. This stage is not yet automated in our framework.

C. Feature engineering

Feature engineering is the next stage in our AutoML pipeline. It usually consists of feature extraction, feature selection, and feature construction. In our ML pipeline, only the process feature selection is utilized. This process builds a feature subset based on the original set by reducing irrelevant or redundant features. This simplifies the model, preventing over-fitting and boosting model performance [22]. There are many manuals or automated ways of selecting the optimal feature set for a given dataset [10].

D. Model generation

Model generation is divided into two components, search space and optimization method, as shown in the 3rd section in Fig. 1. Where search space defines the AE topology and hyper-parameters. The AE architecture refers to a complete blueprint of DNN components such as:

- Topology shape (symmetrical, asymmetrical)
- Size of input, hidden and output layers
- Number of hidden layers
- Number of neurons in hidden layers

On the other side, in our AutoML framework, the following hyper-parameters are available in the search space:

- Activation function
- Number of epochs
- Learning rate
- Optimizer

Another component in the model generation stage is the optimization method. This component is responsible for finding the optimal solution within the edges of search space - parameter values to construct and train a given AE model. The solution is a one-dimensional array of elements from the above search space dimensions, each representing one of the parameters we are trying to optimize. The task of choosing the optimum solution is an iterative process. In this process, we are using the micro-framework NiaPy [25], which is an excellent tool for using the collection of nature-inspired algorithms for optimizing a given problem, such as ours. Each returned solution array from NiaPy framework is mapped according to equations [2-8]. More details are presented in section Proposed method. The AE model is created and trained in PyTorch using mapping rules that are controlled by the proposed method.

E. Model evaluation

The performance of an AE model must be evaluated after it has been constructed. The first process in the model evaluation phase is to use the DNN training and testing technique to evaluate each solution produced by the proposed method. However, this process requires a significant amount of time and computation resources. Once the AE model has been trained, the reconstruction loss and model complexity are evaluated based on the equations 10 and 11. In addition, the fitness function 11 is calculated, and the fitness value is passed back to the optimizer algorithm, which generates a new solution. Sections 3 and 4 of Fig. 1 show the communication flow between model generation and model evaluation.

F. Fittest AE model

After our iterative AutoML pipeline is completed, the fittest AE model is returned. To put it simply, the proposed model is optimal in terms of reconstruction loss and model complexity.

IV. Proposed method

In the following section, we present in detail our proposed method, NiaNet [1]. This research will study whether an AE neural network with topology and hyper-parameters set by our proposed method will provide better encoding performance than AE designed manually. We believe that a nature-inspired search can discover a novel solution that may be hidden by human experts who are limited by their previous experience and knowledge. Our proposed method leans to be very straightforward and utilized on different datasets. This allows users to automatically perform some ML steps when using our AutoML framework, without manually searching for the right AE building blocks.

The method is based on applying the nature-inspired optimizer (we use a collection of algorithms in the NiaPy framework) to the problem of constructing AE typed neural networks. The solution is a one-dimensional array of seven

[1]https://github.com/SasoPavlic/NiaNet

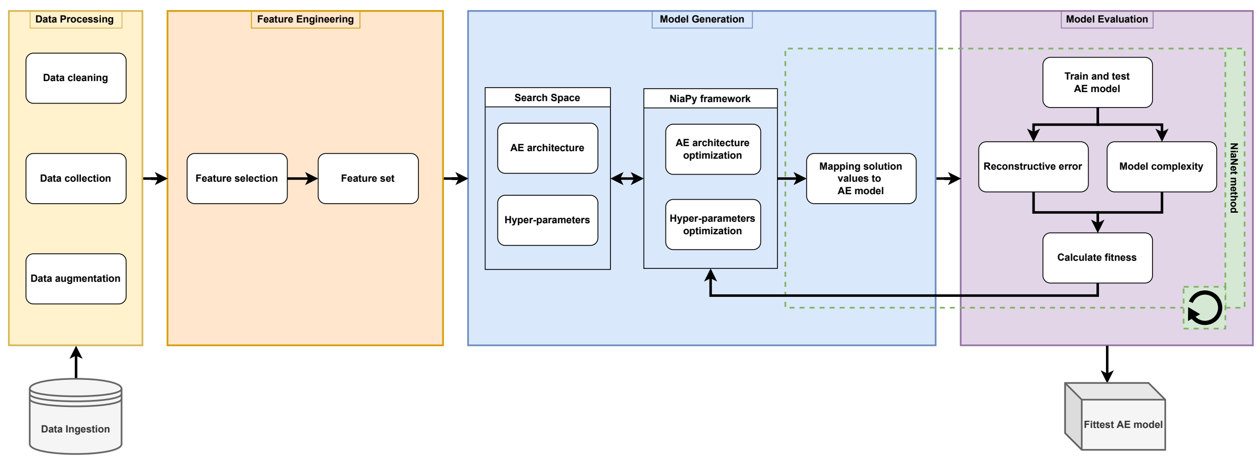

Fig. 1: A high-level overview of our AutoML pipeline, including data preparation (Section 1), feature engineering (Section 2), model generation (Section 3), and model evaluation (Section 4).

elements; each one represents one of the parameters we are attempting to optimize. The first three represent AE topology, and the last four represent hyper-parameters. The solution produced by the optimizer method is then mapped into the AE model using those representations. After the AE model has been built, it is evaluated using a fitness function that measures its performance. The fitness value represents the quality of the discovered solution. The fitness value is then reported back to the optimizer algorithm in the last step, allowing the search for the best optimal solution to continue. The algorithm is presented in Alg. 1.

Algorithm 1 Proposed method

```
Input: Dataset, parameters for NiaNet and NiaPy
Output: The fittest AE model
 1: NiaNet.init()
 2: NiaPy.init()
 3: while terminationConditionNotMet do
 4:     solution ← NiaPy.getBestSolution()
 5:     shape ← NiaNet.mapShape(solution[0])
 6:     layerStep ← NiaNet.mapLayerStep(solution[1])
 7:     layers ← NiaNet.mapLayers(solution[2])
 8:     activation ← NiaNet.mapActivation(solution[3])
 9:     epochs ← NiaNet.mapEpochs(solution[4])
10:     LR ← NiaNet.mapLearningRate(solution[5])
11:     optimizer ← NiaNet.mapOptimizer(solution[6])
12:     fitness ← NiaNet.ModelEvaluation()
13:     NiaPy.generateNewSolution(fitness)
14: end while
15: fittestModel ← NiaNet.model(NiaPy.getBestSolution())
16: return fittestModel
```

A. Representation of individuals

Individuals in NiaNet are presented as real-valued vectors:

$$\chi_i^{(j)} = \left\{x_{i,0}^{(j)}, \dots, x_{i,n}^{(j)}\right\}, \text{for } i = 0, \dots, \text{Np - 1} \tag{1}$$

where each element of the solution is in the interval $\chi_{i,1}^{(j)} \in [0, 1]$. Real values in interval are then mapped according to equations [2-8], where y_1 stands for topology shape, y_2 for number of neurons per layer, y_3 for number of layers, y_4 for activation function, y_5 for number of epochs, y_6 for learning rate, y_7 for optimizer algorithm.

$$y_1 = \lfloor x[i] \rfloor; y_1 \in [0, 1] \tag{2}$$

$$y_2 = \lfloor \frac{x[i]}{features} \rfloor; y_2 \in [0, features] \tag{3}$$

$$y_3 = \lfloor \frac{x[i]}{maxLayers} \rfloor; y_{33} \in [0, maxLayers] \tag{4}$$

$$y_4 = \lfloor x[i] \rfloor; y_4 \in [0, 1] \tag{5}$$

$$y_5 = \lfloor x[i] * 10 + 100 \rfloor; y_5 \in [100, 200] \tag{6}$$

$$y_6 = \lfloor \frac{x[i]}{1000} \rfloor; y_6 \in [10^{-3}, 10^{-0}] \tag{7}$$

$$y_7 = \lfloor x[i] \rfloor; y_7 \in [0, 1] \tag{8}$$

The solution array is separated into two groups of indices, as shown in Fig. 2 . The first three indices are used for topology mapping, while the fourth is utilized for hyper-parameter mapping. Together they form a complete solution that is subsequently used to build an AE model, as demonstrated in

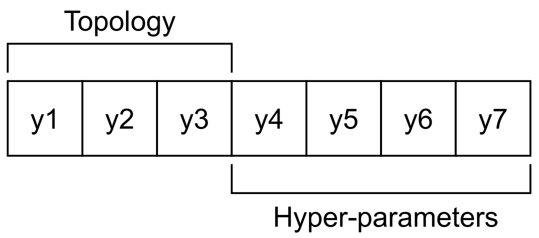

Fig. 2: An illustration of how indices in a solution array are allocated to variables [y_1-y_7].

the algorithm 1. Each of these solutions is retrieved from the NiaPy library during the iterations of the NiaNet algorithm, with the goal of finding the most optimal solution. The fitness function determines the most optimal solution on a given dataset.

Topology representation: As we can see in Fig. 2 the first three elements in solution array are used for constructing the AE topology based on an element value. The first element y_1 is used to determinate the topology shape, which can be symmetrical or asymmetrical. This defines the shape relationship between the encoder and decoder parts. The number of neurons per layer is calculated using the second element y_2. This is dependent on the number of features in dataset. Once we have the number of neurons y_2 we can calculate the number of layers y_3 in AE model. In the case of an asymmetrical AE model, the element value y_3 is used to set a random number of encoder layers before setting the remaining decoder layer number.

Hyper-parameters representation: Second part of solution array as seen in Fig. 2 is used for determining the hyper-parameters values which are utilized throughout the model training. The fourth y_4 and seventh y_7 elements are used for determining the activation function and optimizer algorithm based on a list of possible values. The fifth y_5 and sixth y_6 elements are used for number of epochs and learning rate depending on a defined range.

B. Encoding strategy

Once the real-valued solution array is proposed by the NiaPy framework, the real values are then mapped into the AE model according to the equations [2-8]. The element mapping value of solution array is determined with the binning process for all variables [y_1-y_7]. The bins are created in interval $\in [0, 1]$. Where each bin represents the possible mapping value. For example when mapping the AE's shape y_1, the encoding algorithm takes the element's real value from solution array and map it to corresponding bin (symmetrical or asymmetrical). This can be seen in equation 9.

$$y_1 = \begin{cases} symmetrical & \text{if } x[i] \leq 0.5 \\ asymmetrical & \text{otherwise} \end{cases} \tag{9}$$

C. Fitness function

We defined the fitness function, which measures the individual solution by calculating its reconstruction error and DNN complexity using equations 10 and 11. This enables us to analyze the encoding effectiveness and complexity of its topology.

$$E = (\sum_{i=1}^{D}(x_i - \hat{x_i})^2 * 1000) \tag{10}$$

$$C = \frac{(y_5)^2 + (y_3 * 100) + (bottleneck\_dim * 10)}{100} \tag{11}$$

$$f(\chi_i^{(j)}) = \min E + C \tag{12}$$

Where E represents the reconstruction error, C topology complexity and $f(\chi_i^{(j)})$ represents the fitness value of an individual in evolution. Since equation 12 is designed to seek the global minimum, the fittest individual will be the one with the lowest fitness value. Furthermore, with the variable C, we address the issue of over-fitting. Less complex models are less likely to over-fit during the training process [26].

D. Training conditions

Due to research limitations, some parameters of the constructed AE model were static during the training phase. One of them is batch size, which is always set to 1, and another is the activation function, which remains the same once selected, across all encoder and decoder layers.

E. Data conditions

The data structure that the NiaNet method can process is limited to tabular data with only numerical values. As a result, any other type of data must be transformed first. We plan to expand our research in the near future to include time-series data as well.

V. Experiments and results

A. Introduction to dataset

In our experiments, we used a dataset that includes physiological data about the patients which are identified to have diabetes. The dataset is publicly available for everyone [27]. For each of the 442 diabetic patients, ten baseline characteristics, including age, gender, BMI, average blood pressure, and six blood serum measures, as well as the response of interest, a quantitative measure of disease progression one year after baseline, were collected. The eleventh feature is measuring the diabetes level. The dataset's feature values are represented solely by numerical values, with no missing values and a weak correlation (mean is 0.2). We standardized the data for each feature before using it, so that the distribution has a mean of 0 and a standard deviation of 1. This enables the DNN to be generated later with weights that are more similar across the features, resulting in more uniform topologies. Furthermore, the dataset was divided into the training and testing subset in a ratio 3:1.

B. Enviromental setup

The environment must be properly set up before running our AutoML framework with proposed method NiaNet [28]. In this section, we list the software components and configuration parameters that are utilized in a configuration file. All experiments were carried out using the Python programming language together with the libraries: NiaPy [25] for nature-inspired algorithms, Scikit-learn [29] for evaluating DNN models with metrics, NumPy [30] for working with arrays, PyTorch for DNN initialization [31]. Following computational resources were used for development and training environment: Razer Blade 15 Advanced (Early 2021 model - RZ09-036) with Intel i7-10875H CPU, Nvidia GeForce RTX 3080 with 8 GB GDDR6 memory and 6144 CUDA cores GPU, and 32 GB DDR4 RAM. Running on Windows 11 / Ubuntu 20.04.2 LTS.

C. Experimental settings

We utilized the values in Table I for NiaPy algorithm initialization settings.With parameters supplied, specified methods are utilized to search for optimal values in the encoded solution, which is expressed in equations [2-8]. The algorithms used in our experiment are: PSO [32], Differential Evolution (DE) [33], Firefly Algorithm (FA) [34], Self-adaptive Differential Evolution (jDE) [35], GA [36].

TABLE I: Used parameter values for NiaPy algorithms.

Parameter	Value
Dimensionality problem	7
Population size	default
Max evaluations	100
Runs	2
Lower bound	0.0
Upper bound	1.0

The following is an explanation for selected parameter values in table I: *Dimensionality problem* is set to 7 based on the solution array length, *Population size* is set to default, allowing NiaPy algorithms to have their own default value, *Max evaluations* is specifying number of evaluations on each algorithm separately, *Runs* is specifying number of repetitions (low number due to limited computational resources), *Lower bound* and *Upper bound* are borders within the real value number that can be represented in solution array.

Next we set the list of available activation functions and optimizers, which can be selected based on the mapping rules of y_4 and y_7 variables. Table II shows the activation functions, whereas table III shows the optimizers used in our experiment. All of the listed activation functions and optimizers, are available in PyTorch library, therefore any other ones that are not in the list can be easily added or removed.

TABLE II: List of activation functions in NiaNet method.

Activation function name
ELU
RELU
Leaky RELU
RRELU
SELU
CELU
GELU
Tanh

TABLE III: List of optimizers in NiaNet method.

Optimizer name
Adam
Adagrad
SGD
RAdam
ASGD
Rprop

D. Fittest Autoencoder architecture

We present the fittest AE model in this section, which was built and trained using the proposed NiaNet method in the AutoML framework. The experiment was carried out with the diabetes dataset with the above experimental parameter values. Produced solutions by the NiaNet method were evaluated by the fitness function in equation 12, where PSO produced the fittest solution.

Topology: The proposed solution array was mapped into the AE model based on the encoding strategy. The proposed model was a symmetrical AE, having a single-layer encoder and decoder. The encoder was built to take a 10-D input vector and compress it into an 8-D latent vector. The decoder was a mirrored encoder that took an 8-D latent vector as input and decompressed it back to a 10-D output vector. This indicates that the initial 10-D vector was compressed by the 20%. Another good indication can be seen in a simplicity of a model in terms of AE deepness (see. Fig. 3).

Hyper-parameters: When mapping up the elements in solution, we got the hyper-parameters for the previously mentioned topology. Where an activation function = $RRELU$, number of epochs = 110, learning rate = 0.11 and optimization algorithm = $RAdam$.

Achieved performance: On the testing dataset, we applied the root mean squared error (RMSE) to objectively quantify the fittest AE model's performance. It allows us to examine how close the reconstructed data examples (output) are from the ground truth (input) on average. The closest the result of RMSE metric is to zero, the smaller the difference between input and output. In our case, the fittest AE model reached the value of 0.11, which can be the starting point for future research.

E. Results by optimization algorithm

The following are the findings of our more in-depth analysis. Each of the fittest solutions produced by algorithms PSO, DE, FA, jDE, and GA is listed in table IV. The PSO algorithm

Algorithm	Fitness value	RMSE	Bottleneck size	Topology shape	number of neurons per layer	Layers in AE	Activation function	Epochs	Learning rate	Optimizer algorithm
PSO	**231**	**0.11**	8	symmetrical	En[8], De[10]	**2**	RRELU	**110**	0.11	RAdam
FA	353	0.15	8	symmetrical	En[8], De[10]	2	SELU	140	0.26	Adagrad
jDE	523	0.40	**5**	symmetrical	En[5], De[10]	2	RRELU	110	0.38	Adagrad
GA	553	0.40	9	symmetrical	En[9], De[10]	2	TANH	120	0.04	ASGD
DE	556	0.26	9	symmetrical	En[9], De[10]	2	CELU	170	0.06	RAdam

TABLE IV: The NiaNet fittest solutions found by selected algorithms

achieved a significantly higher fitness value, closely followed by the FA algorithm. Whereas all other algorithms ended up with very comparable fitness values, despite arriving at different solutions in AE model building blocks. This can be explained by looking at the formulation of our fitness function, equation 12. When looking at the proposed bottleneck sizes, we see the values span from 5 to 9. Since the input shape is 10 for the selected dataset, this is relatively considerable variability between algorithms. While the variables such as topology shape, epochs, and number of layers are nearly identical. All the algorithms found the optimal solution in those variables for this problem. The number of neurons in each layer varies between algorithms, since it is related to the size of the bottleneck. Multiple solutions for activation function, learning rate and optimizers were also proposed.

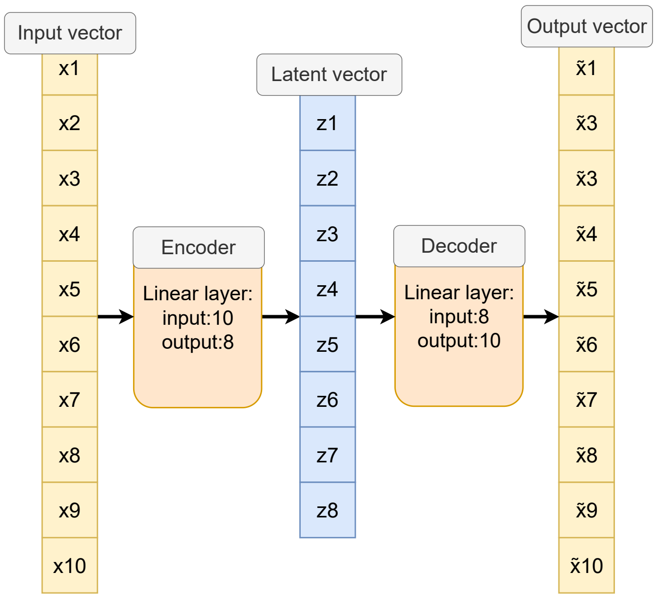

Fig. 3: Fittest autoencoder topology, designed by PSO algorithm.

VI. Conclusion

This paper presented NiaNet, a novel method for building AE models based on the AutoML methodology. The method sets up the topology and the hyper-parameters based on the solutions designed by nature-inspired algorithms. The results gathered in experiments are indicating a promising avenue that has to be further explored. This could help reduce the human resources needed in the model generation stage of AutoML.

Based on these exciting findings, we plan to expand our research toward finding an optimal solution for a broader range of training parameters and AE topologies with various depth, width, and layer types. The objective for the future is to compare our proposed method to existing AutoML methodologies in a more extensive performance comparison using a variety of datasets. Having numerous solutions for various datasets could provide us with insights into how to build optimal AE models in the future.

References

[1] F. Yu, Z. Qin, C. Liu, D. Wang, and X. Chen, "REIN the RobuTS: Robust DNN-Based Image Recognition in Autonomous Driving Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 6, pp. 1258–1271, Jun. 2021, conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[2] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Ł. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *arXiv:1609.08144 [cs]*, Oct. 2016, arXiv: 1609.08144. [Online]. Available: http://arxiv.org/abs/1609.08144

[3] S. Shekhar, A. Singh, and A. K. Gupta, "A Deep Neural Network (DNN) Approach for Recommendation Systems," in *Advances in Computational Intelligence and Communication Technology*, ser. Lecture Notes in Networks and Systems, X.-Z. Gao, S. Tiwari, M. C. Trivedi, P. K. Singh, and K. K. Mishra, Eds. Singapore: Springer, 2022, pp. 385–396.

[4] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis, "Highly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, no. 7873, pp. 583–589, Aug. 2021, number: 7873 Publisher: Nature Publishing Group. [Online]. Available: https://www.nature.com/articles/s41586-021-03819-2

[5] Z. Li, M. Pan, T. Zhang, and X. Li, "Testing DNN-based Autonomous Driving Systems under Critical Environmental Conditions," in *Proceedings of the 38th International Conference on Machine Learning*. PMLR, Jul. 2021, pp. 6471–6482, iSSN: 2640-3498. [Online]. Available: https://proceedings.mlr.press/v139/li21r.html

[6] J. N. K. Liu, Y. Hu, Y. He, P. W. Chan, and L. Lai, "Deep Neural Network Modeling for Big Data Weather Forecasting," in *Information Granularity, Big Data, and Computational Intelligence*, ser. Studies in Big Data, W. Pedrycz and S.-M. Chen, Eds. Cham: Springer International Publishing, 2015, pp. 389–408. [Online]. Available: https://doi.org/10.1007/978-3-319-08254-7_19

[7] P. Dhar, "The carbon impact of artificial intelligence," *Nature Machine Intelligence*, vol. 2, no. 8, pp. 423–425, 2020.

[8] E.-G. Talbi, "Automated Design of Deep Neural Networks: A Survey and Unified Taxonomy," *ACM Computing Surveys*, vol. 54, no. 2, pp. 34:1–34:37, Mar. 2021. [Online]. Available: https://doi.org/10.1145/3439730

[9] G. Vrbančič, I. Fister jr, and V. Podgorelec, *Designing Deep Neural Network Topologies with Population-Based Metaheuristics*, Sep. 2018.

[10] L. Pečnik and I. Fister, "NiaAML: AutoML framework based on stochastic population-based nature-inspired algorithms," *Journal of Open Source Software*, vol. 6, no. 61, p. 2949, May 2021. [Online]. Available: https://joss.theoj.org/papers/10.21105/joss.02949

[11] V. K. Ojha, A. Abraham, and V. Snášel, "Metaheuristic design of feedforward neural networks: A review of two decades of research," *Engineering Applications of Artificial Intelligence*, vol. 60, pp. 97–116, Apr. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197617300234

[12] R. Miikkulainen, "Neuroevolution."

[13] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," vol. 10, no. 2, pp. 99–127. [Online]. Available: https://direct.mit.edu/evco/article/10/2/99-127/1123

[14] A. Conradie, R. Miikkulainen, and C. Aldrich, "Intelligent process control utilising symbiotic memetic neuro-evolution," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, vol. 1, pp. 623–628 vol.1.

[15] A. Hara, J.-i. Kushida, K. Kitao, and T. Takahama, "Neuroevolution by particle swarm optimization with adaptive input selection for controlling platform-game agent," in *2013 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 2504–2509, ISSN: 1062-922X.

[16] E. Galván and P. Mooney, "Neuroevolution in Deep Neural Networks: Current Trends and Future Challenges," *IEEE Transactions on Artificial Intelligence*, vol. 2, no. 6, pp. 476–493, Dec. 2021, conference Name: IEEE Transactions on Artificial Intelligence.

[17] C. Broni-Bediako, "Automated Deep Neural Networks with Gene Expression Programming of Cellular Encoding - Towards the Applications in Remote Sensing Image Understanding-," Mar. 2022. [Online]. Available: https://soka.repo.nii.ac.jp/index.php?active_action=repository_view_main_item_detail&page_id=13&block_id=68&item_id=40743&item_no=1

[18] T. Elsken, J. H. Metzen, and F. Hutter, "Neural Architecture Search: A Survey," *arXiv:1808.05377 [cs, stat]*, Apr. 2019, arXiv: 1808.05377. [Online]. Available: http://arxiv.org/abs/1808.05377

[19] X. Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1423–1447, Sep. 1999, conference Name: Proceedings of the IEEE.

[20] E. Thomas, M. Jan Hendrik, and H. Frank, "Neural Architecture Search: A Survey," *arXiv:1808.05377 [cs, stat]*, Apr. 2019, arXiv: 1808.05377. [Online]. Available: http://arxiv.org/abs/1808.05377

[21] M. Scanagatta, A. Salmerón, and F. Stella, "A survey on Bayesian network structure learning from data," *Progress in Artificial Intelligence*, vol. 8, no. 4, pp. 425–439, Dec. 2019. [Online]. Available: https://doi.org/10.1007/s13748-019-00194-y

[22] X. He, K. Zhao, and X. Chu, "AutoML: A survey of the state-of-the-art," *Knowledge-Based Systems*, vol. 212, p. 106622, Jan. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705120307516

[23] B. Evans, "Population-based Ensemble Learning with Tree Structures for Classification," thesis, Open Access Te Herenga Waka-Victoria University of Wellington, Jan. 2019. [Online]. Available: https://openaccess.wgtn.ac.nz/articles/thesis/Population-based_Ensemble_Learning_with_Tree_Structures_for_Classification/17136296/1

[24] J. Meehan, N. Tatbul, C. Aslantas, and S. Zdonik, "Data ingestion for the connected world," p. 11.

[25] G. Vrbančič, L. Brezočnik, U. Mlakar, D. Fister, and I. Fister, "NiaPy: Python microframework for building nature-inspired algorithms," *Journal of Open Source Software*, vol. 3, no. 23, p. 613, Mar. 2018. [Online]. Available: https://joss.theoj.org/papers/10.21105/joss.00613

[26] C. M. Bishop, *Neural Networks for Pattern Recognition*. USA: Oxford University Press, Inc., 1995, p. 332.

[27] Diabetes data. [Online]. Available: https://www4.stat.ncsu.edu/~boos/var.select/diabetes.html

[28] "NiaNet/autoencoder.py at 408b7fe0f4634439eb69e75f6b0c5afb18ce0702 · SasoPavlic/NiaNet." [Online]. Available: https://github.com/SasoPavlic/NiaNet

[29] "scikit-learn: machine learning in Python — scikit-learn 1.0.2 documentation." [Online]. Available: https://scikit-learn.org/stable/

[30] "NumPy." [Online]. Available: https://numpy.org/

[31] PyTorch. [Online]. Available: https://www.pytorch.org

[32] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4. IEEE, 1995, pp. 1942–1948.

[33] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *Journal of global optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[34] X.-S. Yang, *Nature-inspired metaheuristic algorithms*. Luniver press, 2010.

[35] J. Brest, S. Greiner, B. Boskovic, M. Mernik, and V. Zumer, "Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems," *IEEE transactions on evolutionary computation*, vol. 10, no. 6, pp. 646–657, 2006.

[36] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.