UTILIZING METAHEURISTICS TO GUIDE THE TRAINING OF NEURAL NETWORKS

LUKA MATIJEVIĆ[1]

[1] Mathematical Institute of the Serbian Academy of Sciences and Arts, luka@mi.sanu.ac.rs

Abstract: *Neural networks (NN), have become increasingly popular due to their practical applications. NN training is a crucial stage in constructing a reliable model that can accurately predict data. The goal of NN training is to determine the best internal parameters to optimize the network's performance on test data, according to a specific metric. In this study, we explore the use of metaheuristics to guide the entire training process. Our approach involves identifying favorable areas of the search space and invoking an optimizer to intensify the search in these regions. To train NN, we implemented two metaheuristics, Variable Neighborhood Search (VNS) and the Memetic algorithm (MA), and measured their effectiveness using classification accuracy as an evaluation metric on publicly available classification datasets. The obtained results suggest that MA is able to outperform both VNS and traditional training methods.*

Keywords: *Machine Learning, Combinatorial Optimization, Classification Accuracy, Variable Neighborhood Search, Memetic Algorithm*

1. INTRODUCTION

Neural networks (**NN**) [2, 6] are a well-known type of machine learning algorithm, inspired by the structure of the human brain. They consist of nodes (*neurons*), which process and distribute data across the network. Neurons are organized into layers (Figure 1), each layer taking the output of the previous layer as its input, processing it, and propagating the result to the next layer. Given their capacity for learning from new data and adapting to it, neural networks are particularly advantageous for tasks like audio and picture recognition, natural language processing, and prediction modeling. They have revolutionized fields such as computer vision, robotics, and autonomous vehicles, and are increasingly being applied to a wide range of industries and applications [1]. However, training neural networks can be computationally demanding due to the substantial amount of data needed for effective training.

In Figure 2, we show a generic representation of a neuron. Variables x_i represent input data, w_i are weights, b is a bias, whereas Y denotes the output of a neuron. The goal of the training process is to find values for w_i and b for each of the neurons in the network so that the overall performance of the network is maximized regarding some chosen metric.

The goal of this paper is to assess the possibility of guiding the training process by utilizing metaheuristics, for the purpose of obtaining a better accuracy of the network. More precisely, we will take a look at two different metaheuristics: *Variable Neighborhood Search* (**VNS**) and *Memetic algorithm* (**MA**). Although it is possible to train NN by directly applying some metaheuristics capable of global optimization to determine the optimal set of parameters (for example, a VNS-based global optimization method [4]), this study takes a different approach.

The structure of this paper is as follows. The conventional approach for NN training is presented in Section 2, while Section 3 elaborates on our VNS algorithm for guiding the NN training process. Our MA approach is introduced in Section 4. Section 5 details our experimental

setup and the results obtained. Section 6 concludes the paper by discussing some of the issues associated with this approach and outlining possible areas for future research.

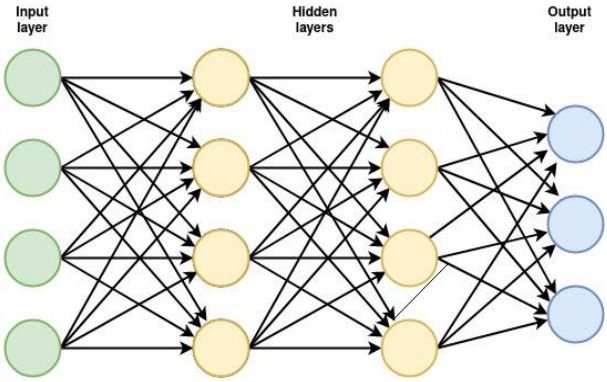

Figure 1: A generic representation of a neural network

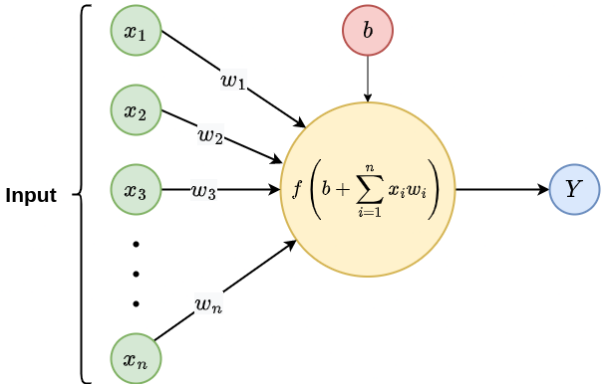

Figure 2: A generic representation of a neuron

2. NEURAL NETWORK TRAINING

Training neural networks involves feeding input data through the network and adjusting the weights and biases of the neurons in order to minimize the difference between the predicted output and the actual output (i.e. the *loss function*). A wide range of loss functions are available [12], and choosing the right loss function depends on the specific nature of the problem at hand. The optimization process typically involves the use of an *optimizer* algorithm, which adjusts the weights and biases of the network in small increments based on the gradient of the loss function. There are many different optimizers to choose from [3], most notably Adam, Gradient Descent, Stochastic Gradient Descent, Adamax, and many others. During training, the network is usually evaluated on a validation set (also known as testing set), allowing us to monitor its performance and prevent overfitting to the training data. Once training is complete, the network can be used to make predictions on new, unseen data. In Algorithm 1 we present a generic method for training neural networks. At first, we define a neural network by specifying layers of nodes and *activation functions* [11], and select an optimizer. We then create a loader, which is a component of the software pipeline responsible for loading and preprocessing data into a format that can be used by the neural network. Loaders are often used when dealing with large datasets, where it is not feasible to load all of the data into memory at once. Instead, the loader loads a batch of data into memory at a time, allowing the neural network to train on that batch before loading in the next one. This approach is known as minibatch training and is commonly used in deep learning. The core of the algorithm (Lines 6-11) involves training the model by using an optimizer and a loss function to update the neural network parameters. This process, which we refer to as *"Local Search"* in this paper, is also used as a subroutine in VNS and MA. Local Search is conducted over a fixed number of epochs, which refers to a single pass through the training dataset.

Algorithm 1: General procedure for training a neural network

```
procedure TRAIN_NN(train_data, test_data, num_epochs)
    model ← defineNeuralNetwork(...)
    optimizer ← selectOptimizer(...)
    loader ← createLoader(train_data)
    for i ← 1 to num_epochs do
        for (features, targets) ∈ loader do
            output ← model(features)
            loss ← lossFunction(outputs, targets)
            gradients ← calculateGradients(loss)
            updateWeightsAndBiases(gradients)
        accuracy ← evaluate(model, test_data)
    return model
```

3. VARIABLE NEIGHBORHOOD SEARCH

Variable Neighborhood Search was first introduced by Mladenović and Hansen (1997) [9]. It comprises of three main steps: *shaking*, local *search*, and *move or not* procedure. The solution is represented as an array of values, where each value corresponds to a specific weight or bias. We obtain the initial solution by taking the initial values of each variable from our defined model. Our shaking procedure is presented in Algorithm 2. It basically chooses one variable at random in each step and then adds or removes the value of ϵ to/from it. The value of ϵ is a predetermined parameter of the algorithm. In general, the value of ϵ can be dynamically determined to better suit each solution component, although this specific aspect was not examined in the scope of this research. As a local search procedure, we call an optimizer with the perturbed solution as its starting point, providing it with the training data. After the optimizer completes its execution, we evaluate the obtained solution using the testing data. If the newly found solution is better than the incumbent solution, we accept it as the new incumbent solution. The main loop of the VNS algorithm follows the standard procedure, as described in Algorithm 3 presented by Matijević et al. [8]. In this basic version, the variables k_{min} and k_{max} represent the minimum and maximum values, respectively, of a variable called k. This k variable governs the size of the neighborhood at each iteration of the algorithm. More details about the VNS method can be found in a publication by Hansen et al. [5].

Algorithm 2: Shaking procedure

```
procedure SHAKE(k, ϵ, solution)
    i ← 1
    for i ≤ k do
        v ← chooseRandomVariable(solution)
        p ← uniformDistribution(0,1)
        if p < 0.5 then
            solution[v] ← solution[v] + ϵ
        else
            solution[v] ← solution[v] − ϵ
        i ← i + 1
    return solution
```

4. MEMETIC ALGORITHM

A memetic algorithm is a type of evolutionary algorithm that combines principles of genetic algorithms with local search methods. It was first proposed by Moscato (1989) [10]. In MA, a population of candidate solutions undergoes a series of operations similar to those found in genetic algorithms, such as mutation and crossover, to produce new candidate solutions. However, in memetic algorithms, the local search is applied to the individuals in the population to improve their fitness before they are selected for breeding. This approach is utilized to better exploit the knowledge of the search space by taking advantage of the existing structures and patterns in the population. The use of local search also helps to prevent premature convergence, where the algorithm becomes stuck in a suboptimal solution.

In Algorithm 3, we present our own MA for guiding NN training. Likewise to VNS, solutions are encoded as arrays of values representing weights and biases. At first, the NN is defined (Line 2), and the initial population is created (Line 3). Generating the initial population is performed by taking the initial values of the variables in the defined model, and for each individual the shaking procedure is performed (Algorithm 2). The neighborhood size for each individual is determined at random, which combined with the stochastic nature of the shaking procedure guarantees the diversity of the population. After the initial population is generated, each individual is evaluated according to some evaluation metric (Line 4). Lines 5-12 represent the main part of the algorithm. In each iteration, we find the best individual (according to the chosen metric) in our current population and improve it by applying an NN optimizer to it, as a form of local search (Line 7). It must be borne in mind that we only apply optimizer to the best individual in the population, not to all of them. This is done to improve performance by avoiding the optimization of non-promising solutions. Although we have not tested it, it could be beneficial to apply an optimizer to the n-best solutions instead of just one. Once the best individual has been improved by an optimizer, a predefined number of offspring is generated through the application of the CROSSOVER operator. The number of offspring produced is determined by the parameter *offspring_num*. Our approach employs a crossover operator that functions as follows: for each offspring, two individuals are randomly chosen and referred to as *parents*. Subsequently, for each variable in the solution, a value is randomly selected with an equal probability from one of the parents. After the offspring are generated, we subject each one to the MUTATION operator (Line 10), which entails applying the SHAKE procedure (Algorithm 2) with a probability determined by the parameter m, after which the offspring is added to the population (Line 11). Additionally, the parameter n_{max} defines the maximum neighborhood size that can be utilized during the shaking procedure. Finally, the worst individuals are removed from the population (Line 12).

Algorithm 3: Memetic algorithm

1: **procedure** MA($train\_data, test\_data, \epsilon, m, pop\_size, offspring\_num, n_{max}$)
2: $model \leftarrow defineModel(\dots)$
3: $population \leftarrow INIT(pop\_size, model, \epsilon)$
4: $evaluate(population, test\_data)$
5: **while** stopping criterion is not met **do**
6: $best \leftarrow findBestIndividual(population)$
7: $best \leftarrow localSearch(best, train\_data)$
8: $offspring \leftarrow CROSSOVER(offspring\_num, population)$
9: **for** $\forall o \in offspring$ **do**
10: $o \leftarrow MUTATION(o, m, n_{max}, \epsilon)$
11: $population \leftarrow population \cup o$
12: $removeWorstIndividuals(offspring\_num, population)$
13: **return** $findBestIndividual(population)$

5. EXPERIMENTAL EVALUATION

To perform experimental evaluation, we selected five benchmark datasets for the classification problem, publicly accessible at 'https://www.kaggle.com/datasets'. Each dataset was divided into a training and testing set of instances, with a 60:40 ratio. All methods were implemented using the Python programming language and the PyTorch framework for machine learning. For all three algorithms, we employed the *Adam optimizer* [7] and utilized the *Cross-Entropy Loss* function to optimize the neural network parameters during training. The experimental tests were conducted on a personal laptop featuring an Intel i7-10750H CPU and 32GB of RAM, operating on Ubuntu 20.04 OS. To showcase the consistency and reliability of the obtained results, we conducted the experiments 30 times, ensuring their stability.

To achieve the optimal results, it is crucial to meticulously choose the hyperparameters for our algorithms. In our study, we employed the iRace[1] package for the R programming language to statistically identify an optimal set of hyperparameters. With a budget of 5000 tests, the iRace package generated the following set of values: the learning rate and weight decay hyperparameters for the Adam optimizer were both set to 0.01, the ϵvalue used in the shaking procedure was set to 0.4884, while for VNS, we set k_{min} and k_{max} to 5 and 20 respectively. Finally, for MA, we set the mutation probability (m) to 0.1, number of offspring to 6, and the population size to 10.

As a stopping criterion for all three methods, we imposed a limit on the number of calls to the local search procedure, which we fixed at 200. While this limit may seem low for certain real-world applications, our primary objective was to showcase the advantages of our approach and encourage more attention from the research community. Additionally, with larger neural networks and datasets, even 200 calls can consume a considerable amount of computational resources.

Table 1 displays the results of our experiments. The first two columns provide details about the datasets used, including the name and number of instances. The third column presents the average accuracy achieved over 30 runs using the traditional neural network training method (as outlined in Algorithm 1), along with the standard deviation in parentheses. The final two columns show the results obtained using our proposed metaheuristics. As shown in the presented table, VNS outperformed the traditional method in four out of five cases, though the improvement was not significant in most cases. Conversely, MA exhibited superior performance in all the tested instances, surpassing both the traditional method and VNS.

Table 1: The obtained results

Dataset	Num. inst.	Traditional	VNS	MA
Iris	148	0.9744 (0.0102)	0.8363 (0.1529)	**0.9904 (0.013)**
Star classification	100000	0.7076 (0.0446)	0.7149 (0.0432)	**0.8142 (0.0492)**
Wine quality	6497	0.5322 (0.0539)	0.5580 (0.0608)	**0.6249 (0.0518)**
Diabetes	253680	0.8438 (0.0651)	0.8517 (0.0668)	**0.8899 (0.0357)**
Driving	6728	0.3878 (0.0675)	0.4778 (0.0811)	**0.5417 (0.1137)**

6. FINAL REMARKS

As demonstrated in the preceding section, MA-guided neural network training yielded models with superior accuracy compared to VNS-guided and traditional training methods. Nevertheless,

[1] https://cran.r-project.org/web/packages/irace/readme/README.html

there are certain problems that warrant more attention. Foremost among them is the fact that metaheuristics, especially MA, utilize significantly more memory, posing increasing challenges with the growth of neural network size. Secondly, determining optimal hyperparameter values presents a formidable challenge of its own. Although it is feasible to employ statistical tools to determine these parameters for smaller networks and datasets, their utility is not always viable when handling vast networks and datasets. Lastly, as metaheuristics are often stochastic in nature (though not universally), their output can rely heavily on the seed value of the random number generator.

This paper aimed at demonstrating the feasibility of metaheuristic-guided neural network training, but further research is necessary to fully explore its potential. Specifically, this approach should be tested on various network architectures and larger datasets to assess its effectiveness and scalability. Moreover, the genetic operators utilized in MA, including crossover, mutation, and selection, are rudimentary, and exploring more advanced operators could potentially yield even better solutions.

ACKNOWLEDGEMENT

This work was supported by the Ministry of Science, Technological Development and Innovation of the Republic of Serbia, Agreement No. 451-03-47/2023-01/200029.

REFERENCES

[1] Abiodun, O.I., Jantan, A., Omolara, A.E., Dada, K.V., Mohamed, N.A. and Arshad, H., 2018. State-of-the-art in artificial neural network applications: A survey. Heliyon, 4(11), p.e00938.

[2] Aggarwal, C.C., 2018. Neural networks and deep learning. Springer, 10(978), p.3.

[3] Bottou, L., Curtis, F.E. and Nocedal, J., 2018. Optimization methods for large-scale machine learning. SIAM review, 60(2), pp.223-311.

[4] Drazić, M., Kovacevic-Vujcić, V., Cangalović, M. and Mladenović, N., 2006. Glob—a new VNS-based software for global optimization. Global optimization: from theory to implementation, pp.135-154.

[5] Hansen, P., Mladenović, N., Brimberg, J. and Pérez, J.A.M., 2019. Variable neighborhood search (pp. 57-97). Springer International Publishing.

[6] Hopfield, J.J., 1982. Neural networks and physical systems with emergent collective computational abilities. Proceedings of the national academy of sciences, 79(8), pp.2554-2558.

[7] Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[8] Matijević, L., Davidović, T., Ilin, V. and Pardalos, P., 2019. General Variable Neighborhood Search for Asymmetric Vehicle Routing Problem.

[9] Mladenović, N. and Hansen, P., 1997. Variable neighborhood search. Computers & operations research, 24(11), pp.1097-1100.

[10] Moscato, P., 1989. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Caltech concurrent computation program, C3P Report, 826(1989), p.37.

[11] Sharma, S., Sharma, S. and Athaiya, A., 2017. Activation functions in neural networks. Towards Data Sci, 6(12), pp.310-316.

[12] Wang, Q., Ma, Y., Zhao, K. and Tian, Y., 2020. A comprehensive survey of loss functions in machine learning. Annals of Data Science, pp.1-26.