Architecture Optimization of Neural Networks using Particle Swarm Optimization

Jason Ravagli[1] and Antonio Acunzo[2]

[1] *jason.ravagli@stud.unifi.it*
[2] *antonio.acunzo@stud.unifi.it*

March 29, 2020

Abstract

The optimization of neural networks hyperparameters and, in particular, of neural networks architecture is a crucial task to get good quality networks. In this report, we present an implementation of a Particle Swarm Optimization algorithm to find the optimal architecture for an MLP neural network and we study its performance compared to two simpler methods: the Grid Search and the Quasi-Random Search.
Keywords: neural networks, Particle Swarm Optimization, Grid Search, Quasi-Random Search, architecture optimization, hyperparameters optimization

1 Introduction

Neural networks have become the main topic in the machine learning field and they are now the state-of-the-art approach for classification and regression problems. Basically, neural networks have a certain number of parameters that can be optimized on the problem data by a gradient descent based algorithm. This is the training process. In general, the optimal parameters found are a sub-optimal solution. In fact, several hyperparameters contribute to determining training and network quality. Examples of hyperparameters are the learning rate, the training batch size, the particular optimization algorithm used in training and also the network architecture.

Hyperparameters optimization can be seen as an optimization problem where the objective function to be minimized is some network performance measure (e.g. the classification error in classification problems) achieved by the trained network. Hyperparameters tuning process iteratively chooses some values for the hyperparameters, train the network on a subset of the available problem data (training set) and evaluate the network performance (the classification error in classification problems) on another subset of the problem data, called validation set. The hyperparameters values that determine the best performance are chosen to train the final network. The more values are evaluated by the tuning process, the more training processes are done, hence the more time-consuming is the entire operation.

The design of the network architecture in terms of number of layers and number of neurons is usually driven by experience and trial-and-error processes and can be very time-consuming. A simple and widely used alternative to manual optimizing the network architecture (and the hyperparameters in general) is the grid search method. Grid search almost removes the experience requirements, but, due to its simplicity, it requires a lot of objective function evaluations (i.e. network training) to find good hyperparameters.

In this report we use the Particle Swarm Optimization (PSO) algorithm, a global optimization algorithm, to find the best architecture for an MLP network used in a binary classification problem. We try to optimize the number of hidden layers in the network and the number of neurons in each layer. We then compare the performance of the PSO approach to the Grid Search and the Quasi-Random Search, two simple yet widely used methods.

The report is organized as follows. In section 2, we present an overview of the three optimization methods considered. In section 3, our particular implementation of the PSO algorithm is provided. Then in section 4, we report the experiments we conducted and the results we took from, and finally, in section 5, we briefly present our conclusions.

2 Optimization Methods

2.1 Quasi-Random Search

Quasi-Random Search is one of the simplest search methods that requires almost no knowledge about the particular problem. It generates low-discrepancy random sequences of points in the hyperparameters space and evaluates the objective function values in such points. The naive Pure-Random Search technique generates points using a pseudo-random number generator, with the effect to have clusters of points in some regions, and other regions poorly covered (Figure 1). On the contrary, Quasi-Random Search generates a low-discrepancy random sequence of points in order to

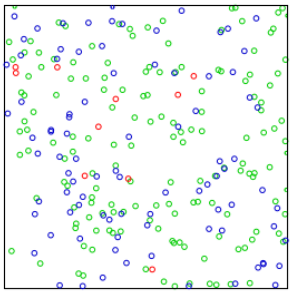


Figure 1: Points created from a pseudo-random generator

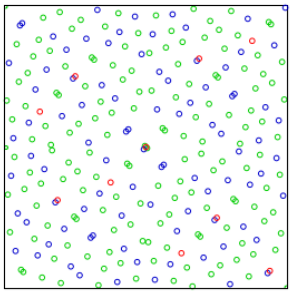


Figure 2: Points belonging to a quasi-random sequence (Sobol sequence)

cover the search space uniformly (Figure 2). In our experiments, we use quasi-random numbers generated from Sobol sequences.

2.2 Grid search

The Grid Search method consists of manually defining the hyperparameters values combination to evaluate. The combinations form a grid in the hyperparameters space to try to cover the entire space uniformly. Unlike Quasi-Random methods, defining the combinations by hand allows assigning more importance to certain hyperparameters evaluating more values for them. Furthermore, it allows concentrating on some areas of the hyperparameters space where it is probably from experience to find good combinations.

Grid search is a very simple method but to find good values for the hyperparameters it requires to evaluate many combinations and therefore it is computationally expensive. To reduce the number of function evaluations the combinations can be defined to cover areas of the search space where it is more likely to find the optimum, but this alternative requires a certain experience with the particular problem and the used network.

2.3 Particle Swarm Optimization

Particle Swarm Optimization is a population-based heuristic global optimization algorithm and it simulates the social behavior of a group of individuals. It was first introduced by Eberhart et al.[1] and it was inspired by the movements of a bird flock, where the position of each bird change according to its previous position and the other birds position.

PSO generates a certain number of particles, assigning them a certain position and velocity. The particle position is a feasible solution to the problem and in our case correspond to a combination of hyperparameters values. Then the algorithm iteratively updates the particles position according to the following formula:

$$\mathbf{V}^i = w \cdot \mathbf{V}^{i-1} + C_1 \cdot R_1 \cdot (\mathbf{P}_{best}^{i-1} - \mathbf{P}^{i-1}) + C_2 \cdot R_2 \cdot (\mathbf{G}_{best}^{i-1} - \mathbf{P}^{i-1}) \quad (1)$$

$$\mathbf{P}^i = \mathbf{P}^{i-1} + \mathbf{V}^i \quad (2)$$

Whereas $\mathbf{V}_i$ and $\mathbf{P}_i$ are the particle velocity and position at the iteration (also called generation) i, and w is the inertia weight of the particle. P_{best} is the best position found by the particle in terms of objective function until the iteration $i-1$, while G_{best} is the best position found by the entire population. The coefficients C_1 and C_2 are the cognitive parameter and the social parameter respectively and denotes how much the best personal position and the best population position influence the particle movements. Finally, R_1 and R_2 are random parameters introducing a stochastic component[2][3].

In the simpler version of the algorithm, the inertia coefficient w has a static value. However, a dynamic value adapts the velocity to the problem during the algorithm execution and helps the particle to do not stuck in local minima at the first iterations. Various methods have been proposed to dynamically update the inertia coefficient, and one of the best and simple approach is to use a linear decreasing inertia weight:

$$w(t) = (w(0) - w(n_t)) \cdot \frac{(n_t - t)}{n_t} + w(n_t) \quad (3)$$

where n_t is the maximum number of iterations of the algorithm, $w(0)$ is the initial inertia weight, $w(n_t)$ is the final inertia weight, and $w(t)$ is the inertia at iteration t. It is required that $w(0) > w(n_t)$. This approach starts with a large inertia value which decreases over time to smaller values. In doing so, particles are allowed to explore in the initial search steps, while favoring exploitation as iterations increase.[2][3]

As we have seen, PSO it's easy to implement and is quite computationally inexpensive in terms of memory usage and operations to be performed. It can have several different types of termination conditions, but it is common to fix a maximum number of iterations. Furthermore, due to its definition, it cannot guarantee to find the global optimum of the problem (as many other global optimization methods), but it is statistically proven that the algorithm finds a solution not so far from the best one.

PSO applies to unconstrained problems with continuous variables and as it is cannot be applied to the network architecture optimization problem. Layer numbers and neurons are integer numbers, cannot be 0 or negative, and we have to fix maximum values for them in order to reduce the problem complexity (the feasible set is a hypercube).

Concerning the constraint handling, many approaches have been proposed but most of them involve

Algorithm 1 PSO Implementation

Input: $swarm\_size$, $num\_generations$, $constraints$
Output: G_{best}, $best_value$
initialize particles
for $i = 0$; $i < num\_generations$; $i = i + 1$ **do**
  $G^i_{best} = G_{best}$
  $best\_value^i = best\_value$
  for $j = 0$; $j < swarm\_size$; $j = j + 1$ **do**
    update velocity of $particle_j$ using Equation (1)
    update position of $particle_j$ using its updated velocity
    update w coefficient of $particle_j$
    check if velocity respect the defined bound $max\_velocity$
    $position\_integer$ = round each component of the $particle_j$ position to the nearest integer
    if $position\_integer$ is feasible **then**
      $value$ = evaluate the objective function at $position\_integer$
      if $value < best\_value^i$ **then**
        $best\_value^i = value$
        $G^i_{best} = position\_integer$
      end if
    end if
  end for
  $G_{best} = G^i_{best}$
  $best\_value = best\_value^i$
end for

exploring also non-feasible solutions, and this is not reasonable in our case (a layer with negative neurons does not exist). Therefore we decided to use the Death-Penalty approach: if a particle ends up in a non-feasible position, the objective function value of that solution is not evaluated [2][4]. The death-penalty method simply ignores non-feasible solutions and this is reasonable in the network architecture scenario. Moreover, it is unlikely to have particle exceedings the boundaries of the constraints due to Equation 1 (particles are attracted from the personal and global best solutions) and to the fact that the feasible space is a hypercube.

Integer variables are obtained by simply rounding to the nearest integer value the particle position components before training the network and evaluating its accuracy.

3 Implementation

Given their simplicity, we omit the implementation for the Grid Search and the Quasi-Random Search and we focus on the PSO one. Details about the configuration used for the first two methods are described in Section 5.

The pseudocode for the implemented version of the Particle Swarm Optimization algorithm is shown in Algorithm 1.

The particles positions are initialized in two different ways: n particles (where n is the problem dimension, the number of hyperparameters to be tuned, that is 2 in our case) are generated at the corners of the constraints hypercube. The remaining particles ($swarm\_size - n$) are initialized with a quasi-random method using a Sobol sequence. The quasi-random initialization distributes the particles in the search space uniformly, favoring the exploration of as much space as possible during early iterations.

The initial velocity of each particle is set to zero, according to [2]. A non-zero initial velocity is not necessary and it must be handled with care (e.g. it must be defined some properly maximum and minimum values for the initial velocity). When all the particles are initialized, we evaluate the associated objective function values: with these, we initialize the personal best position and the global best position of the population.

After this phase the main algorithm is executed: we iterate through all particles and we update their velocities and positions. For each new position the objective function is evaluated (i.e. the neural network is created with the corresponding number of layers and neurons, trained and its performance evaluated) and the personal and global positions are updated. Constraints and integer variables are handled as described in Section 2.3.

Concerning the formula for the velocity update, its parameters are set as follows:

- $C_1 = C_2 = 0.5$

  Choosing the same value for C_1 and C_2 the particles are equally influenced by the personal best and the global positions, and it is always a good choice. On the contrary, finding good distinct values for the coefficients is strongly problem dependent and it might be hard. Furthermore, a low value such as 0.5 allows particles to roam far from good regions to explore before being pulled back towards good regions[2].

- w is updated according to [2], with $w(0) = 0.9$ and $w(n_t) = 0.4$ [4].

- R_1 and R_2 are randomly generated in [0, 1].

The algorithm terminates after *num_generations* iterations, allowing to control the number of function evaluations.

3.1 Variants

Besides the main implementation described above, we also implemented two little variants of the PSO algorithm.

In our experiments, we observed that with only the quasi-random initialization the particles never explored the boundary regions (e.g. combinations with only one layer, which produce good network performance, were never evaluated), attracted from the results of the already visited internal positions. Therefore, we implemented a variant where n (problem dimension) particles are initialized at the corners of the feasible region hypercube. With this version of the algorithm, we observed that particles evaluated also boundary positions.

About the second variant, we implemented the memetic version of the PSO algorithm. After updating each particle position, we round it to the nearest integer position and from there we start a local search. We calculate the objective function value in each of the $2 * n$ integer positions adjacent to the current one. If at least one position is better than the current one we move to the best one and we repeat the evaluation procedure, otherwise we stop: the current position is a local minimum. We update the particle current position with the local minimum found by the local search only if the latter is better than the particle personal best position. The memetic variant is much more expensive than the base one but it is supposed to provide better results.

4 Dataset

We considered the Diabetic Rethinopaty Debrecen dataset [5] for our experiments. It contains features extracted from the Messidor image set to predict whether an image contains signs of diabetic retinopathy or not.

The dataset has 1151 instances, each one having 19 features representing either a detected lesion, a descriptive feature of an anatomical part or an image-level descriptor. Each instance also has a binary label indicating whether it is associated with diabetic retinopathy.

In order to achieve good classification performance, we had to do some preprocessing on the dataset. The first feature of the instances represents the quality of the image from which the data were extracted, where 0 indicates bad quality and 1 good quality. We detected only 4 instances having this feature with a 0 value. Since data from bad quality images can be considered unreliable and there were very few instances associated with this value, making the feature poor in information, we removed these 4 instances and the feature from the dataset. Furthermore, some features have a very large range of values, so we normalized the values of the instances.

Doing these two operations we obtained a significant improvement in terms of classification performance and stability of the results.

5 Experiments and results

In our experiments, we randomly split the dataset into the training set (80% of the entire dataset) and the test set (20%) to have different training sets and test sets at each instance of the experiment. We used the K-fold cross-validation method to train the network and evaluate its average classification accuracy on the validation set. The average classification error ($1 - classification\_accuracy$) was our objective function to minimize.

Concerning the training, we used the binary cross-entropy as loss function and the Adam algorithm with a learning rate equal to 0.00158, which is proved in [5] and confirmed by our experiments to work better for this dataset. We also used an early stopping method to prevent overfitting.

Using the Quasi-Random Search we generated and evaluated only 10 combinations of layers-neurons. For the grid search, we manually provided 130 combinations. Finally, for the PSO versions, we used a swarm size and a maximum number of iterations both equal to 10, hence the algorithm evaluates 110 combinations in total in the base and the border intialization variants.

The results are shown in Figure 3 and the presented accuracy values (and for the memetic PSO version also the number of function evaluations) are the mean values obtained from 20 runs of each method.

We can observe that the PSO memetic variant obtains the best performance with 75.29%, as expected, slightly better than the base variant (75.1%). In general, the PSO variants surpassed the Grid Search and the Quasi Random method.

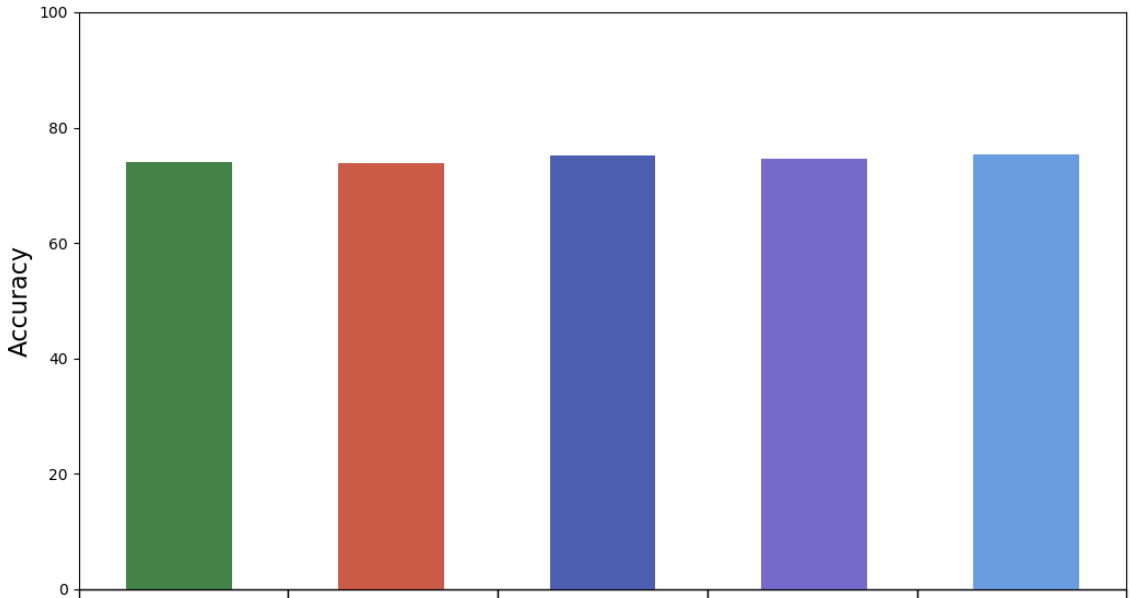
However, considering the number of function evaluations performed by the memetic variant (825) we expected a stronger improvement from it. This becomes much more relevant if we consider the Quasi-Random method, which achieves 74.05% of accuracy with only 10 evaluations.

6 Conclusions

For our work, we took inspirations from some publications where the PSO algorithm was successfully used to optimize the architecture and, in general, the integer hyperparameters of a neural network.

However, our results were not so good as expected. In particular, we cannot observe a significant difference between the methods considered and between the PSO algorithm variants implemented in terms of classification accuracy. Furthermore, with only 15 experiments the collected sample of accuracies could not be reliable with so little differences between the various methods.

Maybe the defined optimization problem is too simple (we constrained the neural network to have the same neurons number in each layer), and the associated objective function does not have sharped minima. In this case, a good optimization algorithm cannot out-



	Quasi-Random	Grid Search	PSO - Base	PSO - Init. on Borders	PSO - Local Search
Accuracy (%)	74.05	73.83	75.1	74.57	75.29
N. Evaluations	10	250	110	110	825

Figure 3: Accuracies and number of function evaluations performed by each considered method

perform other simpler methods.

Another hypothesis is that the PSO algorithm, which is proven to work well for continuous optimization problems, does not work well with integer variables and a very simple method like the Quasi-Random Search can achieve about the same accuracy in a very shorter time.

With more computational power and time, we could evaluate all the hyperparameters combinations within the defined problem constraints to find the minimum value of the objective function and compare the associated accuracy with the results obtained in our experiments.

References

[1] J. Kennedy and R. Eberhart, Particle swarm optimization, *Proceedings of ICNN95 - International Conference on Neural Networks* (1995).

[2] A. P. ENGELBRECHT, *COMPUTATIONAL INTELLIGENCE: an introduction* (JOHN WILEY, 2007).

[3] J. C. Bansal, P. K. Singh, M. Saraswat, A. Verma, S. S. Jadon, and A. Abraham, Inertia weight strategies in particle swarm optimization, *2011 Third World Congress on Nature and Biologically Inspired Computing* (2011).

[4] A. R. Jordehi, A review on constraint handling strategies in particle swarm optimisation, *Neural Computing and Applications* **26**, 1265–1275 (2015).

[5] B. Antal and A. Hajdu, An ensemble-based system for automatic screening of diabetic retinopathy, *Knowledge-Based Systems* **60**, 20–27 (2014).