Multilayer Perceptron Architecture Design using Genetic Algorithm

Ilia Grigorev
Innopolis University, Beijing Institute of Technology
Computer Science and Technology School
il.grigorev@innopolis.university

I. Introduction

Multilayer Perceptron (MLP), or Dense Neural Network (DNN), is a generalization concept of the classical Machine Learning (ML) approaches. The key idea is to assemble multiple regressors together in a net with the result of the weighted sum of each component. For more flexibility, non-linearity is introduced via the concept of activation functions.

The problem arises with deciding which architecture of the MLP to take for a specific problem. In particular, the following parameters are the most relevant: the number of hidden layers, the activation functions of each layer, and the number of neurons on each layer.

In this paper, I provide a genetic algorithm approach for hyperparameter tuning. The genetic algorithm is part of evolutionary algorithms that quickly converge to a suboptimal solution. Although the implemented search produces less promising results compared to the grid search [1], the method is still powerful and needs to be further tuned to achieve better performance.

II. Method

A. MLP

To design, train and test the performance of DNNs, I used the `Keras` framework based on the Python programming language. The framework provides different methods for assembling layers of an MLP: sequential (using `keras.Sequential()`, functional (using `keras.layers` one after another), and class (inheritance from `keras.Model` interfaces.

B. Genetic Algorithm

The genetic algorithm is a nature-inspired optimization algorithm that has the following stages: selection, crossover, and mutation. Other extensions such as elitism are possible. The first key point of the algorithm is to determine the representation of a solution as a chromosome that is the vector or string of values. The second is to define the fitness function to compare chromosomes.

1) Representation: In this research, I decided to follow the classical approach and defined the chromosome as `DNN` class with the list of hidden layers and output layer.

2) Fitness: The fitness score determines how close a solution to the desired one (optimum). I used the accuracy score on the test dataset as the fitness function for the algorithm.

3) Selection: During selection, individuals are chosen for further actions. Selection mainly depends on the fitness of the individual. In this project, I used the tournament selection, where two individuals are chosen from the population proportionally to their fitness, and the fittest from them remains.

4) Crossover: Crossover is the variadic operator that combines chromosomes of several individuals. The combination used in the project is the following: each layer of the first individual may be replaced by a randomly selected layer from the second individual with 50% chance. The result of a combination is another individual, refered as *child*.

5) Mutation: Mutation is another variadic operator that changes the state of a chromosome. In my case, the activation functions are changed in layers with some probability `_PM`.

6) Pipeline: The following algorithm is used to perform steps from above:

```
population := initiate population of size N
M := number of generations
for i in 1..M do
   evaluate fitness of population
   apply crossover to create N/2 children
   apply mutation on children
   evaluate children
   save best individual information
   population := select N individuals from
       current population and children
endfor
```

III. Dataset

To evaluate the performance of models, MNIST [2] dataset was used. The dataset consists of a large number of handwritten digits that need to be classified to 10 classes. To load the dataset, I used `keras` framework. The features were normalized, and the labels were transformed from categorical type to binary categories.

A. Source code

The following notebook contains the source code of the genetic algorithm pipeline for the hyperparameter tuning.

IV. Results

A. Performance and Best Architecture

The proposed pipeline is time-consuming due to large number of calculations. The possible limitations are the following: the dataset consists of many records, the calculations

were performed on the CPU, and the high-level approach programming without significant performance optimizations was used.

The best result after running the first generations are present in the table (Table I). The individual achieved around 97.1% accuracy on the test set.

TABLE I
LAYERS DESCRIPTION

Layer Size	Activation Function
45	relu
44	leaky relu
59	leaky relu

B. Model interfaces

In this section, the obtained model architecture is implemented using three `keras` approaches.

1) `keras.Sequential()`: (code snippet)

```
import keras
import keras.layers as layers

input_shape = (784, )
model_sequential = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Dense(45, activation='relu'),
        layers.Dense(44,
            activation='leaky_relu'),
        layers.Dense(59,
            activation='leaky_relu'),
        layers.Dense(10, activation='softmax')
    ]
)
model_sequential.summary()
```

2) Functional interface: (code snippet)

```
import keras
import keras.layers as layers

input_shape = (784, )

inputs = keras.Input(shape=input_shape)
x = layers.Dense(45,
    activation='relu')(inputs)
x = layers.Dense(44,
    activation='leaky_relu')(x)
x = layers.Dense(59,
    activation='leaky_relu')(x)
outputs =layers.Dense(10,
    activation='softmax')(x)
model_func = keras.Model(inputs=inputs,
    outputs=outputs)
model_func.summary()
```

C. Class interface

(code snippet)

```
import keras
import keras.layers as layers

class DNN(keras.Model):
    def __init__(self, input_dim=784):
        super().__init__()
        self.dense1 = layers.Dense(45,
            activation='relu',
            input_shape=(input_dim,))
        self.dense2 = layers.Dense(44,
            activation='leaky_relu')
        self.dense3 = layers.Dense(59,
            activation='leaky_relu')
        self.dense4 = layers.Dense(10,
            activation='softmax')

    def call(self, inputs):
        x = self.dense1(inputs)
        x = self.dense2(x)
        x = self.dense3(x)
        x = self.dense4(x)
        return x

model_class = DNN()
model_class.summary()
```

V. DISCUSSION

The proposed approach has significant limitations that includes lack of performance optimizations and rigorous analysis of variadic operations. Further research should study different crossover and mutation method to achieve faster convergence and better results. Moreover, optimizations need to be introduced to increase usability of the method.

CONCLUSION

Overall, the proposed method demonstrated promising results that could be achieved by elaborating more rigorous approaches of the genetic algorithms.

REFERENCES

[1] M. Ogunsanya et al., "Grid search hyperparameter tuning in additive manufacturing processes," *Manuf. Lett.*, Volume 35, Supplement, 2023, pp. 1031-1042, ISSN 2213-8463

[2] Y. LeCun et al., "Gradient-based learning applied to document recognition," *Proc. IEEE*, 86, pp. 2278–2324, 1998