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 sub-class (inheritance from `keras.Model` patterns.

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 adjust the classical approach and defined the chromosome as `DNN` class with the list of hidden layers and the output layer.

2) Fitness: The fitness score determines how close a solution is 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, referred as a *child*.

5) Mutation: Mutation is another variadic operator that changes the state of a chromosome. In my case, four types of mutations are performed: changing activation function for each layer with probability `_PACT`, removing a layer if their number is greater than or equal to `upper_bound` with probability `_PREM`, adding a layer if their number is less than or equal `lower_bound` with probability `_PADD`, and adding a dropout layer if there are at least `lower_bound` number of layers with probability `_PDROP`.

6) Pipeline: The following algorithm is used to perform the steps 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
```

C. Dataset

To evaluate the performance of models, Fashion-MNIST [2] and iris [3] datasets were used. The first dataset consists of numerous images of clothes that need to be classified to 10 classes. To load this dataset, I used `keras` framework. The features were normalized, and the labels were transformed from categorical type to binary categories. The second dataset consists of information about three types iris flowers with four features.

D. *Source code*

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

III. Results

A. *Parameters setup*

The proposed model itself requires some parameters to set up. In Table 1, Table 2, and Table 3, values used for the experiment are presented.

TABLE I
Search space for GA

Parameter name	Value
Hidden layers number range	[1, 8]
Activation functions	['relu', 'sigmoid', 'tanh', 'softplus', 'leaky_relu', 'linear']
Layer size range	[8, 128]

TABLE II
Optimizer configuration

Parameter name	Value
batch size	128
epochs	20
validation split	0.1
loss	'categorical_crossentropy'
optimizer	'adam'
metrics	['accuracy']

TABLE III
Parameters of GA

Parameter name	Value
Population size	10
Number of generations	10
_PACT	0.2
_PADD	0.6
_PREM	0.6
_PDROP	0.25
lower_bound	2
upper_bound	5

B. *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.

Table 4 reveals the logs of finding the best parameters of the Fashion-MNIST model using the proposed genetic algorithm.

The architecture of best individual for the Fashion-MNIST classification after the search is present in Table 5. The individual achieved around 89.08% accuracy on the test set. The bootstrap confidence interval for the test accuracy is $(0.8849, 0.8908)$.

TABLE IV
Fashion-MNIST Results of Generations

Generation	Avg	Min	Max
Generation 1	0.8697	0.8586	0.8860
Generation 2	0.8791	0.8625	0.8860
Generation 3	0.8842	0.8798	0.8860
Generation 4	0.8854	0.8834	0.8860
Generation 5	0.8864	0.8856	**0.8886**
Generation 6	0.8867	0.8856	0.8886
Generation 7	0.8872	0.8858	0.8886
Generation 8	0.8697	0.8586	0.8860
Generation 9	0.8877	0.8860	0.8886
Generation 10	0.8887	0.8817	**0.8908**

TABLE V
Fashion-MNIST Layers Description

Layer Size	Activation Function
127	tanh
127	relu
127	relu
44	leaky_relu

Likewise, Table 6 and Table 7 demonstrate the statistics of different generations and the best model for the iris dataset classification, respectively.

TABLE VI
IRIS RESULTS OF GENERATIONS

Generation 1		
Avg	**Min**	**Max**
0.7956	0.6222	0.9778
Generation 2		
Avg	**Min**	**Max**
0.9089	0.6889	0.9778
Generation 3		
Avg	**Min**	**Max**
0.9533	0.9111	0.9778
Generation 4		
Avg	**Min**	**Max**
0.9444	0.7333	0.9778
Generation 5		
Avg	**Min**	**Max**
0.9644	0.8667	0.9778
Generation 6		
Avg	**Min**	**Max**
0.9667	0.8889	0.9778
Generation 7		
Avg	**Min**	**Max**
0.9733	0.9333	0.9778
Generation 8		
Avg	**Min**	**Max**
0.9800	0.9778	**1.0**
Generation 9		
Avg	**Min**	**Max**
0.9578	0.7556	1.0
Generation 10		
Avg	**Min**	**Max**
0.9622	0.7778	1.0

TABLE VII
IRIS LAYERS DESCRIPTION

Layer Size	**Activation Function**
28	linear
69	tanh
85	linear
85	dropout(0.19)
69	tanh
69	dropout(0.19)
69	linear

C. *Model patterns*

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

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

```
import keras
import keras.layers as layers

input_shape = (784, )
model_sequential = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Dense(127, activation=’tanh’),
        layers.Dense(127, activation=’relu’),
        layers.Dense(127, activation=’relu’),
        layers.Dense(44,
            activation=’leaky_relu’),
        layers.Dense(num_classes,
            activation=’softmax’)
    ]
)
model_sequential.summary()
```

2) Functional Pattern: (code snippet)

```
import keras
import keras.layers as layers

input_shape = (784, )

inputs = keras.Input(shape=input_shape)
x = layers.Dense(127,
    activation=’tanh’)(inputs)
x = layers.Dense(127, activation=’relu’)(x)
x = layers.Dense(127, activation=’relu’)(x)
x = layers.Dense(44,
    activation=’leaky_relu’)(x)
outputs =layers.Dense(10,
    activation=’softmax’)(x)
model_func = keras.Model(inputs=inputs,
    outputs=outputs)

model_func.summary()
```

D. *Sub-class Pattern*

(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(127,
          activation=’tanh’,
          input_shape=(input_dim,))
      self.dense2 = layers.Dense(127,
          activation=’relu’)
      self.dense3 = layers.Dense(127,
          activation=’relu’)
      self.dense4 = layers.Dense(44,
          activation=’leaky_relu’)
      self.dense5 = layers.Dense(10,
          activation=’softmax’)

   def build(self, input_shape):
      self.dense1.build(input_shape)
      input_shape =
          self.dense1.compute_output_shape(input_shape)

      self.dense2.build(input_shape)
      input_shape =
          self.dense2.compute_output_shape(input_shape)

      self.dense3.build(input_shape)
      input_shape =
          self.dense3.compute_output_shape(input_shape)

      self.dense4.build(input_shape)
      input_shape =
          self.dense4.compute_output_shape(input_shape)

      self.dense5.build(input_shape)
      input_shape =
          self.dense5.compute_output_shape(input_shape)
```

```
        self.built = True

    def call(self, inputs):
        x = self.dense1(inputs)
        x = self.dense2(x)
        x = self.dense3(x)
        x = self.dense4(x)
        x = self.dense5(x)
        return x

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

IV. Discussion

The proposed GA approach demonstrated promising results by finding a suboptimal but quite accurate solution in a specified search space. Compared to the results in [4], the performance of the resulted MLP for Fashion-MNIST dataset is slightly better $(1\% - 2\%)$.

For both datasets, the MLP architecture was comprised of layers with mostly `'relu'`, `'leaky_relu'`, `'tanh'`, and `'linear'` activation functions. The lack of `'softplus'`, and `'sigmoid'` functions suggests that these activations are less relevant for those problems. Additionally, the presence of `'linear'` function in the iris dataset problem, suggests that the dependency can be described linearly.

The proposed approach has significant limitations that include 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 enhanced further 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] H. Xiao et al., ”Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms,” *CoRR*, v. abs/1708.07747, 2017

[3] R. A. Fisher, ”The use of multiple measurements in taxonomic problems,” *Ann. Eugenics*, 1936

[4] A. S. Henrique et al., ”Classifying Garments from Fashion-MNIST Dataset Through CNNs,” *Advances Sci., Technol. Eng. Syst. J.*, v. 6, no. 1, pp. 989-994, 2021