Introduction: Hyperparameter Optimization of Generic Deep Autoencoders for Time Series using a Genetic Algorithm

Author: Anika Terbuch

History: \change{1.0}{11-Apr-2023}{Original}

Source: Chair of Automation, University of Leoben, Austria

email: automation@unileoben.ac.at *url:* automation.unileoben.ac.at

Table of Contents

Abstract

This script introduces the use and parameterization of the genetic algorithm for the optimization of the hyperparameters autoencoders of the class `AutoencoderDeep`.

Introduction

This toolbox provides a framework for optimizing the hyperparameters of autoencoders of the class AutoencoderDeep.

If you are not familiar with the toolbox "Generic Deep Autoencoder for Time-Series" consider getting familiar with it before using this toolbox. The supplementary material of the autoencoder toolbox provides a more in-depth description of each of the hyperparameters which can be optimized using this toolbox.

Encodings

Each of the optimized hyperparameters is encoded as an integer into the individuals.

Implemented Genetic Operations

Evaluation:

The first step of the genetic algorithm in each of the generations is that the individuals are evaluated to determine their fitness.

The fitness function is based on training the ML model for which HPO is done with different hyperparameters represented by the different individuals.

It is known that the performance of machine learning models varies from run to run. To get a better estimate of the performance of the model 3-fold cross-validation is performed. In this way, three autoencoders are trained on different folds of the data for hyperparameter optimization and as fitness, the mean value of the three runs is used. The fitness is based on the reconstruction error on one of the test sets for each of the 3 folds. For a better generalization and a more reliable estimation of the model performance, this approach was chosen. Furthermore, if an individual, a set of hyperparameters, survives over multiple generations it gets retrained in each of the generations and also this gives a more realistic evaluation of the fitness of a certain individual. In each of the generations, a different permutation of the data is used to perform the 3-fold cross-validation.

Selection strategy

The individuals who have at least the median fitness or better are selected to form the mating pool. Additionally, with the percentage specified in `RandomProbability` individuals who are not fit enough are added to the mating pool to enhance genetic diversity.

Crossover

The next genetic operator applied to the population is crossover.

Two crossover strategies are combined, and each time when the crossover operator is applied one of the two strategies is selected with the same probability.

Random Crossover: For each of the genes the entry of one of the parents is selected.

Mean Crossover: For each of the genes, the value of the current gene is the mean of the parents' genes.

The combination of two crossover functions was chosen due to the relatively low number that can be trained with the computational budget available in comparison with the size of the search space. The mean crossover produces new values for the genes and therefore leads to a better exploration of the search space.

Mutation

With the probability specified in `MutationProbability` the newly created individual is mutated. During mutation the value of a randomly selected gene is set to a random value on the specified domain of the hyperparameter.

Replacement

The generation size; number of individuals per generation, is contant. Therefore crossover is applied as many times as there are individuals in a generation. After this the whole old generation is replaced by the new generation.

Stopping criteria/ termination condition

The termination condition consists of three terms, if one of them is met the algorithm is terminated:

1. the number of maximal generations is met,
2. there is no improvement in the average fitness for four generations,
3. all individuals hold the same hyperparameters.

Description of the class HPOsettingsAED

The class HPOsettingsAED is a class containing all the hyperparameters of instances of the class AutoencoderDeep.

To create an instance of the hyperparameter optimization we need to instantiate an object of this class. This is done by calling the constructor of the class:

```
t=HPOsettingsAED()
```

This object contains 3 structs of parameters:

```
t =

  HPOsettingsAED with properties:

              settingsGA: [1×1 struct]
            settingsAED: [1×1 struct]
    optimizableVariables: [1×1 struct]
```

In the following section these, three structs will be described in more detail.

settingsGA

This struct defines the settings of the genetic algorithm.

The struct contains the following fields:

```
          MaxGenerations: 10
          PopulationSize: 20
      MutationProbability: 0.0500
        RandomProbability: 0.0500
```

- **MaxGeneration**: Maximal number of generations the genetic algorithm is performed if it does not converge earlier.
- **PopulationSize**: The number of individuals per generation. This number is constant over all generations
- **MutationProbability**: Defines the probability that an individual is mutated.
- **RandomProbability**: Defines the probability that an individual with insufficient fitness according to the selection rule is added to the mating pool.

settingsAED

This struct defines the architecture of the autoencoder which should be optimized.

The goal is that the result of the optimization is in a second step used to train an autoencoder with the resulting architecture from these settings and the optimized hyperparameters. The default values are:

```
     LatentDimension: 2
ExecutionEnvironment: 'auto'
     AutoencoderType: 'VAE'
       LayersEncoder: {'FC'  'LSTM'}
       LayersDecoder: {'Bi-LSTM'}
```

- **LatentDimension** [positiveInteger]: Dimension of the latent space.
- **ExecutionEnvironment** ['cpu' or 'gpu or 'auto']: Defines where the training of the neural network should take place. If 'auto' is chosen, then it is checked if a GPU is available for training.
- **AutoencoderType** 'AE' or 'VAE': When this parameter is set to *AE* an Autoencoder is created, when specifying it as *VAE* a variational autoencoder is optimized.
- **LayersEncoder** {'FC', 'LSTM', 'Bi-LSTM'}**:** The following three types of layers are available: FC - fully connected layer; LSTM - Long short-term memory layer and Bi-LSTM - Bi-directional long-short term memory layers. These layers can be stacked to form a deeper architecture. The layers are defined as a cell array, i.e. the cell-array for having 2 layers, a fully-connected layer followed by an LSTM layer is defined by the cell array: layers={'FC','LSTM'}.
- **LayersDecoder** {'FC', 'LSTM', 'Bi-LSTM'}: The same layers are available to form the decoder. The autoencoder does not need to be symmetric - other layers and a different number of layers can be selected in the decoder as in the encoder.

During the optimization objects of the class AutoencoderDeep are created and trained.

More information on these settings and in general on autoencoders can be found in the supplementary documents of the toolbox Generic Deep Autoencoders for Time-Series.

optimizableVariables:

This struct defines the ranges of the hyperparameter optimization for each of the optimized hyperparameters. For each of the optimizable variables a lower and an upper bound needs to be defined. For each layer defined in `LayersEncoder` and `LayersDecoder` the optimization range for the number of neurons in these layers needs to be defined.

The default settings are:

```
    BetaKLDivergence: [0 100]
        LearningRate: [1.0000e-05 1.0000e-03]
       MiniBatchSize: [2 100]
NeuronsDecoderLayer1: [1 30]
NeuronsEncoderLayer1: [1 20]
NeuronsEncoderLayer2: [1 10]
        NumberEpochs: [1 200]
```

- **BetaKLDivervence** [positiveInteger]: This hyperparameter defines the weighting parameter for the Kullback-Leibler divergence, it is only relevant when implementing a variational autoencoder. If this parameter is set to a value greater than 1, a so-called β-VAE is created. The β is used as a Lagrangian multiplier in the cost function. More information on β-VAE can be found for example in https://openreview.net/forum?id=Sy2fzU9gl
- **LearningRate** [between0and1]: The learning rate is a value between 0 and 1. It is used as the start value for the learning rate of the gradient descent algorithm to update the learnable parameters (weights and biases of the network). As a gradient-based optimizer adaptive moment estimation (ADAM) is used. More information can be found in: https://de.mathworks.com/help/deeplearning/ref/adamupdate.html
- **MiniBatchSize** [positiveInteger]: This parameter specifies the mini-batch size used during training. The max. permitted batch size is 2/3 * numberOptimizationExamples. This is the case because 3-fold cross-validation is performed and 2/3 of the available data is used to train one autoencoder instance.
- **NumberEpochs** [positiveInteger]: Defines the number of epochs performed during training.

More information on these settings and in general on autoencoders can be found in the supplementary documents of the toolbox Generic Deep Autoencoders for Time-Series.

Customizable hyperparameter optimization options:

Each of the above presented structs is customizable to tailor the optimization to the architecture and problem on hand.

Function setSettingsAED

This function can be used to customize the values of the struct settingsAED. To change the values pass **name-value pairs** to the function, i.e. an autoencoder with a fully-connected layer in the encoder and an LSTM layer followed by a Bi-LSTM layer in the decoder can be set by addressing the object of the class `HPOsettingsAED` called `t` instantiated above:

```
t.setSettingsAED('AutoencoderType','AE','LayersEncoder',{'FC'},'LayersDecoder',{'LSTM','Bi-LSTM'})
```

The adjusted settings of the object `t` can be displayed by calling:

```
t.settingsAED
```

and the output looks as follows:

```
struct with fields:

       LatentDimension: 2
  ExecutionEnvironment: 'auto'
       AutoencoderType: 'AE'
         LayersEncoder: {'FC'}
         LayersDecoder: {'LSTM'  'Bi-LSTM'}
```

Function setRangesOptimization

For each of the optimizable variables a range of values on which the HPO should be performed needs to be defined.

The default values of the optimizable variables can be displayed by calling the struct `optimizableVariables` of the created object `t`

```
t.optimizableVariables
```

The output is self-adjusting based on the architecture defined in the struct `settingsAED`. In our example we defined one layer in the encoder and two layers in the decoder. The corresponding default struct looks as follows:

```
struct with fields:


  LearningRate: [1.0000e-05 1.0000e-03]
  MiniBatchSize: [2 100]
  NeuronsDecoderLayer1: [1 30]
  NeuronsDecoderLayer2: [1 20]
  NeuronsEncoderLayer1: [1 20]
          NumberEpochs: [1 200]
```

We now want to adjust the NeuronsDecoderLayer1 to be optimized in the range between 20 and 200 as well as the number epochs to be between 40 and 80. This can be done by passing name-value pairs to the function setRangesOptimization:

```
t.setRangesOptimization('NeuronsDecoderLayer1',[20 200],'NumberEpochs',[40 80])
```

and getting the optimizable variables with the changed ranges:

```
struct with fields:

  LearningRate: [1.0000e-05 1.0000e-03]
  MiniBatchSize: [2 100]
  NeuronsDecoderLayer1: [20 200]
  NeuronsDecoderLayer2: [10 20]
  NeuronsEncoderLayer1: [1 20]
  NumberEpochs: [40 80]
```

Function setValuesGA

To define the parameters of the genetic algorithm the function `setValuesGA` can be used. With this function the struct `settingsGA` can be adjusted.

The current settings of the genetic algorithm can be displayed by:

```
t.settingsGA
```

and the output is:

```
  struct with fields:

        MaxGenerations: 10
        PopulationSize: 20
    MutationProbability: 0.0500
      RandomProbability: 0.0500
```

This function has the signature:

```
setValuesGA(obj,maxGen,populationSize,mutationProbability, randomProbability )
```

whereas the last 2 parameters are optional. I.e. this function can be called with 2,3 or 4 input parameters.

To change the number of max. generations to 15 and the population size to 10 the following call can be done:

```
t.setValuesGA(15,10)
```

and led to the changed settings displayed in the struct `settingsGA`

```
  struct with fields:

        MaxGenerations: 15
        PopulationSize: 10
    MutationProbability: 0.0500
      RandomProbability: 0.0500
```

If also the mutation probability should be increased to 7% the call is changed in the following way:

```
t.setValuesGA(15,10,0.07)
```

and the struct containing the GA settings is changed to:

```
        MaxGenerations: 15
        PopulationSize: 10
    MutationProbability: 0.0700
      RandomProbability: 0.0500
```

Executing the hyperparameter optimization

The hyperparameter optimization can be executed by passing the data the optimization can be performed on called `dataOpti` in our example alongside with the object of the class HPOsettingsAED to the function GA_HPO_AED.

More information on the required data format can be found in the supplementary documents of the class AutoencoderDeep.

When using the object containing the options from above:

```
GA_HPO_AED(dataOpti, t)
```

The output consists of the following variables:

- aed: trained autoencoders of all generations. To evaluate the fitness of a hyperparameter setting 3-fold cross validation is applied. Therefore 3 autoencoders are trained per individual.
- avgFitness: average fitness of all individuals per generation.
- bestFitness: the fitness of the best individual per generation
- bestIndividual: the individual of the best fitness per generation - bestIndividual of last generation represents the winning individual of the optimization
- fitness: fitness of each of the individuals per generation