# *Implement a New Neural Network Regressor*

*The BRAPH 2 Developers*

*August 14, 2024*

This is the developer tutorial for implementing a new neural network regressor. In this tutorial, you will learn how to create the generator file `*.gen.m` for a new neural network regressor, which can then be compiled by `braph2genesis`. All kinds of neural network models are (direct or indirect) extensions of the base element `NNBase`. Here, you will use as example the neural network regressor `NNRegressorMLP` (multi-layer perceptron regressor).

## *Contents*

## *Implementation of a neural network regressor (`NNRegressorMLP`)*

You will start by implementing in detail `NNRegressorMLP`, which is a direct extension of `NNBase`. A multi-layer perceptron regressor `NNRegressorMLP` comprises a multi-layer perceptron regressor model and a given dataset.

Code 1: **NNRegressorMLP element header.** The `header` section of the generator code in `_NNRegressorMLP.gen.m` provides the general information about the `NNRegressorMLP` element.

```
%% iheader!
NNRegressorMLP < NNBase (nn, multi-layer perceptron regressor) comprises a multi-layer perceptron regressor model and a given dataset. 1

%%% idescription!
A neural network multi-layer perceptron regressor (NNRegressorMLP) comprises a multi-layer perceptron regressor model and a given dataset.
NNRegressorMLP trains the multi-layer perceptron regressor with a formatted inputs ("CB", channel and batch) derived from the given dataset.

%%% ibuild!
1
```

1 defines `NNRegressorMLP` as a subclass of `NNBase`. The moniker will be `nn`.

Code 2: **NNRegressorMLP element prop update.** The `props_update` section of the generator code in `_NNRegressorMLP.gen.m` updates the properties of the `NNRegressorMLP` element. This defines the core properties of the data point.

```
%% iprops_update!

%%% iprop!
NAME (constant, string) is the name of the neural network multi-layer perceptron regressor.
%%%% idefault!
'NNRegressorMLP'

%%% iprop!
DESCRIPTION (constant, string) is the description of the neural network multi-layer perceptron regressor.
%%%% idefault!
'A neural network multi-layer perceptron regressor (NNRegressorMLP) comprises a multi-layer perceptron regressor model and a given dataset. NNRegressorMLP trains the multi-layer perceptron regressor with a formatted inputs ("CB", channel and batch) derived from the given dataset.'

%%% iprop!
TEMPLATE (parameter, item) is the template of the neural network multi-layer perceptron regressor.
%%%% isettings!
'NNRegressorMLP'

%%% iprop!
ID (data, string) is a few-letter code for the neural network multi-layer perceptron regressor.
%%%% idefault!
```

```
'NNRegressorMLP ID'

%%% iprop!
LABEL (metadata, string) is an extended label of the neural network multi-layer perceptron regressor.
%%%% idefault!
'NNRegressorMLP label'

%%% iprop!
NOTES (metadata, string) are some specific notes about the neural network multi-layer perceptron regressor.
%%%% idefault!
'NNRegressorMLP notes'

%%% iprop! 1
D (data, item) is the dataset to train the neural network model, and its data point class DP_CLASS defaults to one of the compatible classes within the set of DP_CLASSES.
%%%% isettings!
'NNDataset'
%%%% idefault!
NNDataset('DP_CLASS', 'NNDataPoint_CON_REG')

%%% iprop!
DP_CLASSES (parameter, classlist) is the list of compatible data points.
%%%% idefault! 2
{'NNDataPoint_CON_REG' 'NNDataPoint_CON_FUN_MP_REG' 'NNDataPoint_Graph_REG' 'NNDataPoint_Measure_REG'}

%%% iprop!
INPUTS (query, cell) constructs the data in the CB (channel-batch) format.
%%%% icalculate! 3
% inputs = nn.get('inputs', D) returns a cell array with the
%  inputs for all data points in dataset D.
if isempty(varargin)
    value = {};
    return
end
d = varargin{1};
inputs_group = d.get('INPUTS');
if isempty(inputs_group)
    value = {};
else
    flattened_inputs_group = [];
    for i = 1:1:length(inputs_group)
        inputs_individual = inputs_group{i};
        flattened_inputs_individual = [];
        while ~isempty(inputs_individual)
            currentData = inputs_individual{end};  % Get the last element from the stack
            inputs_individual = inputs_individual(1:end-1);   % Remove the last element

            if iscell(currentData)
                % If it's a cell array, add its contents to the stack
                inputs_individual = [inputs_individual currentData{:}];
            else
                % If it's numeric or other data, append it to the vector
                flattened_inputs_individual = [currentData(:); flattened_inputs_individual];
```

1 defines `NNDataset` which contains the `NNDataPoint` to train this regressor.

2 defines the compatible `NNDataPoint` classes with this `NNRegressorMLP`.

3 is a query that transforms the input data of `NNDataPoint` to the CB (channel-batch) format by flattening its included cells.

```
            end
        end
        flattened_inputs_group = [flattened_inputs_group; flattened_inputs_individual'];
    end
    value = {flattened_inputs_group};
end

%%% ¡prop!
TARGETS (query, cell) constructs the targets in the CB (channel-batch) format.
%%%% ¡calculate! 4
% targets = nn.get('PREDICT', D) returns a cell array with the
%  targets for all data points in dataset D.
if isempty(varargin)
    value = {};
    return
end
d = varargin{1};
targets = d.get('TARGETS');
if isempty(targets)
    value = {};
else
    nn_targets = [];
    for i = 1:1:length(targets)
        target = cell2mat(targets{i});
        nn_targets = [nn_targets; target(:)'];
    end
    value = {nn_targets};
end

%%% ¡prop!
MODEL (result, net) is a trained neural network model.
%%%% ¡calculate! 5
inputs = cell2mat(nn.get('INPUTS', nn.get('D')));  6
targets = cell2mat(nn.get('TARGETS', nn.get('D')));  7
if isempty(inputs) || isempty(targets)
    value = network();
else
    number_features = size(inputs, 2);
    number_targets = size(targets, 2);
    layers = nn.get('LAYERS');8

    nn_architecture = [featureInputLayer(number_features, 'Name', 'Input')];
    for i = 1:1:length(layers)
        nn_architecture = [nn_architecture
            fullyConnectedLayer(layers(i), 'Name', ['Dense_' num2str(i)])
            batchNormalizationLayer('Name', ['BatchNormalization_' num2str(i))])
            dropoutLayer('Name', ['Dropout_' num2str(i)])
            ];
    end
    nn_architecture = [nn_architecture
        reluLayer('Name', 'Relu_output')
        fullyConnectedLayer(number_targets, 'Name', 'Dense_output')
        regressionLayer('Name', 'Output')
        ];

    % specify trianing options  9
    options = trainingOptions( ...
```

4 is a query that collects all the target values from all data points.

5 trains the regressor.

6 and 7 extract the inputs and targets.

8 defines the neural network architecture with user-specified number of neurons and number of layers.

9 defines the neural network training options.

```
        nn.get('SOLVER'), ...
        'MiniBatchSize', nn.get('BATCH'), ...
        'MaxEpochs', nn.get('EPOCHS'), ...
        'Shuffle', nn.get('SHUFFLE'), ...
        'Plots', nn.get('PLOT_TRAINING'), ...
        'Verbose', nn.get('VERBOSE') ...
        );

    % train the neural network 10
    value = trainNetwork(inputs, targets, nn_architecture, options);
end
```

10 trains the model with those parameters and the neural network architecture.

Code 3: **NNRegressorMLP element props.** The `props` section of generator code in `_NNRegressorMLP.gen.m` defines the properties to be used in `NNRegressorMLP`.

```
%% iprops!

%%% iprop! 1
LAYERS (data, rvector) defines the number of layers and their neurons.
%%%% idefault!
[32 32]
%%%% igui!
pr = PanelPropRVectorSmart('EL', nn, 'PROP', NNRegressorMLP.LAYERS, ...
    'MIN', 0, 'MAX', 2000, ...
    'DEFAULT', NNRegressorMLP.getPropDefault('LAYERS'), ...
    varargin{:});

%%% iprop!
WAITBAR (gui, logical) detemines whether to show the waitbar.
%%%% idefault!
true

%%% iprop!
INTERRUPTIBLE (gui, scalar) sets whether the comparison computation is
     interruptible for multitasking.
%%%% idefault!
.001

%%% iprop! 2
FEATURE_IMPORTANCE (query, cell) evaluates the average significance of each
     feature by iteratively shuffling its values P times and measuring the
     resulting average decrease in model performance.
%%%% icalculate!
% fi = nn.get('FEATURE_IMPORTANCE', D) retrieves a cell array containing
%  the feature importance values for the trained model, as assessed by
%  evaluating it on the input dataset D.
if isempty(varargin)
    value = {};
    return
end
d = varargin{1};
P = varargin{2};
seeds = varargin{3};

inputs = cell2mat(nn.get('INPUTS', d));
if isempty(inputs)
    value = {};
    return
```

1 defines the number of neuron per layer. For example, `[32 32]` represents two layers, each containing 32 neurons.

2 is a query that calculates the permuation feature importance. Note that, other neural network architectures, such as convolutional neural network, have other techniques to obtain feature importance.

```
end
targets = cell2mat(nn.get('TARGETS', d));
net = nn.get('MODEL');

number_features = size(inputs, 2);
original_loss = crossentropy(net.predict(inputs), targets);

wb = braph2waitbar(nn.get('WAITBAR'), 0, ['Feature importance permutation ...']);

start = tic;
for i = 1:1:P 4
    rng(seeds(i), 'twister')
    parfor j = 1:1:number_features 5
        scrambled_inputs = inputs;
        permuted_value = squeeze(normrnd(mean(inputs(:, j)), std(inputs(:, j)), squeeze(size(inputs(:, j))))) + squeeze(randn(size(inputs(:, j)))) + mean(inputs(:, j));
        scrambled_inputs(:, j) = permuted_value;
        scrambled_loss = crossentropy(net.predict(scrambled_inputs), targets);
        feature_importance(j) = scrambled_loss;
    end

    feature_importance_all_permutations{i} = feature_importance / original_loss;

    braph2waitbar(wb, i / P, ['Feature importance permutation ' num2str(i) ' of ' num2str(P) ' - ' int2str(toc(start)) '.' int2str(mod(toc(start), 1) * 10) 's ...'])
    if nn.get('VERBOSE')
        disp(['** PERMUTATION FEATURE IMPORTANCE - sampling #' int2str(i) '/' int2str(P) ' - ' int2str(toc(start)) '.' int2str(mod(toc(start), 1) * 10) 's'])
    end
    if nn.get('INTERRUPTIBLE')
        pause(nn.get('INTERRUPTIBLE'))
    end
end

braph2waitbar(wb, 'close')

value = feature_importance_all_permutations;
```

4 and 5 iteratively shuffle the feature values from any given dataset P times and measuring the resulting average decrease in model performance.

Code 4: **NNRegressprMLP element tests.** The `tests` section from the element generator `_NNRegressprMLP.gen.m`. A test for creating example files should be prepared to test the properties of the data point. Furthermore, an additional test should be prepared for validating the value of input and target for the data point.

```
%% ¡tests!

%%% ¡test!
%%%% ¡name!
train the regressor with example data
%%%% ¡code!

% ensure the example data is generated
if ~isfile([fileparts(which('NNDataPoint_CON_REG')) filesep 'Example data NN
      REG CON XLS' filesep 'atlas.xlsx'])
    test_NNDataPoint_CON_REG % create example files
end

% Load BrainAtlas
im_ba = ImporterBrainAtlasXLS( ...
    'FILE', [fileparts(which('NNDataPoint_CON_REG')) filesep 'Example data
     NN REG CON XLS' filesep 'atlas.xlsx'], ...
    'WAITBAR', true ...
    );

ba = im_ba.get('BA');

% Load Groups of SubjectCON
im_gr = ImporterGroupSubjectCON_XLS( ...
    'DIRECTORY', [fileparts(which('NNDataPoint_CON_REG')) filesep 'Example
     data NN REG CON XLS' filesep 'CON_Group_XLS'], ...
    'BA', ba, ...
    'WAITBAR', true ...
    );

gr = im_gr.get('GR');

% create a item list of NNDataPoint_CON_REG
it_list = cellfun(@(x) NNDataPoint_CON_REG( ...
    'ID', x.get('ID'), ...
    'SUB', x, ...
    'TARGET_IDS', x.get('VOI_DICT').get('KEYS')), ...
    gr.get('SUB_DICT').get('IT_LIST'), ...
    'UniformOutput', false);

% create a NNDataPoint_CON_REG DICT
dp_list = IndexedDictionary(...
        'IT_CLASS', 'NNDataPoint_CON_REG', ...
        'IT_LIST', it_list ...
        );

% create a NNData containing the NNDataPoint_CON_REG DICT
d = NNDataset( ...
    'DP_CLASS', 'NNDataPoint_CON_REG', ...
    'DP_DICT', dp_list ...
    );

nn = NNRegressorMLP('D', d, 'LAYERS', [20 20]);
trained_model = nn.get('MODEL');
```

```

% Check whether the number of fully-connected layers matches (excluding Dense_output layer)1
assert(length(nn.get('LAYERS')) == sum(contains({trained_model.Layers.Name}, 'Dense')) - 1, ...
    [BRAPH2.STR ':NNRegressorMLP:' BRAPH2.FAIL_TEST], ...
    'NNRegressorMLP does not construct the layers correctly. The number of the inputs should be the same as the length of dense layers the property.' ...
    )
```

1 checks whether the number of layers from the trained model is correctly set.