Implement a new Neural Network Classifier

The BRAPH 2 Developers

September 20, 2023

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

Contents

Implementation of a Neural Network Classifier

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

Code 1: **NNClassifierMLP element header.** The `header` section of the generator code for `_NNClassifierMLP.gen.m` provides the general information about the `NNClassifierMLP` element.

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

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

(1) defines `NNClassifierMLP` as a subclass of `NNBase`. The moniker will be `nn`.

Code 2: **NNClassifierMLP element prop update.** The `props_update` section of the generator code for `_NNClassifierMLP.gen.m` updates the properties of the `NNClassifierMLP` 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 classifier.
%%%% idefault!
'NNClassifierMLP'

%%% iprop!
DESCRIPTION (constant, string) is the description of the neural network
     multi-layer perceptron classifier.
%%%% idefault!
'A neural network multi-layer perceptron classifier (NNClassifierMLP)
     comprises a multi-layer perceptron classifier model and a given dataset
     . NNClassifierMLP trains the multi-layer perceptron classifier 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 classifier.
%%%% isettings!
'NNClassifierMLP'

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

```

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

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

%%% iprop!
DP_CLASSES (parameter, classlist) is the list of compatible data points.
%%%% idefault! (2)
{'NNDataPoint_CON_CLA' 'NNDataPoint_CON_FUN_MP_CLA' 'NNDataPoint_Graph_CLA' 'NNDataPoint_Measure_CLA'}

%%% 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];
            end
        end
```

(1) defines `NNDataset` which contains the `NNDataPoint` to train this classifier.

(2) defines the compatible `NNDataPoint` classes with this `NNClassifierMLP`.

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

```
75          flattened_inputs_group = [flattened_inputs_group;
        flattened_inputs_individual'];
76      end
77      value = {flattened_inputs_group};
78  end
79
80  %%% iprop!
81  TARGETS (query, cell) constructs the targets in the CB (channel-batch)
        format with one-hot vectors.
82  %%%% icalculate! (4)
83  % targets = nn.get('TARGETS', D) returns a cell array with the
84  %  targets for all data points in dataset D with one-hot vectors.
85  if isempty(varargin)
86      value = {};
87      return
88  end
89  d = varargin{1};
90
91  target_ids = nn.get('TARGET_IDS', d);
92  value = onehotencode(categorical(target_ids), 2);
93
94  %%% iprop!
95  MODEL (result, net) is a trained neural network model.
96  %%%% icalculate! (5)
97  inputs = cell2mat(nn.get('INPUTS', nn.get('D'))); (6)
98  targets = nn.get('TARGET_IDS', nn.get('D')); (7)
99  if isempty(inputs) || isempty(targets)
100     value = network();
101 else
102     number_features = size(inputs, 2);
103     number_targets = size(targets, 2);
104     targets = categorical(targets);
105     number_classes = numel(categories(targets));
106
107     layers = nn.get('LAYERS'); (8)
108     nn_architecture = [featureInputLayer(number_features, 'Name', 'Input')];
109     for i = 1:1:length(layers)
110         nn_architecture = [nn_architecture
111             fullyConnectedLayer(layers(i), 'Name', ['Dense_' num2str(i)])
112             batchNormalizationLayer('Name', ['BatchNormalization_' num2str(i
        )])
113             dropoutLayer('Name', ['Dropout_' num2str(i)])
114             ];
115     end
116     nn_architecture = [nn_architecture
117         reluLayer('Name', 'Relu_output')
118         fullyConnectedLayer(number_classes, 'Name', 'Dense_output')
119         softmaxLayer
120         classificationLayer('Name', 'Output')
121         ];
122
123     % specify trianing options  (9)
124     options = trainingOptions(nn.get('SOLVER'), ...
125         'MiniBatchSize', nn.get('BATCH'), ...
126         'MaxEpochs', nn.get('EPOCHS'), ...
127         'Shuffle', nn.get('SHUFFLE'), ...
128         'Plots', nn.get('PLOT_TRAINING'), ...
129         'Verbose', nn.get('VERBOSE'));
130
```

(4) is a query that constructs the one-hot vectors for the target classes.

(5) trains the classifier with the defined dataset by the code under `icalculate!`.

(6) and (7) firstly extract the inputs and targets with the corresponding format.

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

(9) defines the neural network training options.

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

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

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

```
1  %% iprops!
2
3  %%% iprop!
4  TARGET_IDS (query, stringlist) constructs the target IDs which represent the
       class of each data point.
5  %%%% icalculate! (1)
6  % targets = nn.get('TARGET_IDS', D) returns a cell array with the
7  %  targets for all data points in dataset D.
8  if isempty(varargin)
9      value = {''};
10     return
11 end
12 d = varargin{1};
13 targets = d.get('TARGETS');
14 if isempty(targets)
15     value = {''};
16 else
17     nn_targets = [];
18     for i = 1:1:length(targets)
19         target = targets{i};
20         nn_targets = [nn_targets; target];
21     end
22     value = nn_targets;
23 end
24
25 %%% iprop! (2)
26 LAYERS (data, rvector) defines the number of layers and their neurons.
27 %%%% idefault!
28 [32 32]
29 %%%% igui!
30 pr = PanelPropRVectorSmart('EL', nn, 'PROP', NNClassifierMLP.LAYERS, ...
31     'MIN', 0, 'MAX', 2000, ...
32     'DEFAULT', NNClassifierMLP.getPropDefault('LAYERS'), ...
33     varargin{:});
34
35 %%% iprop!
36 WAITBAR (gui, logical) detemines whether to show the waitbar.
37 %%%% idefault!
38 true
39
40 %%% iprop!
41 INTERRUPTIBLE (gui, scalar) sets whether the comparison computation is
       interruptible for multitasking.
42 %%%% idefault!
43 .001
44
45 %%% iprop! (3)
46 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.
47 %%%% icalculate!
```

(1) is a query that collects all the target class for all data points.

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

(3) 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.

```
% fi = nn.get('FEATURE_IMPORTANCE', D, P, SEED) 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
end
targets = 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: **NNClassifierMLP element tests.** The `tests` section from the element generator `_NNClassifierMLP.gen.m`. A test for creating example files should be prepared to test the properties of the data point. Furthermore, additional test should be prepared for validating the value of input and target for the data point.

```
%% itests!

%%% itest!
%%%% iname!
train the classifier with example data
%%%% icode!

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

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

ba = im_ba.get('BA');

% Load Groups of SubjectCON
im_gr1 = ImporterGroupSubjectCON_XLS( ...
    'DIRECTORY', [fileparts(which('NNDataPoint_CON_CLA')) filesep 'Example
     data NN CLA CON XLS' filesep 'CON_Group_1_XLS'], ...
    'BA', ba, ...
    'WAITBAR', true ...
    );

gr1 = im_gr1.get('GR');

im_gr2 = ImporterGroupSubjectCON_XLS( ...
    'DIRECTORY', [fileparts(which('NNDataPoint_CON_CLA')) filesep 'Example
     data NN CLA CON XLS' filesep 'CON_Group_2_XLS'], ...
    'BA', ba, ...
    'WAITBAR', true ...
    );

gr2 = im_gr2.get('GR');

% create item lists of NNDataPoint_CON_CLA
[~, group_folder_name] = fileparts(im_gr1.get('DIRECTORY'));
it_list1 = cellfun(@(x) NNDataPoint_CON_CLA( ...
    'ID', x.get('ID'), ...
    'SUB', x, ...
    'TARGET_IDS', {group_folder_name}), ...
    gr1.get('SUB_DICT').get('IT_LIST'), ...
    'UniformOutput', false);

[~, group_folder_name] = fileparts(im_gr2.get('DIRECTORY'));
it_list2 = cellfun(@(x) NNDataPoint_CON_CLA( ...
    'ID', x.get('ID'), ...
    'SUB', x, ...
```

```
51      'TARGET_IDS', {group_folder_name}), ...
52      gr2.get('SUB_DICT').get('IT_LIST'), ...
53      'UniformOutput', false);
54
55  % create NNDataPoint_CON_CLA DICT items
56  dp_list1 = IndexedDictionary(...
57          'IT_CLASS', 'NNDataPoint_CON_CLA', ...
58          'IT_LIST', it_list1 ...
59          );
60
61  dp_list2 = IndexedDictionary(...
62          'IT_CLASS', 'NNDataPoint_CON_CLA', ...
63          'IT_LIST', it_list2 ...
64          );
65
66  % create a NNDataset containing the NNDataPoint_CON_CLA DICT
67  d1 = NNDataset( ...
68      'DP_CLASS', 'NNDataPoint_CON_CLA', ...
69      'DP_DICT', dp_list1 ...
70      );
71
72  d2 = NNDataset( ...
73      'DP_CLASS', 'NNDataPoint_CON_CLA', ...
74      'DP_DICT', dp_list2 ...
75      );
76
77  % combine the two datasets
78  d = NNDatasetCombine('D_LIST', {d1, d2}).get('D');
79
80  nn = NNClassifierMLP('D', d, 'LAYERS', [10 10 10]);
81  trained_model = nn.get('MODEL');
82
83  % Check whether the number of fully-connected layer matches (excluding
        Dense_output layer)(1)
84  assert(length(nn.get('LAYERS')) == sum(contains({trained_model.Layers.Name},
        'Dense')) - 1, ...
85      [BRAPH2.STR ':NNClassifierMLP:' BRAPH2.FAIL_TEST], ...
86      'NNClassifierMLP does not construct the layers correctly. The number of
        the inputs should be the same as the length of dense layers the
        property.' ...
87      )
```

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