

Current scope: [all classes](#) | [org.apache.commons.math4.neuralnet](#)

## Coverage Summary for Class: Network (org.apache.commons.math4.neuralnet)

Class	Class, %	Method, %	Line, %
Network	100% (1/1)	63.2% (12/19)	55.8% (48/86)

```

1  /*
2  * Licensed to the Apache Software Foundation (ASF) under one or more
3  * contributor license agreements. See the NOTICE file distributed with
4  * this work for additional information regarding copyright ownership.
5  * The ASF licenses this file to You under the Apache License, Version 2.0
6  * (the "License"); you may not use this file except in compliance with
7  * the License. You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10 *
11 * Unless required by applicable law or agreed to in writing, software
12 * distributed under the License is distributed on an "AS IS" BASIS,
13 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14 * See the License for the specific language governing permissions and
15 * limitations under the License.
16 */
17
18 package org.apache.commons.math4.neuralnet;
19
20 import java.util.NoSuchElementException;
21 import java.util.List;
22 import java.util.ArrayList;
23 import java.util.Set;
24 import java.util.HashSet;
25 import java.util.Collection;
26 import java.util.Iterator;
27 import java.util.Collections;
28 import java.util.Map;
29 import java.util.concurrent.ConcurrentHashMap;
30 import java.util.concurrent.atomic.AtomicLong;
31 import java.util.stream.Collectors;
32
33 import org.apache.commons.math4.neuralnet.internal.NeuralNetException;
34
35 /**
36 * Neural network, composed of {@link Neuron} instances and the links
37 * between them.
38 *
39 * Although updating a neuron's state is thread-safe, modifying the
40 * network's topology (adding or removing links) is not.

```

```

41  *
42  * @since 3.3
43  */
44  public class Network
45  implements Iterable<Neuron> {
46      /** Neurons. */
47      final ConcurrentHashMap<Long, Neuron> neuronMap
48          = new ConcurrentHashMap<>();
49      /** Next available neuron identifier. */
50      final AtomicLong nextId;
51      /** Neuron's features set size. */
52      final int featureSize;
53      /** Links. */
54      final ConcurrentHashMap<Long, Set<Long>> linkMap
55          = new ConcurrentHashMap<>();
56
57      /**
58       * @param firstId Identifier of the first neuron that will be added
59       * to this network.
60       * @param featureSize Size of the neuron's features.
61       */
62      public Network(long firstId,
63                     int featureSize) {
64          this.nextId = new AtomicLong(firstId);
65          this.featureSize = featureSize;
66      }
67
68      /**
69       * Builds a network from a list of neurons and their neighbours.
70       *
71       * @param featureSize Number of features.
72       * @param idList List of neuron identifiers.
73       * @param featureList List of neuron features.
74       * @param neighbourIdList Links associated to each of the neurons in
75       * {@code idList}.
76       * @throws IllegalArgumentException if an inconsistency is detected.
77       * @return a new instance.
78       */
79      public static Network from(int featureSize,
80                                long[] idList,
81                                double[][] featureList,
82                                long[][] neighbourIdList) {
83          final int numNeurons = idList.length;
84          if (idList.length != featureList.length) {
85              throw new NeuralNetException(NeuralNetException.SIZE_MISMATCH,
86                                             idList.length, featureList.length);
87          }
88          if (idList.length != neighbourIdList.length) {
89              throw new NeuralNetException(NeuralNetException.SIZE_MISMATCH,
90                                             idList.length, neighbourIdList.length);
91          }

```

```
92
93     final Network net = new Network(Long.MIN_VALUE, featureSize);
94
95     for (int i = 0; i < numNeurons; i++) {
96         final long id = idList[i];
97         net.createNeuron(id, featureList[i]);
98     }
99
100    for (int i = 0; i < numNeurons; i++) {
101        final Neuron a = net.getNeuron(idList[i]);
102        for (final long id : neighbourIdList[i]) {
103            final Neuron b = net.neuronMap.get(id);
104            if (b == null) {
105                throw new NeuralNetException(NeuralNetException.ID_NOT_FOUND, id);
106            }
107            net.addLink(a, b);
108        }
109    }
110
111    return net;
112 }
113
114 /**
115  * Performs a deep copy of this instance.
116  * Upon return, the copied and original instances will be independent:
117  * Updating one will not affect the other.
118  *
119  * @return a new instance with the same state as this instance.
120  * @since 3.6
121  */
122 public synchronized Network copy() {
123     final Network copy = new Network(nextId.get(),
124                                     featureSize);
125
126
127     for (final Map.Entry<Long, Neuron> e : neuronMap.entrySet()) {
128         copy.neuronMap.put(e.getKey(), e.getValue().copy());
129     }
130
131     for (final Map.Entry<Long, Set<Long>> e : linkMap.entrySet()) {
132         copy.linkMap.put(e.getKey(), new HashSet<>(e.getValue()));
133     }
134
135     return copy;
136 }
137
138 /**
139  * {@inheritDoc}
140  */
141 @Override
142 public Iterator<Neuron> iterator() {
```

```
143     return neuronMap.values().iterator();
144 }
145
146 /**
147  * @return a shallow copy of the network's neurons.
148  */
149 public Collection<Neuron> getNeurons() {
150     return Collections.unmodifiableCollection(neuronMap.values());
151 }
152
153 /**
154  * Creates a neuron and assigns it a unique identifier.
155  *
156  * @param features Initial values for the neuron's features.
157  * @return the neuron's identifier.
158  * @throws IllegalArgumentException if the length of {@code features}
159  * is different from the expected size (as set by the
160  * {@link #Network(long,int) constructor}).
161  */
162 public long createNeuron(double[] features) {
163     return createNeuron(createNextId(), features);
164 }
165
166 /**
167  * @param id Identifier.
168  * @param features Features.
169  * @return {@code id}.
170  * @throws IllegalArgumentException if the identifier is already used
171  * by a neuron that belongs to this network or the features size does
172  * not match the expected value.
173  */
174 long createNeuron(long id,
175                   double[] features) {
176     if (neuronMap.get(id) != null) {
177         throw new NeuralNetException(NeuralNetException.ID_IN_USE, id);
178     }
179
180     if (features.length != featureSize) {
181         throw new NeuralNetException(NeuralNetException.SIZE_MISMATCH,
182                                     features.length, featureSize);
183     }
184
185     neuronMap.put(id, new Neuron(id, features.clone()));
186     linkMap.put(id, new HashSet<>());
187
188     if (id > nextId.get()) {
189         nextId.set(id);
190     }
191
192     return id;
193 }
```

```
194
195 /**
196  * Deletes a neuron.
197  * Links from all neighbours to the removed neuron will also be
198  * {@link #deleteLink(Neuron,Neuron) deleted}.
199  *
200  * @param neuron Neuron to be removed from this network.
201  * @throws NoSuchElementException if {@code n} does not belong to
202  * this network.
203  */
204 public void deleteNeuron(Neuron neuron) {
205     // Delete links to from neighbours.
206     getNeighbours(neuron).forEach(neighbour -> deleteLink(neighbour, neuron));
207
208     // Remove neuron.
209     neuronMap.remove(neuron.getIdentifier());
210 }
211
212 /**
213  * Gets the size of the neurons' features set.
214  *
215  * @return the size of the features set.
216  */
217 public int getFeaturesSize() {
218     return featureSize;
219 }
220
221 /**
222  * Adds a link from neuron {@code a} to neuron {@code b}.
223  * Note: the link is not bi-directional; if a bi-directional link is
224  * required, an additional call must be made with {@code a} and
225  * {@code b} exchanged in the argument list.
226  *
227  * @param a Neuron.
228  * @param b Neuron.
229  * @throws NoSuchElementException if the neurons do not exist in the
230  * network.
231  */
232 public void addLink(Neuron a,
233                    Neuron b) {
234     // Check that the neurons belong to this network.
235     final long aId = a.getIdentifier();
236     if (a != getNeuron(aId)) {
237         throw new NoSuchElementException(Long.toString(aId));
238     }
239     final long bId = b.getIdentifier();
240     if (b != getNeuron(bId)) {
241         throw new NoSuchElementException(Long.toString(bId));
242     }
243
244     // Add link from "a" to "b".
```

```
245     addLinkToLinkSet(linkMap.get(aId), bId);
246 }
247
248 /**
249  * Adds a link to neuron {@code id} in given {@code linkSet}.
250  * Note: no check verifies that the identifier indeed belongs
251  * to this network.
252  *
253  * @param linkSet Neuron identifier.
254  * @param id Neuron identifier.
255  */
256 private void addLinkToLinkSet(Set<Long> linkSet,
257                               long id) {
258     linkSet.add(id);
259 }
260
261 /**
262  * Deletes the link between neurons {@code a} and {@code b}.
263  *
264  * @param a Neuron.
265  * @param b Neuron.
266  * @throws NoSuchElementException if the neurons do not exist in the
267  * network.
268  */
269 public void deleteLink(Neuron a,
270                        Neuron b) {
271     // Check that the neurons belong to this network.
272     final long aId = a.getIdentifier();
273     if (a != getNeuron(aId)) {
274         throw new NoSuchElementException(Long.toString(aId));
275     }
276     final long bId = b.getIdentifier();
277     if (b != getNeuron(bId)) {
278         throw new NoSuchElementException(Long.toString(bId));
279     }
280
281     // Delete link from "a" to "b".
282     deleteLinkFromLinkSet(linkMap.get(aId), bId);
283 }
284
285 /**
286  * Deletes a link to neuron {@code id} in given {@code linkSet}.
287  * Note: no check verifies that the identifier indeed belongs
288  * to this network.
289  *
290  * @param linkSet Neuron identifier.
291  * @param id Neuron identifier.
292  */
293 private void deleteLinkFromLinkSet(Set<Long> linkSet,
294                                    long id) {
295     linkSet.remove(id);
```

```
296     }
297
298     /**
299     * Retrieves the neuron with the given (unique) {@code id}.
300     *
301     * @param id Identifier.
302     * @return the neuron associated with the given {@code id}.
303     * @throws NoSuchElementException if the neuron does not exist in the
304     * network.
305     */
306     public Neuron getNeuron(long id) {
307         final Neuron n = neuronMap.get(id);
308         if (n == null) {
309             throw new NoSuchElementException(Long.toString(id));
310         }
311         return n;
312     }
313
314     /**
315     * Retrieves the neurons in the neighbourhood of any neuron in the
316     * {@code neurons} list.
317     * @param neurons Neurons for which to retrieve the neighbours.
318     * @return the list of neighbours.
319     * @see #getNeighbours(Iterable,Iterable)
320     */
321     public Collection<Neuron> getNeighbours(Iterable<Neuron> neurons) {
322         return getNeighbours(neurons, null);
323     }
324
325     /**
326     * Retrieves the neurons in the neighbourhood of any neuron in the
327     * {@code neurons} list.
328     * The {@code exclude} list allows to retrieve the "concentric"
329     * neighbourhoods by removing the neurons that belong to the inner
330     * "circles".
331     *
332     * @param neurons Neurons for which to retrieve the neighbours.
333     * @param exclude Neurons to exclude from the returned list.
334     * Can be {@code null}.
335     * @return the list of neighbours.
336     */
337     public Collection<Neuron> getNeighbours(Iterable<Neuron> neurons,
338                                           Iterable<Neuron> exclude) {
339         final Set<Long> idList = new HashSet<>();
340         neurons.forEach(n -> idList.addAll(linkMap.get(n.getIdentifier())));
341
342         if (exclude != null) {
343             exclude.forEach(n -> idList.remove(n.getIdentifier()));
344         }
345
346         return idList.stream().map(this::getNeuron).collect(Collectors.toList());
```

```
347     }
348
349     /**
350     * Retrieves the neighbours of the given neuron.
351     *
352     * @param neuron Neuron for which to retrieve the neighbours.
353     * @return the list of neighbours.
354     * @see #getNeighbours(Neuron,Iterable)
355     */
356     public Collection<Neuron> getNeighbours(Neuron neuron) {
357         return getNeighbours(neuron, null);
358     }
359
360     /**
361     * Retrieves the neighbours of the given neuron.
362     *
363     * @param neuron Neuron for which to retrieve the neighbours.
364     * @param exclude Neurons to exclude from the returned list.
365     * Can be {@code null}.
366     * @return the list of neighbours.
367     */
368     public Collection<Neuron> getNeighbours(Neuron neuron,
369                                           Iterable<Neuron> exclude) {
370         final Set<Long> idList = linkMap.get(neuron.getIdentifier());
371         if (exclude != null) {
372             for (final Neuron n : exclude) {
373                 idList.remove(n.getIdentifier());
374             }
375         }
376
377         final List<Neuron> neuronList = new ArrayList<>();
378         for (final Long id : idList) {
379             neuronList.add(getNeuron(id));
380         }
381
382         return neuronList;
383     }
384
385     /**
386     * Creates a neuron identifier.
387     *
388     * @return a value that will serve as a unique identifier.
389     */
390     private Long createNextId() {
391         return nextId.getAndIncrement();
392     }
393 }
```