Current scope: all classes | org.apache.commons.math4.neuralnet

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

Class	Class, %	Method, %	Line, %
Neuron	100% (1/1)	100% (9/9)	85.3% (29/34)

```
 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.concurrent.atomic.AtomicReference;
21 import java.util.concurrent.atomic.AtomicLong;
22
23 import org.apache.commons.numbers.core.Precision;
24 import org.apache.commons.math4.neuralnet.internal.NeuralNetException;
25
26 /**
27  * Describes a neuron element of a neural network.
28  *
29  * This class aims to be thread-safe.
30  *
31  * @since 3.3
32  */
33 public class Neuron {
34     /** Identifier. */
35     private final long identifier;
36     /** Length of the feature set. */
37     private final int size;
38     /** Neuron data. */
39     private final AtomicReference<double[]> features;
40     /** Number of attempts to update a neuron. */
```

```
41      private final AtomicLong numberOfAttemptedUpdates = new AtomicLong(0);
42      /** Number of successful updates  of a neuron. */
43      private final AtomicLong numberOfSuccessfulUpdates = new AtomicLong(0);
44
45      /**
46       * Creates a neuron.
47       * The size of the feature set is fixed to the length of the given
48       * argument.
49       * <br>
50       * Constructor is package-private: Neurons must be
51       * {@link Network#createNeuron(double[]) created} by the network
52       * instance to which they will belong.
53       *
54       * @param identifier Identifier (assigned by the {@link Network}).
55       * @param features Initial values of the feature set.
56       */
57      Neuron(long identifier,
58             double[] features) {
59          this.identifier = identifier;
60          this.size = features.length;
61          this.features = new AtomicReference<>(features.clone());
62      }
63
64      /**
65       * Performs a deep copy of this instance.
66       * Upon return, the copied and original instances will be independent:
67       * Updating one will not affect the other.
68       *
69       * @return a new instance with the same state as this instance.
70       * @since 3.6
71       */
72      public synchronized Neuron copy() {
73          final Neuron copy = new Neuron(getIdentifier(),
74                                         getFeatures());
75          copy.numberOfAttemptedUpdates.set(numberOfAttemptedUpdates.get());
76          copy.numberOfSuccessfulUpdates.set(numberOfSuccessfulUpdates.get());
77
78          return copy;
79      }
80
81      /**
82       * Gets the neuron's identifier.
83       *
84       * @return the identifier.
85       */
86      public long getIdentifier() {
87          return identifier;
88      }
89
90      /**
91       * Gets the length of the feature set.
```

```
92      *
93      * @return the number of features.
94      */
95     public int getSize() {
96         return size;
97     }
98 
99     /**
100     * Gets the neuron's features.
101     *
102     * @return a copy of the neuron's features.
103     */
104    public double[] getFeatures() {
105        return features.get().clone();
106    }
107 
108    /**
109     * Tries to atomically update the neuron's features.
110     * Update will be performed only if the expected values match the
111     * current values.<br>
112     * In effect, when concurrent threads call this method, the state
113     * could be modified by one, so that it does not correspond to the
114     * the state assumed by another.
115     * Typically, a caller {@link #getFeatures() retrieves the current state},
116     * and uses it to compute the new state.
117     * During this computation, another thread might have done the same
118     * thing, and updated the state: If the current thread were to proceed
119     * with its own update, it would overwrite the new state (which might
120     * already have been used by yet other threads).
121     * To prevent this, the method does not perform the update when a
122     * concurrent modification has been detected, and returns {@code false}.
123     * When this happens, the caller should fetch the new current state,
124     * redo its computation, and call this method again.
125     *
126     * @param expect Current values of the features, as assumed by the caller.
127     * Update will never succeed if the contents of this array does not match
128     * the values returned by {@link #getFeatures()}.
129     * @param update Features's new values.
130     * @return {@code true} if the update was successful, {@code false}
131     * otherwise.
132     * @throws IllegalArgumentException if the length of {@code update} is
133     * not the same as specified in the {@link #Neuron(long,double[])
134     * constructor}.
135     */
136    public boolean compareAndSetFeatures(double[] expect,
137                                         double[] update) {
138        if (update.length != size) {
139            throw new NeuralNetException(NeuralNetException.SIZE_MISMATCH,
140                                         update.length, size);
141        }
142 
```

```
143         // Get the internal reference. Note that this must not be a copy;
144         // otherwise the "compareAndSet" below will always fail.
145         final double[] current = features.get();
146         if (!containSameValues(current, expect)) {
147             // Some other thread already modified the state.
148             return false;
149         }
150 
151         // Increment attempt counter.
152         numberOfAttemptedUpdates.incrementAndGet();
153 
154         if (features.compareAndSet(current, update.clone())) {
155             // The current thread could atomically update the state (attempt succeeded).
156             numberOfSuccessfulUpdates.incrementAndGet();
157             return true;
158         } else {
159             // Some other thread came first (attempt failed).
160             return false;
161         }
162     }
163 
164     /**
165      * Retrieves the number of calls to the
166      * {@link #compareAndSetFeatures(double[],double[]) compareAndSetFeatures}
167      * method.
168      * Note that if the caller wants to use this method in combination with
169      * {@link #getNumberOfSuccessfulUpdates()}, additional synchronization
170      * may be required to ensure consistency.
171      *
172      * @return the number of update attempts.
173      * @since 3.6
174      */
175     public long getNumberOfAttemptedUpdates() {
176         return numberOfAttemptedUpdates.get();
177     }
178 
179     /**
180      * Retrieves the number of successful calls to the
181      * {@link #compareAndSetFeatures(double[],double[]) compareAndSetFeatures}
182      * method.
183      * Note that if the caller wants to use this method in combination with
184      * {@link #getNumberOfAttemptedUpdates()}, additional synchronization
185      * may be required to ensure consistency.
186      *
187      * @return the number of successful updates.
188      * @since 3.6
189      */
190     public long getNumberOfSuccessfulUpdates() {
191         return numberOfSuccessfulUpdates.get();
192     }
193 
```

```
194     /**
195      * Checks whether the contents of both arrays is the same.
196      *
197      * @param current Current values.
198      * @param expect Expected values.
199      * @throws IllegalArgumentException if the length of {@code expect}
200      * is not the same as specified in the {@link #Neuron(long,double[])
201      * constructor}.
202      * @return {@code true} if the arrays contain the same values.
203      */
204     private boolean containSameValues(double[] current,
205                                       double[] expect) {
206         if (expect.length != size) {
207             throw new NeuralNetException(NeuralNetException.SIZE_MISMATCH,
208                                          expect.length, size);
209         }
210 
211         for (int i = 0; i < size; i++) {
212             if (!Precision.equals(current[i], expect[i])) {
213                 return false;
214             }
215         }
216         return true;
217     }
218 }
```

generated on 2023-10-23 18:01