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)

```java
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.commons.math4.neuralnet;

import java.util.NoSuchElementException;
import java.util.List;
import java.util.ArrayList;
import java.util.Set;
import java.util.HashSet;
import java.util.Collection;
import java.util.Iterator;
import java.util.Collections;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicLong;
import java.util.stream.Collectors;

import org.apache.commons.math4.neuralnet.internal.NeuralNetException;

/**
 * Neural network, composed of {@link Neuron} instances and the links
 * between them.
 *
 * Although updating a neuron's state is thread-safe, modifying the
 * network's topology (adding or removing links) is not.
 *
 * @since 3.3
 */
public class Network
    implements Iterable<Neuron> {
    /** Neurons. */
    final ConcurrentHashMap<Long, Neuron> neuronMap
        = new ConcurrentHashMap<>();
    /** Next available neuron identifier. */
    final AtomicLong nextId;
    /** Neuron's features set size. */
    final int featureSize;
    /** Links. */
    final ConcurrentHashMap<Long, Set<Long>> linkMap
        = new ConcurrentHashMap<>();

    /**
     * @param firstId Identifier of the first neuron that will be added
     * to this network.
     * @param featureSize Size of the neuron's features.
     */
    public Network(long firstId,
                   int featureSize) {
        this.nextId = new AtomicLong(firstId);
        this.featureSize = featureSize;
    }

    /**
     * Builds a network from a list of neurons and their neighbours.
     *
     * @param featureSize Number of features.
     * @param idList List of neuron identifiers.
     * @param featureList List of neuron features.
     * @param neighbourIdList Links associated to each of the neurons in
     * {@code idList}.
     * @throws IllegalArgumentException if an inconsistency is detected.
     * @return a new instance.
     */
    public static Network from(int featureSize,
                               long[] idList,
                               double[][] featureList,
                               long[][] neighbourIdList) {
        final int numNeurons = idList.length;
        if (idList.length != featureList.length) {
            throw new NeuralNetException(NeuralNetException.SIZE_MISMATCH,
                                          idList.length, featureList.length);
        }
        if (idList.length != neighbourIdList.length) {
            throw new NeuralNetException(NeuralNetException.SIZE_MISMATCH,
                                          idList.length, neighbourIdList.length);
        }

        final Network net = new Network(Long.MIN_VALUE, featureSize);

        for (int i = 0; i < numNeurons; i++) {
            final long id = idList[i];
            net.createNeuron(id, featureList[i]);
        }

        for (int i = 0; i < numNeurons; i++) {
            final Neuron a = net.getNeuron(idList[i]);
            for (final long id : neighbourIdList[i]) {
                final Neuron b = net.neuronMap.get(id);
                if (b == null) {
                    throw new NeuralNetException(NeuralNetException.ID_NOT_FOUND, id);
                }
                net.addLink(a, b);
            }
        }

        return net;
    }

    /**
     * Performs a deep copy of this instance.
     * Upon return, the copied and original instances will be independent:
     * Updating one will not affect the other.
     *
     * @return a new instance with the same state as this instance.
     * @since 3.6
     */
    public synchronized Network copy() {
        final Network copy = new Network(nextId.get(),
                                         featureSize);


        for (final Map.Entry<Long, Neuron> e : neuronMap.entrySet()) {
            copy.neuronMap.put(e.getKey(), e.getValue().copy());
        }

        for (final Map.Entry<Long, Set<Long>> e : linkMap.entrySet()) {
            copy.linkMap.put(e.getKey(), new HashSet<>(e.getValue()));
        }

        return copy;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public Iterator<Neuron> iterator() {
        return neuronMap.values().iterator();
    }

    /**
     * @return a shallow copy of the network's neurons.
     */
    public Collection<Neuron> getNeurons() {
        return Collections.unmodifiableCollection(neuronMap.values());
    }

    /**
     * Creates a neuron and assigns it a unique identifier.
     *
     * @param features Initial values for the neuron's features.
     * @return the neuron's identifier.
     * @throws IllegalArgumentException if the length of {@code features}
     * is different from the expected size (as set by the
     * {@link #Network(long,int) constructor}).
     */
    public long createNeuron(double[] features) {
        return createNeuron(createNextId(), features);
    }

    /**
     * @param id Identifier.
     * @param features Features.
     * @return {@¢ode id}.
     * @throws IllegalArgumentException if the identifier is already used
     * by a neuron that belongs to this network or the features size does
     * not match the expected value.
     */
    long createNeuron(long id,
                      double[] features) {
        if (neuronMap.get(id) != null) {
            throw new NeuralNetException(NeuralNetException.ID_IN_USE, id);
        }

        if (features.length != featureSize) {
            throw new NeuralNetException(NeuralNetException.SIZE_MISMATCH,
                                         features.length, featureSize);
        }

        neuronMap.put(id, new Neuron(id, features.clone()));
        linkMap.put(id, new HashSet<>());

        if (id > nextId.get()) {
            nextId.set(id);
        }

        return id;
    }
```

```java

    /**
     * Deletes a neuron.
     * Links from all neighbours to the removed neuron will also be
     * {@link #deleteLink(Neuron,Neuron) deleted}.
     *
     * @param neuron Neuron to be removed from this network.
     * @throws NoSuchElementException if {@code n} does not belong to
     * this network.
     */
    public void deleteNeuron(Neuron neuron) {
        // Delete links to from neighbours.
        getNeighbours(neuron).forEach(neighbour -> deleteLink(neighbour, neuron));

        // Remove neuron.
        neuronMap.remove(neuron.getIdentifier());
    }

    /**
     * Gets the size of the neurons' features set.
     *
     * @return the size of the features set.
     */
    public int getFeaturesSize() {
        return featureSize;
    }

    /**
     * Adds a link from neuron {@code a} to neuron {@code b}.
     * Note: the link is not bi-directional; if a bi-directional link is
     * required, an additional call must be made with {@code a} and
     * {@code b} exchanged in the argument list.
     *
     * @param a Neuron.
     * @param b Neuron.
     * @throws NoSuchElementException if the neurons do not exist in the
     * network.
     */
    public void addLink(Neuron a,
                        Neuron b) {
        // Check that the neurons belong to this network.
        final long aId = a.getIdentifier();
        if (a != getNeuron(aId)) {
            throw new NoSuchElementException(Long.toString(aId));
        }
        final long bId = b.getIdentifier();
        if (b != getNeuron(bId)) {
            throw new NoSuchElementException(Long.toString(bId));
        }

        // Add link from "a" to "b".
        addLinkToLinkSet(linkMap.get(aId), bId);
    }

    /**
     * Adds a link to neuron {@code id} in given {@code linkSet}.
     * Note: no check verifies that the identifier indeed belongs
     * to this network.
     *
     * @param linkSet Neuron identifier.
     * @param id Neuron identifier.
     */
    private void addLinkToLinkSet(Set<Long> linkSet,
                                  long id) {
        linkSet.add(id);
    }

    /**
     * Deletes the link between neurons {@code a} and {@code b}.
     *
     * @param a Neuron.
     * @param b Neuron.
     * @throws NoSuchElementException if the neurons do not exist in the
     * network.
     */
    public void deleteLink(Neuron a,
                           Neuron b) {
        // Check that the neurons belong to this network.
        final long aId = a.getIdentifier();
        if (a != getNeuron(aId)) {
            throw new NoSuchElementException(Long.toString(aId));
        }
        final long bId = b.getIdentifier();
        if (b != getNeuron(bId)) {
            throw new NoSuchElementException(Long.toString(bId));
        }

        // Delete link from "a" to "b".
        deleteLinkFromLinkSet(linkMap.get(aId), bId);
    }

    /**
     * Deletes a link to neuron {@code id} in given {@code linkSet}.
     * Note: no check verifies that the identifier indeed belongs
     * to this network.
     *
     * @param linkSet Neuron identifier.
     * @param id Neuron identifier.
     */
    private void deleteLinkFromLinkSet(Set<Long> linkSet,
                                       long id) {
        linkSet.remove(id);
```

```java
    }

    /**
     * Retrieves the neuron with the given (unique) {@code id}.
     *
     * @param id Identifier.
     * @return the neuron associated with the given {@code id}.
     * @throws NoSuchElementException if the neuron does not exist in the
     * network.
     */
    public Neuron getNeuron(long id) {
        final Neuron n = neuronMap.get(id);
        if (n == null) {
            throw new NoSuchElementException(Long.toString(id));
        }
        return n;
    }

    /**
     * Retrieves the neurons in the neighbourhood of any neuron in the
     * {@code neurons} list.
     * @param neurons Neurons for which to retrieve the neighbours.
     * @return the list of neighbours.
     * @see #getNeighbours(Iterable,Iterable)
     */
    public Collection<Neuron> getNeighbours(Iterable<Neuron> neurons) {
        return getNeighbours(neurons, null);
    }

    /**
     * Retrieves the neurons in the neighbourhood of any neuron in the
     * {@code neurons} list.
     * The {@code exclude} list allows to retrieve the "concentric"
     * neighbourhoods by removing the neurons that belong to the inner
     * "circles".
     *
     * @param neurons Neurons for which to retrieve the neighbours.
     * @param exclude Neurons to exclude from the returned list.
     * Can be {@code null}.
     * @return the list of neighbours.
     */
    public Collection<Neuron> getNeighbours(Iterable<Neuron> neurons,
                                           Iterable<Neuron> exclude) {
        final Set<Long> idList = new HashSet<>();
        neurons.forEach(n -> idList.addAll(linkMap.get(n.getIdentifier())));

        if (exclude != null) {
            exclude.forEach(n -> idList.remove(n.getIdentifier()));
        }

        return idList.stream().map(this::getNeuron).collect(Collectors.toList());
    }

    /**
     * Retrieves the neighbours of the given neuron.
     *
     * @param neuron Neuron for which to retrieve the neighbours.
     * @return the list of neighbours.
     * @see #getNeighbours(Neuron,Iterable)
     */
    public Collection<Neuron> getNeighbours(Neuron neuron) {
        return getNeighbours(neuron, null);
    }

    /**
     * Retrieves the neighbours of the given neuron.
     *
     * @param neuron Neuron for which to retrieve the neighbours.
     * @param exclude Neurons to exclude from the returned list.
     * Can be {@code null}.
     * @return the list of neighbours.
     */
    public Collection<Neuron> getNeighbours(Neuron neuron,
                                            Iterable<Neuron> exclude) {
        final Set<Long> idList = linkMap.get(neuron.getIdentifier());
        if (exclude != null) {
            for (final Neuron n : exclude) {
                idList.remove(n.getIdentifier());
            }
        }

        final List<Neuron> neuronList = new ArrayList<>();
        for (final Long id : idList) {
            neuronList.add(getNeuron(id));
        }

        return neuronList;
    }

    /**
     * Creates a neuron identifier.
     *
     * @return a value that will serve as a unique identifier.
     */
    private Long createNextId() {
        return nextId.getAndIncrement();
    }
}
```

generated on 2023-11-08 02:37