ПРИЛОЖЕНИЕ А

*(обязательное)*

Исходный текст функции convolve()

public DataSet convolve(final List<DataSet> kernels, final DataSet input, final int padding) {  
 final int inputWidth = input.getDimension().getWidth();  
 final int inputHeight = input.getDimension().getHeight();  
 final int inputDepth = input.getDimension().getDepth();  
  
 final int kernelWidth = kernels.get(0).getDimension().getWidth();  
 final int kernelHeight = kernels.get(0).getDimension().getHeight();  
 final int kernelDepth = kernels.get(0).getDimension().getDepth();  
 final int kernelCount = kernels.size();  
  
 final int outputWidth = inputWidth - kernelWidth + 1 + padding \* 2;  
 final int outputHeight = inputHeight - kernelHeight + 1 + padding \* 2;  
 final int outputDepth = kernelCount;  
  
 if (kernels.size() == 1 && inputDepth > 1) {  
  
 }  
  
 final List<Double> result = new ArrayList<>(outputDepth \* outputHeight \* outputWidth);  
  
 int ay = -padding;  
 for (int y = 0; y < outputHeight; y++, ay++) {  
  
 int ax = -padding;  
 for (int x = 0; x < outputWidth; x++, ax++) {  
  
 for (DataSet kernel : kernels) {

double res = 0.0;  
  
 for (int fx = 0; fx < kernelWidth; fx++) {  
 int ox = ax + fx;

for (int fy = 0; fy < kernelHeight; fy++) {  
 int oy = ay + fy;  
  
 if (oy >= 0 && oy < inputHeight && ox >=

0 && ox < inputWidth) {  
  
 for (int z = 0; z < inputDepth; z++){

res += kernel.get(fx, fy, z) \*

input.get(ox, oy, z);  
 }  
 }  
 }  
 }  
 result.add(res);  
 }  
 }  
 }  
 return new DataSet(result, new Dimension(outputWidth,

outputHeight, outputDepth));  
}

ПРИЛОЖЕНИЕ Б

*(обязательное)*

Исходный текст класса DataSet()

public class DataSet {  
  
 private final List<Double> data = new ArrayList<>();  
 private Dimension dimension;  
  
 public final static DataSet *EMPTY* = new DataSet(Collections.*singletonList*(0.0), Dimension.*EMPTY*);  
  
 public DataSet(final Double[] data, final Dimension dimension) {  
 Validate.*isTrue*(data.length == dimension.getSize(),  
 "data size must match dimension");  
 this.data.addAll(Arrays.*asList*(data));  
 this.dimension = dimension;  
 }  
  
 public DataSet(final double[] data, final Dimension dimension) {  
 Validate.*isTrue*(data.length == dimension.getSize(),  
 "data size must match dimension");  
 this.data.addAll(Arrays.*stream*(data).boxed().collect(Collectors.*toList*()));  
 this.dimension = dimension;  
 }  
  
 public DataSet(final double[][] data, final Dimension dimension) {  
 this(Arrays.*stream*(data).flatMapToDouble(Arrays::*stream*)  
 .boxed().collect(Collectors.*toList*()), dimension);  
 }  
  
 public DataSet(final Collection<Double> data, final Dimension dimension) {  
 Validate.*isTrue*(data.size() == dimension.getSize(),  
 "data size must match dimension");  
 this.data.addAll(data);  
 this.dimension = dimension;  
 }  
  
public DataSet(final DataSet dataSet) {  
 this(dataSet.getData(), dataSet.getDimension());  
 }  
public DataSet(final Dimension dimension, final DoubleSupplier supplier) {  
 this(DoubleStream.*generate*(supplier).limit(dimension.getSize()).toArray(), dimension);  
 }  
  
 public DataSet update(final DoubleUnaryOperator operator) {  
 this.data.replaceAll(operator::applyAsDouble);  
 return this;  
 }  
  
 public DataSet merge(final DataSet other, final DoubleBinaryOperator operator) {  
 Validate.*isTrue*(this.getDimension().getSize() == other.getDimension().getSize(),  
 "Dimensions must match");  
  
 final double[] dataArray = getArrayData();  
 final double[] otherArray = other.getArrayData();  
 Arrays.*setAll*(dataArray, i -> operator.applyAsDouble(dataArray[i], otherArray[i]));  
 this.data.clear();  
 this.data.addAll(Arrays.*stream*(dataArray).boxed().collect(Collectors.*toList*()));  
 return this;  
 }  
  
 public List<Double> getData() {  
 return this.data;  
 }  
  
 public DoubleStream getStreamData() {  
 return this.data.stream().mapToDouble(Double::*valueOf*);  
 }  
  
 public Double[] getWrapperArrayData() {  
 return this.data.toArray(new Double[this.data.size()]);  
 }  
  
 public int getSize() {  
 return data.size();  
 }  
  
 public Dimension getDimension() {  
 return dimension;  
 }  
  
 public DataSet update(final List<Double> data, final Dimension dimension) {  
 Validate.*notNull*(dimension, "dimension must not be null");  
 Validate.*notNull*(data, "data must not be null");  
 Validate.*isTrue*(data.size() == dimension.getSize(), "data size must match dimension");  
 this.data.clear();  
 this.data.addAll(data);  
 this.dimension = dimension;  
 return this;  
 }  
  
 public DataSet update(final DataSet dataSet) {  
 return update(dataSet.getData(), dataSet.getDimension());  
 }  
public DataSet rotate() {  
 final int width = dimension.getWidth();  
 final int height = dimension.getHeight();  
 final int depth = dimension.getDepth();  
 final DataSet result = new DataSet(this);  
  
 for (int k = 0; k < depth; k++) {  
 for (int i = 0; i < height; i++) {  
 for (int j = 0; j < width; j++) {  
 result.set(j, i, k, this.get(width - 1 - j, height - 1 - i, k));  
 }  
 }  
 }  
 return result;  
 }  
  
 public double[][] get2DArrayData() {  
 Validate.*isTrue*(dimension.getDepth() <= 1, "cannot get 2D data with 3 dimensions");  
 Validate.*isTrue*(dimension.getSize() == data.size(), "data size must match dimension");  
  
 final int width = dimension.getWidth();  
 final int height = dimension.getHeight();  
 double[][] result = new double[height][width];  
  
 for (int i = 0; i < width; i++) {  
 for (int j = 0; j < height; j++) {  
 result[j][i] = data.get(j \* width + i);  
 }  
 }  
 return result;  
 }  
  
 public double[] getArrayData() {  
 return data.stream().mapToDouble(Double::*valueOf*).toArray();  
 }  
  
 public double get(int idx) {  
 Validate.*inclusiveBetween*(0, data.size(), idx);  
 return data.get(idx);  
 }  
  
 public double get(int widthIdx, int heightIdx, int depthIdx) {  
 int idx = dimension.getDepth() \* dimension.getWidth() \* heightIdx +  
 dimension.getDepth() \* widthIdx + depthIdx;  
 return data.get(idx);  
 }  
  
 public void set(int widthIdx, int heightIdx, int depthIdx, double value) {  
 int idx = dimension.getDepth() \* dimension.getWidth() \* heightIdx +  
 dimension.getDepth() \* widthIdx + depthIdx;  
 Validate.*inclusiveBetween*(0, data.size(), idx);  
 this.data.set(idx, value);  
 }  
  
 public DataSet getChannel(final int channel) {  
 Validate.*isTrue*(channel >= 0 && channel < this.getDimension().getDepth(),  
 "this should be a valid channel index");  
 final int channelSize = getDimension().getHeight() \* getDimension().getWidth();  
 final double[] channelData = new double[channelSize];  
  
 final int channelCount = this.getDimension().getDepth();  
 for (int i = 0, j = 0; i < this.getSize(); i++) {  
 if (i % channelCount == channel) {  
 channelData[j++] = this.data.get(i);  
 }  
 }  
 return new DataSet(channelData, new Dimension(getDimension().getWidth(), getDimension().getHeight()));  
 }

ПРИЛОЖЕНИЕ В

*(обязательное)*

Спецификация проекта