## Part 1: OOP Concept

Objective

1) Be able to implement Objects and Classes.

Instruction

1) Create a Java Project named “2110215_Midterm_Part1”.
2) Copy all folders in “toStudent/Part1” to your project directory src folder.
3) You are to implement the following classes (detail for each class is given in section 3 and 4)

1) Activation (package util)
2) Neuron (package container)
3) Layer (package container)
4) Data (package container)
5) Network (package container)

4) JUnit for testing is in package test.grader

Problem Statement : NXOR, Neural-XOR

In the realm of computational science, the XOR gate presents a longstanding challenge with its elusive behavior. To overcome this obstacle, researchers are turning to neural networks, leveraging their adaptive learning and parallel processing capabilities. Each neuron within the network represents a spark of innovation, contributing to unraveling the mysteries of the XOR gate. Through iterative experimentation, guided by computational ingenuity, researchers aim to break through barriers and pave the way for transformative advancements. As guardians of digital evolution, their quest for mastery over the NXOR, Neural-XOR gate embodies the relentless pursuit of knowledge and progress, with the potential to reshape the future of computing.

This is how the program should work. This part is already provided.

When the application opens, the model will display the initial weight of the training process. By any user keyboard interaction will start the training process

```
Output functions:
0.4208696704937952
0.4421541157460952
0.46295603308620253
0.48256782826725986

Start Training ? :
```

By giving any user input, the program will continue the training process. Moreover, the training iterations are initialized in the main function (use 1,000,000 iterations).

After the training process is complete, the prediction of the given data will occur by forwarding the input data and display on the terminal.

```
Iteration 999999, Average Loss: 0.50010375595327562
Iteration 1000000, Average Loss: 0.5001037594470683
Output functions:
0.0011181525177909473
0.9980326879650764
0.9980325498160997
0.0016197775440304812
```

Lastly, the program shall exit with success code (code 0).

```
Process finished with exit code 0
```

4.1) Package util

You must implement ***ONE*** class of this package.

4.1.1) Class Activation

***You must implement this class from scratch***

These functions and their derivatives are commonly used in combination with networks for introducing non-linearity and performing backward processes during the training process.

Constructors

| Name | Description |
|---|---|
| + double sigmoid(double x) | Computes the sigmoid activation function<br><br>Definition:<br><br>$sigmoid: \sigma(x) = \frac{1}{1+e^{-x}}$ |
| + double tanh(double val) | Computes the hyperbolic tangent (tanh) activation function<br><br>Definition:<br><br>$tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$<br><br>*Note*: this is a formal definition of tanh function, but you may use Java built-in Math package, other implementation methods are applicable. |
| + double relu(double val) | Computes the Rectified Linear Unit (ReLU) activation function<br><br>Definition:<br><br>$relu(x) = max(x, 0)$<br><br>*Note*: you may use Java built-in Math package, other implementation methods are applicable. |
| + double sigmoidDerivative(double x) | Computes the derivative of the sigmoid activation function<br><br>Definition:<br><br>$sigmoid`(x) = \sigma(x)(1 - \sigma(x))$<br><br>*Note*: this is an easy definition of the derivative of sigmoid, if you are curious of how this equation form came to be, please use the Quotient rule (derivative of fraction) |

| | |
|---|---|
| | that you have learned in the high-school calculus mathematics class to prove on some scratch paper. |
| + double tanhDerivative(double val) | Computes the derivative of the hyperbolic tangent (tanh) activation function<br><br>Definition:<br><br>$tanh`(x) = 1 - tanh^2(x)$<br><br>Note: this is an easy definition of the derivative of tanh, if you are curious of how this equation form came to be, please use hyperbolic trigonometry theorem from calculus (I-II) class to prove on some scratch paper. |
| + double reluDerivative(double val) | Computes the derivative of the Rectified Linear Unit (ReLU) activation function<br><br>Definition:<br><br>$relu`(x) = 1 \; if \; x > 0 \; else \; 0$<br><br>I recommend using a ternary operator for cleaness of the coding style, but conditional statements are acceptable also. |

* For ease of calculation, it is encouraged to use the java.lang.Math library for mathematical functions.

4.2) Package container
You must implement this package (mostly) from scratch.

4.2.1) Class Neuron:
**You must implement this class from scratch.**
This class *encapsulates* the properties and behavior of a neuron, *a basic building block*, including its weights, bias, output value, and methods for updating weights. It is a fundamental building block used in constructing networks for various advanced tasks.

Field

| Name | Description |
|---|---|
| - double minWeightValue | min weight value of the neuron. This field will be the same as the whole entire class. |
| - double maxWeightValue | max weight value of the neuron. This field will be the same as the whole entire class. |
| - double[] weights | weights of the neuron |

| | |
|---|---|
| - double[] cacheWeights | cache weights of the neuron |
| - double gradient | gradient of the neuron |
| - double bias | bias of the neuron |
| - double value | value of the neuron. |

Constructors

| Name | Description |
|---|---|
| + Neuron(double[] weights, double bias) | Create a new Neuron. Set the given parameter respect to the variable as in fields. In addition, cacheWeights is initialized as weights, and gradient to 0. |
| + Neuron(double value) | Create a new Neuron. Set the given parameter respect to the variable as in fields. Set bias and gradient to -1. |

Method

| Name | Description |
|---|---|
| + void SetRangeWeight(double min, double max) | This method will change the min weight value and max weight value from the given parameter. |
| + void updateWeights() | Set the weight of this neuron object to the same with the cache weight. |
| + getter setter for each appropriate variables | If not given,<br>- weights and cacheWeights are null<br>- bias and gradient are -1<br>- value is 0<br><br>*Note*: min and max weights value will be set according to the static modifier in the latter class. |

4.2.2) Class Layer:

**You must implement this class from scratch.**

This class represents a group of neurons. It provides functionality for initializing neurons with random weights and biases, as well as applying activation functions to the output of neurons within the layer.

Field

| Name | Description |
|---|---|
| - Neuron[] neurons | all neurons of this layer. |

| | |
|---|---|
| - Function function | function of this layer. |

Constructors

| Name | Description |
|---|---|
| + Layer(int inNeurons, int nNeurons, Function function) | Create a new layer.Set function as given parameter. set neurons as a new neuron array with nNeurons length.<br><br>For each neuron in neurons, create a new neuron object that has neuron weights as a double array within inNeurons length. And each weight in that neuron, random the weight randomly(Hint: use GenRandom.randomDouble(Neuron.GetMinWeightValue(),Neuron.GetMaxWeightValue())). The neuron weight will be random between 0 and 1. |
| + Layer(double[] input) | This constructor initializes a layer in a neural network using an array of input values, creating neurons for each input value and assigning them to the layer while leaving the activation function as null. |

Methods

| Name | Description |
|---|---|
| + double applyActivation(double x) | This method applies the specified activation of this function to the given input value, throwing an IllegalArgumentException if the function is unknown. As "Unknown activation function: " + function |
| + double applyActivationDerivative(double x) | This method computes the derivative of the specified activation function with respect to the given input value, throwing an exception if the function is unknown. As "Unknown activation function: " + function |
| + getter setter for each appropriate variables | |

4.2.3) Class Data:

**You must implement this class from scratch.**

This class represents a dataset, storing input data and corresponding outputs as arrays of double values. It provides methods to access and modify these attributes, facilitating management and manipulation of datasets.

Field

| Name | Description |
|---|---|
| - double[] input | The input of the data. |
| - double[] output | The output of the data. |

Constructors

| Name | Description |
|---|---|
| + Data(double[] input, double[] output) | Create a new data object with specified input and output.<br>Set related fields with the given parameters. |

Method

| Name | Description |
|---|---|
| + getter setter for each appropriate variables | |

4.2.4) Class Network

**This class is partially given.**

The Network class encapsulates the functionality for constructing and training a neural network, comprising layers of neurons. It facilitates forward and backward propagation for training, computes loss, and updates weights iteratively. With methods for accessing and modifying network layers and datasets, it serves as a key component in building and training neural networks for tasks like classification and regression.

Field

| Name | Description |
|---|---|
| - static Layer[] layers | Arrays of layers containing arrays of neurons. |
| - static Data[] datasets | Array of datasets, in which contains inputs and expected output (see class Data for the references) |

Constructors

| Name | Description |
|---|---|
| + Network(int[] nLayers, Function[] functions, Data[] data) | Set range of weight in the bound of [-1, 1]<br>Initialize new layer array ***the first element of an array is always null** (it is an input layer, so no neuron in this layer). Hence, the function array received from into this |

| | |
|---|---|
| | constructor is less length from the layer array by one.<br><br>Create new datasets array from the input data type, then assign its value (see how to get the value and set value of the data).<br><br>*Note*, if you are confused by how you should set a layer, please see the Layer class for further insights. |

Method

| Name | Description |
|---|---|
| + void forward(double[] inputs) | Initialization layer 0 as an input layer.<br>Next, the method iterates over each layer in the network (starting from the second layer, as the first layer has already been initialized).<br><br>For each layer, it iterates over each next neuron in that layer. (not include the first layer)<br><br>For each neuron, it calculates the weighted sum of inputs from the previous layer:<br><br>1. Iterates over each neuron in the previous layer (layers[i - 1]) and multiplies the value of that neuron by the corresponding weight connecting it to the current neuron in the current layer (layers[i]). Which accumulates these products.<br>2. Apply the activation function to the calculated sum to compute the output of the neuron.<br>3. Finally sets the output value of the neuron to the computed output.<br><br>End Neuron Loop.<br><br>End Layer Loop.<br><br>*Explanation in the matrix form propagation can be defined with this notation.<br><br>Z[l] = W[l]A[l-1] + b[l]<br><br>In which,<br>1) Z is an output of this layer,<br>2) 2) W is weight of this layer,<br>3) A is a value of Neuron in each layer |

| | |
|---|---|
| | (A[0] = X or input),<br>4) b is a bias of each neuron.<br><br>** ***End.*** Keywords indicate the ending of the looping block.<br><br>*** Please be aware of the indexing used to access the order of the neuron to get the correct value of its weight and value. |
| + void backward(double learningRate, Data tData) | method performs the backpropagation to adjust weights based on calculated gradients.<br><br>Output Layer Backpropagation:<br><br>1. Iterate over each neuron in the output layer.<br>2. Calculate the error derivative (`derivative`) by subtracting the target output from the actual output.<br>3. Compute the delta for each neuron, representing the adjustment needed for each weight, using the derivative and the derivative of the activation function applied to the output.<br>4. Set the delta as the gradient for the neuron.<br>5. Update the weights of the neuron based on the gradient and the learning rate.<br><br>Hidden Layer Backpropagation:<br><br>1. Iterate backward over each hidden layer (from the second-to-last layer to the second layer).<br>2. For each neuron in each hidden layer:<br>a. Compute the delta similarly to the output layer, considering the sum of gradients from neurons in the next layer.<br>b. Update the gradient and weights for each neuron |

| | |
|---|---|
| | based on the computed delta, previous layer outputs, and learning rate.<br><br>Weight Update:<br><br>1. After updating gradients and cache weights for all neurons in all layers, iterate over each neuron in each layer.<br>2. Update the weights of neurons based on the cache weights, effectively applying the calculated adjustments.<br><br>Definition: Functional Components<br><br>a. Error Calculation: Derive the error for each neuron by comparing the predicted output with the target output.<br>b. Delta Calculation: Determine the adjustment (delta) required for each weight based on the error and activation function derivative.<br>c. Weight Update: Adjust the weights of neurons based on the calculated deltas and learning rate.<br><br>The `backward` method allows the neural network to learn from training data by iteratively updating weights to minimize the difference between predicted and actual outputs, thereby improving the network's performance over time.<br><br>**You may implement this own method, if you are up to a challenge. Due to the scope and the complexity, this method is given in the Fragment.java. Have Fun!** |
| + double sumGradient(int nIndex, int lIndex) | This method computes the ***SUM*** of gradients for a specific weight index across neurons in a given layer of a neural network. |

| | |
|---|---|
| | Access the layer specified by `*lIndex*` using the layers array.<br>Iterate over each neuron in the current layer<br>For each neuron, retrieve the weight at index `*nIndex*` and multiply it by the gradient of the neuron.<br>Return the computed gradientSum. |
| + void train(int trainingIterations, double learningRate) | This method iterates over the training dataset for the specified number of iterations.<br>Within each iteration:<br>1. Initializes totalLoss to accumulate the total loss over all data in the dataset.<br>2. loops over each data in the datasets array.<br><br>Each loops of the dataset:<br>a. Performs a forward method using data value of getData.<br>b. Performs a backward pass to update the network's parameters (weights) based on the calculated loss and learning rate.<br>c. Calculates the loss for the by comparing the network's output to the expected output.<br>* Network output is at all the neurons of the last layers.<br>End.<br><br>1. Calculates the average loss across all instances (overall loss to the dataset length)<br>2. Print out the iteration number and average loss.<br>End.<br><br>**Due to the high iterations used to train this network, in the process of making the jar, I recommend commenting out the printing of loss, since IO time affects the time of computation within the network. This is NOT MANDATORY, just recommendations.** |
| + double calculateLoss(Neuron[] outputNeurons, double[] expectedOutput) | Iterate over each output neuron using a for loop.<br>- Retrieve the actual output value from the neuron. |

| | |
|---|---|
| | - Calculate the squared error between the actual output and the expected output using the squaredError function in the given class.<br>- Accumulate the squared error.<br><br>Return the average squared error with the total number of output neurons.<br>(See Metrics class for the utility of this class) |
| + void displayOutputLayer() | Prints a header indicating the start of output functions display, `Output functions:`.<br><br>Iterates over each Data instance in the datasets array using an enhanced for loop.<br>- Performs a forward pass through the neural network using the dataset's input data.<br>- Retrieves and prints the output value of each neuron in the output layer.<br><br>*Note:* By forwarding the data, it is the same as prediction of the input value(s). |
| + getter/setter for each appropriate variable | |

4.3) package main

4.2.2) Class Main: The main class
**This class is given, YOU MUST NOT IMPLEMENT THIS CLASS.**

| Name | Description |
|---|---|
| + static void main(String[] args) | This main class initializes a neural network with specified layer sizes and activation functions, along with training data consisting of input-output pairs for simulating the XOR gate. It then trains the network using the training data and displays the output layer before and after training. In addition, this main class should give more intuitive knowledge on how network initialization should be, please inspect this method carefully.<br><br>Note: you will see a comment section in this method. The commented code asks for user commitment through the IO scanner channel for the start of the training process |

| | |
|---|---|
| | of the network. **You may uncomment the code to ensure the pre-instantiation process works correctly** (before the train function was called), but in submitted code and jar artifact creation. **You must comment on the code, or delete this code section. You will be penalized if the code was left uncommented.** |

* The decision to exclude the supplementary class or package from this document was predicated on its lack of relevance or its deviation from the central objective of this assessment. However, the author encourages further exploration of the additional materials given, as they may provide valuable insights to complete this assignment.

5. Scoring Criteria

Class Activation **12 mark(s)**

- sigmoidTest 2 mark(s)
- sigmoidDerivativeTest 2 mark(s)
- tanhTest 2 mark(s)
- tanhDerivativeTest 2 mark(s)
- reluTest 2 mark(s)
- reluDerivativeTest 2 mark(s)

Class Neuron **20 mark(s)**

- constructorWeightedTest 4 mark(s)
- constructorValueTest 4 mark(s)
- rangeWeightSet 3 mark(s)
- getAndSetWeightsTest 1 mark(s)
- getAndSetCacheWeightsTest 1 mark(s)
- getAndSetBiasTest 1 mark(s)
- getAndSetValueTest 1 mark(s)
- getAndSetGradientTest 1 mark(s)
- constructorEmptyWeightsTest 1 mark(s)
- constructorNegativeBiasTest 1 mark(s)
- constructorNegativeValueTest 1 mark(s)

Class Layer **20 mark(s)**

- testConstructorWithFunction 4 mark(s)
- testConstructorWithInput 4 mark(s)
- testApplyActivationSigmoid 2 mark(s)
- testApplyActivationDerivativeSigmoid 2 mark(s)
- testApplyActivationTanh 2 mark(s)
- testApplyActivationDerivativeTanh 2 mark(s)
- testApplyActivationRelu 2 mark(s)
- testApplyActivationDerivativeRelu 2 mark(s)

Class Data **5 mark(s)**

- testConstructorAndGetters 1 mark(s)
- testSetData 1 mark(s)

- testSetOutput 1 mark(s)
- testSetDataWithNull 1 mark(s)
- testSetOutputWithNull 1 mark(s)

Class Network **28 mark(s)**

- testCustomConstructor 6 mark(s)
- testCustomAdditionalLayerFunctions * 4 mark(s) *
- testDefaultConstructor 4 mark(s)
- testForward 5 mark(s)
- testBackward 0 mark(s) (Optional Task)
- testSumGradient 3 mark(s)
- testTrain 3 mark(s)
- testCalculateLoss 3 mark(s)

* Full Score from Coding Section is **85 mark(s)**

UML of all implemented class **(png file only)** 15 mark(s)

- all Classes must be shown
- class details must be correct

The project must contain .jar file, **OTHERWISE THIS QUESTION WILL NOT BE GRADED**

- the .jar file must have the source code inside
- the .jar file must have the .class files

This question has a **full-score at 100 mark(s)**, which will be **scaled down to 10 mark(s).**

* testCustomAdditionalLayerFunctions. This function will test correctness of assigning function to the corresponding layer but it is applicable to test in Network class, due to additional Layers amount. For ease of convenience, the score will be calculated in the network class.

Appendix

a. XOR gate
   The XOR gate, or exclusive OR gate, is a fundamental building block in digital electronics and computational logic. It performs a logical operation where the output is true (or 1) if the inputs are different, and false (or 0) if the inputs are the same. The truth table for the XOR gate is as follows:

| Input A | Input B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

The XOR gate's behavior is distinctive because it produces a true output only when its inputs are different. This property makes it particularly useful in various applications, including digital circuit design, cryptography, and artificial intelligence.

In digital circuitry, XOR gates are commonly used in arithmetic and logic operations, as well as in constructing more complex circuits such as adders, subtractors, and multiplexers. In cryptography, XOR operations are employed for encryption and decryption processes due to their ability to scramble data securely.

From a computational perspective, the XOR gate poses a unique challenge because it cannot be expressed as a simple combination of other basic gates like AND, OR, and NOT gates. This characteristic makes it a crucial test case for evaluating the capabilities of computational models, including neural networks, in solving complex logical problems.