

ML perceptron sigmoid neuron (P3)

v0.1

Generated by Doxygen 1.13.2

1 ML-sigmoid-neuron	1
1.1 Student	1
1.2 Introduction	1
1.3 Documentation	1
1.4 Installing	1
2 ML-backpropation	3
2.1 Student	3
2.2 Introduction	3
2.3 Documentation	3
2.4 Installing	3
3 Hierarchical Index	5
3.1 Class Hierarchy	5
4 Class Index	7
4.1 Class List	7
5 File Index	9
5.1 File List	9
6 Class Documentation	11
6.1 digitData< T > Struct Template Reference	11
6.1.1 Detailed Description	11
6.1.2 Member Data Documentation	11
6.1.2.1 images	11
6.1.2.2 targets	12
6.2 irisData Struct Reference	12
6.2.1 Detailed Description	12
6.2.2 Member Data Documentation	12
6.2.2.1 features	12
6.2.2.2 targets	13
6.3 Neuron Class Reference	13
6.3.1 Detailed Description	14
6.3.2 Constructor & Destructor Documentation	14
6.3.2.1 Neuron() [1/2]	14
6.3.2.2 Neuron() [2/2]	14
6.3.3 Member Function Documentation	15
6.3.3.1 __str__()	15
6.3.3.2 activate()	15
6.3.3.3 computeHiddenDelta()	15
6.3.3.4 computeOutputDelta()	15
6.3.3.5 deltaError()	16
6.3.3.6 getBias()	16

6.3.3.7	getError()	16
6.3.3.8	getWeights()	17
6.3.3.9	predict()	17
6.3.3.10	sigmoid()	17
6.3.3.11	sigmoidDerivative()	17
6.3.3.12	update()	18
6.4	NeuronLayer Class Reference	18
6.4.1	Detailed Description	19
6.4.2	Constructor & Destructor Documentation	19
6.4.2.1	NeuronLayer() [1/2]	19
6.4.2.2	NeuronLayer() [2/2]	19
6.4.3	Member Function Documentation	19
6.4.3.1	__str__()	19
6.4.3.2	computeHiddenErrors()	19
6.4.3.3	computeOutputErros()	20
6.4.3.4	feedForward()	20
6.4.3.5	getNeurons()	20
6.4.3.6	getOutput()	21
6.4.3.7	update()	21
6.5	NeuronNetwork Class Reference	21
6.5.1	Detailed Description	22
6.5.2	Constructor & Destructor Documentation	22
6.5.2.1	NeuronNetwork() [1/2]	22
6.5.2.2	NeuronNetwork() [2/2]	22
6.5.3	Member Function Documentation	22
6.5.3.1	__str__()	22
6.5.3.2	backPropagation()	23
6.5.3.3	feedForward()	23
6.5.3.4	getLayers()	23
6.5.3.5	maskTarget()	23
6.5.3.6	predict()	23
6.5.3.7	setTarget()	24
6.5.3.8	trainInputs2D()	24
6.5.3.9	update()	24
6.6	OutputNueron Class Reference	25
6.6.1	Detailed Description	25
6.7	TrainTestSplit< T > Struct Template Reference	25
6.7.1	Detailed Description	26
6.7.2	Constructor & Destructor Documentation	26
6.7.2.1	TrainTestSplit()	26
6.7.3	Member Data Documentation	26
6.7.3.1	testFeatures	26

6.7.3.2 testTargets	27
6.7.3.3 trainFeatures	27
6.7.3.4 trainTargets	27
7 File Documentation	29
7.1 create_dataset.py File Reference	29
7.1.1 Detailed Description	29
7.1.2 Function Documentation	29
7.1.2.1 save_digits_dataset()	29
7.1.2.2 write_to_csv()	29
7.2 create_dataset.py	30
7.3 common.hpp File Reference	30
7.3.1 Detailed Description	32
7.3.2 Macro Definition Documentation	32
7.3.2.1 INITIAL_BIAS	32
7.3.2.2 INITIAL_BIAS_INPUTN	32
7.3.2.3 INITIAL_WEIGHT	32
7.3.2.4 INITIAL_WEIGHT_INPUTN	32
7.3.3 Function Documentation	32
7.3.3.1 createTrainTestSplit()	32
7.3.3.2 deltaBias()	33
7.3.3.3 deltaGradient()	33
7.3.3.4 gradientBetweenNeurons()	33
7.3.3.5 normalize2DVector()	34
7.3.3.6 normalizeVector()	34
7.3.3.7 printVector()	34
7.4 common.hpp	35
7.5 csv_reader.hpp File Reference	37
7.5.1 Detailed Description	38
7.5.2 Function Documentation	38
7.5.2.1 convert()	38
7.5.2.2 convert< float >()	39
7.5.2.3 convert< int >()	39
7.5.2.4 create2DVector()	39
7.5.2.5 filterData()	39
7.5.2.6 getFeatures()	40
7.5.2.7 getTargets()	40
7.5.2.8 maskData()	40
7.5.2.9 read_csv()	40
7.5.2.10 readDigitData()	41
7.5.2.11 readCsvFlat()	41
7.6 csv_reader.hpp	42

7.7 neuron.hpp File Reference	44
7.7.1 Detailed Description	45
7.8 neuron.hpp	45
7.9 neuronLayer.hpp File Reference	46
7.9.1 Detailed Description	46
7.10 neuronLayer.hpp	47
7.11 neuronNetwork.hpp File Reference	47
7.11.1 Detailed Description	48
7.12 neuronNetwork.hpp	49
7.13 outputNeuron.hpp File Reference	49
7.13.1 Detailed Description	50
7.14 outputNeuron.hpp	51
7.15 neuron.cpp File Reference	51
7.15.1 Detailed Description	51
7.16 neuron.cpp	52
7.17 neuronLayer.cpp File Reference	53
7.17.1 Detailed Description	53
7.18 neuronLayer.cpp	54
7.19 neuronNetwork.cpp File Reference	55
7.19.1 Detailed Description	55
7.20 neuronNetwork.cpp	56
7.21 test.cpp File Reference	57
7.21.1 Detailed Description	58
7.21.2 Macro Definition Documentation	59
7.21.2.1 CATCH_CONFIG_MAIN	59
7.21.3 Function Documentation	59
7.21.3.1 TEST_CASE() [1/5]	59
7.21.3.2 TEST_CASE() [2/5]	60
7.21.3.3 TEST_CASE() [3/5]	60
7.21.3.4 TEST_CASE() [4/5]	61
7.21.3.5 TEST_CASE() [5/5]	61
7.22 test.cpp	61
7.23 testBackpropagation.cpp	62

Chapter 1

ML-sigmoid-neuron

1.1 Student

Name: Stan Merlijn

Student nummer: 1863967

1.2 Introduction

In this repository, we will implement and test a [Neuron](#) using the sigmoid function. This will be demonstrated by creating AND, OR, NOT, NOR gates aswell as an half adder. You can find the assignment [here](#).

1.3 Documentation

For this assignment, the documentation was generated with Doxygen. The LaTeX documentation is available [here](#) and, to view the HTML documentation locally, open [index.html](#) in a browser.

1.4 Installing

Enter the test dir then

Generate build files:

```
cmake -S . -B build
```

Build the project:

```
cmake --build build
```

Run the executable:

```
./build/MLNeuronTest
```


Chapter 2

ML-backpropagation

2.1 Student

Name: Stan Merlijn

Student nummer: 1863967

2.2 Introduction

In this repository, we implement and test the backpropagation algorithm for training a neural network. Building upon a previously implemented neuron network, the project extends the model by incorporating error calculations, gradient computations, and simultaneous weight and bias updates using backpropagation. The network is trained using an online training approach and evaluated through various tasks including learning AND and XOR gates, constructing a half adder, and classifying both the Iris and Digit datasets. Performance is assessed by measuring classification accuracy and training efficiency. You can find the assignment [here](#)

2.3 Documentation

For this assignment, documentation was generated using Doxygen. The LaTeX documentation can be found [here](#) and if you want to run the HTML local website, you can open the [index.html](#) in a browser.

2.4 Installing

Enter the test dir then

Generate build files:

```
cmake -S . -B build
```

Build the project:

```
cmake --build build
```

For enhanced build performance, it's recommended to compile in parallel using:

```
cmake --build build --parallel <n threads>
```

Run the executable:

```
./build/MLPerceptronTest
```


Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

digitData< T >	11
irisData	12
Neuron	13
OutputNueron	25
NeuronLayer	18
NeuronNetwork	21
TrainTestSplit< T >	25

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

digitData< T >	A structure to hold the features and targets read from a CSV file. This is for the digit data set .	11
irisData	A structure to hold the features and targets read from a CSV file. This is for the iris data set .	12
Neuron	Represents a single neuron in a neural network	13
NeuronLayer	Represents a layer of neurons in a neural network	18
NeuronNetwork	Represents a neural network with multiple layers of neurons	21
OutputNeuron	25
TrainTestSplit< T >	A structure to hold the features and targets read from a CSV file. This is for the digit data set .	25

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

create_dataset.py	Module for creating and saving the digits dataset	29
common.hpp	In this file the common utilities are defined	30
csv_reader.hpp	In this class the CSV reader is defined. This is for reading the iris data set	37
neuron.hpp	In this file the Neuron class is declared. This class represents a single neuron in a neural network	44
neuronLayer.hpp	In this file the NeuronLayer class is declared. This class represents a layer of neurons in a neural network	46
neuronNetwork.hpp	In this file the NeuronNetwork class is declared. This class represents a neural network with multiple layers of neurons	47
outputNeuron.hpp	In this file the OutputNeuron class is declared. This class represents a single output neuron in a neural network	49
neuron.cpp	In this file the Neuron class is implemented	51
neuronLayer.cpp	In this file the NeuronLayer class is implemented	53
neuronNetwork.cpp	In this file the NeuronNetwork class is implemented	55
test.cpp	In this file the tests for the Neuron , NeuronLayer and NeuronNetwork classes are implemented	57
testBackpropagation.cpp	62

Chapter 6

Class Documentation

6.1 `digitData< T >` Struct Template Reference

A structure to hold the features and targets read from a CSV file. This is for the digit data set.

```
#include <common.hpp>
```

Public Attributes

- `std::vector< T > images`
- `std::vector< T > targets`

6.1.1 Detailed Description

```
template<typename T>  
struct digitData< T >
```

A structure to hold the features and targets read from a CSV file. This is for the digit data set.

This structure contains two members:

- `images`: A Vector of ints where each 64 elements represent an image of a digit(8x8 image).
- `targets`: A vector of integers where each element represents the target value corresponding to the features.

Definition at line 47 of file [common.hpp](#).

6.1.2 Member Data Documentation

6.1.2.1 `images`

```
template<typename T>  
std::vector<T> digitData< T >::images
```

Definition at line 49 of file [common.hpp](#).

6.1.2.2 targets

```
template<typename T>
std::vector<T> digitData< T >::targets
```

Definition at line 50 of file [common.hpp](#).

The documentation for this struct was generated from the following file:

- [common.hpp](#)

6.2 irisData Struct Reference

A structure to hold the features and targets read from a CSV file. This is for the iris data set.

```
#include <common.hpp>
```

Public Attributes

- `std::vector< std::vector< float > >` [features](#)
- `std::vector< float >` [targets](#)

6.2.1 Detailed Description

A structure to hold the features and targets read from a CSV file. This is for the iris data set.

This structure contains two members:

- `features`: A 2D vector of floats where each inner vector represents a set of features for a single data point.
- `targets`: A vector of integers where each element represents the target value corresponding to the features.

Definition at line 33 of file [common.hpp](#).

6.2.2 Member Data Documentation

6.2.2.1 features

```
std::vector<std::vector<float> > irisData::features
```

Definition at line 35 of file [common.hpp](#).

6.2.2.2 targets

```
std::vector<float> irisData::targets
```

Definition at line 36 of file [common.hpp](#).

The documentation for this struct was generated from the following file:

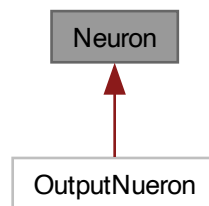
- [common.hpp](#)

6.3 Neuron Class Reference

Represents a single neuron in a neural network.

```
#include <neuron.hpp>
```

Inheritance diagram for Neuron:



Public Member Functions

- [Neuron](#) (int nSizeWeights, float initialWeight, float initialBias)
constructor [Neuron](#) object.
- [Neuron](#) (const std::vector< float > &weights, float bias, float learningRate=0.1)
Constructs a [Neuron](#) with the given weights and bias.
- float [sigmoid](#) (float x)
Computes the sigmoid activation function.
- float [activate](#) (const std::vector< float > &inputs)
Performs an activation operation.
- float [predict](#) (const std::vector< float > &inputs)
Performs a feedforward operation.
- void [deltaError](#) (const std::vector< float > &inputs, const std::vector< [Neuron](#) > &neuronsNextLayer, float target, bool isOutputNeuron)
Calculates errors for the weights and bias of the neuron.
- void [update](#) ()
Updates the weights and bias of the neuron. Using the previously calculated errors.
- float [computeHiddenDelta](#) (const std::vector< float > &inputs, float sum)

- Calculates the error for the output layer.*
 - float `sigmoidDerivative` (float output)
- Calculates the error for the hidden layer.*
 - float `computeOutputDelta` (float target)
- Calculates the error for the output layer.*
 - const std::vector< float > & `getWeights` () const
- Returns the weights of the neuron.*
 - float `getBias` () const
- Returns the bias of the neuron.*
 - float `getError` () const
- Returns the error of the neuron.*
 - void `__str__` () const
- Prints the neuron details.*

6.3.1 Detailed Description

Represents a single neuron in a neural network.

This class models a neuron with a set of weights and a bias. It provides methods to compute the sigmoid activation function and to perform a feedforward operation given a set of inputs.

Definition at line 21 of file [neuron.hpp](#).

6.3.2 Constructor & Destructor Documentation

6.3.2.1 Neuron() [1/2]

```
Neuron::Neuron (
    int nSizeWeights,
    float initialWeight,
    float initialBias)
```

constructor [Neuron](#) object.

Definition at line 13 of file [neuron.cpp](#).

6.3.2.2 Neuron() [2/2]

```
Neuron::Neuron (
    const std::vector< float > & weights,
    float bias,
    float learningRate = 0.1)
```

Constructs a [Neuron](#) with the given weights and bias.

Parameters

<i>weights</i>	A vector of weights for the neuron.
<i>bias</i>	The bias term for the neuron.

Definition at line 26 of file [neuron.cpp](#).

6.3.3 Member Function Documentation

6.3.3.1 `__str__()`

```
void Neuron::__str__ () const
```

Prints the neuron details.

Definition at line 86 of file [neuron.cpp](#).

6.3.3.2 `activate()`

```
float Neuron::activate (  
    const std::vector< float > & inputs)
```

Performs an activation operation.

Parameters

<i>inputs</i>	A vector of input values.
---------------	---------------------------

Returns

The output of the neuron after applying the weights, bias, and activation function.

Definition at line 35 of file [neuron.cpp](#).

6.3.3.3 `computeHiddenDelta()`

```
float Neuron::computeHiddenDelta (  
    const std::vector< float > & inputs,  
    float sum)
```

Calculates the error for the output layer.

Parameters

<i>output</i>	The output of the neuron.
<i>target</i>	The target value.

Returns

The error for the output layer.

Definition at line 69 of file [neuron.cpp](#).

6.3.3.4 `computeOutputDelta()`

```
float Neuron::computeOutputDelta (  
    float target)
```

Calculates the error for the output layer.

Parameters

<i>output</i>	The output of the neuron.
<i>target</i>	The target value.

Returns

The error for the output layer.

Definition at line 75 of file [neuron.cpp](#).

6.3.3.5 deltaError()

```
void Neuron::deltaError (  
    const std::vector< float > & inputs,  
    const std::vector< Neuron > & neuronsNextLayer,  
    float target,  
    bool isOutputNeuron)
```

Calculates errors for the weights and bias of the neuron.

Parameters

<i>inputs</i>	A vector of input values.
<i>target</i>	The target value.

6.3.3.6 getBias()

```
float Neuron::getBias () const [inline]
```

Returns the bias of the neuron.

Returns

The bias value.

Definition at line 111 of file [neuron.hpp](#).

6.3.3.7 getError()

```
float Neuron::getError () const [inline]
```

Returns the error of the neuron.

Returns

The error value.

Definition at line 117 of file [neuron.hpp](#).

6.3.3.8 getWeights()

```
const std::vector< float > & Neuron::getWeights () const [inline]
```

Returns the weights of the neuron.

Returns

A vector of weights.

Definition at line 105 of file [neuron.hpp](#).

6.3.3.9 predict()

```
float Neuron::predict (  
    const std::vector< float > & inputs)
```

Performs a feedforward operation.

Parameters

<i>inputs</i>	A vector of input values.
---------------	---------------------------

Returns

The output of the neuron after applying the weights, bias, and activation function.

Definition at line 52 of file [neuron.cpp](#).

6.3.3.10 sigmoid()

```
float Neuron::sigmoid (  
    float x)
```

Computes the sigmoid activation function.

Parameters

<i>x</i>	The input value.
----------	------------------

Returns

The result of the sigmoid function applied to x.

Definition at line 29 of file [neuron.cpp](#).

6.3.3.11 sigmoidDerivative()

```
float Neuron::sigmoidDerivative (  
    float output)
```

Calculates the error for the hidden layer.

Parameters

<i>output</i>	The output of the neuron.
<i>target</i>	The target value.

Returns

The error for the hidden layer.

Definition at line 81 of file [neuron.cpp](#).

6.3.3.12 update()

```
void Neuron::update ()
```

Updates the weights and bias of the neuron. Using the previously calculated errors.

Definition at line 58 of file [neuron.cpp](#).

The documentation for this class was generated from the following files:

- [neuron.hpp](#)
- [neuron.cpp](#)

6.4 NeuronLayer Class Reference

Represents a layer of neurons in a neural network.

```
#include <neuronLayer.hpp>
```

Public Member Functions

- [NeuronLayer](#) (std::vector< [Neuron](#) > neurons)
Constructs a [NeuronLayer](#) with the given neurons.
- [NeuronLayer](#) (int nNeurons, int nSizeWeights)
Constructs a [NeuronLayer](#) with the given number of neurons and size of weights.
- std::vector< float > & [feedForward](#) (const std::vector< float > &inputs)
Performs a feedforward operation.
- void [computeOutputErrors](#) (const std::vector< float > &targets)
Computes the output errors for the layer.
- void [computeHiddenErrors](#) (const std::vector< float > &inputs, const std::vector< [Neuron](#) > &neurons, [NextLayer](#))
Computes the hidden errors for the layer.
- void [update](#) ()
Updates the neurons in the layer.
- const std::vector< [Neuron](#) > & [getNeurons](#) ()
Returns the neurons in the layer.
- const std::vector< float > & [getOutput](#) () const
Returns the output of the layer.
- void [__str__](#) () const
Prints the layer details.

6.4.1 Detailed Description

Represents a layer of neurons in a neural network.

The [NeuronLayer](#) class has a collection of neurons and provides methods to perform feedforward operations and to represent the layer as a string.

Definition at line 23 of file [neuronLayer.hpp](#).

6.4.2 Constructor & Destructor Documentation

6.4.2.1 NeuronLayer() [1/2]

```
NeuronLayer::NeuronLayer (
    std::vector< Neuron > neurons)
```

Constructs a [NeuronLayer](#) with the given neurons.

Parameters

<i>neurons</i>	A vector of neurons for the layer.
----------------	------------------------------------

Definition at line 13 of file [neuronLayer.cpp](#).

6.4.2.2 NeuronLayer() [2/2]

```
NeuronLayer::NeuronLayer (
    int nNeurons,
    int nSizeWeights)
```

Constructs a [NeuronLayer](#) with the given number of neurons and size of weights.

Parameters

<i>nNeurons</i>	The number of neurons in the layer.
<i>nSizeWeights</i>	The size of the weights for each neuron.

Definition at line 16 of file [neuronLayer.cpp](#).

6.4.3 Member Function Documentation

6.4.3.1 __str__()

```
void NeuronLayer::__str__ () const
```

Prints the layer details.

Definition at line 81 of file [neuronLayer.cpp](#).

6.4.3.2 computeHiddenErrors()

```
void NeuronLayer::computeHiddenErrors (
    const std::vector< float > & inputs,
    const std::vector< Neuron > & neuronsNextLayer)
```

Computes the hidden errors for the layer.

Parameters

<i>inputs</i>	A vector of input values.
<i>neuronsNextLayer</i>	A vector of neurons in the next layer.

Definition at line 56 of file [neuronLayer.cpp](#).

6.4.3.3 computeOutputErros()

```
void NeuronLayer::computeOutputErros (  
    const std::vector< float > & targets)
```

Computes the output errors for the layer.

Parameters

<i>targets</i>	A vector of target values.
----------------	----------------------------

Definition at line 48 of file [neuronLayer.cpp](#).

6.4.3.4 feedForward()

```
std::vector< float > & NeuronLayer::feedForward (  
    const std::vector< float > & inputs)
```

Performs a feedforward operation.

Parameters

<i>inputs</i>	A vector of input values.
---------------	---------------------------

Returns

The output of the layer.

Definition at line 36 of file [neuronLayer.cpp](#).

6.4.3.5 getNeurons()

```
const std::vector< Neuron > & NeuronLayer::getNeurons () [inline]
```

Returns the neurons in the layer.

Returns

A vector of neurons.

Definition at line 71 of file [neuronLayer.hpp](#).

6.4.3.6 getOutput()

```
const std::vector< float > & NeuronLayer::getOutput () const [inline]
```

Returns the output of the layer.

Returns

A vector of floats.

Definition at line 77 of file [neuronLayer.hpp](#).

6.4.3.7 update()

```
void NeuronLayer::update ()
```

Updates the neurons in the layer.

Definition at line 73 of file [neuronLayer.cpp](#).

The documentation for this class was generated from the following files:

- [neuronLayer.hpp](#)
- [neuronLayer.cpp](#)

6.5 NeuronNetwork Class Reference

Represents a neural network with multiple layers of neurons.

```
#include <neuronNetwork.hpp>
```

Public Member Functions

- [NeuronNetwork](#) (std::vector< [NeuronLayer](#) > layers)
Constructs a [NeuronNetwork](#) with the given layers.
- [NeuronNetwork](#) (std::vector< int > layers)
Performs a feedforward operation. On all the layers sequentially.
- const std::vector< float > & [feedForward](#) (const std::vector< float > &inputs)
Performs a feedforward operation. On all the layers sequentially.
- std::vector< float > [predict](#) (const std::vector< float > &input)
Performs a prediction operation. On all the layers sequentially.
- void [backPropagation](#) (const std::vector< float > &targets)
Performs a backpropagation operation. On all the layers sequentially.
- void [maskTarget](#) (float target)
- void [update](#) ()
Updates the neurons in the network.
- void [trainInputs2D](#) (const std::vector< std::vector< float > > &inputs, const std::vector< std::vector< float > > &targets, int epochs)
Trains the network on a set of inputs and targets.
- std::vector< [NeuronLayer](#) > [getLayers](#) () const
Returns the layers in the network.
- void [setTarget](#) (std::vector< float > &targets)
Sets the target values in the network.
- void [__str__](#) () const
Prints the network details.

6.5.1 Detailed Description

Represents a neural network with multiple layers of neurons.

The [NeuronNetwork](#) class has a collection of neuron layers and provides methods to perform feedforward operations and to represent the network as a string.

Definition at line 22 of file [neuronNetwork.hpp](#).

6.5.2 Constructor & Destructor Documentation

6.5.2.1 NeuronNetwork() [1/2]

```
NeuronNetwork::NeuronNetwork (
    std::vector< NeuronLayer > layers)
```

Constructs a [NeuronNetwork](#) with the given layers.

Parameters

<i>layers</i>	A vector of neuron layers for the network.
---------------	--

Definition at line 13 of file [neuronNetwork.cpp](#).

6.5.2.2 NeuronNetwork() [2/2]

```
NeuronNetwork::NeuronNetwork (
    std::vector< int > layers)
```

Performs a feedforward operation. On all the layers sequentially.

Parameters

<i>inputs</i>	A vector of input values.
---------------	---------------------------

Returns

The output of the network.

Definition at line 16 of file [neuronNetwork.cpp](#).

6.5.3 Member Function Documentation

6.5.3.1 __str__()

```
void NeuronNetwork::__str__ () const
```

Prints the network details.

Definition at line 153 of file [neuronNetwork.cpp](#).

6.5.3.2 backPropagation()

```
void NeuronNetwork::backPropagation (
    const std::vector< float > & targets)
```

Performs a backpropagation operation. On all the layers sequentially.

Definition at line 57 of file [neuronNetwork.cpp](#).

6.5.3.3 feedForward()

```
const std::vector< float > & NeuronNetwork::feedForward (
    const std::vector< float > & inputs)
```

Performs a feedforward operation. On all the layers sequentially.

Parameters

<i>inputs</i>	A vector of input values.
---------------	---------------------------

Returns

The output of the network.

Definition at line 39 of file [neuronNetwork.cpp](#).

6.5.3.4 getLayers()

```
std::vector< NeuronLayer > NeuronNetwork::getLayers () const [inline]
```

Returns the layers in the network.

Returns

A vector of neuron layers.

Definition at line 84 of file [neuronNetwork.hpp](#).

6.5.3.5 maskTarget()

```
void NeuronNetwork::maskTarget (
    float target)
```

Definition at line 135 of file [neuronNetwork.cpp](#).

6.5.3.6 predict()

```
std::vector< float > NeuronNetwork::predict (
    const std::vector< float > & input)
```

Performs a prediction operation. On all the layers sequentially.

Parameters

<i>inputs</i>	A vector of input values.
---------------	---------------------------

Returns

The output of the network.

Definition at line 52 of file [neuronNetwork.cpp](#).

6.5.3.7 setTarget()

```
void NeuronNetwork::setTarget (
    std::vector< float > & targets) [inline]
```

Sets the target values in the network.

Parameters

<i>targets</i>	The target values to set.
----------------	---------------------------

Definition at line 91 of file [neuronNetwork.hpp](#).

6.5.3.8 trainInputs2D()

```
void NeuronNetwork::trainInputs2D (
    const std::vector< std::vector< float > > & inputs,
    const std::vector< std::vector< float > > & targets,
    int epochs)
```

Trains the network on a set of inputs and targets.

Parameters

<i>inputs</i>	A vector of input values.
<i>targets</i>	A vector of target values.
<i>inputSize</i>	The size of the input values.
<i>maxTrainingSamples</i>	The maximum number of training samples.

Definition at line 81 of file [neuronNetwork.cpp](#).

6.5.3.9 update()

```
void NeuronNetwork::update ()
```

Updates the neurons in the network.

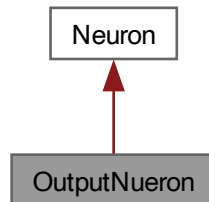
Definition at line 74 of file [neuronNetwork.cpp](#).

The documentation for this class was generated from the following files:

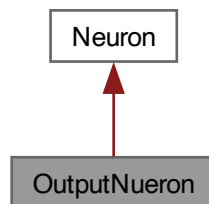
- [neuronNetwork.hpp](#)
- [neuronNetwork.cpp](#)

6.6 OutputNueron Class Reference

Inheritance diagram for OutputNueron:



Collaboration diagram for OutputNueron:



6.6.1 Detailed Description

Definition at line 15 of file [outputNeuron.hpp](#).

The documentation for this class was generated from the following file:

- [outputNeuron.hpp](#)

6.7 TrainTestSplit< T > Struct Template Reference

A structure to hold the features and targets read from a CSV file. This is for the digit data set.

```
#include <common.hpp>
```

Public Member Functions

- [TrainTestSplit](#) (const std::vector< std::vector< T > > &features, const std::vector< std::vector< T > > &targets, float splitRatio)

Public Attributes

- std::vector< std::vector< T > > [trainFeatures](#)
- std::vector< std::vector< T > > [testFeatures](#)
- std::vector< std::vector< T > > [trainTargets](#)
- std::vector< std::vector< T > > [testTargets](#)

6.7.1 Detailed Description

```
template<typename T>
struct TrainTestSplit< T >
```

A structure to hold the features and targets read from a CSV file. This is for the digit data set.

This structure contains two members:

- trainFeatures: A 2D vector of floats where each inner vector represents a set of features for a single data point.
- targets: A vector of integers where each element represents the target value corresponding to the features.

Definition at line 72 of file [common.hpp](#).

6.7.2 Constructor & Destructor Documentation

6.7.2.1 TrainTestSplit()

```
template<typename T>
TrainTestSplit< T >::TrainTestSplit (
    const std::vector< std::vector< T > > & features,
    const std::vector< std::vector< T > > & targets,
    float splitRatio) [inline]
```

Definition at line 76 of file [common.hpp](#).

6.7.3 Member Data Documentation

6.7.3.1 testFeatures

```
template<typename T>
std::vector<std::vector<T> > TrainTestSplit< T >::testFeatures
```

Definition at line 88 of file [common.hpp](#).

6.7.3.2 testTargets

```
template<typename T>
std::vector<std::vector<T> > TrainTestSplit< T >::testTargets
```

Definition at line 90 of file [common.hpp](#).

6.7.3.3 trainFeatures

```
template<typename T>
std::vector<std::vector<T> > TrainTestSplit< T >::trainFeatures
```

Definition at line 87 of file [common.hpp](#).

6.7.3.4 trainTargets

```
template<typename T>
std::vector<std::vector<T> > TrainTestSplit< T >::trainTargets
```

Definition at line 89 of file [common.hpp](#).

The documentation for this struct was generated from the following file:

- [common.hpp](#)

Chapter 7

File Documentation

7.1 `create_dataset.py` File Reference

Module for creating and saving the digits dataset.

Functions

- `create_dataset.write_to_csv` (np.ndarray data, str filename)
Writes a NumPy array to a CSV file.
- `create_dataset.save_digits_dataset` ()
Loads the digits dataset and saves image and target data into CSV files.

7.1.1 Detailed Description

Module for creating and saving the digits dataset.

This module loads the digits dataset from scikit-learn, converts the images and targets to integers, and writes them to CSV files using a helper function.

Definition in file `create_dataset.py`.

7.1.2 Function Documentation

7.1.2.1 `save_digits_dataset()`

```
create_dataset.save_digits_dataset ()
```

Loads the digits dataset and saves image and target data into CSV files.

This function loads the digits dataset from scikit-learn, converts the images and targets to integer type, and then writes them to "digits_images.csv" and "digits_targets.csv" files, respectively.

Definition at line 25 of file `create_dataset.py`.

7.1.2.2 `write_to_csv()`

```
create_dataset.write_to_csv (  
    np.ndarray data,  
    str filename)
```

Writes a NumPy array to a CSV file.

Parameters

<i>data</i>	A numpy array containing the data to write.
<i>filename</i>	The name of the output CSV file.

Definition at line 18 of file [create_dataset.py](#).

7.2 create_dataset.py

[Go to the documentation of this file.](#)

```

00001 #!/usr/bin/env python3
00002
00003
00008 import matplotlib.pyplot as plt
00009 import numpy as np
00010 import csv
00011
00012 # Import datasets, classifiers and performance metrics
00013 from sklearn import datasets, metrics, svm
00014
00015
00018 def write_to_csv(data: np.ndarray, filename: str):
00019     np.savetxt(filename, data, delimiter=",", fmt="%d")
00020
00021
00025 def save_digits_dataset():
00026     # Get the digits dataset
00027     data = datasets.load_digits()
00028     images = data.data
00029     targets = data.target
00030
00031     # Convert the image and target data to int type
00032     images = images.astype(int)
00033     targets = targets.astype(int)
00034
00035     # Save the images and the targets to CSV files
00036     write_to_csv(images, "digits_images.csv")
00037     write_to_csv(targets, "digits_targets.csv")
00038
00039
00040 if __name__ == "__main__":
00041     save_digits_dataset()

```

7.3 common.hpp File Reference

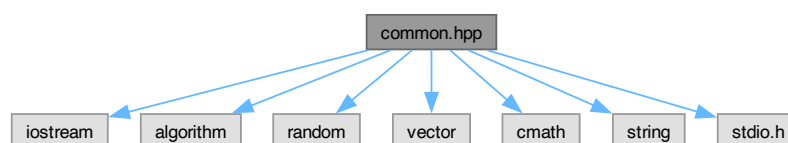
In this file the common utilities are defined.

```

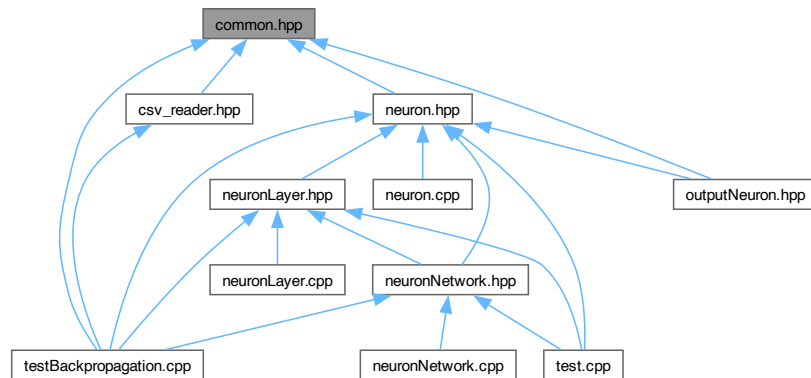
#include <iostream>
#include <algorithm>
#include <random>
#include <vector>
#include <cmath>
#include <string>
#include <stdio.h>

```

Include dependency graph for common.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- struct [irisData](#)
A structure to hold the features and targets read from a CSV file. This is for the iris data set.
- struct [digitData< T >](#)
A structure to hold the features and targets read from a CSV file. This is for the digit data set.
- struct [TrainTestSplit< T >](#)
A structure to hold the features and targets read from a CSV file. This is for the digit data set.

Macros

- `#define INITIAL_WEIGHT 0.5f`
- `#define INITIAL_WEIGHT_INPUTN 0.5f`
- `#define INITIAL_BIAS 0.5f`
- `#define INITIAL_BIAS_INPUTN 0.5f`

Functions

- `template<typename T>`
`TrainTestSplit< T > createTrainTestSplit (const std::vector< std::vector< T > > &features, const std::vector< std::vector< T > > &targets, float splitRatio)`
Create a Train Test Split object from the features and targets.
- `float gradientBetweenNeurons (float &output, float &error)`
This function calculates the gradient between two neurons.
- `float deltaGradient (float &learningRate, float &gradient)`
This function calculates the delta gradient for a neuron.
- `float deltaBias (float &learningRate, float &deltaGradient)`
This function calculates the delta bias for a neuron.
- `template<typename T>`
`void printVector (const std::vector< T > &vec, const std::string extra="")`
This function prints the elements of a vector to the console.
- `template<typename T>`
`void normalizeVector (std::vector< T > &vec)`
This function creates a 2D vector from a 1D vector.
- `template<typename T>`
`void normalize2DVector (std::vector< std::vector< T > > &vec)`
This function creates a 2D vector from a 1D vector.

7.3.1 Detailed Description

In this file the common utilities are defined.

Author

Stan Merlijn

Version

0.1

Date

2025-02-22

Copyright

Copyright (c) 2025

Definition in file [common.hpp](#).

7.3.2 Macro Definition Documentation

7.3.2.1 INITIAL_BIAS

```
#define INITIAL_BIAS 0.5f
```

Definition at line 23 of file [common.hpp](#).

7.3.2.2 INITIAL_BIAS_INPUTN

```
#define INITIAL_BIAS_INPUTN 0.5f
```

Definition at line 24 of file [common.hpp](#).

7.3.2.3 INITIAL_WEIGHT

```
#define INITIAL_WEIGHT 0.5f
```

Definition at line 21 of file [common.hpp](#).

7.3.2.4 INITIAL_WEIGHT_INPUTN

```
#define INITIAL_WEIGHT_INPUTN 0.5f
```

Definition at line 22 of file [common.hpp](#).

7.3.3 Function Documentation

7.3.3.1 createTrainTestSplit()

```
template<typename T>
TrainTestSplit< T > createTrainTestSplit (
    const std::vector< std::vector< T > > & features,
    const std::vector< std::vector< T > > & targets,
    float splitRatio)
```

Create a Train Test Split object from the features and targets.

Parameters

<i>features</i>	The 2D vector of features to split.
<i>targets</i>	The 2D vector of targets to split.
<i>splitRatio</i>	The ratio to split the data.

Returns

[TrainTestSplit<T>](#) The [TrainTestSplit](#) object.

Definition at line 219 of file [common.hpp](#).

7.3.3.2 deltaBias()

```
float deltaBias (  
    float & learningRate,  
    float & deltaGradient) [inline]
```

This function calculates the delta bias for a neuron.

Parameters

<i>learningRate</i>	The learning rate of the network.
<i>deltaGradient</i>	The gradient of the neuron.

Returns

float The delta bias.

Definition at line 124 of file [common.hpp](#).

7.3.3.3 deltaGradient()

```
float deltaGradient (  
    float & learningRate,  
    float & gradient) [inline]
```

This function calculates the delta gradient for a neuron.

Parameters

<i>learningRate</i>	The learning rate of the network.
<i>gradient</i>	The gradient of the neuron.

Returns

float The delta gradient.

Definition at line 112 of file [common.hpp](#).

7.3.3.4 gradientBetweenNeurons()

```
float gradientBetweenNeurons (  
    float & output,  
    float & error) [inline]
```

This function calculates the gradient between two neurons.

Parameters

<i>output</i>	The output of the neuron.
<i>error</i>	The error of the neuron.

Returns

float The gradient between the neurons.

Definition at line 100 of file [common.hpp](#).

7.3.3.5 normalize2DVector()

```
template<typename T>
void normalize2DVector (
    std::vector< std::vector< T > > & vec)
```

This function creates a 2D vector from a 1D vector.

Parameters

<i>vec</i>	The 1D vector to be converted.
<i>size</i>	The size of the inner vectors.

Returns

std::vector<std::vector<T>> The 2D vector.

Definition at line 182 of file [common.hpp](#).

7.3.3.6 normalizeVector()

```
template<typename T>
void normalizeVector (
    std::vector< T > & vec)
```

This function creates a 2D vector from a 1D vector.

Parameters

<i>vec</i>	The 1D vector to be converted.
<i>size</i>	The size of the inner vectors.

Returns

std::vector<std::vector<T>> The 2D vector.

Definition at line 161 of file [common.hpp](#).

7.3.3.7 printVector()

```
template<typename T>
void printVector (
    const std::vector< T > & vec,
    const std::string extra = "") [inline]
```

This function prints the elements of a vector to the console.

Parameters

vec	Vector to be printed.
-----	-----------------------

Returns

* template<typename T>

Definition at line 137 of file [common.hpp](#).

7.4 common.hpp

[Go to the documentation of this file.](#)

```

00001
00011
00012 #pragma once
00013 #include <iostream>
00014 #include <algorithm>
00015 #include <random>
00016 #include <vector>
00017 #include <cmath>
00018 #include <string>
00019 #include <stdio.h>
00020
00021 #define INITIAL_WEIGHT 0.5f
00022 #define INITIAL_WEIGHT_INPUTN 0.5f
00023 #define INITIAL_BIAS 0.5f
00024 #define INITIAL_BIAS_INPUTN 0.5f
00025
00033 struct irisData
00034 {
00035     std::vector<std::vector<float>> features;
00036     std::vector<float> targets;
00037 };
00038
00046 template<typename T>
00047 struct digitData
00048 {
00049     std::vector<T> images;
00050     std::vector<T> targets;
00051 };
00052
00053 // Add forward declaration for TrainTestSplit
00054 template<typename T>
00055 struct TrainTestSplit;
00056
00057 // Forward declaration
00058 template<typename T>
00059 TrainTestSplit<T> createTrainTestSplit(
00060     const std::vector<std::vector<T>>& features,
00061     const std::vector<std::vector<T>>& targets,
00062     float splitRatio);
00063
00071 template<typename T>
00072 struct TrainTestSplit
00073 {
00074     // Constructors
00075     TrainTestSplit() = default;
00076     TrainTestSplit(const std::vector<std::vector<T>>& features,
00077         const std::vector<std::vector<T>>& targets,
00078         float splitRatio)
00079     {
00080         TrainTestSplit<T> tts = createTrainTestSplit(features, targets, splitRatio);
00081         trainFeatures = tts.trainFeatures;
00082         testFeatures = tts.testFeatures;
00083         trainTargets = tts.trainTargets;
00084         testTargets = tts.testTargets;
00085     }
00086
00087     std::vector<std::vector<T>> trainFeatures;
00088     std::vector<std::vector<T>> testFeatures;
00089     std::vector<std::vector<T>> trainTargets;
00090     std::vector<std::vector<T>> testTargets;
00091 };

```

```

00092
00100 inline float gradientBetweenNeurons(float& output, float& error)
00101 {
00102     return output * error;
00103 }
00104
00112 inline float deltaGradient(float& learningRate, float& gradient)
00113 {
00114     return learningRate * gradient;
00115 }
00116
00124 inline float deltaBias(float& learningRate, float& deltaGradient)
00125 {
00126     return learningRate * deltaGradient;
00127 }
00128
00129
00136 template<typename T>
00137 inline void printVector(const std::vector<T> &vec, const std::string extra = "")
00138 {
00139     if constexpr (std::is_floating_point_v<T>) {
00140         for (const T &item : vec) {
00141             printf("%.2f ", item);
00142         }
00143     } else if constexpr (std::is_integral_v<T>) {
00144         for (const T &item : vec) {
00145             printf("%i ", item);
00146         }
00147     }
00148     if (!extra.empty()) {
00149         printf("%s", extra.c_str());
00150     }
00151 }
00152
00160 template<typename T>
00161 void normalizeVector(std::vector<T> &vec)
00162 {
00163     T maxElement = *std::max_element(vec.begin(), vec.end());
00164     T minElement = *std::min_element(vec.begin(), vec.end());
00165     T minMax = maxElement - minElement;
00166
00167     // https://www.statology.org/normalize-data-between-0-and-1/
00168     for (std::size_t i = 0; i < vec.size(); i++) {
00169         T xi = vec[i];
00170         vec.at(i) = (xi - minElement) / minMax;
00171     }
00172 }
00173
00181 template<typename T>
00182 void normalize2DVector(std::vector<std::vector<T>> &vec)
00183 {
00184     // Get the max and min element out of the 2D vector
00185     T maxElement, minElement = 0;
00186     T newMaxElement, newMinElement = 0;
00187     for (std::vector<T> &innerVec : vec) {
00188         newMaxElement = *std::max_element(innerVec.begin(), innerVec.end());
00189         newMinElement = *std::min_element(innerVec.begin(), innerVec.end());
00190         if (newMaxElement > maxElement) {
00191             maxElement = newMaxElement;
00192         }
00193         if (newMinElement < minElement) {
00194             minElement = newMinElement;
00195         }
00196     }
00197
00198     T minMax = maxElement - minElement;
00199
00200     // https://www.statology.org/normalize-data-between-0-and-1/
00201     for (std::vector<T> &innerVec : vec) {
00202         for (std::size_t i = 0; i < innerVec.size(); i++) {
00203             T xi = innerVec[i];
00204             innerVec.at(i) = (xi - minElement) / minMax;
00205         }
00206     }
00207 }
00208
00209
00218 template<typename T>
00219 TrainTestSplit<T> createTrainTestSplit(
00220     const std::vector<std::vector<T>> &features,
00221     const std::vector<std::vector<T>> &targets,
00222     float splitRatio)
00223 {
00224     TrainTestSplit<T> tts;
00225     // Reserve the memory for the vectors
00226     tts.trainFeatures.reserve(features.size() * splitRatio);
00227     tts.testFeatures.reserve(features.size() * (1 - splitRatio));

```

```

00228     tts.trainTargets.reserve(targets.size() * splitRatio);
00229     tts.testTargets.reserve(targets.size() * (1 - splitRatio));
00230
00231     std::random_device rd;
00232     std::mt19937 gen(rd());
00233     std::uniform_int_distribution<> dis(0, 1);
00234
00235     for (std::size_t i = 0; i < features.size(); i++)
00236     {
00237         if (i < features.size() * splitRatio) {
00238             tts.trainFeatures.emplace_back(features[i]);
00239             tts.trainTargets.emplace_back(targets[i]);
00240         } else {
00241             tts.testFeatures.emplace_back(features[i]);
00242             tts.testTargets.emplace_back(targets[i]);
00243         }
00244     }
00245     return tts;
00246 }

```

7.5 csv_reader.hpp File Reference

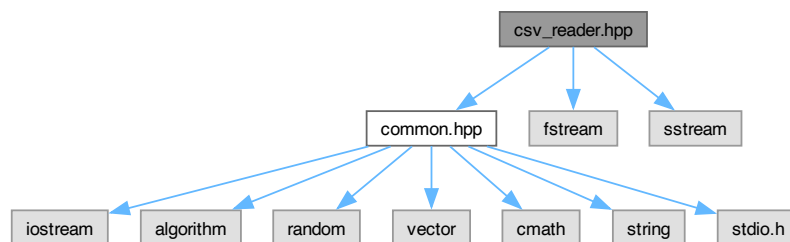
In this class the CSV reader is defined. This is for reading the iris data set.

```

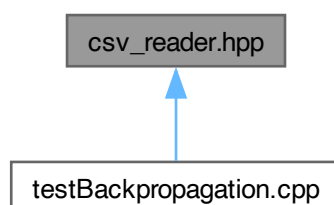
#include "common.hpp"
#include <fstream>
#include <sstream>

```

Include dependency graph for csv_reader.hpp:



This graph shows which files directly or indirectly include this file:



Functions

- `std::vector< std::vector< std::string > > readCsv` (const std::string &filename, char delimiter=',')
Reads a CSV file and returns a vector of vectors.
- `std::vector< float > getTargets` (const std::vector< std::vector< std::string > > &data)
Extracts the features from the data (column).
- `std::vector< std::vector< float > > getFeatures` (const std::vector< std::vector< std::string > > &data)
Extracts the features from the data.
- `irisData filterData` (const std::vector< std::vector< float > > &features, const std::vector< float > &targets, int target)
Filters out data points with a specific target value.
- `template<typename T>`
`T convert` (const std::string &str)
Reads the digit data from a CSV file.
- `template<> int convert< int >` (const std::string &str)
- `template<> float convert< float >` (const std::string &str)
- `template<typename T>`
`std::vector< T > readCsvFlat` (const std::string &filename, char delimiter=',')
Reads a CSV file and returns a vector of the specified type.
- `template<typename T>`
`std::vector< std::vector< T > > create2DVector` (const std::vector< T > vec, int size)
- `template<typename T>`
`digitData< T > readDigitData` ()
- `template<typename T>`
`std::vector< T > maskData` (std::vector< T > &data, std::vector< T > &mask)

7.5.1 Detailed Description

In this class the CSV reader is defined. This is for reading the iris data set.

Author

Stan Merlijn

Version

0.1

Date

2025-02-14

Copyright

Copyright (c) 2025

Definition in file [csv_reader.hpp](#).

7.5.2 Function Documentation

7.5.2.1 convert()

```
template<typename T>
T convert (
    const std::string & str)
```

Reads the digit data from a CSV file.

This function reads the digit data from a CSV file and returns a structure containing the features and target values.

Parameters

<i>filename</i>	The name of the CSV file to read.
-----------------	-----------------------------------

Returns

[irisData](#) A structure containing the features and target values.

7.5.2.2 convert< float >()

```
template<>
float convert< float > (
    const std::string & str)
```

Definition at line 157 of file [csv_reader.hpp](#).

7.5.2.3 convert< int >()

```
template<>
int convert< int > (
    const std::string & str)
```

Definition at line 152 of file [csv_reader.hpp](#).

7.5.2.4 create2DVector()

```
template<typename T>
std::vector< std::vector< T > > create2DVector (
    const std::vector< T > vec,
    int size)
```

Definition at line 197 of file [csv_reader.hpp](#).

7.5.2.5 filterData()

```
irisData filterData (
    const std::vector< std::vector< float > > & features,
    const std::vector< float > & targets,
    int target)
```

Filters out data points with a specific target value.

This function takes a set of features and corresponding target values, and filters out the data points where the target value matches the specified target. The remaining data points are returned in a new [irisData](#) structure.

Parameters

<i>features</i>	A vector of vectors containing the feature data.
<i>targets</i>	A vector containing the target values corresponding to the feature data.
<i>target</i>	The target value to filter out from the data.

Returns

[irisData](#) A structure containing the filtered feature data and target values.

Definition at line 119 of file [csv_reader.hpp](#).

7.5.2.6 getFeatures()

```
std::vector< std::vector< float > > getFeatures (
    const std::vector< std::vector< std::string > > & data)
```

Extracts the features from the data.

This function extracts the features from the data and returns a vector of vectors containing the features.

Parameters

<i>data</i>	A vector of vectors representing the rows in the CSV file.
-------------	--

Returns

A vector containing the features.

Definition at line 93 of file [csv_reader.hpp](#).

7.5.2.7 getTargets()

```
std::vector< float > getTargets (
    const std::vector< std::vector< std::string > > & data)
```

Extracts the features from the data (column).

This function extracts the features from the data and returns a vector of vectors containing the features.

Parameters

<i>data</i>	A vector of vectors representing the rows in the CSV file.
-------------	--

Returns

A vector containing the features.

Definition at line 75 of file [csv_reader.hpp](#).

7.5.2.8 maskData()

```
template<typename T>
std::vector< T > maskData (
    std::vector< T > & data,
    std::vector< T > & mask)
```

Definition at line 231 of file [csv_reader.hpp](#).

7.5.2.9 readCsv()

```
std::vector< std::vector< std::string > > readCsv (
    const std::string & filename,
    char delimiter = ',')
```

Reads a CSV file and returns a vector of vectors.

This function reads a CSV file and returns a vector of vectors. Each inner vector represents a row in the CSV file. The function assumes that the CSV file is well-formed and does not contain any missing values.

Parameters

<i>filename</i>	The name of the CSV file to read.
<i>delimiter</i>	The delimiter used in the CSV file.

Returns

A vector of vectors representing the rows in the CSV file.

Definition at line 32 of file [csv_reader.hpp](#).

7.5.2.10 readDigitData()

```
template<typename T>
digitData< T > readDigitData ()
```

Definition at line 213 of file [csv_reader.hpp](#).

7.5.2.11 readCsvFlat()

```
template<typename T>
std::vector< T > readCsvFlat (
    const std::string & filename,
    char delimiter = ',')
```

Reads a CSV file and returns a vector of the specified type.

This templated function reads a CSV file, splits each line by the given delimiter, converts the tokens to type T, and returns them.

Template Parameters

<i>T</i>	The type to convert the CSV tokens into.
----------	--

Parameters

<i>filename</i>	The name of the CSV file to read.
<i>delimiter</i>	The delimiter used in the CSV file.

Returns

A vector of all the tokens in the CSV file converted to type T.

Definition at line 173 of file [csv_reader.hpp](#).

7.6 csv_reader.hpp

[Go to the documentation of this file.](#)

```

00001
00011 #pragma once
00012 #include "common.hpp"
00013 #include <fstream>
00014 #include <sstream>
00015
00016
00017 // =====
00018 // Section: reading iris data
00019 // =====
00020
00032 std::vector<std::vector<std::string> readCsv(const std::string& filename, char delimiter=',')
00033 {
00034     // Create a vector to store the rows
00035     std::vector<std::vector<std::string> rows;
00036     std::ifstream file(filename);
00037
00038     // Check if the file is open
00039     if (!file.is_open()) {
00040         std::cerr << "Error: Could not open file " << filename << std::endl;
00041         return rows;
00042     }
00043
00044     // Read the file line by line
00045     std::string line;
00046     while (std::getline(file, line)) {
00047         std::stringstream ss(line);
00048         std::vector<std::string> cols;
00049         std::string col;
00050
00051         while (std::getline(ss, col, delimiter)) {
00052             cols.push_back(col);
00053         }
00054
00055         // Add the columns to the rows
00056         rows.push_back(cols);
00057     }
00058
00059     // Close the file
00060     file.close();
00061
00062     return rows;
00063 }
00064
00065
00075 std::vector<float> getTargets(const std::vector<std::vector<std::string>& data)
00076 {
00077     std::vector<float> targets;
00078     for (const auto& row : data) {
00079         targets.push_back(std::stof(row.back()));
00080     }
00081     return targets;
00082 }
00083
00093 std::vector<std::vector<float> getFeatures(const std::vector<std::vector<std::string>& data)
00094 {
00095     std::vector<std::vector<float> features;
00096     for (const auto& row : data) {
00097         std::vector<float> feature_row;
00098         // Skip the last column which contains the target
00099         for (int i = 0; i < row.size() - 1; i++) {
00100             feature_row.push_back(std::stof(row[i]));
00101         }
00102         features.push_back(feature_row);
00103     }
00104     return features;
00105 }
00106
00119 irisData filterData(const std::vector<std::vector<float>& features, const std::vector<float>& targets,
00120 int target)
00121 {
00122     std::vector<std::vector<float> filtered_features;
00123     std::vector<float> filtered_targets;
00124     for (int i = 0; i < features.size(); i++) {
00125         if (targets[i] != target) {
00126             filtered_features.push_back(features[i]);
00127             filtered_targets.push_back(targets[i]);
00128         }
00129     }
00130     return irisData{filtered_features, filtered_targets};
00131 }

```



```

00132
00133 // =====
00134 // Section: reading digits data
00135 // =====
00136
00146
00147 // Helper conversion function template with specializations.
00148 template<typename T>
00149 T convert(const std::string& str);
00150
00151 template<>
00152 int convert<int>(const std::string& str) {
00153     return std::stoi(str);
00154 }
00155
00156 template<>
00157 float convert<float>(const std::string& str) {
00158     return std::stof(str);
00159 }
00160
00172 template<typename T>
00173 std::vector<T> readCsvFlat(const std::string& filename, char delimiter = ',')
00174 {
00175     std::vector<T> data;
00176     std::ifstream file(filename);
00177
00178     if (!file.is_open()) {
00179         std::cerr << "Error: Could not open file " << filename << std::endl;
00180         return data;
00181     }
00182
00183     std::string line;
00184     while (std::getline(file, line)) {
00185         std::stringstream ss(line);
00186         std::string token;
00187
00188         while (std::getline(ss, token, delimiter)) {
00189             data.push_back(convert<T>(token));
00190         }
00191     }
00192     file.close();
00193     return data;
00194 }
00195
00196 template<typename T>
00197 std::vector<std::vector<T>> create2DVector(const std::vector<T> vec, int size)
00198 {
00199     std::vector<std::vector<T>> data;
00200     data.reserve(vec.size() / size);
00201     for (int i = 0; i < vec.size(); i += size) {
00202         std::vector<T> row;
00203         for (int j = 0; j < size; j++) {
00204             row.push_back(vec[i + j]);
00205         }
00206         data.push_back(row);
00207     }
00208
00209     return data;
00210 }
00211
00212 template<typename T>
00213 digitData<T> readDigitData()
00214 {
00215     // Not the best way to do this, but it works for now
00216     // presumes that the data is in the data folder in the root of the project
00217     const std::string filenameImages = "../data/digits_images.csv";
00218     const std::string filenameTargets = "../data/digits_targets.csv";
00219
00220     // Load the data
00221     digitData<T> data;
00222     std::vector<T> images = readCsvFlat<T>(filenameImages);
00223     normalizeVector(images);
00224     data.images = images;
00225     data.targets = readCsvFlat<T>(filenameTargets);
00226
00227     return data;
00228 }
00229
00230 template<typename T>
00231 std::vector<T> maskData(std::vector<T>& data, std::vector<T>& mask)
00232 {
00233     T off, on;
00234     if constexpr (std::is_floating_point_v<T>) {
00235         off = 0.0f;
00236         on = 1.0f;
00237     } else if constexpr (std::is_integral_v<T>) {
00238         off = 0;

```

```

00239     on = 1;
00240 } else {
00241     throw std::runtime_error("Data type not supported");
00242 }
00243
00244 std::vector<T> maskedData;
00245 maskedData.reserve(data.size() * mask.size());
00246 for (std::size_t i = 0; i < data.size(); i++)
00247 {
00248     for (std::size_t j = 0; j < mask.size(); j++)
00249     {
00250         if (data.at(i) == mask.at(j)) {
00251             maskedData.emplace_back(on);
00252         } else {
00253             maskedData.emplace_back(off);
00254         }
00255     }
00256 }
00257 return maskedData;
00258 }

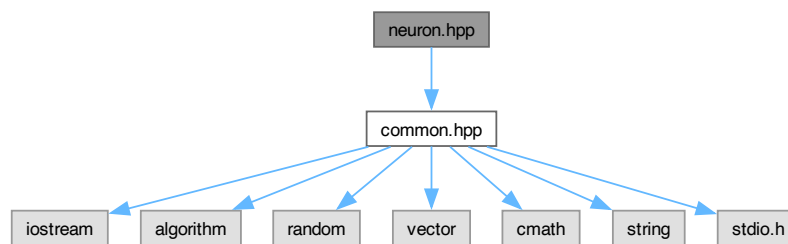
```

7.7 neuron.hpp File Reference

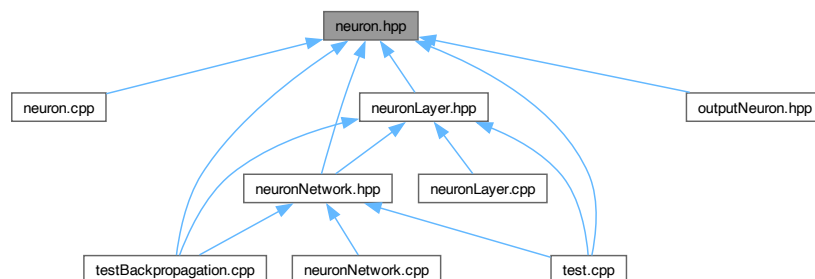
In this file the [Neuron](#) class is declared. This class represents a single neuron in a neural network.

```
#include "common.hpp"
```

Include dependency graph for neuron.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class [Neuron](#)

Represents a single neuron in a neural network.

7.7.1 Detailed Description

In this file the [Neuron](#) class is declared. This class represents a single neuron in a neural network.

Author

Stan Merlijn

Version

0.1

Date

2025-02-14

Copyright

Copyright (c) 2025

Definition in file [neuron.hpp](#).

7.8 neuron.hpp

[Go to the documentation of this file.](#)

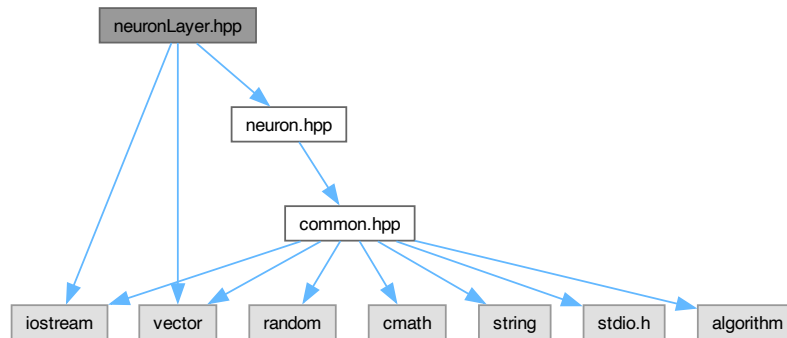
```
00001
00011 #pragma once
00012 #include "common.hpp"
00021 class Neuron {
00022 private:
00023     std::vector<float> _weights;
00024     std::vector<float> _lastInput;
00025     float _bias;
00026     float _learningRate;
00027     float _lastOutput;
00028     float _delta;
00029
00030 public:
00035     Neuron(int nSizeWeights, float initialWeight, float initialBias);
00036
00042     Neuron(const std::vector<float>& weights, float bias, float learningRate = 0.1);
00043
00049     float sigmoid(float x);
00050
00056     float activate(const std::vector<float>& inputs);
00057
00063     float predict(const std::vector<float>& inputs);
00064
00070     void deltaError(const std::vector<float>& inputs, const std::vector<Neuron>& neuronsNextLayer,
float target, bool isOutputNeuron);
00071
00075     void update();
00076
00083     float computeHiddenDelta(const std::vector<float>& inputs, float sum);
00084
00091     float sigmoidDerivative(float output);
00092
00099     float computeOutputDelta(float target);
00100
00105     const std::vector<float>& getWeights() const { return _weights; }
00106
00111     float getBias() const { return _bias; }
00112
00117     float getError() const { return _delta; }
00118
00122     void __str__() const;
00123
00124
00125
00126 };
```

7.9 neuronLayer.hpp File Reference

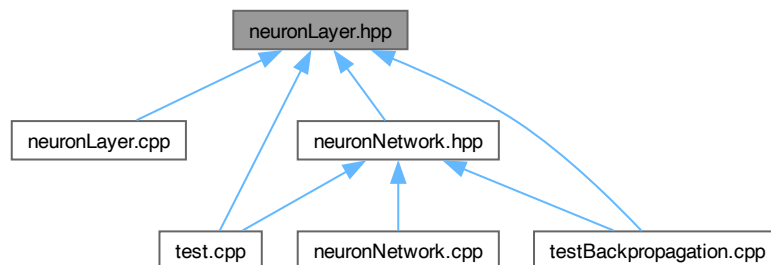
In this file the [NeuronLayer](#) class is declared. This class represents a layer of neurons in a neural network.

```
#include <iostream>
#include <vector>
#include "neuron.hpp"
```

Include dependency graph for neuronLayer.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class [NeuronLayer](#)
Represents a layer of neurons in a neural network.

7.9.1 Detailed Description

In this file the [NeuronLayer](#) class is declared. This class represents a layer of neurons in a neural network.

Author

Stan Merlijn

Version

0.1

Date

2025-02-14

Copyright

Copyright (c) 2025

Definition in file [neuronLayer.hpp](#).

7.10 neuronLayer.hpp

[Go to the documentation of this file.](#)

```

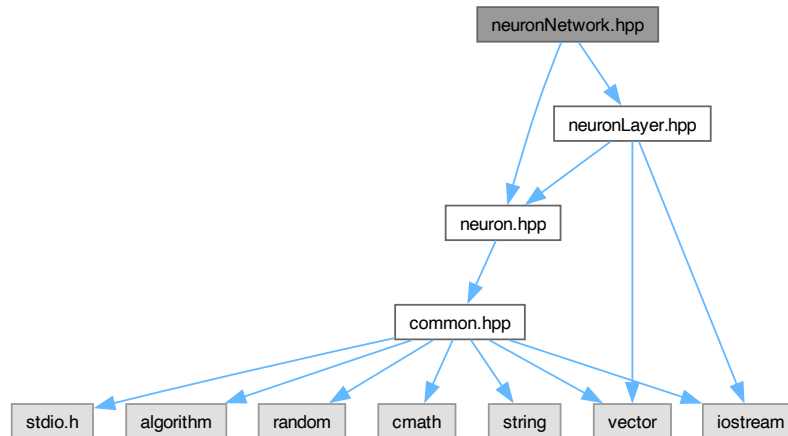
00001
00011 #pragma once
00012 #include <iostream>
00013 #include <vector>
00014 #include "neuron.hpp"
00015
00023 class NeuronLayer {
00024 private:
00025     std::vector<Neuron> _neurons;
00026     std::vector<float> _output;
00027
00028 public:
00033     NeuronLayer(std::vector<Neuron> neurons);
00034
00040     NeuronLayer(int nNeurons, int nSizeWeights);
00041
00047     std::vector<float>& feedForward(const std::vector<float>& inputs);
00048
00053     void computeOutputErrors(const std::vector<float>& targets);
00054
00060     void computeHiddenErrors(const std::vector<float>& inputs, const std::vector<Neuron>&
neuronsNextLayer);
00061
00065     void update();
00066
00071     const std::vector<Neuron>& getNeurons() { return _neurons; }
00072
00077     const std::vector<float>& getOutput() const { return _output; }
00078
00082     void __str__() const;
00083 };

```

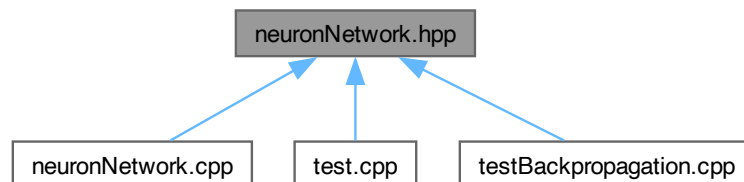
7.11 neuronNetwork.hpp File Reference

In this file the [NeuronNetwork](#) class is declared. This class represents a neural network with multiple layers of neurons.

```
#include "neuron.hpp"
#include "neuronLayer.hpp"
Include dependency graph for neuronNetwork.hpp:
```



This graph shows which files directly or indirectly include this file:



Classes

- class [NeuronNetwork](#)
Represents a neural network with multiple layers of neurons.

7.11.1 Detailed Description

In this file the [NeuronNetwork](#) class is declared. This class represents a neural network with multiple layers of neurons.

Author

Stan Merlijn

Version

0.1

Date

2025-02-14

Copyright

Copyright (c) 2025

Definition in file [neuronNetwork.hpp](#).

7.12 neuronNetwork.hpp

[Go to the documentation of this file.](#)

```

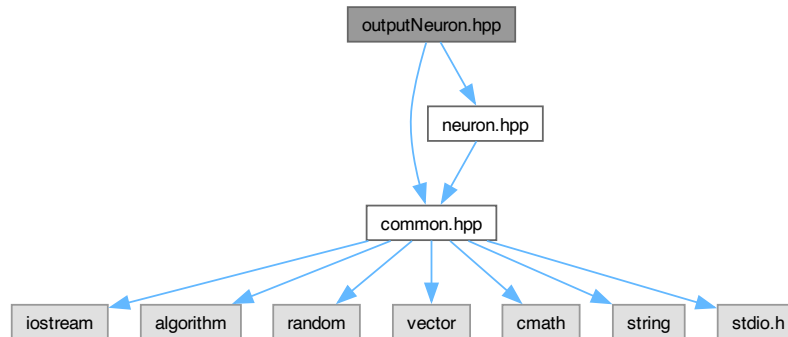
00001
00011 #pragma once
00012 #include "neuron.hpp"
00013 #include "neuronLayer.hpp"
00014
00022 class NeuronNetwork {
00023 private:
00024     std::vector<NeuronLayer> _layers;
00025     std::vector<float> _currentTargets;
00026
00027     std::vector<float> _inputVec;
00028     std::vector<float> _currentLayerOutput;
00029     std::vector<float> _tempOutputBuffer;
00030
00031     std::vector<float> _outputMask;
00032 public:
00037     NeuronNetwork(std::vector<NeuronLayer> layers);
00038
00044     NeuronNetwork(std::vector<int> layers);
00045
00046
00052     const std::vector<float>& feedForward(const std::vector<float>& inputs);
00053
00059     std::vector<float> predict(const std::vector<float>& input);
00060
00064     void backPropagation(const std::vector<float>& targets);
00065
00066     void maskTarget(float target);
00070     void update();
00078     void trainInputs2D(const std::vector<std::vector<float>>& inputs, const
std::vector<std::vector<float>>& targets, int epochs);
00079
00084     std::vector<NeuronLayer> getLayers() const { return _layers; }
00085
00091     void setTarget(std::vector<float>& targets) {_currentTargets = targets;}
00092
00096     void __str__() const;
00097
00098
00099 };

```

7.13 outputNeuron.hpp File Reference

In this file the OutputNeuron class is declared. This class represents a single output neuron in a neural network.

```
#include "common.hpp"
#include "neuron.hpp"
Include dependency graph for outputNeuron.hpp:
```



Classes

- class [OutputNueron](#)

7.13.1 Detailed Description

In this file the `OutputNeuron` class is declared. This class represents a single output neuron in a neural network.

Author

Stan Merlijn

Version

0.1

Date

2025-02-24

Copyright

Copyright (c) 2025

Definition in file [outputNeuron.hpp](#).

7.14 outputNeuron.hpp

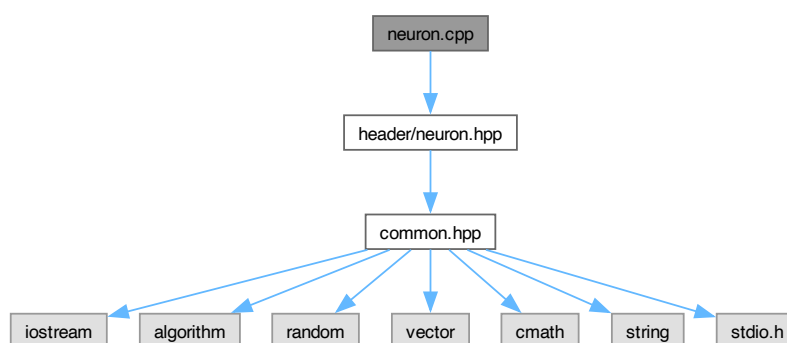
[Go to the documentation of this file.](#)

```
00001
00011
00012 #include "common.hpp"
00013 #include "neuron.hpp"
00014
00015 class OutputNueron : Neuron
00016 {
00017 public:
00018
00019 private:
00020
00021 };
```

7.15 neuron.cpp File Reference

In this file the [Neuron](#) class is implemented.

```
#include "header/neuron.hpp"
Include dependency graph for neuron.cpp:
```



7.15.1 Detailed Description

In this file the [Neuron](#) class is implemented.

Author

Stan Merlijn

Version

0.1

Date

2025-02-14

Copyright

Copyright (c) 2025

Definition in file [neuron.cpp](#).

7.16 neuron.cpp

[Go to the documentation of this file.](#)

```

00001
00011 #include "header/neuron.hpp"
00012
00013 Neuron::Neuron(int nSizeWeights, float initialWeight, float initialBias)
00014 {
00015     // Initialize the weights and bias
00016     _weights = std::vector<float>(nSizeWeights, initialWeight);
00017     _lastInput.reserve(nSizeWeights);
00018
00019     // Initialize the delta and learning rate
00020     _delta = 0.0f;
00021     _bias = initialBias;
00022     _learningRate = 0.5f;
00023     _lastOutput = 0.0f;
00024 }
00025
00026 Neuron::Neuron(const std::vector<float>& weights, float bias, float learningRate)
00027 : _weights(weights), _bias(bias), _learningRate(learningRate) {}
00028
00029 float Neuron::sigmoid(float x)
00030 {
00031     // Sigmoid activation function
00032     return 1 / (1 + exp(-x));
00033 }
00034
00035 float Neuron::activate(const std::vector<float>& inputs)
00036 {
00037     // Calculate the weighted sum of the inputs
00038     _lastInput = inputs;
00039     float weightedSum = _bias;
00040
00041     // Dot product of the weights and inputs
00042     for (std::size_t i = 0; i < _weights.size(); i++)
00043     {
00044         weightedSum += _weights[i] * inputs[i];
00045     }
00046
00047     // Return the result of the sigmoid function
00048     _lastOutput = sigmoid(weightedSum);
00049     return _lastOutput;
00050 }
00051
00052 float Neuron::predict(const std::vector<float>& inputs)
00053 {
00054     // Return 1 if the result is greater than 0.5, otherwise return 0(threshold)
00055     return (activate(inputs) > 0.5) ? 1 : 0;
00056 }
00057
00058 void Neuron::update()
00059 {
00060     // Update the weights and bias
00061     for (std::size_t i = 0; i < _weights.size(); i++)
00062     {
00063         _weights[i] -= _learningRate * _lastInput[i] * _delta;
00064     }
00065     // Update the bias
00066     _bias -= _learningRate * _delta;
00067 }
00068
00069 float Neuron::computeHiddenDelta(const std::vector<float>& inputs, float sum)
00070 {
00071     _delta = sigmoidDerivative(_lastOutput) * sum;
00072     return _delta;
00073 }
00074
00075 float Neuron::computeOutputDelta(float target)
00076 {
00077     _delta = sigmoidDerivative(_lastOutput) * -(target - _lastOutput);
00078     return _delta;
00079 }
00080
00081 float Neuron::sigmoidDerivative(float output)
00082 {
00083     return output * (1 - output);
00084 }
00085
00086 void Neuron::__str__() const
00087 {
00088     // Print the neuron details
00089     printf("\nNeurons with %zu weights: ", _weights.size());
00090     printVector(_weights);
00091     printf("| Bias = %f | Learning rate = %f", _bias, _learningRate);

```

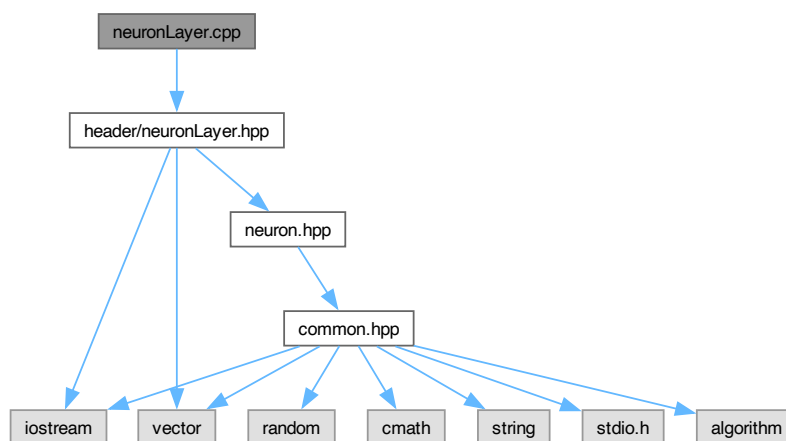
```
00092 }
```

7.17 neuronLayer.cpp File Reference

In this file the [NeuronLayer](#) class is implemented.

```
#include "header/neuronLayer.hpp"
```

Include dependency graph for neuronLayer.cpp:



7.17.1 Detailed Description

In this file the [NeuronLayer](#) class is implemented.

Author

Stan Merlijn

Version

0.1

Date

2025-02-14

Copyright

Copyright (c) 2025

Definition in file [neuronLayer.cpp](#).

7.18 neuronLayer.cpp

[Go to the documentation of this file.](#)

```

00001
00011 #include "header/neuronLayer.hpp"
00012
00013 NeuronLayer::NeuronLayer(std::vector<Neuron> neurons)
00014     : _neurons(neurons) {}
00015
00016 NeuronLayer::NeuronLayer(int nNeurons, int nSizeWeights)
00017 {
00018     _output.resize(nNeurons);
00019     // nNeurons check
00020     if (nNeurons == 0) {
00021         printf("nNeuron must be atleast 1 is %d", nNeurons) ;
00022         return;
00023     }
00024
00025     std::random_device rd;
00026     std::mt19937 gen(rd());
00027     std::uniform_real_distribution<> dis(0.1f, 1.0f);
00028
00029     _neurons.reserve(nNeurons);
00030     for (std::size_t i = 0; i < nNeurons; i++)
00031     {
00032         _neurons.emplace_back(nSizeWeights, dis(gen), dis(gen));
00033     }
00034 }
00035
00036 std::vector<float>& NeuronLayer::feedForward(const std::vector<float>& inputs)
00037 {
00038     // Feed forward through each neuron in the layer
00039     for (std::size_t i = 0; i < _neurons.size(); i++)
00040     {
00041         // For now using the activate instead of predict.
00042         // The predict function is used for binary classification i think.
00043         _output[i] = _neurons[i].activate(inputs);
00044     }
00045     return _output;
00046 }
00047
00048 void NeuronLayer::computeOutputErrors(const std::vector<float> &targets)
00049 {
00050     // Will only run for the output neurons
00051     for (std::size_t i = 0; i < targets.size(); i++) {
00052         _neurons[i].computeOutputDelta(targets[i]);
00053     }
00054 }
00055
00056 void NeuronLayer::computeHiddenErrors(const std::vector<float>& inputs, const std::vector<Neuron>&
neuronsNextLayer)
00057 {
00058     // // Simply get the first neurons weight size
00059     for (std::size_t i = 0; i < _neurons.size(); i++) {
00060
00061         float sum = 0.0f;
00062
00063         // Loop over neurons in next layer
00064         for (std::size_t j = 0; j < neuronsNextLayer.size(); j++)
00065         {
00066             sum += neuronsNextLayer[j].getWeights()[i] * neuronsNextLayer[j].getError();
00067         }
00068
00069         _neurons[i].computeHiddenDelta(inputs, sum);
00070     }
00071 }
00072
00073 void NeuronLayer::update()
00074 {
00075     for (Neuron& n : _neurons)
00076     {
00077         n.update();
00078     }
00079 }
00080
00081 void NeuronLayer::__str__() const
00082 {
00083     // Print the layer details
00084     printf("\nNeuronLayer with %zu neurons", _neurons.size());
00085     for (std::size_t i = 0; i < _neurons.size(); i++)
00086     {
00087         _neurons[i].__str__();
00088     }
00089     printf("\n");
00090 }

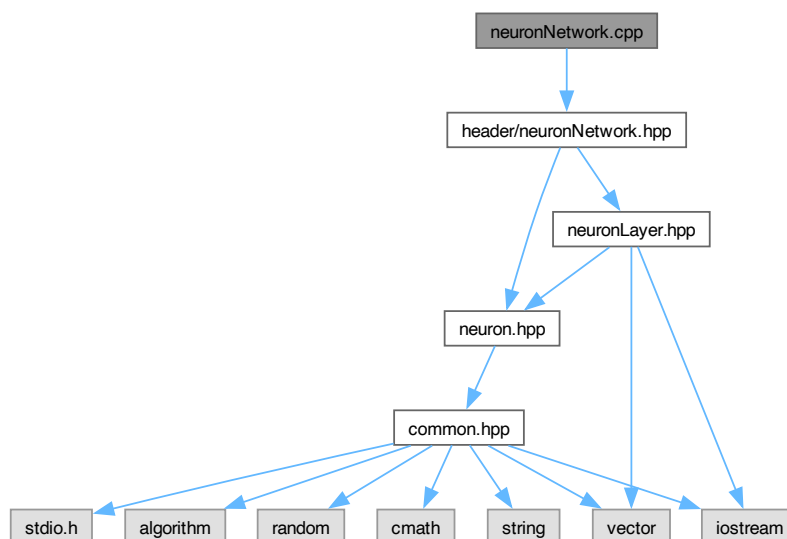
```

7.19 neuronNetwork.cpp File Reference

In this file the [NeuronNetwork](#) class is implemented.

```
#include "header/neuronNetwork.hpp"
```

Include dependency graph for neuronNetwork.cpp:



7.19.1 Detailed Description

In this file the [NeuronNetwork](#) class is implemented.

Author

Stan Merlijn

Version

0.1

Date

2025-02-14

Copyright

Copyright (c) 2025

Definition in file [neuronNetwork.cpp](#).

7.20 neuronNetwork.cpp

[Go to the documentation of this file.](#)

```

00001
00011 #include "header/neuronNetwork.hpp"
00012
00013 NeuronNetwork::NeuronNetwork(std::vector<NeuronLayer> layers)
00014     : _layers(layers) {}
00015
00016 NeuronNetwork::NeuronNetwork(std::vector<int> layerSizes)
00017 {
00018     // Reserve the input vector and the current targets
00019     _inputVec.resize(layerSizes.front());
00020     _currentTargets.resize(layerSizes.back());
00021
00022     // Reserve because there is no default constructor for NeuronLayer
00023     _layers.reserve(layerSizes.size());
00024
00025     // Reserve the temp output buffer and the current layer output
00026     _tempOutputBuffer.resize(*std::max_element(layerSizes.begin(), layerSizes.end()));
00027     _currentLayerOutput.resize(*std::max_element(layerSizes.begin(), layerSizes.end()));
00028     // Create the layers
00029     for (std::size_t i = 1; i < layerSizes.size(); i++) {
00030         // If its the first layer then the input size is the first element in the layerSizes
00031         if (i == 1) {
00032             _layers.emplace_back(layerSizes[i], layerSizes.front());
00033         } else {
00034             _layers.emplace_back(layerSizes[i], layerSizes[i-1]);
00035         }
00036     }
00037 }
00038
00039 const std::vector<float>& NeuronNetwork::feedForward(const std::vector<float>& inputs)
00040 {
00041     // Set the input vector and the current layer output
00042     _inputVec = inputs;
00043     _currentLayerOutput = inputs;
00044
00045     // Feed forward through each layer in the network
00046     for (std::size_t i = 0; i < _layers.size(); i++) {
00047         _currentLayerOutput = _layers[i].feedForward(_currentLayerOutput);
00048     }
00049     return _currentLayerOutput;
00050 }
00051
00052 std::vector<float> NeuronNetwork::predict(const std::vector<float>& input)
00053 {
00054     return feedForward(input);
00055 }
00056
00057 void NeuronNetwork::backPropagation(const std::vector<float>& targets)
00058 {
00059     // Compute the output errors
00060     int last = _layers.size() - 1;
00061     _layers[last].computeOutputErrors(targets);
00062
00063     // Reverse loop For hidden layers
00064     for (int i = last-1; i > -1; i--) {
00065         // If is output neuron compute the output error
00066         if (i == 0) { // If i == 0 then its the input layer
00067             _layers[i].computeHiddenErrors(_inputVec, _layers[i + 1].getNeurons());
00068         } else { // Else compute the hidden error
00069             _layers[i].computeHiddenErrors(_layers[i-1].getOutput(), _layers[i + 1].getNeurons());
00070         }
00071     }
00072 }
00073
00074 void NeuronNetwork::update()
00075 {
00076     for (NeuronLayer& nL : _layers) {
00077         nL.update();
00078     }
00079 }
00080
00081 void NeuronNetwork::trainInputs2D(const std::vector<std::vector<float>>& inputs, const
std::vector<std::vector<float>>& targets, int epochs)
00082 {
00083     // Check if the flat input is the same as the targets
00084     if (!((inputs.size()) == targets.size())) {
00085         throw std::runtime_error("Input and target size are not the same");
00086     }
00087
00088     // Loop over the epochs
00089     for (int x = 0; x < epochs; x++) {
00090         // Loop over each input and target

```

```

00091         for (std::size_t i = 0; i < targets.size(); i++) {
00092             feedForward(inputs[i]);           // Feed forward
00093             backPropagation(targets[i]);       // Back propagate
00094             update();                          // Update the weights
00095         }
00096     }
00097 }
00098
00099 // void NeuronNetwork::trainInputs(const std::vector<std::vector<float>>& inputs, const
std::vector<std::vector<float>>& targets,
00100 //     int inputSize, int targetSize, int epochs)
00101 // {
00102 //     // Check if the flat input is the same as the targets
00103 //     // if (!((inputs.size() / inputSize) == targets.size() / targetSize)) {
00104 //         throw std::runtime_error("Input and target size are not the same");
00105 //     }
00106 //
00107 //     std::vector<float> input(inputSize);
00108 //     std::vector<float> target(targetSize);
00109 //
00110 //     // Loop over the epochs
00111 //     for (int x = 0; x < epochs; x++) {
00112 //         // Loop over each input and target
00113 //         for (std::size_t i = 0; i < targets.size(); i++) {
00114 //             // Set the input for the network
00115 //             std::size_t startIndexInput = i * inputSize;
00116 //
00117 //             // for (std::size_t j = 0; j < inputSize; j++) {
00118 //                 input[j] = inputs[startIndexInput + j];
00119 //             }
00120 //
00121 //             // Set the target for the network
00122 //             std::size_t startIndexTarget = i * targetSize;
00123 //             // for (std::size_t j = 0; j < targetSize; j++) {
00124 //                 _target[j] = targets[startIndexTarget + j];
00125 //             }
00126 //
00127 //             // maskTarget(targets[i]); // Set the target for the network
00128 //             feedForward(inputs[i]);     // Feed forward
00129 //             backPropagation(targets[i]); // Back propagate
00130 //             update();                   // Update the weights
00131 //         }
00132 //     }
00133 // }
00134
00135 void NeuronNetwork::maskTarget(float target)
00136 {
00137     // if (_outputMask.size() != _currentTargets.size()) {
00138     //     throw std::runtime_error("OutputMask and current targets are not the same size");
00139     //     exit(1);
00140     // }
00141 //
00142 //
00143 //     // Set the target to the current target
00144 //     for (std::size_t i = 0; i < _outputMask.size(); i++) {
00145 //         if (_outputMask[i] == target) {
00146 //             _currentTargets[i] = 1.0f;
00147 //         } else {
00148 //             _currentTargets[i] = 0.0f;
00149 //         }
00150 //     }
00151 // }
00152
00153 void NeuronNetwork::__str__() const
00154 {
00155     // Print the network details
00156     printf("\nNeuronNetwork with %zu layers\n", _layers.size());
00157     for (std::size_t i = 0; i < _layers.size(); i++)
00158     {
00159         _layers[i].__str__();
00160     }
00161 }

```

7.21 test.cpp File Reference

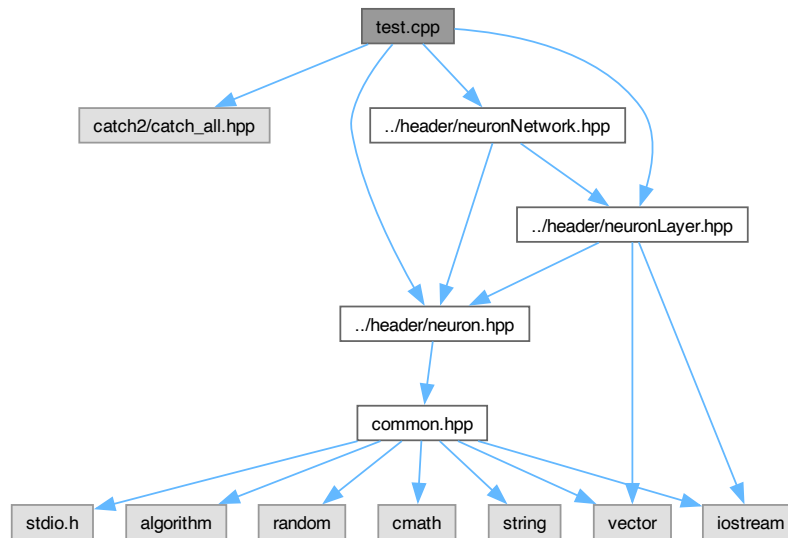
In this file the tests for the [Neuron](#), [NeuronLayer](#) and [NeuronNetwork](#) classes are implemented.

```

#include <catch2/catch_all.hpp>
#include "../header/neuron.hpp"

```

```
#include "../header/neuronLayer.hpp"
#include "../header/neuronNetwork.hpp"
Include dependency graph for test.cpp:
```



Functions

- **TEST_CASE** ("Neuron AND gate", "[neuron]")
In this test case we test the ability of a single neuron to learn the AND gate.
- **TEST_CASE** ("Neuron OR gate", "[neuron]")
In this test case we test the ability of a single neuron to learn the OR gate.
- **TEST_CASE** ("Neuron NOT gate", "[neuron]")
In this test case we test the ability of a single neuron to learn the NOT gate.
- **TEST_CASE** ("Neuron NOR gate (NOT OR)", "[neuron]")
In this test case we test the ability of a single neuron to learn the NOR gate.
- **TEST_CASE** ("Half Adder using Two-Layer Neuron Network", "[half-adder]")
In this test case we test the ability of a two-layer neural network to learn the XOR gate.

7.21.1 Detailed Description

In this file the tests for the [Neuron](#), [NeuronLayer](#) and [NeuronNetwork](#) classes are implemented.

Unit tests for the [Neuron](#), [NeuronLayer](#) and [NeuronNetwork](#) classes.

Author

Stan Merlijn

Version

0.1

Date

2025-02-14

Copyright

Copyright (c) 2025

This file contains a series of test cases to verify the functionality of the [Neuron](#) and [NeuronLayer](#) classes. The tests include training and prediction for various logic gates.

Test Cases:

- [Neuron](#) for AND Gate: Tests the [Neuron](#)'s ability to learn the AND gate.
- [Neuron](#) for OR Gate: Tests the [Neuron](#)'s ability to learn the OR gate.
- [Neuron](#) for NOT Gate: Tests the [Neuron](#)'s ability to learn the NOT gate.
- [Neuron](#) for NOR Gate (3 inputs): Tests the [Neuron](#)'s ability to learn the NOR gate with 3 inputs.
- [NeuronNetwork](#) for the XOR gate with 2 inputs.

Note

The tests use the Catch2 framework for unit testing.

Definition in file [test.cpp](#).

7.21.2 Macro Definition Documentation

7.21.2.1 CATCH_CONFIG_MAIN

```
#define CATCH_CONFIG_MAIN
```

Definition at line 11 of file [test.cpp](#).

7.21.3 Function Documentation

7.21.3.1 TEST_CASE() [1/5]

```
TEST_CASE (
    "Half Adder using Two-Layer Neuron Network" ,
    "" [half-adder])
```

In this test case we test the ability of a two-layer neural network to learn the XOR gate.

The XOR gate is a binary operation that returns true if the inputs are different, and false otherwise. The network consists of two layers: a hidden layer with an OR and an AND neuron, and an output layer with a *XOR* and a carry neuron.

The XOR in this case is not a traditional XOR gate, but a neuron that computes the XOR operation. It works because it can only take 3 inputs which is linearly separable. The inputs are the output of the OR and AND neurons. The are as follows:

x1	x2	OR	AND
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

So the only inputs for the XOR neuron are (0,0), (1,0) and (0,0).

XOR = OR - AND; neuron with weights {1, -1} and bias -0.5

- $x1 = 0, x2 = 0 : 1 * 0 + -1 * 0 - 0.5 = -0.5$
- $x1 = 1, x2 = 0 : 1 * 1 + -1 * 0 - 0.5 = 0.5$
- $x1 = 1, x2 = 1 : 1 * 1 + -1 * 1 - 0.5 = -0.5$

Definition at line 137 of file [test.cpp](#).

7.21.3.2 TEST_CASE() [2/5]

```
TEST_CASE (
    "Neuron AND gate" ,
    "" [neuron])
```

In this test case we test the ability of a single neuron to learn the AND gate.

The AND gate is a binary operation that returns true if both inputs are true, and false otherwise. With a bias of -1.5 the dot product will only be greater than 0 if both inputs are 1

- $x1 = 0, x2 = 0 : 1 * 0 + 1 * 0 - 1.5 = -1.5$
- $x1 = 0, x2 = 1 : 1 * 0 + 1 * 1 - 1.5 = -0.5$
- $x1 = 1, x2 = 0 : 1 * 1 + 1 * 0 - 1.5 = -0.5$
- $x1 = 1, x2 = 1 : 1 * 1 + 1 * 1 - 1.5 = 0.5$

Definition at line 45 of file [test.cpp](#).

7.21.3.3 TEST_CASE() [3/5]

```
TEST_CASE (
    "Neuron NOR gate (NOT OR)" ,
    "" [neuron])
```

In this test case we test the ability of a single neuron to learn the NOR gate.

The NOR gate is a binary operation that returns true if both inputs are false, and false otherwise. With a bias of 0.5 the dot product will be greater than 0 if both inputs are 0

- $x1 = 0, x2 = 0 : -1 * 0 + -1 * 0 + 0.5 = 0.5$
- $x1 = 0, x2 = 1 : -1 * 0 + -1 * 1 + 0.5 = -0.5$
- $x1 = 1, x2 = 0 : -1 * 1 + -1 * 0 + 0.5 = -0.5$
- $x1 = 1, x2 = 1 : -1 * 1 + -1 * 1 + 0.5 = -1.5$

Definition at line 103 of file [test.cpp](#).

7.21.3.4 TEST_CASE() [4/5]

```
TEST_CASE (
    "Neuron NOT gate" ,
    "" [neuron])
```

In this test case we test the ability of a single neuron to learn the NOT gate.

The NOT gate is a unary operation that returns true if the input is false, and false otherwise. With a bias of 1 the dot product will be greater than 0 if the first input is 0

- $x1 = 1 : -2 * 1 + 0 * 1 + 1 = -1$
- $x1 = 0 : -2 * 0 + 0 * 0 + 1 = 1$

Definition at line 83 of file [test.cpp](#).

7.21.3.5 TEST_CASE() [5/5]

```
TEST_CASE (
    "Neuron OR gate" ,
    "" [neuron])
```

In this test case we test the ability of a single neuron to learn the OR gate.

The OR gate is a binary operation that returns true if at least one of the inputs is true, and false otherwise. With a bias of -0.5 the dot product will be greater than 0 if any of the inputs are 1

- $x1 = 0, x2 = 0 : 1 * 0 + 1 * 0 - 0.5 = -0.5$
- $x1 = 0, x2 = 1 : 1 * 0 + 1 * 1 - 0.5 = 0.5$
- $x1 = 1, x2 = 0 : 1 * 1 + 1 * 0 - 0.5 = 0.5$
- $x1 = 1, x2 = 1 : 1 * 1 + 1 * 1 - 0.5 = 1.5$

Definition at line 65 of file [test.cpp](#).

7.22 test.cpp

[Go to the documentation of this file.](#)

```
00001
00011 #define CATCH_CONFIG_MAIN
00012 #include <catch2/catch_all.hpp>
00013
00014 #include "../header/neuron.hpp"
00015 #include "../header/neuronLayer.hpp"
00016 #include "../header/neuronNetwork.hpp"
00017
00034
00045 TEST_CASE("Neuron AND gate", "[neuron]") {
00046     // Create a neuron with weights 1, 1 and bias -1.5
00047
00048     Neuron n({1, 1}, -1.5);
00049     REQUIRE(n.predict({0, 0}) == 0);
```

```

00050     REQUIRE(n.predict({0, 1}) == 0);
00051     REQUIRE(n.predict({1, 0}) == 0);
00052     REQUIRE(n.predict({1, 1}) == 1);
00053 }
00054
00065 TEST_CASE("Neuron OR gate", "[neuron]") {
00066     // Create a neuron with weights 1, 1 and bias -0.5
00067
00068     Neuron n({1, 1}, -0.5);
00069     REQUIRE(n.predict({0, 0}) == 0);
00070     REQUIRE(n.predict({0, 1}) == 1);
00071     REQUIRE(n.predict({1, 0}) == 1);
00072     REQUIRE(n.predict({1, 1}) == 1);
00073 }
00074
00083 TEST_CASE("Neuron NOT gate", "[neuron]") {
00084     // Create a neuron with weights -2 and 0 and bias 1
00085
00086     Neuron n({-2, 0}, 1);
00087     REQUIRE(n.predict({0, 0}) == 1);
00088     REQUIRE(n.predict({0, 1}) == 1);
00089     REQUIRE(n.predict({1, 0}) == 0);
00090     REQUIRE(n.predict({1, 1}) == 0);
00091 }
00092
00103 TEST_CASE("Neuron NOR gate (NOT OR)", "[neuron]") {
00104     // Create a neuron with weights -1, -1 and bias 0.5
00105
00106     Neuron n({-1, -1}, 0.5);
00107     REQUIRE(n.predict({0, 0}) == 1);
00108     REQUIRE(n.predict({0, 1}) == 0);
00109     REQUIRE(n.predict({1, 0}) == 0);
00110     REQUIRE(n.predict({1, 1}) == 0);
00111 }
00112
00137 TEST_CASE("Half Adder using Two-Layer Neuron Network", "[half-adder]") {
00138     // Hidden layer: compute OR and AND
00139     Neuron n_or({1, 1}, -0.5); // OR gate
00140     Neuron n_and({1, 1}, -1.5); // AND gate
00141     NeuronLayer hiddenLayer({n_or, n_and});
00142
00143     // Output layer: compute XOR (for sum) and carry
00144     Neuron n_xor({1, -1}, -0.5);
00145
00146     // Carry = AND; neuron with weights {1, 1} and bias -1.5
00147     Neuron n_carry({1, 1}, -1.5);
00148     NeuronLayer outputLayer({n_xor, n_carry});
00149
00150     // Two-layer network for half adder
00151     NeuronNetwork halfAdder({hiddenLayer, outputLayer});
00152
00153     // Test cases for half adder: {Sum, Carry}
00154     REQUIRE(halfAdder.feedForward({0, 0}) == std::vector<float>{0, 0});
00155     REQUIRE(halfAdder.feedForward({0, 1}) == std::vector<float>{1, 0});
00156     REQUIRE(halfAdder.feedForward({1, 0}) == std::vector<float>{1, 0});
00157     REQUIRE(halfAdder.feedForward({1, 1}) == std::vector<float>{0, 1});
00158 }

```

7.23 testBackpropagation.cpp

```

00001
00011 #define CATCH_CONFIG_MAIN
00012 #include <catch2/catch_all.hpp>
00013
00014 #include "../header/common.hpp"
00015 #include "../header/csv_reader.hpp"
00016 #include "../header/neuron.hpp"
00017 #include "../header/neuronLayer.hpp"
00018 #include "../header/neuronNetwork.hpp"
00019
00020 #include <random>
00021 #include <chrono>
00022
00023
00043
00044 // Function to check new operator
00045 static int s_Allocations = 0;
00046
00047 void* operator new(size_t size)
00048 {
00049     s_Allocations++;
00050     return malloc(size);
00051 }

```

```

00052
00053 // Macro version for when you want to time a block of code in-place.
00054 #define MEASURE_BLOCK(message, code_block) \
00055 { \
00056     auto start = std::chrono::high_resolution_clock::now(); \
00057     code_block; \
00058     auto end = std::chrono::high_resolution_clock::now(); \
00059     auto duration = std::chrono::duration<double, std::milli>(end - start).count(); \
00060     std::cout << message << " took " << duration << " ms" << std::endl; \
00061 }
00062
00063
00064 using namespace Catch::Matchers;
00065
00070 TEST_CASE("Loading digit data", "[backpropagation]") {
00071     // Load the digit data
00072     digitData _digitData = readDigitData<int>();
00073
00074     // Check the size of the data
00075     REQUIRE(_digitData.images.size() == 1797 * 64);
00076     REQUIRE(_digitData.targets.size() == 1797);
00077
00078     // Exit program if the data is not loaded
00079     if (_digitData.images.size() == 0 || _digitData.targets.size() == 0) {
00080         FAIL("Could not load digit dataset");
00081         exit(1);
00082     }
00083
00084     SUCCEED("Successfully loaded digit dataset");
00085 }
00086
00091 TEST_CASE("Testing initialization of the NeuronLayer", "[NeuronLayer]")
00092 {
00093     // create a neuronLayer with 10 neurons
00094     int nNeurons = 10;
00095     NeuronLayer nL(nNeurons, 4);
00096
00097     REQUIRE(nL.getNeurons().size() == nNeurons);
00098     nL.__str__();
00099     SUCCEED("Successfully initialized NeuronLayer");
00100 }
00101
00106 TEST_CASE("Testing initialization of the NeuronNetwork", "[NeuronNetwork]")
00107 {
00108     return;
00109     // Initialize layers sizes
00110     int sizeInput = 4;
00111     int hidden1 = 2;
00112     int hidden2 = 6;
00113     int sizeOutput = 3;
00114
00115     // Initialize the Neural Network
00116     std::vector<int> layers = {sizeInput, hidden1, hidden2, sizeOutput};
00117     std::vector<float> outputMask = {0.0f, 1.0f, 2.0f};
00118
00119     BENCHMARK("Initializing Neural Network"){
00120         NeuronNetwork nn(layers);
00121     };
00122
00123     NeuronNetwork nn(layers);
00124
00125     // Get the neurons
00126     std::vector<NeuronLayer> neuronLayers = nn.getLayers();
00127     // nn.__str__();
00128
00129     // Check the input weights
00130     for(int i = 0; i < neuronLayers.size(); i++) {
00131         NeuronLayer nL = neuronLayers.at(i);
00132         std::vector<Neuron> neurons = nL.getNeurons();
00133
00134         for (Neuron& n : neurons) {
00135             std::vector<float> weights = n.getWeights();
00136
00137             // Get the weight from 1 neuron since all should be same for a initialized layer
00138             float weight = weights[0];
00139             float bias = n.getBias();
00140
00141             // Using require that and withinRel too check floating point numbers.
00142             if (i == 0) {
00143                 SECTION("Input Neurons") {
00144                     REQUIRE(weights.size() == 1);
00145                 }
00146             } else {
00147                 SECTION("Hidden and output Neurons") {
00148                     REQUIRE(weights.size() == layers[i - 1]);
00149                 }
00150             }
00151         }
00152     }

```

```

00151     }
00152 }
00153 }
00154
00155 TEST_CASE("AND neural Network", "[NeuronNetwork][AND]")
00160 {
00161     NeuronNetwork nn({2, 1});
00162     int inputSize = 2;
00163     int targetSize = 1;
00164
00165     std::vector<std::vector<float>> inputs = {
00166         {0.0f, 0.0f},
00167         {0.0f, 1.0f},
00168         {1.0f, 0.0f},
00169         {1.0f, 1.0f}
00170     };
00171     std::vector<std::vector<float>> targets = {
00172         {0.0f},
00173         {0.0f},
00174         {0.0f},
00175         {1.0f}
00176     };
00177
00178     MEASURE_BLOCK("Training the network AND", {
00179         nn.trainInputs2D(inputs, targets, 10000);
00180     });
00181
00182     // Check if the network can predict the correct output
00183     for (int i = 0; i < targets.size(); i++)
00184     {
00185
00186         std::vector<float> prediction = nn.predict(inputs[i]);
00187
00188         printf("For input ");
00189         printVector(inputs[i], " Prediction ");
00190         printVector(prediction, "\n");
00191
00192         if (i == 3) { // Checks for 1 1
00193             CHECK_THAT(prediction[0], WithinAbs(1.0f, 0.05f));
00194         } else { // Checks for 0 0, 0 1, 1 0
00195             CHECK_THAT(prediction[0], WithinAbs(0.0f, 0.05f));
00196         }
00197     }
00198 }
00199
00200 TEST_CASE("XOR Neural Network", "[NeuronNetwork][XOR]")
00205 {
00206     NeuronNetwork nn({2, 2, 1});
00207     int inputSize = 2;
00208     int targetSize = 1;
00209
00210     std::vector<std::vector<float>> inputs = {
00211         {0.0f, 0.0f},
00212         {0.0f, 1.0f},
00213         {1.0f, 0.0f},
00214         {1.0f, 1.0f}
00215     };
00216     std::vector<std::vector<float>> targets = {
00217         {0.0f},
00218         {1.0f},
00219         {1.0f},
00220         {0.0f}
00221     };
00222
00223     MEASURE_BLOCK("Training the network XOR", {
00224         nn.trainInputs2D(inputs, targets, 10000);
00225     });
00226
00227     for (int i = 0; i < targets.size(); i++)
00228     {
00229         std::vector<float> prediction = nn.predict(inputs[i]);
00230
00231         printf("For input ");
00232         printVector(inputs[i], " Prediction ");
00233         printVector(prediction, "\n");
00234
00235         // Check if the network can predict the correct output
00236         if (i == 1 || i == 2) { // Checks for 0 1, 1 0
00237             CHECK_THAT(prediction[0], WithinAbs(1.0f, 0.05f));
00238         } else { // Checks for 0 0, 1 1
00239             CHECK_THAT(prediction[0], WithinAbs(0.0f, 0.05f));
00240         }
00241     }
00242 }
00243
00244 TEST_CASE("Half adder Neuron Network", "[NeuronNetwork][HalfAdder]")
00249 {

```

```

00250     NeuronNetwork nn({2, 3, 2});
00251     int inputSize = 2;
00252     int targetSize = 2;
00253
00254     std::vector<std::vector<float>> inputs = {
00255         {0.0f, 0.0f},
00256         {1.0f, 0.0f},
00257         {0.0f, 1.0f},
00258         {1.0f, 1.0f}
00259     };
00260     std::vector<std::vector<float>> targets = {
00261         {0.0f, 0.0f},
00262         {1.0f, 0.0f},
00263         {1.0f, 0.0f},
00264         {0.0f, 1.0f}
00265     };
00266
00267     MEASURE_BLOCK("Training the network Half Adder", {
00268         nn.trainInputs2D(inputs, targets, 10000);
00269     });
00270
00271     for (int i = 0; i < targets.size(); i++)
00272     {
00273         std::vector<float> prediction = nn.predict(inputs[i]);
00274
00275         printf("For input ");
00276         printVector(inputs[i], " Prediction ");
00277         printVector(prediction, "\n");
00278
00279         // Check if the network can predict the correct output
00280         if (i == 1 || i == 2) { // Checks for 0 1, 1 0
00281             CHECK_THAT(prediction[0], WithinAbs(1.0f, 0.05f));
00282         } else { // Checks for 0 0, 1 1
00283             CHECK_THAT(prediction[0], WithinAbs(0.0f, 0.05f));
00284         }
00285     }
00286 }
00287
00292 TEST_CASE("NeuronNetwork Learning Iris dataset", "[backpropagation][Iris]") {
00293     //
00294     // Load the iris dataset
00295     //
00296     // =====
00297     // Read the iris data set
00298     std::vector<std::vector<std::string>> data = readCsv("../data/iris.csv");
00299
00300     // Extract the features and targets
00301     std::vector<std::vector<float>> features = getFeatures(data);
00302     std::vector<float> targets = getTargets(data);
00303
00304     // We expect 150 examples where each image consists of 4 integer values.
00305     int inputSize = 4;
00306     int outputSize = 3;
00307     // len of features and targets should be the same
00308
00309     // Define a network architecture: an input layer of 4 neurons,
00310     // one hidden layer of 16 neurons, and an output layer of 3 neurons.
00311     std::vector<int> layers = { inputSize, 4, outputSize };
00312     std::vector<float> outputMask = {0.0f, 1.0f, 2.0f};
00313
00314     // Mask the targets
00315     std::vector<float> maskedData = maskData(targets, outputMask);
00316     std::vector<std::vector<float>> targetMaskedData = create2DVector(maskedData, outputSize);
00317
00318     // Create training and test split
00319     normalize2DVector(features);
00320
00321     // Train and test split
00322     TrainTestSplit tts(features, targetMaskedData, 0.90);
00323
00324     //
00325     // =====
00326     // Train the network
00327     // =====
00328     NeuronNetwork nn(layers);
00329
00330     printf("vector size %zu\n", tts.trainFeatures.size());
00331
00332     MEASURE_BLOCK("Training the network", {
00333         nn.trainInputs2D(tts.trainFeatures, tts.trainTargets, 2000);
00334     });
00335
00336     nn.trainInputs2D(tts.trainFeatures, tts.trainTargets, 5000);

```

```

00337
00338 // Print the output mask for the network
00339 printf("\nOutputMask \t ");
00340 printVector(outputMask, "\n");
00341
00342 SECTION("Testing test set") {
00343     for (std::size_t i = 0; i < tts.testFeatures.size(); i++) {
00344         std::vector<float> input = tts.testFeatures[i];
00345         std::vector<float> target = tts.testTargets[i];
00346         std::vector<float> prediction = nn.feedForward(input);
00347         printVector(input, " | ");
00348         printVector(prediction, "\n");
00349         for (std::size_t j = 0; j < prediction.size(); j++) {
00350             if (target[j] > 0.95f) { // Check if the target is 1
00351                 CHECK_THAT(prediction[j], WithinAbs(1.0f, 0.1f));
00352             } else {
00353                 CHECK_THAT(prediction[j], WithinAbs(0.0f, 0.1f));
00354             }
00355         }
00356     }
00357 }
00358
00359 SECTION("Testing Random predictions") {
00360     // Randomly select nToCheck images and check if the network can classify them
00361     int nToCheck = 20;
00362     std::random_device rd;
00363     std::mt19937 gen(rd());
00364     int min = 0, max = tts.testTargets.size() - 1;
00365     std::uniform_int_distribution<> dist(min, max);
00366
00367     for (int i = 0; i < nToCheck; i++) {
00368         int randomIndex = dist(gen);
00369         std::vector<float> input = tts.trainFeatures[randomIndex];
00370         std::vector<float> target = tts.trainTargets[randomIndex];
00371         std::vector<float> prediction = nn.feedForward(input);
00372
00373         printVector(target, " | ");
00374         printVector(prediction, "\n");
00375
00376         for (std::size_t j = 0; j < prediction.size(); j++) {
00377             if (target[j] > 0.95f) { // Check if the target is 1
00378                 CHECK_THAT(1.0f, WithinAbs(prediction[j], 0.1f));
00379             } else {
00380                 CHECK_THAT(0.0f, WithinAbs(prediction[j], 0.1f));
00381             }
00382         }
00383     }
00384 }
00385 }
00386 }
00387
00392 TEST_CASE("NeuronNetwork Learning digit data", "[backpropagation]") {
00393     //
00394     // Load the digit dataset
00395     //
00396     // Load the digit dataset
00397     digitData digits = readDigitData<float>();
00398
00399     // We expect 1797 examples where each image consists of 64 integer values.
00400     int numData = digits.targets.size();
00401     int inputSize = 64;
00402     int outputSize = 10;
00403
00404     // Define a network architecture: an input layer of 64 neurons,
00405     // one hidden layer of 16 neurons, and an output layer of 10 neurons.
00406     std::vector<int> layers = { inputSize, 16, outputSize };
00407     std::vector<float> outputMask = {0.0f, 1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f, 9.0f};
00408
00409     // Mask the targets
00410     std::vector<float> maskedData = maskData(digits.targets, outputMask);
00411
00412     // Create 2D vectors for the target and input data
00413     std::vector<std::vector<float>> targetMaskedData = create2DVector(maskedData, outputSize);
00414     std::vector<std::vector<float>> imagesMaskedData = create2DVector(digits.images, inputSize);
00415     normalize2DVector(imagesMaskedData);
00416
00417     // Train and test split
00418     TrainTestSplit tts(imagesMaskedData, targetMaskedData, 0.90);
00419
00420     //
00421     // Train the network
00422     //
00423

```



```

00424
00425     NeuronNetwork nn(layers);
00426
00427     printf("vector size for test features %zu\n", tts.testFeatures.size());
00428
00429     MEASURE_BLOCK("Training the network", {
00430         nn.trainInputs2D(tts.trainFeatures, tts.trainTargets, 2000);
00431     });
00432
00433     nn.trainInputs2D(tts.trainFeatures, tts.trainTargets, 5000);
00434
00435     // Number of features to check
00436     SECTION("Testing test set") {
00437         printf("Testing the test split for digit dataset\n");
00438         for (std::size_t i = 0; i < tts.testFeatures.size(); i++) {
00439             std::vector<float> input = tts.testFeatures[i];
00440             std::vector<float> target = tts.testTargets[i];
00441             std::vector<float> prediction = nn.feedForward(input);
00442             printVector(target, " | ");
00443             printVector(prediction, "\n");
00444
00445             for (std::size_t j = 0; j < prediction.size(); j++) {
00446                 if (target[j] > 0.95f) { // Check if the target is 1
00447                     CHECK_THAT(prediction[j], WithinAbs(1.0f, 0.1f));
00448                 } else {
00449                     CHECK_THAT(prediction[j], WithinAbs(0.0f, 0.1f));
00450                 }
00451             }
00452         }
00453     }
00454
00455     SECTION("Testing random predictions") {
00456         printf("Testing random predictions for digit dataset\n");
00457         // Randomly select nToCheck images and check if the network can classify them
00458         int nToCheck = 20;
00459         std::random_device rd;
00460         std::mt19937 gen(rd());
00461         int min = 0, max = tts.testTargets.size() - 1;
00462         std::uniform_int_distribution<> dist(min, max);
00463
00464         // Print the output mask for the network
00465         printf("\nOutputMask \t\t\t");
00466         printVector(outputMask, "\n");
00467
00468         for (int i = 0; i < nToCheck; i++) {
00469             int randomIndex = dist(gen);
00470
00471             std::vector<float> input = tts.trainFeatures[randomIndex];
00472             std::vector<float> target = tts.trainTargets[randomIndex];
00473             float targetValue = digits.targets[randomIndex];
00474
00475             // Predict the output
00476             std::vector<float> prediction = nn.feedForward(input);
00477
00478             // Print the predictions
00479             printf("Prediction for target %.2f | ", targetValue);
00480             printVector(prediction, "\n");
00481
00482             // Find the target in the target mask
00483             int targetIndex = static_cast<int>(targetValue);
00484
00485             // NOTE: The prediction is a probability, it can be false.
00486             CHECK_THAT(1.0f, WithinRel(prediction[targetIndex], 0.1f));
00487         }
00488     }
00489 }

```

