# ML perceptron sigmoid neuron (P3)

v0.1

Generated by Doxygen 1.13.2

# 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-perceptron

## 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

## 2.4 Installing

Enter the test dir then

Generate build files:

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

Build the project:

```
cmake --build build
```

Run the executable:

```
./build/MLPerceptronTest
```

# Chapter 3

# Hierarchical Index

## 3.1 Class Hierarchy

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

# Chapter 4

# Class Index

## 4.1 Class List

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

# Chapter 5

# File Index

## 5.1 File List

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

# 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 40 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:

- /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/common.hpp

## 6.2 irisData Struct Reference

A structure to hold the features and targets read from a CSV file. This is fdor 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 fdor 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 26 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:

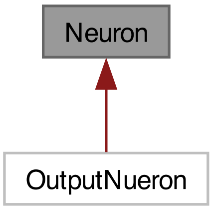
- /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/common.hpp

## 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)

    *constructer 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)
```

constructer 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**

| | |
|---|---|
| *weights* | A vector of weights for the neuron. |
| *bias* | 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**

| | |
|---|---|
| *inputs* | 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**

| | |
|---|---|
| *output* | The output of the neuron. |
| *target* | The target value. |

**Returns**

The error for the ou tput 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**

| | |
|---|---|
| *output* | The output of the neuron. |
| *target* | 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**

| | |
|---|---|
| *inputs* | A vector of input values. |
| *target* | 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**

| | |
|---|---|
| *inputs* | 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**

| | |
|---|---|
| *x* | 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**

| | |
|---|---|
| *output* | The output of the neuron. |
| *target* | 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:

- /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/neuron.hpp
- /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/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 computeOutputErros (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**

| | |
|---|---|
| *neurons* | 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**

| | |
|---|---|
| *nNeurons* | The number of neurons in the layer. |
| *nSizeWeights* | 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**

| | |
|---|---|
| *inputs* | A vector of input values. |
| *neuronsNextLayer* | 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**

| | |
|---|---|
| *targets* | 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**

| | |
|---|---|
| *inputs* | 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:

- /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/neuronLayer.hpp
- /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/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**

| | |
|---|---|
| *layers* | 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**

| | |
|---|---|
| *inputs* | 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**

| | |
|---|---|
| *inputs* | 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**

| | |
|---|---|
| *inputs* | 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**

| | |
|---|---|
| *targets* | 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**

| | |
|---|---|
| *inputs* | A vector of input values. |
| *targets* | A vector of target values. |
| *inputSize* | The size of the input values. |
| *maxTrainingSamples* | 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.

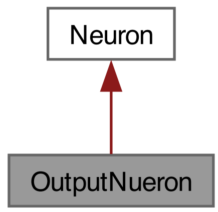
Definition at line 74 of file neuronNetwork.cpp.

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

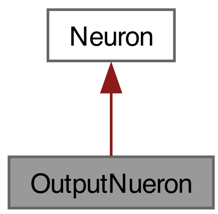
- /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/neuronNetwork.hpp
- /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/neuronNetwork.cpp

## 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:

- /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/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:

- 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 71 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 75 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 87 of file common.hpp.

#### 6.7.3.2 testTargets

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

Definition at line 89 of file common.hpp.

#### 6.7.3.3 trainFeatures

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

Definition at line 86 of file common.hpp.

#### 6.7.3.4 trainTargets

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

Definition at line 88 of file common.hpp.

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

- /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/common.hpp

# Chapter 7

# File Documentation

## 7.1 /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/data/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**

| | |
|---|---|
| *data* | A numpy array containing the data to write. |
| *filename* | 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
00007
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()
```

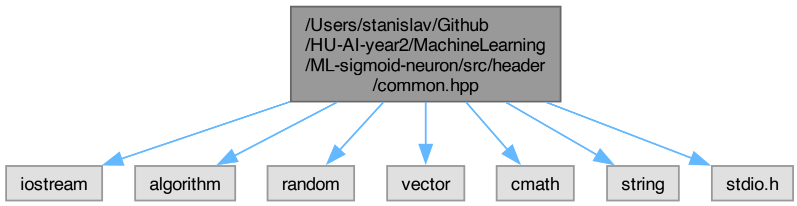
## 7.3 /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/common.hpp File Reference

In this file the common utilities are defined.

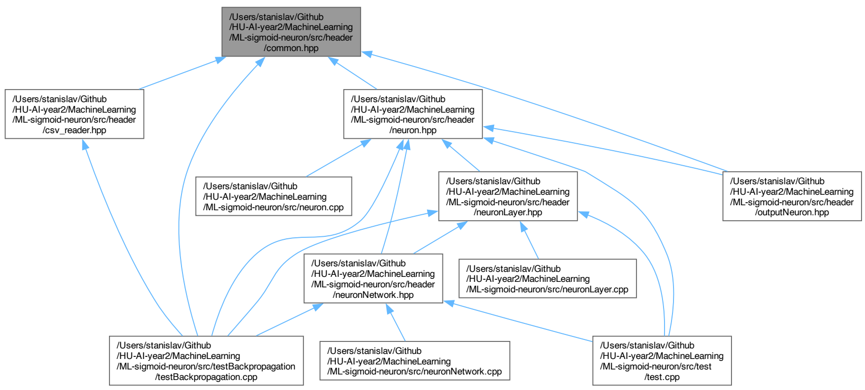
```
#include <iostream>
#include <algorithm>
#include <random>
#include <vector>
#include <cmath>
#include <string>
```

```
#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 fdor 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)
- 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.*

## 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)
```

Definition at line 174 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**

| | |
|---|---|
| *learningRate* | The learning rate of the network. |
| *deltaGradient* | The gradient of the neuron. |

**Returns**

float The delta bias.

Definition at line 123 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**

| | |
|---|---|
| *learningRate* | The learning rate of the network. |
| *gradient* | The gradient of the neuron. |

**Returns**

float The delta gradient.

Definition at line 111 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**

| | |
|---|---|
| *output* | The output of the neuron. |
| *error* | The error of the neuron. |

**Returns**

float The gradient between the neurons.

Definition at line 99 of file common.hpp.

### 7.3.3.5 normalizeVector()

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

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

**Parameters**

| | |
|---|---|
| *vec* | The 1D vector to be converted. |
| *size* | The size of the inner vectors. |

**Returns**

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

Definition at line 160 of file common.hpp.

### 7.3.3.6 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 136 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
00026 struct irisData
00034 {
00035     std::vector<std::vector<float» features;
00036     std::vector<float> targets;
00037 };
00038
00039 template<typename T>
00040 struct digitData
00048 {
00049     std::vector<T> images;
00050     std::vector<T> targets;
00051 };
00052 // Add forward declaration for TrainTestSplit
00053 template<typename T>
00054 struct TrainTestSplit;
00055
00056 // Forward declaration
00057 template<typename T>
00058 TrainTestSplit<T> createTrainTestSplit(
00059     const std::vector<std::vector<T»& features,
00060     const std::vector<std::vector<T»& targets,
00061     float splitRatio);
00062
00070 template<typename T>
00071 struct TrainTestSplit
00072 {
00073     // Constructors
00074     TrainTestSplit() = default;
00075     TrainTestSplit(const std::vector<std::vector<T»& features,
00076         const std::vector<std::vector<T»& targets,
00077         float splitRatio)
00078     {
00079         TrainTestSplit<T> tts = createTrainTestSplit(features, targets, splitRatio);
00080         trainFeatures = tts.trainFeatures;
00081         testFeatures = tts.testFeatures;
00082         trainTargets = tts.trainTargets;
00083         testTargets = tts.testTargets;
00084     }
00085
00086     std::vector<std::vector<T» trainFeatures;
00087     std::vector<std::vector<T» testFeatures;
00088     std::vector<std::vector<T» trainTargets;
00089     std::vector<std::vector<T» testTargets;
00090 };
00091
```

```
inline float gradientBetweenNeurons(float& output, float& error)
{
    return output * error;
}

inline float deltaGradient(float& learningRate, float& gradient)
{
    return learningRate * gradient;
}

inline float deltaBias(float& learningRate, float& deltaGradient)
{
    return learningRate * deltaGradient;
}


template<typename T>
inline void printVector(const std::vector<T> &vec, const std::string extra = "")
{
    if constexpr (std::is_floating_point_v<T>) {
        for (const T &item : vec) {
            printf("%.2f ", item);
        }
    } else if constexpr (std::is_integral_v<T>) {
        for (const T &item : vec) {
            printf("%i ", item);
        }
    }
    if (!extra.empty()) {
        printf("%s", extra.c_str());
    }
}

template<typename T>
void normalizeVector(std::vector<T> &vec)
{
    T maxElement = *std::max_element(vec.begin(), vec.end());
    T minElement = *std::min_element(vec.begin(), vec.end());
    T minMax = maxElement - minElement;

    // https://www.statology.org/normalize-data-between-0-and-1/
    for (std::size_t i = 0; i < vec.size(); i++) {
        T xi = vec[i];
        vec.at(i) = (xi - minElement) / minMax;
    }
}

template<typename T>
TrainTestSplit<T> createTrainTestSplit(
    const std::vector<std::vector<T>>& features,
    const std::vector<std::vector<T>>& targets,
    float splitRatio)
{
    TrainTestSplit<T> tts;
    // Reserve the memory for the vectors
    tts.trainFeatures.reserve(features.size() * splitRatio);
    tts.testFeatures.reserve(features.size() * (1 - splitRatio));
    tts.trainTargets.reserve(targets.size() * splitRatio);
    tts.testTargets.reserve(targets.size() * (1 - splitRatio));

    for (std::size_t i = 0; i < features.size(); i++)
    {
        if (i < features.size() * splitRatio) {
            tts.trainFeatures.emplace_back(features[i]);
            tts.trainTargets.emplace_back(targets[i]);
        } else {
            tts.testFeatures.emplace_back(features[i]);
            tts.testTargets.emplace_back(targets[i]);
        }
    }
    return tts;
}
```

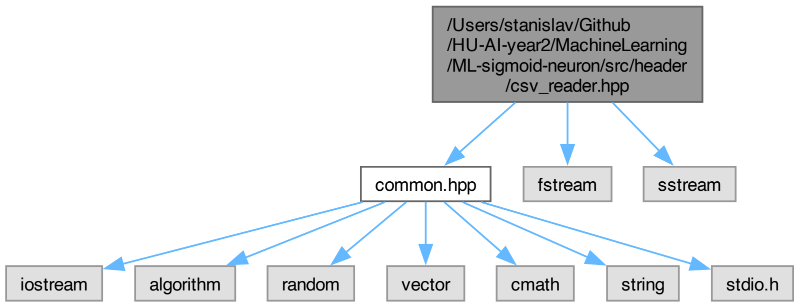
## 7.5 /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/csv_reader.hpp File Reference

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

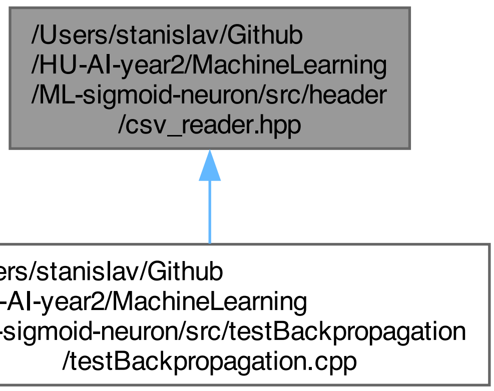
```
#include "common.hpp"
#include <fstream>
```

```
#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 > > read_csv (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 > realdCsvFlat (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**

| | |
|---|---|
| *filename* | 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**

| | |
|---|---|
| *features* | A vector of vectors containing the feature data. |
| *targets* | A vector containing the target values corresponding to the feature data. |
| *target* | 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**

| | |
|---|---|
| *data* | 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**

| | |
|---|---|
| *data* | 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 read_csv()

```
std::vector< std::vector< std::string > > read_csv (
            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**

| | |
|---|---|
| *filename* | The name of the CSV file to read. |
| *delimiter* | 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 realdCsvFlat()

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

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

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

**Template Parameters**

| | |
|---|---|
| *T* | The type to convert the CSV tokens into. |

**Parameters**

| | |
|---|---|
| *filename* | The name of the CSV file to read. |
| *delimiter* | 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.

```
#pragma once
#include "common.hpp"
#include <fstream>
#include <sstream>


// ==================================================================================================
// Section: reading iris data
// ==================================================================================================

std::vector<std::vector<std::string» read_csv(const std::string& filename, char delimiter=',')
{
    // Create a vector to store the rows
    std::vector<std::vector<std::string» rows;
    std::ifstream file(filename);

    // Check if the file is open
    if (!file.is_open()) {
        std::cerr « "Error: Could not open file " « filename « std::endl;
        return rows;
    }

    // Read the file line by line
    std::string line;
    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::vector<std::string> cols;
        std::string col;

        while (std::getline(ss, col, delimiter)) {
            cols.push_back(col);
        }

        // Add the columns to the rows
        rows.push_back(cols);
    }

    // Close the file
    file.close();

    return rows;
}


std::vector<float> getTargets(const std::vector<std::vector<std::string»& data)
{
    std::vector<float> targets;
    for (const auto& row : data) {
        targets.push_back(std::stof(row.back()));
    }
    return targets;
}

std::vector<std::vector<float» getFeatures(const std::vector<std::vector<std::string»& data)
{
    std::vector<std::vector<float» features;
    for (const auto& row : data) {
        std::vector<float> feature_row;
        // Skip the last column which contains the target
        for (int i = 0; i < row.size() - 1; i++) {
            feature_row.push_back(std::stof(row[i]));
        }
        features.push_back(feature_row);
    }
    return features;
}

irisData filterData(const std::vector<std::vector<float»& features, const std::vector<float>& targets, int target)
{
    std::vector<std::vector<float» filtered_features;
    std::vector<float> filtered_targets;
    for (int i = 0; i < features.size(); i++) {
        if (targets[i] != target) {
            filtered_features.push_back(features[i]);
            filtered_targets.push_back(targets[i]);
        }
    }
    return irisData{filtered_features, filtered_targets};
}

```

```
// ==============================================================================================
// Section: reading digits data
// ==============================================================================================


// Helper conversion function template with specializations.
template<typename T>
T convert(const std::string& str);

template<>
int convert<int>(const std::string& str) {
    return std::stoi(str);
}

template<>
float convert<float>(const std::string& str) {
    return std::stof(str);
}

template<typename T>
std::vector<T> realdCsvFlat(const std::string& filename, char delimiter = ',')
{
    std::vector<T> data;
    std::ifstream file(filename);

    if (!file.is_open()) {
        std::cerr « "Error: Could not open file " « filename « std::endl;
        return data;
    }

    std::string line;
    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::string token;

        while (std::getline(ss, token, delimiter)) {
            data.push_back(convert<T>(token));
        }
    }
    file.close();
    return data;
}

template<typename T>
std::vector<std::vector<T» create2DVector(const std::vector<T> vec, int size)
{
    std::vector<std::vector<T» data;
    data.reserve(vec.size() / size);
    for (int i = 0; i < vec.size(); i += size) {
        std::vector<T> row;
        for (int j = 0; j < size; j++) {
            row.push_back(vec[i + j]);
        }
        data.push_back(row);
    }

    return data;
}

template<typename T>
digitData<T> readDigitData()
{
    // Not the best way to do this, but it works for now
    // presumes that the data is in the data folder in the root of the project
    const std::string filenameImages = "../../data/digits_images.csv";
    const std::string filenameTargets = "../../data/digits_targets.csv";

    // Load the data
    digitData<T> data;
    std::vector<T> images = realdCsvFlat<T>(filenameImages);
    normalizeVector(images);
    data.images = images;
    data.targets = realdCsvFlat<T>(filenameTargets);

    return data;
}

template<typename T>
std::vector<T> maskData(std::vector<T>& data, std::vector<T>& mask)
{
    T off, on;
    if constexpr (std::is_floating_point_v<T>) {
        off = 0.1f;
        on  = 1.0f;
    } else if constexpr (std::is_integral_v<T>) {
        off = 0;
```

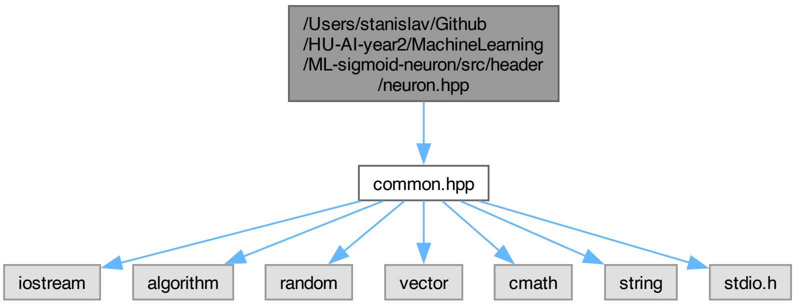
```
        on  = 1;
    } else {
        throw std::runtime_error("Data type not supported");
    }

    std::vector<T> maskedData;
    maskedData.reserve(data.size() * mask.size());
    for (std::size_t i = 0; i < data.size(); i++)
    {
        for (std::size_t j = 0; j < mask.size();j++)
        {
            if (data.at(i) == mask.at(j)) {
                maskedData.emplace_back(on);
            } else {
                maskedData.emplace_back(off);
            }
        }
    }
    return maskedData;
}
```

## 7.7 /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/neuron.hpp File Reference

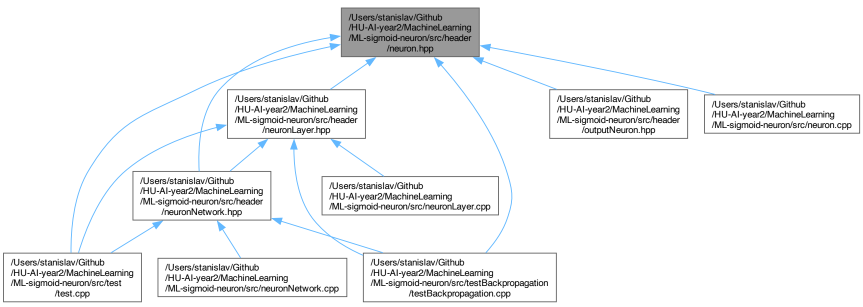
In this file the Neuron class is declared. This class represents a single neuron in a neural network.

`#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.

```

#pragma once
#include "common.hpp"
class Neuron {
private:
    std::vector<float> _weights;
    std::vector<float> _lastInput;
    float _bias;
    float _learningRate;
    float _lastOutput;
    float _delta;

public:
    Neuron(int nSizeWeights, float initialWeight, float initialBias);

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

    float sigmoid(float x);

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

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

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

    void update();

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

    float sigmoidDerivative(float output);
```

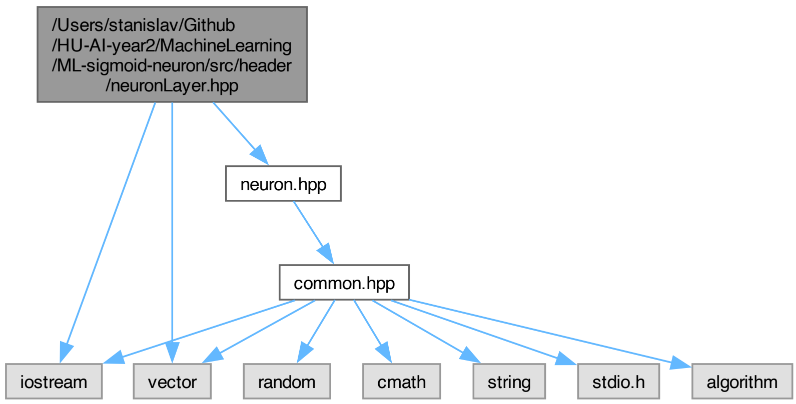
```
    float computeOutputDelta(float target);

    const std::vector<float>& getWeights() const { return _weights; }

    float getBias() const { return _bias; }

    float getError() const { return _delta; }

    void __str__() const;



};
```

## 7.9 /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/neuronLayer.hpp File Reference

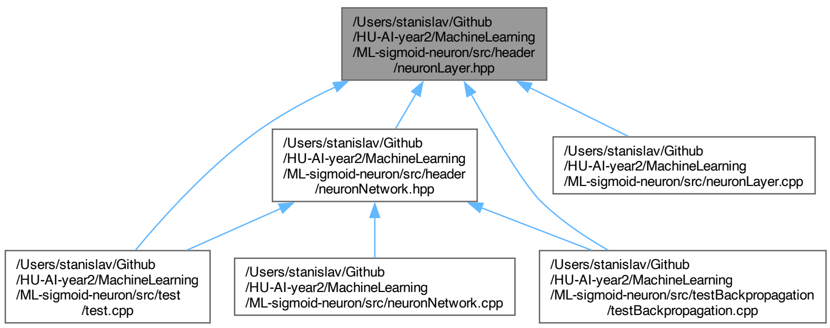
In this file the NeuronLayer class is declared. This class represents a layer of neurons in a neural network.

```
#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**



Definition in file neuronLayer.hpp.

# 7.10 neuronLayer.hpp

Go to the documentation of this file.

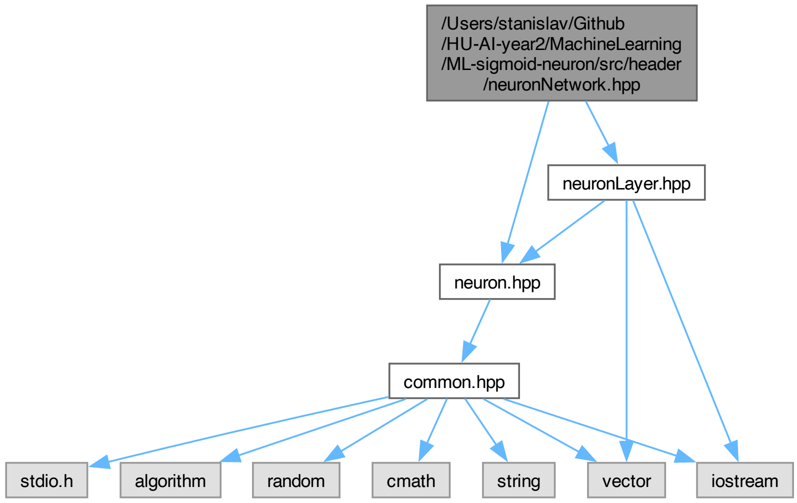
```

#pragma once
#include <iostream>
#include <vector>
#include "neuron.hpp"

class NeuronLayer {
private:
    std::vector<Neuron> _neurons;
    std::vector<float> _output;

public:
    NeuronLayer(std::vector<Neuron> neurons);

    NeuronLayer(int nNeurons, int nSizeWeights);

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

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

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

    void update();

    const std::vector<Neuron>& getNeurons() { return _neurons; }

    const std::vector<float>& getOutput() const { return _output; }

    void __str__() const;
};
```

## 7.11 /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/neuronNetwork.hpp File Reference

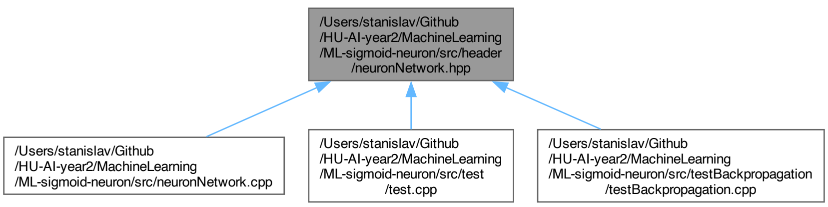
In this file the NeuronNetwork class is declared. This class represents a neural network with multiple layers of neurons.

```
#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.

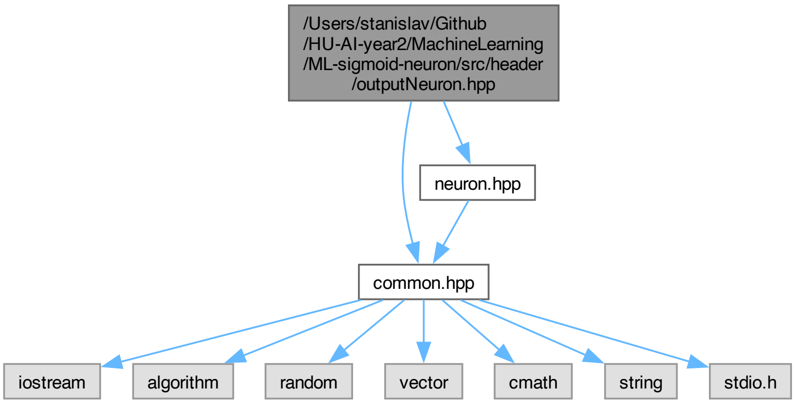
```
#pragma once
#include "neuron.hpp"
#include "neuronLayer.hpp"

class NeuronNetwork {
private:
    std::vector<NeuronLayer> _layers;
    std::vector<float> _currentTargets;

    std::vector<float> _inputVec;
    std::vector<float> _currentLayerOutput;
    std::vector<float> _tempOutputBuffer;

    std::vector<float> _outputMask;
public:
    NeuronNetwork(std::vector<NeuronLayer> layers);

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


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

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

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

    void maskTarget(float target);
    void update();
    void trainInputs2D(const std::vector<std::vector<float»& inputs, const
std::vector<std::vector<float»& targets, int epochs);

    std::vector<NeuronLayer> getLayers() const { return _layers; }

    void setTarget(std::vector<float>& targets) {_currentTargets = targets;}

    void __str__() const;


};
```

## 7.13 /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/header/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.

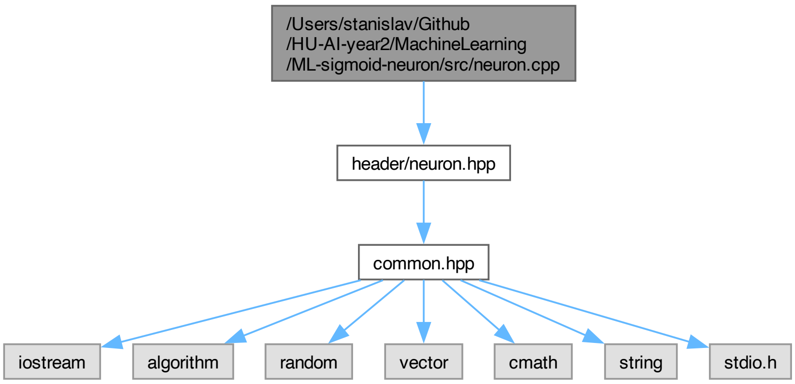
```
#include "common.hpp"
#include "neuron.hpp"

class OutputNueron : Neuron
{
public:

private:

};
```

## 7.15 /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/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**



Definition in file neuron.cpp.

## 7.16 neuron.cpp

Go to the documentation of this file.

```cpp

#include "header/neuron.hpp"

Neuron::Neuron(int nSizeWeights, float initialWeight, float initialBias)
{
    // Initialize the weights and bias
    _weights = std::vector<float>(nSizeWeights, initialWeight);
    _lastInput.reserve(nSizeWeights);

    // Initialize the delta and learning rate
    _delta = 0.0f;
    _bias = initialBias;
    _learningRate = 0.5f;
    _lastOutput = 0.0f;
}

Neuron::Neuron(const std::vector<float>& weights, float bias, float learningRate)
    : _weights(weights), _bias(bias), _learningRate(learningRate) {}

float Neuron::sigmoid(float x)
{
    // Sigmoid activation function
    return 1 / (1 + exp(-x));
}

float Neuron::activate(const std::vector<float>& inputs)
{
    // Calculate the weighted sum of the inputs
    _lastInput = inputs;
    float weightedSum = _bias;

    // Dot product of the weights and inputs
    for (std::size_t i = 0; i < _weights.size(); i++)
    {
        weightedSum += _weights[i] * inputs[i];
    }

    // Return the result of the sigmoid function
    _lastOutput = sigmoid(weightedSum);
    return _lastOutput;
}

float Neuron::predict(const std::vector<float>& inputs)
{
    // Return 1 if the result is greater than 0.5, otherwise return 0(threshold)
    return (activate(inputs) > 0.5) ? 1 : 0;
}

void Neuron::update()
{
    // Update the weights and bias
    for (std::size_t i = 0; i < _weights.size(); i++)
    {
        _weights[i] -= _learningRate * _lastInput[i] * _delta;
    }
    // Update the bias
    _bias -= _learningRate * _delta;
}

float Neuron::computeHiddenDelta(const std::vector<float>& inputs, float sum)
{
    _delta = sigmoidDerivative(_lastOutput) * sum;;
    return _delta;
```

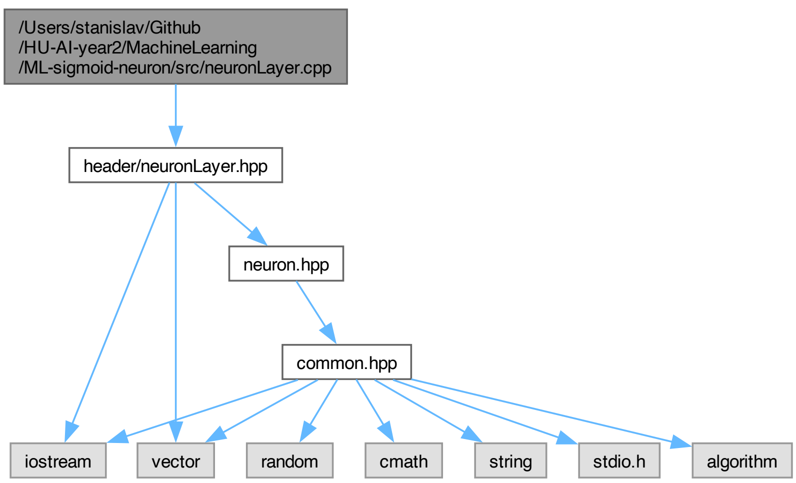
```
}

float Neuron::computeOutputDelta(float target)
{
    _delta = sigmoidDerivative(_lastOutput) * -(target - _lastOutput);
    return _delta;
}

float Neuron::sigmoidDerivative(float output)
{
    return output * (1 - output);
}

void Neuron::__str__() const
{
    // Print the neuron details
    printf("\nNeurons with %zu weights: ", _weights.size());
    printVector(_weights);
    printf("| Bias = %f | Learning rate = %f", _bias, _learningRate);
}
```

## 7.17 /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/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**



Definition in file neuronLayer.cpp.

## 7.18 neuronLayer.cpp

Go to the documentation of this file.

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

NeuronLayer::NeuronLayer(std::vector<Neuron> neurons)
    : _neurons(neurons) {}

NeuronLayer::NeuronLayer(int nNeurons, int nSizeWeights)
{
    _output.resize(nNeurons);
    // nNeurons check
    if (nNeurons == 0) {
        printf("nNeuron must be atleast 1 is %d", nNeurons) ;
        return;
    }

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0.1f, 1.0f);

    _neurons.reserve(nNeurons);
    for (std::size_t i = 0; i < nNeurons; i++)
    {
        _neurons.emplace_back(nSizeWeights, dis(gen), dis(gen));
    }
}

std::vector<float>& NeuronLayer::feedForward(const std::vector<float>& inputs)
{
    // Feed forward through each neuron in the layer
    for (std::size_t i = 0; i < _neurons.size(); i++)
    {
        // For now using the activate instead of predict.
        // The predict function is used for binary classification i think.
        _output[i] = _neurons[i].activate(inputs);
    }
    return _output;
}

void NeuronLayer::computeOutputErros(const std::vector<float> &targets)
{
    // Will only run for the output neurons
    for (std::size_t i = 0; i < targets.size(); i++) {
        _neurons[i].computeOutputDelta(targets[i]);
    }
}

void NeuronLayer::computeHiddenErrors(const std::vector<float>& inputs, const std::vector<Neuron>&
neuronsNextLayer)
{
    // // Simply get the first neurons weight size
    for (std::size_t i = 0; i < _neurons.size(); i++) {

        float sum = 0.0f;

        // Loop over neurons in next layer
        for (std::size_t j = 0; j < neuronsNextLayer.size(); j++)
```

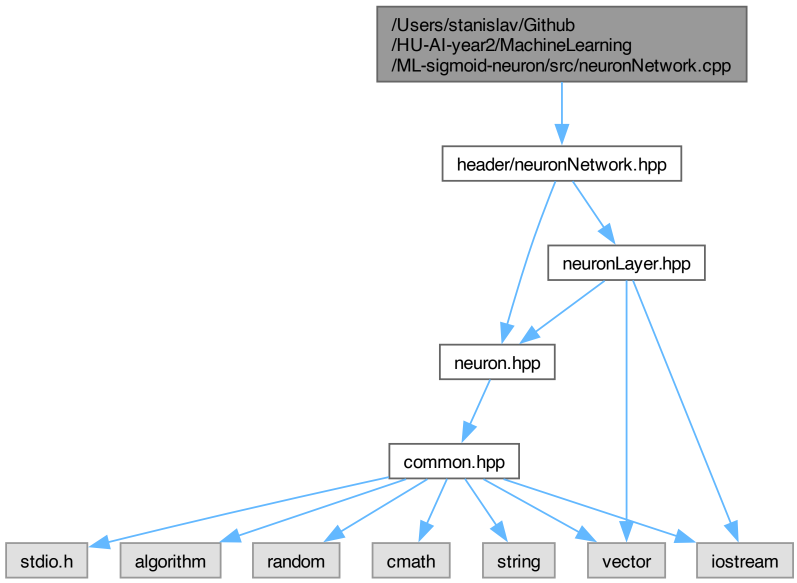
```
        {
            sum += neuronsNextLayer[j].getWeights()[i] * neuronsNextLayer[j].getError();
        }

        _neurons[i].computeHiddenDelta(inputs, sum);
    }
}

void NeuronLayer::update()
{
    for (Neuron& n : _neurons)
    {
        n.update();
    }
}

void NeuronLayer::__str__() const
{
    // Print the layer details
    printf("\nNeuronLayer with %zu neurons", _neurons.size());
    for (std::size_t i = 0; i < _neurons.size(); i++)
    {
        _neurons[i].__str__();
    }
    printf("\n");
}
```

## 7.19 /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/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.

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

NeuronNetwork::NeuronNetwork(std::vector<NeuronLayer> layers)
    : _layers(layers) {}

NeuronNetwork::NeuronNetwork(std::vector<int> layerSizes)
{
    // Reserve the input vector and the current targets
    _inputVec.resize(layerSizes.front());
    _currentTargets.resize(layerSizes.back());

    // Reserve because there is no default constructor for NeuronLayer
    _layers.reserve(layerSizes.size());

    // Reserve the temp output buffer and the current layer output
    _tempOutputBuffer.resize(*std::max_element(layerSizes.begin(), layerSizes.end()));
    _currentLayerOutput.resize(*std::max_element(layerSizes.begin(), layerSizes.end()));
    // Create the layers
    for (std::size_t i = 1; i < layerSizes.size(); i++) {
        // If its the first layer then the input size is the first element in the layerSizes
        if (i == 1) {
            _layers.emplace_back(layerSizes[i], layerSizes.front());
        } else {
            _layers.emplace_back(layerSizes[i], layerSizes[i-1]);
        }
    }
}

const std::vector<float>& NeuronNetwork::feedForward(const std::vector<float>& inputs)
{
    // Set the input vector and the current layer output
    _inputVec = inputs;
    _currentLayerOutput = inputs;

    // Feed forward through each layer in the network
    for (std::size_t i = 0; i < _layers.size(); i++) {
        _currentLayerOutput = _layers[i].feedForward(_currentLayerOutput);
    }
    return _currentLayerOutput;
}

std::vector<float> NeuronNetwork::predict(const std::vector<float>& input)
{
    return feedForward(input);
}

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

```
{
    // Compute the output errors
    int last = _layers.size() -1;
    _layers[last].computeOutputErros(targets);

    // Reverse loop For hidden layers
    for (int i = last-1; i > -1; i--) {
        // If is output neuron compute the output error
        if (i == 0) { // If i == 0 then its the input layer
            _layers[i].computeHiddenErrors(_inputVec, _layers[i + 1].getNeurons());
        } else { // Else compute the hidden error
            _layers[i].computeHiddenErrors(_layers[i-1].getOutput(), _layers[i + 1].getNeurons());
        }
    }
}

void NeuronNetwork::update()
{
    for (NeuronLayer& nL : _layers) {
        nL.update();
    }
}

void NeuronNetwork::trainInputs2D(const std::vector<std::vector<float»& inputs, const std::vector<std::vector<float»& targets, int epochs)
{
    // Check if the flat input is the same as the targets
    if (!((inputs.size()) == targets.size())) {
        throw std::runtime_error("Input and target size are not the same");
    }

    // Loop over the epochs
    for (int x = 0; x < epochs; x++) {
        // Loop over each input and target
        for (std::size_t i = 0; i < targets.size(); i++) {
            feedForward(inputs[i]);             // Feed forward
            backPropagation(targets[i]);        // Back propagate
            update();                           // Update the weights
        }
    }
}

// void NeuronNetwork::trainInputs(const std::vector<std::vector<float»& inputs, const std::vector<std::vector<float»& targets,
//     int inputSize, int targetSize, int epochs)
// {
//     // Check if the flat input is the same as the targets
//     // if (!((inputs.size() / inputSize) == targets.size() / targetSize)) {
//     //     throw std::runtime_error("Input and target size are not the same");
//     // }

//     std::vector<float> input(inputSize);
//     // std::vector<float> target(targetSize);

//     // Loop over the epochs
//     for (int x = 0; x < epochs; x++) {
//         // Loop over each input and target
//         for (std::size_t i = 0; i < targets.size(); i++) {
//             // Set the input for the network
//             std::size_t startIndexInput = i * inputSize;

//             // for (std::size_t j = 0; j < inputSize; j++) {
//             //     input[j] = inputs[startIndexInput + j];
//             // }

//             // Set the target for the network
//             // std::size_t startIndexTarget =  i * targetSize;
//             // for (std::size_t j = 0; j < targetSize; j++) {
//             //     _target[j] = targets[startIndexTarget + j];
//             // }

//             // maskTarget(targets[i]); // Set the target for the network
//             feedForward(inputs[i]);      // Feed forward
//             backPropagation(targets[i]);       // Back propagate
//             update();                 // Update the weights
//         }
//     }
// }

void NeuronNetwork::maskTarget(float target)
{
    // if (_outputMask.size() != _currentTargets.size()) {
    //     throw std::runtime_error("OutputMask and current targets are not the same size");
    //     exit(1);
    // }


```

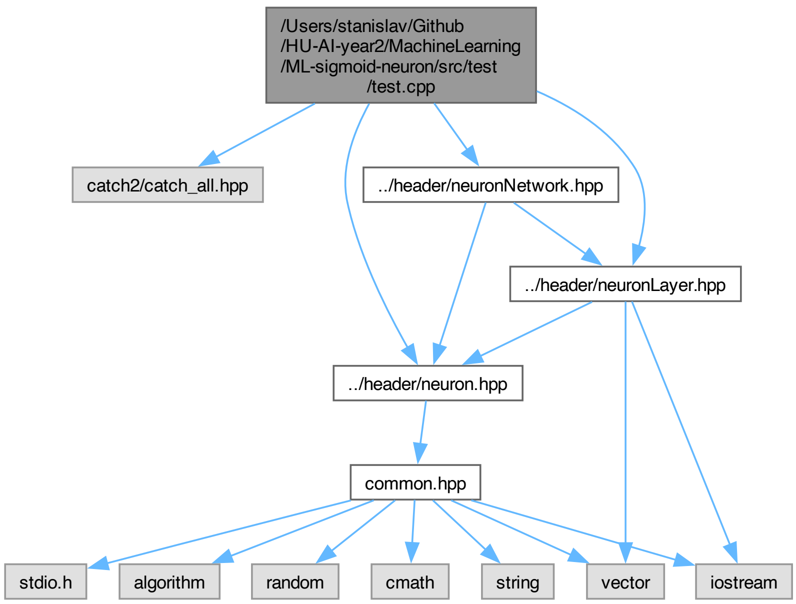
```
    // Set the target to the current target
    for (std::size_t i = 0; i < _outputMask.size(); i++) {
        if (_outputMask[i] == target) {
            _currentTargets[i] = 1.0f;
        } else {
            _currentTargets[i] = 0.0f;
        }
    }
}

void NeuronNetwork::__str__() const
{
    // Print the network details
    printf("\nNeuronNetwork with %zu layers\n", _layers.size());
    for (std::size_t i = 0; i < _layers.size(); i++)
    {
        _layers[i].__str__();
    }
}
```

## 7.21 /Users/stanislav/Github/HU-AI-year2/MachineLearning/ML-sigmoid-neuron/src/test/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**



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, 0}) == 1);
00089     REQUIRE(n.predict({1, 0}) == 0);
00090     REQUIRE(n.predict({1, 0}) == 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);
```

```
    REQUIRE(n.predict({1, 1}) == 0);
}

TEST_CASE("Half Adder using Two-Layer Neuron Network", "[half-adder]") {
    // Hidden layer: compute OR and AND
    Neuron n_or({1, 1}, -0.5);   // OR gate
    Neuron n_and({1, 1}, -1.5);  // AND gate
    NeuronLayer hiddenLayer({n_or, n_and});

    // Output layer: compute XOR (for sum) and carry
    Neuron n_xor({1, -1}, -0.5);

    // Carry = AND; neuron with weights {1, 1} and bias -1.5
    Neuron n_carry({1, 1}, -1.5);
    NeuronLayer outputLayer({n_xor, n_carry});

    // Two-layer network for half adder
    NeuronNetwork halfAdder({hiddenLayer, outputLayer});

    // Test cases for half adder: {Sum, Carry}
    REQUIRE(halfAdder.feedForward({0, 0}) == std::vector<float>{0, 0});
    REQUIRE(halfAdder.feedForward({0, 1}) == std::vector<float>{1, 0});
    REQUIRE(halfAdder.feedForward({1, 0}) == std::vector<float>{1, 0});
    REQUIRE(halfAdder.feedForward({1, 1}) == std::vector<float>{0, 1});
}
```

## 7.23 testBackpropagation.cpp

```

#define CATCH_CONFIG_MAIN
#include <catch2/catch_all.hpp>

#include "../header/common.hpp"
#include "../header/csv_reader.hpp"
#include "../header/neuron.hpp"
#include "../header/neuronLayer.hpp"
#include "../header/neuronNetwork.hpp"

#include <random>
#include <chrono>

// Function to check new operator
static int s_Allocations = 0;

void* operator new(size_t size)
{
    s_Allocations++;
    return malloc(size);
}

// Macro version for when you want to time a block of code in-place.
#define MEASURE_BLOCK(message, code_block)                  \
    {                                                       \
        auto start = std::chrono::high_resolution_clock::now(); \
        code_block;                                         \
        auto end = std::chrono::high_resolution_clock::now();   \
        auto duration = std::chrono::duration<double, std::milli>(end - start).count();    \
        std::cout « message « " took " « duration « " ms" « std::endl; \
    }


using namespace Catch::Matchers;

TEST_CASE("Loading digit data", "[backpropagation]") {
    // Load the digit data
    digitData _digitData = readDigitData<int>();

    // Check the size of the data
    REQUIRE(_digitData.images.size() == 1797 * 64);
    REQUIRE(_digitData.targets.size() == 1797);
    SUCCEED("Successfully loaded digit dataset");
}

TEST_CASE("Testing initialization of the NeuronLayer", "[NeuronLayer]")
{
    // create a neuronLayer with 10 neurons
    int nNeurons = 10;
    NeuronLayer nL(nNeurons, 4);

    REQUIRE(nL.getNeurons().size() == nNeurons);
    nL.__str__();
    SUCCEED("Successfully initialized NeuronLayer");
}
```

```
TEST_CASE("Testing initialization of the NeuronNetwork", "[NeuronNetwork]")
{
    return;
    // Initialize layers sizes
    int sizeInput  = 4;
    int hidden1    = 2;
    int hidden2    = 6;
    int sizeOutput = 3;

    // Initialize the Neural Network
    std::vector<int> layers = {sizeInput, hidden1, hidden2, sizeOutput};
    std::vector<float> outputMask = {0.0f, 1.0f, 2.0f};

    // BENCHMARK("Initializing Neural Network"){
    //     NeuronNetwork nN(layers, outputMask);
    // };

    NeuronNetwork nn(layers);

    // Get the neurons
    std::vector<NeuronLayer> neuronLayers = nn.getLayers();
    // nn.__str__();

    // Check the input weigths
    for(int i = 0; i < neuronLayers.size(); i++) {
        NeuronLayer nL = neuronLayers.at(i);
        std::vector<Neuron> neurons = nL.getNeurons();

        for (Neuron& n : neurons) {
            std::vector<float> weights = n.getWeights();

            // Get the weight from 1 neuron since all should be same for a initialized layer
            float weight = weights[0];
            float bias = n.getBias();

            // Using require that and withinRel too check floating point numbers.
            if (i == 0) {
                SECTION("Input Neurons") {
                    // Input layer must always have 1 weight that is 1
                    REQUIRE_THAT(weight, WithinRel(INITIAL_WEIGHT_INPUTN));
                    REQUIRE_THAT(bias, WithinRel(INITIAL_BIAS_INPUTN));
                    REQUIRE(weights.size() == 1);
                }
            } else {
                SECTION("Hidden and output Neurons") {
                    // Hidden layers and output layer must have 0.1 as weights and inputSize == amount of neurons in the last layer
                    REQUIRE_THAT(weight, WithinRel(INITIAL_WEIGHT));
                    REQUIRE_THAT(bias, WithinRel(INITIAL_BIAS));
                    REQUIRE(weights.size() == layers[i - 1]);
                }
            }
        }
    }
}

TEST_CASE("AND neural Network", "[NeuronNetwork][AND]")
{
    NeuronNetwork nn({2, 1});
    int inputSize = 2;
    int targetSize = 1;

    std::vector<std::vector<float» inputs = {
        {0.0f, 0.0f},
        {0.0f, 1.0f},
        {1.0f, 0.0f},
        {1.0f, 1.0f}
    };
    std::vector<std::vector<float» targets = {
        {0.0f},
        {0.0f},
        {0.0f},
        {1.0f}
    };

    MEASURE_BLOCK("Training the network AND", {
        nn.trainInputs2D(inputs, targets, 10000);
    });

    // Check if the network can predict the correct output
    for (int i = 0; i < targets.size(); i++)
    {

        std::vector<float> prediction = nn.predict(inputs[i]);

        printf("For input ");
```

```
        printVector(inputs[i], " Prediction ");
        printVector(prediction, "\n");

        if (i == 3) { // Checks for 1 1
            CHECK_THAT(prediction[0], WithinAbs(1.0f, 0.05f));
        } else { // Checks for 0 0, 0 1, 1 0
            CHECK_THAT(prediction[0], WithinAbs(0.0f, 0.05f));
        }
    }
}

TEST_CASE("XOR Neural Network", "[NeuronNetwork][XOR]")
{
    NeuronNetwork nn({2, 2, 1});
    int inputSize = 2;
    int targetSize = 1;

    std::vector<std::vector<float» inputs = {
        {0.0f, 0.0f},
        {0.0f, 1.0f},
        {1.0f, 0.0f},
        {1.0f, 1.0f}
    };
    std::vector<std::vector<float» targets = {
        {0.0f},
        {1.0f},
        {1.0f},
        {0.0f}
    };

    MEASURE_BLOCK("Training the network XOR", {
        nn.trainInputs2D(inputs, targets, 10000);
    });

    for (int i = 0; i < targets.size(); i++)
    {
        std::vector<float> prediction = nn.predict(inputs[i]);

        printf("For input ");
        printVector(inputs[i], " Prediction ");
        printVector(prediction, "\n");

        // Check if the network can predict the correct output
        if (i == 1 || i == 2) { // Checks for 0 1, 1 0
            CHECK_THAT(prediction[0], WithinAbs(1.0f, 0.05f));
        } else { // Checks for 0 0, 1 1
            CHECK_THAT(prediction[0], WithinAbs(0.0f, 0.05f));
        }
    }
}

TEST_CASE("Half adder Neuron Network", "[NeuronNetwork][HalfAdder]")
{
    NeuronNetwork nn({2, 3, 2});
    int inputSize = 2;
    int targetSize = 2;

    std::vector<std::vector<float» inputs = {
        {0.0f, 0.0f},
        {1.0f, 0.0f},
        {0.0f, 1.0f},
        {1.0f, 1.0f}
    };
    std::vector<std::vector<float» targets = {
        {0.0f, 0.0f},
        {1.0f, 0.0f},
        {1.0f, 0.0f},
        {0.0f, 1.0f}
    };

    MEASURE_BLOCK("Training the network Half Adder", {
        nn.trainInputs2D(inputs, targets, 10000);
    });

    for (int i = 0; i < targets.size(); i++)
    {
        std::vector<float> prediction = nn.predict(inputs[i]);

        printf("For input ");
        printVector(inputs[i], " Prediction ");
        printVector(prediction, "\n");

        // Check if the network can predict the correct output
        if (i == 1 || i == 2) { // Checks for 0 1, 1 0
            CHECK_THAT(prediction[0], WithinAbs(1.0f, 0.05f));
        } else { // Checks for 0 0, 1 1
            CHECK_THAT(prediction[0], WithinAbs(0.0f, 0.05f));
```

```
        }
    }
}

TEST_CASE("NeuronNetwork Learning Iris dataset", "[backpropagation]") {
    // Read the iris data set
    std::vector<std::vector<std::string» data = read_csv("../../data/iris.csv");

    // Extract the features and targets
    std::vector<std::vector<float» features = getFeatures(data);
    std::vector<float>              targets  = getTargets(data);

    irisData iris = filterData(features, targets, 0);

    // We expect 150 examples where each image consists of 4 integer values.
    int inputSize = 4;
    int outputSize = 3;
    // len of features and targets should be the same

    // Define a network architecture: an input layer of 4 neurons,
    // one hidden layer of 16 neurons, and an output layer of 3 neurons.
    std::vector<int> layers = { inputSize, 4, outputSize };
    std::vector<float> outputMask = {0.0f, 1.0f, 2.0f};

    // Mask the targets
    std::vector<float> maskedData = maskData(iris.targets, outputMask);
    std::vector<std::vector<float» targetMaskedData = create2DVector(maskedData, outputSize);

    // Create training and test split

    TrainTestSplit tts(iris.features, targetMaskedData, 0.95);

    NeuronNetwork nn(layers);

    MEASURE_BLOCK("Training the network", {
        nn.trainInputs2D(tts.trainFeatures, tts.trainTargets, 5000);
    });

    s_Allocations = 0;
    nn.trainInputs2D(tts.trainFeatures, tts.trainTargets,  5000);
    printf("Iris data Allocations: %d\n", s_Allocations);

    // Number of features to check
    int nToCheck = 20;

    // Randomly select nToCheck images and check if the network can classify them
    std::random_device rd;
    std::mt19937 gen(rd());
    int min = 0, max = tts.testTargets.size() -1;
    std::uniform_int_distribution<> dist(min, max);

    // Print the output mask for the network
    printf("\nOutputMask \t\t     ");
    printVector(outputMask, "\n");

    // Loop over each input
    for (int i = 0; i < nToCheck; i++) {
        // Get a random index
        int randomIndex = dist(gen);

        // Get the input and target
        std::vector<float> target = tts.testTargets[randomIndex];
        float targetValue         = targets[randomIndex];

        // Predict the output
        std::vector<float> prediction = nn.feedForward(tts.testFeatures[randomIndex]);

        // Print the predictions
        printf("Prediction for target %.2f | ", targetValue);
        printVector(prediction, "\n");

        // Find the target in the target mask
        auto it = std::find(target.begin(), target.end(), 1.0f);
        int targetIndex = std::distance(target.begin(), it);

        // NOTE: The prediction is a probability, it caa nbe false.
        // CHECK_THAT(1.0f, WithinRel(prediction[targetIndex], 0.1f));
    }
}

TEST_CASE("NeuronNetwork Learning digit data", "[backpropagation]") {
    // Load the digit dataset
    digitData digits = readDigitData<float>();

    // We expect 1797 examples where each image consists of 64 integer values.
    int numData = digits.targets.size();
    int inputSize = 64;
```

```
    int outputSize = 10;

    // Define a network architecture: an input layer of 64 neurons,
    // one hidden layer of 16 neurons, and an output layer of 10 neurons.
    std::vector<int> layers = { inputSize, 16, outputSize };
    std::vector<float> outputMask = {0.0f, 1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f, 9.0f};

    // Mask the targets
    std::vector<float> maskedData = maskData(digits.targets, outputMask);

    // Create 2D vectors for the target and input data
    std::vector<std::vector<float» targetMaskedData = create2DVector(maskedData, outputSize);
    std::vector<std::vector<float» imagesMaskedData = create2DVector(digits.images, inputSize);

    // Train and test split
    TrainTestSplit tts(imagesMaskedData, targetMaskedData, 0.95);

    NeuronNetwork nn(layers);

    std::vector<float> targets(digits.targets.begin(), digits.targets.end());
    std::vector<float> images(digits.images.begin(), digits.images.end());

    MEASURE_BLOCK("Training the network", {
        nn.trainInputs2D(tts.trainFeatures, tts.trainTargets, 5000);
    });

    s_Allocations = 0;
    nn.trainInputs2D(tts.trainFeatures, tts.trainTargets,  5000);
    printf("Digit data Allocations: %d\n", s_Allocations);

    // Number of features to check
    int nToCheck = 20;

    // Randomly select nToCheck images and check if the network can classify them
    std::random_device rd;
    std::mt19937 gen(rd());
    int min = 0, max = tts.testTargets.size() -1;
    std::uniform_int_distribution<> dist(min, max);

    // Print the output mask for the network
    printf("\nOutputMask \t\t     ");
    printVector(outputMask, "\n");

    // Loop over each input
    for (int i = 0; i < nToCheck; i++) {
        // Get a random index
        int randomIndex = dist(gen);

        // Get the input and target
        std::vector<float> input = tts.trainFeatures[randomIndex];
        std::vector<float> target = tts.testTargets[randomIndex];
        float targetValue = digits.targets[randomIndex];

        // Predict the output
        std::vector<float> prediction = nn.feedForward(input);

        // Print the predictions
        printf("Prediction for target %.2f | ", targetValue);
        printVector(prediction, "\n");

        // Find the target in the target mask
        int targetIndex = static_cast<int>(targetValue);

        // NOTE: The prediction is a probability, it can be false.
        CHECK_THAT(1.0f, WithinRel(prediction[targetIndex], 0.1f));
    }
}
```