Assignment 1

COS 110

Neural Network Simulation

Department of Computer Science

Deadline: 11 September 2021 at 23:00

Objectives:

- Become familiar with C++ Classes, Objects and dynamic memory.

General instructions:

- This assignment should be completed individually, **no group effort** is allowed.
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not include any C++ libraries that are not stated in this specification. Doing so will result in a mark of zero.
- If your code does not compile you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- **All submissions will be checked for plagiarism.**
- Read the entire assignment before you start coding.
- You will be afforded three upload opportunities.

Plagiarism:

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to `http://www.library.up.ac.za/plagiarism/index.htm` (from the main page of the University of Pretoria site, follow the *Library* quick link, and then choose the *Plagiarism* option under the *Services* menu). If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding. Also note that the OOP principle of code re-use does not mean that you should copy and adapt code to suit your solution.

Overview

In the field of Artificial Intelligence (AI), Neural Networks (NNs) are powerful tools facilitating the so-called "Fourth Industrial Revolution". This assignment attempts to examine your understanding of C++ classes and dynamic memory for a problem you may not have been previously been exposed to. **No fundamental knowledge of AI is required to complete this assignment**. This specification fully describes what you need to implement. UML class diagrams are used to describe the member variables and member functions. Please ensure that you are familiar with the notation and the meaning of the access modifiers specified in these diagrams.

What is a Neural Network?

A NN, for the purpose of this assignment, is made up of layers of neurons. Connecting these neurons together forms a model of the human brain. Neurons are represented using circles in a graph-like structure. In our scenario there is one input layer, one output layer, and one or more hidden layers which are in-between the input layer and the output layer.

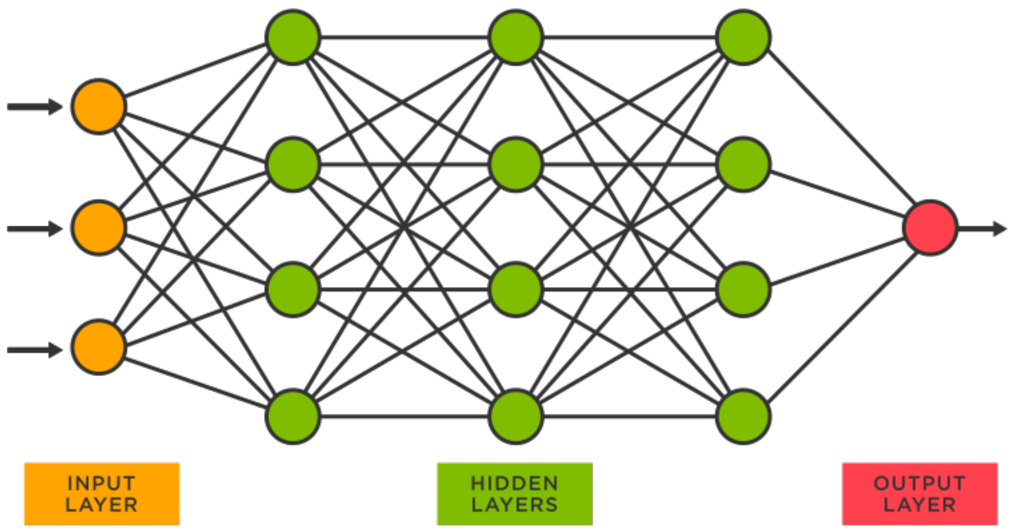

Figure 1: A visual graph demonstration of a fully connected neural network. . Note that hidden layers do not necessarily need to contain the same number of hidden neurons as in the figure. Image source: click here.

Values move from the input layer to the output layer i.e. from **left to right** through the network (this is the forward direction). Each neuron in a layer is connected to every neuron in the next layer via what is known as a *weight*. In order for a value to move forward in a network, it must be multiplied by the weight which attaches it to the next neuron in the network. After a value is aggregated in a neuron a non-linear activation function is applied (this will be explained in more detail later in this specification). The output of a neural network is the value(s) that emerge from the output layer, after the input values have made their way through the entire network. In this assignment, there will only be a single neuron in the output layer, and thus there will only be a single output value from the NN. Arrays are used to store the weights as well as neurons in a layer. These arrays are indexed from zero with zero being the top-most neuron or weight.

Why are Neural Networks so useful?

Neural networks are known as *universal approximators*. Although a NN is essentially a bunch of numbers being multiplied together, the weights of these networks are able to be adjusted in order to model different functions both linear and non-linear. In practice large networks allow computers to be able to perform useful tasks, some of which are classifying images, recommending movies and interpreting speech. This assignment is significantly simplified compared to real-life implementations. You will be implementing a forward pass through a fully connected neural network (for those who may know what biases are, there are no biases in this assignment).

InputLayer

The input layer to the network simply contains an array of `double` input values which represent the values stored in the yellow neurons in Figure 1.

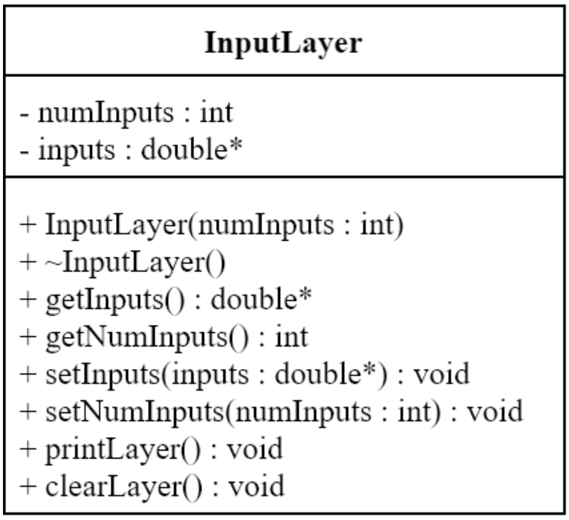

Figure 2: Input layer UML class diagram.

Member variables

- The `numInputs` member variable represents the number of inputs to the first layer of the NN. This is the size of the `double` array that is passed to the NN in the `setInputs` function.
- The `inputs` member variable is a dynamic one-dimensional `double` array. The values in this array are the inputs to the NN. In order to propagate values forward through the network, the `inputs` array must be set through its mutator function.

Member functions

- The `InputLayer` constructor accepts the `numInputs` as a parameter and sets the corresponding member variable. The `inputs` array can be set to `NULL` at this stage.
- The ∼`InputLayer()` destructor should deallocate the `inputs` array if it has been set.
- `getInputs()` returns the inputs array.
- `getNumInputs` returns the number of inputs of this NN.
- `setInputs(double* inputs)` sets the `inputs` member variable to the array argument (without a deep copy). If inputs are already set, memory for the inputs must be deallocated first before setting the new inputs.
- `setNumInputs(int numInputs)` sets the `numInputs` member variable to the value of the argument.

- `printLayer()` should print the string `i:x` where `x` is the `numInputs` of this layer. There should be no spaces in the string. An `endl` should be printed after the string.
- `clearLayer()` sets each element of `inputs` array to zero. If the `inputs` array is `NULL`, then do nothing.

HiddenNeuron

The `HiddenNeuron` class represents a single neuron stored in a hidden layer. In this assignment, hidden neurons are the only neurons that can get *activated*. This is why there is not dedicated `InputNeuron` or `OutputNeuron` in this assignment. A neuron is activated when an activation function is applied to it which is typically non-linear.

HiddenNeuron
- value : double - numWeights : int - weights : double*
+ HiddenNeuron(numWeights : int, weights : double*) + ~HiddenNeuron() + getValue() : double + setValue(value : double) : void + forward(prevLayer : HiddenLayer*) : void + forward(inputLayer : InputLayer*) : void + activateReLU() : void + activateSigmoid() : void

Figure 3: Hidden neuron UML class diagram.

Member variables

- The `value` member variable represents the current value that this neuron holds.
- The `numWeights` member variable holds the size of the `weights` array. This value corresponds to the number of neurons in the previous layer before the layer containing this hidden neuron.
- The `weights` member variable is a one-dimensional dynamic `double` array containing the values of the weights connecting to this neuron from the previous layer.

Member functions

- The `HiddenNeuron(int numWeights, double* weights)` constructor accepts the `numWeights` and `weights` array and sets the corresponding member variables (without deep copying the array). The `value` of this neuron should initially be set to zero.
- The ∼`HiddenNeuron()` destructor should deallocate the `weights` array.
- `getValue()` should return the current value of this neuron.
- `setValue(double value)` should set the `value` of this neuron to the value contained in the argument.

- `forward(HiddenLayer* prevLayer)` is responsible for calculating the new `value` of this neuron. This function multiplies the value of each neuron in the previous hidden layer by the corresponding weight in the `weights` array of this neuron. Each product is then summed up to form the new value for this neuron. Figure 4 shows an example of how the sum of products is used to determine a neurons value.
- `forward(InputLayer* input)` is an overload of the `forward` function which performs a forward operation given an `InputLayer` instead of a `HiddenLayer`. This function is only necessary when the previous layer is the `InputLayer`. This implies that the hidden neuron is in the **first** `HiddenLayer` of the NN. The calculation is the same as `forward(HiddenLayer* prevLayer)` except that it uses the `inputs` array in the provided `InputLayer` instead of a neuron's values in a `HiddenLayer`.

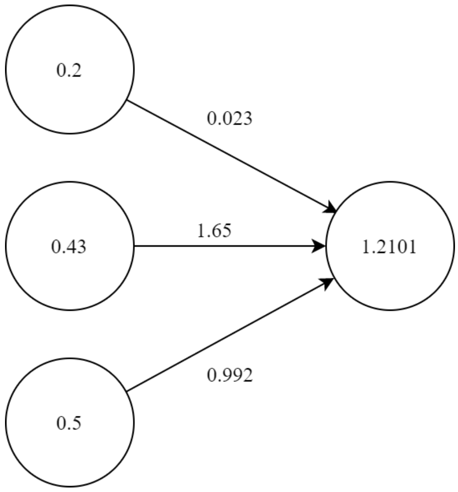

Figure 4: An example of obtaining a neuron's value given the weights connecting to it as well as the values of the previous neurons. The value `1.2101` is the weighted sum of the values of the neurons in the previous layer.

- `activateReLU()` applies the `ReLU` function to the `value` stored in this neuron and sets the `value` to the result. `ReLU` is defined as $f(value) = max(0, value)$.
- `activateSigmoid()` applies the `Sigmoid` function to the `value` stored in this neuron and sets the `value` to the result. You may use `exp()` from the `math.h` library to calculate this function.

$$sigmoid(value) = \frac{1}{1 + e^{-value}}$$

HiddenLayer

A `HiddenLayer` contains an array of `HiddenNeurons`. This class orchestrates the movement of values from the previous layer into this layer by calling the `forward` function for each of its neurons. A special case occurs for **first** hidden layer in the network because it receives input from an `InputLayer` instead of another `HiddenLayer`. After values have moved forward into this layer, an activation function is invoked for each `HiddenNeuron` based on the `activation` member variable.

HiddenLayer
- numNeurons : int - neurons : HiddenNeuron** - activation : string
+ HiddenLayer(numNeurons : int, neurons : HiddenNeuron**, activation : string) + ~HiddenLayer() + getNeurons() : HiddenNeuron** + setNeurons(neurons :HiddenNeuron*) : void + getNumNeurons() : int + setNumNeurons(numNeurons : int) : void + forward(prevLayer : HiddenLayer*) : void + forward(inputLayer : InputLayer*) : void + printLayer() : void + clearLayer() : void

Figure 5: Hidden layer UML class diagram.

Member variables

- The `numNeurons` member variable represents the number of neurons that this layer holds.
- The `neurons` member variable is a one-dimensional array of dynamically allocated `HiddenNeuron` objects. The member variable `numNeurons` refers to the size of this array.
- The `activation` member variable is a string which determines which activation function this `HiddenLayer` will use. The string "relu" implies that the `ReLU` function will be used. The string "sigmoid" implies that the `Sigmoid` function will be used. If the string does not exactly match these two possibilities, then no activation function is called.

Member functions

- The `HiddenLayer(int numNeurons, HiddenNeuron** neurons, string activation)` constructor receives arguments which are to be assigned to their corresponding member variables. No deep copy should be performed for the `neurons` array.
- The ∼`HiddenLayer()` destructor should deallocate the `neurons` array.
- `getNeurons()` returns the `neurons` array.
- `setNeurons(HiddenNeuron* neurons)` sets the `neurons` member variable to the array argument (without a deep copy). If neurons are already set, memory for the neurons must be deallocated first before setting the new neurons.
- `getNumNeurons()` returns the `numNeurons` member variable.
- `setNumNeurons(int numNeurons)` sets the `numNeurons` member variable
- `forward(HiddenLayer* prevLayer)` performs a `forward` operation for each neuron in this `HiddenLayer` using the `forward` function for each individual `HiddenNeuron`. After each neuron has performed a `forward` operation, this function activates each neuron based on the value of the `activation` member variable (unless the member variable does not match any of the specified strings).
- `forward(InputLayer* inputLayer)` is similar to the `forward(HiddenLayer* prevLayer)` function with the only difference being that the forward operation makes use of the `InputLayer` which

assumes that this hidden layer is the first hidden layer in the network. Activations apply in the same way as the `forward(HiddenLayer* prevLayer)` function.

- `printLayer()` should print the string `h:x:a` where `x` is the `numNeurons` of this layer and `a` is the `activation` of this layer. There should be no spaces in the string. An `endl` should be printed after the string.
- `clearLayer()` sets the value of each `HiddenNeuron` in the `neurons` array to zero. If the `neurons` array is `NULL`, then do nothing.

OutputLayer

The `OutputLayer` essentially represents the single output neuron. Therefore, there is one set of `weights` which attaches this layer to the last hidden layer. The single `outputValue` of this layer represents the output of the entire NN. For simplicity, the `OutputLayer` does not apply any activation functions.

OutputLayer
- outputValue : double - numWeights : int - weights : double*
+ OutputLayer(numWeights: int, weights: double*) + getOutputValue() : double + setOutputValue(outputValue :double) : void + forward(lastHiddenLayer : HiddenLayer*) : void + printLayer() : void + clearLayer() : void

Figure 6: Output layer UML class diagram.

Member variables

- The `outputValue` member variable represents the output value of the network.
- The `numWeights` member variable is the size of the `weights` array for this `OutputLayer`. Note that there is only a single output neuron for the NNs in this assignment, thus there will only be one array of weights for this `OutputLayer`. In order to prevent an unecessary `OutputNeuron` class, the `OutputLayer` class performs all the necessary functions for the single neuron.
- The `weights` member variable is a one-dimensional `double` array which contains the weight values between the last hidden layer and the single output neuron.

Member functions

- The `OutputLayer(int numWeights, double* weights)` constructor receives arguments which are to be assigned to their corresponding member variables. No deep copy should be performed for the `weights` array. The initial value of `outputValue` should be set to zero.
- The ∼`OutputLayer()` destructor should deallocate the `weights` array.
- `getOutputValue()` returns the `outputValue`.
- `setOutputValue(double outputValue)` sets the `outputValue` member variable.

- `forward(HiddenLayer* lastHiddenLayer)` performs a forward operation using the `weights` member variable and the last `HiddenLayer` which is passed as a parameter. The `outputValue` is set as the result of this operation.
- `printLayer()` should print the string `o:1` as there is always one output in this layer. There should be no spaces in the string. An `endl` should be printed after the string.
- `clearLayer()` sets the `outputValue` to zero.

NeuralNetwork

The `NeuralNetwork` class is an aggregation of the different layer types. The NN is guaranteed to have one `InputLayer`, one `OutputLayer` and **at least** one `HiddenLayer`. The `NeuralNetwork` class is responsible for feeding values from the `InputLayer` of the network up until the `OutputLayer` of the network and returning the single `int` result.

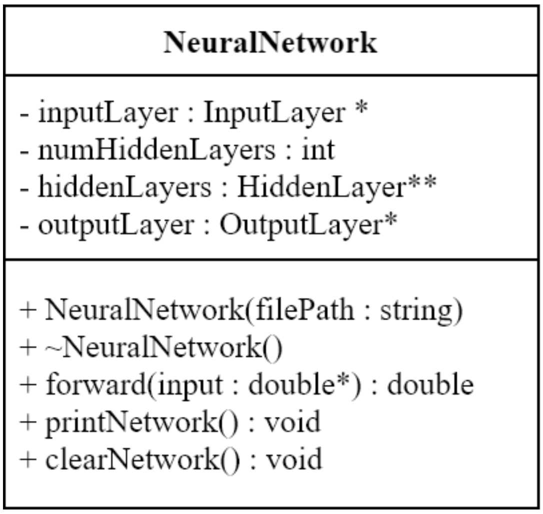

Figure 7: Neural network UML class diagram.

Member variables

- The `inputLayer` member variable is a pointer to a dynamically allocated `InputLayer` object. This is the input layer of the NN. A NN only has one input layer.
- The `numHiddenLayers` member variable contains the number of `HiddenLayer`'s contained in the `hiddenLayers` array.
- The `hiddenLayers` member variable is a dynamic one-dimensional array containing pointers to `HiddenLayer` objects.
- The `outputLayer` member variable is a pointer to a dynamically allocated `OutputLayer` object. This is the output layer of the NN. For this assignment, `OutputLayer` is synonymous with an output neuron since there will only be one output neuron for NNs in this assignment.

Member functions

- The `NeuralNetwork(string filePath)` constructor receives a path to a text file which contains a definition for how the NN should be constructed. The text file provides all the necessary information to create the NN. This constructor will use the text file to instantiate all of the member variables of this class, dynamically allocating memory as required. Figure 8 below contains an example of the text file structure you will receive. You may assume that the text file will always provide a valid configuration. Weights are space separated and are of arbitrary precision. You should also be able to read scientific notation doubles. Fortunately `fstream` caters for this with the conventional streaming into `doubles` using the `>>` operator. See the values in the example text files and make sure your program can read them.

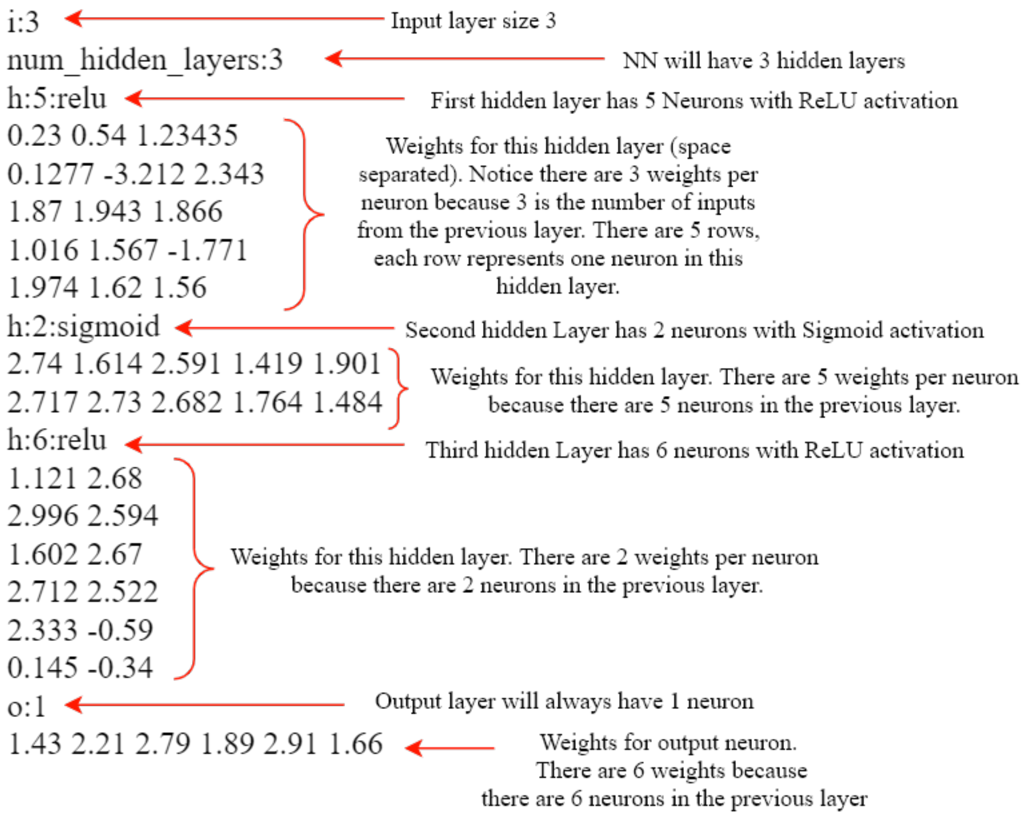

Figure 8: An example text file NN configuration. The `NeuralNetwork` constructor must create a NN based on files given in a similar structure. Note that the weights can be any `double` including negative numbers. There can also be an arbitrary number of hidden layers and hidden neurons per layer.

- The `~NeuralNetwork()` destructor should deallocate all dynamic memory created in this class.
- `forward(*double input)` feeds the `input` array of inputs forward through the entire network. Values should propagate from the input layer to the output layer. Finally, the `outputValue` in the `outputLayer` should be returned.
- `printNetwork()` should call the `printLayer` function of each layer in the NN in the forward order of the NN. Do not add any whitespace or `endl` in this function because the `printLayer` functions make use of their own `endl`.
- `clearNetwork()` calls the `clearLayer` function for each layer in this NN.

Libraries

The following libraries will be allowed:

- string
- fstream
- sstream

- iostream
- iomanip
- stdlib.h
- math.h
- `X.h` where `X` is a class name that is specified in one of the UML diagrams in this specification.

Implementation Considerations

- All variables, function names, class names and file names must be **exactly** the same as those specified in this specification. If the spelling or the case of these names is incorrect, a compilation error will occur and you will receive zero.
- The files names for each class are exactly the same as the class name. For example `NeuralNetwork.h` and `NeuralNetwork.cpp` describe the `NeuralNetwork` class.
- You **must** use the `-std=c++98` flag in your makefile in order to compile your code with C++98.
- Some of your files may be overwritten on FitchFork. You may assume that if your `.h` files get overwritten that they will include all of the relevant `Libraries` specified in the previous section for each particular class. You **may not assume** that the standard namespace is used in the `.h` files. You may `use` the standard namespace in your `.cpp` files or use the `std::` prefix.
- You must make a `main.cpp` and thoroughly test your code.
- The `HiddenLayer` and `HiddenNeuron` class both reference eachother. Therefore, you may need to forward declare the `HiddenLayer` class in your `HiddenNeuron.h` file.
- If you are detected for plagiarism you will receive zero for this assignment.

Provided example files

Some example networks are given to you which have weights adjusted to approximate different functions. The `Example NN structures` folder contains the structure of NNs which are used to model the x^2, $sin(x)$ and the $mean$ function. The `Example NN outputs` folder contains some input/output examples using these networks. These examples can help you ensure that your code is working correctly.

File check-list

You must submit the following files:

- `OutputLayer.h OutputLayer.cpp`
- `NeuralNetwork.h NeuralNetwork.cpp`
- `InputLayer.h InputLayer.cpp`
- `HiddenNeuron.h HiddenNeuron.cpp`
- `HiddenLayer.h HiddenLayer.cpp`
- `makefile`
- `main.cpp` (optional)

Submission

You need to submit your source files on the Fitch Fork website (https://ff.cs.up.ac.za/). Place all of your `.h` files and your `.cpp` files in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number. Also place your **makefile** in this archive. There is no need to include any other files in your submission. **Do not put any folders in the .zip archive**. You have 5 submissions and your best mark will be your final mark. Do not use Fitch Fork to test your code because you have limited marking opportunities. Upload your archive to the Assignment 1 slot on the Fitch Fork website for COS110. Submit your work before the deadline. No late submissions will be accepted!