Project Report

SEN402 GRADUATION PROJECT-II

DEEP LEARNING FRAMEWORK

Prepared by

Betül Nur YILDIRIM B2005.090047 Software Engineering
Mehmet ÖRS B2105.090156 Software Engineering

Supervisor

Dr. Öğr. Üyesi ROA'A ALI ABDULLAH MOHAMMEDQASEM

Table of Content

1. Project Title

DEEP LEARNING FRAMEWORK

2. Introduction

In the current era of artificial intelligence, the concept of deep learning is becoming popular in a wide range of applications, including image recognition like cancer detection from medical images, computer vision like detecting objects on frames, natural language processing like understanding text inputs and acting accordingly to produce an output, recommendation systems for recommending customer specified products in e-commerce websites etc. So, in time it became a field of interest for very diverse set of compaines-people that can utilize deep learning models to improve their business process, or for hobby purposes.

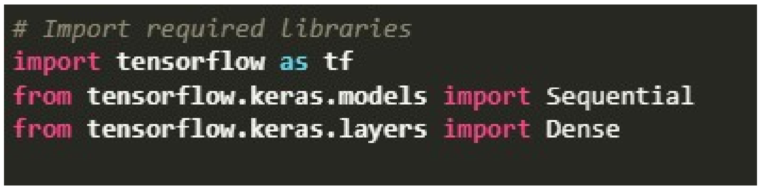
Our strategy for this problem is to develop our independet framework which includes direct solutions that focus on specified functionalities that do not include any interlayers to serve multiple platforms, rather works with low level C++ code that we can easily specify arbitrarily. We have designed our project in a modular, scalable, flexible, reusable way and implemented with raw and clean code from scratch, implementing each function and classes one by one, that abides with Object Oriented Programming structure and certain design principles.

3. Related Work

To develop a deep learning model, most popular way people apply is to use an open source framework-library where core functionalities are designed and developed by the developers of the community whom produces and maintains the framework. This common solution is very useful for user audience, where they can use these functionalities provided directly without needing to implement the low level logic each they need to develop a model. They can just specify the internal architecture of the pipeline of their model by utilizing the corresponding functionalities explained in the document.

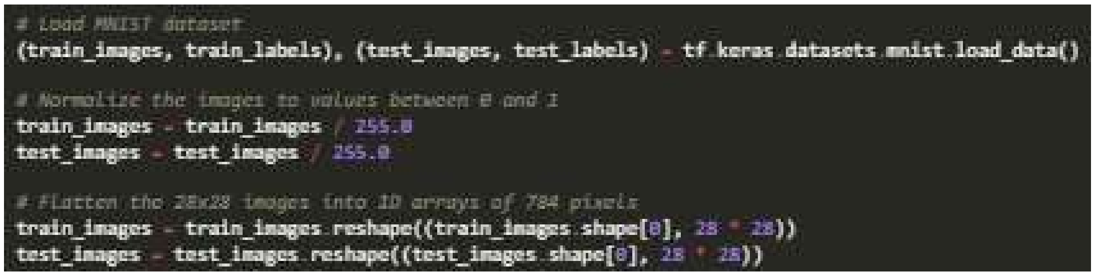
Today, in this sense, post popular frameworks used for deep learning models development are TensorFlow and PyTorch. Both are widely used world wide, and are similar in based an most factors like performance, training time, usablity etc. When the conditions like hardware, model details are same. So in the report we will generally use Tensorflow to compare our framework in detailed implementation. These comparisions will be handled and explained in the Methodology section.

Here is an example code in Python programming language where Tensorflow is used to create and train a ANN(Artificial Neural Network) deep learning model:



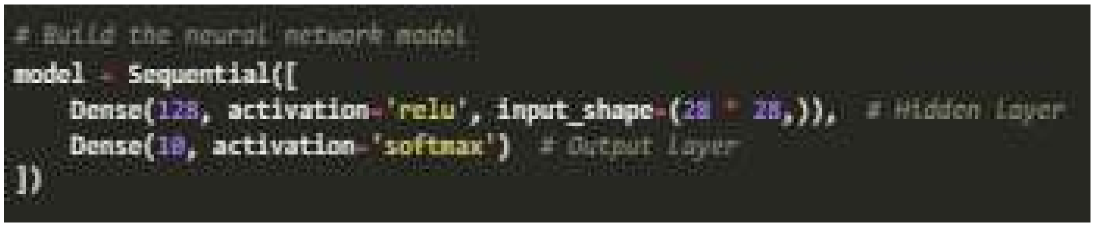
```python
# Import required libraries
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

First the library that contains the provided functionalities is imported.



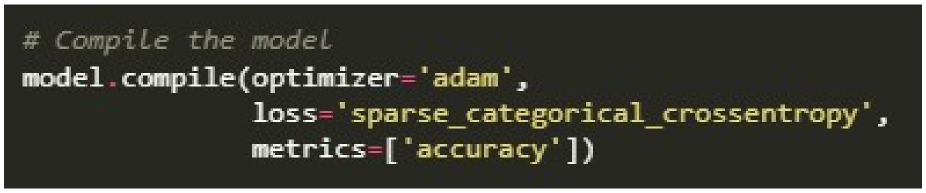
```python
# Load MNIST dataset
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.mnist.load_data()

# Normalize the images to values between 0 and 1
train_images = train_images / 255.0
test_images = test_images / 255.0

# Flatten the 28x28 images into 1D arrays of 784 pixels
train_images = train_images.reshape((train_images.shape[0], 28 * 28))
test_images = test_images.reshape((test_images.shape[0], 28 * 28))
```

In this step, the input data is prepared to train the model. First it is loaded, then it is scaled, and finally it is reshaped to be compatible for model.



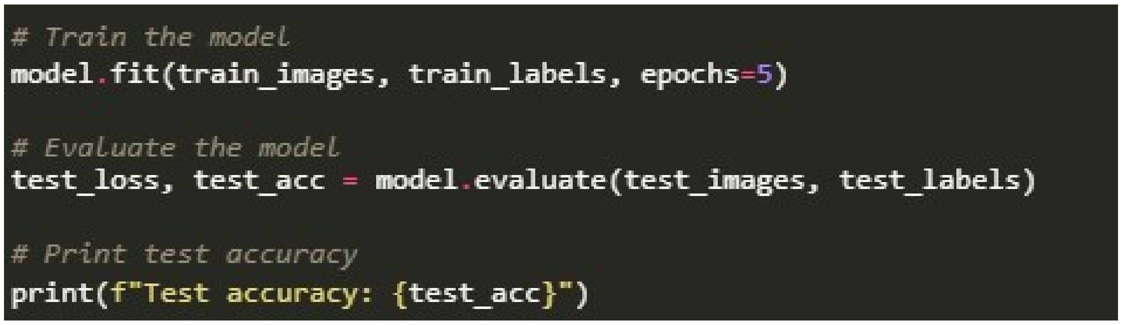
```python
# Build the neural network model
model = Sequential([
    Dense(128, activation='relu', input_shape=(28 * 28,)),  # Hidden Layer
    Dense(10, activation='softmax')  # Output Layer
])
```

Here is the main part where we specify the architecture of the model. In this example we have two dense layers with first layer having 128 neurons and 10 neurons in the second layer, with first layer using Relu as activation function, and second layer using softmax (which generally used in final layers for multi class classification tasks). Here means we have 10 possible classes as an output, and this model will predict which class does the input belongs after training is finished.



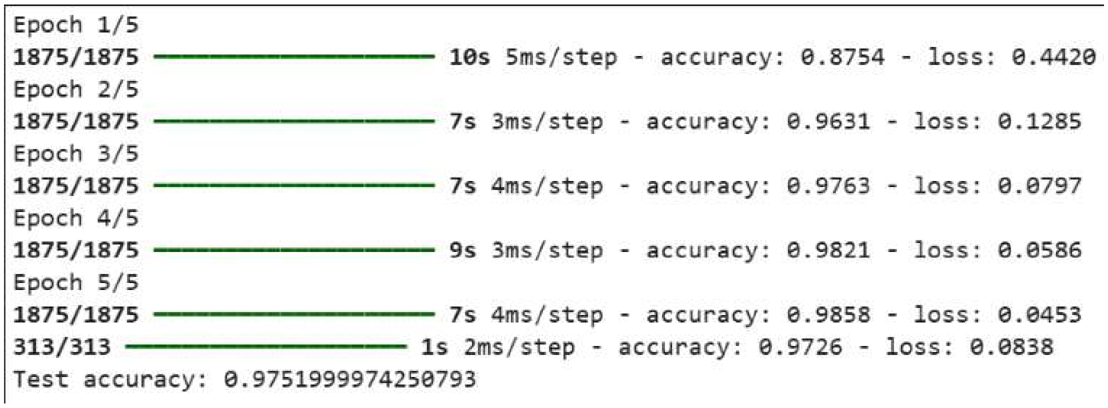
```
# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Now the model is compiled, where its optimizer is selected as Adam, loss function as sparse categorical cross entropy (which is used usually for neural networks with final layer having softmax activation function), and the metric used to evaluate the models performance is the accuracy score.



```
# Train the model
model.fit(train_images, train_labels, epochs=5)

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)

# Print test accuracy
print(f"Test accuracy: {test_acc}")
```

Finally, with the fit functon, we provide some hyperparameters as argument like the number of epochs (an epoch is the training process that feeds the whole data set once to the model), and also the training images and labels that we will use to train the model. This process is where training is handled. Then to test the model on the data that it did not see during training, we use the evaluate method and pass the test images and test labels as arguments and recieve the loss and accuracy scores.



```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.8754 - loss: 0.4420
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.9631 - loss: 0.1285
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9763 - loss: 0.0797
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.9821 - loss: 0.0586
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9858 - loss: 0.0453
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9726 - loss: 0.0838
Test accuracy: 0.9751999974250793
```

This is how the console looks during the training process. It allows users to visually track how the training proceeds in realtime.

Now even though these frameworks provide mentioned capabilities, they come with significant challenges which concern the users that want to achieve their goals efficiently. These challenges include complex usage structure that requires prior knowledge and experience in using the tools where users need to be familiar with the functionalities (where the documentation is needed to be checked for detailed comprehensibility) and have enough coding skills in , heavyweight dependencies that makes it hard to setup the environment and packages for developers each time even if they need to develop a simple model for simple tasks, limited low-level control over implementation details which restricts the ability to customize the structure of the library based on preferences or specialization for, and performance overhead which causes too long training times.

Our project differentiates from these tools by offering a lightweight deep learning framework developed in raw C++ where each functionality is implemented with its core logic one by one by us, which makes it faster in execution time. Simplicity, modularity, and performance is prioritized during development. Unlike existing solutions, we have full low level control over our frameworks model architecture since we are the one who builld it from scratch, and can add or remove any component to its modular structure later. Additionally, for the users that we will provide the functionalities, there will not be any dependencies for setting up an environment or requiring to write any code. We will provide a UI and full-stack deployment where users can directly interact and visually, resulting with an easier model development and deployment.

4. Problem Statement

Artificial Intelligence and its sub branches like Machine Learning and Deep Learning are now used in many different areas. In healthcare like helping doctors diagnose diseases or discovering new drugs for ills, in finance predicting stock market or fraud detections, or in automotive improving self driving cars technologies. Because of their huge impact on almost every industry, people and companies are eager to use these technologies. So, there is a growing need for these kind of AI tools that are easier to use and understand. There are already some popular tools. The most popular ones are TensorFlow and PyTorch. These frameworks are used for building deep learning models and they are very powerful.

However, they are mainly used by people who are related with that area. So usage of them are not easy for beginners or people without any technical background. There are 3 main problems for using these tools. First problem is for using these tools users need setup environments, manage their dependencies. Installing compatible versions of CUDA and TensorFlow/PyTorch can be confusing. If their versions does not match GPU support will not work or other tools not run. Second problem is complexity of usability. For able to use Pytorch/Tensorflow it is necessary to have knowledge of the literature and be able to read the documentation. Third and last problem about these tools is their low performance when we compare with our framework. Training models with them can be slow because they use extra layers in their software instead of directly running the code on the computer hardware. Every call from Python to C++ introduces latency and memory overhead. TensorFlow and Pytorch are primarily Python libraries even though they use C++ in background. So, these reasons can affect performance, especially when working with large amounts of data.

Most importantly, for now there is no system currently where users can build, train, test their models by just using an interface without dealing with the above problems. There is no the system people can work on AI without no code.

So the main problems and needs are here:

- **Problem 1:** Using TensorFlow or PyTorch requires complex environment setup and dependency management, often causing compatibility issues
- **Solution :** Our framework avoids complex setups by running directly in a C++ environment and it offers a user-friendly UI. So, it removes the need for any manual installations.
- **Problem 2:** These tools require technical knowledge to understand and use.
- **Solution:** We solve this by providing a no code user interface that abstracts all the technical complexity and lets users build and train models through on UI.
- **Problem 3:** Python based tools like TensorFlow and Pytorch can be slow because they have extra software layers and need to switch between Python and C++ when running code.
- **Solution:** Our framework is built fully in C++, so it runs directly on the computer without extra layers to make training faster.

Our project is designed to solve these exact problems. We aim to create a new deep learning framework and connect it to a user friendly application, where people can build and manage machine learning models without needing to be an expert.

5. Goal

The goal of this project is to design and develop a high performance, user friendly deep learning platform that enables users to build, train, and evaluate deep learning models without writing any code. Technical purpose is to implement the core logic of deep learning algorithms in c++ aligning with software development design patterns and object-oriented programming. Also, as the structure being designed modularly, project abides to separation of concerns and high cohesion rule and by designing the interface user friendly, the users could develop models without any code.

Core Components of The Platform:

1. **C++ based Deep Learning Framework (Computation/Training Engine):** It handles all the mathematical operations, training, and inference at in low level code.
2. **Backend (API Server) (Logic & Data Management Layer):** Manages communication between the frontend and the framework, handles user sessions, authentication, data persistence, and model control.
3. **Frontend (User Interface) (Presentation Layer):** The graphical interface where users interact with the platform without needing to write any code.

Key Features:

1. **Model Creation without Code:** Users can build custom neural networks visually, choosing layers, activations, and parameters using UI components.
2. **Model Training and Evaluation:** As the frontend sends the request to start training, the C++ framework component start training the model.
3. **Model Visualization:** Real time, interactive display of network architecture to help users build and understand models visually. These neural network elements are nodes, layers and connections.
4. **Project Save/Load Functionality:** Users can save, revisit, and modify their deep learning projects with ease. The related data to each user account and project is stored in the database component.

Technical Objectives:

1. Performance: The framework maximizes training and running speed by utilizing the efficiency of C++ language, since C++ communicates with hardware more efficiently comparing to other programming languages like Python.
2. Modularity: We are building the system with 4 separate parts. These parts are deep learning framework, backend server, database and frontend. Because of this separation, components will work perform their specific tasks and integrate to work synchronically in a distributed system manner. But this separation allows debugging, maintaining, fixing, updating or deleting without bothering others (seperation of concerns). When something is changed in the frontend, this change does not touch backend. This way, the project is easier to manage and grow in the future.
3. Usability: With the UI integration, the user will not need to care about the technical details. For example people who do not even know coding can easily create and train models easily.

6. Methodology

The methodology for our project involves developing core components one by one and applying appropriate testing to each module. The steps are explained in the subtitles.

6.1 Framework Design and Implementation

The main and initial idea of developing this project was to implement a deep learning framework from scratch using C++ library. So this is the core component of the system where each functionality that is used to during model creation, training and testing phase is implemented one by one in a modular way.

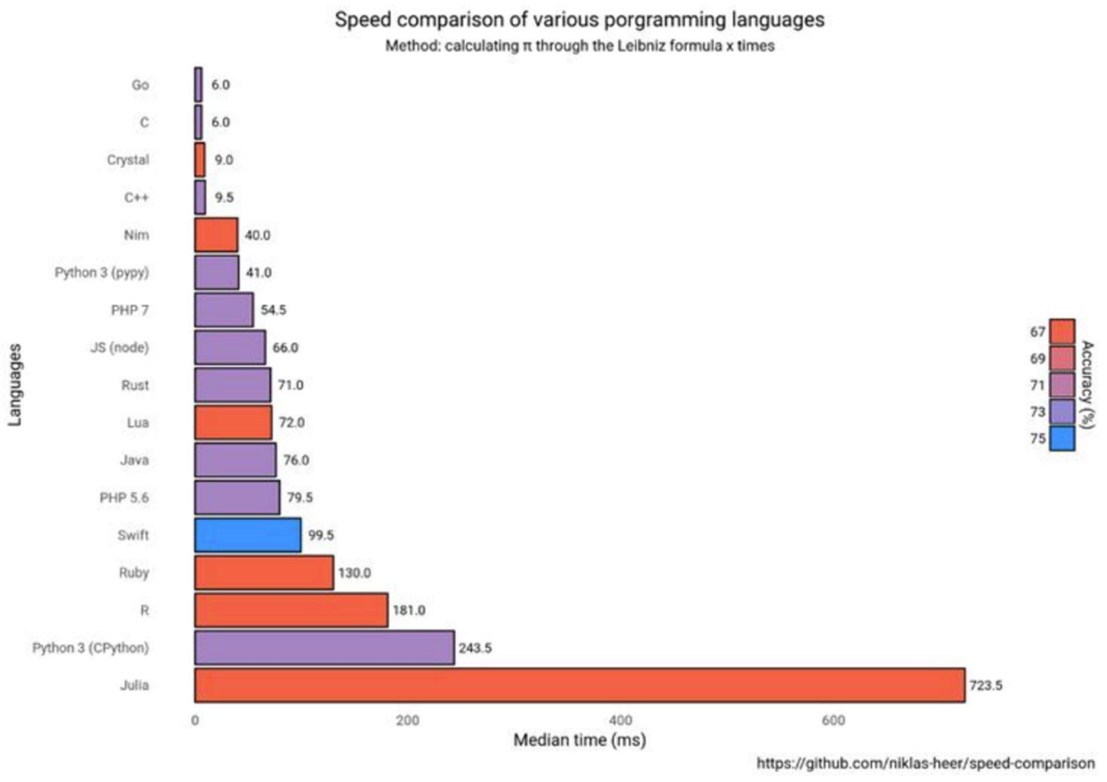
6.1.1 Reason for Choosing C++

The reasons to choose C++ as the programming language include:

- Fast execution and thus significantly better peroformance achievement in run time, when comparing it to other popular programming languages.
- It being low level language which makes it closer to machine language, which means

no extra layer overheads that abstracts the logic from developers, and thus meaning low memory usage and computational lost, where in other languages like Python, there are too many extra layers that is add to simplify development process. But since we are proficient enough to develop this framework in C++ programming language, we can get rid of such need.

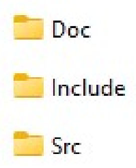
- Openness to optimization via techniques like multithreading or GPU acceleration in future stages.



As it is seen, C++ is faster in execution speed, which is the most important factor because of the long training times needed to train models, than most of the programming languages whith a significant difference. Like according to this data, C++ is more than 27 times faster than Python.

Another option to choose as the programming language could be C. It is even a more lightweight language that has less layers in compilation, and according to the graph it is 33% faster than C++ in average. However we still stick with C++ due to another crucial feature it provides. C++ supports Object Oriented Programming (OOP), which is essential for building modular, reusable, and maintainable clean code structure in modern applications. C language is an old language where at the time it was being developed, OOP logic was not yet. It is general-purpose imperative language, supporting structured programming. But in developing a framework especially in a large one like this, it is inevitable needing to use the utilities OOP, such as inheritance for inheriting the layer type functionalities from the layer interface for subclasses that represent different layer types (Dense Layer, Convolutional Layer etc.), or polymorphism which enables to override the functionalities in the sub classes for their specific distinct needs that super class defines. So this is why, to have a better project code structre, it was crucial to select C++ against C, where a large amount of files and the relationship between them is very important.

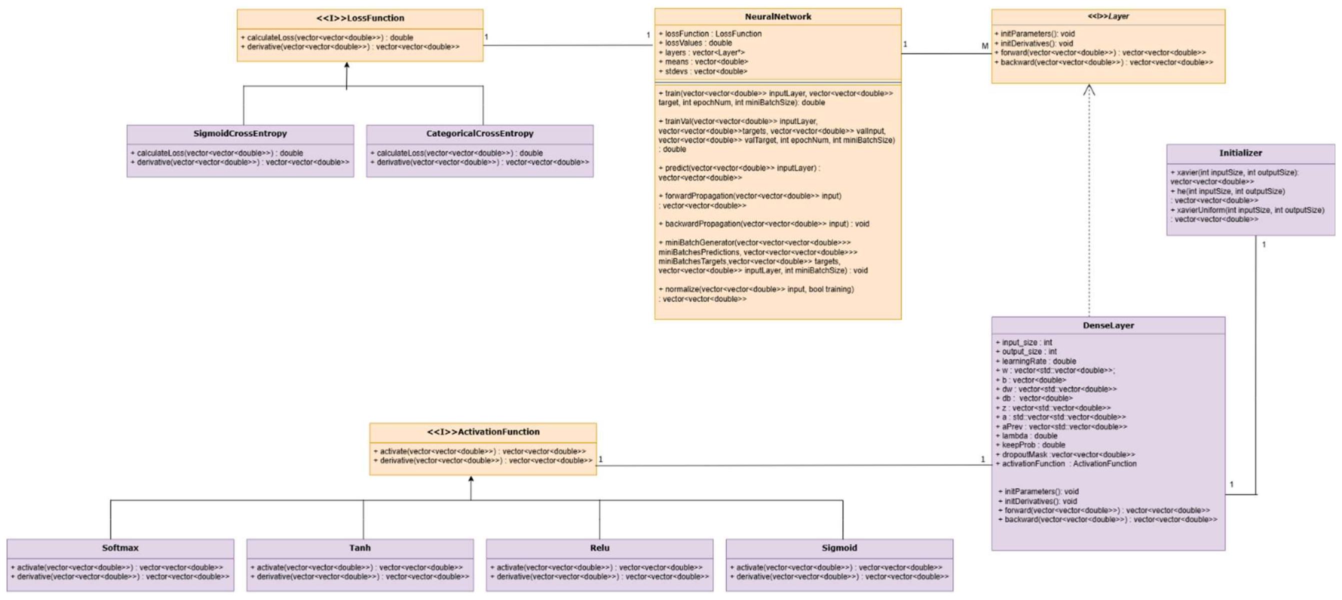
6.1.2 Project Folder and Class Structure Design



The structure of our work is divided to 3 main folders under the root of the project folder. Doc includes the necessary documentations about the project. Include and Src is the essential folders that contains the code written by us, where Include folder contains the definition of classes and their attributes-methods, which are the header files, and Src contains the implementation of each method of each class, which are the actual cpp files. It is common in C++ projects to divide header files and cpp files in this format.

6.1.3 Class Definitions and Implementation

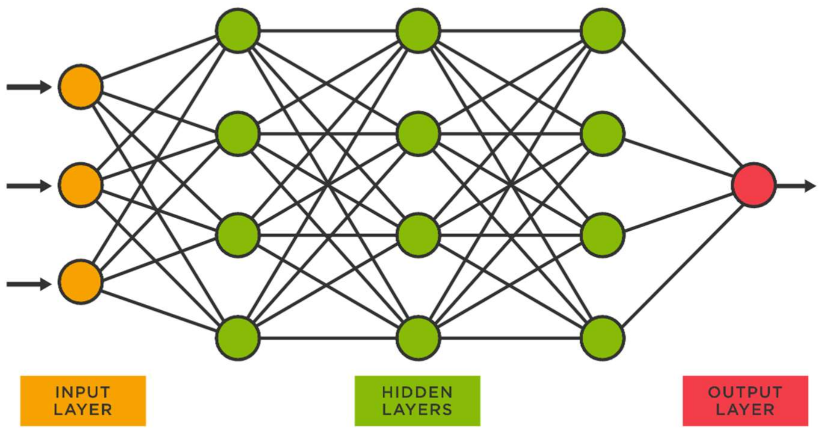
Before stepping to implementation of the classes, they were first designed by considering the project from top view by answering questions like *"What component classes would be needed to achieve the goal? How should the relationships between these classes be structured to ensure a modular and extensible architecture?"* Then the UML class diagram was generated as depicted in below image.


<<I>>LossFunction
+ calculateLoss(vector<vector<double>>) : double
+ derivative(vector<vector<double>>) : vector<vector<double>>
1
1
NeuralNetwork
+ lossFunction : LossFunction
+ lossValues : double
+ layers : vector<Layer*>
+ means : vector<double>
+ stdevs : vector<double>
+ train(vector<vector<double>> inputLayer, vector<vector<double>> target, int epochNum, int miniBatchSize): double
+ trainVal(vector<vector<double>> inputLayer, vector<vector<double>>targets, vector<vector<double>> valInput, vector<vector<double>> valTarget, int epochNum, int miniBatchSize) : double
+ predict(vector<vector<double>> inputLayer) : vector<vector<double>>
+ forwardPropagation(vector<vector<double>> input) : vector<vector<double>>
+ backwardPropagation(vector<vector<double>> input) : void
+ miniBatchGenerator(vector<vector<vector<double>>> miniBatchesPredictions, vector<vector<vector<double>>> miniBatchesTargets,vector<vector<double>> targets, vector<vector<double>> inputLayer, int miniBatchSize) : void
+ normalize(vector<vector<double>> input, bool training) : vector<vector<double>>
1
M
<<I>>Layer
+ initParameters(): void
+ initDerivatives(): void
+ forward(vector<vector<double>>) : vector<vector<double>>
+ backward(vector<vector<double>>) : vector<vector<double>>
SigmoidCrossEntropy
+ calculateLoss(vector<vector<double>>) : double
+ derivative(vector<vector<double>>) : vector<vector<double>>
CategoricalCrossEntropy
+ calculateLoss(vector<vector<double>>) : double
+ derivative(vector<vector<double>>) : vector<vector<double>>
Initializer
+ xavier(int inputSize, int outputSize): vector<vector<double>>
+ he(int inputSize, int outputSize) : vector<vector<double>>
+ xavierUniform(int inputSize, int outputSize) : vector<vector<double>>
1
DenseLayer
+ input_size : int
+ output_size : int
+ learningRate : double
+ w : vector<std::vector<double>>;
+ b : vector<double>
+ dw : vector<std::vector<double>>
+ db : vector<double>
+ z : vector<std::vector<double>>
+ a : std::vector<std::vector<double>>
+ aPrev : vector<std::vector<double>>
+ lambda : double
+ keepProb : double
+ dropoutMask :vector<vector<double>>
+ activationFunction : ActivationFunction
+ initParameters(): void
+ initDerivatives(): void
+ forward(vector<vector<double>>) : vector<vector<double>>
+ backward(vector<vector<double>>) : vector<vector<double>>
1
1
<<I>>ActivationFunction
+ activate(vector<vector<double>>) : vector<vector<double>>
+ derivative(vector<vector<double>>) : vector<vector<double>>
1
Softmax
+ activate(vector<vector<double>>) : vector<vector<double>>
+ derivative(vector<vector<double>>) : vector<vector<double>>
Tanh
+ activate(vector<vector<double>>) : vector<vector<double>>
+ derivative(vector<vector<double>>) : vector<vector<double>>
Relu
+ activate(vector<vector<double>>) : vector<vector<double>>
+ derivative(vector<vector<double>>) : vector<vector<double>>
Sigmoid
+ activate(vector<vector<double>>) : vector<vector<double>>
+ derivative(vector<vector<double>>) : vector<vector<double>>

To explain the class diagrams in detail:

- Neural Network:

This is the main class that represents the pipeline of the model. It has the attribute which is the list that contains the layer type objects.

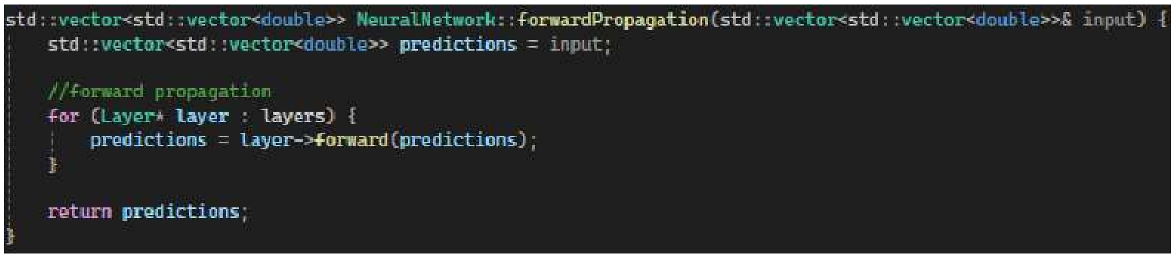


For example this image represents a Artificial Neural Network (ANN) that contains 4 hidden including the output layer which is also a hidden layer (not including input layer because that layer is not a layer type with parameters such as weights and biases, it represents the input data like images). So to represent this in our code, the attribute of this Neural Network class object will have 4 elements of type Layer (specifically DenseLayer in this case) class object.

Another important attribute here is the lossFunction attribute, that represents which type of loss function will be used for this Neural Network object. The way the loss function of a Neural Network class object is selected in runtime using Factory design pattern is explained in the subsequent sub title "Loss Factory".

The methods of this class provides essential functionalities that are used for training, testing and predicting the input samples. Forward and backward propagation is the most important method here for training the model.

Forward propagation method takes the input data set as parameter, the first layer in the layers list executes its forward method to produce its output and passes to the subsequent layer, and the second layer takes the first layers output as its input and does the same by executing its forward method and again produces an output, then the third layer takes the second layers output and again produces its output, and this process continues in a loop until the last layer is reached in the list. The output of the last layer is considered as the final output of the neural network.

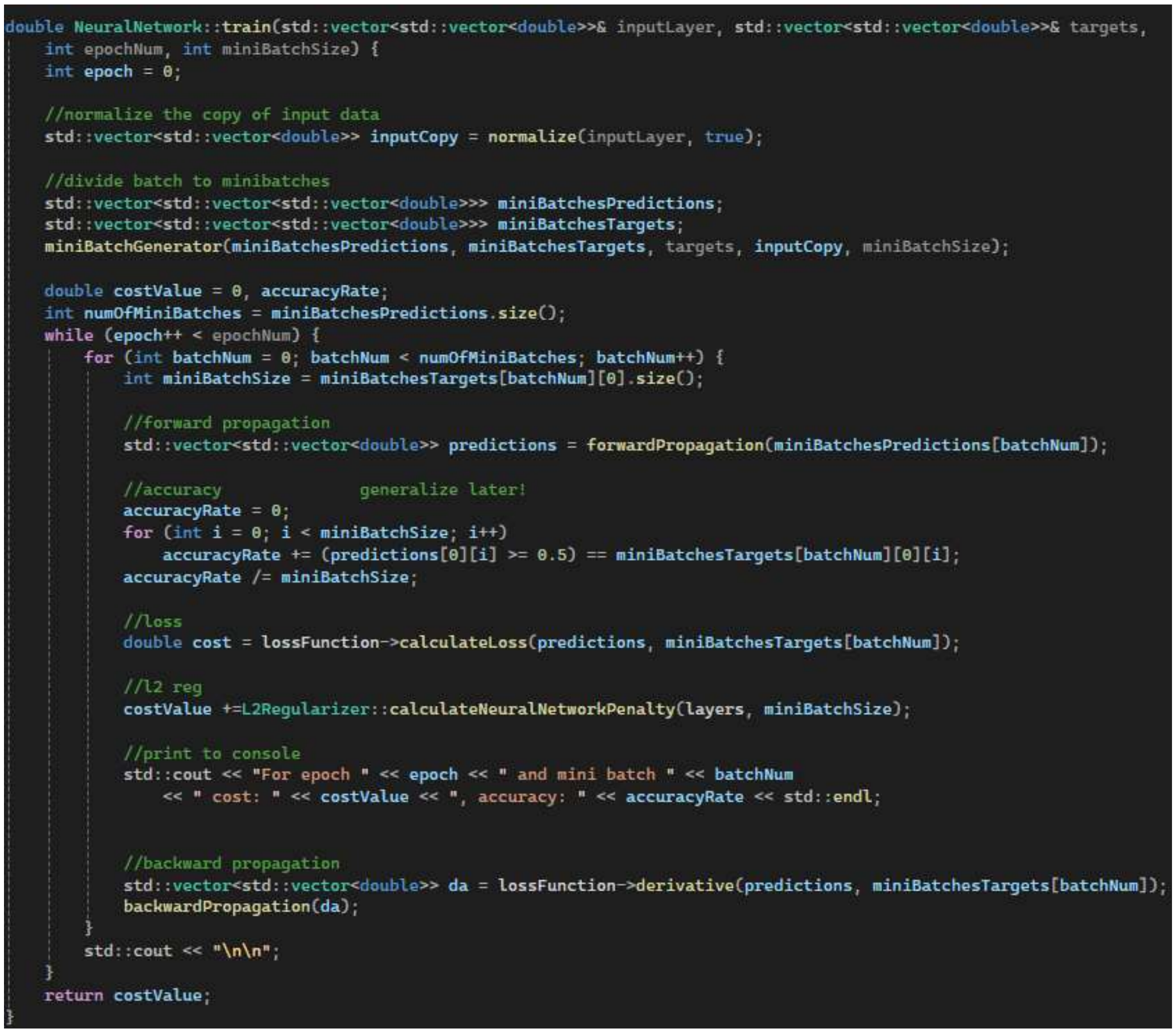


```
std::vector<std::vector<double>> NeuralNetwork::forwardPropagation(std::vector<std::vector<double>>& input) {
    std::vector<std::vector<double>> predictions = input;

    //forward propagation
    for (Layer* layer : layers) {
        predictions = layer->forward(predictions);
    }

    return predictions;
}
```

This is the implementation of the forward propagation function. The detail of the forward function implementation depends on the type of the layer (Dense Layer, Convolutional layer etc.) and is explained in Layers class explanation.

The output calculated after the forward propagation is passed to the loss function to determine how correct is the network predicting the output of given inputs by calculation the penalty.

Backward propagation is executed after the cost penalty is calculated to update the parameters (weights, biases) of the layers of the neural network to adjust them so that next time in the neural network can predict the output closer to the correct result. This time the loop starts from the last layer which is the nth layer by taking the loss vector as an input, updates its weights based on this input, and passes the output to the (n-1)th layer. Again this process contiues until the first layer updates its parameters. The implementation of each layers backward propagation (the way they update their weights) depends on the type of the layer



```cpp
double NeuralNetwork::train(std::vector<std::vector<double>>& inputLayer, std::vector<std::vector<double>>& targets,
    int epochNum, int miniBatchSize) {
    int epoch = 0;

    //normalize the copy of input data
    std::vector<std::vector<double>> inputCopy = normalize(inputLayer, true);

    //divide batch to minibatches
    std::vector<std::vector<std::vector<double>>> miniBatchesPredictions;
    std::vector<std::vector<std::vector<double>>> miniBatchesTargets;
    miniBatchGenerator(miniBatchesPredictions, miniBatchesTargets, targets, inputCopy, miniBatchSize);

    double costValue = 0, accuracyRate;
    int numOfMiniBatches = miniBatchesPredictions.size();
    while (epoch++ < epochNum) {
        for (int batchNum = 0; batchNum < numOfMiniBatches; batchNum++) {
            int miniBatchSize = miniBatchesTargets[batchNum][0].size();

            //forward propagation
            std::vector<std::vector<double>> predictions = forwardPropagation(miniBatchesPredictions[batchNum]);

            //accuracy             generalize later!
            accuracyRate = 0;
            for (int i = 0; i < miniBatchSize; i++)
                accuracyRate += (predictions[0][i] >= 0.5) == miniBatchesTargets[batchNum][0][i];
            accuracyRate /= miniBatchSize;

            //loss
            double cost = lossFunction->calculateLoss(predictions, miniBatchesTargets[batchNum]);

            //l2 reg
            costValue +=L2Regularizer::calculateNeuralNetworkPenalty(layers, miniBatchSize);

            //print to console
            std::cout << "For epoch " << epoch << " and mini batch " << batchNum
                << " cost: " << costValue << ", accuracy: " << accuracyRate << std::endl;


            //backward propagation
            std::vector<std::vector<double>> da = lossFunction->derivative(predictions, miniBatchesTargets[batchNum]);
            backwardPropagation(da);
        }
        std::cout << "\n\n";
    }
    return costValue;
}
```

and the optimizer type used (gradient descent, adam, rmsprop etc.). For now, only gradient descent style is implemented, but other important ones like “adam” will be implemented later.

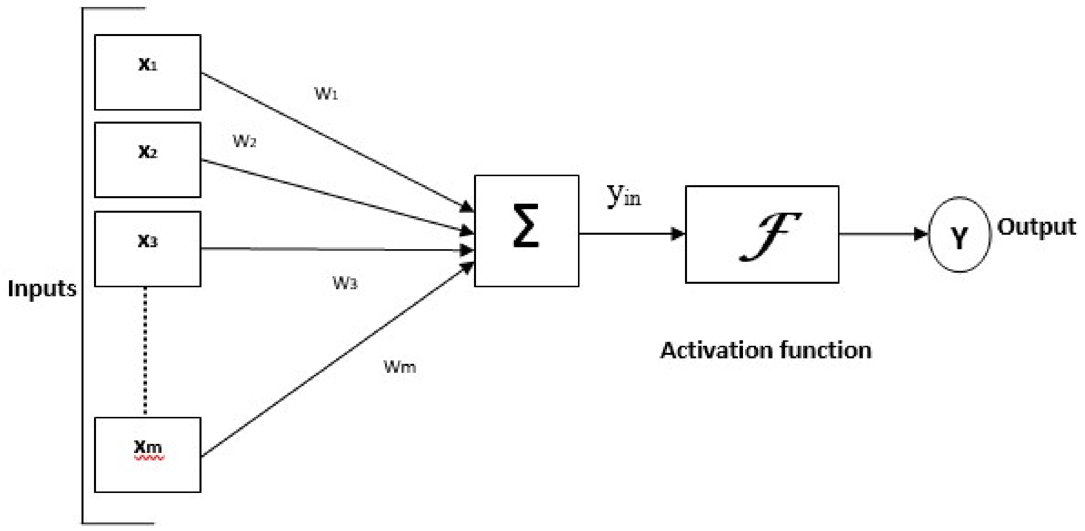
This is the implementation of train method, where it executes fowrardPropagation method to generate an output for the given input, calculates the cost value of the network, and executes backwardPropagation method to update the neural network parameters to reduce the cost. This process is done in the while loop for the number of epoch times, which is a hyperparameter of the network that specifies how many times will this training process done on the batch input, that is passed as a parameter to the train method. Before the training loop starts, first the input batch is divided to minibatches, where the size of each minibatch is also a hyperparameter that the user provides as an argument.

- Layer:

This is the abstract class (or interface) which is implemented as a header file in the project that defines the essential attributes and methods that each layer type (the subclasses like Dense Layer, Convolutional layer etc.) should have (inheritance) and override (polimorphism) based on its specialized implementation, like both Convolutional and Dense layers have to have forward and backward methods, but the way they implement them in the method differs.

For now, only Dense Layer class type is implemented since the project is at its initial phase and on progress. Later, layer types for different type of Neural Networks architectures, such as CNN's, RNN's or LSTM'S, will be implemented and added to the modular structure of the project.

To explain the implementation of DenseLayer class, the most important attributes are the weight (w) and bias (b) parameters, which are used to calculate the linear output of the layer (z) (forward), and are optimized (backward) during training to improve prediction accuracy. Activation function is another important attribute that takes the linear output z as an input and produces the non linear output to break the linearity and generate a better learnable system which is closer to reality. The type of activation function is determined in runtime by user input parameter (hyperparameter), just like the type of loss function was specified in NeuralNetwork class using the factory design pattern.



As shown in the image, this is the logic of how forward method of Dense Layer works.

So what happens here is, the vector X is taken as input, each of its feature (x1, x2, x3 etc.) is multiplied with the weights of the neuron where each weight (w1, w2, w3 etc.) of the neuron is multiplied with the corresponding feature, and the sum of the result of each multiplication is taken. The bias value (b) is added to the final sum, and the linear result Z is calculated. This is the

Step 1:

$$\begin{tikzcd} & y = \omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3 + bias \\ \boxed{\text{Step 1:}} \arrow[ur] \arrow[dr] & \\ & y = \sum_{i=1}^{n} \{(\omega_i * x_i) + b\} \end{tikzcd}$$

The y in the formula is z in our implementation (and several others).

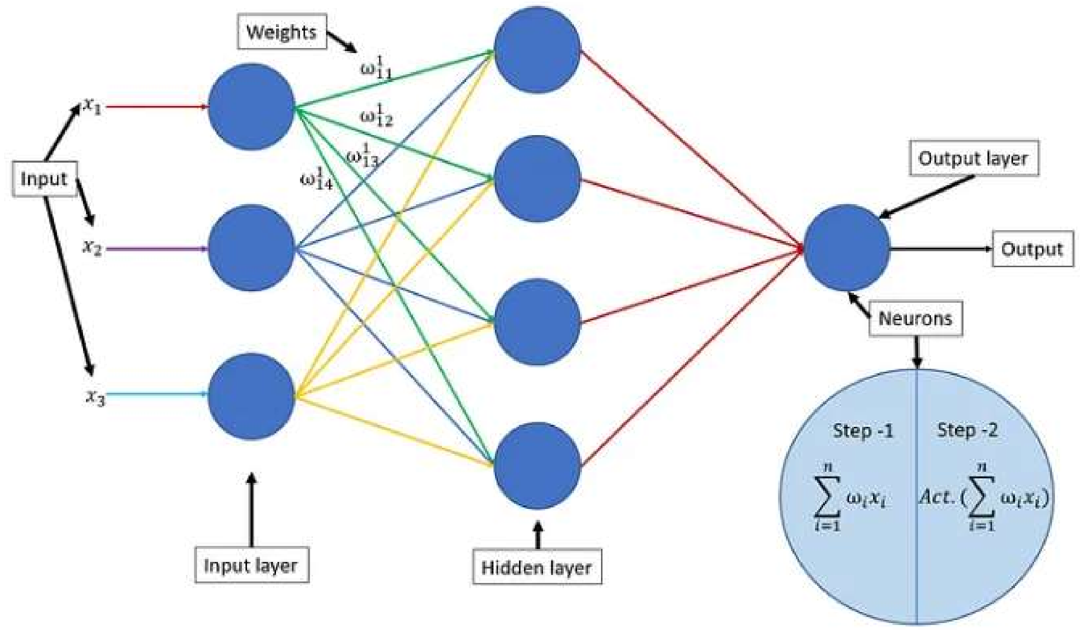
Step 2:

Step 2 is to pass the z value to the activation function as an input, and the output value (a) will be the final result of the forward calculation of this specific neuron.

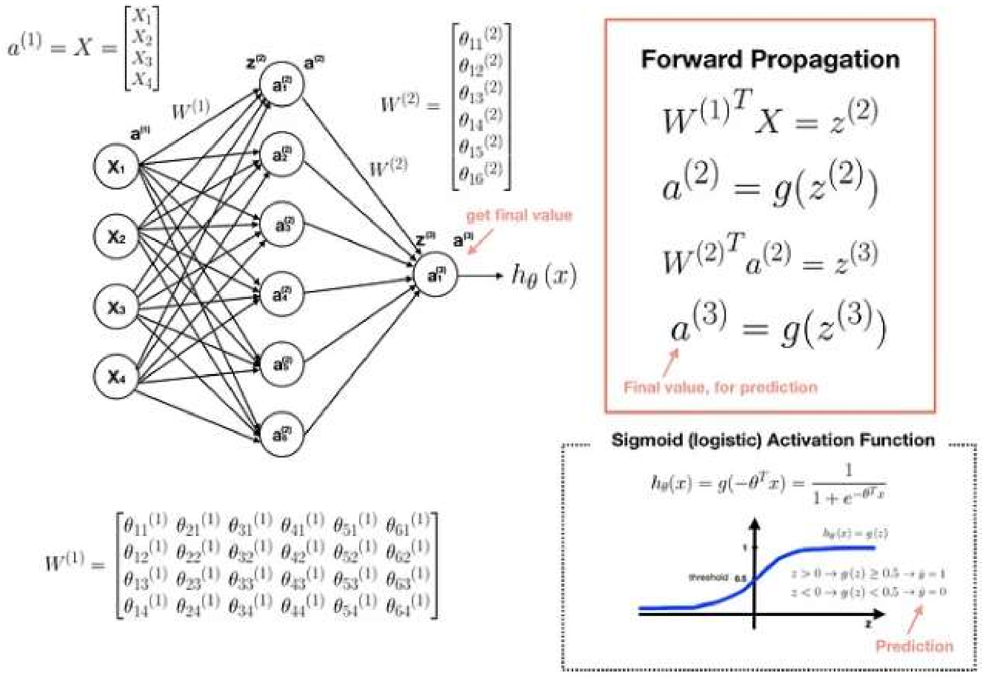
$$\boxed{\text{Step 2:}} \longrightarrow z = y_{pred} = \frac{1}{1 + e^{-(\omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3 + b)}}$$

Here z is a in our case, which is the output of the activation function. In this case, here the type of activation function used is sigmoid, where its formula is used in the image.

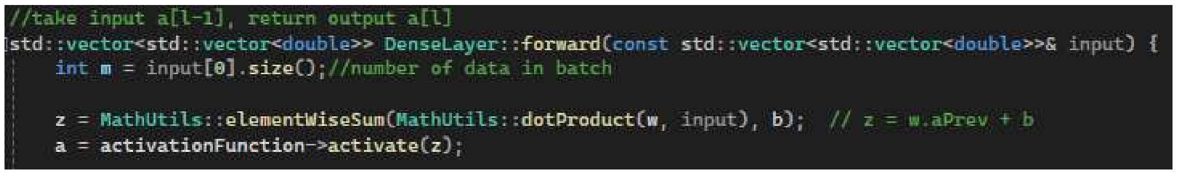
Now in our case, we will have several neurons (the number of neurons is a hyperparameter specified by the user and given as input when creating the layer) in a neuron like in the following image:



So to generalize this computation, we need to use linear algebra operations, like the dot product of the weights of the neurons of a layer and the input it takes, and the element wise sum operation of the result of dot product (W.X) and the bias vector (which represents the bias scaler value of each neuron in the layer).


$a^{(1)} = X = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix}$
$W^{(2)} = \begin{bmatrix} \theta_{11}^{(2)} \\ \theta_{12}^{(2)} \\ \theta_{13}^{(2)} \\ \theta_{14}^{(2)} \\ \theta_{15}^{(2)} \\ \theta_{16}^{(2)} \end{bmatrix}$
$W^{(1)}$
$W^{(2)}$
get final value
$h_\theta(x)$
Forward Propagation
$W^{(1)T} X = z^{(2)}$
$a^{(2)} = g(z^{(2)})$
$W^{(2)T} a^{(2)} = z^{(3)}$
$a^{(3)} = g(z^{(3)})$
Final value, for prediction
$W^{(1)} = \begin{bmatrix} \theta_{11}^{(1)} & \theta_{21}^{(1)} & \theta_{31}^{(1)} & \theta_{41}^{(1)} & \theta_{51}^{(1)} & \theta_{61}^{(1)} \\ \theta_{12}^{(1)} & \theta_{22}^{(1)} & \theta_{32}^{(1)} & \theta_{42}^{(1)} & \theta_{52}^{(1)} & \theta_{62}^{(1)} \\ \theta_{13}^{(1)} & \theta_{23}^{(1)} & \theta_{33}^{(1)} & \theta_{43}^{(1)} & \theta_{53}^{(1)} & \theta_{63}^{(1)} \\ \theta_{14}^{(1)} & \theta_{24}^{(1)} & \theta_{34}^{(1)} & \theta_{44}^{(1)} & \theta_{54}^{(1)} & \theta_{64}^{(1)} \end{bmatrix}$
Sigmoid (logistic) Activation Function
$h_\theta(x) = g(-\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$
threshold 0.5
$h_\theta(x) = g(z)$
$z > 0 \to g(z) \geq 0.5 \to \hat{y} = 1$
$z < 0 \to g(z) < 0.5 \to \hat{y} = 0$
Prediction

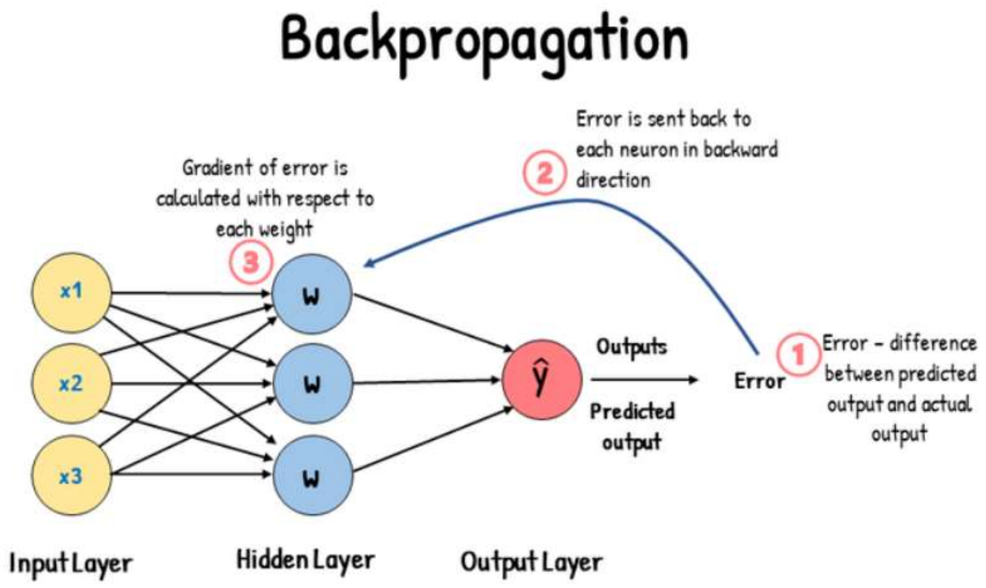
Here is the implementation in the code:



```
//take input a[l-1], return output a[l]
std::vector<std::vector<double>> DenseLayer::forward(const std::vector<std::vector<double>>& input) {
    int m = input[0].size();//number of data in batch

    z = MathUtils::elementWiseSum(MathUtils::dotProduct(w, input), b);  // z = w.aPrev + b
    a = activationFunction->activate(z);
```

The linear algebra function mentioned, here used as elementWiseSum and dotProduct, are implemented by us in the MathUtils class.

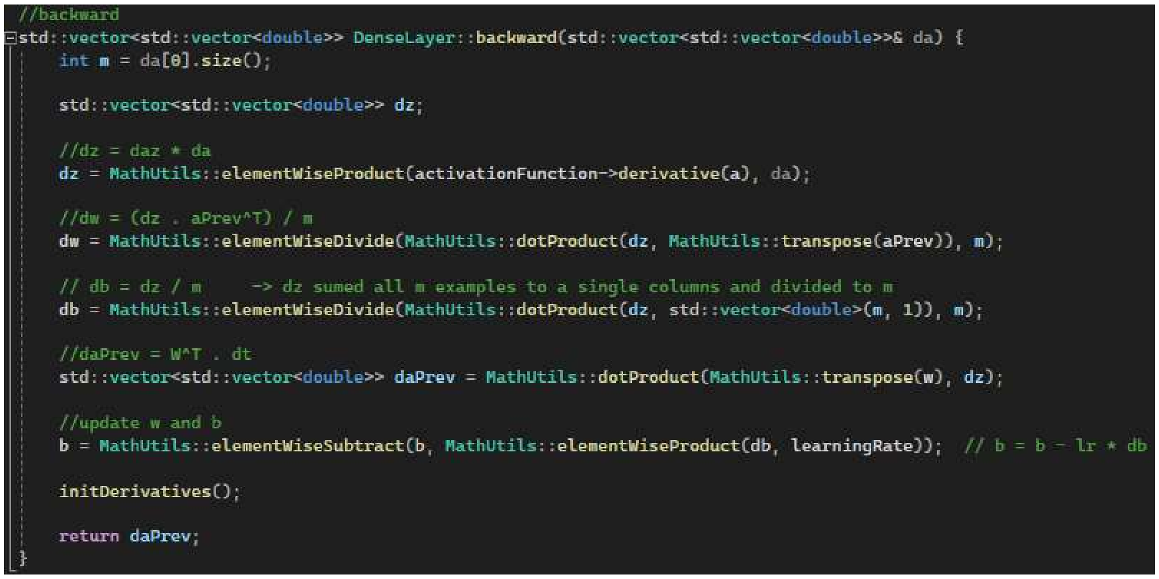
Now to explain back propagation method (backward) of dense layer, the following image provides a good overview.


Backpropagation
Error is sent back to each neuron in backward direction
2
Gradient of error is calculated with respect to each weight
3
x1
x2
x3
w
w
w
ŷ
Outputs
Predicted output
Error
1
Error - difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer

So the as the output of the final layer in the neural network is passed as parameter to the loss function, the cost value, which represents how far is the prediction value is from the actual label, is calculated as the output. Then to reduce this cost, the backward process, which is adjusting the parameters (weights and biases) of each neuron in the network, gradient descent optimizaiton method is used (others like adam, rmsprop will be added in the future). The formula of gradient descent is:

$$w = w - \alpha \frac{\partial}{\partial w} J(w,b)$$
$$b = \boxed{b} - \alpha \boxed{\frac{\partial}{\partial b} J(w,b)}$$

Here alpha is the learning rate (which is another hyperparameter defined by the user) which indicates how big will the optimization steps be. So, a single neuron will be optimized by taking the derivative of the cost function with respect to that neurons parameters, and substracting that value times learning rate from its current value. This will make the cost value lower next time in forward propagation. Again to calculate and apply this process to the whole network (all neurons), linear algebra operations are utilized (which, as indicated earlier, are implemented by as in the class Math Utils).



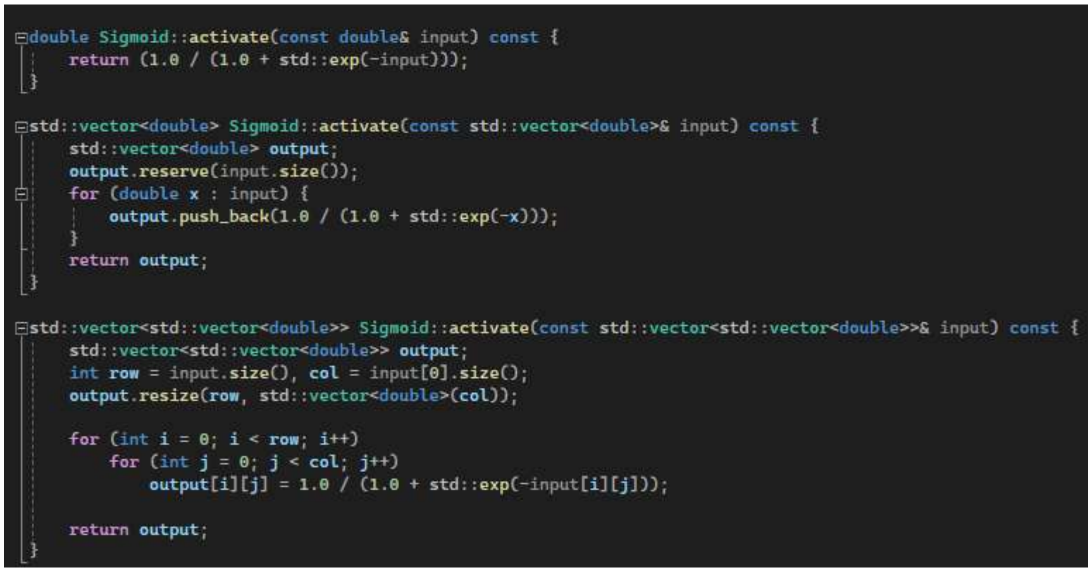
```
//backward
std::vector<std::vector<double>> DenseLayer::backward(std::vector<std::vector<double>>& da) {
    int m = da[0].size();

    std::vector<std::vector<double>> dz;

    //dz = daz * da
    dz = MathUtils::elementWiseProduct(activationFunction->derivative(a), da);

    //dw = (dz . aPrev^T) / m
    dw = MathUtils::elementWiseDivide(MathUtils::dotProduct(dz, MathUtils::transpose(aPrev)), m);

    // db = dz / m     -> dz sumed all m examples to a single columns and divided to m
    db = MathUtils::elementWiseDivide(MathUtils::dotProduct(dz, std::vector<double>(m, 1)), m);

    //daPrev = W^T . dt
    std::vector<std::vector<double>> daPrev = MathUtils::dotProduct(MathUtils::transpose(w), dz);

    //update w and b
    b = MathUtils::elementWiseSubtract(b, MathUtils::elementWiseProduct(db, learningRate));  // b = b - lr * db

    initDerivatives();

    return daPrev;
}
```

- Activation Functions:

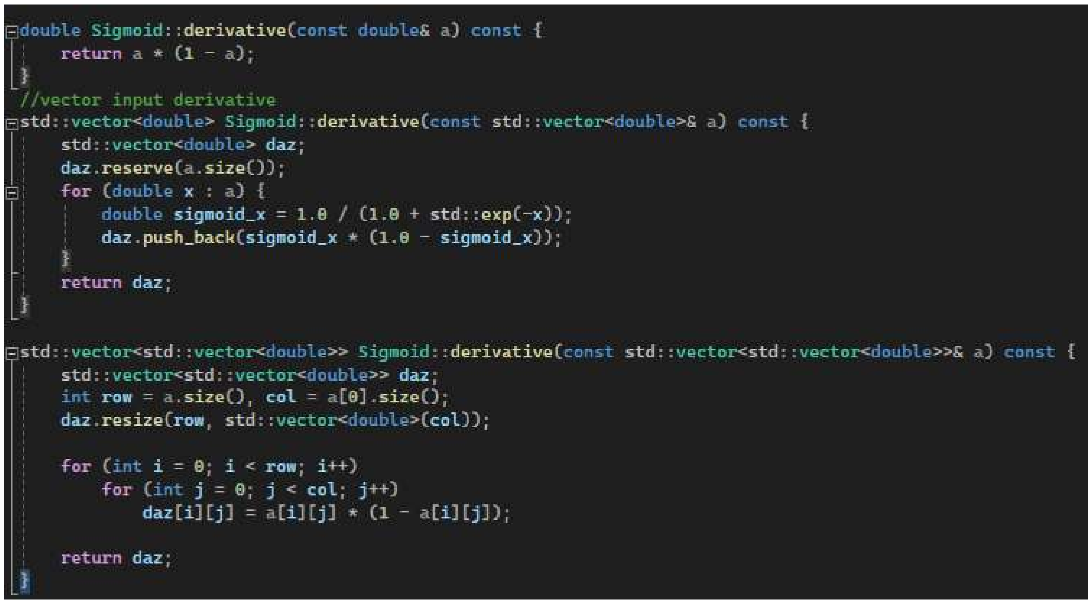
Activation functions, as mentioned earlier, takes the linear output of a neuron and produces a non linear output, which allows the network to learn and represent complex patterns that cannot be captured by linear transformations alone.

In our framework, types of activation functions (ReLu, sigmoid, tanh etc.) are implemented as separate classes under the abstract class ActivationFunction, as shown in the class diagram. All of them override the activate and derivative method. Activate method is used in forward propagation and derivative is used in backward propagation (which returns the partial derivative of the loss function output with respected to the activation function of that neuron-layer).

Here is one of the activation functions, which is sigmoid, implementation code:



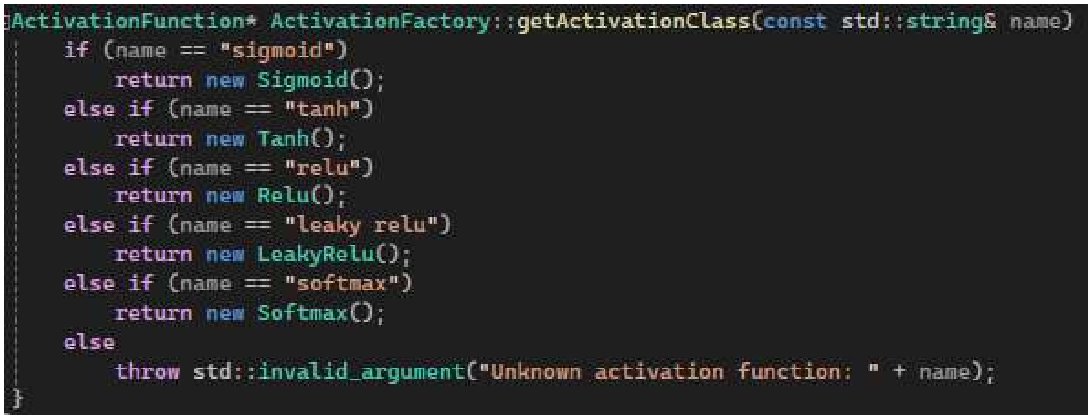
```cpp
double Sigmoid::activate(const double& input) const {
    return (1.0 / (1.0 + std::exp(-input)));
}

std::vector<double> Sigmoid::activate(const std::vector<double>& input) const {
    std::vector<double> output;
    output.reserve(input.size());
    for (double x : input) {
        output.push_back(1.0 / (1.0 + std::exp(-x)));
    }
    return output;
}

std::vector<std::vector<double>> Sigmoid::activate(const std::vector<std::vector<double>>& input) const {
    std::vector<std::vector<double>> output;
    int row = input.size(), col = input[0].size();
    output.resize(row, std::vector<double>(col));

    for (int i = 0; i < row; i++)
        for (int j = 0; j < col; j++)
            output[i][j] = 1.0 / (1.0 + std::exp(-input[i][j]));

    return output;
}
```



```cpp
double Sigmoid::derivative(const double& a) const {
    return a * (1 - a);
}
//vector input derivative
std::vector<double> Sigmoid::derivative(const std::vector<double>& a) const {
    std::vector<double> daz;
    daz.reserve(a.size());
    for (double x : a) {
        double sigmoid_x = 1.0 / (1.0 + std::exp(-x));
        daz.push_back(sigmoid_x * (1.0 - sigmoid_x));
    }
    return daz;
}

std::vector<std::vector<double>> Sigmoid::derivative(const std::vector<std::vector<double>>& a) const {
    std::vector<std::vector<double>> daz;
    int row = a.size(), col = a[0].size();
    daz.resize(row, std::vector<double>(col));

    for (int i = 0; i < row; i++)
        for (int j = 0; j < col; j++)
            daz[i][j] = a[i][j] * (1 - a[i][j]);

    return daz;
}
```

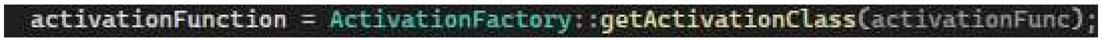
The activation and derivative methods are overloaded based on the type of input (as a single scalar, vector, or matrix). Usually in training process, the ones with matrix input is used since the inputs are given as batches or mini batches.

The type of activation function of a specific layer is selected in run time using Factory Pattern based on user input string.



```
ActivationFunction* ActivationFactory::getActivationClass(const std::string& name)
    if (name == "sigmoid")
        return new Sigmoid();
    else if (name == "tanh")
        return new Tanh();
    else if (name == "relu")
        return new Relu();
    else if (name == "leaky relu")
        return new LeakyRelu();
    else if (name == "softmax")
        return new Softmax();
    else
        throw std::invalid_argument("Unknown activation function: " + name);
}
```

Here is how the activationFunction attribute is initialized in the constructor of DenseLayer:

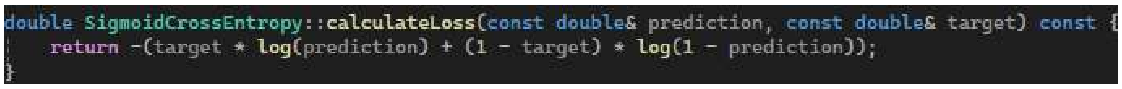


```
 activationFunction = ActivationFactory::getActivationClass(activationFunc);
```

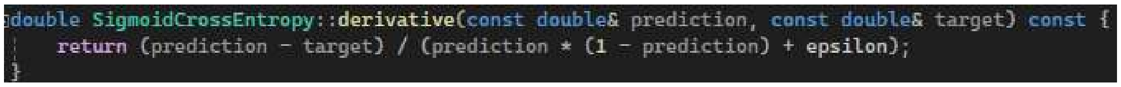
- Loss Functions:

Loss functions measure how close is the models predictions to the actual target values, where the smaller the output means the better the model can accurately predict. Later the loss values it generates is used in backward propagation in the training process, where the optimizer uses its partial derivatives to adjust the model’s weights to improve it.

The structure of its implementation is the same as Activation functions. Again factory pattern class is used to select the loss function type in neural network class based on the user input (hyperparameter).



```
double SigmoidCrossEntropy::calculateLoss(const double& prediction, const double& target) const {
    return -(target * log(prediction) + (1 - target) * log(1 - prediction));
}
```

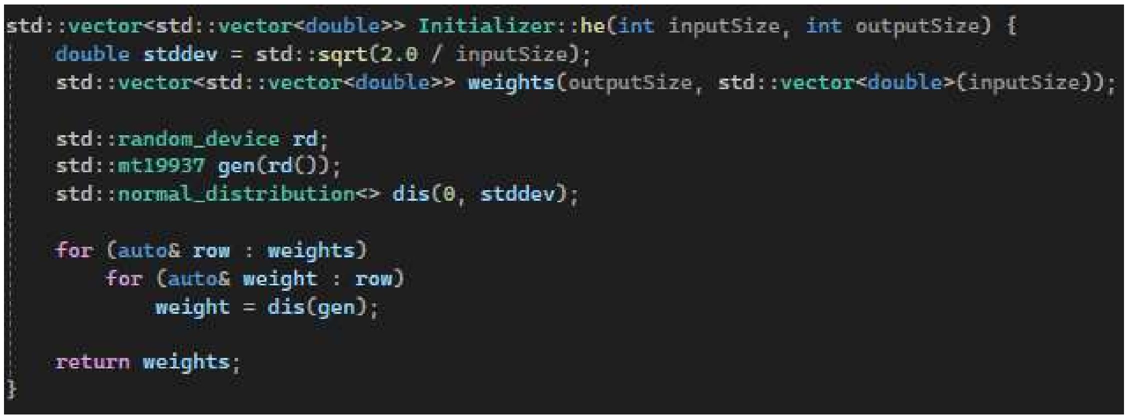


```
double SigmoidCrossEntropy::derivative(const double& prediction, const double& target) const {
    return (prediction - target) / (prediction * (1 - prediction) + epsilon);
}
```

These methods are also overloaded for different types of input dimension provided.

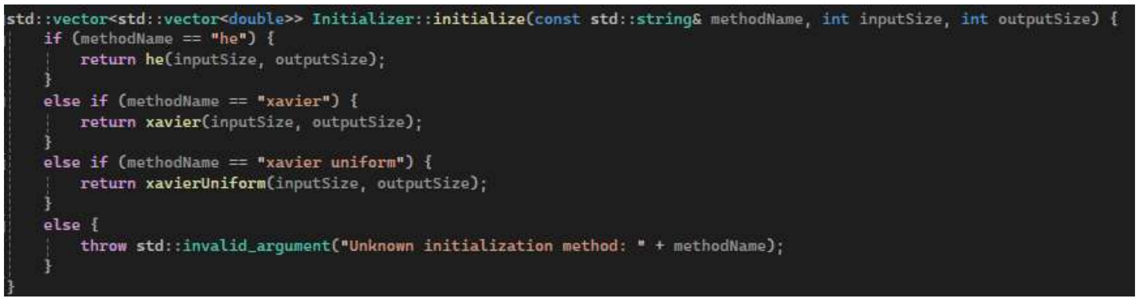
- Initializer:

This class is used to separate the weight initalization logic in a different module. Different types of weight initalization techniques (such as Xavier,He) are implemented.



```
std::vector<std::vector<double>> Initializer::he(int inputSize, int outputSize) {
    double stddev = std::sqrt(2.0 / inputSize);
    std::vector<std::vector<double>> weights(outputSize, std::vector<double>(inputSize));

    std::random_device rd;
    std::mt19937 gen(rd());
    std::normal_distribution<> dis(0, stddev);

    for (auto& row : weights)
        for (auto& weight : row)
            weight = dis(gen);

    return weights;
}
```

Another factory pattern implementation:

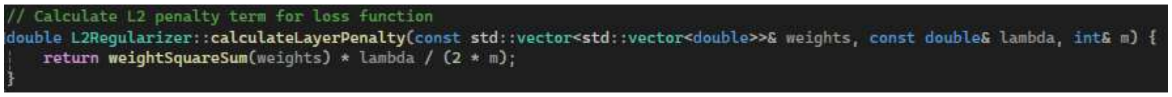


```
std::vector<std::vector<double>> Initializer::initialize(const std::string& methodName, int inputSize, int outputSize) {
    if (methodName == "he") {
        return he(inputSize, outputSize);
    }
    else if (methodName == "xavier") {
        return xavier(inputSize, outputSize);
    }
    else if (methodName == "xavier uniform") {
        return xavierUniform(inputSize, outputSize);
    }
    else {
        throw std::invalid_argument("Unknown initialization method: " + methodName);
    }
}
```

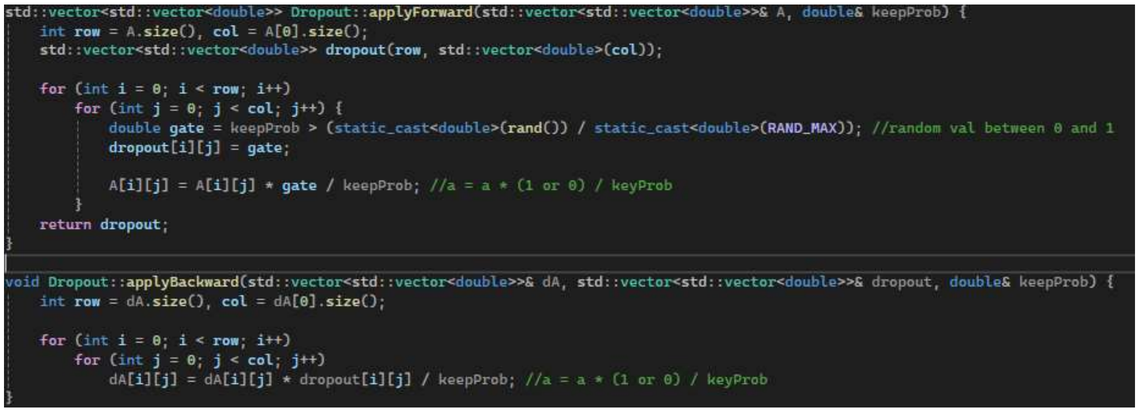
- Regularizers:

The final implementaion for now is about the regularizer functions L2 and Dropout. These techniques are used to prevent models from overfitting, which is memorizing the train set too much.

L2 regularizer parameter is added to the loss function to penalize large or complex model parameters. Dropout excludes certain neurons during training by selecting these neruons randomly based on the keepProb parameter. This way it reduces the layers learning complexity.



```
// Calculate L2 penalty term for loss function
double L2Regularizer::calculateLayerPenalty(const std::vector<std::vector<double>>& weights, const double& lambda, int& m) {
    return weightSquareSum(weights) * lambda / (2 * m);
}
```



```
std::vector<std::vector<double>> Dropout::applyForward(std::vector<std::vector<double>>& A, double& keepProb) {
    int row = A.size(), col = A[0].size();
    std::vector<std::vector<double>> dropout(row, std::vector<double>(col));

    for (int i = 0; i < row; i++)
        for (int j = 0; j < col; j++) {
            double gate = keepProb > (static_cast<double>(rand()) / static_cast<double>(RAND_MAX)); //random val between 0 and 1
            dropout[i][j] = gate;

            A[i][j] = A[i][j] * gate / keepProb; //a = a * (1 or 0) / keyProb
        }
    return dropout;
}

void Dropout::applyBackward(std::vector<std::vector<double>>& dA, std::vector<std::vector<double>>& dropout, double& keepProb) {
    int row = dA.size(), col = dA[0].size();

    for (int i = 0; i < row; i++)
        for (int j = 0; j < col; j++)
            dA[i][j] = dA[i][j] * dropout[i][j] / keepProb; //a = a * (1 or 0) / keyProb
}
```

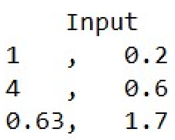
6.1.4 Testing

The testing part includes functional and non functional testing of the framework. Functional testing is about the unit test (black box testing) done on the component classes seperately, and non functional testing is the performance-speed comparision of the framework to Tensorflow in terms of execution (more spesifically training) time.

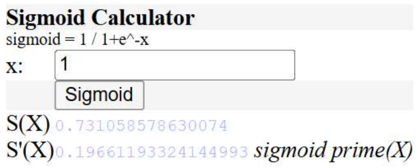
6.1.4.1 Unit Testing

Example of unit testing, for activation function class "Sigmoid" is as following:

The case is started by preparing the test case. A matrix of random values are generated.

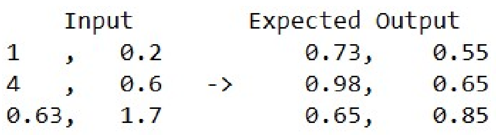


```
     Input
1    ,    0.2
4    ,    0.6
0.63,    1.7
```



Sigmoid Calculator
sigmoid = 1 / 1+e^-x
x: 1

S(X) 0.731058578630074
S'(X) 0.19661193324144993 *sigmoid prime(X)*

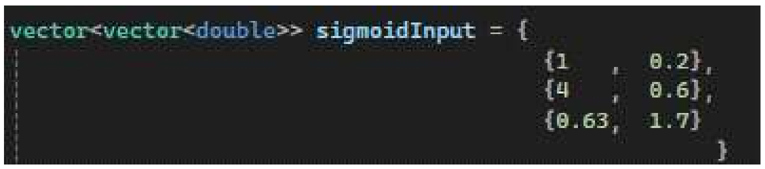


```
       Input          Expected Output
1    ,    0.2              0.73,    0.55
4    ,    0.6     ->       0.98,    0.65
0.63,    1.7              0.65,    0.85
```

The output of each corresponding input is calculated using a sigmoid calculator.

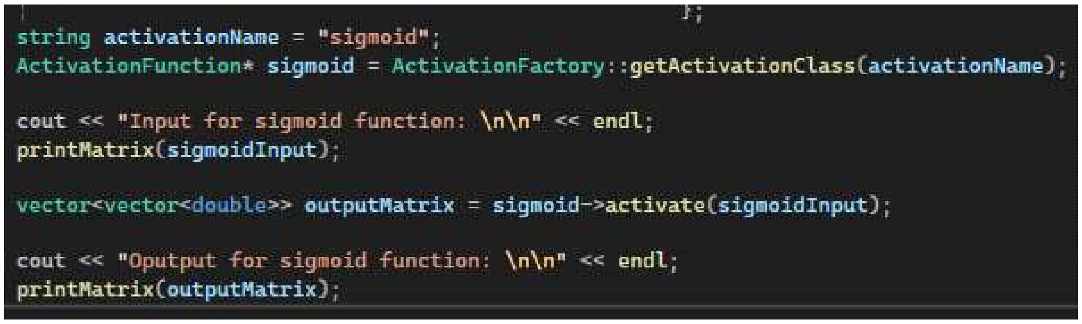
The expected output matrix is generated.

Now to test on the code implemented, input matrix is generated.



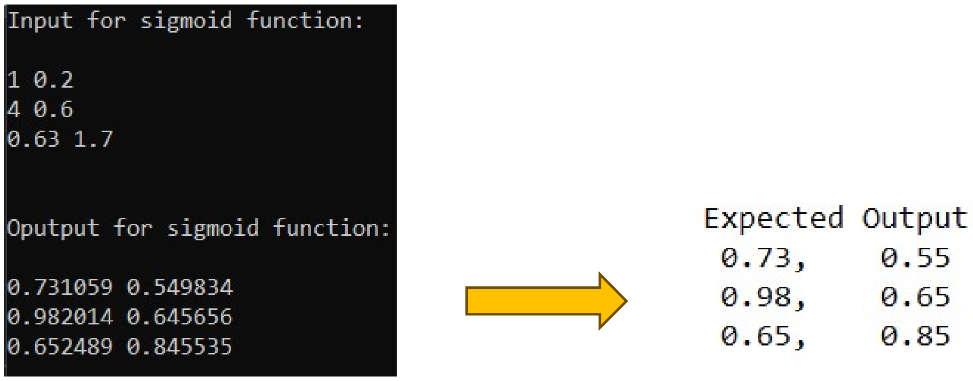
```
vector<vector<double>> sigmoidInput = {
                                        {1   ,  0.2},
                                        {4   ,  0.6},
                                        {0.63,  1.7}
                                                    }
```

The sigmoid function object is created from factory pattern class, and the input matrix is passed as the argument.



```
string activationName = "sigmoid";
ActivationFunction* sigmoid = ActivationFactory::getActivationClass(activationName);

cout << "Input for sigmoid function: \n\n" << endl;
printMatrix(sigmoidInput);

vector<vector<double>> outputMatrix = sigmoid->activate(sigmoidInput);

cout << "Oputput for sigmoid function: \n\n" << endl;
printMatrix(outputMatrix);
```

The output is printed to the console and compared to the expected result.



```
Input for sigmoid function:

1 0.2
4 0.6
0.63 1.7


Oputput for sigmoid function:

0.731059 0.549834
0.982014 0.645656
0.652489 0.845535
```

```
Expected Output
 0.73,    0.55
 0.98,    0.65
 0.65,    0.85
```

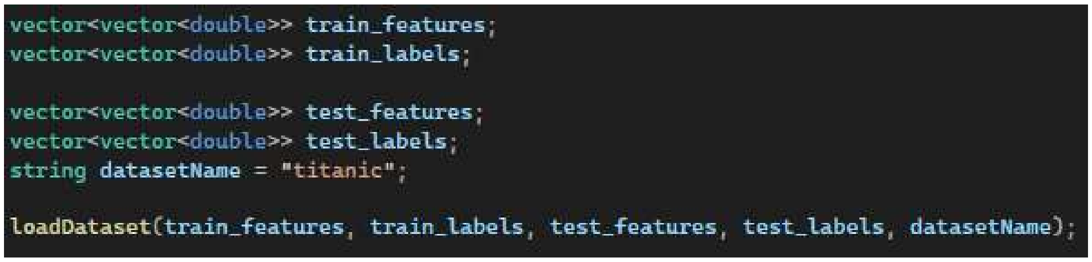
6.1.4.2 Integration and Non-Functional

In this step, it is first proved that each component implemented can come together to work as a whole and make the system work in an integrated manner as expected. This is done by creating a model, training it, and prompting the results of each epoch to the console. Second, the same model created in our framework is also created in Tensorflow and the training time is measured to compare to each other.

To create a sample model, the input dataset used is the open source Titanic Dataset. It contains the information of each passenger in the ship, and the target value indicates if the passenger survived or not. This makes it a binary classification problem.

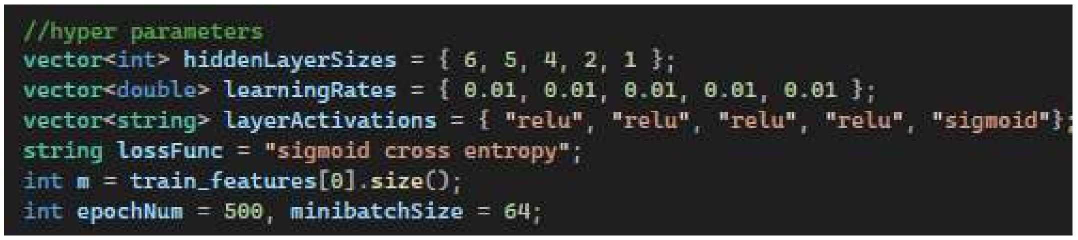
Our Framework Version:

First the proprocess dataset is loaded from the csv file.



```
vector<vector<double>> train_features;
vector<vector<double>> train_labels;

vector<vector<double>> test_features;
vector<vector<double>> test_labels;
string datasetName = "titanic";

loadDataset(train_features, train_labels, test_features, test_labels, datasetName);
```

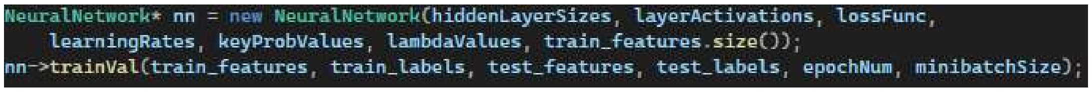
So the deep learning model is created with 5 hidden layer, where the last layer contains one neuron (for binary classsification output between 0 or 1).



```
//hyper parameters
vector<int> hiddenLayerSizes = { 6, 5, 4, 2, 1 };
vector<double> learningRates = { 0.01, 0.01, 0.01, 0.01, 0.01 };
vector<string> layerActivations = { "relu", "relu", "relu", "relu", "sigmoid"};
string lossFunc = "sigmoid cross entropy";
int m = train_features[0].size();
int epochNum = 500, minibatchSize = 64;
```

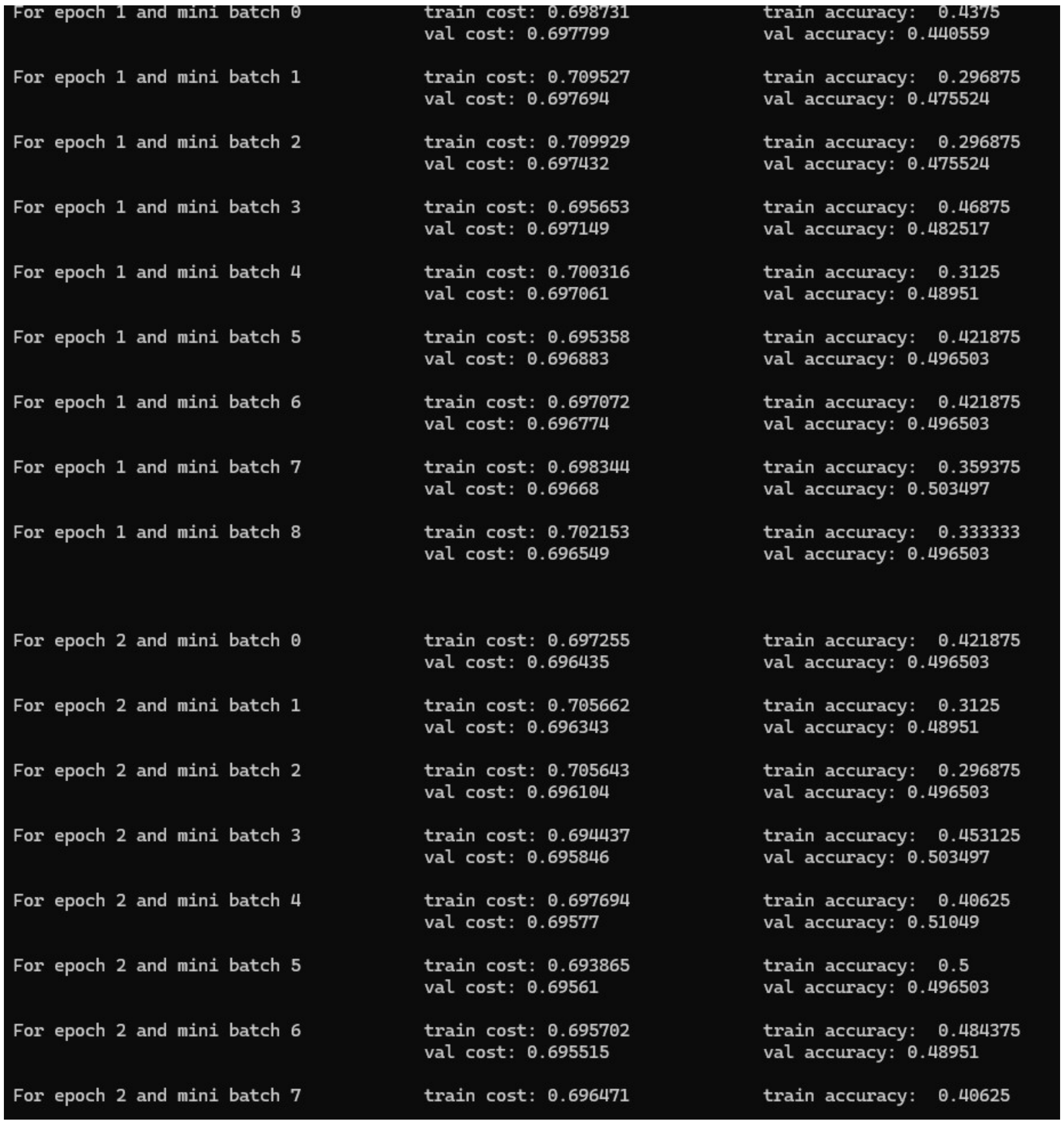
The creation of model starts by specifying the hyperparameters by arbitrary user inputs. The number of neurons in each layer (based on the corresponding index) is given in hiddenLayerSizes vector, like the first layer has 6 neurons, second layer has 5 neurons etc. Learning rates, again specified for each layer (here all layers have learning rate of 0.01). Activation function types are specified as relu except for last layer, which is sigmoid, since sigmoid activation is used in the last layer for binary classification problems. Loss function is selected as sigmoid cross entropy since again this is a binary classification case. The number of epochs are specified as 500 and minibatch size is 64.

Next, the NeuralNetwork class instance is created by passing these hyperparameters as arguments for the constructor, and the trainVal method is executed by passing the train and test inputs and labels (expected results) as arguments.



```
NeuralNetwork* nn = new NeuralNetwork(hiddenLayerSizes, layerActivations, lossFunc,
    learningRates, keyProbValues, lambdaValues, train_features.size());
nn->trainVal(train_features, train_labels, test_features, test_labels, epochNum, minibatchSize);
```

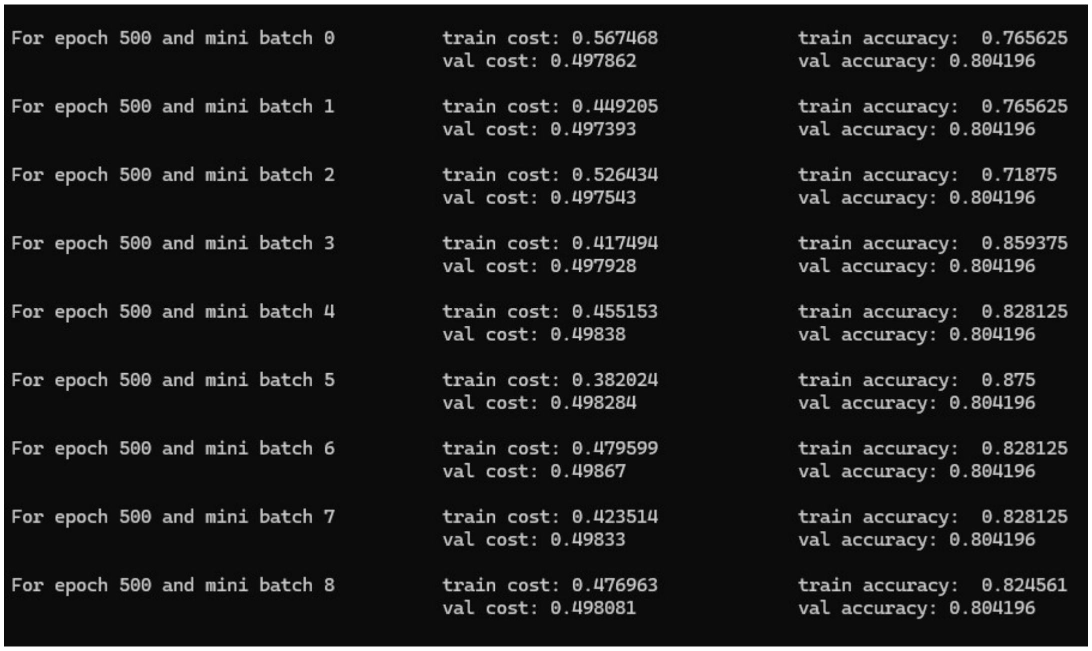
The console output is like this:



```
For epoch 1 and mini batch 0          train cost: 0.698731          train accuracy:  0.4375
                                      val cost: 0.697799            val accuracy: 0.440559

For epoch 1 and mini batch 1          train cost: 0.709527          train accuracy:  0.296875
                                      val cost: 0.697694            val accuracy: 0.475524

For epoch 1 and mini batch 2          train cost: 0.709929          train accuracy:  0.296875
                                      val cost: 0.697432            val accuracy: 0.475524

For epoch 1 and mini batch 3          train cost: 0.695653          train accuracy:  0.46875
                                      val cost: 0.697149            val accuracy: 0.482517

For epoch 1 and mini batch 4          train cost: 0.700316          train accuracy:  0.3125
                                      val cost: 0.697061            val accuracy: 0.48951

For epoch 1 and mini batch 5          train cost: 0.695358          train accuracy:  0.421875
                                      val cost: 0.696883            val accuracy: 0.496503

For epoch 1 and mini batch 6          train cost: 0.697072          train accuracy:  0.421875
                                      val cost: 0.696774            val accuracy: 0.496503

For epoch 1 and mini batch 7          train cost: 0.698344          train accuracy:  0.359375
                                      val cost: 0.69668             val accuracy: 0.503497

For epoch 1 and mini batch 8          train cost: 0.702153          train accuracy:  0.333333
                                      val cost: 0.696549            val accuracy: 0.496503


For epoch 2 and mini batch 0          train cost: 0.697255          train accuracy:  0.421875
                                      val cost: 0.696435            val accuracy: 0.496503

For epoch 2 and mini batch 1          train cost: 0.705662          train accuracy:  0.3125
                                      val cost: 0.696343            val accuracy: 0.48951

For epoch 2 and mini batch 2          train cost: 0.705643          train accuracy:  0.296875
                                      val cost: 0.696104            val accuracy: 0.496503

For epoch 2 and mini batch 3          train cost: 0.694437          train accuracy:  0.453125
                                      val cost: 0.695846            val accuracy: 0.503497

For epoch 2 and mini batch 4          train cost: 0.697694          train accuracy:  0.40625
                                      val cost: 0.69577             val accuracy: 0.51049

For epoch 2 and mini batch 5          train cost: 0.693865          train accuracy:  0.5
                                      val cost: 0.69561             val accuracy: 0.496503

For epoch 2 and mini batch 6          train cost: 0.695702          train accuracy:  0.484375
                                      val cost: 0.695515            val accuracy: 0.48951

For epoch 2 and mini batch 7          train cost: 0.696471          train accuracy:  0.40625
```

The model starts with a low accuracy and high cost value which means it is poor in predicting the result correctly at the beginning of training.

In final epoch, here is the result:

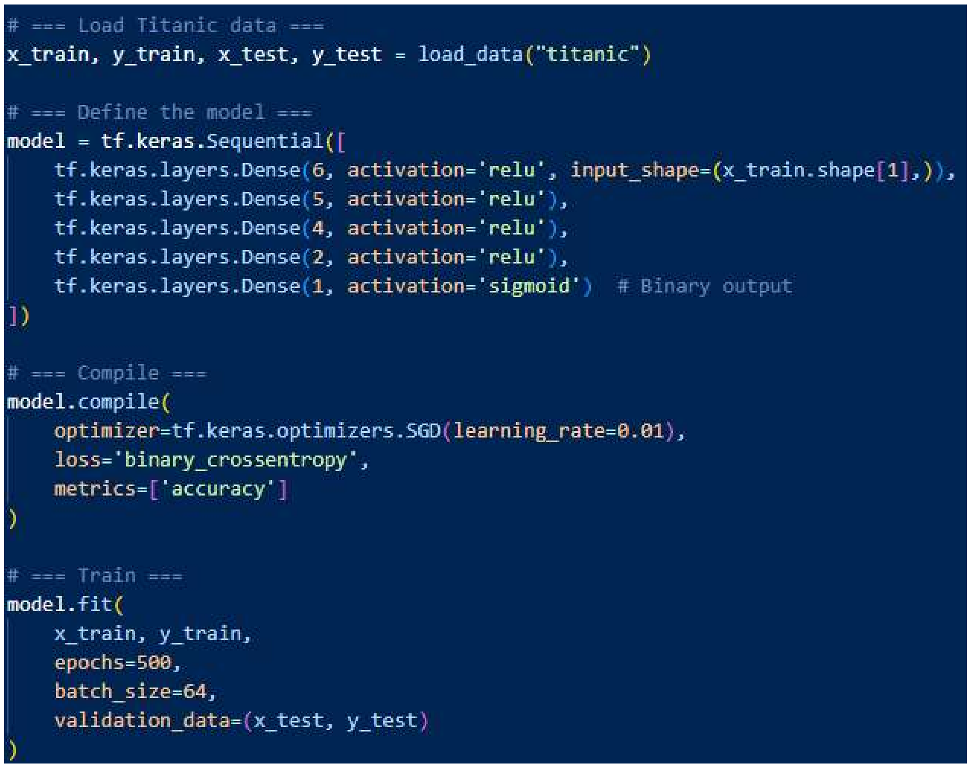


```
For epoch 500 and mini batch 0          train cost: 0.567468             train accuracy:  0.765625
                                        val cost: 0.497862               val accuracy: 0.804196

For epoch 500 and mini batch 1          train cost: 0.449205             train accuracy:  0.765625
                                        val cost: 0.497393               val accuracy: 0.804196

For epoch 500 and mini batch 2          train cost: 0.526434             train accuracy:  0.71875
                                        val cost: 0.497543               val accuracy: 0.804196

For epoch 500 and mini batch 3          train cost: 0.417494             train accuracy:  0.859375
                                        val cost: 0.497928               val accuracy: 0.804196

For epoch 500 and mini batch 4          train cost: 0.455153             train accuracy:  0.828125
                                        val cost: 0.49838                val accuracy: 0.804196

For epoch 500 and mini batch 5          train cost: 0.382024             train accuracy:  0.875
                                        val cost: 0.498284               val accuracy: 0.804196

For epoch 500 and mini batch 6          train cost: 0.479599             train accuracy:  0.828125
                                        val cost: 0.49867                val accuracy: 0.804196

For epoch 500 and mini batch 7          train cost: 0.423514             train accuracy:  0.828125
                                        val cost: 0.49833                val accuracy: 0.804196

For epoch 500 and mini batch 8          train cost: 0.476963             train accuracy:  0.824561
                                        val cost: 0.498081               val accuracy: 0.804196
```

It is obvious that model managed to learn the dataset and improved significantly comparing to its initial scores.

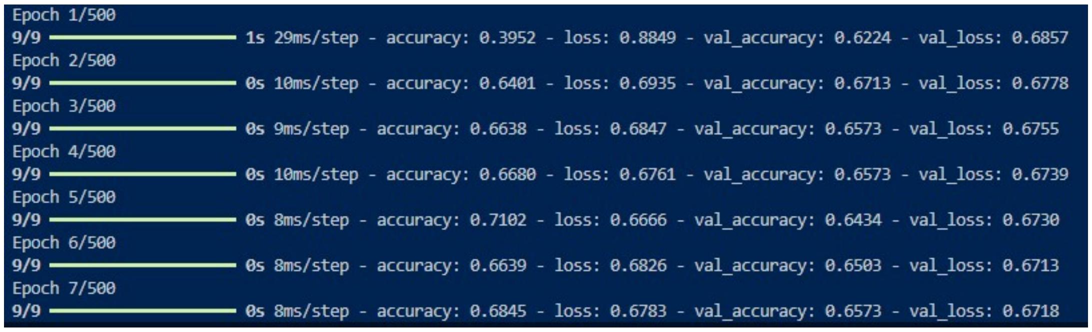
TensorFlow Version:

Now to compare our frameworks performance with Tensorflow, the same model is generated using Tensorflow in Python code and the same dataset is used.



```python
# === Load Titanic data ===
x_train, y_train, x_test, y_test = load_data("titanic")

# === Define the model ===
model = tf.keras.Sequential([
    tf.keras.layers.Dense(6, activation='relu', input_shape=(x_train.shape[1],)),
    tf.keras.layers.Dense(5, activation='relu'),
    tf.keras.layers.Dense(4, activation='relu'),
    tf.keras.layers.Dense(2, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')  # Binary output
])

# === Compile ===
model.compile(
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# === Train ===
model.fit(
    x_train, y_train,
    epochs=500,
    batch_size=64,
    validation_data=(x_test, y_test)
)
```

When it is run, console output:



```
Epoch 1/500
9/9 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - accuracy: 0.3952 - loss: 0.8849 - val_accuracy: 0.6224 - val_loss: 0.6857
Epoch 2/500
9/9 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.6401 - loss: 0.6935 - val_accuracy: 0.6713 - val_loss: 0.6778
Epoch 3/500
9/9 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.6638 - loss: 0.6847 - val_accuracy: 0.6573 - val_loss: 0.6755
Epoch 4/500
9/9 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.6680 - loss: 0.6761 - val_accuracy: 0.6573 - val_loss: 0.6739
Epoch 5/500
9/9 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.7102 - loss: 0.6666 - val_accuracy: 0.6434 - val_loss: 0.6730
Epoch 6/500
9/9 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.6639 - loss: 0.6826 - val_accuracy: 0.6503 - val_loss: 0.6713
Epoch 7/500
9/9 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.6845 - loss: 0.6783 - val_accuracy: 0.6573 - val_loss: 0.6718
```

Comparing the time passed:

- Tensorflow: 55,34 seconds
- Our framework: 18,67 seconds

This concludes that our framework can execute about 3 times faster then Tensorflow, which is a significant amount of difference since deep learning models can take too long to train

6.2 Full Stack Application Integration

This part of the project is planned for future implementation and aims to build a fully functional full-stack application that will be integrated with our custom deep learning framework written in C++. The goal is to provide an intuitive interface for users where they can interact to define model architectures, train the models and view results without needing to deal with any bit of writing code.

The frontend will be developed using React, offering a modern and interactive user interface. Through this interface, users will be able to specify neural network hyperparameters (such as layer types, activation functions of each layer, learning rate, loss function type etc.), start the training process and visualize performance metrics and results.

The backend will be implemented with Node.js, acting as the bridge between the frontend and database enabling data interactions, savings etc. Using REST API.

The C++ framework component will communicate with React frontend via API requests from the frontend.

The C++ framework component will be implemented using Crow, a lightweight framework that can recieve and sende http requests-responses, enabling the React component to make HTTP requests to the C++ component, send training (hyper)parameters in JSON format, and receive the results in response where user can visualize.

Finally, MongoDB will be used as the database system to store user configurations, past model runs, training histories and other relevant data.

This architecture ensures a clean separation of concerns between the UI, the backend logic, and the core training engine, allowing better maintainability, scalability, and performance in the final system.

7. References

https://yuk068.github.io/2025/02/08/cuda-tensorflow-pytorch-guide

https://www.spotfire.com/glossary/what-is-a-neural-network

https://www.geeksforgeeks.org/artificial-neural-network-in-tensorflow/

https://www.pinterest.com/pin/speed-comparison-of-programming-languages--596938125605007485/

https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_basic_concepts.htm

https://medium.com/analytics-vidhya/what-do-you-mean-by-forward-propagation-in-ann-9a89c80dac1b