Neural Networks and Deep Reinforcement Library in C++

Progress report

B.Tech Project Evaluation
November 2020

by

VISHNU AGARWAL

(2019BCS-072)

under the supervision of

Dr. VINAL PATEL

ABV-INDIAN INSTITUTE OF INFORMATION TECHNOLOGY AND MANAGEMENT
GWALIOR-474 015

CANDIDATE'S DECLARATION

I hereby certify that I have properly checked and verified all the items as prescribed in the check-list and ensure that my thesis/report is in proper format as specified in the guideline for thesis preparation.

I also declare that the work containing in this report is my own work. I, understand that plagiarism is defined as any one or combination of the following:

1. To steal and pass off (the ideas or words of another) as one's own
2. To use (another's production) without crediting the source
3. To commit literary theft
4. To present as new and original an idea or product derived from an existing source.

I understand that plagiarism involves an intentional act by the plagiarist of using someone else's work/ideas completely/partially and claiming authorship/originality of the work/ideas. Verbatim copy as well as close resemblance to some else's work constitute plagiarism.

I have given due credit to the original authors/sources for all the words, ideas, diagrams, graphics, computer programs, experiments, results, websites, that are not my original contribution. I have used quotation marks to identify verbatim sentences and given credit to the original authors/sources.

I affirm that no portion of my work is plagiarized, and the experiments and results reported in the report/dissertation/thesis are not manipulated. In the event of a complaint of plagiarism and the manipulation of the experiments and results, I shall be fully responsible and answerable. My faculty supervisor(s) will not be responsible for the same.

Signature:

Name: VISHNU AGARWAL
Roll No.: 2019BCS-072
Date: 09/10/2020

ABSTRACT

This project implements a Neural Network Library and Deep Reinforcement Learning Library using that Library. Neural Network Library can be used to make Multilayer Artificial Neural Networks. There are several parameters of Neural Network which can be modified for more versatile and custom models. If required you can also make your own parameter classes by inheriting the abstract class of that parameter.

Deep Reinforcement Learning Library consists of Environments and an Agent that can be used for your Deep Reinforcement projects. There are multiple environments available to play around with, also an agent whose parameters can be changed according to the requirement. We can also make our own environment inheriting the abstract class of the environment it is so that the provided agent can still work with your custom environment.

Contents

1 Introduction

This Project consists of mainly two libraries. First is Neural Network and Second is Deep Reinforcement Library. Both libraries were made taking the main aim as easy to use. These both libraries are made in such a way, such that, anyone can add their own custom components to work with the library according to their requirements. In further subsections we will be going into details of both libraries.

2 Neural Network Library

My main aim while designing this library was to make it easy to use, such that those who are new to deep learning and do not have a computer with high end graphics card or cpu, they can still enter this field taking the advantage of speed of C++, while also be easily able to make their first neural network. For ex:-

```
auto nn = NeuralNetwork();
nn.add(Layers::Input(3));
nn.add(Layers::Dense(4));
```

Above code generates a 2 layer neural network with first layer taking 3 inputs and last layer giving 4 outputs. These layers will be densely connected, i.e each node of first layer is connected to each node of second layer, with linear activation function and initial weights as zero and quadratic cost function.

Now we will get into more details about each class and feature about this library.

2.1 NeuralNetwork Class

Main class of library is NeuralNetwork. This library mangaes all layers and performs forword propagation, backward propagation and calculates gradients. There are many construtors provided, according to your use, some of those are:-

```
NeuralNetwork(costFn, optimizer); //initialize NN with 0 layers

NeuralNetwork(vector<unique_ptr<Layer>> layers, costFn, optimizer);
    //initialize with given layers.

NeuralNetwork(layers, fstream& fin, costFn, optimizer); //initialize with
    given layers and initialize weights from given file.

NeuralNetwork(const vector<int>& layerSizes,
  vector<MatrixXd> weights, int expectedOutputSize,
  costFn, activation, optimizer); //initialize dense layers with same
      activation for each layer.

NeuralNetwork(layerSizes, weights, activation, optimizer);
```

One example of defining a slightly more advanced neural network using one of these constructors is:-

```
vector<unqiue_ptr<Layer>> layers;
layers.emplace_back(new Layers::Input(4));
layers.emplace_back(new Layers::Dense(7, Activations::Relu()));
layers.emplace_back(new Layers::Linear(Activations::Tanh()));
layers.emplace_back(new Layers::ConstLinear(Activations::Sigmoid()));
auto nn = NeuralNetwork(layers, CostFns::CrossEntropy());
```

This object consists of many important methods/functions too, some of those are:-

2.1.1 add()

```
add(const Layer& layer);
```

It is used to add a layer to your neural network. Ex-

```
nn.add(Layers::Linear(3, Activations::Relu()));
```

2.1.2 putWeights()

```
putWeights(fstream& fin);
```

This method is used to put weights in a file. So that we can upload those weights later in our neural network. ex-

```
nn.putWeights(fin);
```

2.1.3 uploadWeights()

```
uploadWeights(fstream& fin);
```

This method is used to upload weights from a file in our neural network. ex-

```
nn.uploadWeights(fin);
```

2.1.4 printResults()

```
printResults(inputData, expectedOutput, double lambda, double (*accuracyFn));
```

This is method prints the average loss and accuracy. Here accuracyFn is pointer to the accuracy funciton, this parameter is optional. ex-

```
double accuracyFn(const VectorXd& output, const VectorXd& expected);
    //defining accuracy function

nn.printResults(inputData, expectedData, 0, accuracyFn);
```

2.1.5 train()

```
train(inputData, expectedOutput, int batchSize, int totalRounds, double
    lambda = 0); //set printWeightsAndLastChange = false, and
    costRecordInterval = 1000

lossData train(inputData, expectedOutput, double lambda, int batchSize, int
    totalRounds, bool printWeightsAndLastChange, int costRecordInterval);
```

This method is used to train our neural network. In this method printWeightsAndLastChange value tells if to print weights and last updates after the training. And costRecordInterval is the interval of trainings after which it should record average loss data over whole dataset int loss data. This method returns lossData. Below is an example to train neuralNetwork and print loss data.

```
auto lossData = nn.train(inputData, expectedOutput, 30, 10);

for(int i = 0; i < lossData.size(); i++) {
  cout << "after training " << loss[i][0] << " times";
  cout << ", avg loss: " << lossData[i][1] << endl;
}
```

2.1.6 getOutput()

```
getOutput(inputData);
```

This method is used to get output from our neural network. It returns output in the form of MatriXd. ex-

```
auto output = nn.getOutput(inputData);
for(int i = 0; i < output.rows(); i++) {
  cout << "output for row[" << i << "]: ";
  cout << output.row(i) << endl;
}
```

2.1.7 clone()

```
clone();
```

This method is used to make a deep copy of your neural network. It returns unique_ptr<NeuralNetwork> to the copied object. ex-

```
nn.printWeights(); //used to print weights of our model.
auto nnCopy = nn.clone();
nnCopy->printWeights();
```

2.1.8 getWeights()

```
getWeights();
```

This method is used to get weights of our neural network. It returns in the form of vector<MatrixXd>. An example to print weights after getting weights from this method is:-

```
auto weights = nn.getWeights();
for(int i = 0; i < weights.size(); i++) {
  cout << "Weights for layer[" << i << "]:\n";
  cout << weights[i] << "\n\n";
}
```

2.2 Layer Class

Layer class defines the layer of a Neural Network. There many types of layer provided, where each layer is inherited from the abstract class Layer. Each layer defines the number of nodes in the layer, weight matrix according to last layer, forward and back propagation through the layer. Anyone can define their own layer by inheriting the abstract class Layer. I provide 4 types of layers in library:-

2.2.1 Input Layer

For The Neural Network this should always be the first layer, and there should always be only one input layer in whole NN. For this layer there are no weights and back propagation defined. This layer can be added to NNas follows:-

```
nn.add(Layers::Input(numOfInputs));
```

2.2.2 Dense Layer

This layer defines a layer which is densely connected to the last layer, such that each node of this layer is connected to every node of last layer. An example of a dense layer with activation function as leaky relu and weights/kernel initializer as he normal:-

```
nn.add(Layers::Dense(4, Activations::LeakyRelu(), Initializers::HeNormal));
```

2.2.3 Linear Layer

This layer defines a layer which is linearly connected to the last layer, such that this layer has equal number of nodes as the last layer and each node is connected to the corresponding node in the last layer. An example of a Linear layer with activation functiona as Sigmoid and kernel initializer as Glorot Normal:-

```
nn.add(Layers::Linear(Activations::Sigmoid(), Initializers::GlorotNormal()));
```

2.2.4 Constant Linear Layer

This layer is exactly as last layer, just with constant weights, which are all equal to one. This layer is helpful when we want to add an activation function above the last layer. An example of a constant linear layer to apply Tanh activation above last layer:-

```
nn.add(Layers::Dense(3, Activations::Relu()));
nn.add(Layers::ConstLinear(Activations::Tanh()));
```

2.3 ActivationFunction Class

Activation Function class mainly defines two methods, first activation function and second activation function gradient. One can create his own activation function by inheriting abstract class Activation. There are currently 5 types of activation function provided with the library, which are:-

2.3.1 Sigmoid

This is the sigmoid activation function, this activation function takes a vector of real numbers and returns the vector of real numbers ranging from (1 - ϵ) to ϵ. Formula used to generate output:-

$$g(z) = (1 - 2\epsilon)\frac{1}{1 + e^{-z}} + \epsilon$$

$$g^{'}(z) = (1 - 2\epsilon)g(z)g(1 - z)$$

2.3.2 Tanh

This is the tanh activation function, this activation function takes a vector of real numbers and returns a vector of tanh(number). All the edge cases are considered while creating this function such that it never returns 1, -1 and 0, to avoid nan situations. Formula used to generate output:-

$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g^{'}(z) = 1 - (g(z))^{2}$$

2.3.3 Relu

This is the relu activation function, this activation function takes a vector of real numbers and returns the vector of positive real numbers, with all negative numbers converted to 0. Formula used to generate output:-

$$g(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

$$g^{'}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

2.3.4 LeakyRelu

This is the leaky relu activation function, this activation function takes a vector of real numbers and returns the same number if number is greater than zero and α * number if number is smaller than zero. Formula used to generate output:-

$$g(z) = \begin{cases} z & \text{if } z \geq 0 \\ \alpha z & \text{if } z < 0 \end{cases}$$

$$g^{'}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ \alpha & \text{if } z < 0 \end{cases}$$

2.3.5 Linear

This is the linear activation function, this activation function takes a vector of real numbers and returns the same number. This is activation function is default activation function for every inbuilt layer. Formula used to generate output:-

$$g(z) = z$$

$$g^{'}(z) = 1$$

2.4 CostFunction Class

Cost Function Class also mainly defines two methods one is cost function and the other is cost function gradient. One can create its own cost function by inheriting CostFunciton abstract class. There are currently 2 types of cost function provided with library.

2.4.1 Binary Cross Entropy

This cost funciton is mainly used in logistic problems, to calculate error when expected output is either 1 or 0 and output from NN is between 1 and 0. Formual used to calculate cost:-

$$f(y, x) = -ylog(x) - (1 - y)log(1 - x)$$

$$\frac{\delta f(y, x)}{\delta x} = \frac{x - y}{x(1 - x)}$$

An example to define this cost function:-

```
auto nn = NeuralNetwork(CostFns::CrossEntropy());
```

2.4.2 Quadratic

This cost function can be used with outputs with any range. It is most widely used with NN output as Linear. Formula used to calculate cost:-

$$f(y, x) = \frac{1}{2}(y - x)^2$$

$$\frac{\delta f(y, x)}{\delta x} = x - y$$

An example to define this cost function:-

```
auto nn = NeuralNetwork(CostFns::Quadratic());
```

2.5 KernelInitializer Class

This class is used to initialize weights for layers, it sets biases as zero. One can inherit abstract class KernelInitializer to define its own initializer. There 4 types of Kernel Initializers provided in libarary:-

2.5.1 Zero

This is the default initializer for inbuilt layers. This initializer initializes all weights as zero. two examples using this initializer is:-

```
nn.add(Layers::Dense(4, Initializers::Zero()));
nn.add(Layers::Dense(4));
```

In the above examples both lines adds a dense layer with 4 nodes in our NN.

2.5.2 Ones

This initailizer initializes all weights as one and all biases as zero. an example using this initializer is:-

```
nn.add(Layers::Dense(4, Initializers::Ones()));
```

2.5.3 He Normal

This initializes initalizes all weights accr to the he normal method of initializing weights, in which each weights are initialized in a random normal distribution. This Initializer is mainly used with Relu activation function. Formula used to initialize weights:-

$$\mu = 0$$

$$\sigma = \frac{2}{fan\_in}$$

an example using this initializer is:-

```
nn.add(Layers::Dense(4, Initializer::HeNormal()));
```

2.5.4 Glorot Normal

This initializes initalizes all weights accr to the he normal method of initializing weights, in which each weights are initialized in a random normal distribution. This Initializer is mainly used with Sigmoid activation function. Formula used to initialize weights:-

$$\mu = 0$$

$$\sigma = \frac{2}{fan\_in + fan\_out}$$

an example using this initializer is:-

```
nn.add(Layers::Dense(4, Initializer::GlorotNormal()));
```

2.6 Optmizer Class

This class is used to define optmizers for your NN. Optimizers are used to update weights according to gradients. One can inherit Optimizer Abstract class to create its own optimizer. currently there are 2 types of optmizers provided in the library:-

2.6.1 Stochastic Gradient Descent

This is the most commonly used optmizer. It simply updates weights according to learning rate, momentum and gradients. This is the default optmizer for NeuralNetwork class. Defualt value of learning rate is 0.01 and momentum is 0. Some example of using this optimizer in NN:-

```
auto nn = NeuralNetwork(Optimizers::SGD());
auto nn = NeuralNetwork(Optmizers::SGD(0.01, 0));
auto nn = NeuralNetwork();
```

2.6.2 Adam

This is one of the advanced optimizers, this is mostly used with batch learning. This was made for better preventing NN to stuck at local minimas. This is algo is according the paper published on Adam Optimizer. Examples of using this optimzer in NN:-

```
auto nn = NeuralNetwork(Optimizers::Adam());
auto nn = NeuralNetwork(Optimizers::Adam(0.001, 0.9, 0.999));
```

3 Deep Reinforcement Learing Library

This library was made with the help of above library. It provides Some Environments to practice your algorithm and a Single player agent. This library was also made by taking in mind the aim of easy to use. This library requires Eigen and SFML libraries.

3.1 Environment Class

This class defines an environment for the agent. One can inherit Environment Abstract class to create their own environment. some of the main methods provided by environment class are:-

```
string getName(); //returns name of env, which is used to differentatie
    between environments.

StepInfo step(int actionIndex); //take a step according to action, and
    returns StepInfo i.e rewards and isDone

void resetEnv(); //reset environment to the starting point.

RowVectorXd getInputs; //get state in the form of row vector for NN.
```

There are currently 2 types of environments provided with the library.

3.1.1 Mouse And Cheese

This is the simple game in which cheese is generated randomly at any position after each time mouse eats it and aim of the game is for mouse to eat cheese as many time as possible without colliding with walls. State of the environment is returned in the form of row vector of size 12. Actions to move mouse are 0, 1, 2 and 3 for UP, DOWN, LEFT and RIGHT respectively.

To create this environment, we need to provide window size, length of side of square game window, body size, diameter of mouse, and color of mouse.

```
Mouse(int windowSize, int bodySize, sf::Color color);
```

An example of using this environment and inbuilt agent to teach an agent to play this game is:-

```
auto nn = NeuralNetwork(CostFns::Quadratic());
nn.add(Layers::Input(12));
nn.add(Layers::Dense(7, Activations::LeakyRelu(), Initializers::HeNormal()));
nn.add(Layers::Dense(4, Initializers::HeNormal));

Envs::Mouse env(600, 60, sf::Color::Red);
Agent agent(nn, env);
agent.learn();
```

3.1.2 Flappy Bird

This is the famouse and simple game in which we have to prevent the bird from falling and colliding with pipes. State of the environment is returned in the form of row vector of size 3. Actions to move bird are 0 and 1 for NOTHING and JUMP respectively.

To create this environment, we need to provide screen width, screen height, body width, body height, pipe velocity, jump velocity, gravity.

```
FlappyBird(int screenWidth, int screenHeight, int bodyWidth, int bodyHeight,
    int pipeVelocity, int jumpVelocity, int gravity);
```

default values are as follows:-

```
int pipeVelocity = 1;
int jumpVelocity = 10;
int gravity = 1;
```

An example of using this environment and inbuilt agent to teach an agent to play this game is:-

```
auto nn = NeuralNetwork(CostFns::Quadratic(), Optimizers::Adam());
nn.add(Layers::Input(3));
nn.add(Layers::Dense(5, Activations::LeakyRelu(), Initializers::HeNormal()));
nn.add(Layers::Dense(3, Activations::LeakyRelu(), Initializers::HeNormal()));
nn.add(Layers::Dense(2, Initializers::HeNormal()));

Envs::FlappyBird env(300, 600, 10, 10, 2);
Agent agent(nn, env);
agent.learn();
```

3.2 Agent Class

This class defines an agent for single player games. Agent constructor is as follows:-

```
Agent(NeuralNetwork, Environment, void (*print)(MatrixXd matrix));
```

print function is optinal parameter which is used to print total turns and rewards after each episode. where matrix is of type:-

- matrix(i, 0) = total times action index i called this episode.
- matrix(i, 1) = total reward achieved for action index i in this episode.

Important public members:-

```
double explorationRate = 1;
double explorationDecay = 0.995;
double minExplorationRate = 0.01;
int totalEpisodes = 2000;
int maxActionsPerEp = 200;
int batchSize = 50;
```

```
int totalMemoryDbSize = 500;
int updateTrainNnInterval = 150;
double discount = 0.5;
```

all the above are public members which are customizable according to our needs.

4 Expected outcome

- The provided library will allow people to make Deep Learning Models in C++, which is a significantly faster than python.
- The provided library will let beginners to easily start with Deep Learning, even without a super fast laptop or PC.
- Allow people to also easily make Deep Reinforcement Models in C++.
- Allow people to simple make Environments and use provided Agent to train Neural Network.

5 3rd Party Libraries Used

Table 1: 3rd party libraries used in this project.

Sr. No.	Version	Developer	Library used	Objective
1	3.3.8	Benoît Jacob Gaël Guennebaud	Eigen[1] C++	For Matrix and Vector classes needed for both Libraries.
2	2.5.1	Laurent Gomila	SFML[2] C++	For functionality to draw Graphics using C++ needed for Deep Reinforcement Library

6 Key Related Research

Table 2: Research done for building this project.

Sr. No.	Technique used	Objective
1	Neural Networks[3]	To Learn about Neural Networks
2	Activation Functions[4]	To learn the maths behind different activation functions
3	Cost Functions[5]	To learn the maths behind different cost functions
4	Weight Initializers[6]	To learn about different Kernel Initializers
5	SGD Optimizer[7]	To learn about Stochastic Gradient Descent Optimizer
6	Adam Optimizer[8]	To learn about Adam Optimizer
7	Deep Reinforcement Learning[9]	To build Deep Reinforcement Learning Agent

7 Results

To show off these libraries I have used these libraries for 3 small projects.

7.1 Classify points inside or outside circle

In this example, I used Neural Network library for supervised learning to teach computer to classify points to be either inside or outside the circle.

First I generate 500 random points and classify them inside or outside the circle. Neural Network used was of 4 layers, such that there were 2 hidden layers, 1 input layer and 1 output layer.

Following is the test data. Note:- Test Data is not the data used for training it is only for testing purpose.

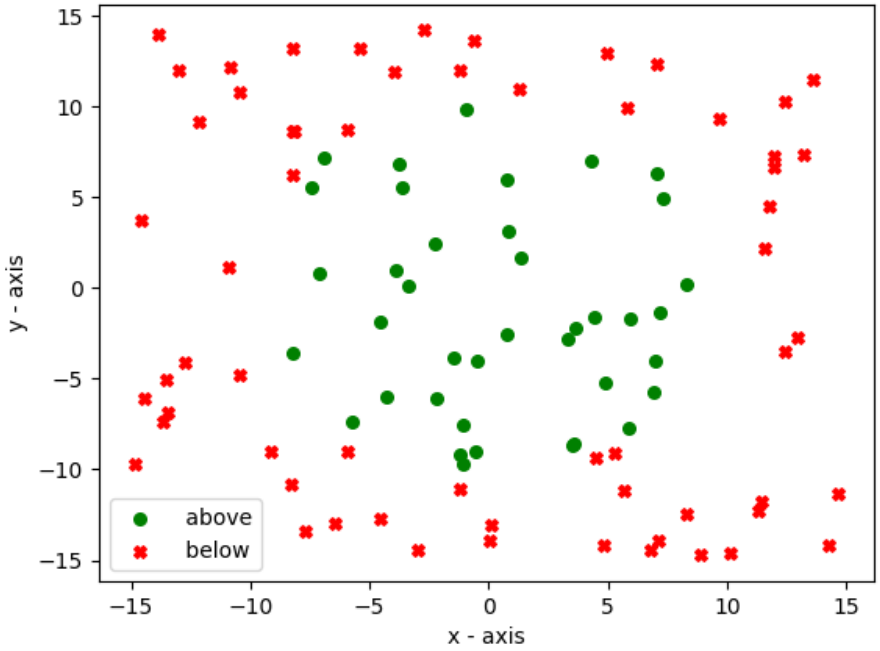
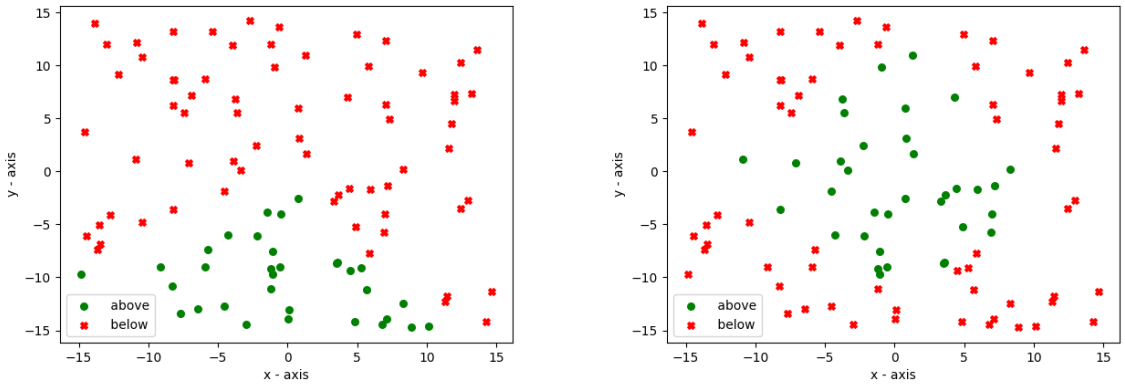


Figure 1: Test Data, which is expected as output

I trained my NN with batches of size 50 for 500 rounds, on the dataset of 500 randomly generated points. Results of training are shown in Figure 2. With rate of loss with training after each batch in Figure 3.



(a) Initial output before training (b) Output after training

Figure 2: NN outputs before and after training

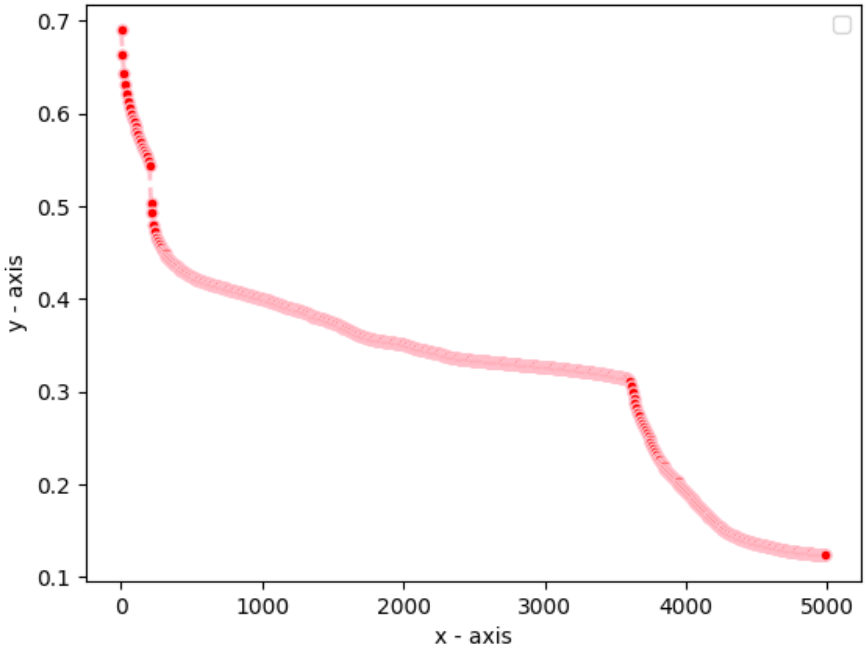


Figure 3: Graph of loss vs batch training

Initially accuracy for test data was about 13, but after training accuracy was increased to greater than 92. Below is also the code used for generating and training neural network.

```
auto nn = NeuralNetwork(Optimizers::Adam(0.01));
nn.add(Layers::Input(2));
nn.add(Layers::Dense(3, Activations::LeakyRelu(), Initializers::HeNormal()));
nn.add(Layers::Dense(2, Activations::LeakyRelu(), Initializers::HeNormal()));
nn.add(Layers::Dense(1, Activations::Sigmoid(),
    Initializers::GlorotNormal()));

MatrixXd inputData = generateInputData(500, 2);
MatrixXd outputData = generateOutputData(inputData, 1);
inputData = squashInputs(inputData); //to squash coordinates between -1 to 1

nn.trainNetwork(inputData, outputData, 50, 500);
```

7.2 Mouse and Cheese

In this example used Neural Network and Deep Reinforcement Learning Library to train computer to control mouse to follow cheese and prevent walls.

Following code was used for training and generating Neural Network.

```
auto nn = NeuralNetwork(Optimizers::Adam(0.00025), CostFns::Quadratic());
nn.add(Layers::Input(12));
nn.add(Layers::Dense(24, Activations::LeakyRelu(),
    Initializers::HeNormal()));
nn.add(Layers::Dense(24, Activations::LeakyRelu(),
    Initializers::HeNormal()));
nn.add(Layers::Dense(4, Activations::Linear()));

Env::Mouse body(600, bodySize);
Agent agent(nn, body);
agent.learn();
```

I trained this Neural Network for about 9 hours. Figure 4 shows the graph of rewards vs episodes for last 7000 episodes, where each point shows average reward for last 20 episodes.

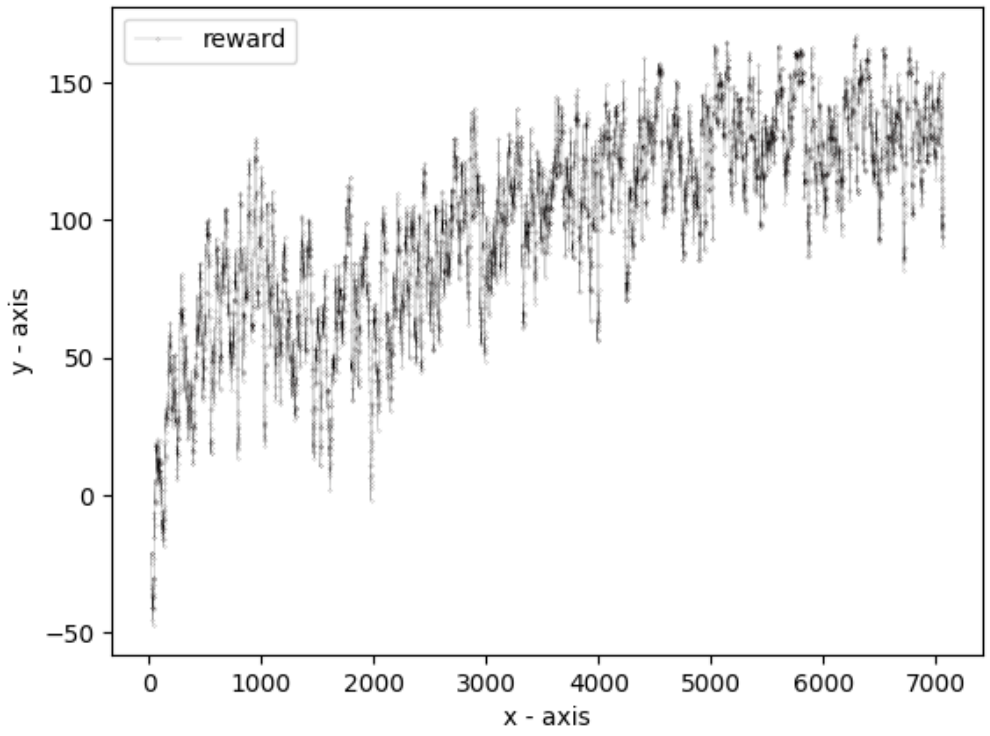


Figure 4: Graph of rewards vs episodes

7.3 Flappy Bird

In this example I used NeuralNetwork and Deep Reinforcement Learning Library to train computer to control flappy bird, to avoid pipes and falling down.

Following code was used for training and generating Neural Network.

```
auto nn = NeuralNetwork(Optimizers::Adam(), CostFns::Quadratic());
nn.add(Layers::Input(3));
nn.add(Layers::Dense(5, Activations::LeakyRelu(), Initializers::HeNormal()));
nn.add(Layers::Dense(3, Activations::LeakyRelu(), Initializers::HeNormal()));
nn.add(Layers::Dense(2, Initializers::HeNormal()));

FlappyBird env(width, height, bodyWidth, bodyHeight, 2);
Agent agent = Agent(nn, env, print);
agent.maxActionsPerEp = 1750;
agent.learn();
```

I trained this Neural Network for about 2 hours or 1750 episodes. And under this period of time this Neural Network had almost mastered this game. Figure 5 shows the reward vs episodes graph, where each point shows the average reward in last 20 episodes.

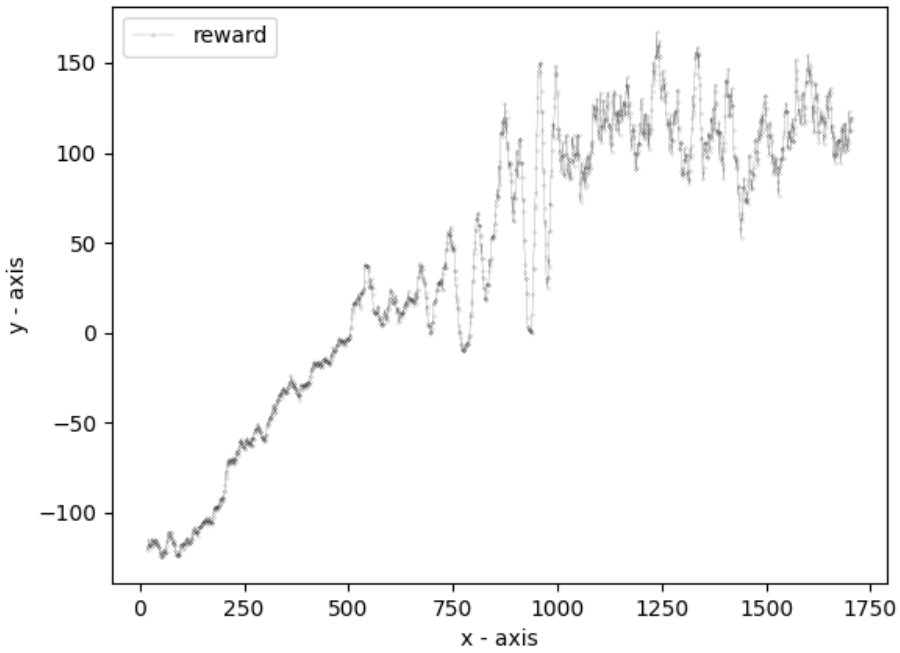


Figure 5: Graph of rewards vs episodes

In the above graph only times the computer loses was when it explores, which happens for every 1 in 100 moves after certain amount of time int training. And limited rewards per episodes is mostly due to max actions per episodes limit which was of 200 for this game.

8 Conclusion

While making this project I made my progress solving a lot of problems, researching a lot about how neural networks works, maths begind the working of neural networks and than implementing it in code was even more challenging. I researched and studied a lot about how Reinforcement Learning works using Q-Learning and than how does Deep Reinforcement Learning works using Deep Q-Learning by approximating Q-Table with the help of Neural Networks. And all this is just the fraction of everything that I researched and studied.

Second the most challenging part was implementing all of this in C++ code, and not just implementing it, I also had to make sure almost everything was customizable + it was super easy for anyone to use these libraries, all of this required deep knowledge of C++ and OOPS concepts also this was a really challenging question of software and class designing, one of the most challenging part was to take child classes as parameters details of this problem can be seen HERE, this ques was asked by me and after a lot of research and brain storming I also answered it myself, while earlier I was planning to go with the most upvoted answer, but than I figured out the best solution myself.

This project was full of difficult challenges like these specially about program architecture but it was overall a lot of fun.

References

[1] B. J. G. Guennebaud, "Eigen c++ library," http://eigen.tuxfamily.org/index.php?title=Main_Page.

[2] L. Gomila, "Sfml c++ library," https://www.sfml-dev.org/.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[4] S. Sharma, "Activation functions in neural networks," https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[5] P. (https://stats.stackexchange.com/users/78563/phylliida), "A list of cost functions used in neural networks, alongside applications," Cross Validated, uRL:https://stats.stackexchange.com/q/154880 (version: 2020-10-16). [Online]. Available: https://stats.stackexchange.com/q/154880

[6] J. Dellinger, "Weight initialization in neural networks: A journey from the basics to kaiming," https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79.

[7] A. V. Srinivasan, "Stochastic gradient descent — clearly explained !!" https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31.

[8] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[9] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *arXiv preprint arXiv:1811.12560*, 2018.