

CM0671 Artificial Intelligence and Affective Computing

**Report :**

**Implementing The Monster Artificial Intelligence With Multi-layer Perceptron and Multi-Threading In Two-Dimensional Defense Game**

Name : Chanhaeng Lee

Student No. : 17030159

Tutor : Dr Li Zhang

Date : 3rd January 2018

**Contents**

1. Introduction ------------------------------------------------------------------ 3
2. Problem Recognition and Background Research -------------------- 3
3. The Application of Multi-Layer Perceptron to monster AI -------- 5
4. The Application of Resilient Propagation to monster AI ----------- 6
5. The Application of MLP using Multi-Threading to monster AI --- 7
6. Conclusion and Future Work --------------------------------------------- 8
7. Appendix ---------------------------------------------------------------------- 9
8. Reference -------------------------------------------------------------------- 21
9. Introduction

Implementing the monster AI that does simple behavior like going to the target and attacking it is simple. However, If the advanced monster AI is needed which decides the behavior of a monster like human, the problem is harder. Because the way human thinks and behaves is complex. However, the machine learning, especially deep learning, can give a chance to imitate the way processing information like human. This report gives an emphasis on implementing the monster AI with Deep Learning algorithm in two-dimensional defense game. ( *Appendix 1* )

1. Problem Recognition and Background Research
2. Problem Recognition

There are two things important to consider when implementing an advanced monster AI. One of the two is the interaction between monster and game factors. Monster AI should respond in real time according to the environment. It means accepting the inputs and then deciding the output behavior like human thinks and then behaves. The other one to consider is the performance. The performance in game is very important, because if a user feel uncomfortable with the performance of a game, the user will not play the game more. Therefore, these problems have to be solved to implement the good monster AI.

As for the first problem, many variables can be chosen as inputs of the monster AI, such as the number of monsters, monster HP, player HP, and so on. The more the input variables are, the harder it is to make a connection between input and output(behavior). However, making a smart monster AI cannot help adding many variables to input. So, the solution to this problem is using the multi-layer perceptron(MLP). It can take many inputs and make the output according to training data or purpose. So, the main feature of monster AI on this report is using the MLP.

When it comes to the performance, many things can be considered to reduce the computation cost, such as the time to train the neural network, the period to decide the behavior of the monster, the structure of the game play, and so on. When using the neural network, the training speed of neural network is very important. Because after training the neural network, the neural network can be used. For this reason, if the time to train the neural network is too much long, a user can feel uncomfortable with playing a game. In addition, the natural behavior of the monster is important to the user. So, when using the multi-layer perceptron as monster AI, the period to decide the behavior of the monster needs to be considered. Because if the monster’s behavior is decided every frame (usually, 0.016 second), the user can see the monster jitter in some cases. The structure of the game play is also significant because it can affect the way to use the multi-layer perceptron as monster AI.

1. Background Research

The output behavior and input variable should be determined to make monster AI. Considering that the game is a kind of defense game, monsters should destroy the defense object or kill the player smartly. So, the entire purpose of monster AI is to balance the monster attack to the player or the defense object. Consequently, the output behaviors should be :

1. Attack the character
2. Attack the Defense Object
3. Flee

The inputs of monster AI should be chosen with relation to the output behavior. So, the inputs are decided, assuming human control the monster to attack the character or defense object. The inputs are :

1. Monster HP
2. Character HP
3. Defense Object HP
4. Distance between character and monster
5. Distance between defense object and monster
6. The number of friend monsters

And then, the connection between input and output should be made. The connection is regulated by these rules :

1. A monster should do only one behavior.
2. If the HP of character or defense object is less than 30%, a monster attacks the less HP and near one.
3. If the HP of a monster is more than 80%, the monster only attacks the defense object.
4. If the HP of a monster is less than 80%, the monster attacks the near one.
5. If the HP of a monster is less than 30% and the number of friend monsters is less than 30%, the monster flees.

As the result of the research of the fast neural network for the performance of the game, there is another propagation algorithm for the multi-layer perceptron, Resilient Propagation(Rprop). Rprop is a local adaptive learning scheme, performing supervised batch learning in MLP (Riedmiller, 1994, p. 1). The benefits of Rprop compared to Bprop are fast and not requiring parameters. In addition, the Rprop can use the multi-threading because the Rprop performs a regular feed forward and calculate Errors for all training data set, and then apply all the errors to the weight.

1. The Application of Multi-Layer Perceptron to monster AI
2. Design and Implementation

Using multi-layer perceptron requires the training data set for the desired output. However, because there is no training data set and standard rules for the defense game, the training data set has to be made. For this reason, the log of input variables in game was collected and used as training data. ( *Appendix 2* ).

After collecting the training data, the experiment is performed that the implemented MLP works out correctly. The result of the test indicates that the MLP can reach to the 99% accuracy with the collected training data set. ( *Appendix 3* ). However, the goal of accuracy, learning rate and momentum parameters and epoch limit should be adjusted to ensure the performance of the game. Even if the perceptron can reach to the 99% accuracy, it takes 32 seconds to have the accuracy. Therefore, the proper value of the parameters should be set not to harm the performance.

1. Application to the game

The structure of the code of this game is object-oriented. So, the MLP class is made, and then the monster AI class including the MLP class is implemented. The monster AI class have one neural network, and monsters use the network to decide its behavior. The training process happens in the start of the game. And when a stage is clear, the monster AI class trains the neural network again to make the game more difficult to play. Whenever the monster AI class trains the neural network again, the error rate is adjusted lower than before. In addition, because the accuracy of monster AI on each stage should be adjusted, the different noises for the input on each stage are applied (not the noise for avoiding the overfitting of neural network). ( *Appendix 4* )

Each monster packs the input of monster information and pass it to the neural network, and then get the decision result from the neural network. ( *Appendix 5* ) Each monster behaves according to the decision result.

1. Testing

The neural network works well for deciding the behavior of the monster AI. However, there are two problems identified. One problem is a player can see some monsters jitter in case, because some inputs can be in the ambiguous state that the little change of the inputs alternates two behaviors constantly. To solve this problem, the code is fixed to make a decision every two seconds. ( *Appendix 6* ) The other problem is that the program is not responding for a little time because of the neural network training. This is a kind of the performance problem. For this reason, Resilient propagation, so-called faster propagation algorithm, needs to be implemented to solve this problem.

1. The Application of Resilient Propagation to monster AI
2. Design and Implementation

The Resilient Propagation(Rprop) don’t require parameters, but use similar structure and variables. So, the Rprop can be implemented in the same MLP class. ( *Appendix 7* ) The structure of training the neural network with Rprop is a little different from back propagation (Bprop). ( *Appendix 8* ) After calculating all the errors from all the data training, the weights of each layer will be updated. This structure makes it possible to use multi-threading when training the neural network. The multi-threading can be used to calculate all the errors, splitting the training data by the number of the threads.

1. Testing

Before applying the Rprop to the monster AI, the experiment is performed that the Rprop works correctly for the training data. ( *Appendix 9* ) However, the result of the test is not better than the Bprop, because there is the inconsistency of the test result with Rprop. ( *Appendix 10* ) On the other hand, the test with Bprop can make the consistency for the result. There is also even the case that the value of mean square error oscillates without reaching the goal of mean square error when using Rprop. So the training doesn’t stop. The inconsistency doesn’t give the control of difficulty of the game and makes uncertainty of the program behavior.

Consequently, The Rprop is not proper to use in monster AI of Game program because of its unstableness. It can be valid in case that there are lots of training data and the training environment endures the unstableness. Therefore, another solution should be developed to solve the performance problem of monster AI in game.

1. The Application of MLP using Multi-Threading to monster AI
2. Design and Implementation

Because Using the Rprop for monster AI is not proper, the normal back propagation should be used. The performance problem that the program isn’t responding because of the training process in the middle of changing the stage can be circumvented by two strategies.

The first strategy is training the neural networks of each stage in the start time of the program. Generally, a user can wait for the loading time for the start of game program. Using this point, if different neural networks of each stage are trained in the loading time, a user will not go through the problem that the program is not responding because of the training. However, training the multiple neural networks also requires the long loading time. So, there is a chance that a user can feel uncomfortable with the performance. The second strategy is used in this point. The second strategy is training multiple neural networks by using the multi-threading at the same time. Training many neural networks sequentially requires the time multiplied by the number of stage, but when using multi-threading, the time can be smaller. With these two strategies, monster AI class is implemented like ( *Appendix 11* )

1. Application to the game

After Applying the monster AI class and the strategy into the game, there is a problem about the accuracies between neural networks. ( *Appendix 12* ) Even though the parameters are set to adjust the accuracies for each stage , there may be a coincidence the neural network accuracy of earlier stage is higher. In this case, the code to train the Neural network again by using multi-threading in three seconds is implemented. ( *Appendix 13* )

1. Testing

The test results came out as intended. The accuracies differ and are valid on each stage. ( *Appendix 14* ) So, it affects the behavior of the monsters. As a player go to the next stage, it’s harder to clear the stage because of the more clever monsters to make the player fail.

1. Conclusion and Future Work

The interaction and the performance are the main problem to solve on this report. The interaction problem between many of the game factors and monsters can be solved by setting the input, outputs, and the rules to make a behavior. The MLP is applied to process the inputs, the outputs and the rules. To solve the performance problem caused by the MLP, resilient propagation was considered. However, because it’s unstable to use in the game, two other strategies are applied: training in the start of the program, multi-threading. As the result of the research and implementation on this report, the monster AI works as intended.

The future work to implement for the smarter monster AI is the unsupervised learning. Because the supervised learning depends on the training data set, the monster AI only decides the behavior by the training data set. It means if there is no more training data set, the AI cannot improve. However, the unsupervised learning can be the solution. Because the AlphaGo Zero, developed by DeepMind and trained by unsupervised learning, proves the effectiveness of the unsupervised learning even without the training data set.

1. Appendix
2. Development Platform and Game Description

* Operation system : Windows 10
* Window Software Development Kit(SDK) : 10.0.15063.0
* Development IDE : Visual Studio 2017
* Programming language : C/C++
* Development library : Simple DirectMedia(SDL) 2.0
* The execution platform of the developed program : Win32(x86)

The kind of the game developed on the above platform is a defense game, which is the game that a player protects a defense object from many monsters. The developed game consists of three stages now, and a player should protect the defense object from monsters, which means not making the HP(Health Point) of the defense object the zero. The player should also keep his or her HP. If not keeping either the HP of two, the player will fail in the game. The player should kill the monster to win the game, using the short-distance and the long-distance attack to the monster. If the player kills all of monsters at each stage, the player can go to the next stage. The number of the kind of monsters is two. One is the Dust monster and the other one is the Warrior monster. They just short-distance attack to the player or the defense object according to the Monster AI.

1. The way of collecting the log of input variables in game

Many training data are needed to train the neural network. So, the input variables in game collected like this :

The fprintf function is used to make a training data file.

// Input Log

For ( all monsters)

{

fprintf(“{ %lf, %lf, %lf, %lf, %lf, %lf, ”,

normalize(monsterHP),

normalize(characterHP),

normalize(DefenseObjectHP),

normalize(DistancetoCharacter),

normalize(DistancetoDefenseObject),

normalize(the number of friend monsters));

// Output Log

fprtinf(“%lf, %lf, %lf ),\r\n“,

AttacktoCharacter(InputData),

AttacktoDefense(InputData),

Flee(InputData));

}

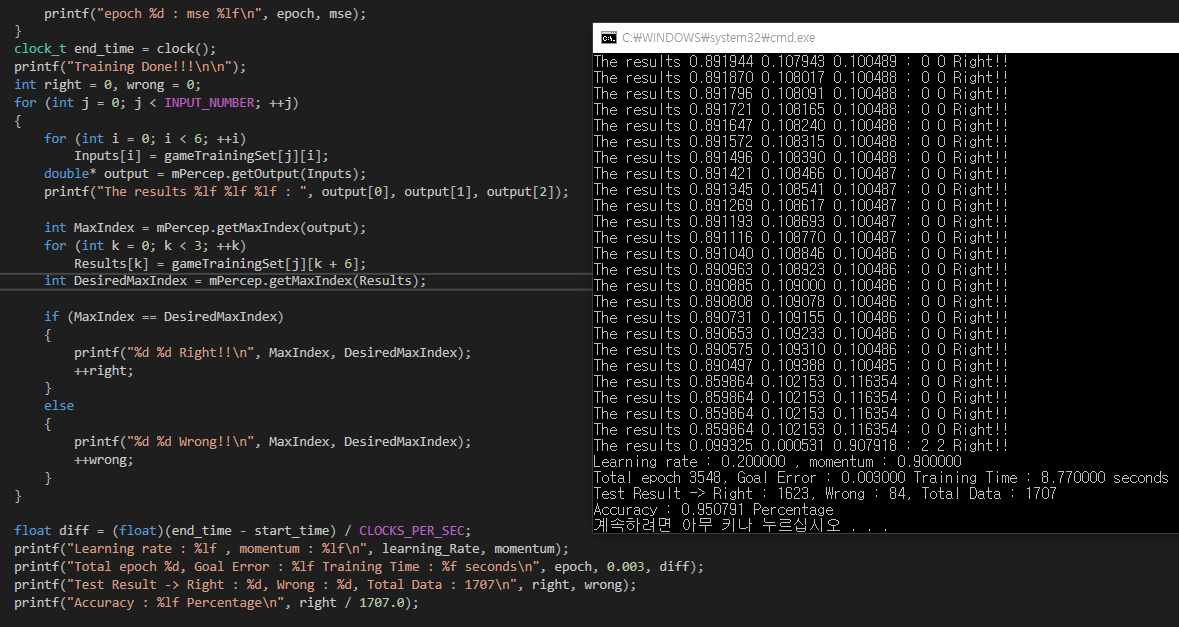
The input data is normalized between 0 and 1. And then, the output training data is collected using the InputData. 1707 of training Data is collected using this way.

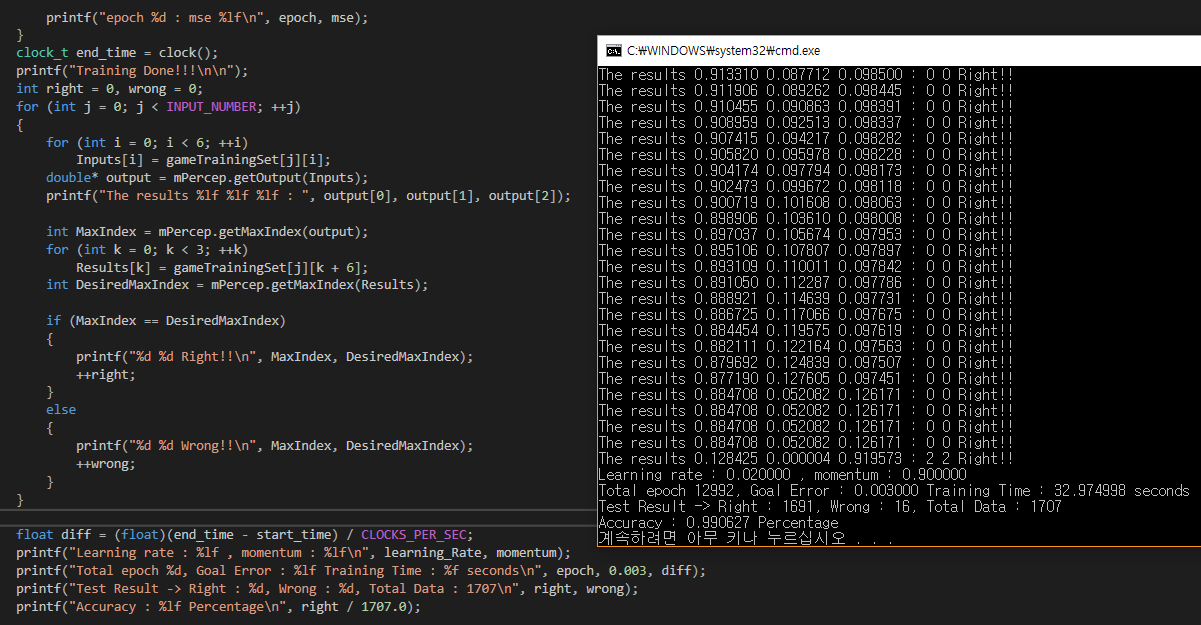
The example of the log is :

{ 0.400000, 0.200000, 0.500000, 0.042286, 0.002286, 0.571429, 0.9, 0.1, 0.1 },

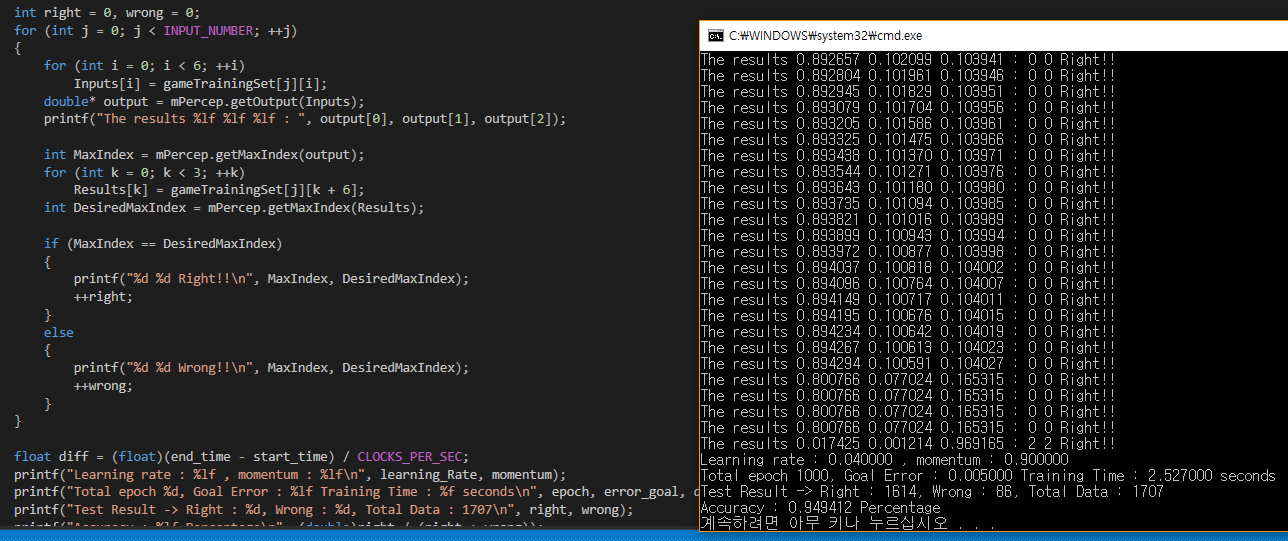
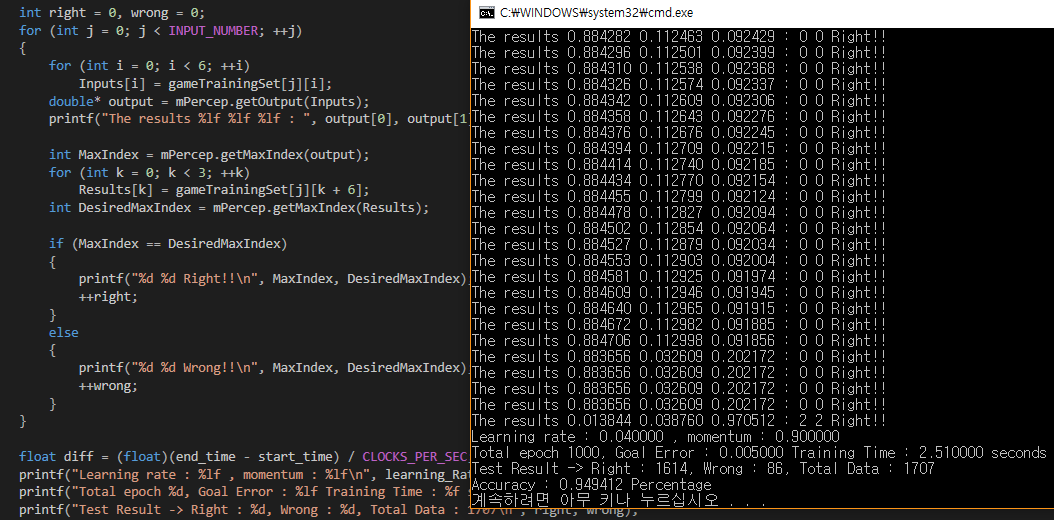
1. Testing of the multi-layer perceptron with the collected training data set

* Without the limit of epoch





* With the limit of epoch



1. The application of noise for input to adjust the accuracies of monster AI on each stage

void cMLP::setInputvalue(double\* value, double noise)

{

double real\_noise = (1.0 - noise);

if (noise >= 1.0) real\_noise = 0.0000001;

for (int i = 0; i < Ninput; ++i)

{

value[i] \*= real\_noise;

}

memcpy(inputValue, value, Ninput \* sizeof(double));

}

1. The code to pass monster information and get the decision from neural network

inputAdditive temp;

temp.monsterHP = (double)monsterHP / max\_HP;

temp.monsterXPos = rMonsterCollider[DEFAULT\_MOVE\_COLLIDER].x;

AIDecision = SCore::getMap()->MonsterANNDecision(temp);

if (AIDecision == 0)

printf("Stage %d, Warrior Monster %d, Decision : Attack Character\n", SCore::getMap()->getNumberCurrentLevel() + 1, id);

else if (AIDecision == 1)

printf("Stage %d, Warrior Monster %d, Decision : Attack Defense Object\n", SCore::getMap()->getNumberCurrentLevel() + 1, id);

else if (AIDecision == 2)

printf("Stage %d, Warrior Monster %d, Decision : Flee\n", SCore::getMap()->getNumberCurrentLevel() + 1, id);

1. Fixed code to make a decision in two seconds

if(SDL\_GetTicks() - iAITime >= 2000)

{

inputAdditive temp;

temp.monsterHP = (double)monsterHP / max\_HP;

temp.monsterXPos = rMonsterCollider[DEFAULT\_MOVE\_COLLIDER].x;

AIDecision = SCore::getMap()->MonsterANNDecision(temp);

if (AIDecision == 0)

printf("Stage %d, Warrior Monster %d, Decision : Attack Character\n", SCore::getMap()->getNumberCurrentLevel() + 1, id);

else if (AIDecision == 1)

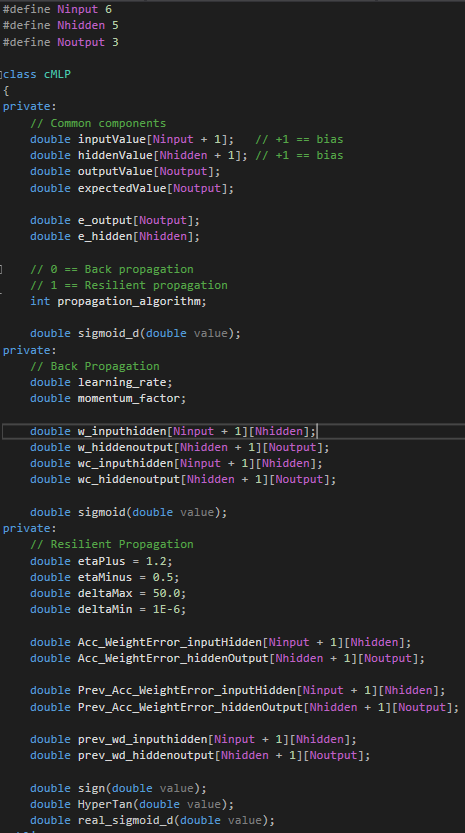
printf("Stage %d, Warrior Monster %d, Decision : Attack Defense Object\n", SCore::getMap()->getNumberCurrentLevel() + 1, id);

else if (AIDecision == 2)

printf("Stage %d, Warrior Monster %d, Decision : Flee\n", SCore::getMap()->getNumberCurrentLevel() + 1, id);

}

1. The structure of Multi-Layer Perceptron class



1. The structure of training the neural network with resilient propagation

/\* Training Zone \*/

double mse = 999;

int epoch = 0;

while (mse > m\_errorRate && epoch < m\_epochLimit)

{

mse = 0;

m\_NeuralNetwork.zeroOut();

for (int i = 0; i < INPUT\_NUMBER; ++i)

{

for (int j = 0; j < Ninput; ++j)

Inputs[j] = gameTrainingSet[i][j];

for (int k = 0; k < Noutput; ++k)

Results[k] = gameTrainingSet[i][k + 6];

m\_NeuralNetwork.setInputvalue(Inputs);

m\_NeuralNetwork.setExpectedvalue(Results);

m\_NeuralNetwork.Rprop\_feedforward();

m\_NeuralNetwork.Rprop\_calculateErrors();

mse += m\_NeuralNetwork.getErrors();

// MonsterAI.showStates();

}

m\_NeuralNetwork.Rprop\_updateWeights();

mse /= INPUT\_NUMBER;

++epoch;

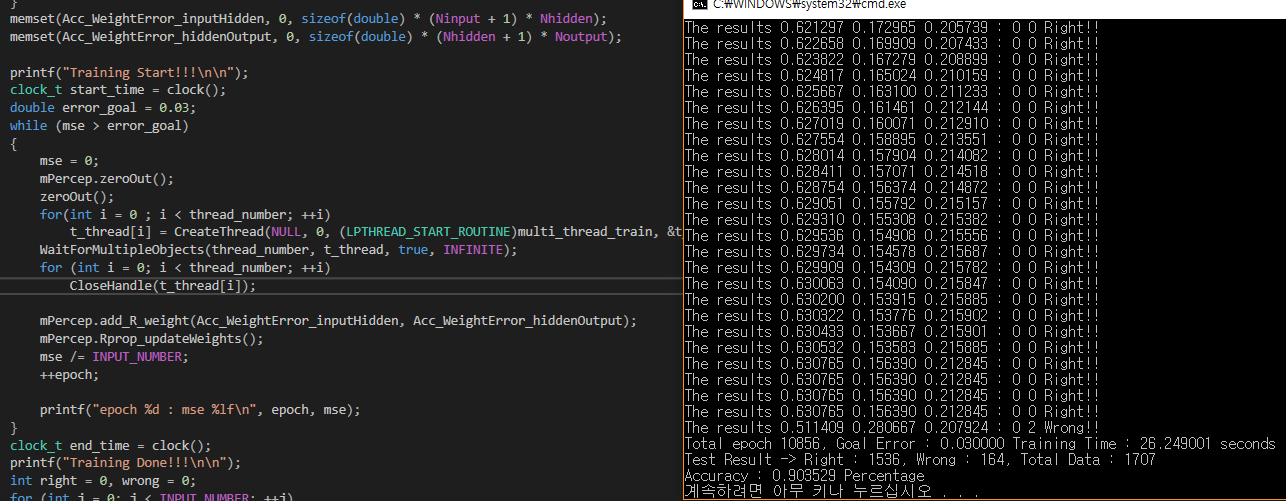
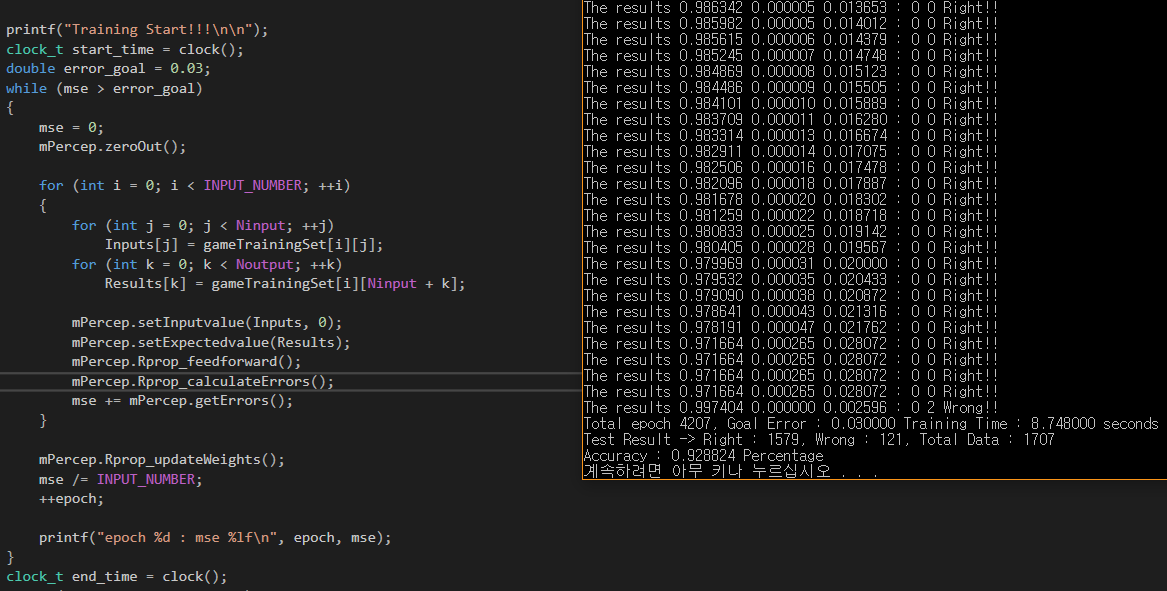
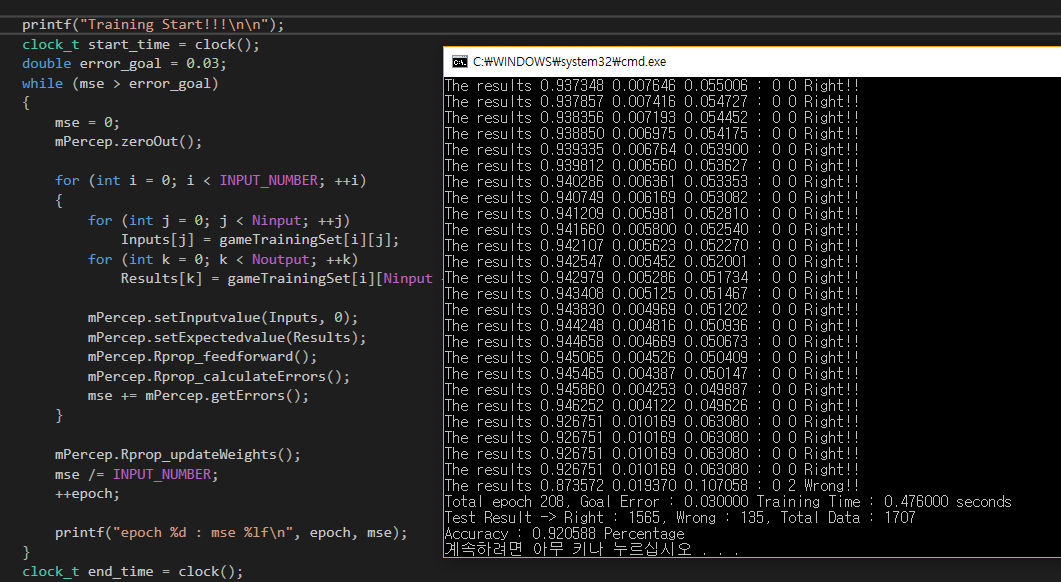
printf("epoch %d : mse %lf\n", epoch, mse);

}

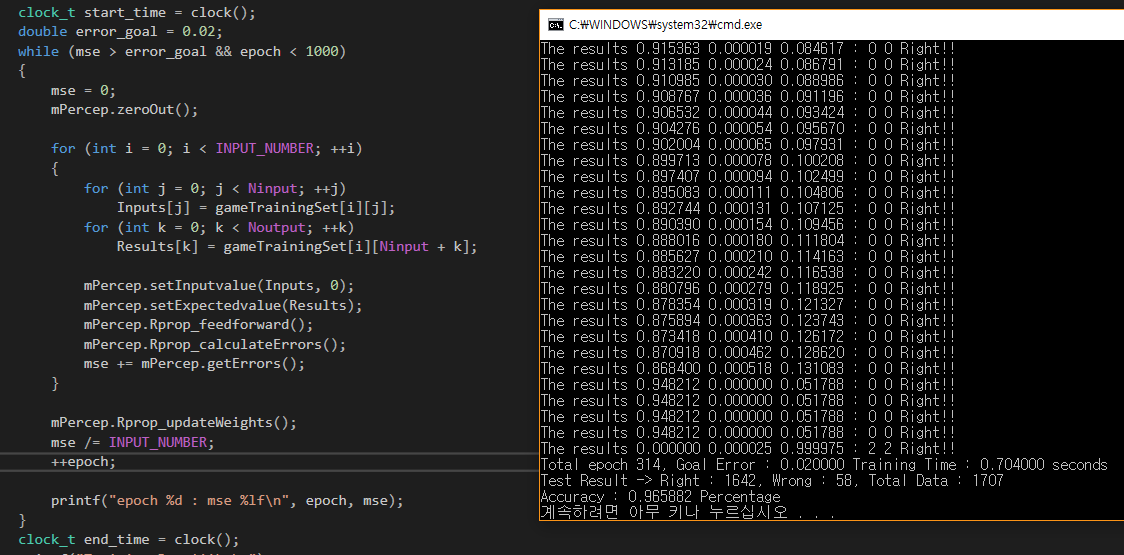
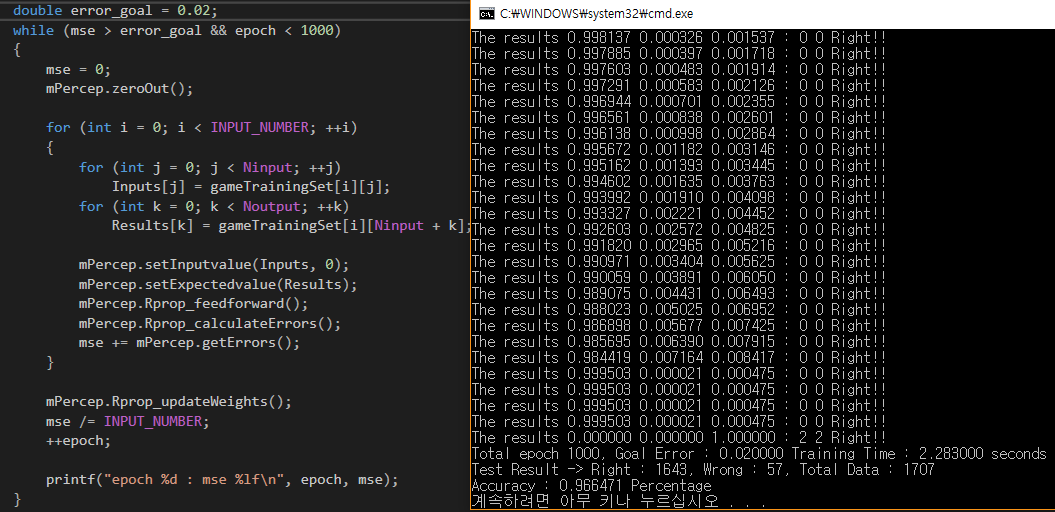
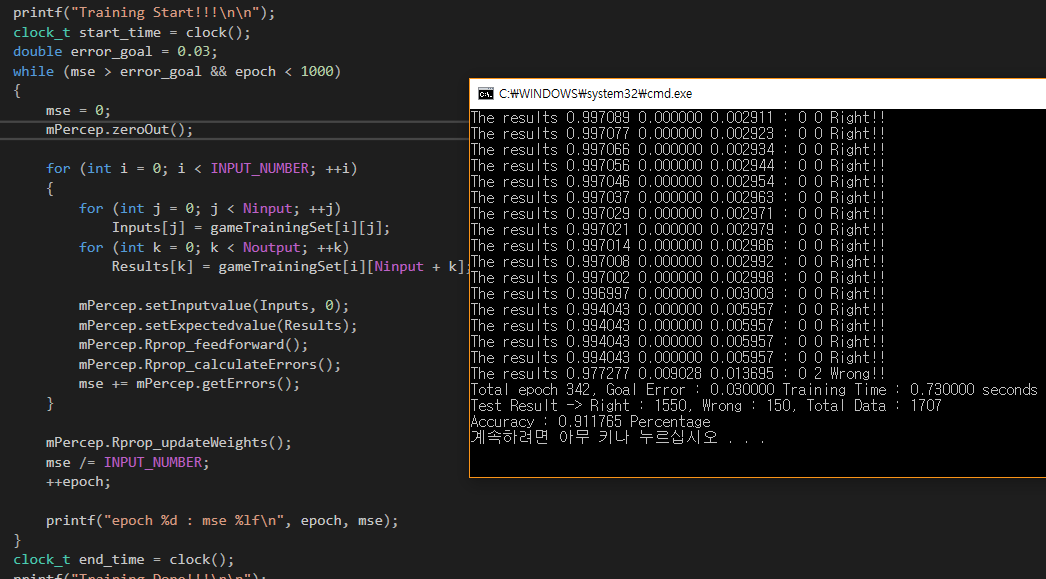
/\* Training Zone \*/

1. The result of the test with Rprop

* Training with Rprop without the limit of epoch



* Training with Rprop which has the limit of epoch, 1000

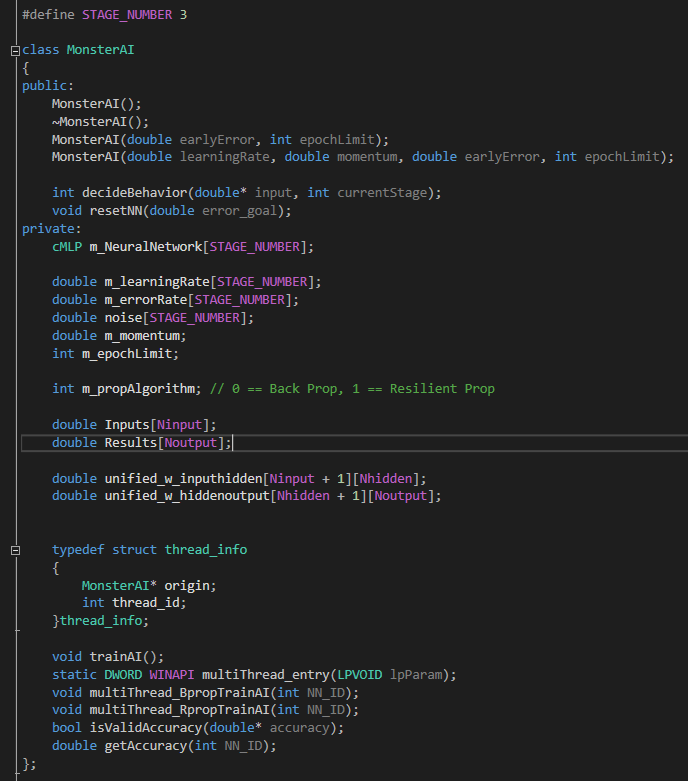


1. The comparison between Bprop and Rprop

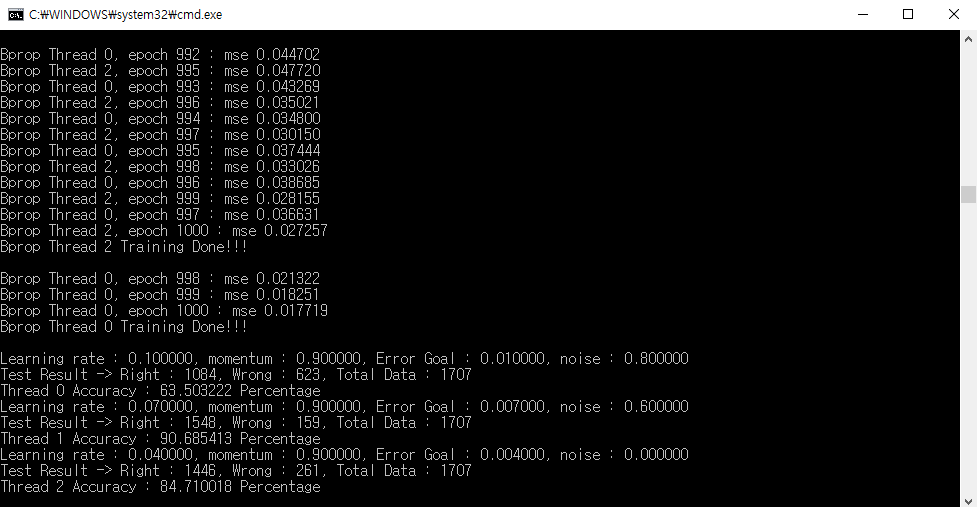
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Back Propagation | | | | | | | |
| Test Case | Learning Rate | Momentum | The goal of Error | Limit of Epoch | Epoch | Time | Accuracy |
| 1 | 0.2 | 0.9 | 0.003 | - | 3548 | 8.77 sec | 95% |
| 2 | 0.02 | 0.9 | 0.003 | - | 12992 | 32.97 sec | 99% |
| 3 | 0.04 | 0.9 | 0.04 | 1000 | 1000 | 2.51 sec | 94% |
| 4 | 0.04 | 0.9 | 0.04 | 1000 | 1000 | 2.527 sec | 94% |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Resilient Propagation | | | | | |
| Test Case | The goal of Error | Limit of Epoch | Epoch | Time | Accuracy |
| 1 | 0.03 | - | 208 | 0.476 sec | 92% |
| 2 | 0.03 | - | 4207 | 8.748 sec | 92% |
| 3 | 0.03 | - | 10856 | 26.249 sec | 90% |
| 4 | 0.03 | 1000 | 342 | 0.73 sec | 91% |
| 5 | 0.02 | 1000 | 1000 | 2.283 sec | 96% |
| 6 | 0.02 | 1000 | 314 | 0.704 sec | 96% |

1. The structure of Monster AI



1. Invalid Accuracies



1. The code to retrain the neural network by checking the invalid accuracies

while(1)

{

for (int i = 0; i < STAGE\_NUMBER; ++i)

{

t\_Thread[i] = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)multiThread\_entry, &temp[i], 0, &threadID);

}

WaitForMultipleObjects(STAGE\_NUMBER, t\_Thread, true, INFINITE);

for (int i = 0; i < STAGE\_NUMBER; ++i)

{

CloseHandle(t\_Thread[i]);

}

double comp\_accuracy[STAGE\_NUMBER];

for (int i = 0; i < STAGE\_NUMBER; ++i)

comp\_accuracy[i] = getAccuracy(i);

if (isValidAccuracy(comp\_accuracy))

{

goto trainingend;

}

else

{

printf("\nThe Accuracies are invalid to use for Monster AI\n");

printf("Retraining will start in 3 sec\n");

unsigned int itimePassed = SDL\_GetTicks();

int second = 0;

while(1)

{

if (SDL\_GetTicks() - itimePassed > 1000)

{

itimePassed = SDL\_GetTicks();

++second;

printf("%d second passed\n", second);

}

if (second == 3)

break;

}

printf("Retraining Start\n\n");

if (m\_propAlgorithm == 0)

{

for (int i = 0; i < STAGE\_NUMBER; ++i)

m\_NeuralNetwork[i] = cMLP(m\_learningRate[i], m\_momentum);

}

else if (m\_propAlgorithm == 1)

{

for (int i = 0; i < STAGE\_NUMBER; ++i)

m\_NeuralNetwork[i] = cMLP(1);

}

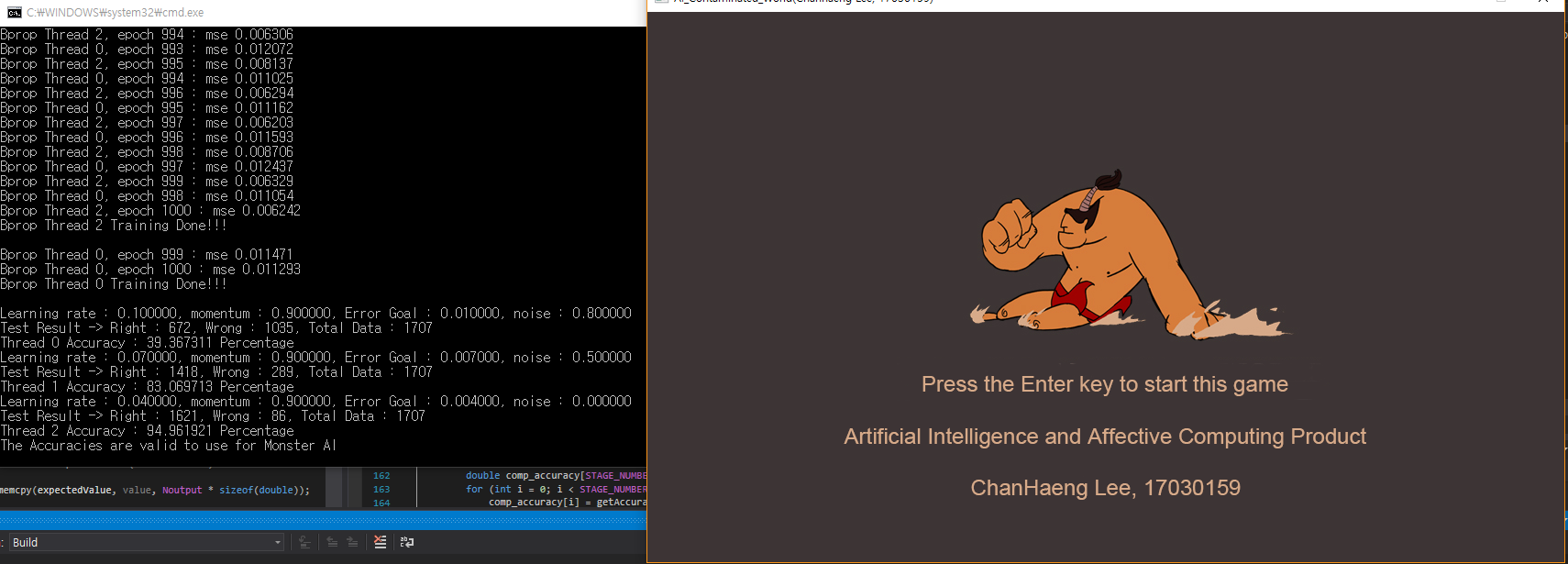
}

}

trainingend:

printf("The Accuracies are valid to use for Monster AI\n");

1. The valid accuracies of neural networks



1. Reference
2. Riedmiller, M (1994) Rprop - Description and Implementation Detatils. Available at: http://www.inf.fu-berlin.de/lehre/WS06/Musterererkennung/Paper/rprop.pdf (Accessed: 2 January 2018).
3. McCaffrey, J (2015) Visual Studio Magazine. Available at: https://visualstudiomagazine.com/articles/2015/03/01/resilient-back-propagation.aspx (Accessed: 2 January 2018).
4. Heaton, J (2009) Heaton Research. Available at : <http://www.heatonresearch.com/encog/mprop/compare.html> (Accessed: 2 January 2018).