Honglu Xu
EECS 491
Final Projest

Apply Artificial Neural Networks on AIs to Play Video Games.

Same part with EECS 531:

Note: This is a project that cannot all implemented in the Jupyter Notebook, and it is too hard to separate the code and copy them into the notebook, so there will not be any code showing in this notebook. All the file-names and descriptions are listed below, and I will try to explain the code part as clear as possible.
Please make sure you have correctly the configured your envirionment before you run the code. The main packages that will be needed are: "pygame", "numpy", "pgmpy", "keras".

Files:

1. Snake.py: Main game file. Contains all the modules needed to run the game.
2. VisionNN.py: Vision part of nerual networks. AI model for get input data from the screen pixels.
3. InputAgent.py: Auto input agent for naive logic game input generations. Will be used to train the logic AI.
4. LogicNN.py: Logic part of nerual networks. Will be used to generate output (player input) through the ANN.

About the game

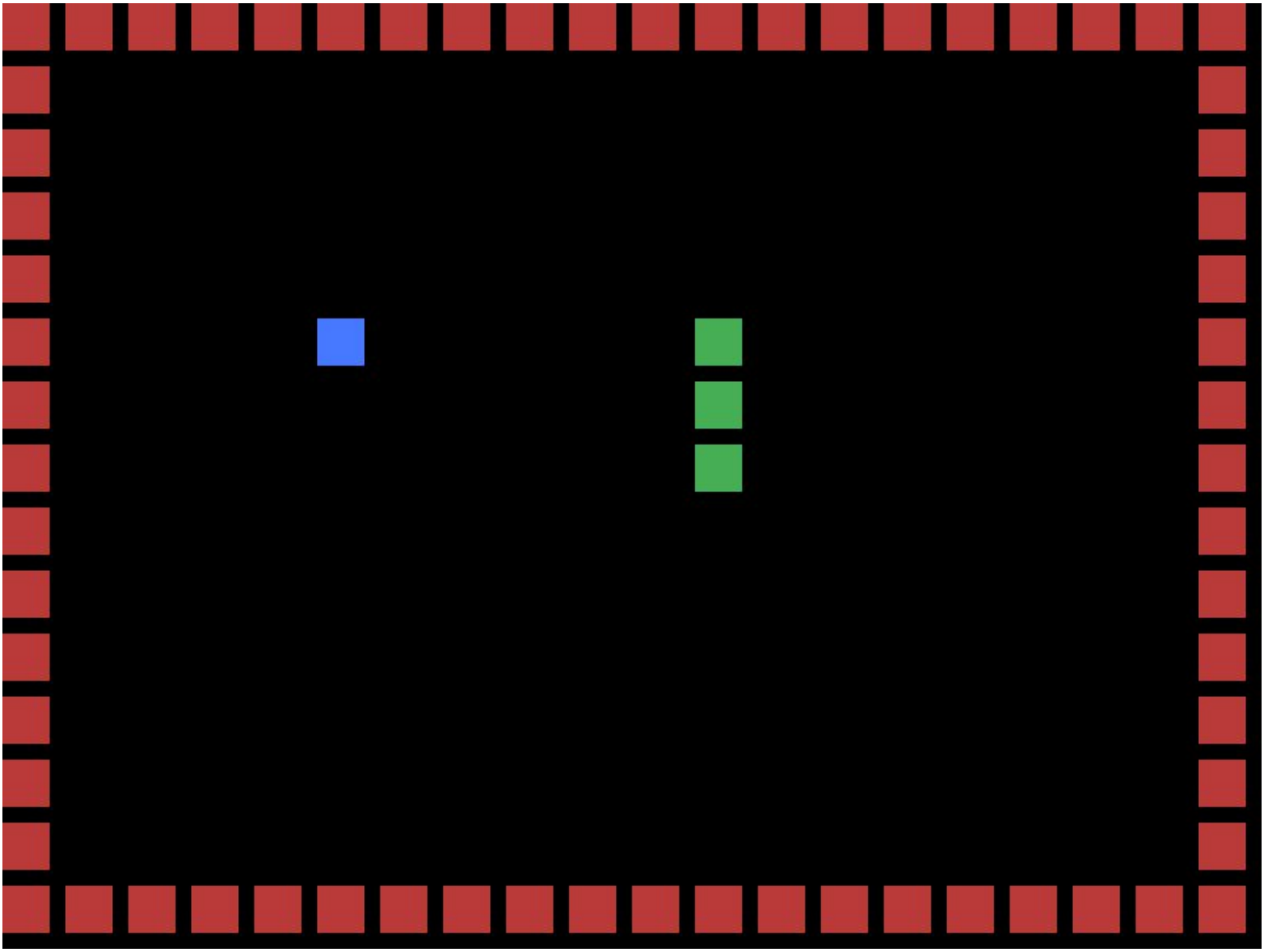
To build a game video AI, we firstly need to build a game. In this project, I will use the package "pygame" as the game engine.
The main game logic is simple. As long as we are running the game, it will:

1. Check all the input I/O from the player/system.
2. Update the game, which will apply all the inputs on the game and calculate all the data for next frame.
3. Render the next frame.
4. Keep looping from step 1 until end.

The game I will use for my experiment is "Snake". I chooese it for several reasons.

1. It is easy to implement.
   The snake is sort of an simple game. We might have encountered this game many times, we are familiar with it, and since it is in 2D and block based, we do not have many rendering part.
2. Game logic is simple enough for a simple AI.
   The game logic for snake is simple, for the player part, the input should be in 5 statement: "up key", "down key", "left key", "right key" and the "r key" for restart the game. And the output (feedback) should also be simple: survived (continue the game), ate the apple, and died. Thus, we have 5 input and 3 feedback statement, which should be simple enough for a starter experment.
3. Game logic can also be complicated.
   The game can also be complicated if we want to behave very good on this game. Sometimes we surrounded ourselves as a circle, then that area is a "trap", we should avoid our snake goes in. Sometimes we need to go through a narrow path that seems dangerours. Sometimes we need to count the steps so we can survive just one step early. Thus, it also can be a complicated game.

This is the original game scree capture. I used the code from the tutorial of Python official site [1].



Game Tags:
If you want to try the code, you might need to configure the game tags that in "Snake.py" so that it will work as you wanted. Here is the list and descriptions of the game tags.

1. using_AI: Main tag that will enable or disable all the logic AI.
2. AI_type: Logic part AI type.
   random: Randomly generating inputs.
   bys1: Decision trees with believe network.
   beta: Alpha-beta algorithms.
   bys1nbeta: using both bys1 and beta
   NN1: Neural network model1.
   NN2: Neural network model2.
3. beta_step: Step configuration for alpha-beta.
4. train_Model: Logic models that will be used for taining.
5. critical_train: Train logic models only in the critical condition.
6. reading_Model: Reading logic models from file (file name hardcoded).
7. test_Model: Logic models that will be used for testing.
8. using_VS: Main tag that will enable or disable all the vision AI.
9. Vision_type: Vision part AI type.
   NN1: Neural network model1.
10. train_VModel: Vision models that will be used for taining.
11. critical_vtrain: Train vision models only in the critical condition.
12. reading_VModel: Reading vision models from file (file name hardcoded).
13. test_VModel: Vision models that will be used for testing.

For default, you can disable all the tags, then enable it one by one to see if it works.

For EECS491

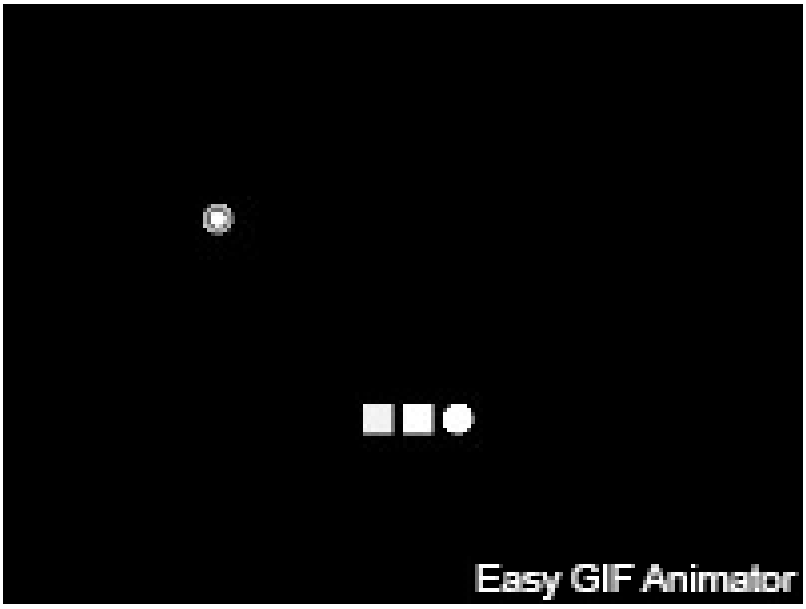
As stated in the EECS531 FP, I changed the color of the game to grayscale, so the components can be easier to detect.

1. First approach: Game agent.

This model will be shown as the name of "random" for the logic model.

To give the AI the power to control the game, we need let the AI access the game control system. Fortunatly, we build the game ourselves, so we can left some APIs to the AI as it can control the game.

To test the game agent, we can use the simplest kind of agent ---- random generation agent, to test is it's work able. Basiclly, this will generate 4 values, which stands for move-up, move-down, move-left and move-right, which are 4 basic inputs for the snake game.



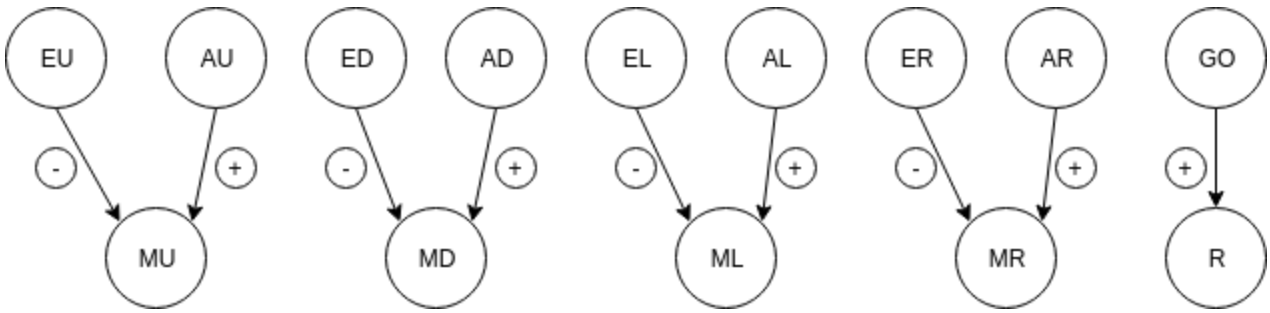
We can see from the above animation that we can not control the snake with inputs that generated by the agent, and since it is automatically generated, we can use that to train the neural network we will build later.

2. Decision tree.

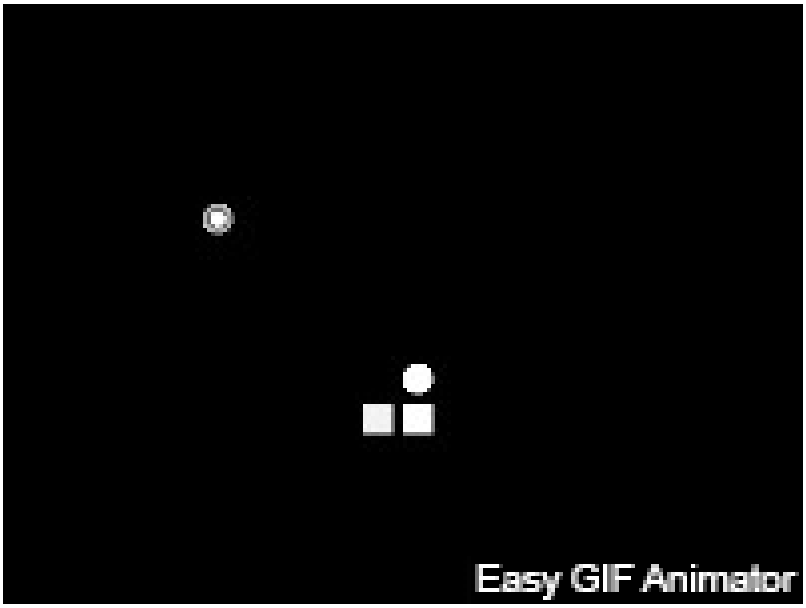
This model will be shown as "bys1" as the name of logic model.

One of the most common way to build an AI for the game is use the decision tree. It works similar as the state machine, and it always did good on some simple logics.

Here, I used the bayesian network package "pgmpy.models.BayesianModel" to bulid a network as the decision tree, so we can use weights on it and see what will happen.



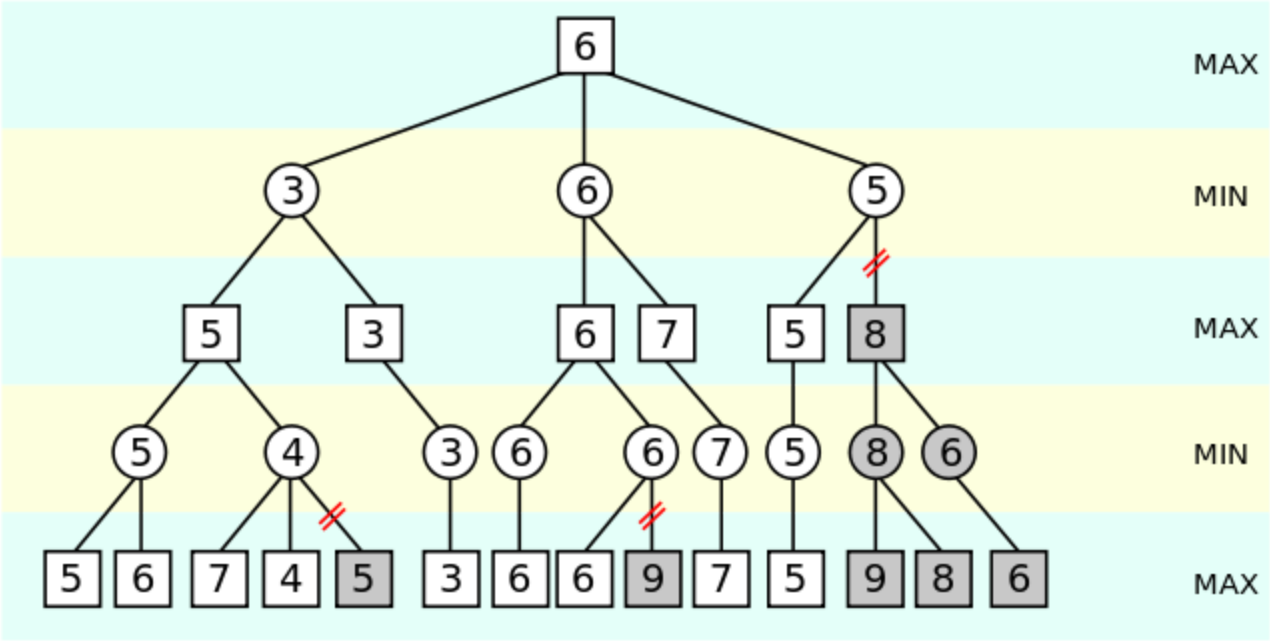
Above is the decision tree that I made for the game. We have these variables:
EU: Enemy is above within one block.
ED: Enemy is down within one block.
EL: Enemy is left within one block.
ER: Enemy is Right within one block.
AU: Apple is up generally.
AD: Apple is down generally.
AL: Apple is left generally.
AR: Apple is right generally.
GO: Game Over.
MU: To move up.
MD: To move down.
ML: To move left.
MR: To move right.
R: Press 'R' to restart the game.
If the sign is '-', I will give nagative weights on this edge.
If the sign is '+', I will give positive weights on this edge.
Use the first tree as an example. If the enemy is up within one block, we definatly won't move up. But if there is an apple in that direction, we might try to move up.
Here, the enemy means all the elements that will make the snake die, which will contain, all the wall blocks, and all the tail blocks.



We can see that this is already a not bad AI. It knows to get the apple, and it knows to avoid critical damage for 1 step. However, since the judgement of the enemy is just for 1 step, it cannot predict whether it is going into a "trap" or surround itselves that can never get out. Thus, more work will be needed for a better AI.

3. Alpha-beta algorithm.

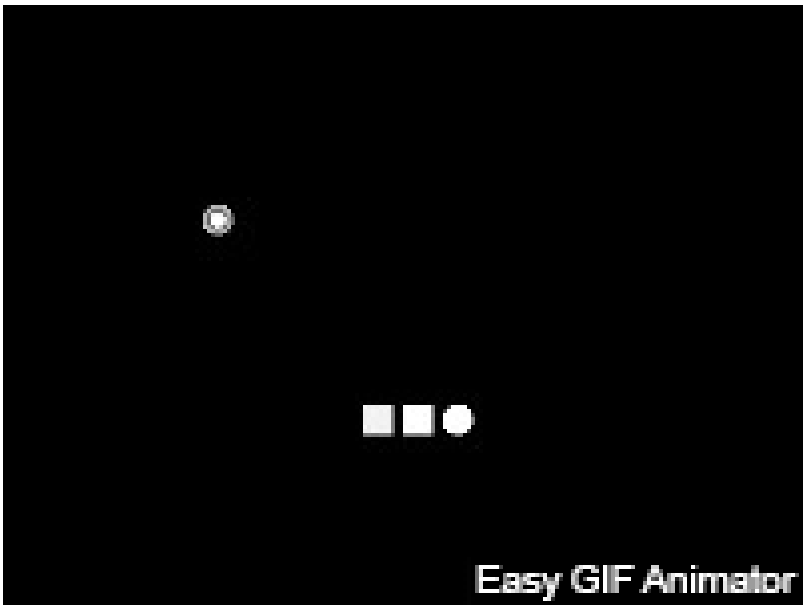
This model will be presented as name of "beta" in the logic model.



[2] Wikipedia

The a-b algorithm is mostly used for steps predicting for many situations. By using this algorithm, the AI can generate all the situations for next few steps, marks them as scores, and decied which step will lead to the trail taht have most winning possibilitiy. A common usage is an AI in the chess game, which it can generate all the steps that pissibile based on the current state.

For this situation we don't have enemies that will take steps to move, so all the situation is sigle sided. We only need to maximize the snake's steps, and it will be good enough.



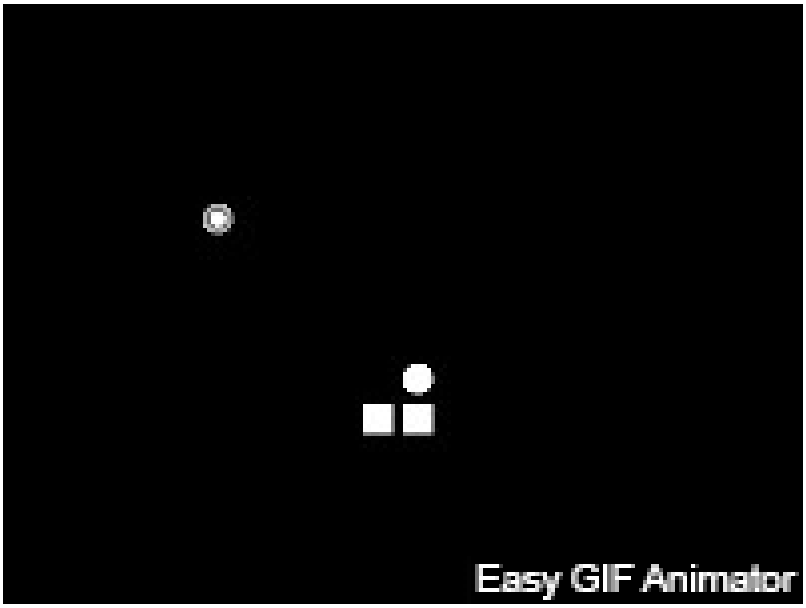
We see this time, the AI is much more careful right now. Since we set the prediction step as 5. It can only predict next 5 steps. It will efficiently avoiding some dangerages "traps", but if the apple is not reachable for 5 steps, it will not know where to go. Also, if the AI think it is very dangerages to take the apple, it rather not taking it. That's why at the at end, it is looping all the time

4. Combine decision tree and Alpha-beta algorithm

This model will be presented as name of "bys1nbeta" in logic model.

Sicne we know the pros and cons for these two algorithms now. We could combine these two method and extract all the advantages from both of them.

Since the decision tree will lead the main direction to the apple, and the a-b algorithm will avoid cirtical death for next few steps. We could use the decision tree as the main guide, and if the a-b algorirhm detected some bad things, we could avoid it.



We see that now, it not only know where to go, it also trying to be very careful not went into any traps that can be detected within 5 steps.

5. Using ANN

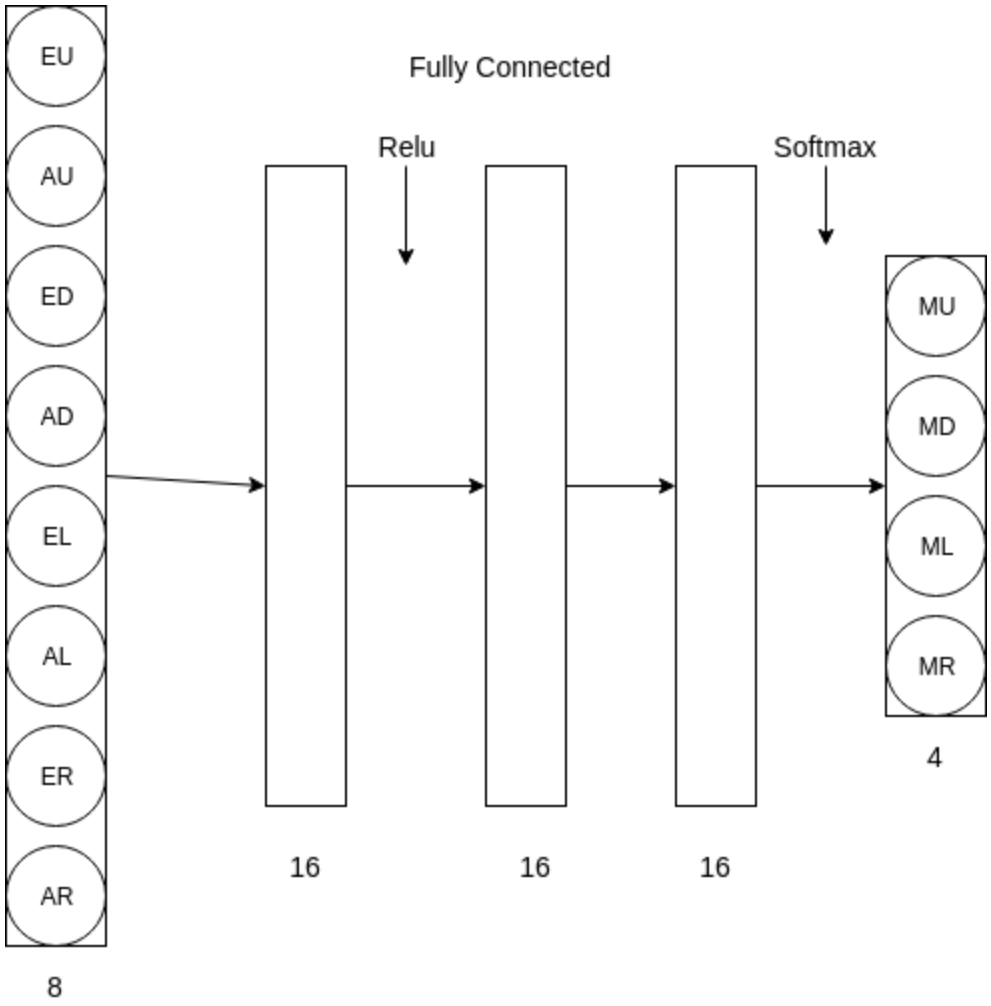
This model will be presented as "NN1" and "NN2" in the logic model.

We have a sort of decent AI for the snake game, but it is far enough from our goal. What we want ot built, is that a genetic AI that can be used on many different games. Which should have the abilities of:

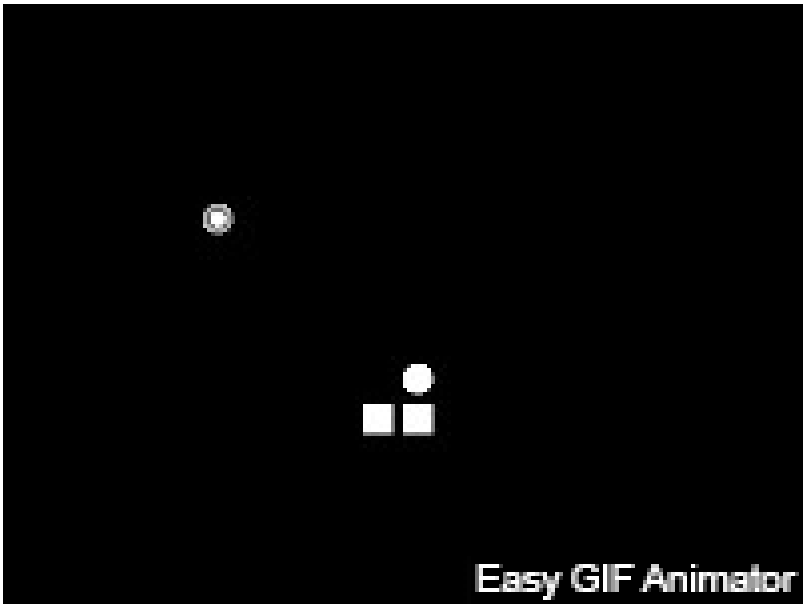
1. Not based on any internal game data.
2. Should be able to suit many different types of game input logic, should not be restricted to 'EU', 'ED', 'AP' these things.
3. Should be able to suit many different types of game output logic, not only we could move around, for some games we should be able to 'jump', or even fire weapon.

That's why the goal for this project is to use the ANN. That means we train the AI with a certain rules and feedbacks (getting hurt, getting score), it could obey the rules and persuit the positive feedbacks. A lot of possible solutions I have come up with, such as using RNN and time based modules. Unfortunately, I am running out of time for this project. Thus, I will try to implement some simple solutions at least.

The simple way is to use the decision tree to train the neural networks. The decision tree is pretty straight foward, so the AI should be able to understand the logic easily.

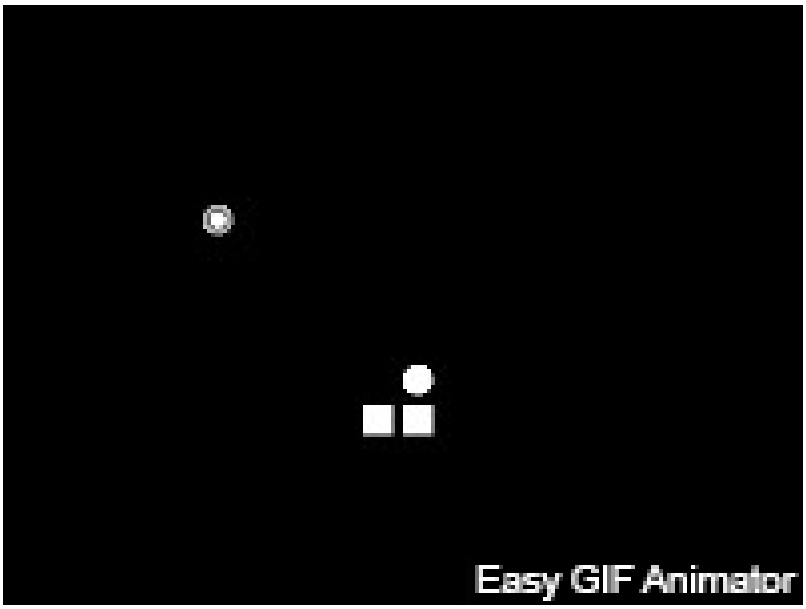


This is the neural network I used for the simple approach. Since we will use the decision tree to train the neural network, it will make sense that we have the same inputs and outputs for this networks.



We see that indeed, it is following the rules of the decision tree. We can confirm at this is working right now.

Also, since the decision tree is such a simple logic, which is generated by only 1 edge layer. i want to see whether it is possible to use only one inner layer of neural network to learn this basic idea. I reduced the network above to only one inner layer with 16 nodes, and the result shows below.



Though in theory, it should work, but actually it seems not well trained. There are many basic mistakes that should not occur, but happened on this AI.

Future Works

This is only a simple approach for the AI. There are still a lot to do next. To train the AI with a-b algorithm, since it is steps based, we should use the RNN instead of Dense. And similar to the combination of these methods, we could combine these two neural networks together to see what will happen. Also, to generate more intelligent data, we could use our onw plays as the input data to train the AI. Finally, we should get rid of all the internal game logics, and use the CV module as the input, the feedback (whether it survives, whether it hit the wall, whether it ate the apple) as the regression, and built a genetic AI that can be used for many different video games.

Reference

[1] Python Official Site, *Snake with Pygame*, https://pythonspot.com/snake-with-pygame/ (https://pythonspot.com/snake-with-pygame/)
[2] Wikipedia, *Alpha–beta pruning*, https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning (https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning)