NARS-FighterPlane

A game AI based on NARS

Boyang Xu

Spring, 2021

1 Introduction

NARS-FighterPlane is a game in which NARS serves as an AI controlling a fighter plane to hit enemy planes. In the game, NARS begins with zero experience and babbles in the game environment. When interacting with the game environment, NARS needs to learn how to control the fighter plane and hit enemy planes to gain a higher score. The source code can be downloaded on github[1].

Non-Axiomatic Reasoning System (NARS) [2], is a general-purpose AI system towards Artificial General Intelligence (AGI) which aims at the building of a "thinking machine" – a computer system with human-like general intelligence [3]. NARS can work and learn in real-time in the open environment, under the Assumption of Insufficient Knowledge and Resources (AIKR) as its working definition of intelligence.

NARS as a developing theoretical model has its open-source implementation in Java, OpenNARS [4], which is sometimes referred to as "OpenNARS for Research" to distinguish from "OpenNARS for Applications" (usually referred to as ONA) [5], which is implemented in C by Patrick Hammer and its control mechanism is totally different from OpenNARS. In this project, OpenNARS 3.0.4 and ONA are both used to control the fighter plane so they can be compared. For simplicity, NARS will be used in the following to refer to OpenNARS 3.0.4 and ONA.

2 Program Design

The program can be divided into two parts, the game part that is programmed in Python by using pygame module, and the NARS part that uses either OpenNARS or ONA implementation. The game is running in the main thread while NARS is running as a subprocess in a command window and they communicate with each other asynchronously. The general idea is that the game will give NARS the objects' positions (i.e., the sensor information) at set intervals, and then let NARS infer. There is a thread monitoring NARS in the background

and once NARS gives an operation (i.e., the motor information) the thread reads the operation and passes it to the game to execute. The overall process is shown in Figure 1.

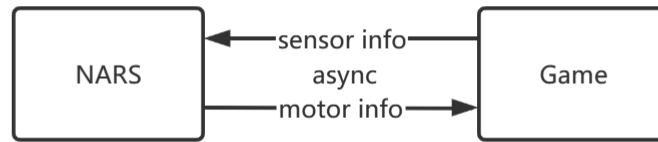

Figure 1: NARS-Game

2.1 Game

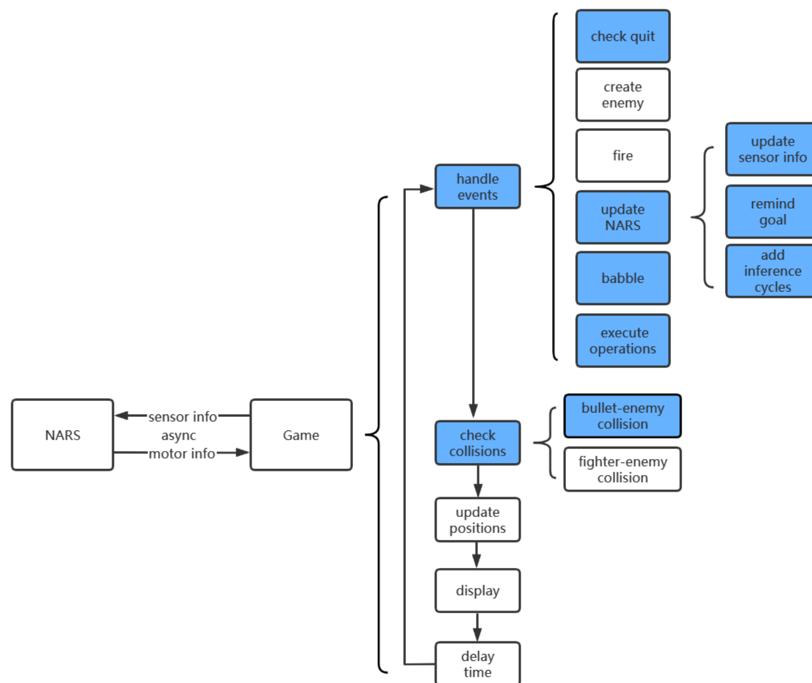

Figure 2: Main loop

As Figure 2 shows, the main logic of the game is a loop executed once per frame. In each frame, the game keeps doing five things: handle events, check collisions, update positions, display, and delay time. The boxes painted in blue indicate that these function modules will communicate with NARS. The most

important part is the module of handling events, which performs the following tasks:

1. Check if the game is quit. Once the game is closed, it will call the termination of NARS and then the game.
2. Create an enemy plane per second.
3. Fire per 0.5 second.
4. Update NARS. The game will pass the position of the fighter plane and the enemy to NARS (sensor info). However, NARS do not use their coordinates directly. The reason will be further discussed in the section of Lessons Learned. What passes to NARS is the enemy's positions relative to the fighter plane, i.e., enemy on the left, enemy on the right, and enemy in the face of the fighter plane. Then, NARS will be set a goal of hitting enemy. Since the event happens at set intervals, the goal is reminded repeatedly. Finally, add certain inference cycles to NARS.
5. Babble event will be triggered only for OpenNARS and last for a period of time at the beginning. After that, this event type will be removed. For ONA, the babbling process is innate.
6. Read the current operations in the "mind" of NARS and execute them.

Events 1-5 will be triggered once in a while according to their timers.

In the "check collisions" module, the game checks if two sprites collide. If a bullet collides with an enemy plane, it will give NARS a reward (i.e., satisfy the goal).

"Update positions" module updates all the sprites' current positions and check if they are out of bounds. It also updates the text shown on the surface, like scores, time, FPS, etc. After that, the "display" module will show them.

The module of delaying time is used to complete one frame. It will first calculate how many milliseconds have passed since its previous call, including the time spent by other modules. Then, it calculates how much time it needs to wait and finally delays the time.

2.2 NARS

2.2.1 Preparation

Before using OpenNARS or ONA for this project, operations need to be registered first. OpenNARS 3.0.4 has already prepared some operations for testing, such as ˆ*left*, ˆ*right*, ˆ*strike*, ˆ*open*, and so on. Here ˆ denotes an operation in Narsese (an internal language in NARS). We can pick ˆ*left* and ˆ*right* for the game and the babbling process will activate them from all of these operations. To add or delete operations in OpenNARS, someone can modify the code related to operations in *defaultConfig.xml* which is located at /src/main/recourses/config in the source code and then recreate a jar file.

For ONA, though the code in the github [5] already contains some operations, the operations we do not want to use have to be removed, otherwise all operations will be invoked during babbling as the babbling process is innate in ONA and we cannot modify it. To modify operations, check *Shell.c* and *Config.h* located at /src. Register the operations we want and remove the operations we do not want in *Shell.c*, and set OPERATIONS_MAX in *Config.h* to the number of operations we have. Finally, compile ONA through *build.sh* by cygwin (gcc-g++ needs to be installed) in Windows.

The program will launch OpenNARS by inputting "java -Xmx1024m -jar opennars.jar" to the command window or launch ONA by inputting "NAR shell". Then, for both of them, "*volume=0" should be set to NARS, which mutes NARS except executable operations and answers to questions, or else NARS will show almost everything in its mind to the output and heavily slow down the program because of I/O. There is a thread monitoring the command window all the time, reading and parsing every sentence spoken out by NARS to sift out an executable operation.

2.2.2 Learning Mechanism

NARS-FighterPlane mainly utilizes the logic of NAL-7 (temporal inference, including Events) and NAL-8 (procedural inference, including Goals and Operations) [2]. As the "update NARS" module in Figure 2 shows, there are three steps: update sensor info, remind goals, and add inference cycles. From the sensor info, the program calculates the enemy's coordinates and derives positions relative to the fighter plane, and then writes them as Narsese to tell NARS the following events ("//" denotes comments):

```
1 //:|: denotes the sentence is true at present
2 <{enemy} --> [left]>. :|:  //Now an enemy is on the left
3 <{enemy} --> [right]>. :|: //Now an enemy is on the right
4 <{enemy} --> [ahead]>. :|: //Now an enemy is ahead
```

Then, a goal is set to NARS once in a while:

```
1 //! denotes the sentence is a goal.
2 //It can be read as "I want to feel good."
3 <{SELF} --> [good]>! :|:
```

Finally, add inference cycles to NARS by directly inputting a number, which helps NARS form its time sense. In the process of babbling, NARS will be forced to randomly execute operations in it,

```
1 ^left
2 ^right
3 ^deactivate
```

so the situation of the fighter plane will be changed accordingly and so as the sensor info which will be passed to NARS later as the consequences of its actions. Once "check collisions" module detects a bullet shoot down an enemy, NARS will instantly get feedback and be "praised", i.e., satisfying the goal:

```
1 <{SELF} --> [good]>. :|: //Now I am feeling good
```

By interacting with the game environment, NARS gradually gets some clues and forms expectations: e.g., if I move left, then the enemy on the left will be ahead of me; if the enemy is ahead, then the bullet will hit it so I can feel good, etc. It can also come up with wrong expectations. NARS uses backward inference to reduce the given initial goal into derived goals until the derived goals can be satisfied by executing operations. Then, NARS uses forward inference to achieve them one by one in turn until the initial goal is achieved. Thus, NARS can give operations from time to time while doing reasoning. By constantly updating and getting feedback, NARS gradually learns which expectations are more proper and reliable as for the current situation, which finally turns into its experience.

3 Demo Show

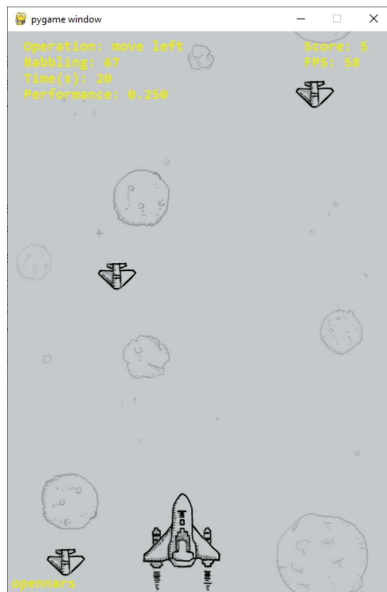

Figure 3: Demo 1

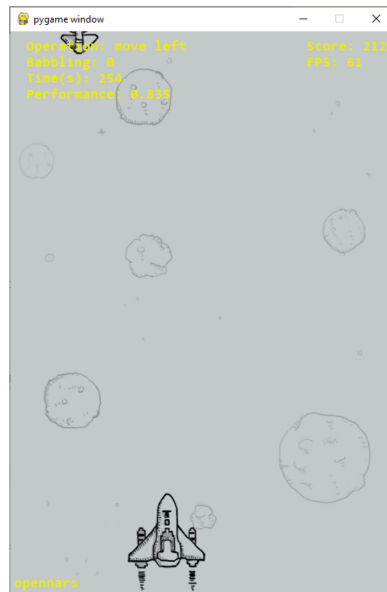

Figure 4: Demo2

- Operation: the current operation, "move left", "move right", or "stay still"
- Babbling: the remaining times of babbling, only for OpenNARS. When it decreases to zero, the babbling process is finished
- Time(s): the time (seconds) that has passed

- Score: the number of the enemy hit by bullets
- FPS: frame per second
- Performance: scores divided by time (s)
- opennars: denotes which type of implementation is called, OpenNARS or ONA

As Figure 3&4 shows, at first NARS behaves poorly and rarely shoots down an enemy plane. After a while, it can hit almost every enemy. We can see there is a clear improvement by comparing their performance index.

4 Learn Firing by Itself

In the above talking, the fighter plane fires automatically at set intervals. In fact, we can let NARS learn when to fire by itself, which will be an improvement. It has been implemented in NARS-FighterPlane v2.0. The principle is similar to the above. The difference is that we need to add an operation into NARS and recompile it, modify the babbling process for OpenNARS, and allow NARS to control the firing action of the fighter plane in the game. The performance of NARS-FighterPlane v2.0 is satisfying though it is not as good as NARS-FighterPlane due to the higher learning costs. Some NARS can finally learn when to fire and perform very well, while some NARS can hardly master the skill. It needs some luck.

5 Improvable Points

Though NARS shows its strength on learning ability, there are still a lot of things that could be improved as for the game:

1. Add punishment when the fighter plane collides with the enemy planes, which can make the game more interesting. Imagine a situation in which if NARS moves left it would hit an enemy and be praised but in the meanwhile it would collide with another enemy plane and be punished, how NARS will do?

2. Add more sensors to improve the intelligence of NARS. For example, we can allow NARS to sense the enemies' positions in the vertical direction.

3. Add more operations to enhance the capabilities of NARS and make the game more complicated, for example move upward and move downward. With point 2 and 3, NARS can have more options when coping with the situation mentioned in point 1.

4. The running performance of the game is limited by the asynchronous I/O between NARS and the game. In the project, I uses a thread monitoring

the comman window. Maybe the performance can be improved by other async I/O methods in Python, like asyncio or multiprocessing module. Or more radically, combine NARS and the game by implementing NARS in Python or move the game to Java or C, but it would be more difficult.

6 Compare with NARS-Pong

Many ideas of NARS-FighterPlane comes from NARS-Pong in Unity3D by Christian Hahm [6]. By reading NARS-Pong's code, I have learned a lot and I appreciate for the help from Christian and Patrick. Though the main design of NARS-FighterPlane is similar to NARS-Pong, there are still some differences:

1. The game part is implemented in Python and uses pygame module.
2. The game environment of NARS-FighterPlane is more complicated. NARS-Pong has only one object, the pong, while NARS-FighterPlane has many enemy planes. NARS can face a dilemma in which an enemy on the left and an enemy on the right happen at the same time. Besides, one more operation is added so that NARS can fire by itself in the NARS-FighterPlane v2.0.
3. The program has good extensibility. As discussed in the previous section, we can either add more sensors and operations to NARS or make the game mechanics more complex.

7 Lessons Learned

The hardest part lies in the overall design and how to use NARS in Python. Even though I was enlightened by NARS-Pong, it still took me a lot of time to realize the asynchronous communication between NARS and Python. This is a technical problem I encountered, but I would like to talk more about the sensors in NARS.

What sensor information should be passed to NARS? This is a big question. At the beginning I thought it can be simply done by passing the coordinates of every object to NARS, but later I found it is impossible and does not make any sense to NARS. The key reason is that NARS does not understand math and also lacks other background knowledge (concepts like intersection, overlapping, parallel, etc.), so NARS cannot utilize digits efficiently. Each coordinate to NARS is a concept, and there are hundreds of thousands of different coordinates. Performance could be a big issue if NARS does reasoning among such a huge concept network without knowing any math concepts. Perhaps NARS can learn some concepts in the end, I am not sure, but it is sure to take a huge amount of resources to learn. The most common conditions could be that NARS gets bogged down in number superstition.

To avoid this problem, we need to take advantage of our math knowledge: calculate the coordinates and pass the positions relative to the fighter plane to

NARS, which boils down to three Narsese sentences eventually. Then, NARS uses the ruminated information to do reasoning. Maybe in the future NARS could calculate and derive conclusions by itself after mastering math.

Another important thing is that if we want NARS to satisfy its goals through learning, it should be equipped with proper sensors and get correct feedback. Think that if NARS does not know the horizontal positions of the enemy planes, how can it learn to hit enemies by moving left or right? In fact, it requires us to first figure out the process, and the logic should make sense to our human. Only when we can explain the process can NARS achieve its goals. In this sense, the logic of NARS is similar to human and it is explainable.

References

[1] NARS-FighterPlane. `https://github.com/Noctis-Xu/NARS-FighterPlane`.

[2] Pei Wang. *Non-Axiomatic Logic: A Model of Intelligent Reasoning*. World Scientific, Singapore, 2013.

[3] AGI Society. `http://www.agi-society.org/`.

[4] OpenNARS. `https://github.com/opennars/opennars`.

[5] OpenNARS for Applications. `https://github.com/opennars/OpenNARS-for-Applications`.

[6] NARS-Pong in Unity3D. `https://github.com/ccrock4t/NARS-Pong`.