PySnake

Kaj Habegger & Fabio Lenherr

Miniproject

Table of Contents

1. Introduction

We decided to do a small reinforcement learning project.
As we have never done anything in terms of reinforcement learning before, we used this video as a starting point. It is a tutorial of creating a Snake game and design a reinforcement learning AI model which is able to play the game after some iterations.

2. Tooling

Like the in the tutorial linked above, we used Pytorch to design our AI model and Pygame to create our game.
It is worth pointing out that we planned to use Pytorch anyway because we wanted to try out a tool similar to Tensorflow.
Furthermore, support for OpenCL is also better with Pytorch compared to Tensorflow. To use something like ROCM(AMD opencl implementation), you would need to use a specific docker container, which might or might not work for your usecase.
With pytorch, there are easy installation instructions for all GPU vendors and all operating systems.

For the game itself we used an open source game engine written for python, namely Pygame.
Pygame is a composition of Python modules like computer graphic and sound libraries. Creating a game with Pygame is straightforward.

2.1. Problems

While it is true that Pytorch supports OpenCL, it is still questionable whether or not one can actually go ahead and use this.
The reason for this is the lackluster support from AMD and Intel. They often only provide OpenCL support for very specific GPUs on very specific platforms.
This means that one could potentially still not use the GPU for AI learning, and it explains why Nvidia has the clear monopoly in this space.
The other vendors simply do not invest enough time into supporting GPU compute.

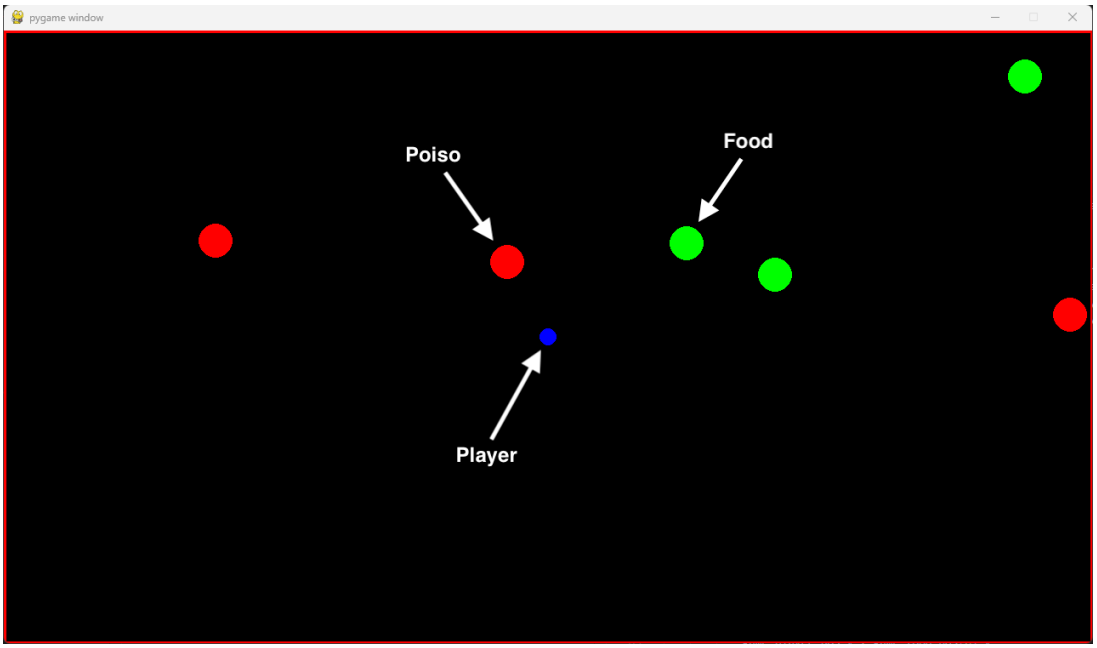
3. Making Of

We started out with just creating the base game.
Here we just made a character that can move around and has to collect green dots on the map, which are considered as food.
The red dots serve as a "poison", which simply gives the player the opposite of "food".
Hence, food will increase the player size and poison will decrease the food size.
After the base game worked, we were able to adapt the Model from the tutorial to our game, we therefore specifically wanted to make a different game to see how the AI fares on something other than what was shown.
Here we had to map all the states that our game offers to the AI and store/update all values accordingly.
It is important to note, that we started with this project before we learned about the algorithm in the AI course, this means we were rather confused at first about all the terms and calculations.
But as we started to learn about all these terms, it became clear as to what function servers which purpose.

3.1. The Game

We wanted to create our own little game and not just copy the game from the tutorial we watched.
Therefore, we designed a two dimensional game where the player is a small ball and has to collect or eat other balls.
There are two different types of balls to eat; green and red ones.
By collecting a green ball the player grows in size and gets smaller when collecting red balls respectively.

If the player has minimal size and collects a red ball the game is over. Touching the wall also ends the game.
As the player grows each time eating a green ball it gets more difficult over time to avoid the wall as well as red balls.
It is possible playing the game manually or letting the AI play the game.



3.2. The AI

Although we looked at reinforcement learning in a theoretical part at the lectures it was still an unknown field to us in a practical way.
Therefore, we initially invested some time to understand how the basics work.
We did this by reading through the code base of the already mentioned tutorial.
Furthermore, some parts are copied from the tutorial code base.
The AI of our project is divided into an agent- and a model part.

3.2.1. Model

The model consists of two classes. One of which is the Linear_QNet.
The Linear_QNet holds the two methods for forward propagation as well as the save method to save the model to a file.
It is basically the model network representing the input, hidden and output layer.
The other class in the model is the QTrainer it holds only one function which is called train_step.
This train_step method is used after each action the agent performs to optimize the model.
Roughly described the train_step method updates the loss and performs backward propagation.

3.2.2. Agent

The Agent consists of a single class which is called agent as well.
Besides the methods it holds all important parameters for our AI.
Those parameters are:

- Epsilon: Controls how much percent of the AI actions should initially be random rather than predicted by the model.
  This parameter is important as the AI has no idea what it should do at the beginning.
  As the AI learns over time this parameter value gets lowered after each game over.
- Gamma: The discount rate is a parameter that determines the relative importance of immediate rewards compared to future rewards.
  It represents the extent to which the agent values immediate rewards over delayed rewards.
  A discount rate close to 1 means the agent considers future rewards, while a discount rate closer to 0 places more emphasis on immediate rewards.
- Memory: The memory is represented by a deque datastructure.
  It is basically the memory of the AI where it can store current states, actions, rewards, next states and if it is currently in game over mode.
- Model: This field holds an instance of the previously described Linear_QNet class.
  Furthermore, here we define how many neurons the input-, hidden- and output-layer has.
- Trainer: This field holds an instance of the previously mentioned QTrainer.

3.3. Coding Log

We started off implementing our own game which has a similar simplicity to Snake implementation in the tutorial.
While the game is obviously overly simple, we wanted to focus on the AI training part and not on the game itself.
In other words, the goal was to learn how to adapt the algorithm to work well with such a game.

Afterwards, we started implementing our AI.
First we copied the whole model and agent code from the tutorial, as we were curious if it would work with our game straight away.
But it didn't work immediately of course.

The first problem we encountered was, that there was one crucial difference from our game to the tutorial game.
The player in the tutorial game moves automatically while in our game input is mandatory to make the player move.
Each time an input key is pressed the player moves one step into the respective direction in our game.
In the tutorial game the player moves permanently and changes direction each time an input key is pressed to the respective direction.
This led to another issue, which was that our game has four different input possibilities (up, right, down and left) instead of two (left and right).
This means our AI needs four output layers and not only two.

Those initial problems were sorted out rapidly. We increased the output layers in the Linear_QNet to 4 and changed the method to get the next action accordingly.

Soon after another problem arose. We noticed that the parameters in the state of the tutorial were substantially different to the ones we actually need.
Because the state formation is fundamental for a working AI it is really important to get that right.
But we had to find out that it is rather hard to find the optimal parameters for the state.

3.4. Issues

- AI learns well until it moves in a straight line
  At first, the AI seemed to not be learning enough, for which we then decided to increase the learning rate.
  Interestingly, this resulted in the AI suddenly stopping to change directions at all, meaning it would just pick a direction, and then move to its demise.
- AI unsure about next move
  In this case the AI seemed to learn the basic idea of chasing after the green ball.
  However, after a while, the AI seemed to be unsure what the next move should be, resulting in the AI moving back and forth, essentially doing the same move over and over again.
  This resulted in a stopped learning process, as the game would never end.
- AI loses interest in Food
  This is the strangest of all the issues, despite receiving input on where the food is, the AI only moves towards the food a few times.
  Meaning after 1,2,3 or x amount of rewards, it just stops trying to go after the food and eventually moves into the wall, resulting in a game over.
- Unused Information
  This is probably the hardest problem to properly solve.
  Despite the fact that we provide information to the AI about potential wall collisions, it still seemingly does not see them as a threat, and continuously moves into it.
- Where to reward?
  Just like a dog, we would like to “reward” good behavior and “discourage” bad behavior.
  Problem is, how can we make sure we actually reward the right action?
  For this example, if we simply reward moving towards the food, then the AI could figure out that it could continuously make the same move, one step towards the food and one back.
  This would result in an infinite loop of rewards, with the only action being the discouragement of moving away from the food.
  However, constant negative rewards might also confuse the AI.
  In short, the hardest part is making sure, that the AI sees a clear path to the food from each possible state.
  Providing only the location of the food is not good enough, as it doesn’t give the AI a clear enough path towards the food.

CODING BUGS

TRYING OUT DIFFERENT VALUES (reward, reward by how close the player is to obstacles)

MULTI OBSTACLES

3.5. Input Mapping

The input mapping is done with a simple integer, this means that we have the value 0 to 3.
0 is up, 1 is right, 2 is down, and 3 is left. (CSS style)

STATE INPUTS