PySnake

Kaj Habegger & Fabio Lenherr

Miniproject

Table of Contents

1. Introduction

For this project, we decided to test reinforcement learning with PyTorch on a small game made with PyGame.
As a starting point to learn from, we used this tutorial on Youtube: Link to video
The video explains the steps needed to create an agent for the snake game, which we adapted to work for our own game.

2. Tooling

We decided to use PyTorch in particular to see both the differences to TensorFlow and to get better support for other GPU vendors such as AMD or Intel.
The game engine was a simple choice as it is also written in python, and therefore offers easy setup and integration with the agent.
For a bigger game, it would be a better choice to use an established game engine, or at least a non-interpreted programming language for better performance.

2.1. Problems

While PyTorch does support other vendors, these vendors themselves do not necessarily support GPU compute.
We also had to face this issue as AMD only supports very specific cards on very specific platforms.
This would mean that one would still have to potentially fall back to using Nvidia.

2.2. Usage of PyTorch

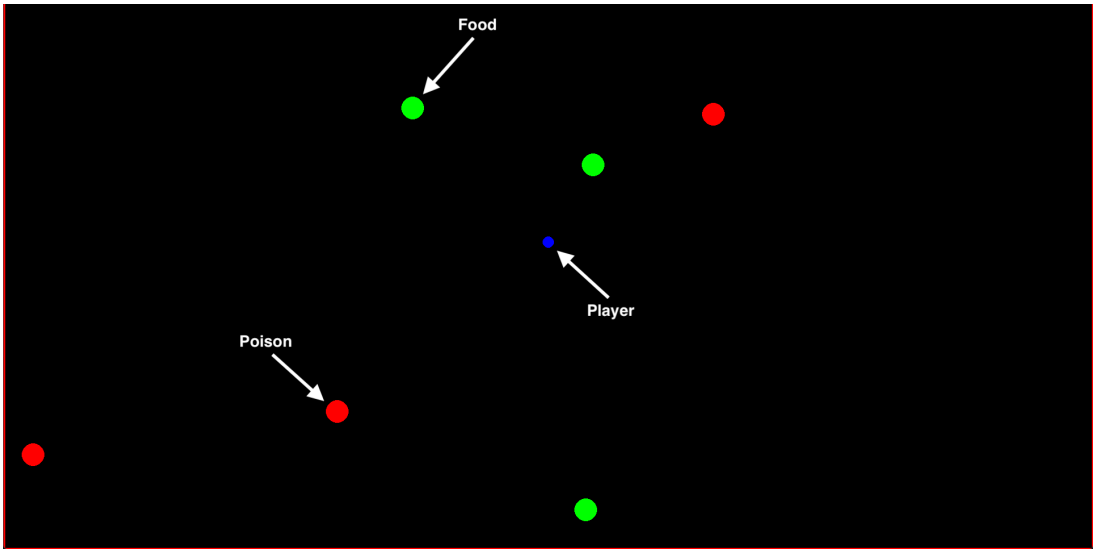
PyTorch is very similar in usage to TensorFlow with a few different terms being used.
In general, one can just import partial packages, which will then do the vast majority of the heavy lifting.
E.q. No one implements their own model by hand, instead we can just call functions from PyTorch that provide us functionality such as relu.

2.3. Costs

In our case, both the game engine and the machine learning library are free and open source, meaning anyone can go ahead and create a game with an AI by simply using their own time.
In terms of learning, PyTorch might not be as well known as TensorFlow, but there are already tons of tutorials and other resources on the internet, which will help you get started with this library.

3. Making Of

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 green balls, the player grows in size and gets smaller when collecting red balls respectively.
If the player is at the 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 practically.
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 adapted 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.
In this train_step, there is obviously also the Q function that was explained in the AI Application lectures:

```
1 Q_new = reward[idx]     # idx is the state iteration
2 if not game_over[idx]:  # in other words, this is done for each possible state
3     Q_new = reward[idx] + self.gamma * torch.max(self.model(next_state[idx]))
```

This refers to the formula: $r_t + \gamma * \mathbf{max}Q(s_t + 1, \alpha)$ which means current reward plus discount rate multiplied by the next best state (max rewards)

In other words, it calculates the best next action according to the current reward and according to the max reward at the end, with the discount rate diminishing rewards that are "too far away".

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 data structure.
  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.

Afterward, 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.
Unsurprisingly, this was not the case. In snake, there are only 3 possible actions, left, straight, or right.
This means the AI has 3 output states, however, our game allows up, right, down, left. This means we now need 4 output states, and it also means that our AI has to be a bit bigger with more states to care about.

The problem about states was easy to solve, we just had to increase the output layer of the Linear_QNet to 4.
However, the rest of the different states were not as straight forward.
We tried a variety of input states, even going as far as to mathematically calculate the next best move.
Something that is of course not feasible in other games, situations.

3.4. Problems to solve

- 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
  Despite the fact that we provide information to the AI about potential wall collisions, it still seemingly does not always see them as a threat, and sporadically moves into it.
  Here, we can only improve this by trying to provide more and more "useful" information that the AI can work with.
- 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, something we would want to discourage.
  In this case one would need to punish the AI every time it moves away from the food, however, one has to be careful to not invoke other unseen consequences.
- What states to use?
  The states that the AI receives can heavily impact the learning process, providing more information automatically makes it harder for a single state to matter.
  This means one should keep the states as few as possible, while not withholding important information.

3.5. Multi Obstacles

At first, we only had one single food on the map, this was to simply train the model to chase after it.
Later on, we decided to also introduce poison for the AI to avoid, and further to increase the amount of both types.
This meant, once again, that the input states for our AI had to increase, as suddenly the amount of possible paths to both positive and negative rewards increased drastically.

3.6. 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)
For the random part, we simply use the rand functionality from python, and on the Linear_QNet side, we can use the max function to receive the highest match. (each direction has 1 neuron -> we receive the 1 neuron that had the highest value)

```
current_state = torch.tensor(state, dtype=torch.float) # transform our state to a tensor
prediction = self.model.forward(current_state) # use the tensor as the input for our model
prediction = torch.argmax(prediction).item() # set highest value neuron in the model to be output
predicted_move[prediction] = 1               # this sets the output for our game -> ex. predicted_move[0] == up
```

4. Conclusion

Reinforcement learning provides a very sophisticated way of creating agents for various applications. The issue, however, is that these agents are often not consistent enough.
Even for this game, it is not entirely trivial to find the right input states and parameters in order to get the AI to behave correctly in each situation.
This means that for bigger games it is even harder. A good example is Dota2, while the AI by OpenAI has managed to beat professional players, it has only done that against regular play.
As soon as players started using strange and supposedly nonsensical strategies, the AI seemed to crumble, as it

could not manage to adapt to it fast enough.
Reinforcement learning is thereby a great tool, but one also needs to know the limitations of it.

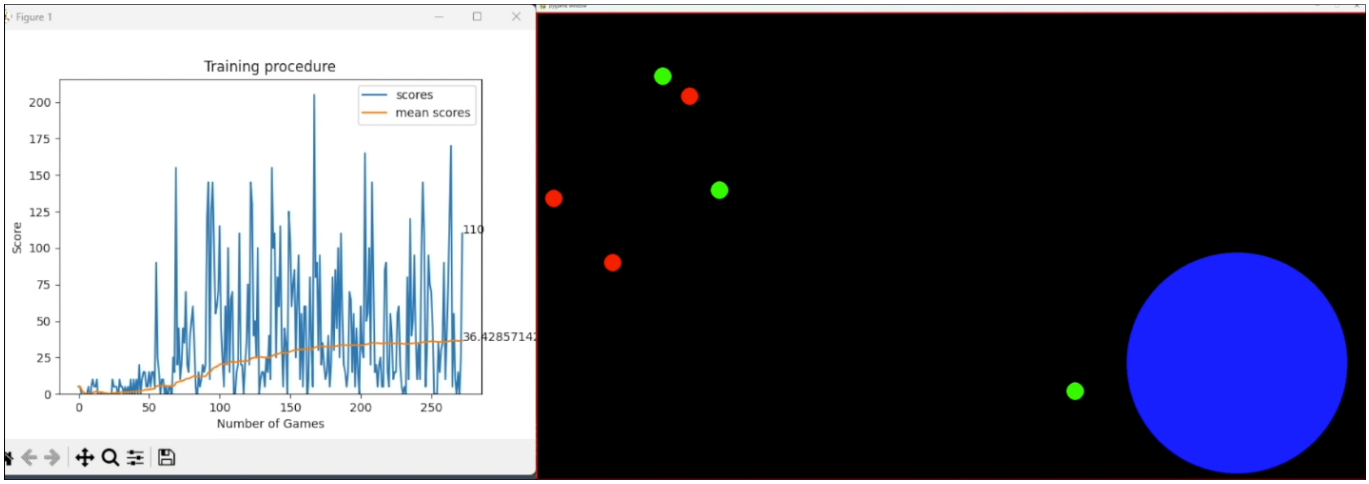


Figure 2: Plot of the AI learning to play the game