Implementing a Reinforcement Learning based Snake AI Agent using PyTorch and PyGame

Parth Shah
AU1940065
parth.s5@ahduni.edu.in
Ahmedabad University

Malav Doshi
AU1940017
malav.d@ahduni.edu.in
Ahmedabad University

Khushi Shah
AU1920171
khushi.s18@ahduni.edu.in
Ahmedabad University

Kashvi Gandhi
AU1940175
kashvi.g@ahduni.edu.in
Ahmedabad University

***Abstract*—Training a model using negative or positive rewards is termed as Reinforcement Learning and extending it using neural networks is known as Deep Q-Learning. Using these two methodologies and libraries like PyGame, PyTorch, etc., this project report focuses on explaining how an agent can be trained to play the snake game and score as high as possible.**

***Keywords*—Reinforcement Learning, Deep Q Learning, PyGame, Pytorch, Bellman Equation**

I. Introduction

Reinforcement Learning is teaching a software agent how to behave in an environment by telling it how good it's doing. Deep Q Learning extends reinforcement learning by using a deep neural network to predict the actions of any agent. In this project we have used a snake game which uses Pygame and can be played by any user. Next, we build a model using PyTorch. The model used here is a simple feed forward neural net with few linear layers. Finally, we build an agent that plays the game and trains itself using the created model to predict the best possible move. The rewards, states and actions for the game have been defined.

Rewards

eat food	+10
otherwise	0
game over	-10

Actions

[1,0,0]	straight
[0,1,0]	turn right
[0,0,1]	turn left

The state has 11 values. Firstly, it gives information for whether the danger is ahead, right or left. It also states the direction of the snake's head and the food. All these values are represented as `boolean`.

The model is a feed forward neural net with three layers - input layer, hidden layer and output layer. The input layer gets the 11 different states

II. About the Game

The Snake Game is the same as one might have tried playing in old GSM mobile phones. The aim is for the user to survive without hitting the walls or the snake's tail. Moreover, the user scores by eating the food presented on the screen. We have used an already existing game coded in python build using PyGame.

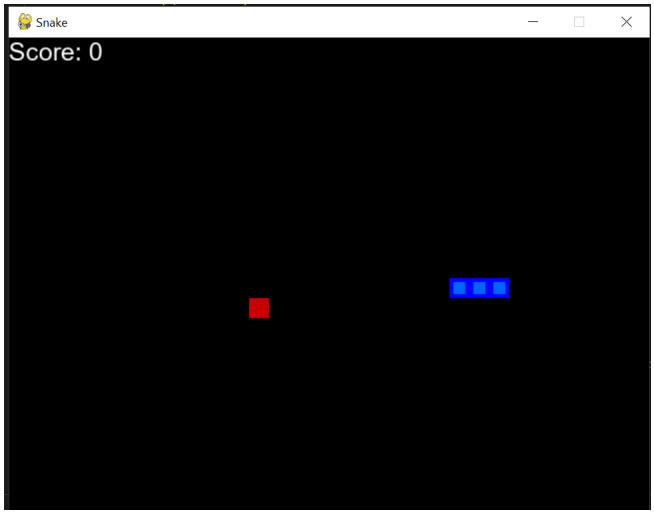


Fig. 1. Snake Game

III. Possible Approaches

There can be three broad approaches to an AI snake agent. The three categories are *non-ML approaches*,*Genetic algorithms*, *reinforcement learning*.

- Non-ML Techniques: This approach has a trivial, unbeatable solution. This approach create a cyclic path that goes through every square on the board without crossing itself which is called Hamiltonian Cycle. However, this wastes a lot of time and and a lots of moves with increasing time complexity with increasing board size.
- Genetic Algorithm: Another approach which is modeled of biological evolution and natural selection.Here the perceptual inputs are mapped to actions outputs. The input can be of the for directions and output can be action. In this case, the fitness function select est individuals from

a given generation. Following, a new generation is then bred from the best individuals with addition of random mutations. The advantage is that predicting next move is faster. However, it can get very difficult became mutations are random.

- Reinforcement Learning: Involves an agent,an environment, set of actions and reward function. Here the agent explores as well as exploits the environment with a goal to maximise reward function. We extend this to Deep-Q-Learning. Where the neural network learns the "Q-function". We use Bellman Equation that guides the Q in the right direction.

We have considered Reinforcement learning as our primary focus of discussion.

IV. Model Used

This model based agent uses a Deep Q Learning model to train itself. The aim of the agent should be to improve the Q value, the quality of action, of the model. The training process of the agent is such that first, the model is initialized with a Q value with some random parameters. Next, the agent chooses an action using the *model.predict(state)*, the action is chosen at random. the agent performs this action and measures the reward. The reward then updates in the Q value and used to train the model. At the beginning of training the model, we choose a random move in order to understand the action-reward system of the environment. After a while, the model does a trade-off between random move and model prediction, this is also known as a trade-off between exploration and exploitation. This process is done in an iterative manner to train the model for better accuracy. Next, in order to train the model we use a loss function to minimize and maximize, we have used the Bellman equation which is used to update the Q value.

$$Q_{new}(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma maxQ'(s',a') - Q(s,a)]$$

- $Q_{new}(s,a)$ represents the new Q value for that state and action
- *Q(s,a)* is the current Q value
- α represents the learning rate
- *R(s,a)* reward generated for taking that action for that state
- γ represents the discount rate
- *maxQ'(s',a')* maximizes the future reward given the new state and all possible actions in that state

$$loss = (Q_{new} - Q)^2$$

The loss function is thus defined as the mean squared error, i.e. average squared difference between the new Q value and old Q value.

To explain in detail, the Deep Q-Learning model uses the past knowledge or experience to learn in small batches in order to avoid skewing the dataset distribution of different state_actions, rewards, and next_states that the neural network gets as input. In order to update the model weights, we use the Bellman Equation. This equation provides an approximation for Q to guide the neural network in the right direction. As the network improves, the output of the Bellman Equation also improves. The Bellman variant equation used in our project is: $Q(s,a) = reward(s,a) + \gamma * max(Q(s',a'))$

V. Methodology

A. Modifying base game file

Currently, game.py can be played by a human user. Now, we modify the class such that it can interact with the game. Thus, we add the following functionalities: `self.reset()` , `play_step()`, `_move()` to reset the game, play the next step and change direction of movement respectively.

B. Setting up agent file

```
class Agent:

    def __init__(self) -> None:
        pass

    def get_state(self):
        pass

    def remember(self, state, action, reward, next_state, done):
        pass

    def train_long_memory(self):
        pass

    def train_short_memory(self):
        pass

    def get_action(self,state):
        pass

def train():
    pass

if __name__=='__main__':
    train()
```

Fig. 2. Base functions for Agent

Furthermore, we initialise the agent with the following values {n_games: 0, ϵ: 0, γ: 0.9, memory: deque(maxlen=MAX_MEMORY} (where ϵ controls the randomness, γ controls the discount rate, and memory is a deque data structure with size MAX_MEMORY)
Additionally, complete a `get_state()` that returns 11 states (in our case). Briefly, this includes information about danger, move_direction and food_direction.
Finally, We also complete the `train_long_step()` and `train_short_step()` which trains for a series of steps and the previous step respectively.

C. Exploration vs Exploitation trade off

Tradeoff of exploration (in the beginning, explore as much as you can) vs exploitation (use what you know better after a certain point)

The more games you will play, the smaller the ϵ value will get, the less number of times will the `random.randint(0,200)` be less than `self.epsilon`, (and might also become a negative value and not go in the if at all) $\rightarrow$ and the chances of exploration would decrease.
Here, it is imperative to note that `self.model.predict` returns a raw value of prediction, and the one with maximum value is assigned 1; while others are assigned 0.

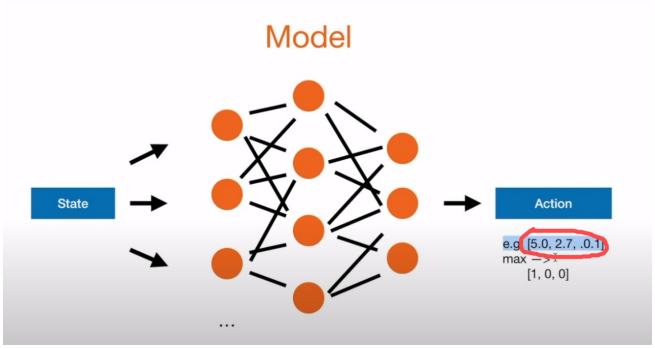


Fig. 3. argmax of raw value

```
def get_action(self,state):
    # random moves: exploration vs exploitation tradeoff
    self.epsilon =80-self.n_games
    final_move=[0,0,0]
    if random.randint(0,200)<self.epsilon:
        move=random.randint(0,2)
        final_move[move]=1
    else:
        state0=torch.tensor(state, dtype=torch.float)
        prediction=self.model.predict(state0)
        move=torch.argmax(prediction).item()
        final_move[move]=1
    return final_move
```

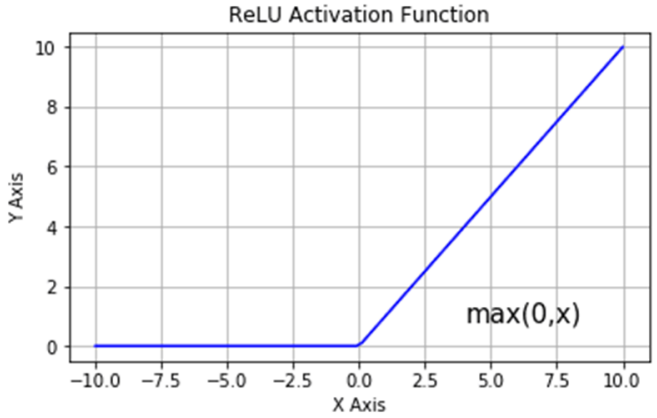
Fig. 4. getAction() function

D. Linear QNet and QTrainer

1) QNet: We setup a two layered neural-network using the `nn.Linear()` function. The input and output are *input_size* and *hidden_size* for the first layer and are *hidden_size* and *output_size* for the second layer.

```
class Linear_QNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size) -> None:
        super().__init__()
        self.linear1=nn.Linear(input_size,hidden_size)
        self.linear2=nn.Linear(hidden_size, output_size)
```

2) ReLU Activation function: Rectified Linear Unit Function or ReLU is applied on the layers using the `torch.nn.functional()` method.



3) QTrainer: Passing the learning rate, gamma (discounting factor) and model to Adam optimizer using `optim.Adam()` method

```
You, 1 second ago | 1 author (You)
class QTrainer:
    def __init__(self, model, lr, gamma):
        self.lr=lr
        self.gamma=gamma
        self.model=model
        self.optimizer=optim.Adam(model.parameters(), lr=self.lr())
                    You, now • Uncommitted changes
```

4) Loss function: Mean Squared Error is used as the loss function, and is implemented by `nn.MSELoss()`

$$loss = (Q_{new} - Q)^2$$

To implement the loss function, we set the optimizer to `zero_grad()`, calculate loss between target and prediction and finally apply backpropagation using `loss.backward()`

5) Using Bellman's equation:

$$Q = model.predict(state_0)$$

$$Q_{new} = reward + \gamma * max(Q(state_1))$$

Using the simplified update rule, we predict values + update Q_{new} (if not `done`)

VI. Results

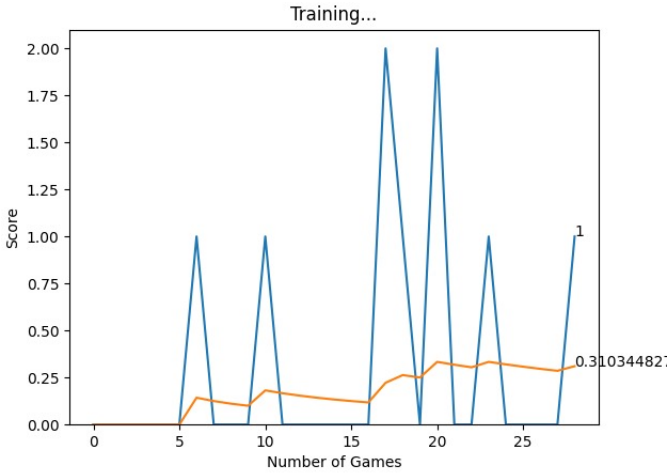


Fig. 5. Results after 25 games

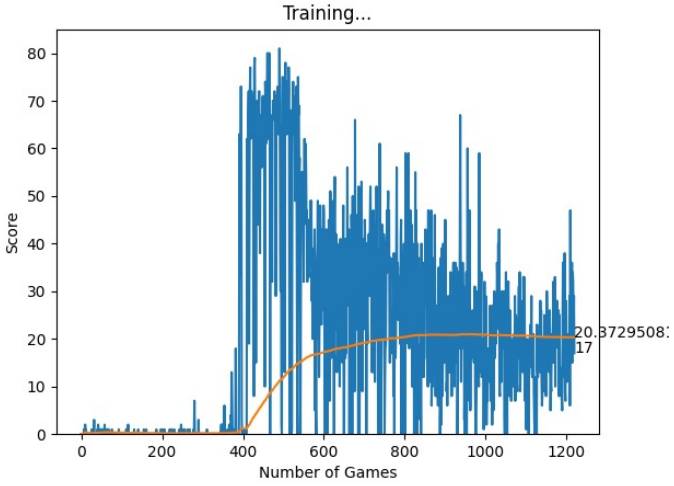


Fig. 6. Results after 1000 games

VII. Future Work

On implementation, we noticed that the agent gets stuck in a loop several times because it fixates on a position that it comes across during the learning process and has the tendency to stick to that position after several iterations. This causes the agent to fall into the loop and eventually the game ends. The target is that at any point of time in the game the head of the snake must be able to reach at least 80 percent of all empty plots and if we start out using A* to find the quickest route then once the snake gets a little bigger, the opposite of A* must be used to find the longest path.

References

[1] Clark, T. (2021). Training a snake game AI: A literature review. Training a snake game AI. Retrieved November 16, 2022, from https://towardsdatascience.com/training-a-snake-game-ai-a-literature-review-1cdddcd1862f

[2] Deep Q-learning tutorial: Mindqn - towardsdatascience.com. Deep Q-Learning

[3] Tutorial: minDQN. (2020). Retrieved November 16, 2022, from https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc

[4] Torres, J. (2020). The bellman equation. V-function and q-function explained — by Jordi ... The bellman equation. Retrieved November 16, 2022, from https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7 Wikimedia Foundation. (2022, November 15).

[5] emSnake (vid_game_genre). Wikipedia. Retrieved November 18, 2022, from https://en.wikipedia.org/wiki/Snake_(video_game_genre)