B20AI013_B20AI061_A4_DeepRL

Team Members

Vikash Yadav - B20AI061

Ishaan Shrivastava - B20AI013

Choosing Game Seed

As per the rules:

Roll Number chosen: `B20AI061`, corresponding seed: `22019061`

Dependencies

Install the packages `pygame, pytorch (perfferably cuda version), numpy` .

You can do this via *pip install packagename*. Then run game.py.

Part 1 - Describing the game, reward function

This is a simple game where our the aim is to avoid enemies and reach a target location.

- **AGENT:**

  Our agent is a green square, who can move around the window in any direction (combination of up,down,right,left). The aim of the agent is to reach the target (which is a blue box) while avoiding contact with enemies.

- **ENVIRONMENT**

  The environment is simple, there are 4 walls of the window, the agent bounces off the walls, there is no penalty associated with hitting walls. There are several enemies (moving red colored squares) in the environment (by default `7`). There is one target block (blue colored square). The agent has to avoid contact with the enemies and reach the target square.

  When the agent reaches the target or gets hit by an enemy, the location of target square is reset randomly and the game proceeds. The game never stops technically.

- **ACTIONS:**

  The agent has 5 possible actions to take at any state of the game. Those are: Do nothing, Move left, Move right, Move up, Move down

- **OBSERVATIONS:**

  There are just 3 observations the environment gives us: nothing, the agent reached the target, an enemy hit the target.

- **STATE OF THE GAME:**

  This represents the whole game basically, given us exact details of the whole state the game is in. Now many of the parameters of the state of the game don’t change when our player moves around like board size, friction etc. So we first extract the useful attributes of the state of the game (these uniquely identify every possible state of the game) and only use these as input to our Nerual Network of the AI Controller.

  So, the useful attributes of the state of the game that we chose are:

  - Location of target (x and y)
  - Location of player (x and y)
  - Velocity vector of player (x component & y component)
  - Location of each enemy (x and y)
  - Velocity vector of each enemy (x component & y component)

  So in total we have `6 + ENEMY_COUNT * 4` number of concerned attributes of the state of the game. The agent has to decide which action to take based off these values of the state. **This is the version of code we submitted.**

- In a later version of our code, we removed the inputs of the enemy positions, as we thought this was making the network too complex to be able to learn meaningful things in just 100 epochs. To make the network simpler, we removed the enemy locations and velocities, so now,
  - Location of target (x and y)
  - Location of player (x and y)
  - Velocity vector of player (x component & y component)

  These are our only inputs to the neural net. So input size = `6` for this version.

Reward Function

Based on the understanding of the game. Our reward function is such:

$$\text{reward} = \begin{cases} +100 & : obs == Reached\_Goal \\ \text{-1000} & : obs == Enemy\_Attacked \\ \text{-5} + \dfrac{5*(\text{map\_width - dist(agent, goal)})}{\text{map\_width}} & : obs == Nothing \end{cases}$$

- Survival is a very important instrumental goal for the agent to pursue, without which it cannot hope to accumulate rewards by collecting the lucrative goal objects which are the terminal goal of the game. We penalise hitting the enemy red boxes much more than reaching the goal. (`+100` versus `-1000`). This is to encourage the agent to ensure its survival before attempting to get the goal reward.
- It is important to note that the values of the rewards corresponding to the `Reached_Goal` and `Enemy_Attacked` event are much larger in magnitude than that of the idle reward in the `Nothing` state. This is because the idle reward is computed for most of the frames during the simulation, hence it quickly accumulates to have a comparable influence to the instantaneous hit-based rewards.
- A moderately low value of the default reward `game_state.GameObservation.Nothing` encourages the agent to seek out ways to maximise its reward instead of doing nothing. Hence we set it to `-5`.
- We further guide the agent by increasing the reward for every frame proportional to the square of how close the agent is to the reward on the screen. We take the square so that the agent is encouraged to approach closer and closer to the reward instead of camping nearby it.

  `reward += 5 * ( 1 - new_dist / max(game_constants.GAME_HEIGHT, game_constants.GAME_WIDTH))**2`
- The configuration of the training environment is such that **every agent starts from the same velocity and position**, which means that the agent is **susceptible to memorising the positions** to take during the training epochs. We ensure that this does not happen by:
  - Regularising the DQN network using `weight_decay` parameter in the optimizer. **Weight decay encourages the model to learn regularized representations** and not overfit, which can help combat memorization.
  - Running the **training simulations for every epoch for a very long number of frames** `1000`. This, along with weight decay, we found, allows the agent to perform well indefinitely during the training time, as shown in the video.
- **Lowered the value of learning rate** in order to stabilise the learning process and make it less prone to the sparse gradients. Thus, I set the learning rate to `1e-5`.
- We checked for **reproducibility**, and in two separate evaluation runs, we reached the goal `62` and `49` times with the above settings which is very very good in comparison to the myriad of other settings we tried. (you may see them commented in the code itself in the reward calculation portion).

- After our tweak (of removing the enemy inputs):

  On our best run, we observed `276` times goal reached. However, this was unstable and was not reproducible across multiple runs. This only shows how much dependent the DQN training is to the initial values of the neural network weights, which are randomly taken at initialization on every run.

Part 2 - DQN RL Algorithm

The base algorithm we have used is Deep Q-Learning (DQN). Here the Q-Table of the Q-Learning algorithm (table which give q values for each state and action pair) is replaced by a neural network. This involves feeding the states of the game to a feed forward neural network and getting the q values for all the possible actions from that state as the output of the neural network.

So, given a state, we can run a forward pass, get the q-values, and choose the action that has the max q-value. This way the agent chooses its action. The agent receives a new state and reward as the result of the action.

Then principles of Bellman Equation (Q-Learning) is used to calculate the loss of the network (using Huber Loss) and the loss is backpropagated so we get a network that learns to assign correct q-values.

$$\Delta w = \alpha[(R + \gamma\, max_a\, \hat{Q}(s', a, w)) - \hat{Q}(s, a, w)]\, \nabla_w \hat{Q}(s, a, w)$$

Change in weights

learning rate

Maximum possible Qvalue for the next_state (= Q_target)

Current predicted Q-val

TD Error

Gradient of our current predicted Q-value

In essence, the Bellman equation assumes that if the DQN has converged to its optimal value, that is, if $\hat{Q} = Q$, then the below equation would hold in an exact sense:

$$\hat{Q}(s, a) = R(s) + \gamma \max_{a'} \hat{Q}(s', a') P(s'|s, a)$$

The loss is essentially the $L2$ norm of the difference between the LHS and the RHS of the above equation. If the DQN network has converged, then the loss of the network would also be zero over any state-action sequence.

Note: in our case, the probabilities for $P(s'|s, a)$ are known and the state transition probabilities for a state, action pair (s, a) are either 0 or 1, as this is a Markov Decision Process.

- **Network Architecture**:

  My Neural Network looks like: (We chose a small model to prevent overfitting)

```
class QNetwork(nn.Module):

    def __init__(self, state_size, action_size):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 64)
        self.fc4 = nn.Linear(64, action_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        q_values = self.fc4(x)
        return q_values
```

- **What is ϵ?**

  In reinforcement learning, we need a tradeoff between exploration and exploitation. ϵ helps us realise that. ϵ is the probability with which we shall take a random action vs. an action from our knowledge.

  Initially we have set ϵ = 1. That means the first action is completely random. Then as our model trains, at each training step, we decay the value of ϵ as ϵ * decaying_factor (we have used 0.99) untill ϵ reaches a minimum value that we have set.

So the idea is, at start of training, take more random actions to explore the enviroment, but as the agent trains, start to exploit the knowledge more and explore less.

- **What is replay_memory?**

  Our neural network updates from experience. Instead of updating from each experience, we store the last k experiences in a buffer we are calling `replay_memory`. Then at each step, instead of updating from the latest experience, we update the network from a batch of experience made form randomly sampling `batch_size` number of experiences from the `replay_memory`.

  By randomly sampling from the replay memory, the algorithm can learn from a more diverse set of experiences, preventing it from getting stuck in local optima, which can happen if we call batch updates on the loss from adjacent frames, as they are heavily correlated to each other.

  We have used a deque to implement `replay_memory`.

- **Training pipeline**:

  For n number of epochs/episodes:

  - We intialise a `state` and for `max_steps_per_epoch` number of moves, we call `GetAction(state)` (which gives an action based on ϵ, exploration vs. exploitation tradeoff) to get an action and do the action.
  - We remember the `(state, action, reward, new_state)` tuple in our `replay_memory`.
  - For exploitation actions, our actions are coming from a forward pass of the neural network embedded in our `AI_controller`.
  - For each training step, we sample batch_size sized batches of previous experiences (those tuples we stored) and find the loss based off the bellman equation, we backward propagate the loss to update the network.
  - This way, slowly our network learns to give better and better Q-values which give better and better actions.

Part 3 - Modifying `game_constants.py`

The objective of this part is to edit the various game constants such as:

1. `GAME_SEED`: To verify that the agent does not memorise the randomness associated with the game seed and works for a variety of different game seeds instead of memorising and overfitting the one it is trained on.

   Explanation - It is very very likely that allowing this hyperparameter to vary during the training process, the training process will overfit much less and the final performance, although it may take some time to converge, will generalise much better than our current setup.

2. `GAME_WIDTH` and `GAME_HEIGHT`, `GAME_FRICTION` and `FPS`: To verify whether the agent is overfitting on the game dimensions, we must test whether it learns any implicit biases relevant to the dimensions and the physics of the game. A good agent will be able to compensate for changes in these constants.

   Explanation - Our agent is designed to accomodate to these `GAME_WIDTH` and `GAME_HEIGHT` as they are taken as variables in our reward formulation.

   As for `GAME_FRICTION`, we have observed that the agent tends to move in sudden bursts in order to avoid and pursue points of interest. It is to be expected that Increasing the value of friction should not change this behaviour at all due to the jerky nature of its movements as the agent takes into account not only the velocity, but also the position of objects which allows for a self-correcting nature of the response from the model.

   The `FPS` hyperparameter, as far as we know, does not affect how the game will play out, except for the fact that it will appear slower/faster during display depending on increase or decrease in the FPS.

3. `GOAL_SIZE`: To verify changes in the behaviour of the game to the size of the goal object.

Explanation - The goal size is already very small, which poses a lot of difficulties to the agent and makes it difficult to collect the goal which is visible during test runs. Further decreasing the value of the goal size may make it even harder to impossible to train the agent to collect the goal. However, conversely, increasing the goal size may incentivise the agent to pursue goal collection to maximise its reward even more, which could mitigate some of the stalling behaviour observed in the agent.

4. `ENEMY_COUNT`: This is an important part of testing the implicit biases associated with the training of the RL agent. We must test whether it is able to adapt its strategy to lesser or more number of agents.

   Explanation - The agent already has a lot of difficulty adapting to the locations of the enemies. We expect that increasing the number of enemies could pose significant challenge to the model and may make it impossible for the agent to collect the reward regardless of any reasonable formulation for the reward used.