RL GAMES

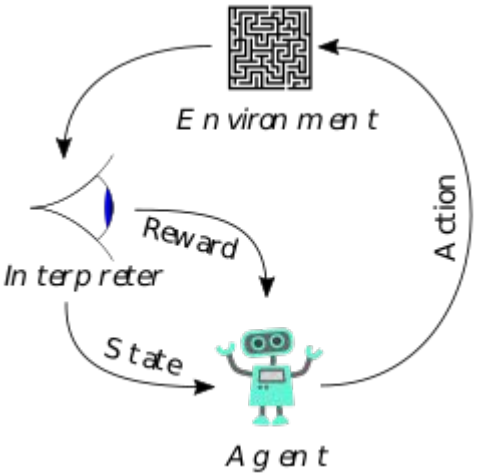
Team:

Chetan Reddy N
Sriram S M
Dhananjay Balakrishnan

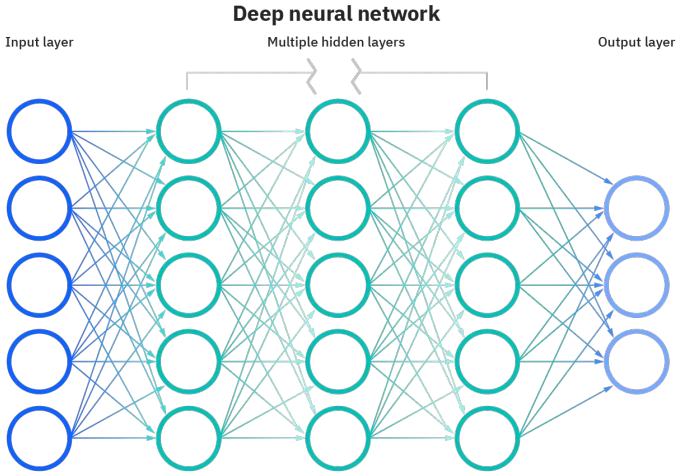
Introduction to the Problem Statement:

- We are given an 8x8 grid on which 2 agents play against each other. At any instance, the agent can choose to move in 1 of 3 directions.
- On each move, the agent gets a **penalty of 1**, and on successfully moving into a square with food, it gets a **reward of +4**. On hitting the corner, it gets a penalty of 2, and on annihilating the other agent, based on the score difference, it gets an arbitrarily valued reward or penalty.
- Naively, we can start by assuming a 265-bit long vector that is returned by the encode() function as the state space.
- As there is a large number of possible states, we will need to use some kind of function approximation to the state space. We have used Neural Networks for this purpose.

Deep Reinforcement Learning



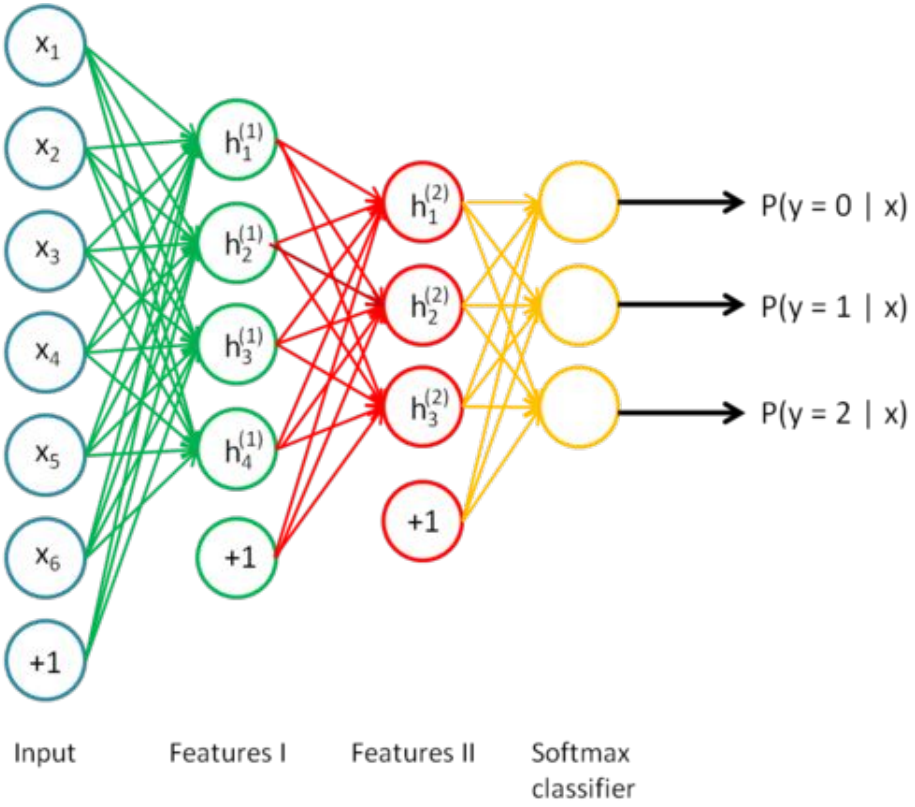
+



We have explored 2 paradigms of Deep Learning based RL methods, one involves directly predicting the action probability distribution using the neural network, and the other involves predicting the Q-Values, and deciding actions based on it.

Feed Forward Neural network with Policy Gradient methods

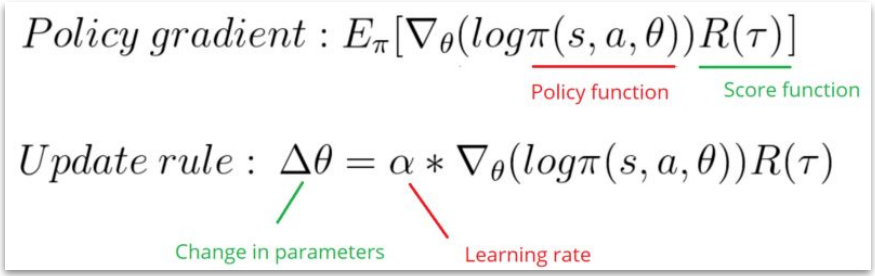
- **A densely connected feed forward neural network** was used with the current state as the input and a **probability distribution** as the output corresponding to to the 3 possible actions
- Experimented by varying the number and size of hidden layers, Input structure of input dimensions to improve efficiency of the model



Feed Forward Neural network with Policy Gradient methods

Policy gradient steps:

- Action is chosen according to the action probability distribution output by the model, using the current input state
- **Cross-entropy loss** is computed using the distribution, assuming the taken action is the optimal one - **Confidence**.
- **Gradients** of the trainable variables of the model are calculated based on the loss



- These gradients are multiplied by the corresponding reward of the taken action - **Accuracy** - and this product is cumulatively added over a number of timesteps
- The weights of the model are updated once every couple of episodes with the cumulative gradients - **Adam** update rule
- Over a number of episodes the gradients tend to decrease and **convergence** is observed

What to use as the input to the neural network?

Input structures tried:

- **265 long binary vector** containing a flattened encoded state (def encode in environment code) and some booleans indicating head positions and scores of the 2 agents
- **16 long vector** containing the absolute location of the 4 food locations and the 2 agents, the absolute scores and head positions of the agents

We observed **infinite loops**. Possible causes of failure:

First input structure: Too long. The neural network is not able to classify each situation differently. The binary entries do not give clear information about the current state.

Second input structure: The inputs are not binary. States that are numerically close in this input structure need not be actually close for action-taking in the environment.

In both cases, the model was conflicting its own learnt weights in different states due to the flawed input structure.

Empathy

We then thought that maybe giving the entire current state as the input to the neural network isn't the right way forward.

To figure out the new way of input, we tried to put ourselves in the position of the agent and try to figure out what the most important features would have to be. Some feature engineering led to a **set of 15 features** that we felt, would accurately and compactly represent the current state.

How Empathy?

- We came up with the list in the image:
  - Food position
  - Opponent position
  - Border position
- The ‘food’ is the **nearest food relative** to the location of the agent
- Proximity of enemy considered
- The directions left, right, front, back are decided considering the head position of the agent
- Comparable to a decision-tree approach

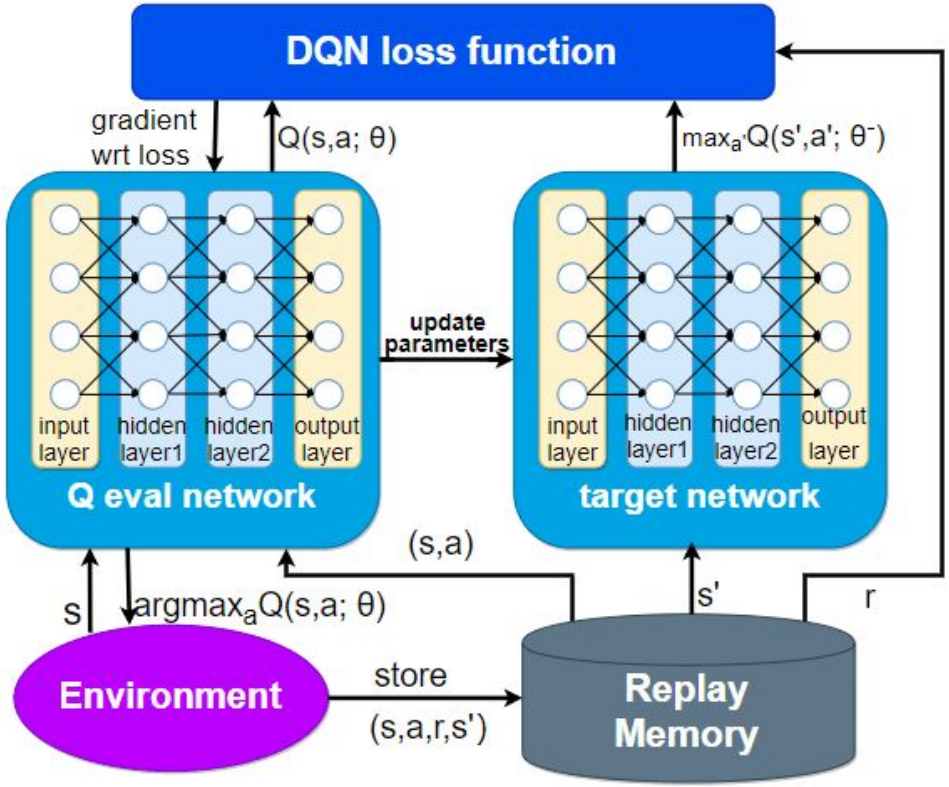
```
food_is_behind_of_B,
food_is_front_of_B,
food_is_left_of_B,
food_is_right_of_B,
A_is_behind_of_B_and_scoreB_more_than_scoreA,
A_is_front_of_B_and_scoreB_more_than_scoreA,
A_is_left_of_B_and_scoreB_more_than_scoreA,
A_is_right_of_B_and_scoreB_more_than_scoreA,
A_is_behind_of_B_and_scoreB_less_than_scoreA,
A_is_front_of_B_and_scoreB_less_than_scoreA,
A_is_left_of_B_and_scoreB_less_than_scoreA,
A_is_right_of_B_and_scoreB_less_than_scoreA,
border_is_left_of_B,
border_is_right_of_B,
border_is_front_of_B,
```

Perspective of agent B

Few things about NN + Policy Gradient Methods

- Firstly, when we tried training, we noticed that there was not really a significant improvement over the previous case. It was still not really going towards the food and kept going around in circles.
- We had a brain-wave, and **to train**, we used food reward to be **+50**, while we used the food reward to be +4 in the evaluation step.
- We could see the improvement! The agent went after the food and was consistently defeating the other agent (agent B, which makes random moves at every timestep).
- However, we wanted to come up with a method where we could train the agent using the food reward to be +4 itself.

PyTorch neural network using DQN

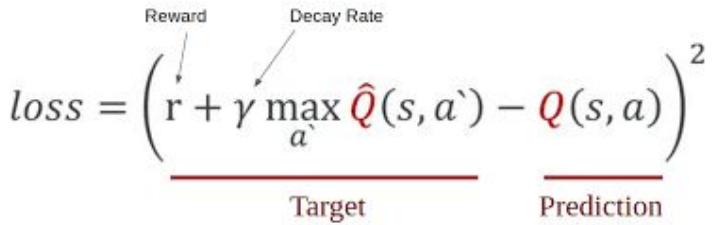


Few Things about DQN

- The essence of a DQN is to obtain the **Q-values using a neural network**
- However, unlike the usual neural network training, we don't have the true values.
- In a DQN, these values are approximated using the following formula:

$$y_t^{DQN} = r_{t+1} + \gamma \max_a Q(s', a; \theta_t)$$

- With this, the **loss is calculated and backpropagation is performed** to update and learn the weights

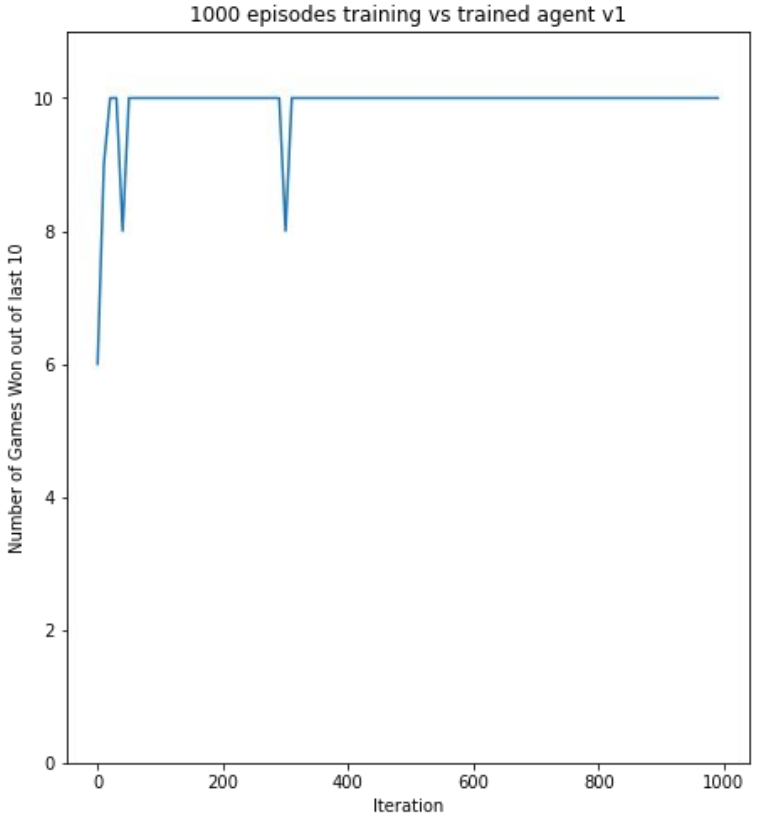


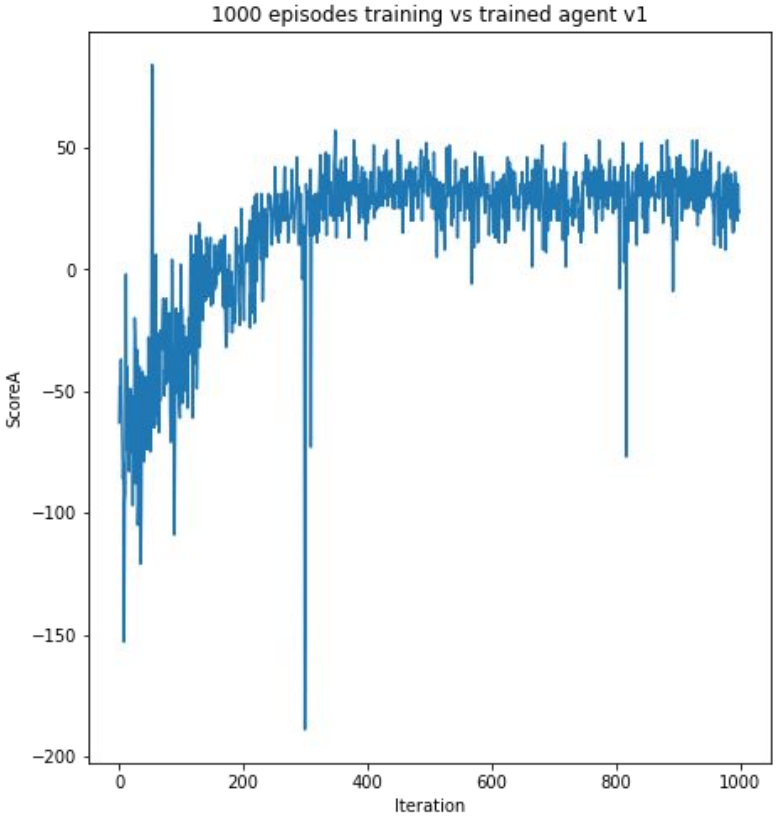
Neural Network Architecture Used

Layer (type)	Output Shape	Param #
Linear-1	[-1, 64]	1,024
ReLU-2	[-1, 64]	0
Linear-3	[-1, 64]	4,160
ReLU-4	[-1, 64]	0
Linear-5	[-1, 32]	2,080
ReLU-6	[-1, 32]	0
Linear-7	[-1, 3]	99

Total params: 7,363
Trainable params: 7,363
Non-trainable params: 0

The Results of DQN Training





Some Experimentation and Areas of Improvement:

- The training of the agent was done completely against a random agent. We felt that training the agent against a smart agent might make it robust and learn better.
- We tried training the agents against each other (2nd phase of training) using the initially trained model. We witnessed close competition among the agents.
- We also noticed that often, the agent disregarded the opponent. To further improve its performance, we could incentivize the agent to go after the opponent
- To take it forward another step, we enabled a user input as the actions for one of the agents. This was a way for a human to compete against the trained agent - **Human vs Smart Machine**.
- Further this could be used as a 3rd phase of training over a number of Human vs Machine episodes to make the model more robust.

Thank You!