Programming Assignment 2: Report

CS F407 - Artificial Intelligence

Name: Meghana Srinidhi Seshasai
BITS ID: 2021B5A72517G
Email: f20212517@goa.bits-pilani.ac.in

1. Introduction

Problem statement: The goal of this assignment is to train a Shallow Q-Network (SQN), which is a simplified version of Deep Q-Networks (DQN), to learn and play the game of Tic-Tac-Toe using Q-learning. The task involves implementing and training an SQN to improve its playing capabilities by interacting with the game environment through an experience replay buffer and the Bellman equation for target Q-value computation.

To test the effectiveness of the SQN, it's performance is tested against a rule based opponent with varying levels of skill (smart move player 1).

The implementation was carried out according to the instructions and it shows consistently successful wins against SmartMovePlayer1 = 0. Screenshot of the evaluation script provided is below.

```
_____Smartness 0_____

Testing full game at smartness = 0, seed = 42
Test passed
Testing full game at smartness = 0, seed = 1233
Test passed
Testing full game at smartness = 0, seed = 41
Test passed
Total reward: 3 for smartness: 0
Marks 4

_____Smartness 0.8_____

Testing full game at smartness = 0.8, seed = 42
Test failed
Testing full game at smartness = 0.8, seed = 1233
Test failed
Testing full game at smartness = 0.8, seed = 41
Test passed
Total reward: -1 for smartness: 0.8
Marks 1.5


Total marks: 5.5 out of 8
```

2. Methodology

1. **Storing the state**: Experience data is stored as a tuple of 5:
    a. State: A 9-dimensional input that represents the state of the board at that point.

b. Action: An integer from 0 to 8 to represent the move chosen by the player SQN.
c. Reward: defined as 1 if player SQN wins, -1 if player 1 wins and 0 otherwise.
d. Next State: Next state obtained on taking the action.
e. Done: True if there is a winner or the board is full.

2. **Training**: To perform training, two steps are used: bootstrapping and finetuning.

- **Bootstrapping**: To generate an initial model, the approach of generating initial training data by random number generation as provided in the assignment input is retained. However, to ensure preponderance of training data generated by actual game play, a lower value of 200 experiences has been generated in the bootstrap training phase. Based on this randomly generated training data, a model is generated and saved to a model file named 2021B5A72517G_MODEL.keras.
  During the bootstrap training, a mini-batch of experiences is sampled, and Q-values are predicted for the current and next state.

  Q values are predicted using the bellman equation:

  Q_target = reward + GAMMA*np.max(q values from the next state)

  These values are stored, and gradient descent is performed on the mini batch. The model is then saved, to be re-loaded for subsequent iterations of finetuning.

- **Finetuning**: A cyclical training approach is used to finetune the model using the input generated from using the model generated in the previous training sessions. To implement finetuning, actual experience data needs to be generated by playing against the rule-based agent, and this new data is added to the replay-buffer. As the number of iterations increase, more of the random data from the bootstrapping process is replaced by the actual data. The flowchart below shows the process adopted for iterative fine-tuning.

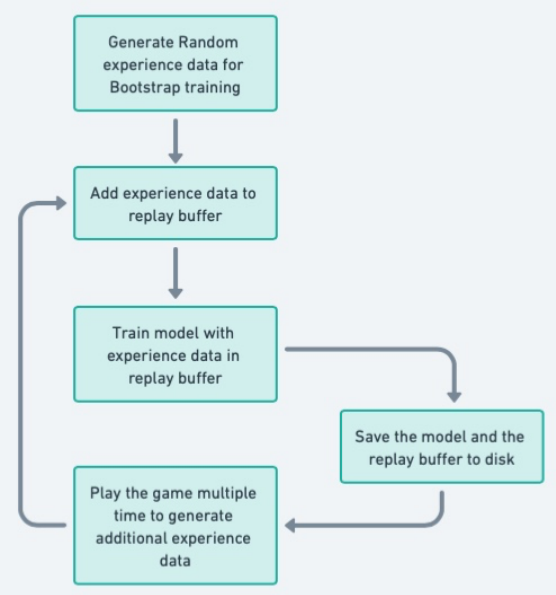


Figure 1: High level view of Training Process

3. **Epsilon greedy method** of selecting the next action:

   In the move method of the player SQN class, the next action is selected using the epsilon greedy policy. Epsilon is set to 0.1. Thus, the model explores and chooses a random action 10% of the times and exploits 90% of the time (chooses the action with the highest q value). To ensure that the model makes correct suggestions for action, a list of all possible valid moves is created. Then Q values corresponding to only these valid actions are considered and in case of the exploitation, the maximum Q value amongst the possible valid actions are selected and returned.

4. **Neural network:** A sequential model is used, with 2 hidden layers (each with 64 neurons) and a linear output layer. The model is compiled with the Adam optimizer and the MSE loss function.

5. **Q-learning:** Q learning is an off-policy reinforcement learning algorithm, where the policy used to generate behavior (the policy used to choose actions) is different from the policy being learned (the target policy we need to optimize). The learned action value function Q directly approximates the optimal action value function, independent of the policy being followed. While the agent explores the environment and chooses actions based on the exploratory policy (epsilon greedy here), the algorithm is learning the optimal policy independent of this behavior. Thus, Q-learning can learn from past experiences even if those experiences were generated by non-optimal behavior.

6. **Replay buffer handling:** The replay buffer is a dequeue structure with experience data with a memory allocation of 10,000 tuples. Each tuple consists of state, action, next-state, reward and a boolean value indicating if the game has ended

or not. Initially the replay buffer is seeded with random **experience** data. thereafter in each finetuning phase, multiple games are played and the experience data added to the replay buffer. If the replay buffer is full, then the oldest items are removed and the new items are appended thereafter. The replay buffer is serialized to the disk as a JSON so that the process of finetuning and playing can happen at different points of time.

7. **Flow chart for the game play:**
   The flow chart below shows how the code provided as a part of the assignment and the implemented classes work together in order to play the game as playerSQN.

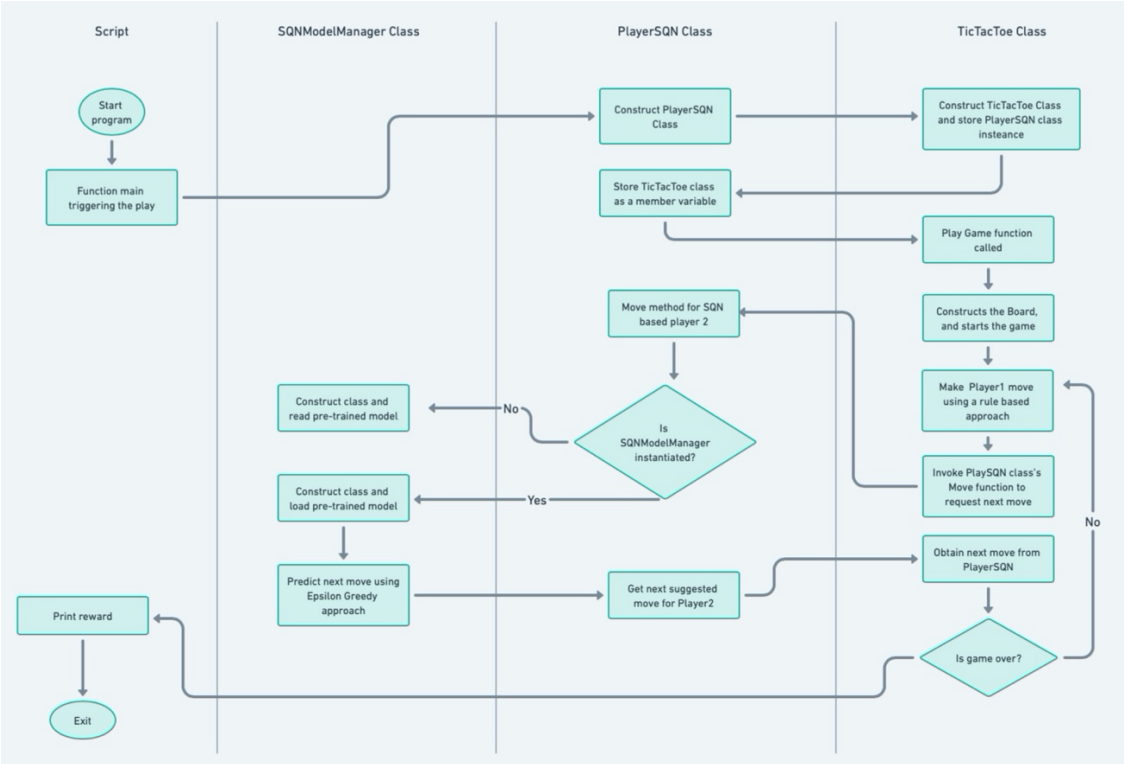


Figure 2: Flowchart for PlayerSQN playing the game

8. **Flow chart for the training process:**
   The process of finetuning needs the game to be played multiple times and the state of the game, actions and consequent rewards are used as additional experience data for use in training. The flowchart below shows the training process.

   Note: The TicTacToe.class is used for playing and is unchanged even in the case of training.

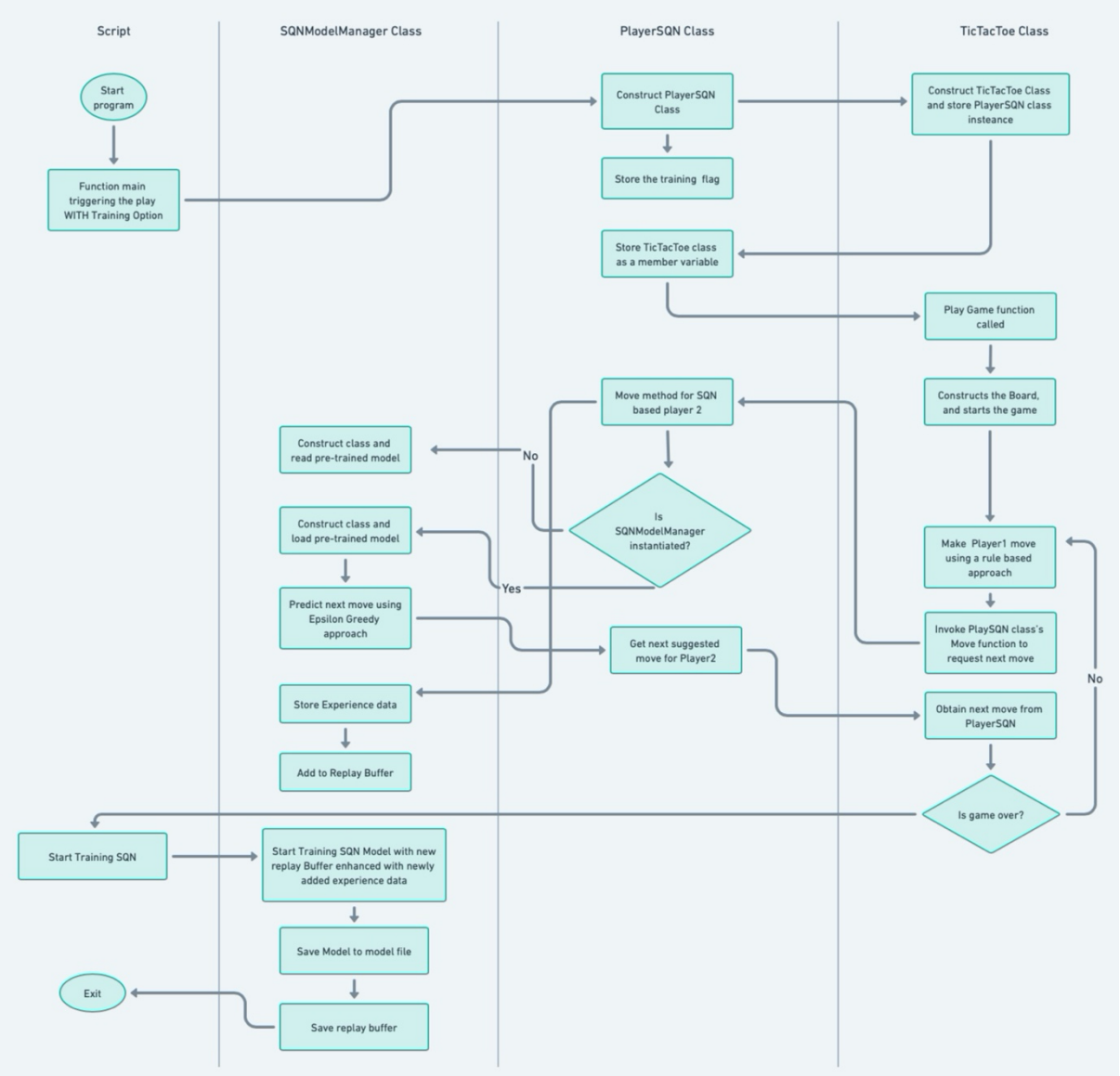


Figure 3: Flowchart for the Finetuning process

9. **Commands to run the code:**

   Bootstrapping: python 2021B5A72517G.py 0 bootstrap
   Finetuning: python 2021B5A72517G.py 0 finetune
   To play: python 2021B5A72517G.py 0

3. Improvements

Improvements made:

1. **Batch processing:** The initial training was based on the approach provided in the assignment input. Couple of complete training were carried out using this

approach. However, the training process was extremely slow. Further reading showed that TensorFlow is capable of and is optimized for batch processing. Hence multiple calls in the for loop was replaced with a single call with batch input. The results of consistent with the once obtained via the iterative call method. The performance improvement was to the tune of factor of 10.

2. **Performing finetuning against different smart move player 1 probabilities**: Gradually increased the smart move player 1 probability from 0 to 0.8 during the process of finetuning.

3. **Decreasing the learning parameter**: Decreased the learning rate (which determines how quickly the model updates weights during training) from 0.001 to 0.0001 to see if the model converges better.

Other suggested improvements (that weren't tested):

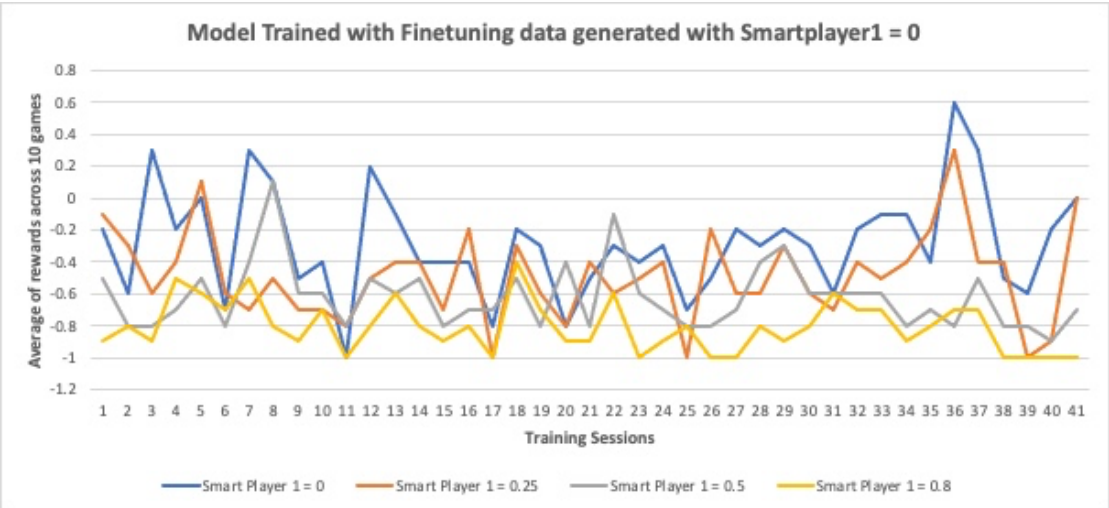
1. Increasing the batch size and number of training sessions.
2. Changing number of epochs: Start with 10 epochs per training phase and increase to 20-30 to allow the model to extract more patterns.
3. Change the replay-buffer size: A larger buffer improves diversity while a smaller one increases recency. Limit the size of the buffer to contain recent, relevant, or high-quality experiences.
4. Vary epsilon values: Start with a higher epsilon, and gradually decay to prioritize exploitation over exploration.
5. Change the reward function: A sparse reward function fails to guide the model toward better strategies.

4. Results

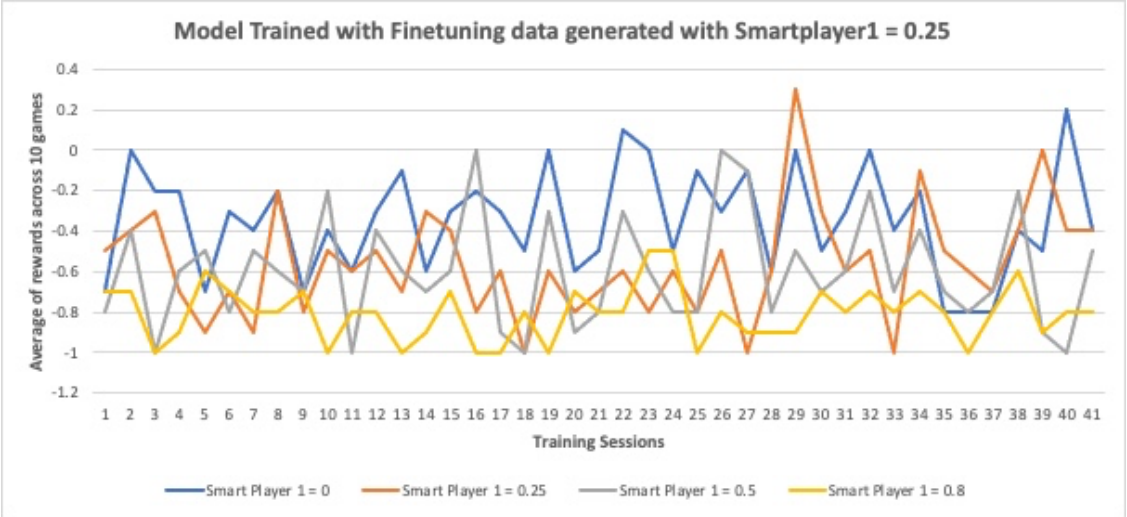
Batch processed data

Model trained with:

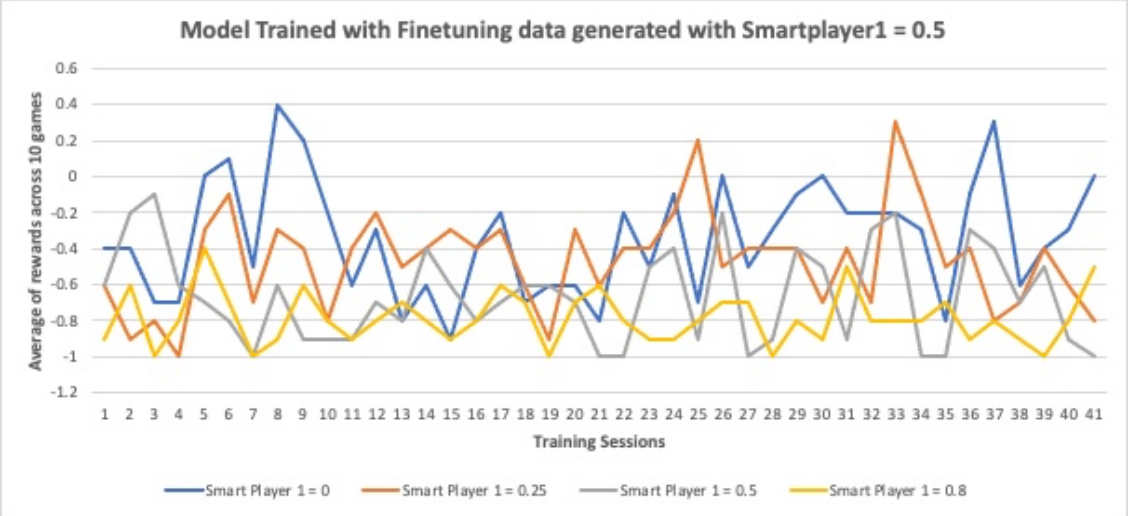
1. Finetuning data generated with smartplayer1 = 0



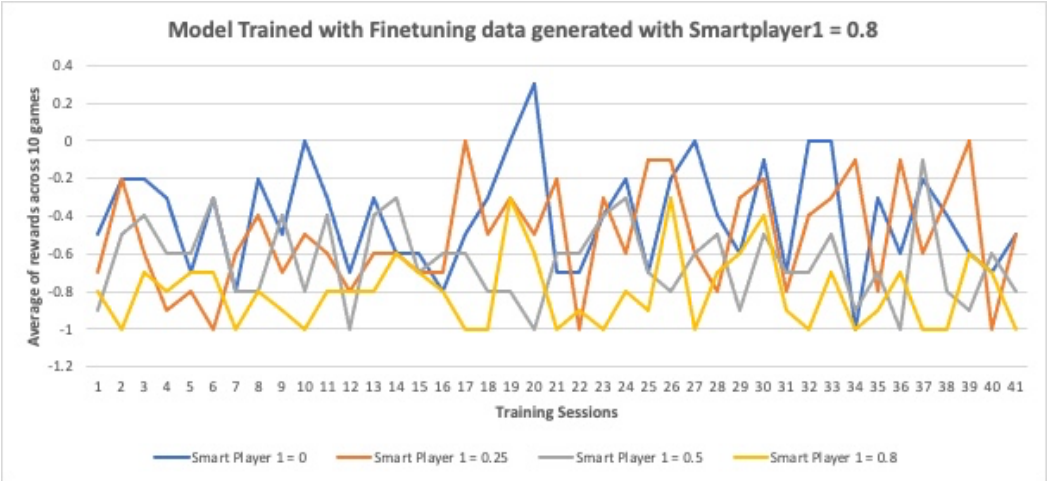
2. Finetuning data generated with smartplayer1 = 0.25



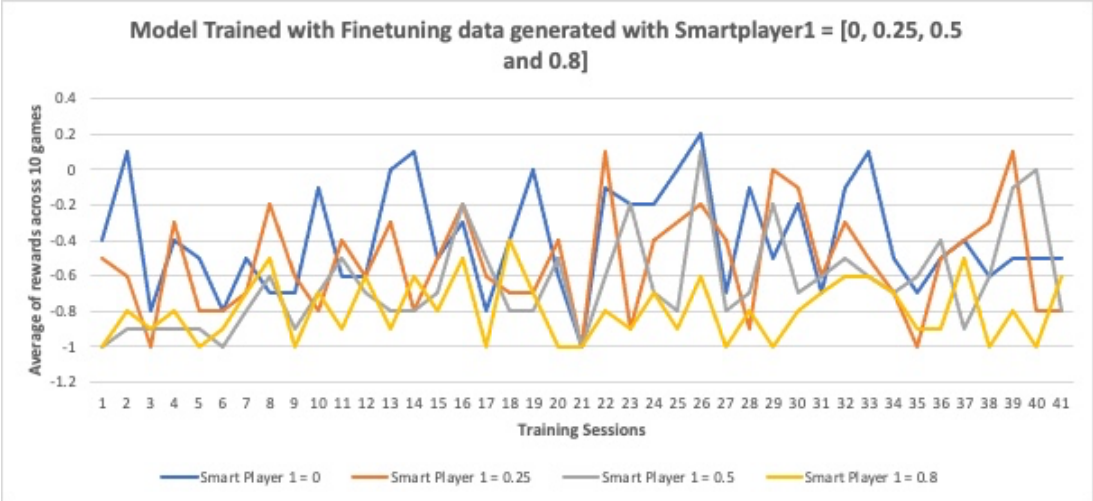
3. Finetuning data generated with smartplayer1 = 0.5



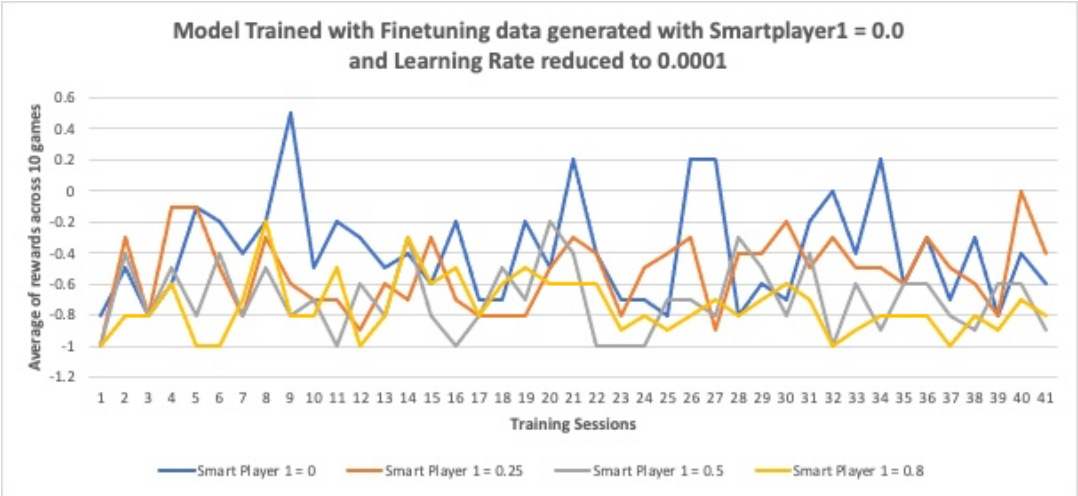
4. Finetuning data generated with smartplayer1 = 0.8



5. Finetuning data generated with smartplayer1 = [0, 0.25, 0.5, 0.8]



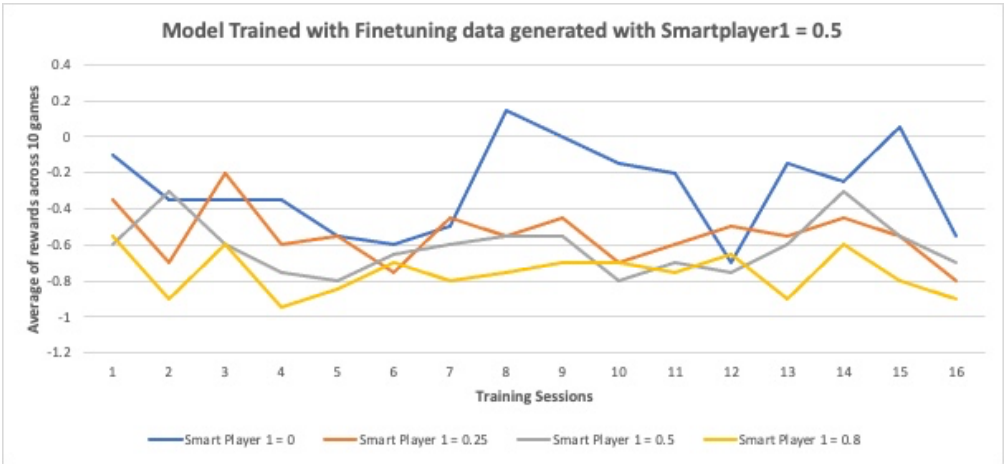
Finetuning data generated with smartplayer1 probability = 0 and learning rate reduced to 0.0001



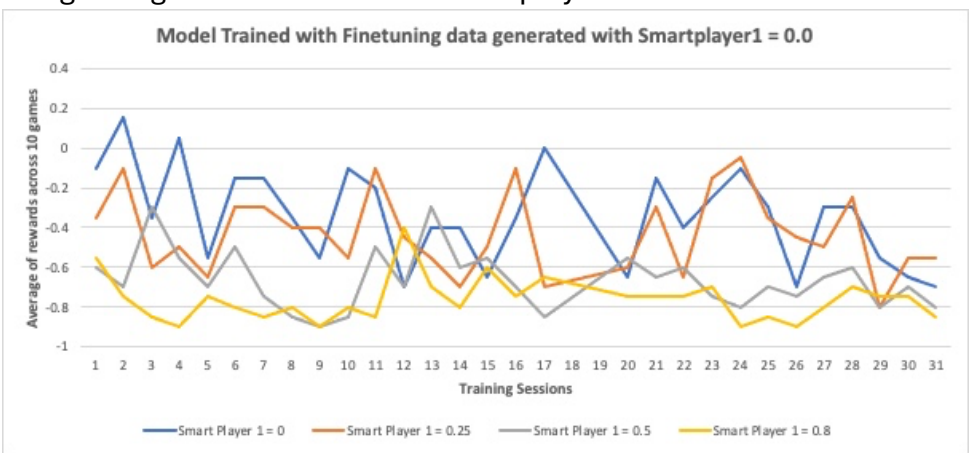
Without batch processing:

Model trained with:

1. Finetuning data generated with smartmoveplayer1 = 0.5



2. Finetuning data generated with smartmoveplayer1 = 0



5. Discussion and Conclusion

- **Accuracy of training**: The model is reasonably accurate especially when playing against SmartMovePlayer1 = 0. It shows a win percentage of 100% against SmartMovePlayer1 = 0 over 20 test cycles.

```
_____________Smartness 0_____________

Testing full game at smartness = 0, seed = 42
Test passed
Testing full game at smartness = 0, seed = 1233
Test passed
Testing full game at smartness = 0, seed = 41
Test passed
Total reward: 3 for smartness: 0
Marks 4

_____________Smartness 0.8_____________

Testing full game at smartness = 0.8, seed = 42
Test failed
Testing full game at smartness = 0.8, seed = 1233
Test failed
Testing full game at smartness = 0.8, seed = 41
Test passed
Total reward: -1 for smartness: 0.8
Marks 1.5


Total marks: 5.5 out of 8
```

- **Probability of Success**: We observe that in all cases, the reward obtained is the highest when we play against smartmoveplayer1 with probability 0 and decreases as we increase smartmoveplayer1's probability from 0 to 0.8.

- **Performance improvement** of approx. 10X when using Batch processing.

- **Convergence**: The model did not converge for any of the smartPlayerProbabilities for training sessions up to 40.