Name: Viraj Sharma
ID: 2020A7PS1011G

Assignment 2 CS F407 - Artificial Intelligence

Problem overview

In this problem we have to train a shallow Q-network to play a tic tac toe game against a smart player and the objective is to maximize the number of wins against the smart player. It has been given the name shallow as it will contain few hidden layers in its neural network. We solve this problem in three phases: The first phase is the data preparation phase,
second phase is the training phase of the network and in the final phase we test the trained network against the smart player

Data preparation phase: In this phase we define an untrained neural network and allow it play several thousand games against the smart player and collect the data in the form :
(state,action, reward, next_state, done).
I allowed for mild exploration during the preparation phase by setting the epsilon value to a number between 0 and 1. The data created from this interaction between the untrained network and the smart player is then stored in a buffer which will later be used to train the network.

Training phase of the network: In this phase we take the data from the buffer, define input and target batches, allow the network to generate the values of the next state and use that to define the target batch. Once the target batch has been defined we feed the input batch and compare the output of the network with the network with the target batch which will then help me in backpropagating the loss across the neural network thereby improving it.

Methodology

For this assignment I have trained 8 different neural networks differing in their number of layers, number of episodes used for data generation and epsilon values.
In the data preparation phase I used the untrained neural network and allowed it to play with the smart player for thousands of episodes **(3000,6000,12000,24000,48000** episodes). This created a large amount of training data as the number of data points varied from **8000 to 200000+** data points. The training data in the buffer was of the form (state,action,reward,next_state,done). For each episode the untrained neural network would play with the agent till the game was over and store the data points in the buffer and then reset the game to generate data points for the next episode.

In the training phase of the model I divided the training data from the buffer into batches similar to the way given in the NN_training.py file. I have used PyTorch to train the neural network which required me to convert the training data into PyTorch tensors so that the neural network could ingest the data.

The batch size for the training data is 32. The input data and targets are initially defined as a matrix of zeros do size (32,9). A minibatch is randomly sampled from the training set and (state,action,reward,next_state,done) is iterated over this minibatch to define the input data and target data.
The input data is equal to the state whereas for the target value, the **q_value(state)** and **q_value(next_state)** have to be calculated using the neural network which are then fed into it.
Q_target = reward + gamma*q_value(next)

The reason for using **q_values(state)** and then altering **q_values(state)[action] = q_target** is so that when the model is fed the input data , the loss of the output data will be in the form **[0,0,0….0,Q_value(state,action)-Q_target(state,action),0,0,0,...]**. This is a clever way to calculate the loss between **Q(s,a)** and **Q_target(s,a)**.

After calculating the loss the gradients are calculated with respect to the loss and then back propagated across the network to make it more refined.

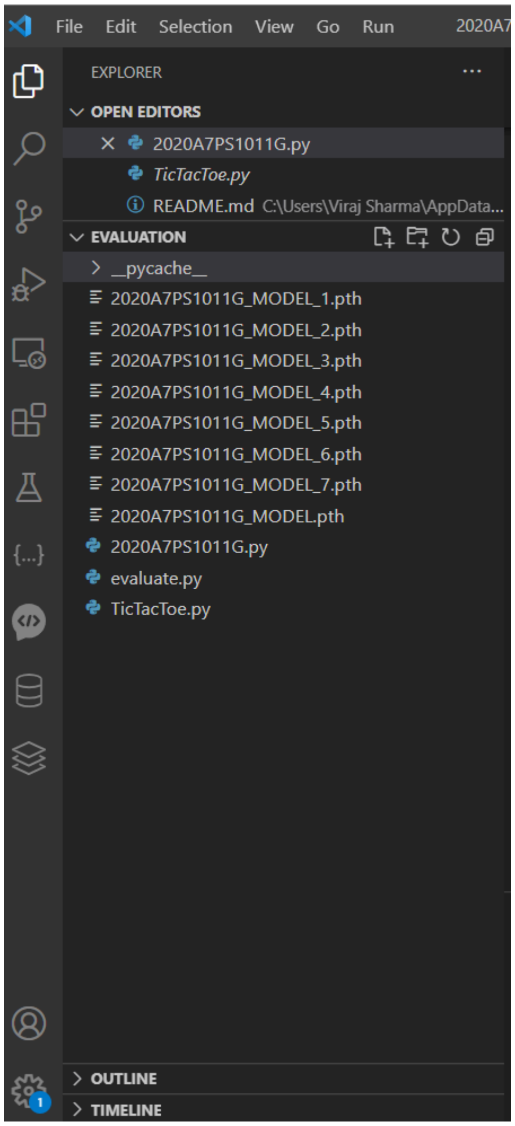
After the model has been trained the model is then saved to a file. The model is then loaded from this file into the PlayerSQN class which is then tested to play against the smart player.

I have used three different neural network architectures which are of the form:
NN1: Input: 9 vector input
Layer1:64 neurons ReLU activation
Layer2:64 neurons ReLU activation
Output: 9 vector output linear
NN2: Input: 9 vector input
Layer1:32 neurons ReLU activation
Layer2:32 neurons ReLU activation
Output: 9 vector output linear
NN1: Input: 9 vector input
Layer1:128 neurons ReLU activation
Layer2:128 neurons ReLU activation
Output: 9 vector output linear

As I said earlier I have defined 8 different models using the three neural networks above and with varying training episodes
The description of 8 models is as follows:

1. **NN1, 6000 episodes**
2. **NN1, 3000 episodes**
3. **NN1, 12000 episodes**
4. **NN1, 48000 episodes, with decaying epsilon at the rate 1% per episode**
5. **NN1, 24000 episodes**
6. **NN2, 12000 episodes**
7. **NN3, 12000 episodes**
8. **NN3, 24000 episodes**

Proof of 8 models:

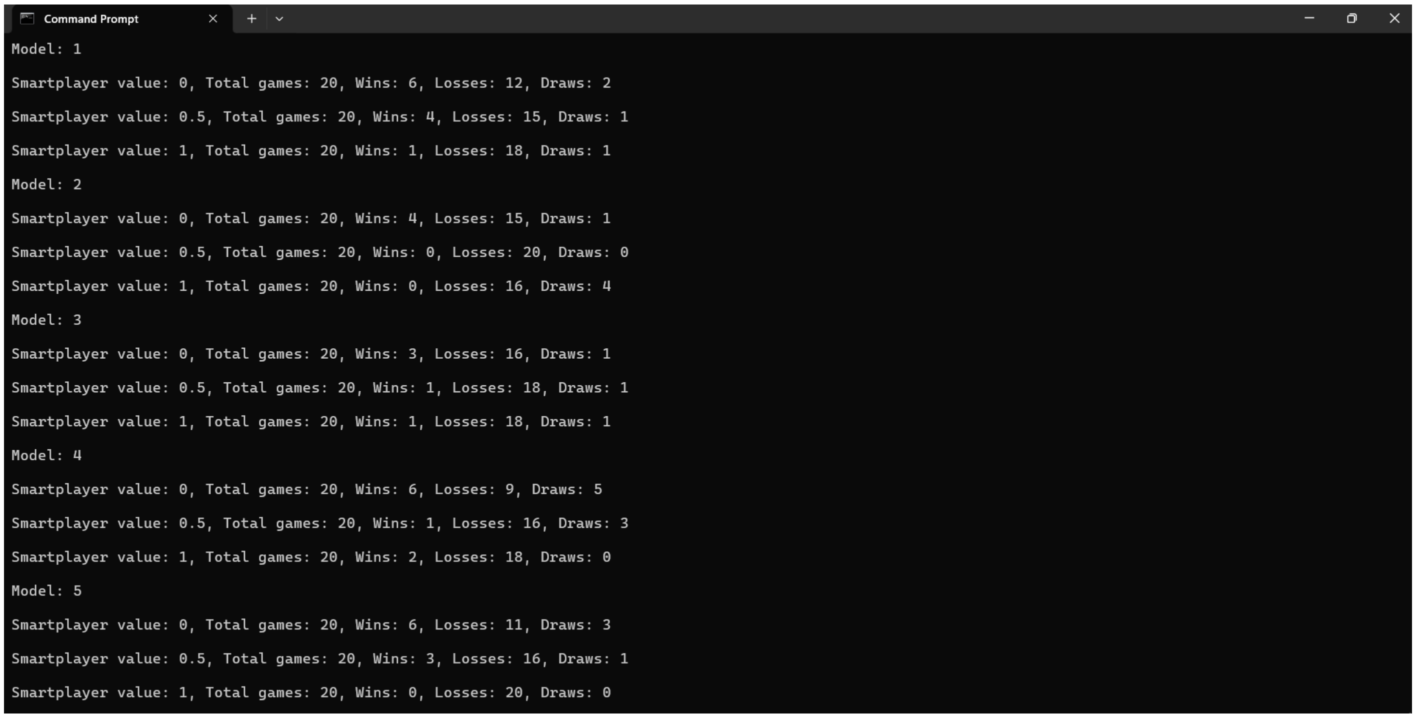


2020A7PS1011G.py
TicTacToe.py
README.md C:\Users\Viraj Sharma\AppData...

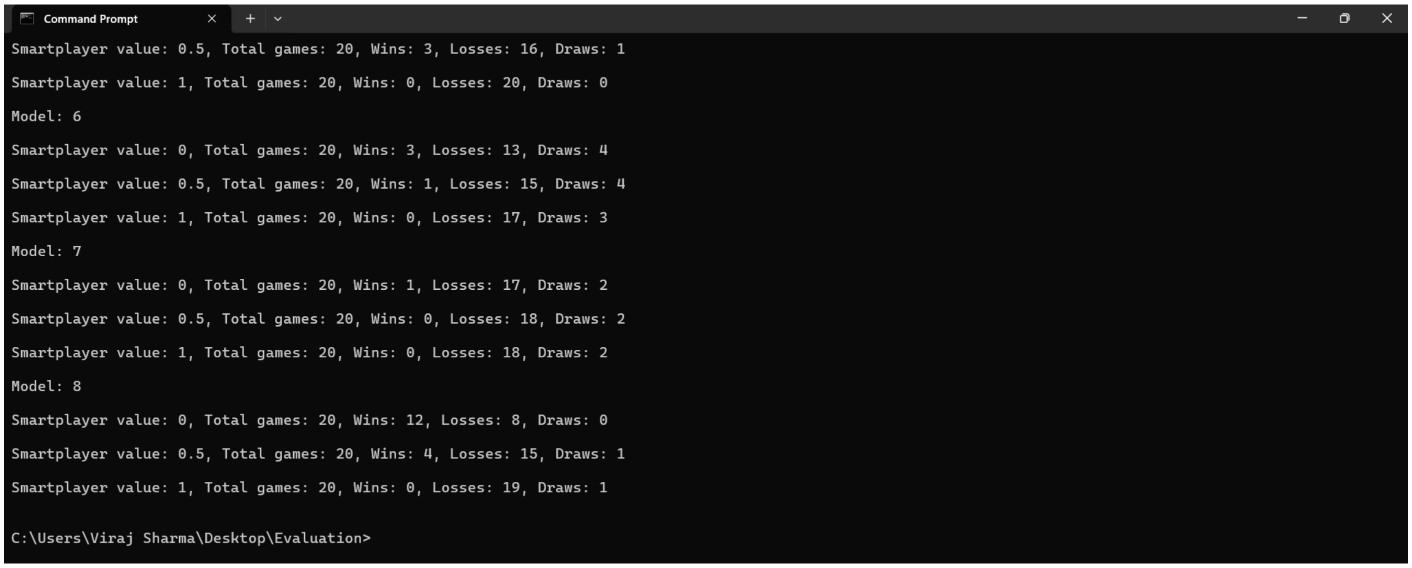
__pycache__
2020A7PS1011G_MODEL_1.pth
2020A7PS1011G_MODEL_2.pth
2020A7PS1011G_MODEL_3.pth
2020A7PS1011G_MODEL_4.pth
2020A7PS1011G_MODEL_5.pth
2020A7PS1011G_MODEL_6.pth
2020A7PS1011G_MODEL_7.pth
2020A7PS1011G_MODEL.pth
2020A7PS1011G.py
evaluate.py
TicTacToe.py

Results

Game statistics of the 8 models with smart player values = (0, 0.5, 1):



```
Model: 1
Smartplayer value: 0, Total games: 20, Wins: 6, Losses: 12, Draws: 2
Smartplayer value: 0.5, Total games: 20, Wins: 4, Losses: 15, Draws: 1
Smartplayer value: 1, Total games: 20, Wins: 1, Losses: 18, Draws: 1
Model: 2
Smartplayer value: 0, Total games: 20, Wins: 4, Losses: 15, Draws: 1
Smartplayer value: 0.5, Total games: 20, Wins: 0, Losses: 20, Draws: 0
Smartplayer value: 1, Total games: 20, Wins: 0, Losses: 16, Draws: 4
Model: 3
Smartplayer value: 0, Total games: 20, Wins: 3, Losses: 16, Draws: 1
Smartplayer value: 0.5, Total games: 20, Wins: 1, Losses: 18, Draws: 1
Smartplayer value: 1, Total games: 20, Wins: 1, Losses: 18, Draws: 1
Model: 4
Smartplayer value: 0, Total games: 20, Wins: 6, Losses: 9, Draws: 5
Smartplayer value: 0.5, Total games: 20, Wins: 1, Losses: 16, Draws: 3
Smartplayer value: 1, Total games: 20, Wins: 2, Losses: 18, Draws: 0
Model: 5
Smartplayer value: 0, Total games: 20, Wins: 6, Losses: 11, Draws: 3
Smartplayer value: 0.5, Total games: 20, Wins: 3, Losses: 16, Draws: 1
Smartplayer value: 1, Total games: 20, Wins: 0, Losses: 20, Draws: 0
```



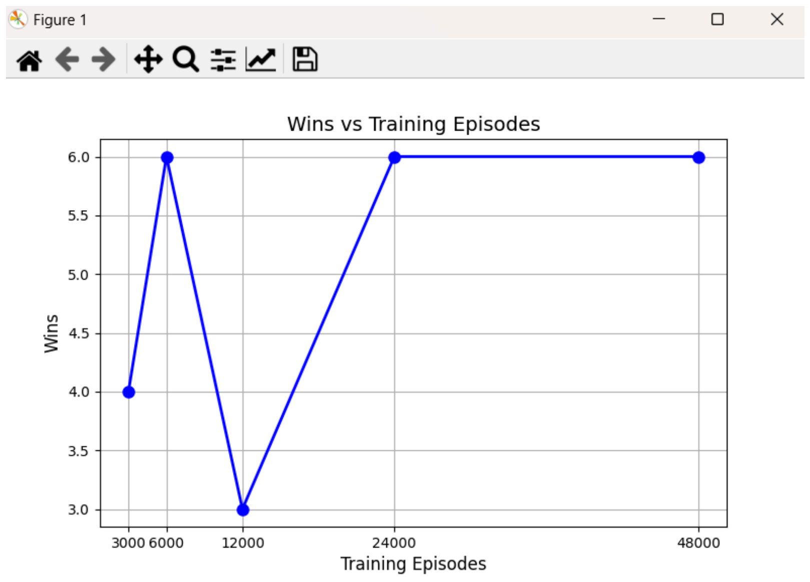
```
Smartplayer value: 0.5, Total games: 20, Wins: 3, Losses: 16, Draws: 1
Smartplayer value: 1, Total games: 20, Wins: 0, Losses: 20, Draws: 0
Model: 6
Smartplayer value: 0, Total games: 20, Wins: 3, Losses: 13, Draws: 4
Smartplayer value: 0.5, Total games: 20, Wins: 1, Losses: 15, Draws: 4
Smartplayer value: 1, Total games: 20, Wins: 0, Losses: 17, Draws: 3
Model: 7
Smartplayer value: 0, Total games: 20, Wins: 1, Losses: 17, Draws: 2
Smartplayer value: 0.5, Total games: 20, Wins: 0, Losses: 18, Draws: 2
Smartplayer value: 1, Total games: 20, Wins: 0, Losses: 18, Draws: 2
Model: 8
Smartplayer value: 0, Total games: 20, Wins: 12, Losses: 8, Draws: 0
Smartplayer value: 0.5, Total games: 20, Wins: 4, Losses: 15, Draws: 1
Smartplayer value: 1, Total games: 20, Wins: 0, Losses: 19, Draws: 1

C:\Users\Viraj Sharma\Desktop\Evaluation>
```

From the above statistics Model 8 is the most superior model:
Model 8 is **NN3** with **24000 episodes**
Wins + Draws value = (12,5,1) with respect to **smart player value = (0,0.5,1)**

Since models 1 to 5 are trained using the same neural network architecture we can compare the win rate against the number of episodes:

(We will assume the smart player value to be 0 as it will give us a better graphical representation):



I will therefore be using model 8 for the submission of this assignment

Challenges

As you can see I have done quite a bit of experimentation with the models , could have used more models but the only thing restricting it was the training time as the number of episodes and the number of hidden layers increased the amount of time taken to train the model increased substantially. I also noticed that increasing the number of epochs did not result in better model performance but increased the model loss after epoch 30 or 40.

Conclusion

The model can be improved by trying various different neural network architectures instead of a limited number of architectures as above. Expanding the neural network architecture search space will surely give superior results but needs a lot of computation and time. Another improvement that can be suggested is to reduce the number of duplicate data points in the training data generation which will allow the model to be exposed to all sorts of possible paths instead of a subset of data points. Tic Tac Toe has around **25000+** possible legal games so minimizing duplicate games is important for superior performance of the model.