Week 3: Reinforcement Learning

Introduction

After playing around with your TicTacToe game from last week, you probably realised that it is not actually very good and incredibly easy to win! This is where **Artificial Intelligence** can help improve upon this. We will use something called reinforcement learning to train our computer to make better decisions than just random ones!

We will be building on our previous work from the past 2 worksheets. **Please move your 'sheet1.py' and 'sheet2.py' files into the same folder that your are currently working in!**
If you did not complete either of the two worksheets then no worries - I have provided solutions for you. Please change the name from 'sheet2solution.py' to 'sheet2.py' and the same for sheet1 so all the code works smoothly!

The process of how the reinforcement learning works will be explained as you move through the tasks, but first it is important to understand the 'dictionary' data structure in Python.

Dictionaries

DISCUSSION: Has anybody used a dictionary before? If so, what advantages do they have over other data structures (lists, pairs ...)?

A dictionary is a way of storing pairs in terms of 'keys' and 'values'. In Python, a dictionary in Python may look like:

```
ShoppingBasket = {'Apples':5, 'Bananas':2, 'Carrots':10}
```

If we wanted to find out how many Bananas were in our basket we could call the following and it would output 2:

```
ShoppingBasket['Bananas']
```

If we were unsure on whether 'Tomatoes' were in our shopping basket, calling the above code would cause an error, instead use the following (this would return 0 if 'Tomatoes' is not in the basket, and the value if they are present):

```
ShoppingBasket.get('Tomatoes',0)
```

Artificial Intelligence: Reinforcement Learning

To make our game better, we are going to introduce a method of reinforcement learning called 'Q-learning'.

There are 3 important aspects of reinforcement learning:

1. State
2. Action
3. Reward

In our game of TicTacToe we will set the following:

1. State: The board in string form, e.g 'X OX OX O'
2. Action: The next move we make (out of the possible moves)
3. Reward: +1 for the winner, -1 for the loser, 0 if the game is a tie

How it works:
We will first train our computer (by getting it to play lots of games). Within each game the computer remembers the states and actions it has played (you have already implemented this within 'self.States'). If the computer wins then it gives these states a positive reward, and if it loses it will give it a negative reward. The pairs of states and rewards will be stored in a dictionary (named 'StateValues'). This is our learning stage!

Once the learning is complete, instead of making random moves to play, the computer will look at the dictionary and choose the best possible move that it can play.

If the following game is played:

```
- - - - - - -            - - - - - - -            - - - - - - -
| X |   |   |            | X |   |   |            | X |   | X |
- - - - - - -            - - - - - - -            - - - - - - -
|   |   |   |            | O |   |   |            | O |   |   |
- - - - - - -            - - - - - - -            - - - - - - -
|   |   |   |            |   |   |   |            |   |   |   |
- - - - - - -            - - - - - - -            - - - - - - -
- - - - - - -           - - - - - - -
| X |   | X |           | X | X | X |
- - - - - - -           - - - - - - -
| O |   |   |           | O |   |   |
- - - - - - -           - - - - - - -
|   |   | O |           |   |   | O |
- - - - - - -           - - - - - - -
```

Then we will have:

```
StatesX = ['X        ', 'X XO     ', 'XXXO    O']
StatesO = ['X  O     ', 'X XO    O']
```

Player X won and hence this player will be given the positive reward. We will give player O a negative reward so perhaps in the future it knows that playing that move can lose the game.

And so afterwards our State-Values will look like this (provided this is our first game that we play):

```
StateValuesX = {'X        ': 0.008, 'X XO     ': 0.04, 'XXXO    O':
    0.2}
StateValuesO = {'X  O     ': -0.02, 'X XO    O': -0.1}
```

Do not worry too much about why the numbers are specifically those numbers (for a better explanation see the 'Q-Function' section at the end). What is important is that the boards that get us closer to winning have a higher score and vice versa for losing.

Tasks

Task 1: Learn

Our first task is to write the outer function for the learning of our computer. In the sheet3.py file, complete the 'Learn(n)' function.

INPUT: n - The number of games we want our computer to play
OUTPUT: StateValuesO - Our dictionary of states and rewards which is updated with each game. We will only output the State-Value dictionary for Player O as this is our opponent and we will be Player X.

The steps of this function which you need to implement are:

1. Initialise State-Value dictionaries for both X and O
2. For each learning iteration:
    (a) Play a game
    (b) Update the State-Values for both the winner and the loser (a function has been provided for you - see below)

3. Return the State-Value dictionary for our opponent (the computer - player O).

UpdateSVs: A function has been provided for you which updates the values of the state values. It inputs the current State-Value dictionary, the list of states (self.States) and a score (+1 for winner, -1 for loser).

If you would like to test this function, then you can uncomment the 'Play(1,0)' at the bottom of the code. For now, keep the 'TrainingN' low as you have to play each game! TestingN will have to be 0 for now as we have not implemented the 'Human-VsComputer' game yet.

Don't forget to add what you have learnt about dictionaries to your Python cheat sheet from week 1 if you want!

Task 2: Discussion

For AI to work well, you often need to do a lot of training. How could we change our game play so that we do not have to play each training game?

Task 3: BestTurn

A function called 'BestTurn' has been partially provided for you below - you need to fill in the bits with the '### FILL ###'. Please copy and paste this function into your **SHEET 2** class! Note: be careful with indentation when copying and pasting - make sure everything is right!

This function will be used by our computer to calculate the best move possible for it to take. This function will be used in both the learning stage (explained later) and also when we want to play against our optimised computer.

How it works:

1. The computer looks at the current board
2. For each possible move the computer can make (for each empty position), the computer calculates what the board would look like if it made that move. Note: here we need to use 'self.Board.copy()', otherwise we will accidentally update our board
3. It looks at the 'StateValues' input and stores the value of that board (use the dictionary 'get' function here)
4. After all possible moves have been iterated through and each value has been stored, the computer finds the highest value. If there are multiple moves that would give the same value, then the computer should choose a random choice out of those possibilities. *Hint: sheet1.*

5. Complete the move for the position just computed.

You need to complete only step 3 where it says ### FILL ###.

```python
def BestTurn(self,StateValues):
        # For each possible move, get the new state and store the values in a list
        Values = []
        for Index in self.EmptyCells:
            # Place a marker in a copy of the board
            Board = self.Board.copy()
            Board[Index] = self.Turn

            ### FILL ###
            # 1) Turn the board into a string
            # 2) Get the corresponding value from the State-Values for that state
            # 3) Add the value to the list 'Values'
            ############

        # Find the indices of the highest values
        Indices = s1.MaxIndices(Values)
        # Choose a random index if multiple (if there is just one then the function will choose that one)
        Index = s1.GetRandomElement(Indices)
        # Get the corresponding move to make
        Position = self.EmptyCells[Position]
        # Make the move on the actual board (not the copy)
        self.Board[Position] = self.Turn
```

Task 4: HumanVsComputer

So far, we only have a function so that we can play against a random computer. Let's build a function so that we can play against a computer which always chooses the best turn possible when it is given a State-Value dictionary.

We will need to input our State-Values for our computer as our BestTurn function relies on them. Although we have two State-Value dictionaries in our Learn function, we will end up only putting one into all these functions as they are only used for one player at a time.

1. Copy the template below over into your **SHEET 2** class.
2. Complete the function - this will be similar to your 'HumanVsRandom' function, except when it is Player O's turn we want to utilise our 'BestTurn' function.
3. Add a Boolean input to the function called 'PrintGame' and edit your code so that the game (and outcomes) are only printed out when this is set to True.

CODE TEMPLATE:

```python
def HumanVsComputer(self,StateValues):
    return
```

Task 5: LearningTurn

For our AI to learn, we need it to explore a bit. If, during our training games, we get our agents to always choose the best options then it will constantly play the same games over and wont explore new options. Your job is to write a function for this. Use a random number so that 30% of the time our player takes a random turn and then 70% of the time they take the best turn.

A template has been given for you. Please copy and paste this into your SHEET 2 class and complete it.

```python
def LearningTurn(self,StateValues):
    return
```

Task 6: ComputerVsComputer

Now,to make our learning more efficient - instead of us playing each learning game, we are going to create another computer 'agent' that will play against our computer. Our final piece of the puzzle is to build the function in which these two computers play against each other.

CODE TEMPLATE:

```python
def ComputerVsComputer(self,StateValuesX,StateValuesO):
    return
```

1. Copy over the code template above into your **SHEET 2** class.
2. Write the 'ComputerVsComputer' function similar to that of HumanVsComputer and HumanVsRandom, except with every move we want the agent to take a 'LearningTurn' with their corresponding StateValues (note both are inputs to the function).
3. Add a Boolean input to the function called 'PrintGame' and edit your code so that the game (and outcomes) are only printed out when this is set to True.
4. Change your 'Learn' function in sheet 3 so instead of playing a game using 'HumanVsRandom', you use 'ComputerVsComputer'.
5. Comment out the print statement for 'NEW GAME' (if you have it) in the __init__ function for the TicTacToe class.

Once this is complete, then you can hopefully run the 'Play' function in the sheet4.py file. Try having 1,000,000 training iterations and setting testingN = 10. If all the steps have been followed, then all our functions will be tied together and our complete learning stage will be complete.

Discussion

How much better is our game play now?

What happens if we change self.Turn to initialise as 'O' so our computer plays first. Is this now harder or easier to win?

Has this changed your mind at all about how artificial intelligence might work?

Topic Suggestions

If you have any topics which you would like covered in these sessions then feel free to let me know by clicking here.
This may be content that isn't covered within your units which you would find useful or a particular topic within your current units that you feel you would benefit from more help with!

Q-Function: Updating the SVs

To update the state-values, we use something called a Q-function (also known as the Bellman equation).

Let's denote the function by $V(s_t, a_t)$, where s_t and a_t are the state and action at step t. Note that in our case, if we have $s =$'X O XX O' and $a = 1$ (i.e player O goes plays in the 1st position), we combine this pair to make a new 'temporary' state $s =$'XOO XX O', and it is this temporary state which we look up in our State-Value dictionary, instead of the state-action pair.

We update the values according to the following function:

$$V(s_t, a) = (1 - \alpha) \cdot V(s_t, a) + \alpha \cdot V(s_{t+1}, a) \tag{1}$$

We set α to be our learning parameter, which is set to be 0.2. This determines how quickly we update each state - so currently for each update we take 80% of our previous value and 20% of our new value and sum them.

To use this iterative function in practice we start with our winning/losing state (depending on whether we are updating X's or O's SVs). We give this state a 20% update of 1 or -1 depending on the score of the game. Then, we iterate backwards through the states of the game using our function to update the other states also.