AI Programming (IT-3105) Project Module # 3:
On-Policy Monte Carlo Tree Search for Game Playing

Purpose:

- Apply the general-purpose Monte Carlo Tree Search (MCTS) system – written for the previous project module – to a more complex 2-person game.
- Learn to employ a neural network as the target policy (and behavior/default policy) for on-policy MCTS.
- Gain proficiency at training policy networks with MCTS and then re-deploying them in head-to-head competitions with other networks.

1 Assignment Overview

In this project module, you will reuse your generic MCTS system to play Hex, a board game with simple rules but myriad possible board configurations. This explosion in possible game states precludes table-based RL and necessitates the use of function approximators (e.g. neural nets) for the value function and/or policy. In this case, we focus on the policy (i.e., the mapping from game states to a probablity distribution over the possible actions) as the function in need of neural-net approximation.

Runs of MCTS will provide target probablity distributions for training the policy network, which will fulfill the roles of both *target policy* and *behavior policy*, thus yielding an *on-policy* version of MCTS. Once trained via MCTS, the policy network will participate in competitions against other Hex-playing agents.

2 Introduction

In the past few years, the field of Artificial Intelligence (AI) has witnessed several ground-breaking achievements that combine reinforcement learning (RL) with Deep Neural Networks (DNNs). Most prominent among these is the Alpha Go system (Silver et. al., 2016) that integrates Monte Carlo search, deep nets and RL to play the game of goat a level above (and beyond) that of the world's top players. A precursor to Alpha Go was an AI agent (Mnih et. al., 2015) that achieved human-level (or above) performance on 29 of 49 Atari games, all using the same general combination of RL and deep nets. These projects, involving large research teams at Google DeepMind, have rejuvenated interest in RL for complex tasks. In addition, the general Deep RL (DRL) architecture used in these systems is now being extended to play other games (such as chess and shogi), and handle other tasks, such as drug discovery and diagnostic image analysis.

Although attacking goor a collection (even a small one) of Atari games is beyond the scope of this class, we can investigate a scaled-down situation involving a) an easier game, and b) a simpler combination of tools. The game is Hex, and the architecture still involves RL and neural networks, but the nets do not need to be particularly deep.

As discussed in the lecture notes, MCTS may incorporate different levels of learning. In the previous project module, only the bare minimum of MCTS components were adaptive: the tree policy. In this project, we choose an on-policy approach (i.e., the target policy and behavior policy are the same), and we choose to implement that policy as a neural network in Tensorflow. MCTS can then provide training cases for that policy network, in the same spirit as Google DeepMind's use of MCTS to train deep networks for playing go, chess and shogi. That same network, in fulfilling its role as the behavior policy, will guide rollout simulations in MCTS,

This is a challenging assignment that requires a tight integration between two complex programs: Tensorflow and your MCTS system.

3 The Game of Hex

Hex, also known as *Con-tac-tix*, is played on a diamond-shaped grid with hexagonal connectivity: nodes on the interior have 6 neighbors. As shown in Figure 1, two players (black and red) alternate placing single pieces on the grid. Placed pieces can never be moved or removed. Each player "owns" two opposite sides of the diamond and attempts to build a connected chain of pieces between those two sides; the first player to do so wins. It can be proven mathematically that ties are not possible: a filled Hex board always contains at least one chain between opposite sides. In Figure 1, black owns the northwest and southeast sides, while red owns the northeast and southwest sides. The bottom right of the figure shows black's winning chain.

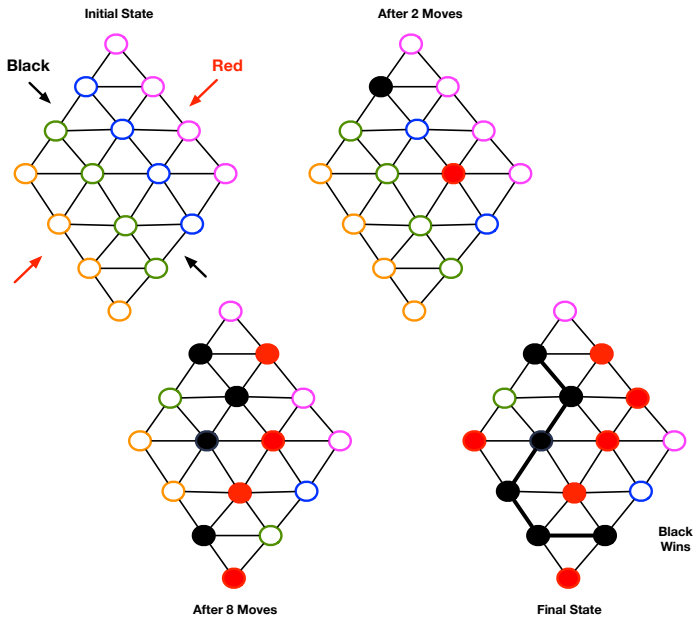

Figure 1: A 4 x 4 Hex Board in 4 different states of play. Colors of empty cells indicate their row index in a 4 x 4 array representation (discussed below). Arrows indicate the borders owned by each player.

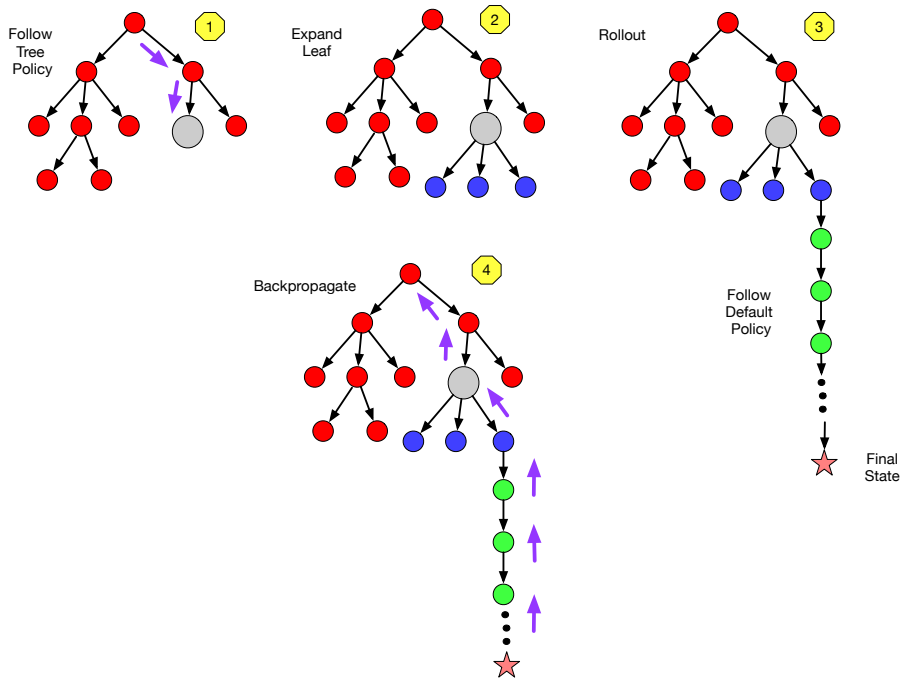

Figure 2: Overview of the main elements of MCTS search.

4 On-Policy Monte Carlo Tree Search

In the terminology of Reinforcement Learning (RL), *on policy* means that the policy employed for exploring the state space (a.k.a. the *behavior policy*) is also the policy that is gradually refined via learning (a.k.a. the *target policy*). In MCTS, the behavior policy is also known as the *default policy*: the policy used to perform rollouts from a leaf node to a final state in the MCTS tree. In MCTS, a third policy, the *tree policy* controls search from the tree's root to a leaf. In summary, on-policy MCTS involves 1) a tree policy and, 2) a target policy that is also used as the behavior/default policy for managing rollouts.

In this project module, a neural network – a.k.a. the *actor network* (ANET) – constitutes the target policy. It takes a board state as input and produces a probability distribution over all possible moves (from that state) as output. The main goal of On-Policy MCTS is to produce an *intelligent* target policy, which can then be used independently of MCTS as an actor module.

4.1 The Reinforcement Learning (RL) Algorithm

This RL procedure involves three main activities:

1. Making *actual moves* in a game, with many such games being played. Each completed actual game constitutes an *episode* in this form of RL.
2. Making *simulated moves* during MCTS. These will be refered to as *search moves* in this document, with any sequence of search moves leading from an initial to a final state deemed a *search game*. Each actual move taken in an episode will be based upon hundreds or thousands of search games in the MC tree.

3. Updating the target policy via supervised learning, where training cases stem from visit counts of arcs in the MC tree.

Just like normal Monte Carlo versions of RL, MCTS RL involves episodes of game play, and no updates to the target policy occur until the end of each episode. However, MCTS packs a lot of (search) activity into each episode. Each actual move in an episode is based on many (hundreds or thousands) of search games, each of which forms one vine in the MC tree. Each search game begins at the root of the MC tree and uses the tree policy (which is usually highly exploratory) to make a series of moves from the root to a leaf of the tree. From the leaf, it employs the target policy (realized by the ANET and also reasonably exploratory) to determine all rollout moves along a path to a final game state. After reaching a final state, the MCTS algorithm backpropagates the reward signal (based on which player wins) along the vine from final state to root. As described in the lecture notes, this causes updates to visit counts along with Q(s,a) and u(s,a) values. Figure 2 summarizes the MC tree-growing process that underlies each RL episode.

Statistics (visit counts) accumulated over all of the MC tree vines (also described in the lecture notes) then provide the basis for an intelligent choice of a single actual move from the current state (s) in the current episode. In addition, the normalized visit counts serve as a target probability distribution (D) over the set of possible actions. The pair (s,D) then becomes a training case for ANET. Each such case is stored in a Replay Buffer (RBUF), and at the end of each episode (i.e. each actual game), a random minibatch of cases from RBUF is used to train the ANET. Figure 3 summarizes this interaction between the MC tree and the target policy embodied in the ANET.

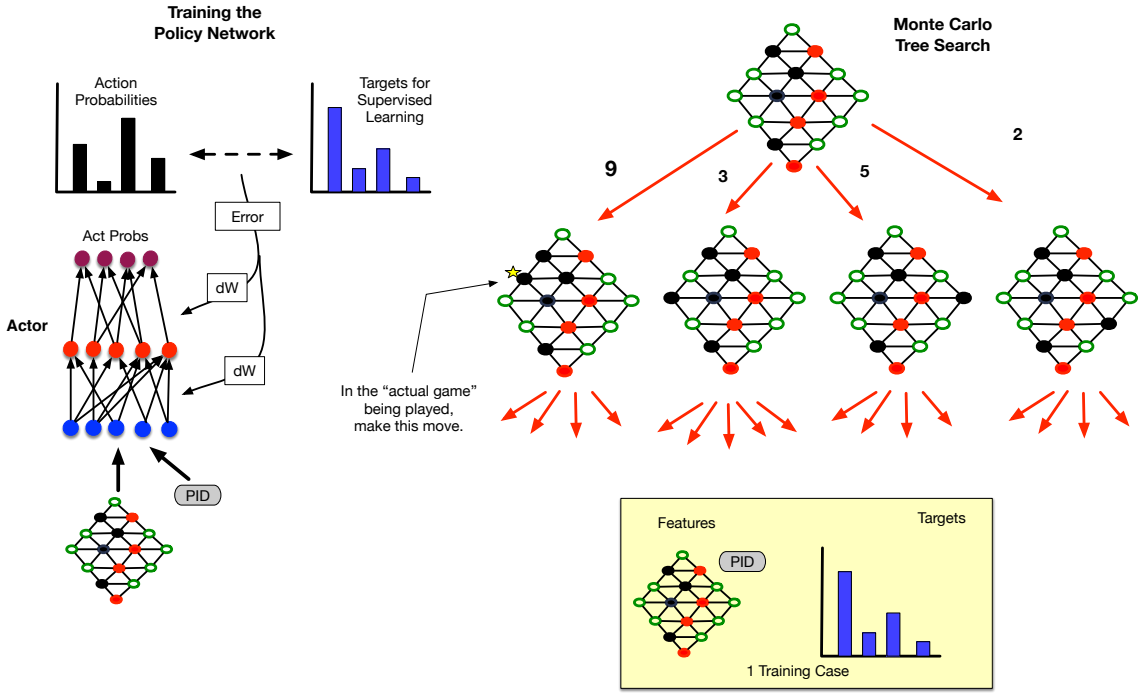

Figure 3: Overview of supervised learning of the target policy network based on results of Monte Carlo tree search (MCTS). Numbers on MCTS branches denote visit counts accrued during the multiple search games associated with each single move in the actual game. Each case (yellow box) is stored in the Replay Buffer and used for training at the end of each episode; an episode involving m moves should add m cases to the Replay Buffer. PID = Player Identifier.

Pseudocode for the entire algorithm appears below:

1. i_s = save interval for ANET (the actor network) parameters
2. Clear Replay Buffer (RBUF)
3. Randomly initialize parameters (weights and biases) of ANET
4. For g_a in number_actual_games:
   (a) Initialize the actual game board (B_a) to an empty board.
   (b) $s_{init} \leftarrow$ starting_board_state
   (c) Initialize the Monte Carlo Tree (MCT) to a single **root**, which represents s_{init}
   (d) While B_a not in a final state:
      - Initialize Monte Carlo game board (B_{mc}) to same state as root.
      - For g_s in number_search_games:
        - Use tree policy P_t to search from root to a leaf (L) of MCT. Update B_{mc} with each move.
        - Use ANET to choose rollout actions from L to a final state (F). Update B_{mc} with each move.
        - Perform MCTS backpropagation from F to root.
      - next g_s
      - D = distribution of visit counts in MCT along all arcs emanating from root.
      - Add case (root, D) to RBUF
      - Choose actual move (a*) based on D
      - Perform a* on root to produce successor state s*
      - Update B_a to s*
      - In MCT, retain subtree rooted at s*; discard everything else.
      - root $\leftarrow$ s*
   (e) Train ANET on a random minibatch of cases from RBUF
   (f) if g_a modulo i_s == 0:
      - Save ANET's current parameters for later use in tournament play.
5. next g_a

Note that during the rollout phase of MCTS training, the same ANET will be used to chose moves for both players. However, the choice of actual moves during an episode is based on visitation statistics returned from the MCTS tree. Only later, in the tournament (see below), will the ANET be used to generate moves in a real game.

As described in the lecture notes, the choice of moves during the early phases of each tree search (using the tree policy) is strongly influenced by the particular player: moves that are good for player 1 are bad for player 2, and vice versa. This search bias should insure that, after many search games, the visit distribution of the root node's children will be atuned to the current player. Thus, this distribution will provide a helpful target distribution (D*) for a feature set (F) that consists of the board state **plus an indicator of the current player**, where (F,D*) then becomes a training case for the ANET.

This means that initially, the ANET will not be very *intelligent* in its action choices during rollouts. But eventually, after considerable training, it should make wiser selections that benefit each player in turn.

5 The Tournament of Progressive Policies (TOPP)

In order to gauge the improvement of your target policy over time, you will periodically save the current state (i.e. weights, biases, etc.) of ANET to a file. By doing this K times during one complete run of the algorithm, you will produce K different policies. For example, if you are training for a total of 200 episodes with K=5, you will have ANETs trained for 0, 50, 100, 150 and 200 episodes.

The saved policies can then be reloaded into K different agents whose relative Hex skills can be assessed by a simple tournament. This will have a round-robin format: every agent plays every other agent in one series of G games. For example, if $K = 5$ and $G = 25$, then there will be $\frac{5*4}{2} = 10$ different series, each involving 25 games. In the TOPP, none of the ANETs will learn. They will just receive game states and return action-probability distributions, which their respective agents will then use to decide on a move.

In theory, policies that receive less training (i.e. those cached earliest) will perform more poorly in this tournament, while the final policy (that receives all of the training) will achieve the best results. However, many factors can impede this ideal of steady progress. Still, if MCTS and supervised learning are working properly, you should see a strong tendency for highly trained policies to dominate those with very little training.

The ANETs used in the final version of your TOPP must be saved to file for later use in a (very short) tournament – but one involving all of them – during the demonstration session.

6 Implementation Issues

6.1 Representing a Hex Game State

Hex boards consist of equilateral triangles, which, when combined, produce a hexagonal structure wherein all interior nodes have six neighbors. For this assignment, the board's overall shape is a perfect diamond, which, as shown at the top of Figure 4, becomes a perfect square when rotated 45°. Thus, your code can work with a simple square array of cells that it treats as a diamond by adhering to the hexagonal neighborhood relationships. In this document, the *size* of the Hex board refers to the number of rows (or columns) of that square array; the board of Figure 4 has size 4.

In testing for chains of matching-colored pieces, your code will need to check a cell's six (or less for boundary cells) neighbors. The coordinates shown on the bottom right of Figure 4 provide the locations of those neighbor cells. If you choose to implement each cell as an object, then one of its properties can be a neighbor list, thus avoiding redundant calculations (of a cell's neighbors) during action choice.

Unlike other board games (such as checkers or chess), the set of legal moves for a Hex state are trivial to calculate. Given the current player, its legal moves are stone placements in each of the unfilled cells.

You will probably have two different representations for each board state: one that includes neighbor relationships and other details that allow your code to analyze states, and another (considerably simplified) data structure, such as a one-dimensional array, to serve as input to a neural network. A standard representation of a board consists of 2 bits per cell, where the bits represent the three possible states of the cell: (0,0), 'empty'; (1,0), 'filled by player 1'; and (0,1), 'filled by player 2'. Each bit becomes the input for one neuron in the neural network.

One essential feature of any game state is the index of the player who will make a move from the given board configuration. So as a bare minimum, a game state should include the status of every cell on the board along with a

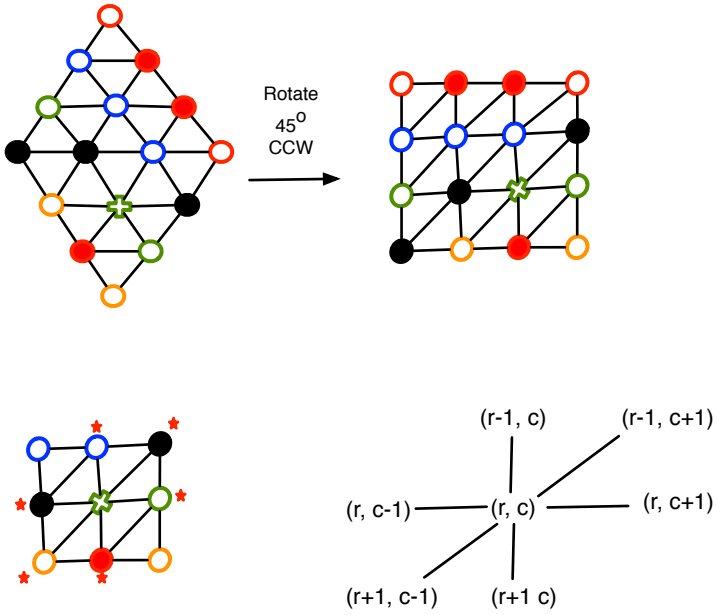

Figure 4: Tips for representing a diamond-shaped Hex board. (Upper left) Board has a perfect diamond shape, with hexagonal neighborhood structure; all neighbors are connected by edges. (Upper right) When rotated 45° counter clockwise, the board becomes a perfect square with each interior cell still having the same 6 neighbors, but visually, the hexagonal patterns may become obscured. (Lower left) Cutout of the neighborhood surrounding the node marked as a cross in the upper diagrams, with all 6 neighbors marked with small stars. (Lower right) In the coordinate system of the perfect square (whose origin (0,0) is in the upper left) the cross has indices (r,c) for *row* and *column*, and the indices of all 6 neighbors are shown.

player id of some sort. A standard id representation consists of 2 bits: (1,0) for player 1, and (0,1) for player 2.

6.2 Displaying a HEX Board

You will need to generate some useful visualization of the HEX Board, whether in a special graphics window or just on the command line. Either way, your diagram needs to show the diamond structure of the board: the top of your board should be a single cell, not a whole row. It also needs to clearly differentiate between an empty cell, a cell filled by a player-1 stone, and a cell filled by a player-2 stone. The game situation should be very clear based on a quick glance at the diagram.

During any phase of system operation, whether training or TOPP play, it must be possible to view the progress of individual games via this graphic. This does not include individual rollout simulations, but does include each *actual game*, i.e. episode, during training and each game of a tournament.

This display feature must be easy to turn on/off by the user. During the demonstration session, we will want to watch a few actual games in progress....but not all games.

6.3 The State Manager

As discussed in the previous project, your code needs to separate game logic from MC tree search, and both of those must be cleanly separated from neural network code. In this project, all game logic for Hex should reside in a "state manager" object that *understands* game states and a) produces initial game states, b) generates child states from a parent state, c) recognizes winning states, and performs any other functions related to the game of Hex. Your MCTS code will not make any references to specific aspects of Hex, but will make only generic calls to the state manager, requesting start states, child states, confirmation of a final state, etc. **Failure to follow this simple design principle will incur a (potentially serious) point loss during the demonstration.**

6.4 The Policy Network

You will implement a single neural network (ANET) that will serve as an actor (as discussed above). As shown in Figure 3, ANET takes game states as input and produces a probability distribution over all possible moves as output. Key design decisions include the representation of board states given to ANET, along with the dimensions of the network, activation functions, optimization method, etc.

The output layer of ANET must have a fixed number of neurons, one for each possible move in a game. For an n x n Hex board, there are simply n^2 possible moves. However, from any given state (S), there are only $n^2 - q$ moves, where q are the number of pieces already on the Hex board for state S. Thus, although the ANET output layer may use softmax to produce a probablity distribution over **all** n^2 moves, those probabilities need to be rescaled to a distribution over only the legal $n^2 - q$ moves: the probabilities for the q no-longer-available moves should be set to zero, while those for the remaining $n^2 - q$ should be renormalized to sum to 1. That probability distribution then serves as the basis for action choice during the rollout phase of each search game.

You will use the ANET in several ways:

1. During the rollout phase of each MC search game, ANET's output distribution will be the basis for action choice.

2. At the end of each episode (i.e. actual game), a minibatch of cases from the Replay Buffer will be used to train ANET via supervised learning.
3. In preparation for the Tournament of Progressive Policies (TOPP), ANETs will be periodically stored away (with weights, biases and other key parameters saved to file) throughout the entire multi-episode training phase of your system.
4. In the TOPP, ANETs will be given game states and produce output distributions, which will support action choices by their respective agents. ANETs will not do any learning during the TOPP.

It is important to note that the output probability distribution (D) of ANET may be used in 3 different ways for each of these three situations, respectively. First, during rollouts, ANET serves as a behavior (default) policy, which normally needs to be explorative. Thus, an ε-greedy interpretation of D is often appropriate. This means that with probability ε, a random move is taken; and with a probability of $1 - \varepsilon$, the move corresponding to the highest value in D is chosen.

During supervised learning, D will get compared to the target probability distribution (from the training case) as the basis for an error term, which backpropagation then uses to compute gradients and modify weights in ANET.

Finally, in a post-training tournament, D might be used in a purely greedy fashion: the move with the highest probability is always chosen. This mirrors the general philosophy that the final target policy should follow a much more exploitative than exploratory strategy. However, you may also continue to use it in an epsilon-greedy fashion or even in a purely probabilistic manner, where moves are chosen stochastically based on the probabilities in D. Finding the best such strategy may require a good deal of experimentation.

It is recommended that you reuse your neural network interface from project 1 in order to configure and run ANET. In addition, the code in tutor3.py for closing, saving and restoring sessions might prove useful in saving and reloading different versions of ANET for use in the tournament. And, of course, this project will also require building and testing many different neural networks before you find the right design, so your earlier system should streamline that process. However, you are free to explore other approaches for implementing ANET, including high-level tools such as Keras.

6.5 Modularity and Flexibility of your Code

Your code must show a clean separation between the following key components:

- Logic for the game of Hex, e.g. generating initial board states, successor board states, legal moves; and recognizing final states and the winning player.
- The neural network(s)
- Monte Carlo Tree Search (MCTS)
- The Tournament of Progressive Policies (TOPP)

Each of these must be handled by its own class (in whatever object-oriented language that you choose). **Failure to demonstrate this minimal level of modularity will result in significant point loss.**

In terms of flexiibility, your system will need to handle variations along several dimensions. Each of the parameters mentioned below must be easily modified during the demo session.

- The size (n) of the n x n Hex board, where $3 \leq n \leq 8$.

- Standard MCTS parameters, such as the number of episodes, number of search games per actual move, etc.
- In the ANET, the learning rate, the number of hidden layers and neurons per layer, along with any of the following activation functions for hidden nodes: sigmoid, tanh, RELU.
- The optimizer in the ANET, with (at least) the following options all available: Adagrad, Stochastic Gradient Descent (SGD), RMSProp, and Adam.
- The number (K) of ANETs to be cached in preparation for a TOPP. These should be cached, starting with an untrained net prior to episode 1, at a fixed interval throughout the training episodes.
- The number of games, G, to be played between any two ANET-based agents that meet during the round-robin play of the TOPP.

Although these parameters do not need to be incorporated into a fancy graphical user interface (GUI), they need to be **extremely easy** to modify during the demonstration session, since we will briefly test out many combinations during that session. **Any long delays for searching through source code to find and change these parameters will incur a significant point loss during the demonstration.**

7 Deliverables

At the demonstration session, you will be asked to do the following:

1. Explain any aspects of the code when questioned by an evaluator (i.e., course instructor or assistant) (**5 points**)
2. Illustrate the modularity and flexibility of your code by walking through the source files. This includes explaining how your code implements the details of the pseudocode described earlier in this document. (**10 points**)
3. Show that your system works under several different choices (by the evaluator) of the key flexibility parameters described above. (**10 points**)
4. Run a very short training session in which several ANETs are saved and then played off in a (very brief) TOPP. (**5 points**)
5. Using ANETs saved from a previous (long) training session, run a TOPP that clearly shows the differences between well- and poorly-trained ANETs. The training session for this test must involve a minimum of 200 episodes and 4 cached ANETs, and it must involve a 5 x 5 Hex Board. (**5 points**)
6. Usie the best ANET that your system can produce (for a 5 x 5 Hex Board), qualify for the playoffs of the Online Hex Tournament (OHT) (**5 points**)

There is no written report for this project module.

A zip file containing your commented code must be uploaded to BLACKBOARD directly following your demonstration. You will not get explicit credit for the code, but it is crucial that we have the code online in the event that you decide to register a formal complaint about your grade (for the entire course).

7.1 The Online Hex Tournament (OHT)

In the Online Hex Tournament (OHT), your agent will go through a 2-stage process:

1. Qualifying round - Your agent will play a round-robin tournament against a few of our own neural-net based agents, all designed by MCTS. If your agent's score exceeds the qualifying threshold (yet to be determined), then it will gain entry into the playoff tournament.
2. Playoffs - This is a single-elimination tournament, with seeding based on the scores achieved in the qualifying round: in the preliminary rounds, agents with the best scores in qualifying will meet those with the worst.

Both phases of the OHT will employ the same protocol, so once your system is capable of playing in round 1, it shouldn't require any recoding to play in round 2.

You will receive 5 points for the OHT if and only if you qualify for the playoffs. There will be no partial credit. There is no limit to the number of agents that can make the playoffs, but the threshold score during the qualifying round must be met. Top finishers in the playoffs may receive prizes (of some sort), but no extra points in the course.

There are no restrictions on training between the qualifying round and playoffs. Once qualified, you are free to continue training and improving your agent right up until the day of the playoffs. The qualifying period will last for approximately a week, while the playoffs will be run online over the course of an hour or two, depending upon the number of qualifiers. No individual (or pair of working partners) can enter more than a single agent in the tournament.

All games in both phases of the OHT will involve a 5 x 5 Hex Board

7.1.1 Technical Details of the Online Hex Tournament

An OHT consists of one or more *series*, where each series consists of several games between the same two players, as described earlier for the TOPP. At the beginning of each series, the OHT server notifies each of the two players as to whether they will be designated as player 1 (red) or player 2 (black). That designation will hold for the entire series. Player 1 will **always** try to build a path that spans all **rows**, while player 2 will always try to span the columns. On the diamond, this means that Player 1 seeks a path between the northeast and southwest sides, while player 2 wants a path between the northwest and southeast sides. Within a series, the only factor that varies between games is the starting player, which will alternate.

Figure 5 illustrates the conversion of a game state into a flat representation - coded as a tuple in Python. The main action by a player during a series will be to receive a flat game-state representation and to respond with a tuple - (r,c) - indicating the row and column of their next stone. At the end of each game, both players receive the final game state and the id (1 or 2) of the winning player. At the end of each series, both players receive the final win-loss information for each player. And at the end of the entire tournament, each player receives a score, reflecting the percentage of games that they won. There is no message indicating the start of a game, only one for the start of each series within a tournament.

This section will be expanded (next week) with the actual methods that you should write in order to interface with the OHT, but the details above should be enough to get you started.

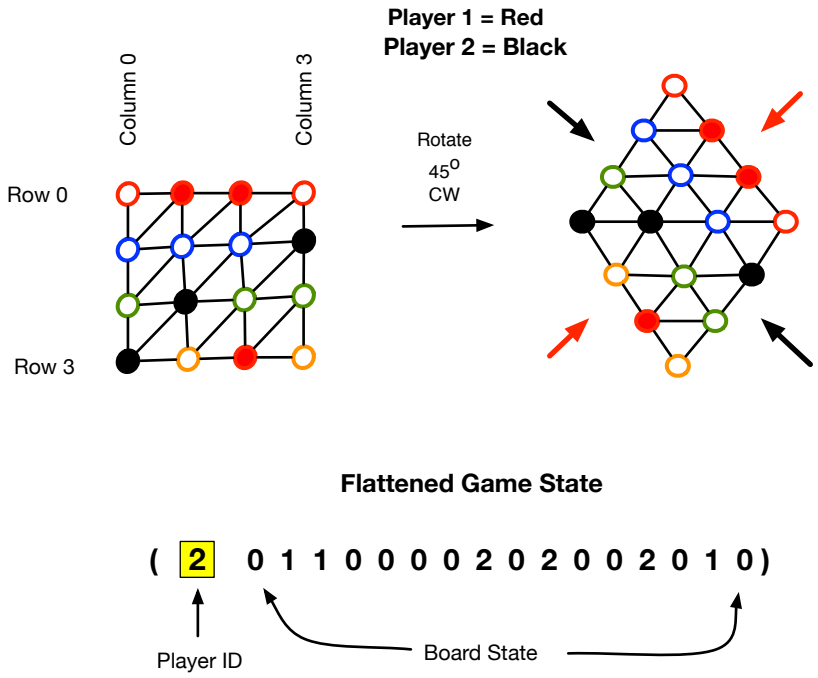

Figure 5: Representation of game states used in the Online Hex Tournament (OHT). A diamond board is a square grid that is rotated 45 degrees clockwise; when converted to a flat representation, the rows are simply concatenated. Cell values are 0 (empty), 1 (player-1 (red) stone) or 2 (player-2 (black) stone). Note that each game state begins with the ID (1 or 2) of the player who must choose the next move. In this particular game, player 2 made the first move.