AI Programming (IT-3105) Project # 2:
On-Policy Monte Carlo Tree Search for Game Playing

Due date: Thursday, April 15 (demo session)

Purpose:

- Implement a general-purpose Monte Carlo Tree Search (MCTS) system for use in a 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, you will implement an MCTS system and combine it with Reinforcement Learning (RL) and Deep Learning (DL) 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 probability distribution over the possible actions) as the function in need of neural-net approximation.

Runs of MCTS will provide target probability 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

For many years, the standard AI algorithm for two-player games was Minimax with Alpha-Beta Pruning. IBM's Deep Blue used it to defeat Gary Kasparov in 1997, and up until very recently, it has been the core of AI chess players, the best of which now routinely beat top human players.

In addition to Minimax, these algorithms include a host of expert knowledge embodied in a) huge databases of book-move opening and end-game strategies, and b) complex heuristic functions, which often include hundreds of weighted factors, such as center control, piece advantage, pawn advancement, etc. All of this expertise seemed mandatory for an AI chess bot until December of 2017, when Google DeepMind's *Alpha Zero* system soundly defeated *Stockfish*, the reigning world-champion computer chess player. Alpha Zero employs no expert knowledge whatsoever. It acquired

chess expertise by playing millions of games against itself, improving via a combination of Reinforcement Learning (RL) and Deep Learning (DL) (a.k.a. Deep Reinforcement Learning - DRL). In addition, the general DRL architecture used in these systems is now being extended to play other games and handle other tasks, such as drug discovery and diagnostic image analysis.

Alpha Zero, like its predecessors: AlphaGo (2016) and AlphaGo Zero (2017), uses Monte Carlo Tree Search (MCTS) as its core RL algorithm. Designers of AI Go-playing systems have long recognized MCTS as superior to Minimax for the game of Go for the simple reason that Go heuristics, on which Minimax crucially depends, have been extremely evasive: even Go experts cannot produce effective heuristics. In MCTS, instead of evaluating leaf nodes of the search tree with heuristics, a rollout simulation (using a default action-choice scheme, a.k.a. *policy*) produces a chain of states from the leaf to a final state, which is trivial to evaluate since it represents a win, loss or tie for the current player. MCTS performs hundreds or thousands of such rollouts and averages the results across the search tree as the basis for each move in the actual game. Over time, the evaluations of nodes and edges in the tree approach those achieved by Minimax, but without the need for a heuristic. MCTS only needs to know the rules of the game and how to evaluate final states.

Although attacking a 19 x 19 Go board 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 MCL, RL and neural networks, but the nets do not need to be particularly deep. But, like everyone else these days, we can still use the sexy term *Deep Learning*.

In this project, you will employ an on-policy MCTS approach (i.e., the target policy and behavior policy are the same), and you will implement that policy as a neural network. 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 four relatively complex programs: a game manager (i.e. a Simworld for the game), your MCTS and RL modules, and a neural-net package (e.g. Tensorflow or PyTorch).

3 Your Kernel MCTS System

Details of the MCTS algorithm can be found in lecture notes for this class, in the article "Monte-Carlo tree search and rapid action value estimation in computer Go" (Gelly and Silver, 2011) – provided on the "Materials" section of the course web page – and in a host of online sources. The exact details of the MCTS algorithm that you decide to implement may vary slightly from these sources, but your code must perform these four basic processes:

1. Tree Search - Traversing the tree from the root to a leaf node by using the *tree policy*.
2. Node Expansion - Generating some or all child states of a parent state, and then connecting the tree node housing the parent state (a.k.a. *parent node*) to the nodes housing the child states (a.k.a. *child nodes*).
3. Leaf Evaluation - Estimating the value of a leaf node in the tree by doing a rollout simulation using the *default policy* from the leaf node's state to a final state.
4. Backpropagation - Passing the evaluation of a final state back up the tree, updating relevant data (see course lecture notes) at all nodes and edges on the path from the final state to the tree root.

Each process must exist as a separate, modular unit in your code: a method, function or subroutine. These units should be easy to isolate and explain during a demonstration of your working system. As described later in this document,

you will have the option to include a critic for leaf evaluation, but only as a supplement to (not a replacement of) the rollout.

3.1 Actual Games versus Rollout Games

Taken together, the four processes listed above constitute one *simulation* in MCTS. Whenever leaf evaluation involves a rollout, an entire *rollout game* is played to assess the value of that leaf node. The rollout game should not be confused with the *actual game* that the system is playing. Each actual game constitutes an RL *episode*, and each move in the actual game stems from the results of M simulations in MCTS, i.e. M pairs of tree search (to a leaf node) followed by a rollout.

Your system must include M as a user-specifiable parameter. Your laptop's computing power will determine the range of M's that your MCTS can handle. For the two simple games below (NIM and Ledge), when the game size (roughly viewed as the number of stones for NIM and the board length for Ledge) is approximately 10, then an M value of 500 is often sufficient.

Note that it is also possible to use a dynamic value of M. When faced with limited computing power, it may be wise to use a smaller M at the beginning of an actual game (when paths from root to end-state are long) but to then increase it as the game progresses. Alternatively, you may just set a time limit (of a second or two) on each call to the MCTS module, as this has a similar effect to that of a gradually-increasing M.

4 Simple Preliminary Games

The MCTS algorithm is complex enough in its own right, so it is advantageous to test it on a simple task before deploying it on Hex, the primary task of this project. What follows are brief descriptions of two games that you may want to use as testbeds for MCTS. This is not mandatory, only a helpful suggestion that could save a lot of debugging time, in the long run. Also, by applying your MCTS to more than one game, its generality should be enhanced. When implementing the second game, you'll often discover implicit assumptions involving the first game that were accidentally built into the MCTS. Handling the second game forces a deeper assessment of the code's true generality.

4.1 A Basic NIM Game

Believed to have originated in ancient China, NIM is actually a variety of games involving pieces (or *stones*) on a non-descript board that players alternatively remove, with the player removing the last piece being the winner, or loser, depending on the game variant. Typical constraints of the game include a minimum and maximum number of pieces that can be removed at any one time, a partitioning of the pieces into heaps such that players must extract from a single heap on a given turn, and boards with distinct geometry that affects where pieces reside, how they can be grouped during removal, etc.

A very simple version of NIM provides a nice challenge for MCTS. The 2-player game is defined by two key parameters: N and K. N is the number of pieces on the board, and K is the maximum number that a player can take off the board on their turn; the minimum pieces to remove is always ONE.

Given N and K, the remaining rules of play are extremely simple: Players take turns removing pieces, and the player who removes the **last** piece is the **winner**. By changing the values of N and K, you can produce many variations of

the game, some of which are harder than others.

4.2 The *Old Gold* Game

Old Gold, which I will call *Ledge* for this assignment, involves a one-dimensional board divided into cells, each of which may be empty or may contain coins of either type 1 (copper) or type 2 (gold). Board length may vary, but the leftmost spot (position 0) of the board is always known as *the ledge*. There is only ONE gold coin on the board, while there can be many copper coins (or none at all). No cell can contain more than 1 coin at a time.

Initial configurations of the board can vary dramatically. As long as one and only one gold coin exists on the board, the configuration is a legal start state.

Players alternate making moves, where a move consists of either a) picking up the coin currently sitting on the ledge, or b) moving a coin from its current location to another location **to the left**. If the coin to be moved resides at location j, and its nearest leftmost neighbor resides at location i (where $j - i \geq 2$), then the coin can be moved to any cell from i+1 to j-1. So the coin cannot hop over any neighbors to the left but is free to move into any of the cells between it's current location and the neighbor's cell. The winner is that player who removes the gold coin from the ledge. The copper coins incur no benefits when removed from the ledge.

The difficulty of a Ledge game depends upon the initial board configuration, B_{init}. This can be a wide range of vectors of any length L, containing 1 gold coin, and anywhere from zero to L-1 copper coins.

4.3 Guaranteed Wins

Given the values of N and K (for NIM), and B_{init} (for Ledge), one of the two players can guarantee a win. Multiple runs with the same starting configuration and same starting player will often have very skewed final results, e.g. Player 1 wins 90%. A properly-working MCTS should eventually discover these guaranteed-win strategies, though it may take several hundred or thousand rollout games to do so. For larger values of N (e.g. $N > 15$) for NIM and longer boards for Ledge, your MCTS may struggle to find optimal strategies due to the enormous search spaces. But for moderate values of these parameters, the results of batch runs with a fixed starting player give a good indication of the proper functioning of your MCTS. In short, a good way to check your MCTS for logical bugs is to implement one of these two games, fix the initial game setup and starting player, and then check that the player who has the guaranteed win does in fact win most of the time (when given enough rollouts).

5 The Game of Hex

The (simple) game rules of Hex are explained in the file hex-board-games.pdf (*Board Games on Hexagonal Grids*) in the Materials section of the course web page.

6 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

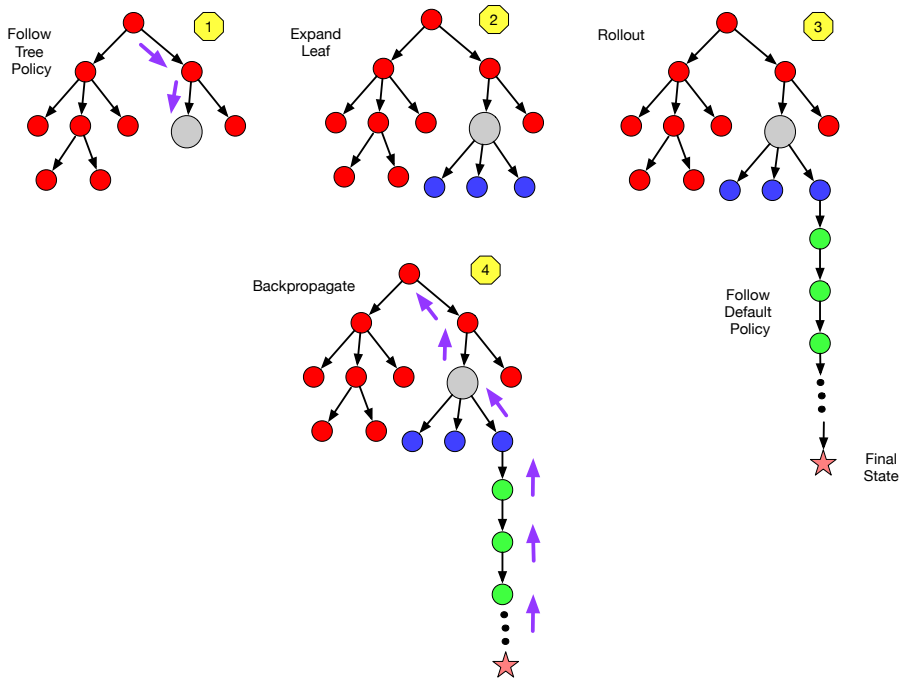

Figure 1: Overview of the main elements of MCTS search.

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, 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.

6.1 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 behavior 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 1 summarizes the MC tree-growing process that underlies each RL episode.

Statistics (e.g. 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 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 2 summarizes this interaction between the MC tree and the target policy embodied in the ANET.

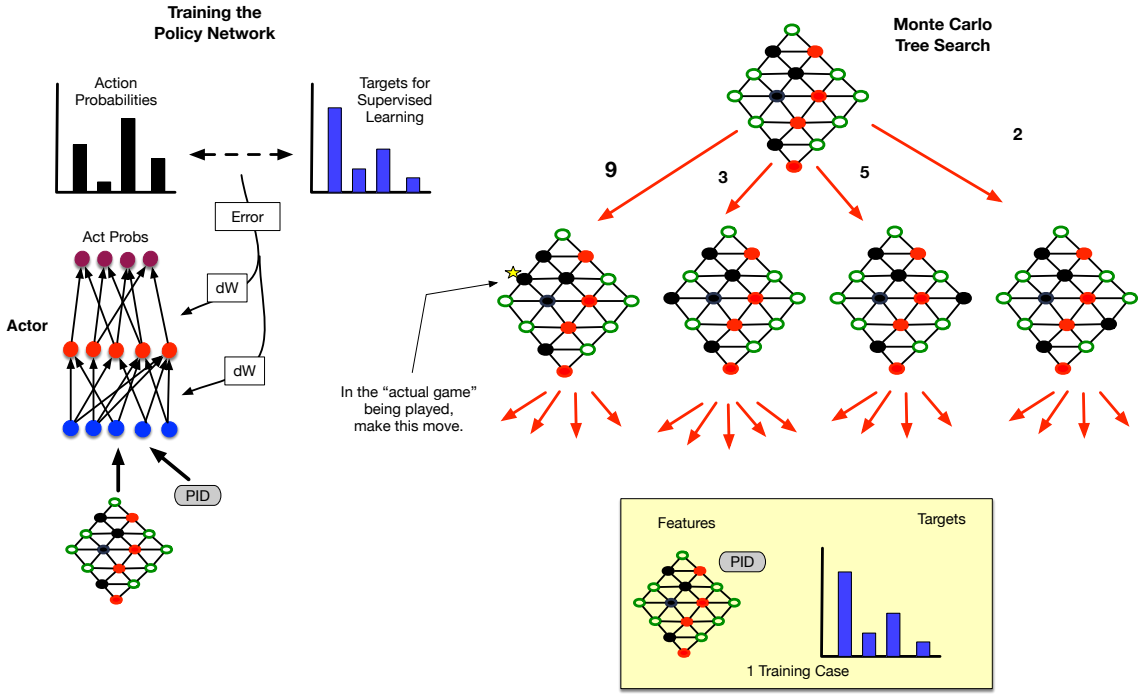

Figure 2: Overview of supervised learning of the target policy network based on results of Monte Carlo Tree Search (MCTS). Numbers on MCTS branches denote visit counts accrued during the multiple search games associated with each single move in the actual game. Each case (yellow box) is stored in the 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 choose 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.

6.2 Adding a Critic

One of the major differences between AlphaGo and AlphaGo Zero involves the treatment of leaf nodes in the Monte Carlo Tree. The former performs complete rollouts from leaves to final states, while the latter employs a critic (another neural network) that maps states to values. [1] This saves considerable runtime, although the critic's evaluations for early phases of a run will be quite unreliable.

In this project, you cannot skip rollouts completely, since this is an important exercise. However, you are free to add a critic along with a new parameter, σ, that represents the probability of doing a rollout (instead of calling the critic) during any given visit to a leaf node. You will probably want to begin an MCTS run with $\sigma \approx 1.0$ and then gradually decrease it over the episodes.

As discussed in the class lecture notes on MCTS, the critic can be trained by using the score obtained at the end of each actual game (i.e. episode) as the target value for backpropagation, wherein the net receives each state of the recent episode as input and tries to map that state to the target (or a discounted version of the target).

Note that this is optional and is only a supplement to (not a replacement of) rollouts for this assignment.

7 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 M times during one complete run of the algorithm, you will produce M different policies. For example, if you are training for a total of 200 episodes with M=5, you will have ANETs trained for 0, 50, 100, 150 and 200 episodes.

The saved policies can then be reloaded into M 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 $M = 5$ 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.

[1] AlphaGo Zero actually employs a single network that maps states to both a probability distribution over actions and to an evaluation.

8 Implementation Issues

8.1 The Hex Board

See the file hex-board-games.pdf (*Board Games on Hexagonal Grids*) in the Materials section of the course web page for tips on representing a Hex, (a.k.a., Con-tac-tix) board.

8.2 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.

8.3 The State Manager

As discussed in the previous projects, 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.**

8.4 The Policy Network

You will implement a single neural network (ANET) that will serve as an actor (as discussed above). As shown in Figure 2, 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 a k x k Hex board, there are simply k^2 possible moves. However, from any given state (S), there are only $k^2 - q$ 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** k^2 moves, those probabilities need to be rescaled to a distribution over only the legal $k^2 - q$ moves: the probabilities for the q no-longer-available moves should be set to zero, while

those for the remaining $k^2 - q$ 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. This comparison of output and target probablity distributions normally calls for a *cross-entropy* loss/error function; and to produce a probability distribution in the output vector, the *softmax* activation function is recommended.

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.

8.5 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 Reinforcement Learning System - which houses the actor (and a critic, if you're using one).
- 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 flexibility, your system will need to handle variations along several dimensions, as defined by the following *pivotal parameters* for this project:

- The size (k) of the k x k Hex board, where $3 \leq k \leq 10$.
- 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: linear, 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 (M) 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.**

9 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) (**6 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. (**6 points**)
3. Show that your system works under several different choices (by the evaluator) of the *pivotal parameters* described above. (**6 points**)
4. Run a very short training session in which several ANETs are saved and then played off in a (very brief) TOPP on a k x k Hex Board, where k is 4 or 5, depending upon your machine's computing power. (**6 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 either a 4 x 4 or a 5 x 5 Hex Board. The 5 x 5 is preferable if your laptop permits it. (**6 points**)
6. Using the best ANET that your system can produce for a K x K Hex Board , achieve a **good** score on the Online Hex Tournament. Each of K, the definition of *good* and any levels of partial credit will be determined at a later date, possibly after all results of the OHT are finalized. See below for more details of the OHT. (**10 points**)

There is no written report for this project. The 40 points for this project are 40 of the total (100 points) for this course.

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).

10 The Online Hex Tournament (OHT)

In the Online Hex Tournament (OHT), your agent will play a *round-robin* tournament against several of our online Hex players, some of which are more intelligent than others. Round-robin implies that your agent will play a fixed number of games against each of our opponents. The total number of games that your agent wins will then provide the basis for a final score. All Hex agents (max one per student) will then be sorted by these final scores. From these scores, one (or maybe two) cutoffs will be determined, with all agents at or above the cutoffs earning additional points for their student designers. Making the highest cutoff also entails qualification for *the playoffs*, which we occasionally set up between the top players. However, there is no guarantee that the playoffs will be held on any given year. There is no pre-determined limit as to how many students can achieve various cutoff scores.

Students working in a group must enter the OHT as separate individuals. They will be scored solely on the basis of their own entry into the OHT and not on the basis of a teammate's score.

10.1 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 (i.e. the player making the first move), which will alternate.

Figure 3 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 start of each game, both players will receive a simple message indicating the player (1 or 2) who will start that particular game. At the end of each game, both players will receive the final game state and the id (1 or 2) of the winning player. At the end of each series, both players will receive the final win-loss information for each player. And at the end of the entire tournament, each player will receive a score, reflecting the percentage of games that they won.

You will receive two source-code files: BasicClientActor.py and BasicClientActorAbs.py. The former is a subclass of the latter, and you should only need ot modify the former. Modifying the superclass, BasicClientActorAbs, is strongly discouraged, and we (the course instructor and assistants) take no responsibility for any errors incurred by such modifications.

In BasicClientActor.py, there are seven methods (clearly marked) that you need to fill out in order to activate your player for the OHT. Six of these are merely informative: they require no action (i.e. return value) on your part. Only the method handle_get_action requires a return value. The 7 methods are as follows:

1. handle_series_start(unique_player_id,series_id,player_id_map,num_games,game_params) - This signals the start of a series and provides several items that you may or may not find useful:
   - unique_player_id - Each player in the online tournament registry has a unique id (a large integer). This is the id for your player in that registry. It will not change from one series to the next (nor from one tournament to the next), but you receive it at the beginning of each series anyway.
   - series_id - This is a small integer (1 or 2) indicating which player you will be in the current series. You will be the same player throughout the entire series, but the system will alternate the starting player: player 1

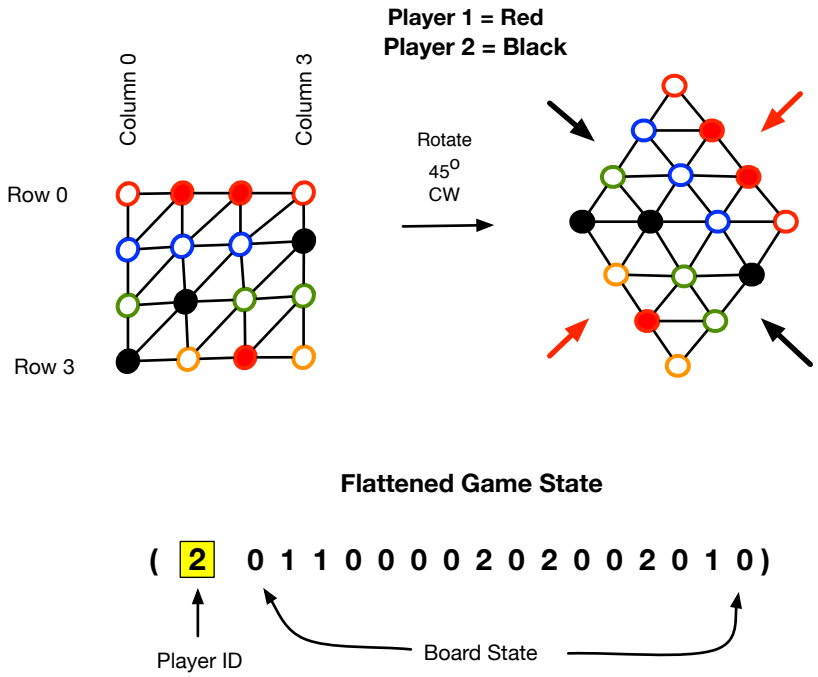

Figure 3: Representation of game states used in the Online Hex Tournament (OHT). A diamond board is a square grid that is rotated 45 degrees clockwise; when converted to a flat representation, the rows are simply concatenated. Cell values are 0 (empty), 1 (player-1 (red) stone) or 2 (player-2 (black) 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.

will start the first game, player 2 the second, player 1 the third, etc. Your series id may change from one series to another but will remain constant within any given series.

- player_id_map - This is a list of tuples, where each tuple contains a unique player id and its corresponding series id (for the current series). This information is only useful if you plan to adjust your strategy based on the opponent (indicated by the unique player id that is not your own id).
- num_games - The number of games to be played in this series.
- game_params - A list of important parameters for the game. For HEX, the only item in this list is the board size, K.

2. handle_game_start(series_id) - This signals the start of a game and provides the series id (1 or 2) of the player who will make the first move.
3. handle_game_over(winner_series_id,final_state) - This signals the end of a game, and provides the series id of the winner (1 or 2), along with the final game state.
4. handle_series_over(stats) - This signals the end of a series and provides the win-loss statistics for each player in the form of a list containing two tuples, where each tuple has the form:

   (unique_player_id, series_id, wins_in_this_series, losses_in_this_series)

5. handle_tournament_over(score) - This signals the end of the tournament and provides your final score: percentage of games that you won.
6. handle_illegal_action(state,act) - This indicates that you tried an illegal move and provides the state and illegal action (act) that you attempted.
7. **handle_get_action(state)** - This **requests** an action in response to the current game state. You need to respond by returning a tuple (row, column) indicating the spot where you will place the next stone. Note that the row and column must be zero-based indices, so for a game with board size K:

   $0 \leq row \leq K-1$, and $0 \leq column \leq K-1$.

In addition to the two source-code files, you will receive a certificate for the OHT server (server.crt). Save this in the same directory as BasicClientActor.py and BasicClientActorAbs.py.

Once you have filled in the code for the 7 methods in BasicClientActor.py, you should only need to do two things to begin playing in the OHT:

1. Create a BasicClientActor object.
2. Call the method **connect_to_server**, which your object will inherit from BasicClientActorAbs.

The OHT requires you to verify your student status by providing your NTNU username and password. This process goes through NTNU's server, which verifies that you are a valid NTNU student. Our system does not record nor save your password anywhere.

Once logged on to the system, you will be asked to provide a "nickname" for your player. Be creative! You can change nicknames between logins to the system if you wish. They are only used for the purpose of printouts during a tournament but are not cached in the OHT database.

After providing your nickname, the system will ask whether or not you want to participate in an **official tournament round**. You will receive a message similar to the following:

Do you want to attempt to play an official tournament? You have 5 remaining attempts. (Y/n)

By answering ”Y” to this question, you will invoke a complete round-robin tournament, which entails many (e.g. 50 or 100) games against each of our opponents. You have a maximum of 5 such attempts, with your highest score serving as the basis for your assessment. You will receive your score at the end of each qualifying attempt via the handle_tournament_over method to your BasicClientActor object. The cutoff scores for points and entry into the playoffs will be determined at the end of the qualifying period, when the complete distribution of scores (over all student participants) can be analyzed.

By answering ”n” to this question, you will automatically begin a **verification tournament**, which is much shorter in duration (only 100 games) but long enough to check whether or not your system properly interfaces with the OHT. You can run as many rounds of the verification tournament as you wish, though each round will require a separate log in to the system. One of the opponents that your player will face in the verification tournament is a random player, so over the course of 100 games, your player should experience a wide array of game states. If you have any errors in your move generator, they should pop up within those 100 games.

It is strongly advised that you run several verification tournaments before attempting a playoff-qualifying round.

Your remaining playoff-qualification attempts are tied to your NTNU username, not to the nickname(s) that you give during each login to our system: changing your nickname does not earn you a fresh set of attempts.

Other notes:

1. In BasicClientActor.py, some of these 7 methods have small code bits in them already. These can be retained or deleted at your own discretion. None of the code bits are essential.
2. When you create a BasicClientActor object, the ”verbose” option to the init function defaults to True. This entails that a lot of game states will be printed to your screen. Simply set it to False to avoid these prints.
3. The OHT server **only** responds to requests from computers on the NTNU network. Therefore you either need to be on campus or use VPN to have access.
4. Remember that your code will be uploaded to BLACKBOARD at the end of project 3. For privacy and security reasons, be sure **not** to include your NTNU password in any automatic login code that you may want to write (but should not write).