

Design specification document

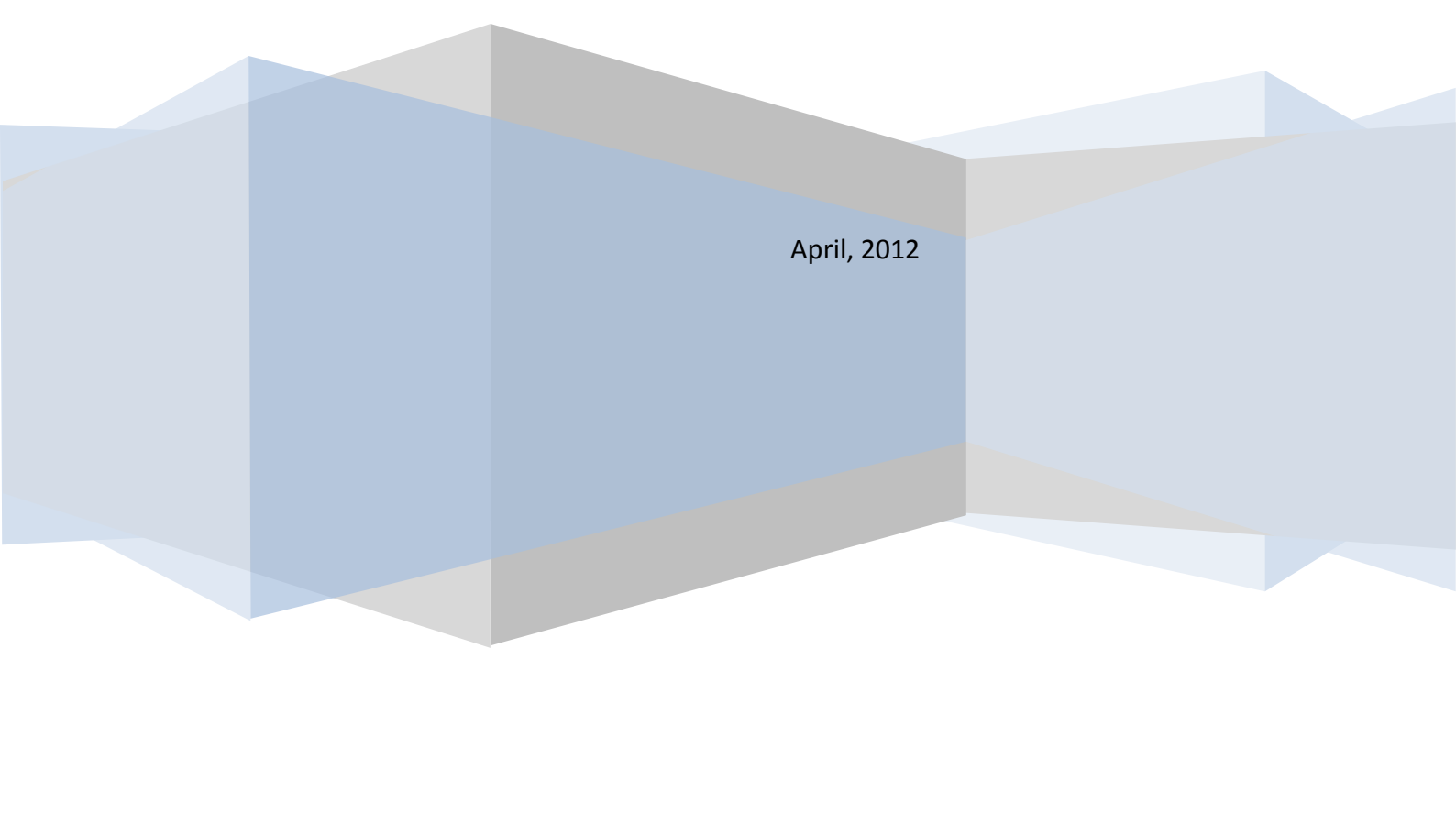
Reinforcement Learning Workshop

Eli Libman, 062896113

Nur Lan, 026495598

Shanee Lavi, 301271474

April, 2012



Revisions

Version	Comments
1.0	First version, including basic information
1.1	<ul style="list-style-type: none">• Fixing minor mistakes (comments after presentation)• Adding Application design• Conforming to new design spec doc template
1.1.1	<ul style="list-style-type: none">• Adding XML file format and more design alterations
1.1.2	<ul style="list-style-type: none">• Minor change in XML file
1.1.3	<ul style="list-style-type: none">• GUI Design
2.0	<ul style="list-style-type: none">• Adding testing method
2.1	<ul style="list-style-type: none">• Adding Class Diagrams and Sequence Diagrams (for three main use cases)
2.2	<ul style="list-style-type: none">• Updating GUI design description
2.3	<ul style="list-style-type: none">• Added more details to introduction and Learning method
2.4	<ul style="list-style-type: none">• Adding two new feature types documentation
2.5	<ul style="list-style-type: none">• Added summary of algorithmic testing process
3.0	<ul style="list-style-type: none">• Adding Trial and Error chapter• Adding final training parameters

Contents

Revisions	2
Introduction	4
Learning method	4
Game representation	5
Board Features	5
Training	6
High Level Design	8
Source Control	8
Project Executable	8
GUI	8
File Format	13
Debugging and Testing	14
Testing Method	14
Results and Conclusions	15
Trial and error	15
Agents quality	16
Learn parameters tuning	17
Issues recognized during development and testing	22
Detailed Class Description	24
Appendix	25
Class Diagrams	25
Learning process Class Diagram	25
Testing process Class Diagram	26
Playing process Class Diagram	27
Sequence Diagrams	28
Use Case: Test GUI	28
Use Case: Train GUI	28
Use Case: Play GUI	29
Use Case: Run single game in train mode	30

Introduction

This project's main purpose is implementing a self learning 4 in a row player using a reinforcement learning algorithm. 4 In A Row is a fully observable adversarial two player game. In the reinforcement learning scheme the opponent would be modeled as the stochastic part of the environment - When the player takes a move the resulting state depends on the opponents choice of action which is unknown by the agent.

The project will include implementation of a GUI application allowing 3 types of use cases: Learn, Test and Play. The application will focus on creating and using a learner state data object that holds the parameters learned. Learn mode will allow the user to easily select reinforcement learning parameters (i.e. epsilon, gamma, length of learn session) and run a learning session. Test mode allows the user to test two learned parameters against each other. Play mode will allow the user to run a player based on the loaded learnt parameters in the GameManager infrastructure.

Finally, we will implement a java applet for online player vs. our resulting best player.

Learning method

We will implement an epsilon-greedy Q-learning algorithm of the SARSA variety. Learning the Q function instead of the environment is useful in an adversarial setting like 4 In A row as it allows us not to study the opponent (i.e. the environment) but the utility of the states instead.

Because of the asymmetry between game roles (first/second player) we decided to learn the Q function separately for either role.

Learning the Q function directly requires holding a table between each possible <state, action> pair to a real value. As the number of states increases this representation becomes very large (for our version of FIAR it is $O(3^{48})$). To overcome this problem we won't represent $Q(s, a)$ directly but will learn a linear approximation of it instead. More specifically we will maintain $Q_a(s)$ approximation for each action a (chosen column). Thus we represent the global Q-function as a sort of one level decision tree.

The linear function approximation receives a vector of board features (see next section) representing the current state and returns some linear combination of these

features. The learning process goal is to find the locally optimal weights of this linear combination.

SARSA learning algorithm, as its name implies, requires a $\langle \text{State}, \text{Action}, \text{Reward}, \text{State}, \text{Action} \rangle$ 5-tuple for its update rule. In FIAR there's no immediate reward for actions not leading to a victory or loss - only the terminating move receives a reward. So the Q-values of terminating states are either 1 for victory or -1 for loss. Because these values need to propagate to the Q-Values of earlier states, we choose the gamma discount factor to be 1.

We will run the update rule after each time we see the result of the move the agent took (board after both agent and opponent took their moves):

$$\theta^{(t+1)} = \theta^{(t)} + \alpha [r_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)] \vec{s}_t.$$

Where:

$$\forall i: \theta_i^{(0)} \sim N(0, 1),$$

$\vec{s}_t = \text{feature vector extracted from the current state}$ and

$$a_{t+1}: \begin{cases} \max_a Q_t(s_{t+1}, a) & w.p. 1 - \epsilon \\ a \sim U(a_1, \dots, a_8) & w.p. \epsilon \end{cases}.$$

If s_{t+1} wins with a_{t+1} then: $Q_t(s_{t+1}, a_{t+1}) = 1$

If s_{t+1} lost with a_{t+1} then: $Q_t(s_{t+1}, a_{t+1}) = -1$

ϵ is our exploration vs. exploitation parameter. At first we would like to explore the state space so we can gather a crude but broad knowledge of the utility of states. The more we know we would like to exploit our knowledge and gain new knowledge about more complicated game sequences (achieved with our self play training scheme, see Training). ϵ will allow our agent to be trained with this sort of balance between exploration and exploitation. We tried several ϵ update methods as will be described later.

Game representation

We represent the game board using a 6x8 grid. Each cell in the grid will have one of the following three states: RED, BLACK or '_' (an empty cell).

Board Features

The linear approximation will be using the following game features:

1. Raw board state - each game cell will have a feature representing its state (6*8=48 features)

- 1 – player owned cell
 - -1 – opponent owned cell
 - 0 – neutral cell
2. Action based potential – for each player for each column we'll have the following features: ($2 \times 3 \times 8 = 48$)
- Completion to 3 in top used row
 - Completion to 3 in top unused row
 - Completion to 4 in top unused row
3. Global board features
- Parity – Because 4 in a row is a fully observable 2 player game, board parity information plays an important role. If a player has a threat on an even row, and all columns contains an even number of columns it can force a win. (see Zugzwang analysis in Allis, 1988)
 - i. 2 features that represent whether the above condition is true for the agent and for the opponent.

Altogether we extract from the board 98 features.

Training

- Training will be done by having two copies of the player playing each other. Both learning their respected roles.
- Bootstrapping would be achieved by hard coding
 - winning moves – will cause victory rewards to be awarded early
 - random start moves – Allowing us to explore more of the state space
 - self play – Allows player to be more challenged as the learning process continues
- Learning rate $\alpha = \frac{1}{1+\sqrt{t/2}}$. t – number of games played so far in the training session.

This way we achieve a properly decaying learning rate. (see site for how we chose this alpha)

- Discount factor γ – one, because rewards are only given at the end of the game.
- Epsilon – in order to balance between exploration and exploitation we'll choose the next action with a ϵ – *greedy* policy. We tested three methods used to determine ϵ throughout the training:

- A constant value: with probability ε we choose a random action, otherwise – a greedy action.
- Epsilon decreasing strategy: decreasing epsilon as the learning progresses allows us to explore more at the beginning of the learning process and exploit as we converge. We tested two methods:
 - $\varepsilon_{t+1} = \varepsilon_t \cdot c$. We reduce the exploration rate after each game. The value of c will affect the time it will take ε to converge to zero. It is common to choose $0.8 \leq c < 1$.
 - Using a reduction function: $\varepsilon_t = \frac{\varepsilon_0 \cdot \log(t)}{\sqrt[4]{t}}$.
- Q Approximation weights – These are the values to be learned by the training process.

Final training parameters

Throughout the training process we tried different values and functions for the learning parameters that influence the quality of our agent. Between the parameters tested were: learning rate function (α), epsilon update method, number of games played etc.

Most of the parameters are bound together and have a big influence on each other. The main goal of our training process was to find the combination of parameters that allows us to create the best player we could.

Our final player was trained using the following technique:

- Gamma was set to 1
- Initial epsilon value was set to 0.9
- The epsilon update method chosen was annealing
- Altogether we played 2 million games:
 - We ran a session of 1 million games with an annealing factor 0.99999
 - We ran another session of 1 million games with an annealing factor 0.9999. This run continued the learning of the previous run (meaning the initial epsilon of this run is the final epsilon of the previous run), the only difference was the change of the anneal factor.

High Level Design

Source Control

The project source SVN trunk is hosted at <http://xp-dev.com/svn/autoplayer>. SVN was chosen as source control platform to allow us easily manageable code sharing capabilities. This enabled us to efficiently work on our code base at unison while maintaining code integrity.

Project Executable

The GUI application has three modes of action:

- Learn – Application sets up two copies of the player to train against each other for n games. During this training sequence, learning parameters and learned parameters are maintained on persistent storage (file/s).
- Test – Application runs two players against each other and returns statistics.
- Game manager player – Applications sets up one copy of the player as a FourInARow gamer that connects to a GameManager server. This mode is used in a final setting where the objective is to test our gamer against other workshop gamers.

GUI

Global learned parameters management buttons

We manage the loading and saving of learned parameters xml file (see file format section) globally in our application using these buttons:

- Load: loads existing player parameters from file.
- Save: saves the current player parameters to file. If parameters are new, a choose file dialog will appear.
- Save as: saves the current player parameters to a specific file (a choose file dialog will appears).
- New: initializes the value of epsilon and gamma to default values and resets the weights to random ($\sim N(0,1)$).

Application Tabs

Learn and play tabs uses the global loaded learned parameters, while the Test tab loads two configuration files with no regard to the globally loaded one.

Learn

this mode is used to manage the training process.

Use case:

1. User chooses learn parameters
2. User chooses log file name
3. User chooses number of games to play
4. User saves initial version in order to name the configuration
5. User chooses whether he wants incremental versions to be saved during the learning sequence
6. User pushes Start Training
7. User saves configuration after training is over

The screenshot shows a software window with three tabs: 'Learn', 'Test', and 'Play'. The 'Learn' tab is active. It displays the following elements:

- Total Played:** 0
- Gamma:** 1.0
- Epsilon:** 0.9
- Update Options:** Three radio buttons are present: 'No update' (selected), 'Annealed' (with a value of 0.9999), and 'Num of games'.
- Statistics Log File:** A text input field containing 'file name of statistics log'.
- Start Training:** A button.
- Save incremental version:** A checked checkbox.
- #Games to play:** 10000
- Bottom Buttons:** 'Load...', 'Save', 'Save As...', and 'New'.

This tab contains the following parameters:

- Games played so far in learn mode (for loaded learning information) – informative, non-editable
- Gamma
- Epsilon:
 - Initial epsilon value

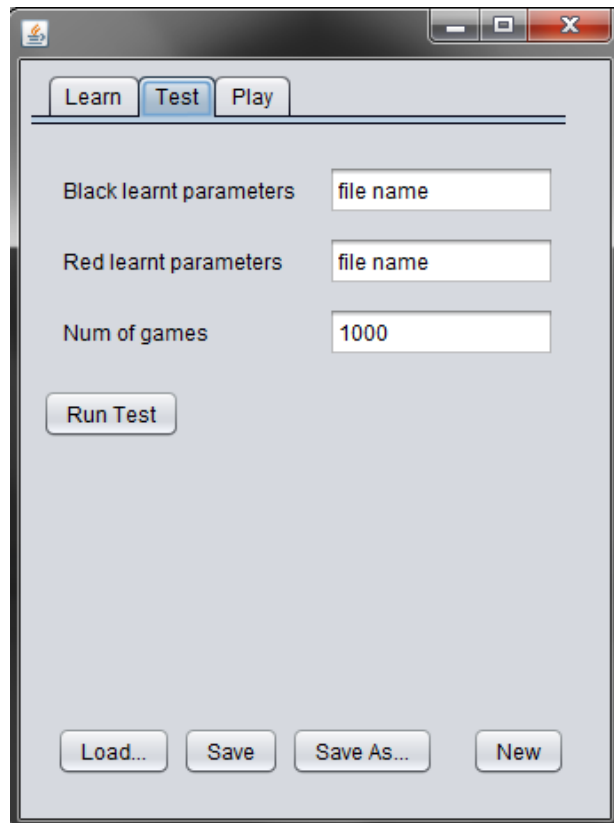
- Epsilon update method (if the annealing method is chosen – there is a parameter that represents the annealing factor).
- Statistics log file name. (see file format section)
- Save incremental versions - a check-box that when checked, saves 20 incremental versions throughout the training session (every $\approx \frac{n}{20}$ games).
- Number of games to be played in the training session.
- When the start training button will be pushed a progress bar will open and will include the following:
 - Number of games played so far.
 - Game rate (number of games per second).
 - Current epsilon value.
 - Num of version (changed only when we save incremental versions).

Test

This mode is used to generate win statistics of trained player VS. another trained player.

Use case:

1. User chooses to player configuration files to compare
2. User pushes run test

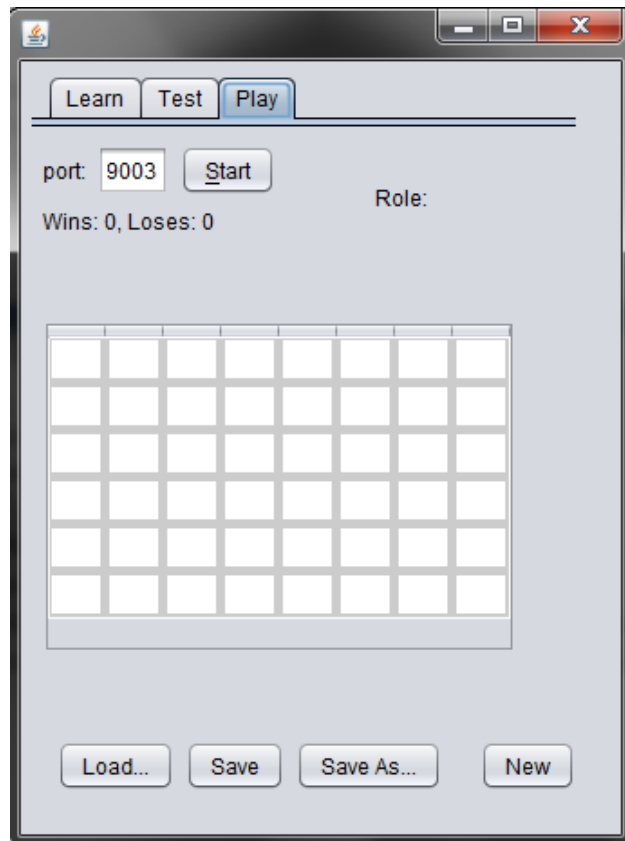


This tab contains the following parameters:

- A path to each of the players learnt parameters file.
- The number of games to be played in the testing session.
- When the run test button will be pushed
- A progress bar will open and show the number of games played so far.
- At the end of the test a dialog will open and include the following:
 - The number of total games played
 - The number of game each player won in each role (first or second player).

Play

- this mode is used to launch a player in the game manager framework.
- Use case:
 1. user loads a player configuration
 2. user chooses a port number
 3. user press start



This tab contains the following parameters:

- Port number.
- Role (black or red player).
- Game play status: wins and losses.
- A board game showing the progression of the game

File Format

Learned Parameters file

The application will save the learning parameters in an XML file used by all modes (learning, playing and testing) with the following format:

```
<RLGamerData>
  <GlobalParameters>
    < Gamma > value </Gamma>
    < GamesPlayed > value </GamesPlayed>
    <Epsilon> value </Epsilon>
  </GlobalParameters>

  <FirstQLinearApproximation numOfActions = [numOfActions]>
    <ActionQApproximator numOfFeatures = [numOfFeatures] >
      <weight number=0> value </weight>
      :
      <weight number=[numOfFeatures]> value </weight>
    </ActionQApproximator >
    :
    <ActionQApproximator numOfFeatures = [numOfFeatures] >
      <weight number=0> value </weight>
      :
      <weight number=[numOfFeatures]> value </weight>
    </ActionQApproximator >

  </FirstQLinearApproximation>
  <SecondQLinearApproximation numOfActions = numOfActions>
  :
  </SecondQLinearApproximation>

</RLGamerData>
```

Learn statistics log file

During the learning process the application will produce a csv log file where each line describes statistics for a single game of the learn sequence.

Columns

- Game num – game number in learn sequence
- Winners Color – color of the winner
- Num of turns – Number of plays the game took
- Game sequence – space separated integer list representing the moves sequence
- maxUpdateDelta – the maximal Q error of the sarsa update rule during the game

Debugging and Testing

The application needs to be tested both at the code unit level (unit tests), as well as at the algorithmic level (i.e. parameters tuning, choosing an optimal train process, etc).

The unit tests would be implemented using the JUnit framework, while the code design is driven towards modularity and abstraction to allow both easily tested code and reusable code.

The following section describes the algorithmic testing and debugging process in more detail.

Testing Method

The algorithmic testing process has two main objectives:

- Test our agent's quality.
- Test learning parameters such as the method used to update the epsilon.

Agents quality

Testing the agent's progress: at different stages of the learning process (e.g. every 1,000 games) we will save the current state (the weights learned up to that point). When the learning process is over, we will take the final player and test it against all the players that were saved during the process. In order to determine that the player is getting better at the game, we expect to see a decrease in the percentage of games won by the final player. This will imply that the difference in abilities between the versions is getting smaller as the versions are getting closer to the final one.

Parameter tuning

Testing learning parameters: we will create different agents using different parameters in the learning process and then we will test the agents against each other. The differences between each two agents we will test against each other will be such that will allow us to try and isolate the factor that might have been the one which improved the winner. The parameters we will be testing include:

- Epsilon update method:
 - Testing the different methods.
 - In each method – testing the initial epsilon and in the annealing update case – testing the factor.

Since four in a row is not a symmetric game (one of the players can assure he won't lose), we will test two versions of the player against each other such that each version will have the same amount of games in each role (first and second player).

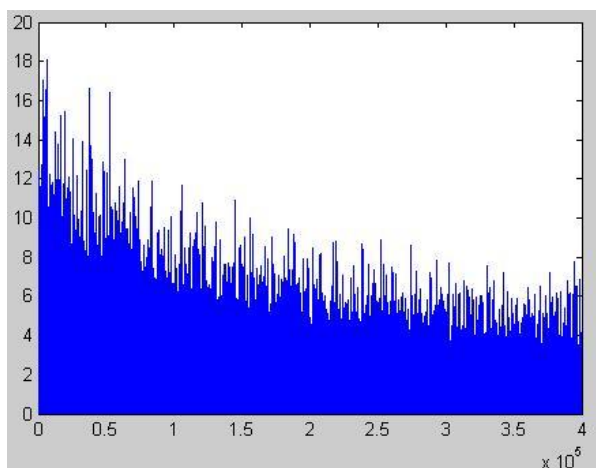
Results and Conclusions

Trial and error

The learning algorithm has several free parameters: $\varepsilon, \alpha, \theta^{(0)}$ values, etc. After we implemented the basic learning algorithm we tested several of these parameters in order to generate the best player.

Q convergence

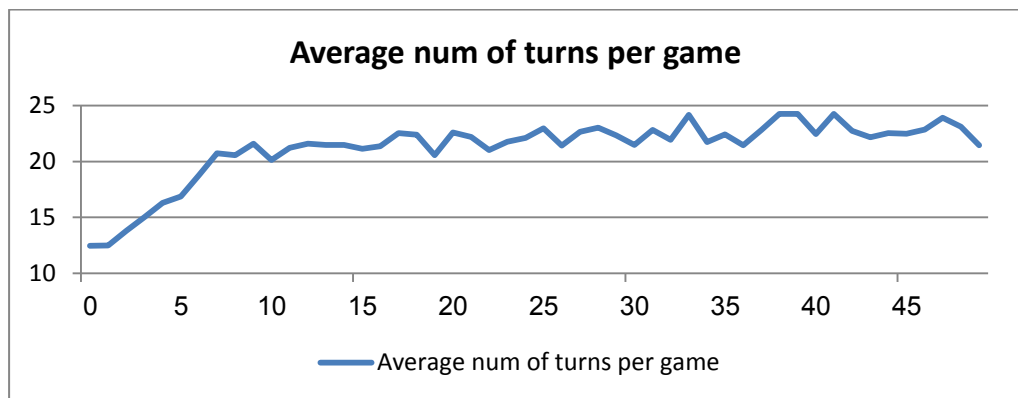
First we wanted to see if the Q function seems to be converging on some value. This graph represents the $r_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)$ part of the update rule during 400k simulated games.



Average number of turns per game

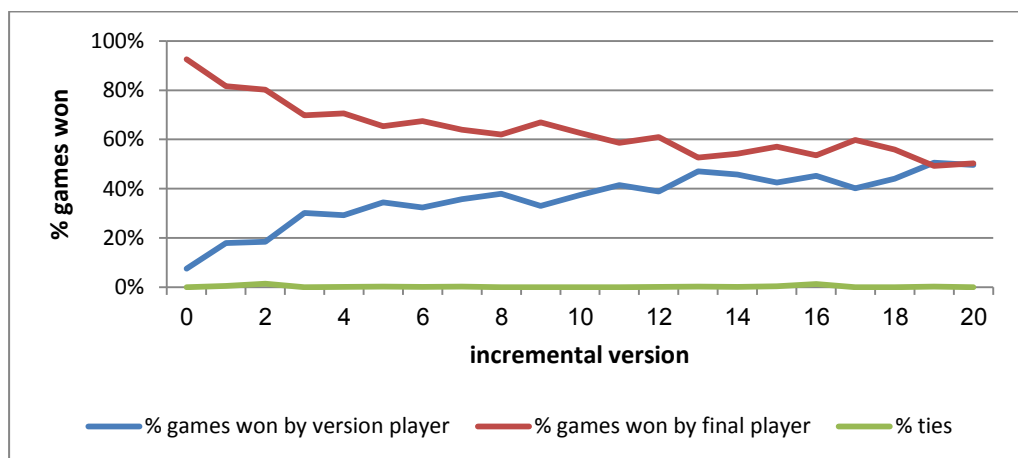
As the learning process progresses, we have noticed that the average number of turns per game increases. As we see it, the meaning of this increase is that the players are getting better, for example: the number of "easy wins" (4 in a column or a row with no attempt to block, for first player it is a win in 7 moves) decreases, the number of ties grows (a tie fills the board and takes 48 turns) etc.

The following graph shows the average number of turns per game calculated over 10k games for a 500k games learning sequence (50 different averages calculated every 500k/50 games).



Agent quality

We wanted to see that as the learning process proceeds, the player is getting better. The following graph represents the win distribution (wins for final player, wins for the incremental version players and ties) generated by a testing sequence that included 1,000 games (for each incremental version).



As can be seen in the graph, as the number of the incremental version increases the percent of games won by the intermediate version increases while the percent of games

won by the final player decreases. Meaning, the intermediate version players abilities are matching up to those of the final player.

Note: the final version was received after a training session the included 100k games (meaning, an incremental version was saved every 5k).

Learn parameters tuning

update function for the learning rate parameter α :

- Tested functions

1. $\alpha_1 = \frac{1}{1+\sqrt{t}}$

2. $\alpha_2 = \frac{1}{1+\sqrt[3]{t}}$

3. $\alpha_3 = \frac{1}{1+\sqrt{\frac{t}{2}}}$

4. $\alpha_4 = \frac{1}{1+\sqrt[4]{t}}$

5. $\alpha_5 = \frac{1}{1+\sqrt[3]{\frac{t}{2}}}$

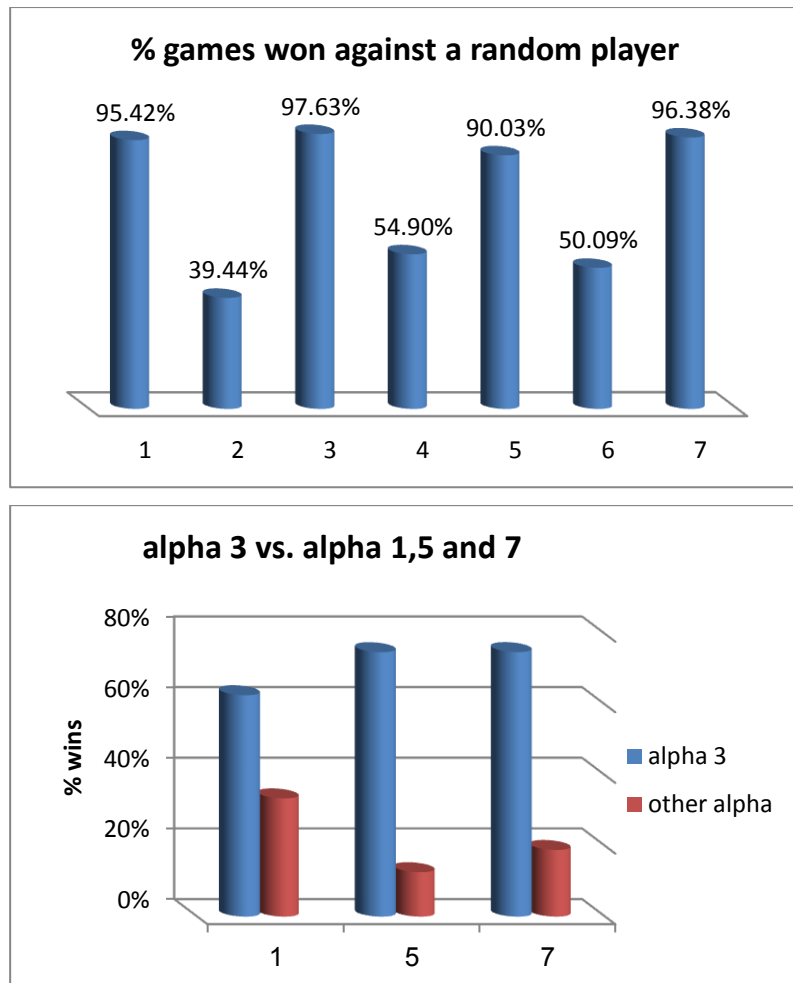
6. $\alpha_6 = \frac{1}{1+\sqrt[5]{t}}$

7. $\alpha_7 = \frac{1}{1000+\sqrt{t}}$

- Test Method

1. We ran the learning sequence for 2 million games for each update function
2. Each player was tested against a random player.
3. The best update functions (1,3,5,7) were tested against each other and update method 3 was better than all the others.

- Graphs



Choosing epsilon update method variables

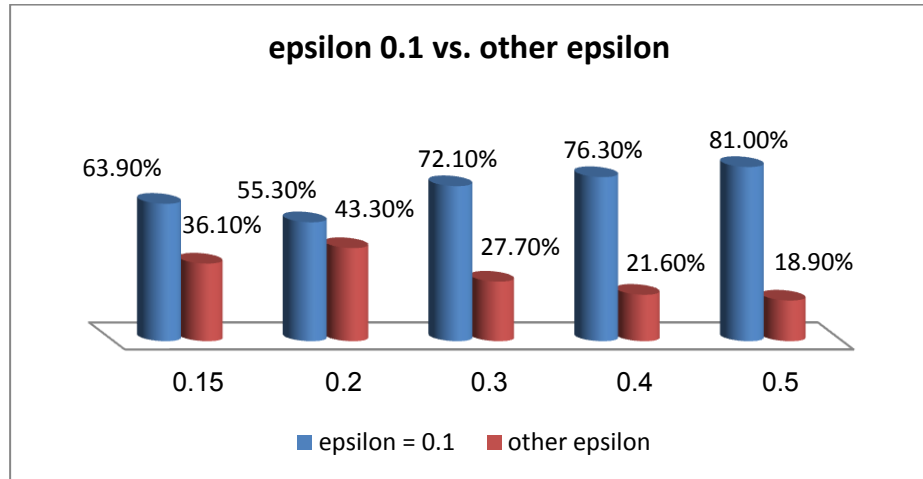
Each one of the three update methods has different parameters that influence the outcome of the agent (his quality). Those parameters include the initial epsilon value (for all methods) and the annealing factor for the annealing update method.

Constant epsilon method

- Tested values:
 1. $\epsilon = 0.1$
 2. $\epsilon = 0.15$
 3. $\epsilon = 0.2$
 4. $\epsilon = 0.3$
 5. $\epsilon = 0.4$
 6. $\epsilon = 0.5$
- Testing method:
 1. We ran the learning sequence for 500k games for each ϵ .

2. We tested (with 1,000 games) each of the players against the player with $\varepsilon = 0.1$. The results show that using $\varepsilon = 0.1$ is better than all the other ε we have tested.

- Graphs:



Annealing method

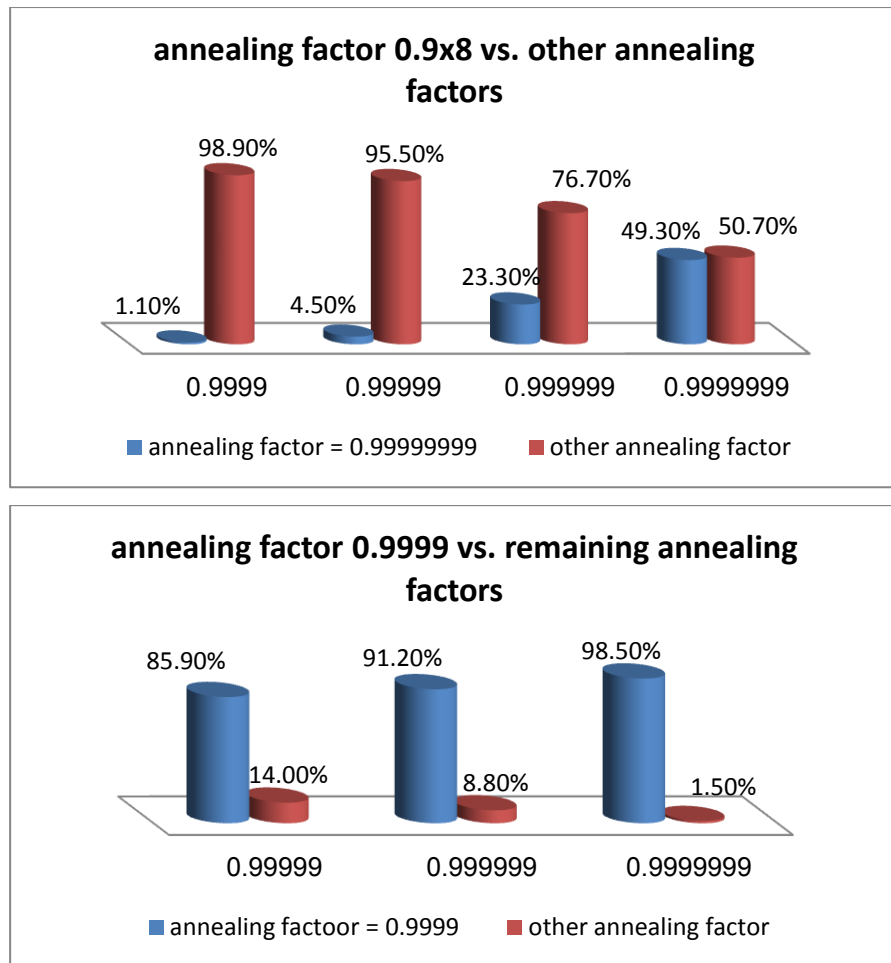
- Tested values:

1. Annealing factor = 0.9999
2. Annealing factor = 0.99999
3. Annealing factor = 0.999999
4. Annealing factor = 0.9999999
5. Annealing factor = 0.99999999

- Testing method:

1. We ran the learning sequence for 500k games with $\varepsilon^{(0)} = 0.9$ for each annealing factor.
2. We tested (with 1,000 games) each of the players against the player with annealing factor = 0.99999999. The results show that using any other annealing factor is better than using 0.99999999.
3. We continued to test annealing factor = 0.9999 against the remaining factor values. We chose to test next 0.9999 since it had the best results against 0.99999999. The results show that using annealing factor = 0.9999 is better than all the other annealing factors we have tested.

- Graphs:

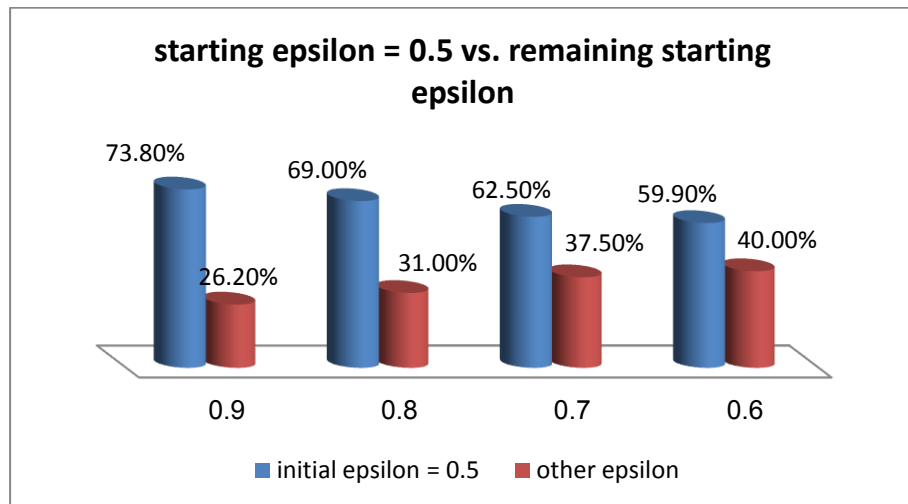
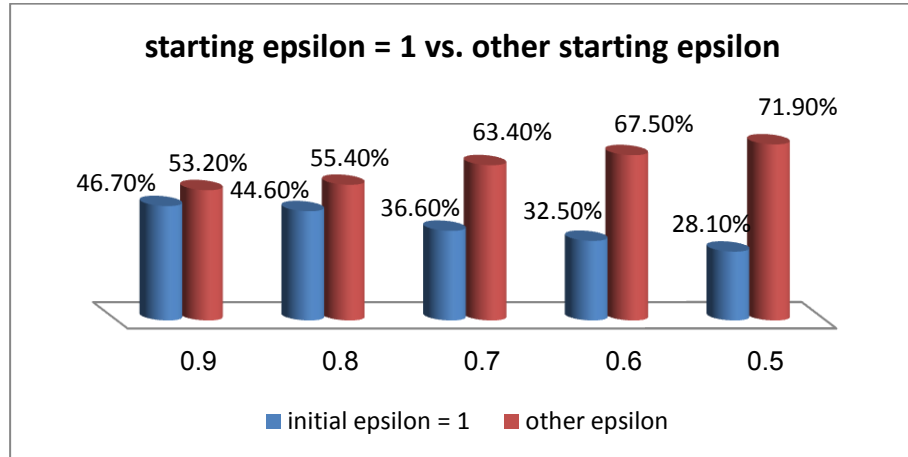


Number of games method

- Tested values:
 1. $\varepsilon^{(0)} = 1$
 2. $\varepsilon^{(0)} = 0.9$
 3. $\varepsilon^{(0)} = 0.8$
 4. $\varepsilon^{(0)} = 0.7$
 5. $\varepsilon^{(0)} = 0.6$
 6. $\varepsilon^{(0)} = 0.5$
- Testing method:
 1. We ran the learning sequence for 500k games for each $\varepsilon^{(0)}$.
 2. We tested (with 1,000 games) each of the players against the player with $\varepsilon = 1$. The results show that using any other $\varepsilon^{(0)}$ is better than using $\varepsilon^{(0)} = 1$.

3. We continued to test $\varepsilon^{(0)} = 0.5$ against the remaining values of $\varepsilon^{(0)}$. We chose to test next $\varepsilon^{(0)} = 0.5$ since it had the best results against $\varepsilon^{(0)} = 1$. The results show that using $\varepsilon^{(0)} = 0.5$ is better than all the other $\varepsilon^{(0)}$ we have tested.

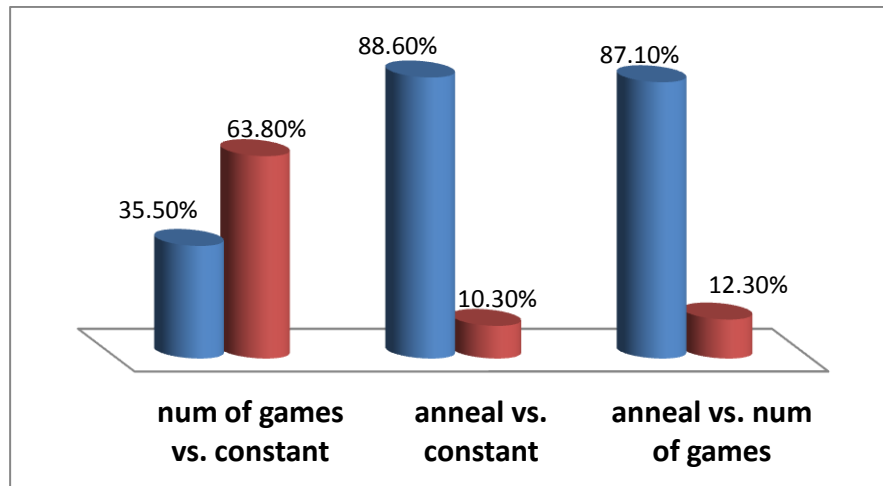
- Graphs:



Choosing ε update function

- We implemented 3 update functions:
 - $\varepsilon^{(t+1)} = \varepsilon^{(t)}$
 - $\varepsilon^{(t+1)} = c^t \varepsilon^{(0)}$
 - $\varepsilon^{(t+1)} = \frac{\varepsilon^{(0)} \log t}{\sqrt[4]{t}}$
- Test Method
 - We ran a learning sequence of 500k games for each update function. In the training session, we used the parameters achieved in the previous section, meaning:
 1. Constant $\varepsilon = 0.1$

2. Annealing update using $\varepsilon^{(0)} = 0.9$ and annealing factor = 0.9999
 3. Number of games update using $\varepsilon^{(0)} = 0.5$.
 - In order to choose the epsilon update function we tested each pair of functions against each other.
 - We ran a learning sequence
- Graphs



Initial Weights

In our basic implementation we chose to initialize all linear weights of the linear function approximations to 1. We realized that the weights should converge (after normalization) to values between $[-1,1]$ (As our rewards are 1,-1 for a win and a loss). We decided to change the initialization values to a random value distributed $N(0,1)$. The learning sequence using this method proved to be better (converges faster to better weights)

Issues recognized during development and testing

Normalizing weights

During the learning process we noticed weight values were becoming too large to handle. To solve this we added a normalization process to the weights. Because there are both negative and positive weights we couldn't just divide by the sum of weights so we divided by the margin between the minimal and maximal weights.

Feature selection

We tested several feature combinations and observed their weight at the end of the learning session. Features that we weighted near zero in all linear approximators were sifted out. Such features are: completion to 3,4 in the second top empty row of each columns. We thought that this feature might give our agent a sort of look-ahead ability, but this features weights were negligible after the training process and were sifted.

Hard coded blocking made the player stupid

We tried adding to our bootstrapping hard coded move policy a blocking reflex. i.e. when the policy notices a threat it blocks it and bypasses the Q-Value evaluation. This turned out to be a major mistake as it caused the player to avoid receiving negative rewards for states that lead to a loss, thus when in actual play it didn't block.

Bias to left columns

After several learning processes we noticed a bias towards taking actions at left columns. We understood that at the start of each training all states has equal value. Our greedy mechanisms thus choose the left columns more frequently and caused our agent not to explore starting positions in the middle and left of the board. We fixed this by adding a random element when choosing between equal positions.

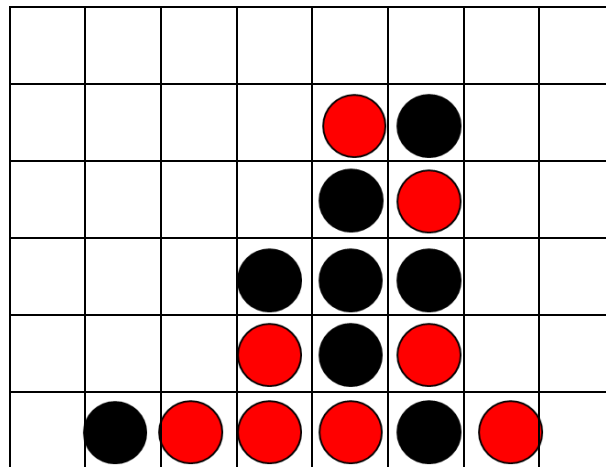
Q value 0 for Empty board

All our features are 0 when the board is empty. So there was no Q-value assigned to the empty board and either action, This means the agent has no preference regarding a starting move. We initially solved this by running the agent (in test and play mode) with a minimax look ahead tree. We also added a "Bias" weight for our linear approximators enabling us to learn the best starting move.

Agents behavior when faces a definite loss

When a situation occurs where for any action the agent will choose he will lose, all actions are equal to each other (value wise) and it seems as if the agent chooses the action randomly. Meaning, when there is a definite loss ahead, the agent might not even try to block one of the existing threats since he knows it won't change the final upcoming result. For illustration, in

the following board the red player has 2 immediate threats. Therefore, he might choose any column and not necessarily columns 3 or 5 which block.



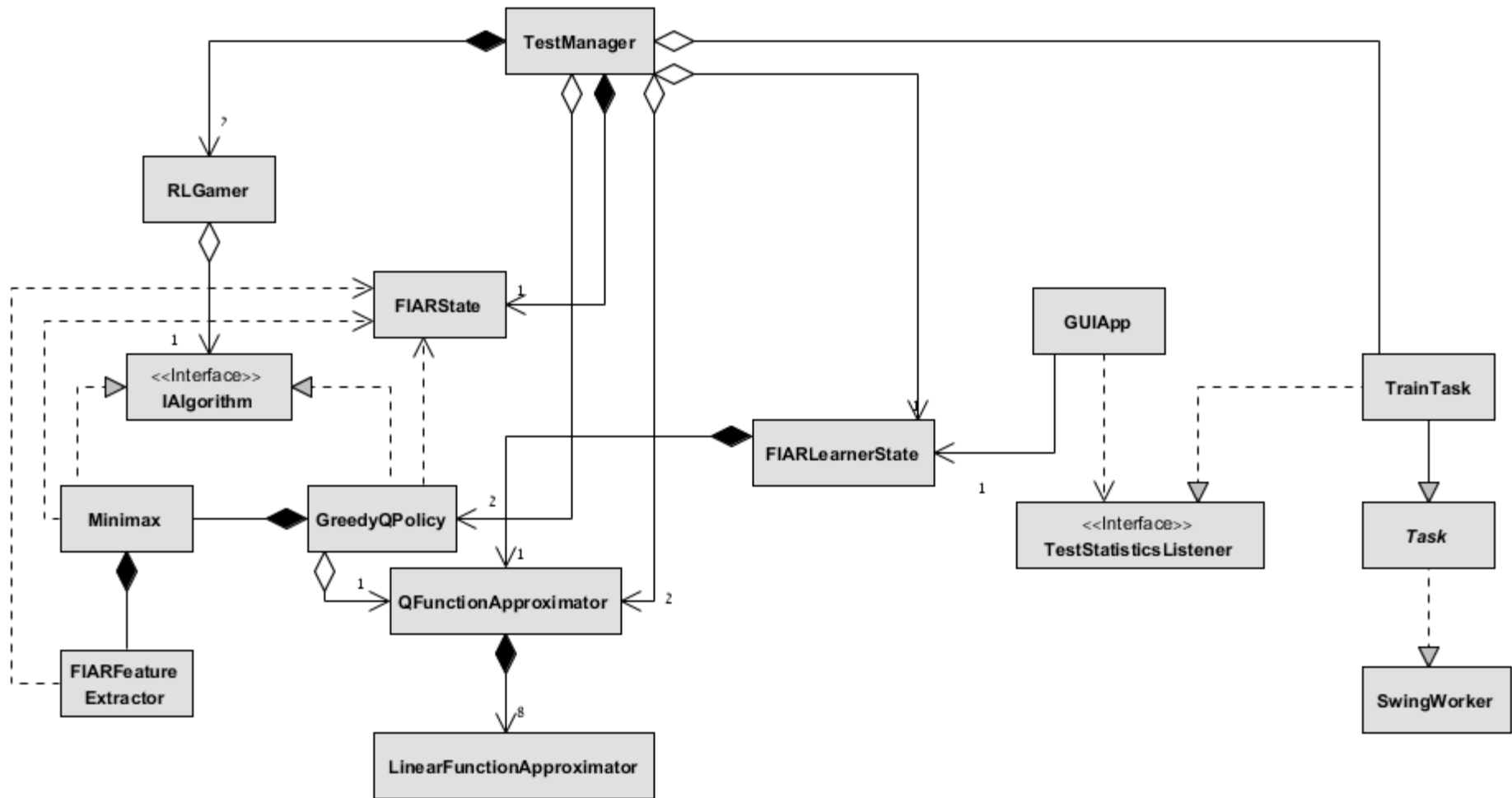
Detailed Class Description

See java doc at: <http://www.cs.tau.ac.il/~nurlan/rlworkshop/javadoc>

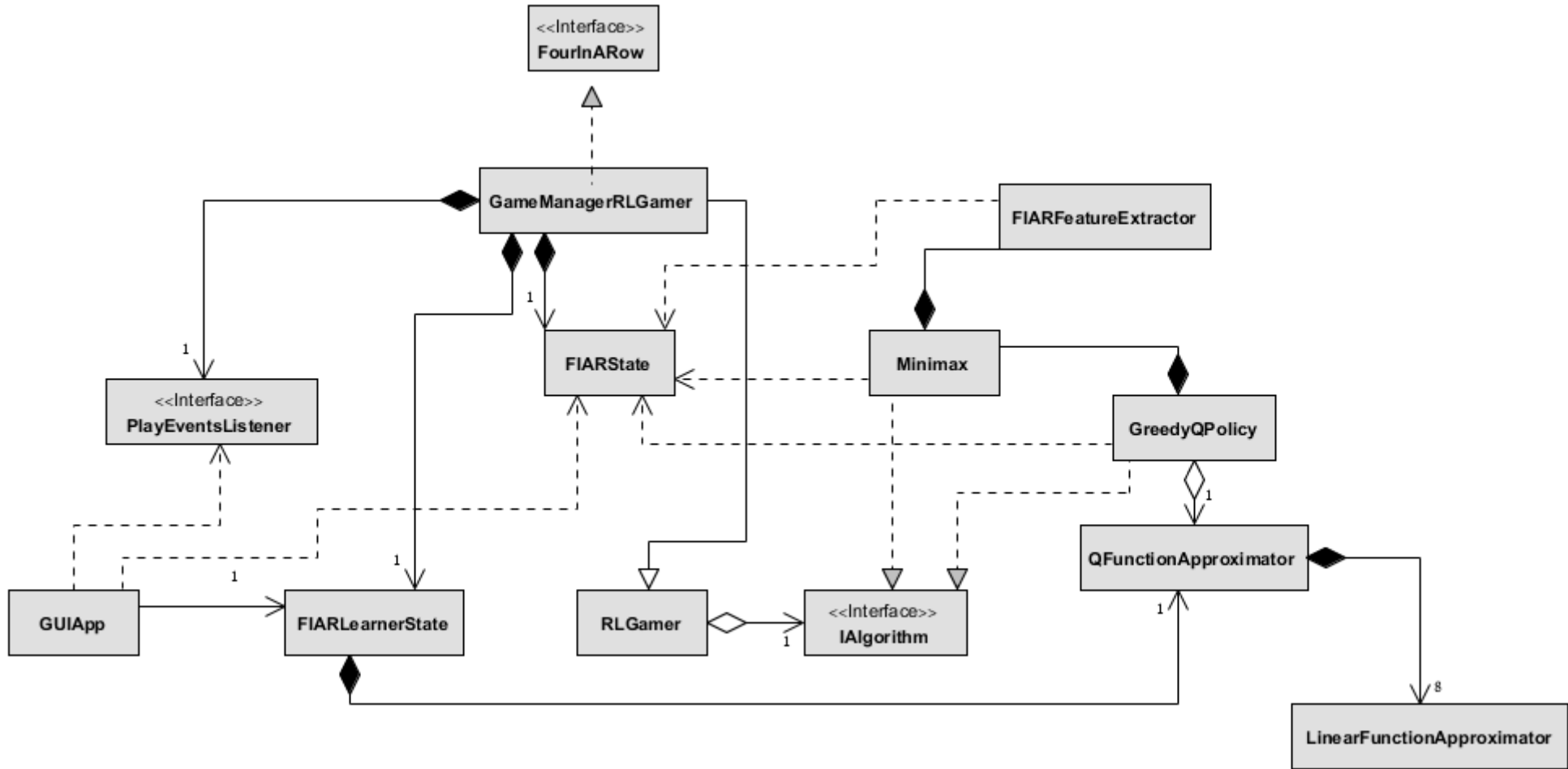
Learning process Class Diagram



Testing process Class Diagram



Playing process Class Diagram



Sequence Diagrams

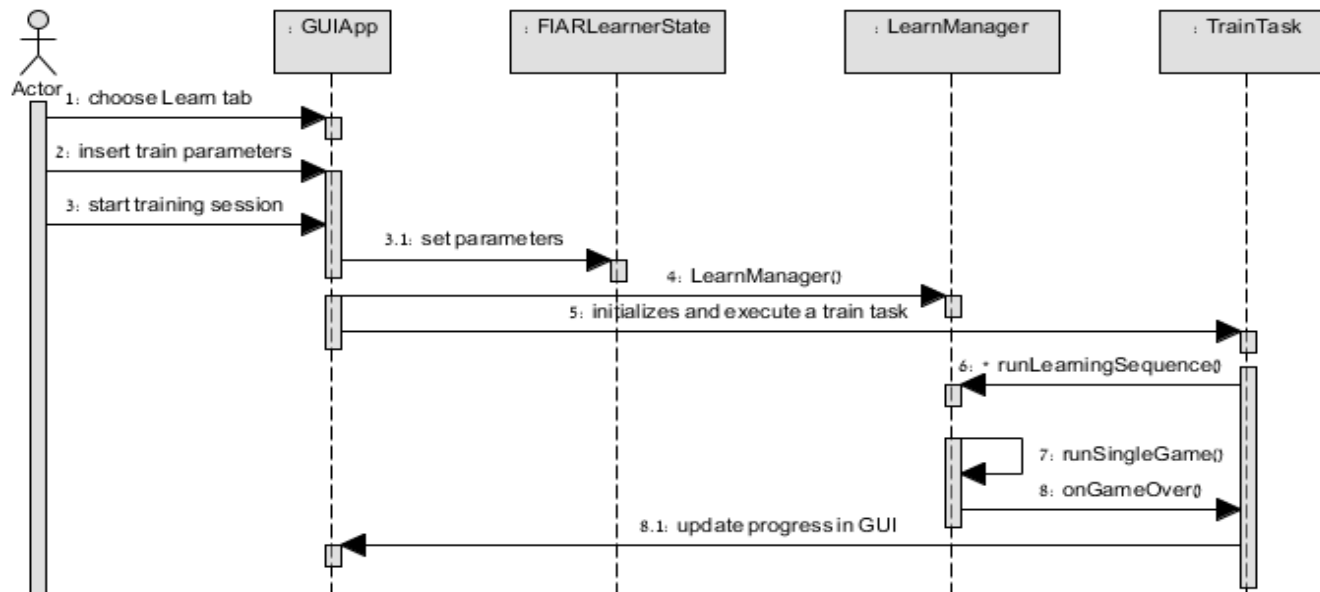
Use Case: Test GUI

1. User chooses to player configuration files to compare
2. User pushes run test

The sequence diagram is identical to the train use case, replace TrainTask with TestTask and LearnManager with TestManager.

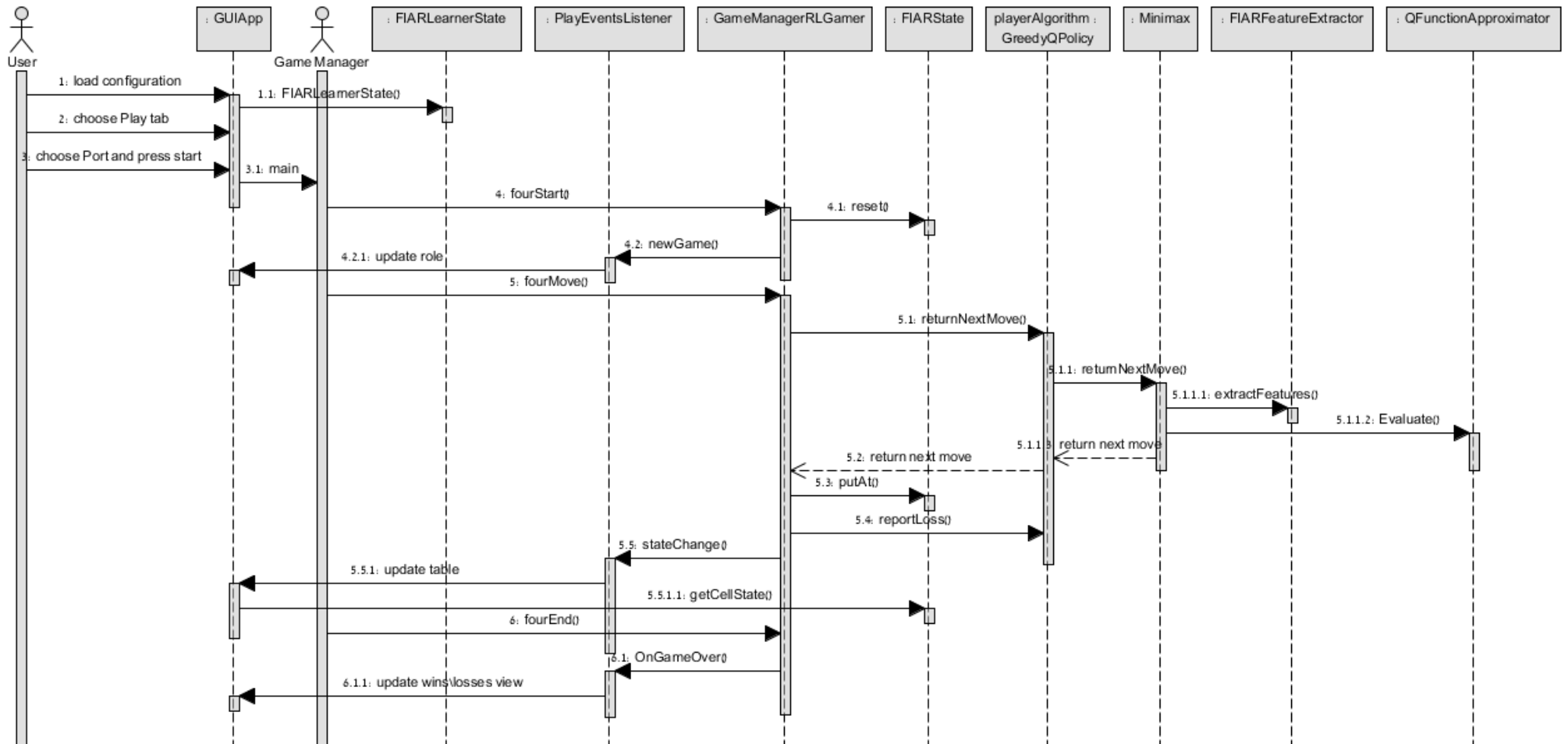
Use Case: Train GUI

1. User chooses learn parameters
2. User chooses log file name
3. User chooses number of games to play
4. User saves initial version in order to name the configuration
5. User chooses whether he wants incremental versions to be saved during the learning sequence
6. User pushes Start Training
7. User saves configuration after training is over



Use Case: Play GUI

1. user loads a player configuration
2. user chooses a port number
3. user press start



Use Case: Run single game in train mode

