A Comparison Of Reinforcement Learning Algorithms Using Blackjack*

N Miri[1], V Nagisetty[2], and S Parson[3]

***Abstract*— This paper explores the application of two major reinforcement learning algorithms to a popular card game and reports the results obtained with the goal of seeking an optimal strategy. Monte Carlo (MC) methods are pitted against Temporal Difference (TD) methods with both on-policy and off-policy control in a battle for fastest time to convergence. The results lead us to infer that MC is the best fit for the environment due to ease of implementation, fastest time to convergence, and the smallest deviation from known best strategies.**

I. INTRODUCTION

In this project, we aim to develop an optimal playing strategy for the popular game Blackjack. The game is commonly played at casinos worldwide. The rules are straightforward and it serves as an interesting model to compare reinforcement learning algorithms. Which reinforcement learning algorithm produces the best results with minimal parametric tweaking? This is the question that we aim to answer.

II. PROBLEM DESCRIPTION

A. Overview of the Problem

To establish an objective performance measure, the algorithms are independently used to learn the best actions (hereafter referred to as the policy) to take in a simplified model of the game.

The resulting policy is then applied to play many Blackjack games in the same environment and the loss record is used as a measure of performance. An algorithm that results in a lower loss record is considered to have performed better. It is important to note that we specifically chose to compare loss rates, since it eliminates the problem of two separate metrics for win and draw.

The reinforcement learning algorithms and their performance are compared and the result will be used to identify the algorithm that works best in our model.

A graphical user interface of Blackjack is also developed, displaying the optimal strategy produced by the algorithms as a guide for the player to use.

*This work was created for COMP 3200.

[1]V. Nagisetty is an undergraduate student with the Department of Computer Science, Memorial University, St. John's, NL A1C5S7, Canada vnagisetty@mun.ca

[2]N. Miri is an undergraduate student with the Department of Computer Science, Memorial University, St. John's, NL A1C5S7, Canada nm6268@mun.ca

[3]S. Parson is an undergraduate student with the Department of Computer Science, Memorial University, St. John's, NL A1C5S7, Canada k47swp@mun.ca

B. Rules and Objective of Blackjack

Blackjack is one of the most widely played casino game in the world. In Blackjack, one or more players play against a dealer, who deals one or more decks of 52 cards. Each card has a value. Number cards {Two to Ten} have the value corresponding to their number, and face cards {Jack, Queen, King} are worth ten. Ace has a value of either 1 or 11, depending on which is more beneficial to the person holding Ace. The suits of each card are not important to this game.

The game starts with each player placing an initial bet. The dealer deals two cards to each player and places one card face up. The objective of the game for each player is to beat the dealer and maximize their money earned. There are several ways this can happen:

- Achieve a final score higher than the dealer without exceeding 21.
- Reach a score of exactly 21, while the dealer does not.
- Let the dealer receive a score of over 21.

If the player wins, they make double the amount of the initial bet. If the dealer wins, the player loses the entire bet. If both the player and dealer have the same total, it is considered a "push" and no money is won or lost that round.

There are several actions that a player can take in the game of Blackjack:

- **Hit**: prompts the dealer to deal one more card to the player.
- **Stand**: prompts the dealer to play out the game using a fixed casino policy.
- **Double Down**: this action acts as a macro action. It performs a Hit, then Stand for the player, and doubles their bet.
- **Split**: possible only if two of the same cards are initially dealt. The dealer would then deal two more cards, splitting the initial hand to two hands. The player then plays both hands independently.

This project uses a simplified version of the game. The number of players is capped at one, and a bet of one unit is assumed. This would not affect the optimal policy, since one hundred games played with a bet of one dollar is equivalent to having one game played with a bet of one hundred dollars, and so on. Conversely, playing several games with one player is equivalent to playing one game with several players.

As for the actions, split is not implemented. This could change the results very slightly, considering that the act of splitting in some scenarios would result in higher reward. However, the policy learned should still be very close to

optimal since the best actions to take given a certain combination of player hand and dealer card stays the same.

C. Properties of Blackjack

There are several key properties of the Blackjack environment that makes applying reinforcement learning algorithms interesting.

This project considers a state of the game to be a function of the value of player's hand and the card shown by the dealer. A player hand with an Ace is considered to be a separate state, since the best action to take might differ in that scenario due to Ace having a value of Eleven or One.

A reward of **+1** is assigned for a won game, **-1** for a lost game and **0** for a tied game. A Double-Down action doubles the reward.

Other key properties of Blackjack are:

- **Partially Observable**: the player can only see one of the dealer's card. This makes it an Imperfect Information Game.
- **Stochastic**: the cards are shuffled and dealt at random.
- **Episodic**: each game is considered as an episode, and does not depend on the previous game.
- **Static**: the game is turn-based, and waits for the player to make a move before continuing.
- **Discrete**: there is a discrete set of percepts and actions.
- **Single Agent**: the dealer has a fixed policy, therefore only the player is the agent in the game.
- **Complete Information**: the game rules and actions are all known.

Some screenshots of the states in Blackjack GUI in this project include:

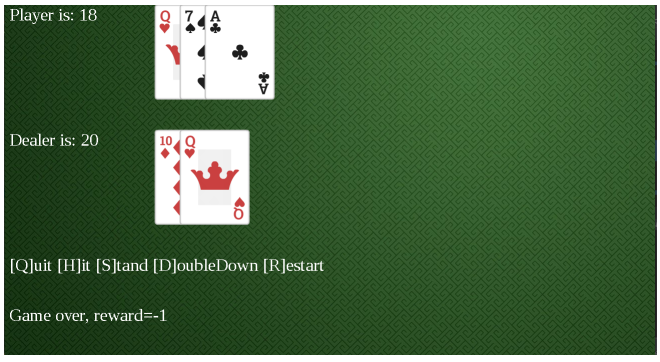

Fig. 1. Screenshot of a Lost Game

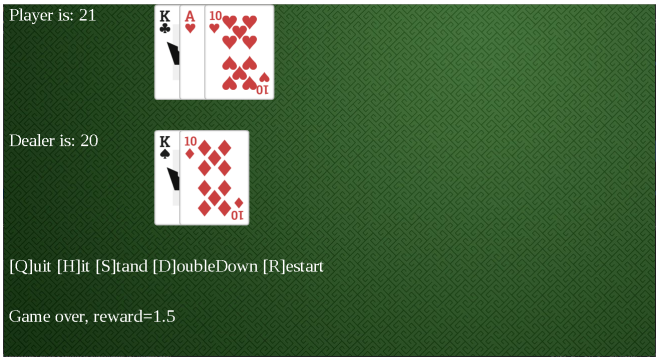

Fig. 2. Screenshot of a Won Game

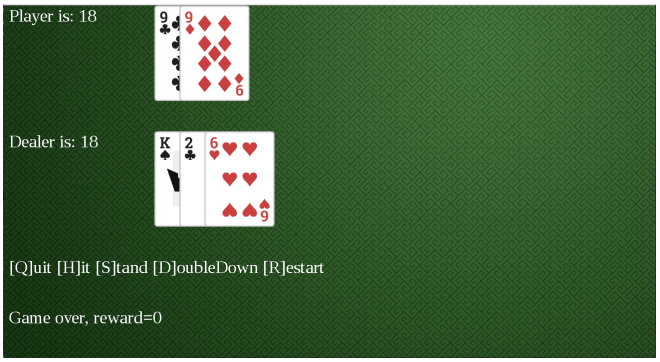

Fig. 3. Screenshot of a Drawn Game

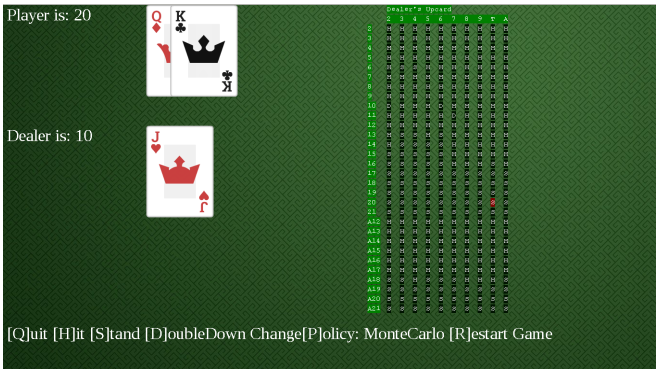

Fig. 4. Screenshot of a Sample Game with the Policy

III. METHODOLOGY

Reinforcement learning is an area of machine learning inspired by behavioural psychology. The idea is for a software agent to learn via interaction with its environment. Agents perform actions, and then receive rewards from the environment based on the action. The goal of an agent is to learn how to maximize their cumulative reward.

At a practical level, reinforcement learning methods specify how to estimate the value of an action in a given state. They can be broadly classified into On-Policy and Off-Policy control methods. On-Policy methods refer to using the same policy to generate data and learn from that data. An action selection strategy is required for On-Policy methods, which

is discussed in the next section. The agent in an Off-Policy control method uses a policy to generate data and a separate policy to learn from that data.

Two major reinforcement learning methods are Monte-Carlo and Temporal Difference control methods. Monte-Carlo methods require the environment to be episodic, and update value estimates at the end of each episode. Temporal Difference control methods are more flexible since they can be applied to both episodic or continuous environments.

There are various parameters passed on to the reinforcement learning methods:

- **Step Size**: representing how much a new value estimate affects the existing value in a given state-value pair. A higher step-size parameter would "pull" the result towards the new estimation more. It is beneficial to have a relatively higher step-size in a dynamic environment so the agent learns the changes made better. This project uses a low step-size of **0.05**, since this is a static environment and to reduce the effect of noise, i.e. a rare card. To offset this, many more games are played.
- **Discount Factor**: this determines the present value of future rewards. This is only applied in situations where the environment is treated as a continuation task, i.e. Temporal Difference methods. If the discount factor is 0, the agent only cares about the next reward. If the discount factor is 1, the formula is similar to episodic task formula. This project uses a discount factor of **1** since the environment is episodic.
- **Average Update Rule**: this refers to updating value estimates based on the average of all rewards. Practically, this is done by using a dynamic step-size, equal to **1/(number of state-action pair occurrences)**. This project runs some experiments using an average update rule for all reinforcement learning methods considered.

This project focuses on Every-Visit Monte Carlo On-Policy Method, SARSA Temporal Difference On-Policy method and Q-Learning Temporal Difference Off-Policy to cover a wide range of reinforcement learning methods.

In this project, the reinforcement learning agent learns by playing a certain number of Blackjack games using the simplified Blackjack model created. The agent uses one of the three reinforcement learning methods. The agent selects an action to take based on the action-selection strategy. Once the game has ended, the policy, or the probability of a given action being the best choice is updated based on the value estimations. Code for the agent and all three reinforcement learning methods are found in the file "Controller.py"

A. Action Selection Strategy

Selection of an action based on the value estimate of a state addresses an important concept known as "exploration vs. exploitation".

Exploration refers to the sampling of the various actions in a state to estimate how good that action is. This is important for a reinforcement learning agent to learn which action is ideal in a given state.

Exploitation refers to applying the action believed to be the best one in the current state. This is important for a reinforcement learning agent in order to maximize the reward.

A good action selection strategy must balance exploration and exploitation. If the agent does not explore enough, it produce suboptimal results. Exploration is necessary to see if some suboptimal actions will result in an optimal reward further in the game. This project uses two distinct ways of selecting actions: Epsilon-Greedy and Upper Confidence Bound selection methods.

- The idea behind Epsilon-Greedy selection method is to select a random action with a low probability. The rest of the time, it selects the currently believed best action. This probability, epsilon, is passed on as a parameter. This addresses the exploration versus exploitation problem since the action chosen is exploitative most of the time, while still exploring at a small probability.
  In this project, the epsilon value used is **0.15**. This means that a random action is selected 15% of the time, and the currently believed best action is chosen more than 85% of the time. This found in the select_action_epsilon function in the "Controller.py" file.
- The idea behind Upper Confidence Bound selection method is to introduce an uncertainty factor. An action is selected among sub-optimal actions that may be close to optimal, with a measure for how certain the agent is about their values. An exploration constant is used to control the degree of exploration.
  This addresses the exploration vs. exploitation problem. The optimal action is selected most of the time, while still selecting the non-optimal action occasionally. This non-optimal action is chosen based on how certain we are of the range of the reward it produces.
  In this project, the exploration constant used is **2**. This found in the select_action_ucb function in the "Controller.py" file.

B. Every-Visit On-Policy Monte-Carlo Control Method

Monte-Carlo refers to a popular set of methods used for estimating the value of a state. A model of the environment is not required for these methods, although the agent in this project has access to the model. Monte-Carlo methods require the environment to be episodic, as well as recording the state, action taken at that state, and the reward received from that action. Once the agent has reached a goal or a terminal state, the episode is considered to have ended. The sequence of state, action and rewards generated throughout the episode are used to estimate a value of a given state, action pair.

Monte-Carlo methods can be broadly classified into two categories: First-Visit and Every-Visit. In First-Visit, the value of a state-action pair is only estimated the first time it is visited in a given episode. In Every-Visit, the value of a state-action pair is estimated every time it is visited.

This project uses Every-Visit Monte-Carlo method. The game model is built in such a way that a state is only

encountered once in a given game. Any action taken at a state either ends the episode, or moves the agent to a new state that has not been visited in the current game. This project uses On-Policy control method.

mc.jpeg

Pseudocode for Every-Visit MC Method

```
1. Function EveryVisitMC (policy π)
2.  values[s] = initial value for each state
3.  returns[s] = empty list for each state
4.  while(true):
5.      generate episode π (S0, A0, R1, S1,...,St-1, At-1, Rt)
6.      totalreward = 0
7.      for (t= T-1 -> 0):
8.          totalreward = totalreward + reward t+1
9.          append totalreward to returns[state t]
10.         values[state t]  = average(returns[state t])
```

Fig. 5. Pseudocode for Every-Visit On-Policy Monte-Carlo Learning

This project uses several optimizations for this method:

- **Uniformly Distributed Exploring Starts**: this refers to generating random states with a uniform distribution so that the value estimations converge faster. This effectively ensures rarer states, such as 2 twos, are generated with a higher frequency. This would help reduce the number of iterations needed for a policy to converge.
- **Using Value of a Hand as a State**: this reduces the state space by a huge amount, without sacrificing optimal policy. This results in a more efficient complexity and a game can be run faster.
- **Checking for Convergence**: this refers to finding out if a policy has converged. Every 10,000 iterations, the agent checks if the estimated values have converged, i.e., not changed by over **0.15**. If the estimated values have converged, the agent stops running more iterations, effectively making the algorithm more efficient.

The parameter taken is step-size which is either **0.05**, or an average update value of **1/(number of state-action pair occurrences)**.

The value estimation is found in the update_values_mc function in the ”Controller.py” file.

C. SARSA On-Policy Temporal Difference Control Method

SARSA is an On-Policy Temporal Difference Control method. The name comes from the sequence of **S**tate, **A**ction, **R**eward, next **S**tate, and next **A**ction which are all needed to estimate a value of a given state-action pair.

sarsa.jpeg

Pseudocode for SARSA

```
Initialization():
1.  Q[s][a] = 0
Repeat Forever (for each episode):
1.  S = initialize starting state for episode
2.  A = choose action using action selection strategy
3.  Repeat (for each step of episode):
4.      R, S' = Take action A at state S
5.      A' = Choose action using action selection strategy
6.      Q[S][A] = Q[S][A] + α*(R + γQ[S'][A'] − Q[S][A])
7.      S = S', A = A'
```

Fig. 6. Pseudocode for SARSA On-Policy Temporal Difference Learning

Since SARSA is an On-Policy control method, it uses the same policy to generate data and learn from it. Either of the action selection strategies, Upper Confidence Bound, or Epsilon-Greedy, can be used for SARSA in this project.

The parameters used by SARSA are step-size and discount factor. Step-size is either **0.05**, or an average update value of **1/(number of state-action pair occurrences)**. A Discount factor of **1** is used.

The value estimation is found in the update_values_sarsa function in the ”Controller.py” file.

D. Q-Learning Off-Policy Temporal Difference Control Method

Q-Learning is an Off-Policy Temporal Difference Control method. Since it is an Off-Policy control method, it uses a policy to generate data and a separate policy to learn. In this project, Q-Learning uses either Upper-Confidence Bound or Epsilon-Greedy action selections from its policy to generate data, while the optimal action from its policy is used to estimate a value of a given state-action pair.

ql.jpeg

Pseudocode for Q-Learning

```
Initialization():
1.  Q[s][a] = 0
Repeat Forever (for each episode):
1.  S = initialize starting state for episode
2.  Repeat (for each step of episode):
3.      A = choose action using action selection strategy
4.      R, S' = Take action A at state S
5.      Q[S][A] = Q[S][A] + α*(R + γmaxaQ[S'][a] − Q[S][A])
6.      S = S', A = A'
```

Fig. 7. Pseudocode for Q-Learning Off-Policy Temporal Difference Learning

The parameters used by Q-Learning are step-size and discount factor. Step-size is either **0.05**, or an average update value of **1/(number of state-action pair occurrences)** A Discount factor of **1** is used.

The value estimation is found in the update_values_ql function in the ”Controller.py” file.

E. Possible Improvements

We suspect that there may be a bug in the temporal difference algorithm since the graph of the loss rate for both SARSA and QL do not seem to converge asymptotically. It is possible that they need many more iterations to converge, however given the initial downward trend it is more likely that this is due to a bug, wrong parameter, or our initial assumptions are suspect.

Given more time it would have been possible to debug more thoroughly and explore the results, however we are satisfied with the convergence of the Monte Carlo algorithm.

Python may not be the ideal language for these computations, since the computation rate for the algorithm is only about 16,000 iterations per second on a modern CPU (Core i5-3320M).

We suspect that the switch to a compiled language would speed up this process substantially. One other factor is that the floating point operations are orders of magnitude slower than their integer counterparts. In another language, we would have better control over this.

One other thing that would help, is that if we were to create a GUI that would plot the loss rate over the natural logarithm of the iterations in real time, so that we can tweak the parameters live.

In short, here are the improvements that we are confident would improve our results:

- More debugging time
- Switch to integer based math
- Switch to compiled language
- Create a GUI to plot the results

IV. RESULTS & DISCUSSION

A. Description of the Experiments

The experiments that we used to plot our result began with the idea to generate a policy using a given reinforcement learning algorithm with a distinct set of parameters. Subsequently, we increase the number of iterations of the RL algorithm so that we could plot the game loss rate. Our algorithm for this process was as follows:

- 1. Set n to be the number of iterations (1000 to start)
- 2. Generate policy with n iterations
- 3. Play 100,000 games with this policy and record the loss rate
- 4. Double the value of n.
- 5. If n $\leq$ 5,000,000, go to 2

We performed this process for Monte Carlo with the following parameters (illustrated in Figure 7):

- Monte Carlo, Epsilon Greedy = 0.15, Average Update
- Monte Carlo, Epsilon Greedy = 0.15, Step = 0.05
- Monte Carlo, UCB C=2, Average Update
- Monte Carlo, UCB C=2, Step = 0.05

Likewise, our parameters for SARSA (illustrated in Figure 8):

- SARSA, Epsilon Greedy = 0.15, Discount=1, Average Update
- SARSA, Epsilon Greedy = 0.15, Discount=1, Step = 0.05
- SARSA, UCB C=2, Discount=1, Average Update
- SARSA, UCB C=2, Discount=1, Step = 0.05

And finally, our parameters for Q Learning (illustrated in Figure 9)

- QL, Epsilon Greedy = 0.15, Discount=1, Average Update
- QL, Epsilon Greedy = 0.15, Discount=1, Step = 0.05
- QL, UCB C=2, Discount=1, Average Update
- QL, UCB C=2, Discount=1, Step = 0.05

B. Results and Discussion of the Experiments

It is clear from the graphs that Monte Carlo is the best choice given the asymptotic convergence of the loss rate. From the graph it appears that this will happen at around 1,000,000 iterations. Given our test setup, this will occur in about one minute, which is acceptable performance for a static policy map. The loss rate does not substantially deviate with more than 1,000,000 iterations. Our experimental convergence function confirms this result, reporting no substantial change of each state-action value estimate after about 1,200,000 iterations. The convergence of the algorithm could be measured by many metrics, so we have designated this as a subjective measure.

C. Temporal Difference Algorithm

As previously mentioned, we believe that there is a bug in the implementation of our temporal difference algorithm, since the loss trend is initially a downward slope, but then becomes unstable after about 60,000 iterations. This behavior is common to both SARSA and QL, which differ in their state-value estimate update characteristics. This lends more evidence to the idea that the graph is illustrating a bug, however it is possible that this is the normal behavior of the algorithm. Despite this, the results are acceptable given that it still only loses about half of the time. Given more time we would explore this further by including more data points and using statistical methods (such as ANOVA) to see if there is a real convergence problem. It is possible that the variance in the graphs are due to graph scaling and a lack of data points.

Monte Carlo:

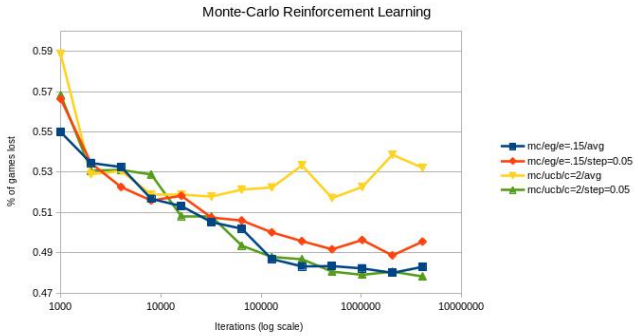

Fig. 8. Results of Experiments using Monte-Carlo Learning

SARSA:

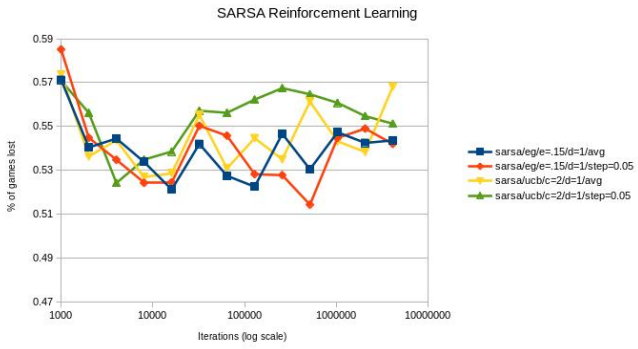

Fig. 9. Results of Experiments using SARSA Learning

Q-Learning:

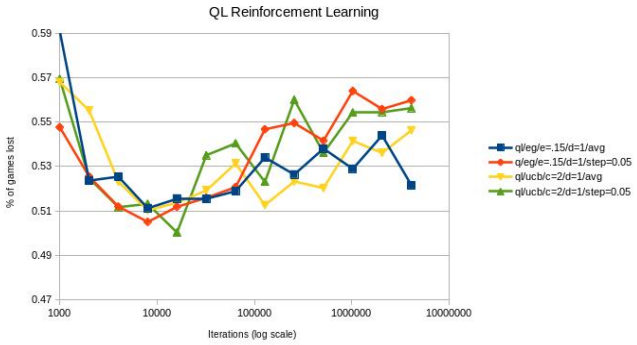

Fig. 10. Results of Experiments using Q-Learning

V. CONCLUSION

This project was successful in creating a working model of a simplified Blackjack game, applying several reinforcement learning methods, and in comparing them via an objective performance measure. Another successful task was updating different policies to the view so the player could see what action to take in a given state of the game.

However, as previously mentioned, SARSA and Q-Learning methods did not produce intuitive results, i.e. the loss rate increased after a certain number of iterations. Due to time-constraints, this was not investigated further. Had there been more time, investigating the cause and updating the project would be the first priority.

Incorporating a Split action was also considered, but eventually dropped due to time constraints. Adding that feature to the model would make for a more realistic policy that would reflect Blackjack game better.

A further improvement would be to generate a policy in real-time based on the parameters and reinforcement learning method through the view. Currently this is generated real-time via the ”TestModel.py”, but not using the view, which would require using multiple threads for concurrent processes.

A final improvement would be to fine-tune the parameters for each reinforcement learning method to find the best performing version of each, and then compare them.

Overall this project is considered a success, and established the fact that Every-Visit Monte-Carlo learning method performs better than the other two in this simplified Blackjack environment.

REFERENCES

[1] http://www.cs.mun.ca/ dchurchill/courses/3200/slides/

[2] Schiller M.R.G., Gobet F.R. (2012) A Comparison between Cognitive and AI Models of Blackjack Strategy Learning. In: Glimm B., Krger A. (eds) KI 2012: Advances in Artificial Intelligence. KI 2012. Lecture Notes in Computer Science, vol 7526. Springer, Berlin, Heidelberg.

[3] A. Perez-Uribe and E. Sanchez, ”Blackjack as a test bed for learning strategies in neural networks,” 1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98CH36227), Anchorage, AK, 1998, pp. 2022-2027 vol.3.

[4] Noonan, Robert Edward, ”Comparing AI Archetypes and Hybrids Using Blackjack” (2012). Theses, Dissertations, and Other Capstone Projects. Paper 336.