# Using Reinforcement Learning to Play Blackjack

**Advanced Artificial Intelligence (CS 6700)**

Authors Redacted for Privacy Considerations

**Abstract**

*With the emergence of AlphaZero, built by DeepMind, to master the games chess, shogi, and go, reinforcement learning has become a large avenue to address performing tasks autonomously. In this project, reinforcement learning was used to try to beat the performance of a random player in Blackjack. Four different reinforcement learning approaches were used and compared with each other to find the best performing solution; Deep Query Network (DQN), Dueling DQN, Deep Convolutional Query Network, Q-learning.*

## Introduction

Reinforcement learning is known as one of the three subsections of deep learning. It differs from supervised learning in that it does not require the use of already labelled data about inputs and their correlated outputs. Instead, the agent assigned with learning the task explores its environment during the training process. If a successful action was made by the agent, the agent receives a reward. Otherwise, it receives no reward. In some instances it could be penalized for performing the action. In a game, the task normally is to win and the agent learns to win the game by exploring all possible outcomes and creates a strategy for the best possible outcomes.

Blackjack is a popular banking card game with a fixed set of rules the player and dealer must follow. Blackjack is played with one or more standard decks of cards that do not contain joker cards. At the start of each round, the player must place a wager and then the dealer deals two cards face up to the player and keeps two face down. The dealer must then reveal one of its cards (turn it face up) and give the player an opportunity to either “hit,” “stand,” “surrender,” “double down” or “split.” The goal of the player is to get as close to 21 as possible without going bust (going above 21). The dealer must hit if after revealing its cards, its total is less than 17. If the total is 17-21, the dealer must stand, and if the total is over 21, the dealer “busts.”

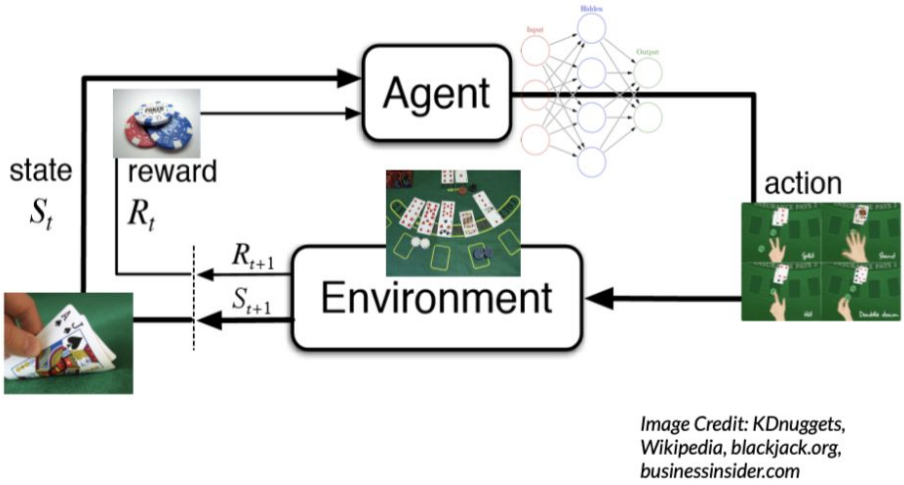
Hitting gives the player another card whose value is then added to the initial value of the player's displayed cards. Kings, Queens and Jacks each have a numerical value of 10 whereas Aces can either have a value of 1 or 11 depending on whichever value gets the total score of the player closest to 21 without busting. Standing fixes the player's total for that round. If a player surrenders, they get to keep 50% of the value of the wager. Doubling down can only be performed after the first two hands have been dealt. When a player doubles down, they add a second wager to their initial wager, receive a card face up and stand immediately. A split is possible only if the player has two cards that hold the same value eg. two 7's, two 3's, etc. When this happens, the player can choose to place an additional wager equal to the value of the first wager and will receive a new card. From this point, they can either stand or hit. If a player is dealt an Ace and a 10-valued card at the start of a round, the player is said to have a Blackjack if they choose to stand at the end of that round. The player receives 1.5 times the value of the wager. At the end of each round, if the player's total is closer to 21 than the dealer's total is, the player wins otherwise the dealer wins. If the dealer's total is equal to the player's total, no one loses and we have a "Push".

The goal of this project was to use reinforcement learning, with neural networks, in an attempt to achieve a performance better than an average human player that relies on blind luck or an AI player that plays using Q-learning. The game is played with an AI player against a dealer in the pre-modeled OpenAI gym environment. No other parties are playing. The dealer is allowed to deal cards, store the value of its own cards, maintain a store of wagers, respond to player requests to hit or stand. The AI player is able to keep a store of the total value of its cards, make requests to hit or stand, and place and update wagers.

For the Blackjack environment utilized in this project, OpenAI Gym's implementation which had several notable differences from normal Blackjack. Namely it restricted the action space to hit or not hit, excluding betting and splitting. They also do not utilize a real deck of cards instead simply generating cards through random choice among the thirteen different cards. The reward values and episodes that follow are based off of OpenAI Gym's environment. The reinforcement learning approach to training the AI player is quite simple. The agent is given a state, which represents what the sum of its cards is, the sum of the dealer's cards, and if it has a usable ace. The agent then performs an action, either hitting or staying. The OpenAI environment then

indicates the next state of the game and the agent plays the game until it ends. When the game ended, the AI player would receive a reward of +1 if it won, 0 if it tied with the dealer, or -1 if it lost.

Three different architectures were used to demonstrate the capability of neural networks in reinforcement learning and to find demonstrate the effect of different types of neural nets on reinforcement learning performance; a Deep Query Network, a Deep Convolutional Query Network, and a Dueling Deep Query Network.




Reinforcement learning for blackjack

The work that our project and most other approaches presented here come from is Watkins' PhD thesis in 1989[2] which described the concepts behind Q-Learning. The basic idea behind Q-learning is to decide on an optimal policy to use at each level of game play by analyzing future rewards.

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize $s$
  Repeat (for each step of episode):
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $a$, observe $r$, $s'$
    $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
    $s \leftarrow s'$;
  until $s$ is terminal

Q learning formulation [3]

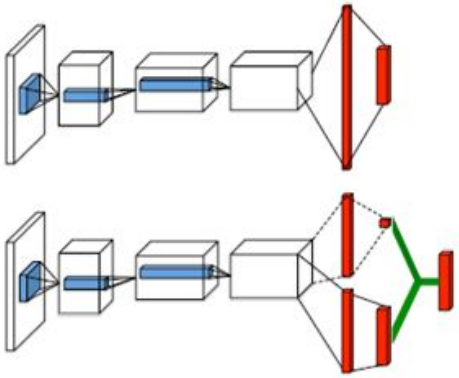
Q-Learning is an off-policy algorithm which means that it updates its Q value based on the Q value of the next state and the greedy action of that state. In contrast to an on-policy algorithm, such as SARSA which updates its Q value based on the Q value of the next state and the action of the current policy. Although Q-learning is generally slower than on-policy algorithms, we chose it because it is ideal for cases where more exploration is needed such as in a game of Blackjack. This is because it starts at a high level of randomness and slowly reduces randomness as it gathers more information from partially random moves.

A notable development built on Q-Learning was Deep Q Learning which utilized neural networks to fit Q-values as opposed to a lookup table. The most famous paper based on deep Q-learning is Google Deepmind's paper: Deep Q-Learning on Atari video games[3]. In their paper they presented a novel algorithm that would adapt existing reinforcement learning approaches to optimize a neural network through gradient descent. Several of the most noteworthy aspects of their algorithm was occasionally performing a random action as opposed to selecting the best action as well as subsampling of the set of actions performed. The goal of these two modifications was to address several problems with utilizing neural networks with Reinforcement Learning, namely that naively using a sequence of actions performed during the game would result in the data being heavily correlated/biased, as well as allowing the algorithm to explore more of the problem surface through random actions. Notably, many aspects of playing Atari games is also relevant to Blackjack, namely that the states are only partially observable due to the opponent's facedown cards as well as the fact that the next card in the deck is unknown.

Another notable paper is TD-Gammon which Deepmind's paper was based on[4]. In this case they did not utilize the randomness parameters above, rather they simply fed a given sequence of actions back into the network. In the Deepmind paper above they noted that the reason why the methodology applied to TD-Gammon was not replicable for many other games was primarily due to the randomness of the dice which allowed the network to explore more of the problem space.

Another variant of DQN, the dueling network architecture for deep reinforcement learning[1], introduced a new network architecture that can be applied to any existing and new RL algorithms to result in a significant improvement. The insight behind the dueling network is the decoupling of the representation of state values and it's associated action advantages, so the

network can learn the estimated state value function without the effect of its associated actions and learn the advantages of taking each action of the dependent state.



Network Architecture Comparison [1]

As the Figure above illustrates, the first portion of the network has a similar setup as the standard single stream DQN network architecture, then the network split into two streams, with one stream for estimating state value that ends with a dense layer of size one and the other stream for estimating the action advantage that ends with a dense layer of size of the number of possible actions of the game. Lastly, the two streams are aggregated together to produce a Q value for each possible action using the following equation.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

Dueling Aggregation Formula [1]

This function module combines the two streams of fully connected layers and offsets the action advantages by the mean advantage value to preserve the identifiability that allows us to recover the state value and action advantages uniquely when given the Q function. Moreover, this aggregation module should be implemented as part of the network to provide the same input and output interface as the standard DQN network. The paper was able to demonstrate that this dueling network can quickly learn which states are more valuable since during each update the state value function will be updated despite the action that was taken compared to the standard one stream DQN network only the value of one state-action pair is updated and the values of

other state-action pairs with the same state doesn't change. The results of the experiment demonstrated the dueling network architecture dramatically improved existing RL approaches in various Atari games.

**Approaches**

Q learning

Input : States, $s \in S$, Actions $a \in A(s)$, Initialize Q(s, a), $\overline{v}$, $\alpha$, $\gamma$, $\pi$ to an arbitrary policy (non-greedy);
Output: Optimal action value Q(s, a) for each state-action pair;
while *True* do
  for ($i = 0$; $i \leq$ *# of episodes*; $i++$) do
    Initialize $s$ and data table;
    Choose $a$ from $s$, using policy derived from Q;
    for *Repeat(for each step of episodes):* do
      Take action $a$; observe reward, $r$, and next state, $s'$;
      Record state, action, and associated reward in data table;
      $Q(s, a) \longleftarrow Q(s, a) + (1 - \overline{v})\alpha[r + \gamma max_a Q(s', a') - Q(s, a)]$;
      $s \longleftarrow s'$; $a \longleftarrow a'$;
      until $s$ is terminal
    end
    Generate $\overline{v}$ from results recorded in data table for current episode;
    Update new value of $\overline{v}$ ;
    Clear data table;
  end
end

Q-learner

Our implementation was based on the above algorithm [4]. This algorithm uses the following forms of the Bellman equations to help an agent decide on an optimal move by calculating the expected value of future moves and then choosing the policy with the maximum expected value.

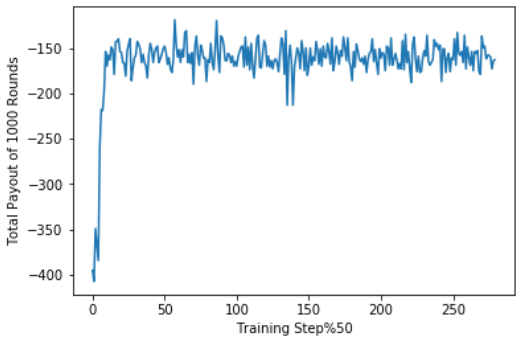
$$Q^{\pi}(s,a) = \mathbb{E}[r_{t+1} + \lambda r_{t+2} + \lambda^2 r_{t+3} + \ldots | s, a]$$

$$= \mathbb{E}_{s'}[r + \lambda Q^{\pi}(s', a') | s, a]$$

$$Q^{*}(s,a) = \mathbb{E}_{s'}\left[r + \lambda \max_{a'} Q^{*}(s', a') | s, a\right]$$

Optimizing Q

Where R refers to the return, E refers to the expected value, lambda is the discount factor, Q is the measure of the quality of the move, Q* is the optimal value of Q, s is the state, a is the action. As the Q-learner's policy becomes more refined, we see the total payout improve from -400 to about -166.




Q-learner payout vrs training step

DQN

**Algorithm 1** Deep Q-learning with Experience Replay

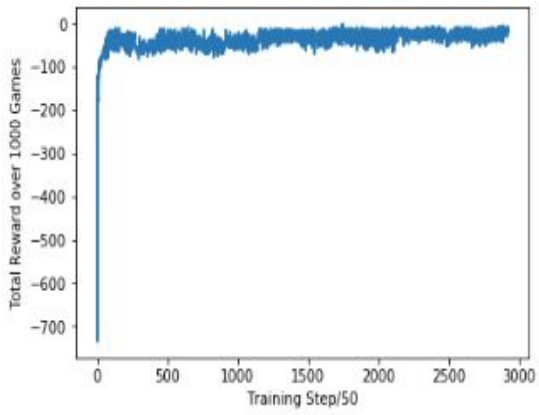
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  **for** $t = 1, T$ **do**
    With probability $\epsilon$ select a random action $a_t$
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  **end for**
**end for**

Deep Q Learning from Atari[5]

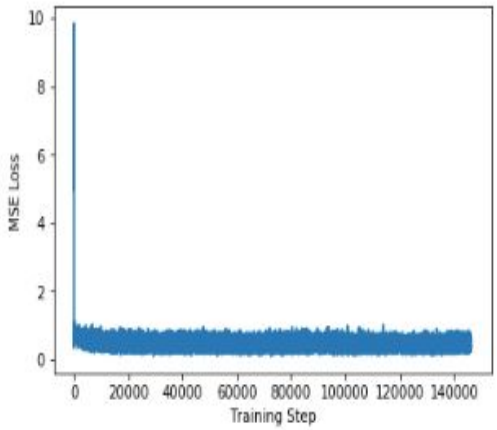
For vanilla DQNs we utilized the algorithm described in Playing Atari with Deep Reinforcement Learning by Minh. Some notable ideas this paper introduced was retaining past games to be reused in training as well as choosing random actions in order to further explore the problem space. One major difference between the Atari games and the Blackjack environment we utilized was that Atari games were played via images and our Blackjack environment captured the game state instead. As a result, we utilized fully-connected layers as opposed to the convolutional layers in the paper. Some other differences between our implementation and the paper's were the exclusion of linearly decaying epsilon and scaling down the number of training episodes and memory to 100000 episodes and a memory size of 10000. Linearly decaying epsilon seemingly just made training more unstable and the models seemed to converge well within 100000 episodes.

The neural network architecture utilized was two fully-connected layers of size 10 with ReLu activations followed by a linear layer of size 2 to predict values for the corresponding actions: to hit and not hit. The optimizer used for gradient descent was RMSprop with learning rate 0.001 and rho 0.9.

Below are the plots of our evaluation function, sum of rewards over 1000 games and the training loss.



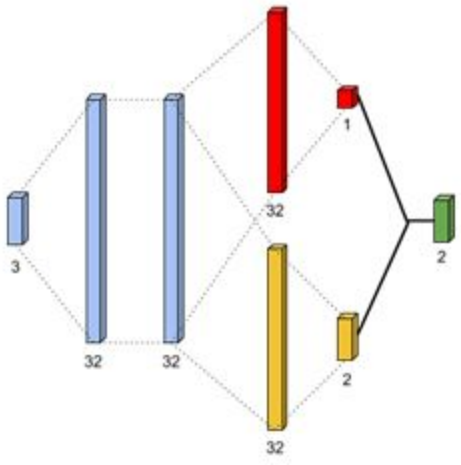

DQN payout vrs training step




DQN loss vrs training step

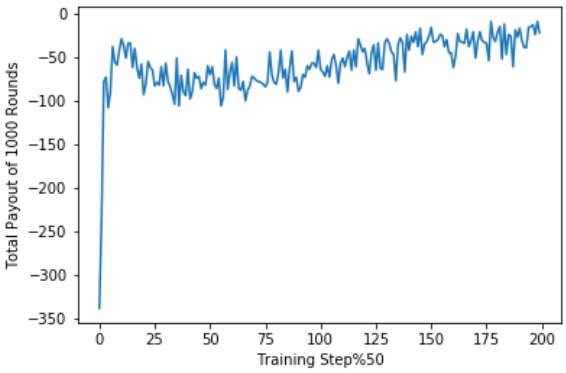
Dueling DQN

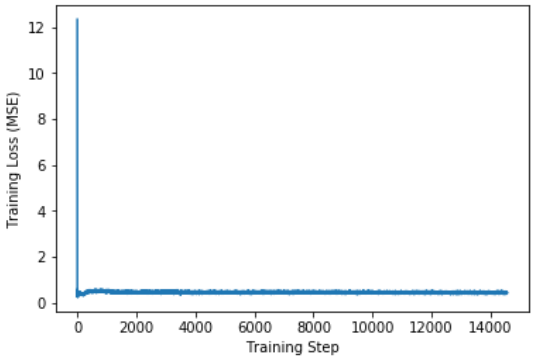
The dueling DQN was implemented following similar network architecture as the original paper that introduced it [1]. Particularly, the first stage of the network consists of two dense layers, size 32, as opposed to the 2D convolutional layers implemented in the original paper since for our use case the input is not an image. Next, the network split into two streams of two-level dense layers of size 32, one for estimating state values and estimating action advantages, and the output of the two streams has size one for state value and size two for action advantages since there are only two possible actions, hit and stand. Last, the two streams are aggregated together using the same equation module presented in the original paper to produce a set of two Q-values, one for each action. The figure below illustrates the dueling network architecture.




Dueling Architecture

We evaluated the performance of the dueling network DQN with different size dense layers, for any network with a wider layers than the network, described in the method section, the result doesn't demonstrate any noticeable improvement in the performance. Figure ## illustrate the network empirical training loss, as the loss rapidly reduced during initial few training epochs and then fluctuates at a slowly decreasing trend. The following figure illustrates the average payout of 1000 games at every 50 training steps, the result illustrates a generally increasing trend. Lastly, the evaluation of the dueling DQN implementation on the same seeded 1000 rounds of blackjack, the payout shows a significant improvement.



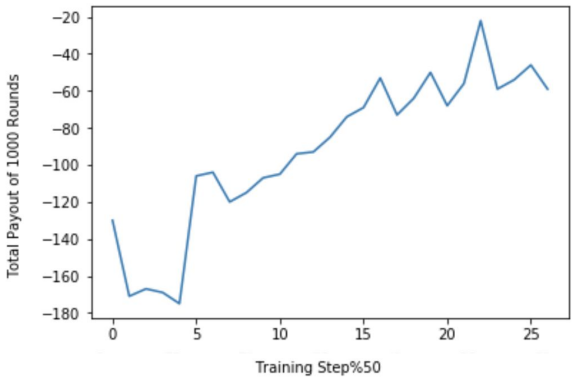



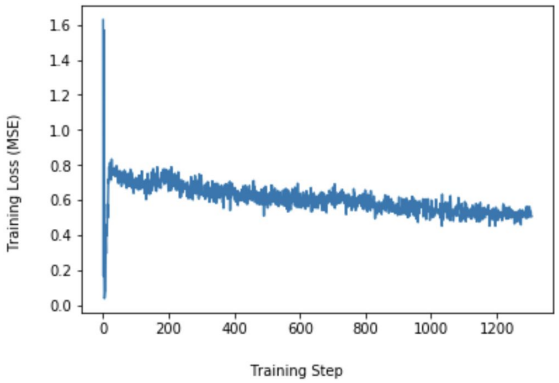

Dueling DQN payout vrs training step

Dueling DQN loss vrs training step

DCQN

The convolutional layer is the fundamental piece of convolutional neural networks. It convolutes a filter of a specified size on an input to create and output feature maps that highlight different aspects of the original data. The network will then learn the feature maps which show different features under a specified activation function. Deep Convolutional Query Networks follow the architecture of DQNs. However, convolutional layers replace the fully connected layers that compose the hidden layers of the DQN. This allows the network to exploit spatial correlation of the data. Each convolutional layer included 24 filters and a kernel size of 3. This results in 24 output feature maps after each layer. To make performance comparisons possible, all other parameters were kept the same as the DQN and Dueling DQN. As shown in the plots below, the DCQN's performance improves as the number of epochs, or number of times the training set is run through. This can be seen both by the increase in wins as in the figure below, but also in the decreasing loss of the network over 14,000 epochs.



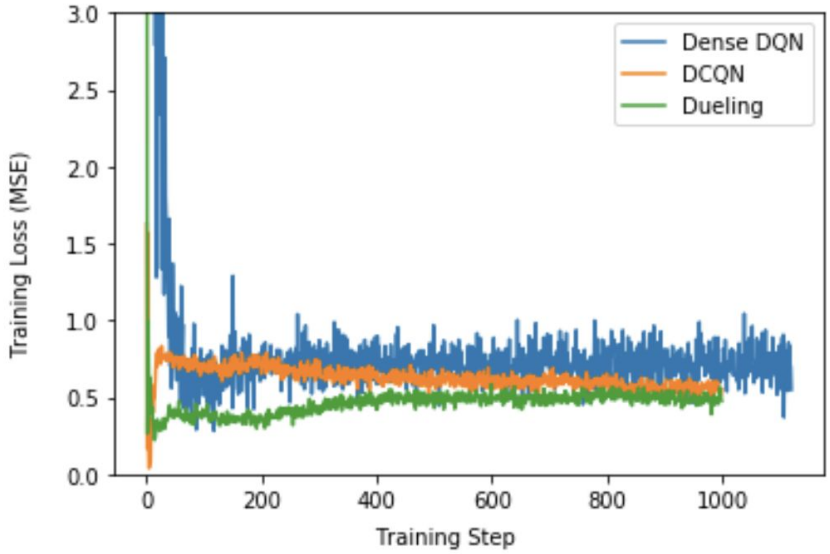
DCQN payout vrs training step DCQN loss vrs training step

**Simulations / Results**

| Average Payout for Types of RL Algo Over 1000 Rounds | |
|---|---|
| Architecture | Seeded Payout |
| DQN, Dense Networks | -14 |
| DQN, Convolutional Layers | -43 |
| Dueling DQN | -8 |
| Q-Learning | -166 |
| Random Player | -395 |

| Dataset 1 | Dataset 2 | $\hat{x}_1$ | $\hat{x}_2$ |
|---|---|---|---|
| Simple Strategy | Monte Carlo | -9.73 | -3.22 |
| Simple Strategy | Basic Strategy I | -9.73 | -2.28 |
| Simple Strategy | Basic Strategy II | -9.73 | -0.84 |
| Simple Strategy | Counting Cards | -9.73 | -0.07 |
| Monte Carlo | Basic Strategy I | -3.22 | -2.28 |
| Monte Carlo | Basic Strategy II | -3.22 | -0.84 |
| Monte Carlo | Counting Cards | -3.22 | -0.07 |
| Basic Strategy I | Basic Strategy II | -2.28 | -0.84 |
| Basic Strategy I | Counting Cards | -2.28 | -0.07 |
| Basic Strategy II | Counting Cards | -0.84 | -0.07 |

[6] Table displaying existing strategies for blackjack




We noticed that the dueling DQN performed best with an average payout of -8. Q-learning performed worse compared to both DQN methods. The loss plots for all the Dense DQN, DQN with convolutional layers and the dueling DQN converge as expected showing that the models were trained successfully and improved over time. The poor performance of Q learning might be due to its initial reliance on simply updating the Q value based on the greedy action of the next state, whereas the DQNs take a more balanced approach to learning.

**Conclusion**

We implemented four different approaches of reinforcement learning on playing Blackjack and evaluated each approach under the same metric for a direct comparison of performance. As the results demonstrated, all of our RL approaches outperformed an average human player that relies on random actions, and all deep reinforcement approaches outperformed the basic Q-Learning. Indeed, the dueling network architecture DQN yield the top performance among the approaches in the seeded payout of 1000 rounds of blackjack. Although the dueling network yields a dramatic improvement over random action player, the resulting average payout is still negative. Further, we believe if money betting is added to the evaluation implementation, with a reasonable betting strategy, we could yield a positive monetary payout using the dueling DQN as an action selection agent. This will be an interesting area to explore as the next step in the

study of using reinforcement learning to learn to play blackjack and attempting to yield a positive payout.

## References


[1] Wang, Z., Freitas, N.D., & Lanctot, M. (2016). Dueling Network Architectures for Deep Reinforcement Learning. *ICML*.

[2] Watkins, C.J.C.H. (1989). Learning from Delayed Rewards. PhD thesis, Cambridge University, Cambridge, England

[3] Sutton, Richard S, and Andrew G Barto. “Reinforcement Learning: An Introduction.” The MIT Press Cambridge, 2014.

[4] Lockery, Daniel & Peters, James. (2008). Adaptive learning by a target-tracking system. International Journal of Intelligent Computing and Cybernetics. 1. 10.1108/17563780810857121.

[5] Mnih, Volodymyr et al. “Playing Atari with Deep Reinforcement Learning.” *CoRR*abs/1312.5602 (2013): n. Pag.

[6] https://math.dartmouth.edu/theses/undergrad/2014/Vaidyanathan-Thesis.pdf