Can you teach a computer to play Blackjack better than a human player?

Compiled by {Mehtab Singh Gill, Amitoj Singh Gill, Gurinder Hans}

Problem:

According to current field research, there is not much work done on a creating models that use Reinforcement Learning to play Blackjack. Reinforcement Learning refers to goal-oriented algorithms. These algorithms learn how to take a complex objective and maximize it along a particular dimension by iterating through many steps. In this kind of learning, the starting state is from scratch and by slowly performing many steps over time, these algorithms attain very good performance. Throughout the learning process, these algorithms are rewarded and penalised according to the steps that they take and then learn from it's mistake. [4]

Our report is about using Reinforcement Learning to teach the computer to play Blackjack better than an average human player. The computer has no knowledge about how to play the game, and is going to learn itself by going through many iterations of the game. We looked at the current techniques used for training Blackjack computer agents and try to improve the results with our own implementations. After playing with the current implementation, we tried three new improvements: Double Q-learning, Experience Replay Reinforcement Learning and Recurrent Neural Network, with the goal of achieving better results than the current methods.

Data Generation:

We used 'OpenAI Gym' environment *'Blackjack-v0'* to produce data to train our 3 different models. At the beginning of every round the environment returns 'Player Sum', 'Dealer Face Up' card, and a boolean indicating if player can use Ace card as 11(*True*) or not (*False*). Once the agent analyzes the state of the game returned by the environment, it provides the environment with an action to take, 'Hit' or 'Stand'. This action is played in the game and the environment returns the 'next state' of the game, the 'Payout' and 'isDone' (i.e if the game is over not). This process is repeated until the agent has won/lost the game. The target value for each round is that the payout should be +1, which means the player won the round and -1 for loss and 0 for draw.

The Total Sample Space for game state:

1. Possible values for Player's sum: [2, 21] = 20
2. Possible values for Dealer's sum: [1, 10] = 10

3. Possible values for usable Ace card: [True, False] = 2

Size of total sample space is: 20 * 10 * 2 = 400.

What Has Been Done Before?

'Q - Learning' is the technique currently used to train an agent to earn more payout than average Blackjack player. [1] The procedure is to build a table for all possible state and action pairs, the values for the possible actions 'Hit' and 'Stand' are updated with respect to the reward an action receives. Once, the agent has explored the environment enough then we will have the resulting table which represents the evolved strategy which our agent has learned to play Blackjack. We call this table the Q-table and the action values Q-values, giving birth to the name 'Q - Learning'. The benchmark below compares this technique against the Random Action strategy, which is simply picking a random action. The two graphs below show the comparison between the two techniques:

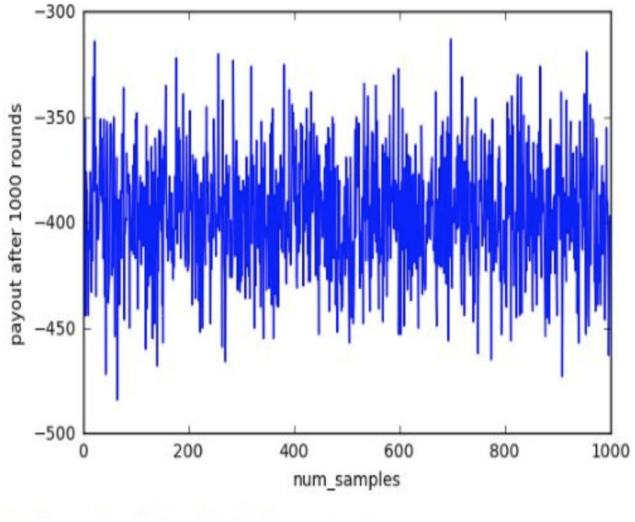


Figure-1: Random Strategy (Avg = -400.69)

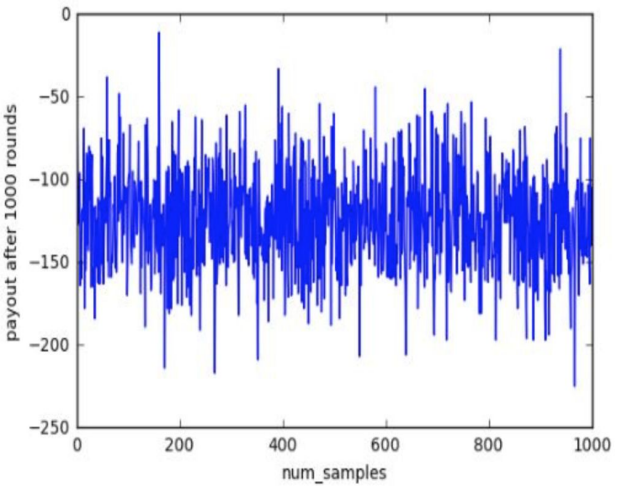


Figure-2: Q-learning Strategy (Avg = -120.49)

Improvement # 1 - Double Q learning:

In order to improve the Q-learning technique's performance for some stochastic environments we tried the Double-Q learning technique. The stochastic environments are caused because of the large overestimations of action's Q-values and these large values makes it biased towards an action. Consequently, it takes a lot of iterations to improve the Q-values. That is why we implemented Double- Q learning that helps the agent explore the environment faster. [2]

1.1 Procedure of learning :

Q - Learning takes way more iterations to converge the negative average payout than Double Q learning because it is single agent trying to learn every possible state of the game and then update the Q-values for the two possible actions 'Stand' and 'Hit' for a specific state. On the other hand, in Double Q learning we train two different agents playing two different games but sharing their Q-table. When both the

agents see a state for the first time they take a random action. If a random action was taken the Q-values are calculated by the following formula (Formula-1):

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Where 'Reward' is the average of the payouts seen by the agents for at state 't' and 'learning rate' determines how much we learn for that state and 'Discount factor' determines how we learn from the the 'Next observation' in order to take the action with maximum Q-value for the next observation. But if the state is seen by both the agents then we compare the Payouts(i.e the reward) for the state in the two Q-tables, Whichever table has the maximum reward the agent takes ation with highest Q-value from that table. Represented by the following equations:

1. $\hat{Q}_1(s_t, a_t) = r_t + \ . Q_2(s_{t+1}, argmax_{a^*} Q_1(s_{t+1}, a^*))$
2. $\hat{Q}_2(s_t, a_t) = r_t + \ . Q_1(s_{t+1}, argmax_{a^*} Q_2(s_{t+1}, a^*))$

Once, it has picked which action to take then the agent learns the Q-values for performing that action in the given state of the game by Formula-1.
The values stored in the Q-table could be as follows:

$State(P.S,\ D.S,\ T/F)$	$Q-value('Bust')$	$Q-value('Less\ than\ 21')$
$(15,\ 5,\ False)$	-95	-90

Improvement # 2 - Experience Replay:

"Experience Replay" Reinforcement Learning is a method where the system stores different states in a table along with Q-values for actions ('Hit' / 'Stand'') and then for a given state, picks the action with maximum payout. This way we are using what we learned earlier to make future decisions. [5]. In comparison with the previous improvement, we have shifted the focus of the agent to store probabilities of getting 'Bust' for the possible states of the game as the Q-values instead of storing Q-values of the actions. We shifted our focus because we wanted the agent to try and learn the possibilities of getting bust in order to learn whether to 'Hit' or 'Stand' if the player sum is already pretty close to 21.

2.1 Procedure of learning:

For implementing Experience Replay our objective was to update Q-values for the given state by using a Feed Forward Recurrent Neural Network (RNN). Given a state in a round which comprises of the values of '$Player\ Sum$', the '$Dealer\ Sum$'(i.e the dealer face up card) and a boolean value for usable ace card are fed to the RNN (see Figure-6). Given these values the first Hidden Layer-1 calculates the probabilities $Pr('Stand')$ and $Pr('Hit')$ *(equations 3 & 4)* (below). These probabilities are passed onto the Hidden Layer-2, where the probabilities of $Pr('Less\ than\ 21')$ and $Pr('Bust')$ are calculated *(equations 5 & 6)*. These probabilities are passed on to the output layer where they are compared to take decision on which action to take. If the

value of $Pr('Less\ than\ 21')$ is greater than $Pr('Bust')$ then we take the action ***'Stand'*** else we ***'Hit'***. This action is the output $h(t)$, and it is passed back into the RNN as a bias for the next state $h(t+1)$ and the input is the next observation $X(t+1)$ (see Figure 7). We keep feeding these values back into the RNN and updating the Q-values of $Pr('Bust')$ and $Pr('Less\ than\ 21')$ for the intermediate states of each round until the round is completed.If the next observation is a seen state in the Replay Table and if the payout is greater than zero then the Q-values of the table are used to determine the values of ***'Hit'*** and ***'Stand'*** by using *equations 7 & 8* respectively.

Weights used in the three layers were all of value 1 because we wanted to directly compare the probabilities at every node. This helped us figure out that the following probabilities are directly proportional:

1. $Pr('Less\ than\ 21') \propto Pr('Hit')$
2. $Pr('Bust') \propto Pr('Stand')$

The values stored in the Experience Replay table could be as follows:

$State(P.S,\ D.S,\ T/F)$	$Pr('Bust')$	$Pr('Less\ than\ 21')$	$Average\ Payout$
$(15,\ 5,\ True)$	0.766	0.576	$-\ 8.0$

Equations used:

1. $P.S$ - $Player\ Sum$
2. $D.S$ - $Dealer\ Sum$
3. $Hit_1 = (P.S\ +\ New\ Card)/D.S$
4. $Stand_1 = P.S/D.S$
5. $Less\ than\ 21 = stand_1/hit_1$
6. $Bust = hit_1/stand_1$
7. $Hit_2 = (([Observation]['Less\ Than\ 21']) + hit_1)$
8. $Stand_2 = (([Observation]['Bust']) + stand_1)$

For Equations 7 and 8 Sigmoid function was used to make probabilities comparable by squashing the values between 0 and 1.

Result of Improvement #1 and Improvement #2:

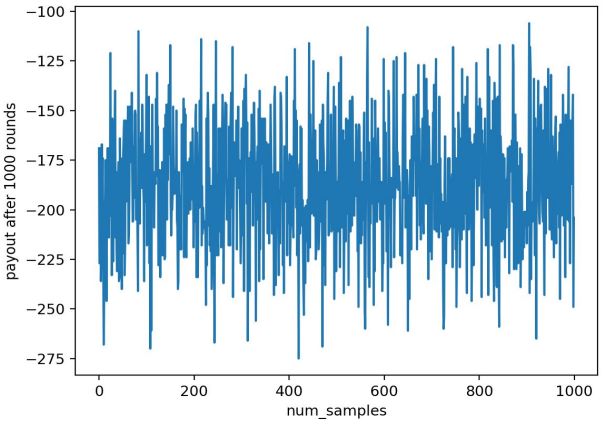


Figure-3: Experience Replay Reinforcement Learning graph

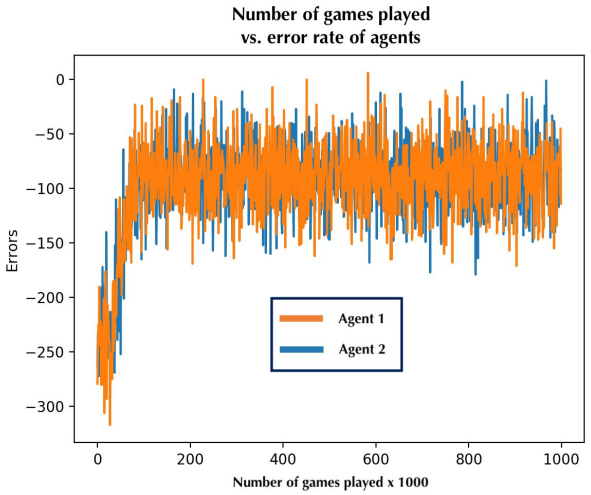


Figure-4: Double Q convergence graph

The Average payout of Double-Q learning (i.e Improvement-1) is -95.47 and for Experience Replay (i.e Improvement-2) is -185.45. Therefore, Improvement-1 performs better than Improvement-2 and the Q-learning.

Improvement # 3 - Recurrent Neural Network

Recurrent Neural Network is a neural network that has loop in it. In this network, the previous information persists and you don't have to start everything over from scratch [3]. Figure 6 shows the neural network that we devised for this model and then we used this and created a Recurrent Neural Network model as shown in Figure 7.

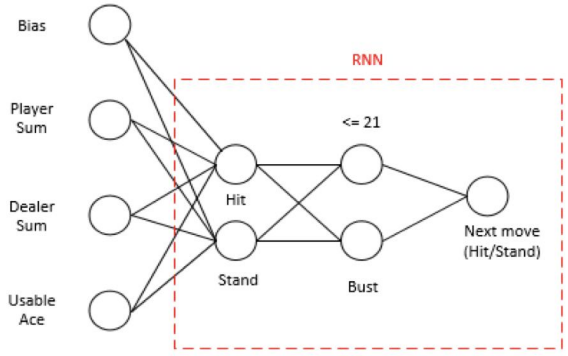


Figure-6: Neural Network

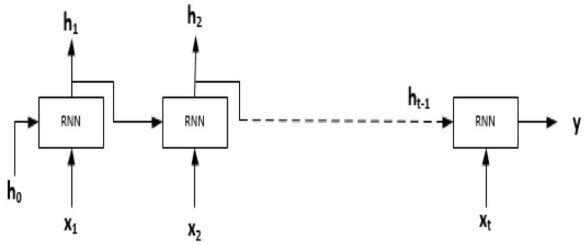


Figure-7: Recurrent Neural Network (RNN)

3.1 Procedure of learning:

The recurrent neural network starts by taking the X_t and previous state (y) and virtually takes both *"Hit"* and *"Stand"* action by computing the *equations 1 & 2* (below). These outcomes of these actions are passed into the next layer, where the chance of getting "Busted" or "<= 21" is calculated *(equations 3 & 4)*. The "Less than 21" and "Bust" values are the **Hit:Stand** and **Stand:Hit** ratios respectively. This gives us a high and low value on both of the nodes. After calculating these values, we pass it to our final decision making layer, which runs the output through sigmoid and

chooses *"Hit"* action if the sigmoid returned a value > 0.5, *"Stand"* otherwise. This decision (y) is the final output action of the RNN agent and is therefore taken in the real game environment, yielding a new game state (X_{t+1}). The previous action taken by the RNN Agent plus the new game state is then passed back into the RNN and this process repeats until the game is over. At the end of the game, we check if the RNN agent has won or lost the game. This result is used to calculate the total payout made by the RNN Agent. If the RNN Agent wins the game, it is given a payout of **+0.5**, if it loses a payout of **-1.0** is applied. Initially we would calculate the error and do backprop after every game, however this did not yield good improvement in the recurrent neural network. We then shifted to accumulating the total error for every 10 games and then use the total error over 10 games to do back-propagation *(equation 7)* resulting in a much better improvement in the overall performance of the RNN Agent. This process was then repeated for a total of 120 thousand games.

Equations used in RNN Agent training:

1. $H = ((w_{11}{}^{(1)} PS + New\ Card)/w_{12}{}^{(1)} DS)$
2. $S = ((w_{21}{}^{(1)} PS)/w_{22}{}^{(1)} DS)$
3. $Less\ than\ 21 = (w_{11}{}^{(2)} H/w_{12}{}^{(2)} S)$
4. $Bust = (w_{21}{}^{(2)} S/w_{22}{}^{(2)} H)$
5. $y = ((Less\ than\ 21) w_{11}{}^{(3)}/(Bust) w_{12}{}^{(3)})$
6. $E_w/w_{11}{}^{(1)} = (a_1{}^{(1)})(1 - (a_1{}^{(1)}))[w_{11}{}^{(2)}{}_1{}^{(3)} + w_{21}{}^{(3)}{}_2{}^{(3)}]$
7. $X_t = [PS,\ DS,\ Usable\ Ace]$

Training the RNN Agent:

The error rate of the RNN Agent was measured every 100 games and it yielded the following chart:

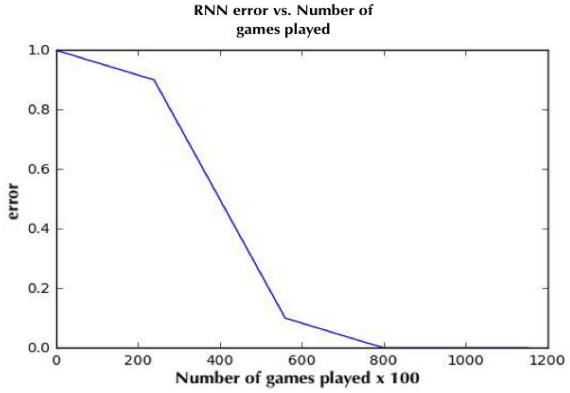


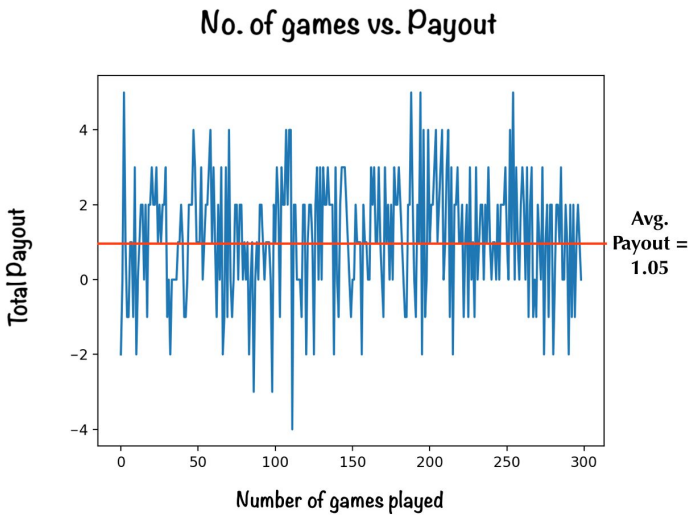
Figure-8: Training error of RNN

The error rate drops rapidly between 20 thousandth 60 thousandth game and eventually converges at around 80 thousand. The whole training process only takes a few minutes due to the games completing in a few moves, resulting in win/loss.

Testing the RNN Agent:

Finally, as seen in Figure 3 below, after training the RNN Agent, we let it play about 300 games, and the average payout was 1.05, which means we actually made money! Interestingly we notice that even with the RNN, the payout hovers between

-2 and +2, and we are constantly losing and winning the games. But this is a much



better improvement over the methods we tried before.

Figure-9: Testing error of RNN

Conclusion and Future goals:

Observing and comparing all different methods, Random Agent, Q-Learning, Double-Q Learning, Experience Replay, and RNN network, the RNN network outperforms all other methods by a huge difference. This goes to show that we can use RNN networks and our neural model and apply it to other games simply by modifying the localized assumptions made about the said game. In this neural model, we were calculating the Bust to Less than 21 ratio and Hit to Stand action ratio. Applying this recurrent model to other games would simply require changing these steps to the localized rules for the other said games.

References:

[1] Bijja, P. and City, N. (2018). *Teaching a computer blackjack using Reinforcement Learning*. [online] Curious Coder. Available at: https://curiouscoder.space/blog/machine%20learning/teaching-a-computer-blackjack-using-reinforcement-learning/ [Accessed 15 Dec. 2018].

[2] Hasselt, H. (2018). *Double Q-learning*. [online] Papers.nips.cc. Available at: https://papers.nips.cc/paper/3964-double-q-learning [Accessed 15 Dec. 2018].

[3] Medium. (2018). *An Introduction to Recurrent Neural Networks – Explore Artificial Intelligence – Medium*. [online] Available at: https://medium.com/explore-artificial-intelligence/an-introduction-to-recurrent-neural-networks-72c97bf0912 [Accessed 15 Dec. 2018].

[4] Skymind. (2018). *A Beginner's Guide to Deep Reinforcement Learning*. [online] Available at: https://skymind.ai/wiki/deep-reinforcement-learning [Accessed 15 Dec. 2018].

[5] Web.stanford.edu. (2018). [online] Available at: http://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf [Accessed 15 Dec. 2018].