Playing Blackjack with Memory Information Using Reinforcement Learning

Kexin Yang (ykx1998@g.ucla.edu)
Department of Psychology, 502 Portola Plaza
Los Angeles, CA 90095 USA

William Weng (williamweng917@gmail.com)
Department of Psychology, 502 Portola Plaza
Los Angeles, CA 90095 USA

Abstract

Deep Q learning is a powerful and commonly used reinforcement learning algorithm which has been successfully applied in many strategic and probabilistic games such as blackjack. Previous research has established that, by simply observing the game state, a DQN agent can achieve an average payout close to using the optimal strategy in a blackjack game. Inspired by the common strategy human players used when playing blackjack, we proposed an additional Belief Network which processes the memory information a player has over the current game to represent the player's belief over the remaining cards distribution. By adding this Belief Network in addition to DQN, we find a positive, sigmoid-like relationship between players' memory size and their average payout from the game. Ultimately, we believe that the design of a more explicit model structure could not only improve the overall model performance, but also in some degree, reduce the uncertainty and inexplicability of Neural Network due to its complicated structure.

Keywords: reinforcement learning; Deep Q learning; blackjack

Introduction

In recent year, many efforts and researches have been done to train artificial intelligence agents on different kinds of human tasks and games, and compare their performance with other AI agents or even with human players. Huge improvements were made through these years, and AI agents achieved tremendous success at playing many games including chess, video games and casino games (Mnih et al., 2013; Silver et al., 2017; Wu, 2018).

Deep Q learning is a reinforcement learning algorithm that has been proven to be effective in learning many different types of game. First proposed by DeepMind, it is an algorithm that integrates the traditional Q-learning methods with Neural Network. In this paper, we focused on reinforcement learning agent trained in the context of blackjack. Blackjack is a typical Partially Observable Markov Decision Process where one of the dealer's card is hidden from the playing agent. Due to this partially observable characteristic, blackjack doesn't any simple good strategy to follow given only the information about the current game state. The optimal strategy using only current game information gives an expected return of -0.0957. However, by memorizing previously drawn cards and calculating the a posteriori probability distribution, some expert human players are able to achieve a positive average payout and gain a huge profit from this game. Therefore, we think it is interesting to investigate how different input and process of information can affect the performance of a DQN agent.

Past researches show good performance of DQN agent when playing blackjack with only current game information. In the paper by Wu (2018), by implementing only a DQN to train the agent, the agent is able to learn a good policy for playing blackjack with an expected return of -0.1074, close to the expected return of optimal policy.

Looking through these past researches, however, we noticed they only focused on achieving a good expected return by setting the optimal policy as their baseline. The limitation of this focus on "optimality" is that they were forced to only consider the current state information since the optimal policy were created as a standard for any single hand. This limitation undermines the potential performance of their agents because as a probabilistic game, past information is clearly an important part of this game.

Therefore, with the inspiration from human player's strategy to memorizing past cards and apply a probabilistic policy to achieve a better payout, we argue that it is reasonable to add memory information as an additional input for our agent. We want to investigate the relationship between the amount of memory information utilized by an agent and their resulting average payouts, an aspect that remains unexplored from the previous researches. We hypothesize that the results will show a sigmoid relation between players' memory size and their average payout. This is rationalize by the fact that while remembering more cards gives a more accurate calculation of the conditional probability of getting certain cards, this increase in calculation accuracy may not be as important as the memory size grows.

To make sure that the agent applies memory information into their action choice process, we implement an addition Neural Network structure called the Belief Network. This Belief Network is designed to give the agent an explicit process of the memory information they have, and combine it with observation of the current game to obtain a belief estimation of the probabilistic distribution of the remaining cards. These belief estimation, along with other information about the current game state, will be sent into the regular Deep Q Network.

Our results show that player's memory size and their average payout does have a sigmoid-like relationship, which supports our hypothesis. Also, the model converges with less training samples when using the additional Belief Network in comparison to only use a DQN. This suggests that a more explicit structure such as the Belief Network could be helpful for the agent in learning the game, and we believe with some more tuning and improvements to the model, our agent can achieve an even better payout from the game, and learn a good policy to play this game.

Implementation

Our implementation consists of two major parts — a game engine that stimulates the blackjack game environment and a player which is trained with our Neural Network model to make action choice based on the game environment.

Figure 1 (a)

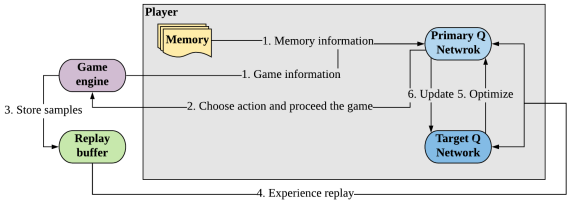

Figure 1 (b)

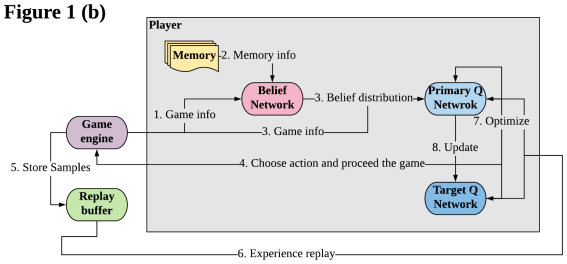

Figure 1 . Implimentation structure for agent with (a) only a Deep Q Network (b) Belief Network and Deep Q Network

Game engine

The game engines is set to use 4 decks of cards at a time. These cards are shuffled randomly before distributed, and whenever 170 cards are drawn from the deck, the game engine will discard the remaining cards and continue the game with new, randomly shuffled cards. During each turn, the player can choose from 2 valid actions – hit or stand to indicate whether they want to draw a new card.

To start a new round of game, the player will first draw two cards from the deck, both facing up. Then the dealer will also draw two cards with the first facing down and the second facing up. After the player chooses an action, the game engine will proceed the game. If the player choose to hit, a new card will be drawn from the deck, and if the sum of player's cards add up to 21 or higher, the game engine will automatically terminate this round of game, otherwise, the game will proceed to the next turn. If the player choose to stand, the current round of game will also terminate. When the game engine detect a terminal state, it will reveal the value of dealer's hidden cards, and draw cards from the deck until the dealer reaches a hand sums above 17. The rewards for this game will be determined by comparing the value of player and dealer's hand, where -1, 0 or 1 is assigned when player lose, get a tie or win respectively. If either the player or dealer has a blackjack, the rewards will be doubled. After a round is finished, the game engine will proceed to start another round.

Player

The player is trained with a Neural Network model to make decision on blackjack action based on both the information about the current game state provided by the game engine, and their own memory information from previous rounds of game. Each player is assigned a memory size, which indicates the maximum number of past cards they can remember. The player will only remember the most recent cards played in the past rounds that their memory size allowed.

Two Neural Network model structures are used separately to train the player. Model 1 only uses a Deep Neural Network (DQN), and model 2 incorporates both the Deep Neural Network and a Belief Network (DQN + BN) to train an agent.

Deep Q Network DQN is a reinforcement learning network designed by Mnih et al. (2013) where they integrated the traditional Q-learning method with a neural network to train an artificial intelligence agent to play Atari.

The DQN is used to train the player to develop a policy on choosing their action based on game inputs. In a single-agent, partially observable Markov Decision Process like blackjack, at each time step t, the player observes some partial information about the current game state, and based on information they have over the game, player then chooses an action in the action space using their learned policy. The objective for the

player is to choose an action to maximize its expected return.

The implementation of DQN consists of three major parts – a primary Q network, a target Q network and a replay buffer. The primary and target Q network are initialized with identical values, but the primary Q network is used by the player to play the game, whereas the target Q network is only used to calculate loss and update network weights. So for each rounds of game, the player will act based on output from primary Q network, and after a round is finished, all the information about this round, such as player's action and rewards at each turn, will be stored in the replay buffer as a sample.

The DQN uses a batch learning methods. To update the network's weights, a batch of sample will be selected randomly from the replace buffer, and the Q networks are optimized by minimizing the following loss function calculated using the batch of samples.

$$L(\theta) = \sum_{i=1}^{m} (Q_{target} - Q_{predicted}(s^t)^2$$

This loss function is similar to the squared error used in supervised learning to measure how different is our predicted value from the desired value. Here, however, because there isn't a correct label for each sample, we instead use a target Q value to represent the maximum expected value over all possible action, and this target Q value is obtained from the target Q network.

$$Q_{target} = r^t + \gamma \cdot \max_{a^{t+1}} Q_{target}(s^{t+1})$$

In both model 1 and model 2, our Q networks are implemented with a single input layer, two hidden layers of 64 and 32 nodes and a rectify linear unit activation, and a single output layer of 2 nodes with a soft-max activation to represent the player's probability of choosing each of the two valid actions.

For model 1 (DQN), the input layer has 36 nodes. The first 30 nodes represent observation of player's current hands, dealer's current observable hands and player's memory of past cards, each using one-hot encoding of dimension 10 to indicate the card value. The last 6 nodes encodes further information of the current game state, including sum of player hands, 21 - sum of player hands, sum of dealer hands, 21 – sum of dealer hands, whether this is the first turn in a round and whether there is an ace in player's hands that can be used as 11.

For model 2 (DQN + BN), the input layer consists of 18 nodes where the first 10 nodes are the output from belief network. The last 8 nodes also encodes information of the current game, where the first 6 nodes is the same as the last 6 node for model 1 and 2 more nodes about the memory size of this player and the total number of cards being played in the previous rounds.

Belief Network The intuition for adding a Belief Network is to give the player a more explicit process of their memory information, and train the player to develop a belief over the probability distribution of the remaining cards. We hope the player can then use this belief to estimate how likely their desired card will appear in the next draw as a human player.

In our game setting, each player is limited by their memory size, so they don't have access to the actual distribution of remaining cards in the deck. Therefore, to train the network without a correct label, we can only try to push the player's belief distribution output towards a correct direction.

Without any further knowledge, we best standard is that, overall, the cards that are drawn at the next turn should have a somewhat high probability distribution. So here we use only the cards from next round to get a target distribution, and calculated the loss function as the squared error between the target distribution and the predicted distribution.

$$L(\theta_b) = (D^t_{beliefnetwork} - D^{t+1}_{newcardsdraw})^2$$

The implementation for Belief Network also consists of a input layer, two hidden layers of 64 and 32 nodes and a random normal initializer, and a output layer of 10 nodes with a soft-max activation to represent the player's belief distribution of the remaining cards. The input layer has 21 nodes. The first 10 nodes represented player's observation of the entire current workplace. The second 10 nodes represents player's memory information along with their observation of the game, where each nodes encodes the remaining number of card of that value. This is calculated by subtracting the number of cards of that value in player's memory and observation from how many cards of that value should be in four decks of cards. The last nodes represents player's uncertainty over their memory, and is calculated by subtracting how many card they remember from the total number of cards played in past rounds.

Methods

Exploration

An ε-greedy exploration is applied by the player to choose action based on the probability output from the Deep Q Network. This methods allows the player to have a better exploration of all possible game states at the beginning of the training by choosing more random actions, and gradually increase the amount if exploitation as the training proceeds.

Specifically, this method is implemented with an annealing ε. During training, each time when the player needs to choose an action, a random number will be generated. If this number is smaller than ε, then player will choose the action randomly, and if the number is greater than ε, the player will choose the action with the higher q value outputted from the Deep Q Network. During testing, only the action with higher q value will be chosen.

$$\epsilon = \epsilon_{start} + \frac{\epsilon_{start} - \epsilon_{end}}{k_{decay}}$$

$$k_{decay} = e^{\frac{-k_{iter}}{\epsilon_{decay}}}$$

$$k_{iter} = \text{current iteration mod switch iteration}$$

Using the formula above, the value for ε is calculated at the beginning of each training iteration, and it is set to 1 during the first 500 training iteration to further facilitate exploration.

Experience Replay

To allow batching learning for the Deep Q Network, we applied the experience replay method. This allows the network to be updated by replaying a batch of past experience at a time sampled from the replay buffer with uniform probability distribution.

There are three major advantages for using this methods. First, it diminishes the probability that the network will overfit to recent experience, and gives a better account for uncommon events by feeding the network with a combination of past and new experiences in the chosen batch of samples.

Also, it provides the network with samples that can better represents the entire data distribution. This is because the replay buffer can keep a large number of experiences, which allows them to have a close representation of the true distribution, and the samples that the network use to learn are drawn from these experiences with uniform probability.

Lastly, it increases the sample efficiency by reusing experiences in the replay buffer.

Hyperparameters

The two models are trained for 2e5 and 1e5 iterations respectively. The learning rate for Deep Q Network and Belief Network are set to 1e-5 and 1e-3 respectively, with a decay to half the original value half way through the entire training. The discount factor γ to calculate the target Q value is set to 0.9. The replay buffer is set to keep a maximum of 10000 experiences, where the older experience will be replaced by a new one if this maximum threshold is met. The batch size of each training sample is set as 20 and 1 for the two models (DQN and DQN + BN). For the annealing ε, we set the initial ε as 0.95, final ε as 0.01, switch iteration as 5e4 and epsilon decay rate as switch iteration / 3.

Results

Figure 2 shows the results we got by training the agent using only DQN. The first graph shows the model performance across different training iterations. While this results suggests that the model converges after around 50000 iterations of training, there is still a relatively large pattern of oscillation in player's average payout as the training proceeds. This can be partially attributed to our optimization is non-convex, so the network is likely to fail into a local minimum. So as we continues the training after reaching this local minimum, the resulting performance may fluctuate within a certain range.

Figure 2 (a)

Figure 2 (b)

Figure 2. Resulting model performance for model with structure 1 (a) Model performance accross each training iteration (b) Model performance across each memory size

The second graph in figure 2 shows the performance of agents with difference memory size, where the red line indicates the maximum average payout an agent achieve during the above training process, and the blue line is the average of the all the average payouts after 100000 training iteration when the model converges. We calculate this average after convergence to account

for the fluctuation of model performance due to its non-convexity. This graph does not show any clear relationship between player's memory size and their average payout when the player is trained using only the DQN.

Figure 3 shows the results from agents trained with both DQN and BN. The first graphs shows a lot more stable performance after the model converges. This indicates that the network achieves a better learning with this model. It is also noticeable that while both models converges after around 50000 training iterations, the first model updates the DQN with a batch size of 20 experiences during each training, whereas the second model only uses 1 experience as a time. So overall, model 2 is able to learn faster with a lot less experiences comparing to model 1.

Figure 3 (a)

Figure 3 (b)

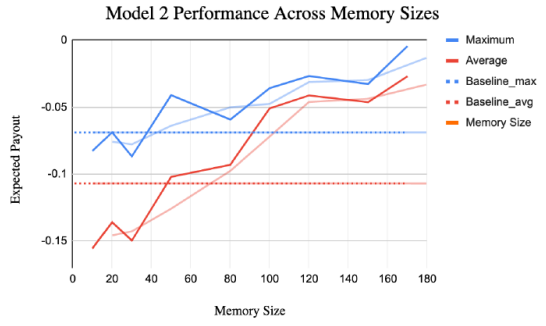

Figure 3. Resulting model performance for model with structure 2 (a) Model performance accross each training iteration (b) Model performance across each memory size

The second graph in figure 3 shows a clear positive relationship between a player's memory size and their expected payout for both the maximum value and the average value after convergence. When looking at the average after convergence value, the relationship shows a sigmoid-like shape, which support our initial hypothesis that the increase in average payout is larger when memory size is relatively small, whereas when memory size is large, the effect on average payout becomes smaller.

It is worth mentioning, however, that the average payout for agent with memory size smaller than 40 is even lower than agent that does not use any memory information. We belief this is cause by the fact that an agent forgot more cards played in previous rounds than they can remember when their memory size is very small. Because our model 2 fails to account for this fact, the agent can get a very biased output belief distribution from the Belief Network, which further affects their output from the Q network.

Conclusion and Future Direction

The results from our second model support our initial hypothesis that there exists a positive and more specifically, sigmoid-like relationship between how much memory information an agent have and their average payout in a blackjack game. This suggests that while more information is helpful for getting better performance, the marginal amount of improvement decreases as the amount of information increases.

Another take away from our experiment is that we find the importance of a more explicit structure for a good neural network structure. Without adding the Belief Network, our first model failed to display any clear relationship between an agent's memory size and their average payout. As we looked into our training data, we realized that the agents were probability not even using these additional memory information. The reason for that is likely because the DQN fails to associates the memory inputs with the rewards from their action by themselves.

Therefore, the necessity of adding an extra neural network to process these memory information arises with this inner characteristic of neural network. Because of its complicated structure and nonlinearity between each layer, a neural network is like a black box. It is almost impossible to know for sure what kind of connections it makes between the inputs and output. So it can be extremely helpful to take the advantage of some external knowledge such as how human process information and make decisions, and give the neural network a clear instruction to process their inputs in the same way through implementation of a more explicit network structure.

Our model is not perfect, and there are a lot of ways to make further improvements. Firstly, we can validate the accuracy of our belief network by comparing its output with the actual card distribution, and draw the relation between this difference and memory size to make sure the performance of belief network matches our expectation. Secondly, we can optimize the error correction method for our Belief Network. Lastly, to account for the fact that our current model performs

poorly with a very small memory size, we can add another explicit network to teach the agent learn about the direct connection between the memory information, memory uncertainty and their action rewards. This is similar to adding the Belief Network in the sense that we are trying to apply our external knowledge memory uncertainty from a small memory size hurts the validity of belief distribution. So we can give our model an explicit instruction to learn a policy to choose action with this knowledge by adding the extra network.

References

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). *Playing atari with deep reinforcement learning*. arXiv preprint arXiv:1312.5602.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Lillicrap, T. (2017). *Mastering chess and shogi by self-play with a general reinforcement learning algorithm.* arXiv preprint arXiv:1712.01815.

Wu, A. (2018). *Playing Blackjack with Deep Q-Learning.*