Deriving the Optimal Blackjack Policy Using Deep Q-Learning

Allen Wu
CS 230 Final Project, Stanford University

Introduction

- Blackjack is one of the oldest casino games, and remains one of the most popular. In blackjack, the player's goal is to have the sum of the cards in his hand be greater than the sum of the cards in the dealer's hand, without that sum exceeding 21. He starts with two cards and can hit to receive more. If the player wins, he doubles his bet.
- Because blackjack is an almost purely Markov process, has a clear reward scheme, and has a closed-form optimal policy, it represents a promising environment for exploring and testing reinforcement learning techniques.
- Deep Q-learning is an extension of Q-learning that uses a neural network instead of a Q-matrix and utilizes a replay memory buffer, generalizing Q-learning to more complex state and action spaces.
- We attempt to derive the optimal policy for blackjack using deep Q-learning.

Blackjack Model

- We use a state space of 5-dimensional tuples, representing the sum of the player's hand, whether it's the first action of an episode, whether the hand contains an ace, whether the hand is paired, and the card the dealer has showing.
- We need to consider whether it's the first action of an episode because some actions, such as surrendering and doubling, are only allowed as the first action in a hand.
- Similarly, splitting is only allowed when the player's hand is paired.
- Aces may be counted as either 1 or 11, and allow the player greater flexibility.
- We consider all five actions available in casino blackjack: hitting, standing, surrendering, doubling, and splitting.
- When a player hits, he gets another card. When he stands, he requests that the dealer complete his hand and score the game. To complete his hand, the dealer hits according to a predetermined rule. In most casinos, the dealer hits when is hand sums to 16 or lower, and on 17 when his hand contains an ace. We follow this convention.
- Doubling means doubling the bet, hitting, and standing.
- We model splitting as doubling the bet and starting a new hand featuring one of the original cards, which has the same expectation as starting two hands from the same card.
- We assume an infinite deck, which both approximates the standard practice of using a "shoe" of many decks and greatly simplifies simulation.

Deep Q-Network Implementation

- **Experience Replay Buffer**
  - Following the seminal paper *Playing Atari with Deep Reinforcement Learning*, we implement a deep Q-network that learns from an experience replay buffer.
  - Whereas traditional Q-learning updates after online experiences, Mnih et al.'s deep Q-network stores experiences in a buffer of (state, action, next state, reward) tuples and learns by sampling from this buffer after set time intervals. This allows sparse rewards to fully propagate through the network and provides a more stable target for training.
  - We use a buffer of the last 20,000 experiences and sample batches of size 256. These numbers are larger than the parameters Mnih et al. used because we have fewer states, more of which are terminal.
- **Deep Q-Network**
  - We represent our Q-matrix with a fully connected neural network with ReLU activations and batch normalization.
  - The network takes a state and outputs 5 scores that approximate the expected reward of taking each action from our given state.
  - To prevent our agent from considering illegal actions, we mask the scores of illegal actions with arbitrarily negative rewards.
- **Hyperparameters**
  - We primarily parameterize our Q-network by a neuron scaling factor k and the number of layers.
  - The scaling factor k determines how many neurons are in the first hidden layer, after which we use a geometrically decreasing number of layers until we output the 5 scores for each action.
  - We find that increasing both k and the number of layers generally improves performance. We haven't fully explored this pattern of improvement yet, having tested only up to 7 layers and $k = 7$.
  - We tuned our other hyperparameters by observing how long our network took to train and by checking how well our network learned that taking the surrender action deterministically loses half a bet.
  - Ultimately, we decided to take a descent step every 4 episodes, used a learning rate of 0.0001, a discount rate of 0.999, and stochastic gradient descent with momentum 0.9 and weight decay 0.0001.
- **Exploration**
  - Perhaps our biggest deviation from convention is our exploration method. We use ε-greedy exploration with an annealing ε according to the formula 0.05 + 0.95*e^[(# episodes - decay_period)/decay_scale].
  - We found that our agent had difficulty learning the rewards of the more obscure actions, so we wanted it to explore more aggressively.
  - Traditionally, people anneal ε according to a factor of e^(-# episodes), which has the nice property of tending to 0 as the number of episodes tends toward infinity. However, this factor is also convex, causing the agent to begin exploiting relatively quickly.
  - To this end, we also restrict random selection of the best-scoring action.

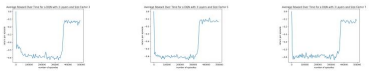
Results

Evaluation

- We evaluated our networks by two criteria: the average reward over 50,000 episodes following training and a score measuring the similarity of the derived optimal policy to the closed-form optimal policy. This latter score is out of 340, with 1 point for state.
- The following table summarizes the scores for deep Q-networks with several numbers of layers n and scaling factors k. For brevity, we use the notation DQN(n, k). We also implemented a traditional Q-network for comparison, which we trained for 50 million episodes with similar parameterization to the deep Q-networks.

Network	Average Reward	Policy Score
QN	-0.2240	228
DQN(3,1)	-0.1904	247
DQN(3,7)	-0.0967	245
DQN(7,7)	-0.0902	254

- For comparison, these are the average rewards over time during training for networks with 3 layers and $k = 3$, 5, and 7.



- Lastly, these are graphs of policy score as a function of k and n.

Analysis and Future Exploration

- We were surprised by how hard blackjack was to learn. Although we outperformed traditional Q-learning, we could not learn the optimal policy. We originally assumed a small network would be sufficient.
- To improve our model, we could further tune our hyperparameters, use a more sophisticated network structure, or explore regularization.
- We don't account for player blackjack, which pays 1.5x the bet. This may have skewed our results, especially regarding doubling.

References

[1] Mnih et al. (2013, December). *Playing Atari with Deep Reinforcement Learning.*
[2] Roderick, MacGlashan, Tellex. (2017, November). *Implementing the Deep Q-Network.*
[3] https://www.blackjack-chart.com/