Deep Q-Learning in Blackjack: Theory and Implementation

Jerome Rodrigo
Student at Northeastern University
Boston, MA
rodrigo.j@northeastern.edu

***Abstract*—This paper explores the math behind my implementation of a Deep Q-Learning agent for my implementation of Blackjack. It examines the theoretical basis of Q-learning, the mathematical formulation of the Q-learning algorithm, and the neural networks used to approximate Q-values. Additionally, it details the implementation process, training methodology, and performance evaluation against a simple dealer. This paper is a personal deep dive to enhance my understanding of the math for reinforcement learning and Deep Q-Networks (DQN) Not for publication.**

I. Introduction

This project originally started as I believed that BlackJack would be a simple game to implement Reinforcement Learning, as there are not many states and actions to worry about. So I first started this project by implementing my own blackjack game and setting it up where I was the player, with a simple dealer bot as the dealer. The dealer's logic was pretty simple: either keep hitting till you beat the player or till it reaches 17 or greater.

I then started to implement Reinforcement Learning using Open AI's Gymnasium and Baseline3. I chose to use Deep Q-Networks as the action space is discrete, as the player can only hit or stay.

II. Reinforcement Learning

A. What is Q-Learning?

To start Q-learning is a value-based algorithm, which means it uses a value function to determine the reward, and from there it figures out the optimal policy, which is the best action to take given the state. Q-learning uses state-action value functions where both the state (current setup of the game, i.e., the dealer has 11 and the player has 10) and the action (hit or stay) are used to determine a number, which is the Q-value, quantifying How good it is to be in the state s and take an action given this state? The goal of Q learning is to get the best Q-function to maximize the reward, finding the optimal policy. Typically the agent/AI will start with an arbitrary Q-function and an exploration policy, where it'll randomly explore the environment. The Q-function is updated using the Bellman equation.

a) *Adjusted Bellman Equation for Q-Learning:*

$$Q(s,a) = \sum_{s'} P(s'|s,a)\left[R(s,a) + \max_{a'} Q(s',a')\right] \quad (1)$$

While this function looks complicated, it is quite simple. The left side is the value function, and it's a recursive function where it finds the maximum future value over all possible actions. The summation of probabilities is there due to the stochasticity of the environment. This is due to the fact that the next state is not guaranteed with the action taken. For example, in blackjack, the next state isn't guaranteed with a hit (i.e., you have 10, the next card is not guaranteed to be an ace), as the next card pulled is random. The summation accounts for all possible future states, weighted by their probabilities, in order to evaluate the expected value of taking each action.

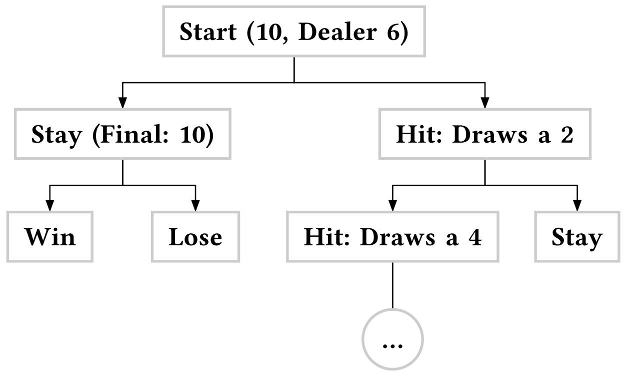

Fig.1 Tree Diagram representing how the Q-Function would roughly work

Note: This diagram simplifies the Q-function's decision process and does not fully capture the game's stochasticity. In reality, each hit could result in any remaining card, skyrocketting the number of branches, each weighted by its probability.

B. Deep Q-Networks (DQN)

So I opted for a DQN due to the stochastic nature of BlackJack. In standard Q-learning, I would have to make a Q-table/function for every possible state-action pair. The number of possible states is huge due to the nature of blackjack's random drawing; creating a Q-table for blackjack wouldn't be feasible due to all the possibilities you'd have to account for.

Instead of having to compute all possible states, I thought it would be better to use DQN as it uses 2 neural networks to approximate the Q-values. This allows it to generalize across similar states rather than computing for every possible scenario.

So, the DQN has two input layers, one is the player's current hand value, and the other is the dealers' current hand value. After that there is two layers of 64 neurons, and then a final output layer of 2 neurons, one for each q-value for each action (hit or stay).

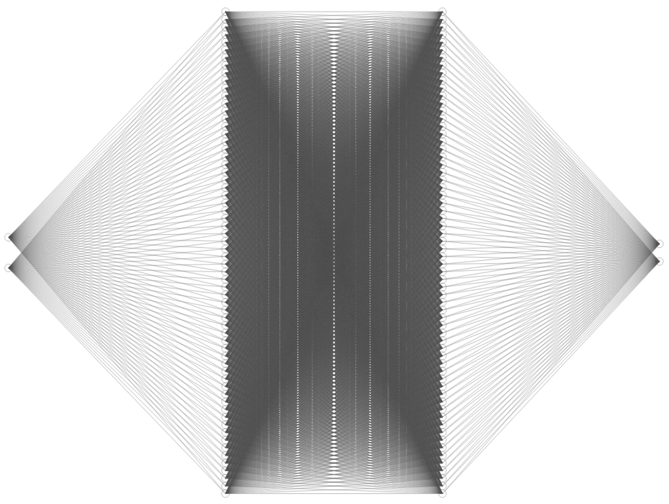

Fig. 1: Neural Network Architecture for DQN

So the first neural network is the *Q-network*, which is randomly intitialized, and the second is the *target network*, which is initially a copy of the Q-network. The final component for this system is the *experience replay buffer*, which interacts with the enviroment and stores data to train the Q-network. From there both the *Q-network* and the *target network* are fed a sample of the data from the *experience replay buffer*.

Then, the Q-network uses the current state and action to predict the Q-value for that specific action. The target network uses the next state and predicts the best Q-value out of all actions that can be taken in that state. From there the loss function is calculated with:

$$loss = (y - Q(s,a))^2 \tag{2}$$

where y is the target Q-value, and $Q(s,a)$ is the predicted Q-value. The loss function is then used to update the Q-network using backpropagation. Also, the target network is updated every T steps, typically every 1000 steps, where the target network weights are set to the Q-network weights in order to stabilize the training process, as if we just updated every single time without checking against the target network, the Q-network would be unstable and may not converge.

a) *Backpropagation: What is backpropagation?*
Backpropagation is a method used to update the weights of the neural network by minimizing loss using gradient descent.

Gradient descent is an algorithm from multivariable calculus that finds the local min of a function in machine learning and neural networks it looks to find the local min (Global min is typically too costly to calculate) of the loss function.

Now, to look at the neural network, we'll look at a neuron in the network, here's an equation for a neuron:

$$z = W * x + b \tag{3}$$

where W is the weight, x is the input, and b is the bias. The output of the neuron is then passed through an activation function, typically a sigmoid or ReLU function (which simply converts the output to a number between 0 and 1). The output of the neuron is then passed to the next layer.

From here backpropgation works by calculating how much the loss changes with respect to each weight, using chain rule. For each weight W we have to compute:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Q} * \frac{\partial Q}{\partial a} * \frac{\partial a}{\partial z} * \frac{\partial z}{\partial W} \tag{4}$$

Where:

$\frac{\partial L}{\partial W}$: is the derivative of the loss with respect to the weight.

$\frac{\partial L}{\partial Q}$: is the derivative of the loss with the predicted Q-value.

$\frac{\partial a}{\partial z}$: is the derivative of the activation function with the pre-activation output of the neuron.

$\frac{\partial z}{\partial W}$: is the derivative of the pre-activation output with respect to the weight.

From here we can use the gradient descent algorithm to update the weights for all neurons:

$$W = W - a * \tfrac{\partial L}{\partial W}$$

Where a is the learning rate, which tells the algorithm how fast to update the weights. If the learning rate is too high, the algorithm may overshoot the local min, and if it's too low, it may take too long to converge. In my implementation I used a learning rate of 0.001.

b) *What is the point of all this?:*
Well, through backpropagation and gradient descent, the neural network updates to minimize the loss function, which in turn updates the Q-values to find the best course of action to win in my game.

III. Training and Results

So I used OpenAI's Gymnasium and Baseline3′s DQN implementation to make this implementation. I trained on 100,000 steps and made the agent observe two things: its current hand value, and the dealer's current hand value. I used a simple reward system: winning was +1, losing was −1, and drawing was 0. Overall, I'd say it worked pretty well, especially since BlackJack is inherently a game of chance, where the house has the advantage. The bot won about 45% of the time, lost about 45% of the time, and tied about 10% of the time.