SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER SCIENCE

AI Assist in Blackjack

AI LAB: COMPUTER VISION AND NLP

Professors:
Daniele Pannone

Students:
Gabriele Dal Molin,
Orsi Alessandro

Academic Year 2024/2025

Contents

1 Introduction

The goal of this project is to develop an AI agent that is able to assist a player in a blackjack game. It primarily involves two components: first, a computer vision system that serves as the agent's eyes to recognize cards, and second, a decision-making mechanism to determine the most profitable move for the agent. The first component was achieved by using a light and yet quite effective approach, leveraging OpenCV's powerful features, in particular cv2, while the latter utilizes reinforcement learning, in particular Q-learning and deep Q-learning.

2 Method

2.1 Computer vision

2.1.1 Cards detection

The computer vision part takes as input a video, processed as a stream of frames. Every frame of the sequence will be treated as a single image and will go through the following steps. The first step consists of preprocessing the image. In order to do that, the image is first converted into grayscale and then smoothed by applying a filter to reduce the noise. Afterwards, the image is converted to a binary image using a threshold to decide whether a pixel should be black or white. After preprocessing

Figure 1: Image before processing

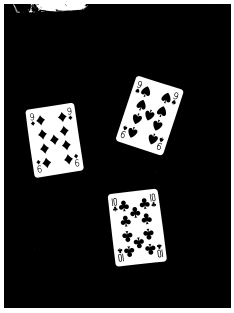

Figure 2: Image after processing

the image, we can now easily detect the cards by finding their contours. We ensure each contour we get corresponds to a card by filtering out contours of objects which span a small area and contours that the approximation does not correspond to a figure with 4 sides.

Figure 3: Cards' contours highlighted in green

2.1.2 Cards classification

After detecting the cards, it is now time to classify them, that is, find their rank and suit. In order to do that, we had to first create templates for every suit and number, which we will use later for comparison. Then, for every card, we proceed to approximate its contour to a polygon to obtain the corners of the card. Once we have the corners , we use them to remove the inclination and the effect of perspective on the card, so that we can get a clear image of the card. Once we have this standardized rectangular image of the card, we can use the top left corner to extract rank and suit similarly to what we did to detect the cards in the images.

Figure 4: Corner image

Figure 5: Suit image

Figure 6: Rank image

Once we have the suit and rank to classify the card we perform, we take our templates for the ranks and suits, and we perform the bitwise subtraction. We then

return the rank and the suit template which give us the smallest differences.

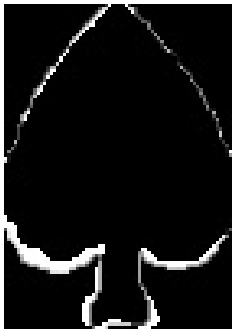

Figure 7: Suit Difference

Figure 8: Rank difference

2.2 Reinforcement Learning

2.2.1 Q-learning

Before looking at the slightly more complex deep q-learning, bare and simple q-learning was first studied and tested as a first approach to making the agent learn. The gymnasium documentation from the "Farama Foundation" was a good starting point. In fact, gymnasium provides a basic [1]blackjack environment that allows users to play and train their models. The observation space consists of a tuple of 3 elements:

- The sum of the cards the player is dealt
- The dealer's card which is face up
- Whether the ace is "usable" or not

As the action space, gymnasium's blackjack environment provides us only with:

- Hit
- Stand

The agent is rewarded with 1 point for winning, with 0 points for scoring as much as the dealer (situation referred to as a "push"), and -1 points for losing (either busting or scoring less than the dealer). It is also rewarded 1.5 points if it wins by natural blackjack.

The simplicity of such an environment, where the number of possible scenarios (and hence the number of rows in the q-table) lies in the few hundreds, justifies a q-learning approach. Gymnasium shows how a win rate of $\tilde{4}5\%$ is easily achievable

with the proper hyper parameters, a percentage which is considered to be the result of "a good blackjack strategy".

Generally speaking, having an agent learn through q-learning consists in having the agent act randomly at first, and then having it use previous experiences as a benchmark to have an idea of what to expect. This idea becomes progressively more accurate the more the agent trains. This means an important aspect of q-learning is finding a good exploration/exploitation balance that changes over time. In particular, we make the value of epsilon decrease linearly over time, by multiplying it by a decay factor.

$$a = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \arg\max_{a'} Q(s, a'), & \text{with probability } 1 - \epsilon \end{cases} \tag{1}$$

The knowledge of the agent is stored in a q-table that is filled over time. During the training process, this q-table is constantly updated by using the Bellman Equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right], \tag{2}$$

As previously stated, this approach works with a restricted state space like the one described previously, but struggles when the state space becomes larger.

The objective of this project, however, was for the agent to learn the strategies employed by a human player in a casino blackjack game. To achieve this, we needed to broaden the action space, resulting in the following configuration:

- Hit
- Stand
- *Split*[1]
- *Surrender*
- *Double down*

The 3 actions that were added are although only available in the first iteration of the game, that is to say when the first 2 cards are dealt to the player. In addition, the action "split" is only available if the player has a pair. To enforce these rules to the agent expanding the observation as follows was the most efficient way:

[1]The implementation of the split operation is a simplified one. This is because of how q-learning (or DQN) is structured, requiring a state, an action, a reward, and the next state. However split doesn't generate one future state but two. In our implementation if the agent splits it doubles its bet, but only considers one card of the pair, and then keeps playing as usual.

- The sum of the cards the player is dealt
- The dealer's card which is face up
- Whether the ace is "usable" or not
- *A boolean flag to allow selecting the "split" action*
- *A boolean flag to allow selecting the "surrender" and "double down" actions*

Furthermore new actions mean new rewards, in particular a player that splits doubles their bet, hence winning gives a reward of +2 and losing a reward of -2. Doubling down also consists of doubling the bet, but consists of hitting once and standing, without the option to hit again. Surrendering means acknowledging the dealer has a much better hand and lets the player lose only half their bet, hence a reward of -0.5 is given.

These additions to both action space and state space translate into a much much greater cardinality of state space, consisting of $\tilde{1}500$ entries.

When using bare bone q-learning as before the agent struggled to learn a good policy, failed to explore the new actions and relied more on just hit/stand. For these reasons deep q-learning was explored.

2.2.2 Deep Q-learning

Much of the principles discussed regarding Q-learning also apply here. However, instead of utilizing a Q-table, deep Q-learning addresses space complexity issues by approximating a Q-function. In fact, instead of updating a Q-table the goal is now to minimize a loss function.

The additional elements include a neural network (NN) that approximates the function and a replay buffer to store the experiences for the NN to learn from. After a batch of specified size (256 in this case) is sampled, it is filtered to include only non-terminal states, as computing q-values for terminated states is unnecessary. The NN receives a state input, calculates the predicted q-values and the temporal difference. Finally, the loss is computed and backpropagated as standard practice. In particular we chose to minimize the Huber Loss, as it's more robust and less sensitive to outliers than MSE. This is because as a piecewise function it behaves as MSE (quadratic approach) if the error is small enough, but as an absolute error (linear approach) if it's larger.

$$L(\theta) = \begin{cases} \frac{1}{2}(y - Q(s,a;\theta))^2, & \text{if } |y - Q(s,a;\theta)| < \delta \\ \delta \cdot (|y - Q(s,a;\theta)| - \frac{1}{2}\delta), & \text{otherwise} \end{cases} \tag{3}$$

Of particular note is the fact that the target value is each time (in our case every 4 episodes) computed from the batch of experiences that is sampled, meaning it is not a fixed value like a label in supervised learning, but a moving target. This explains why the loss function will not look as smooth as it would normally do.

For the NN we tried multiple hyper-parameters combinations, but got the best results with the following:

Table 1: Neural network hyperparameters.

Hyperparameter	**Value**
Number of layers	11
Discount factor (γ)	0.999
Learning rate (α)	0.0001
Batch size	256
Decay factor	20.000

These parameters were mostly inspired by Mnih et al.'s Playing Atari with Deep Reinforcement Learning and Allen Wu's implementation, which can be found in the references.

3 Conclusions

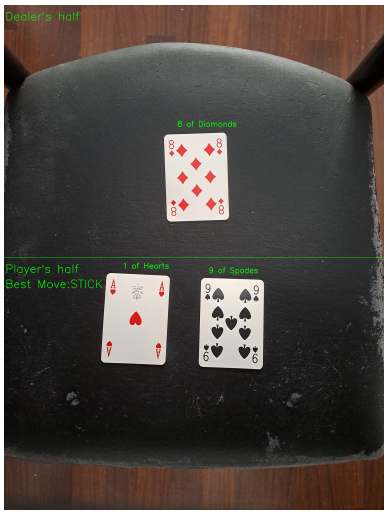

Figure 9: Image of final project

In conclusion, by putting everything together, we can now get real-time suggestions that can help us to play blackjack optimally. However, the final model comes with

some flaws. In order to keep the computer vision part light and fast, we had to make some trade-offs and accept some compromises. The model is, in fact, 1) vulnerable to excessive lightning and 2) needs a dark background, as well as the input recordings coming from a source placed perpendicularly to the surface on which the cards are on, to work smoothly.

The agent was able to successfully learn a good policy, but not equivalent to blackjack's basic strategy, the mathematical model to play optimally. Below, the loss over time can be observed:

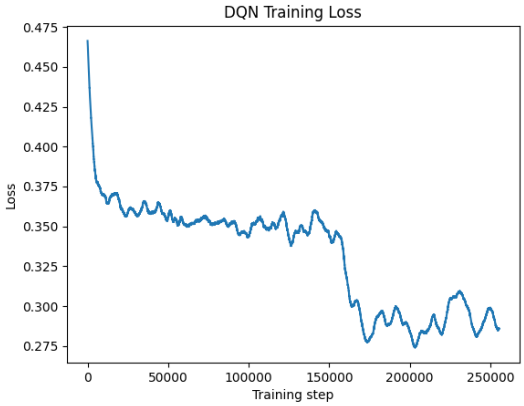

Figure 10: Loss over training steps

Clearly, the agent was able to learn that on high cards (17 and above) it should always stand, while on low cards (below 11) it should always hit. It managed to successfully learn some scenarios where it should always split (like pair of 8s, or pair of aces), and when it should surrender. However it doesn't sufficiently utilize double down and has a tendency to abuse split, probably because of the simplified implementation. A similarity of 66% with basic strategy was achieved, which tells us there is a decent margin of improvement that is yet to be found.

References

[1] Farama Foundation. Gymnasium blackjack environment. https://gymnasium.farama.org/environments/toy_text/blackjack/.

[2] Farama Foundation. Gymnasium documentation. https://gymnasium.farama.org/.

[3] Timothy Leung. Deriving blackjack's basic strategy using reinforcement learning. https://actamachina.com/posts/blackjack, 2023.

[4] Mnih et al.'s'. Human-level control through deep reinforcement learning. https://www.nature.com/articles/nature14236.

[5] Stanford University CS230. Stanford cs230 project: Blackjack reinforcement learning. https://cs230.stanford.edu/files_winter_2018/projects/6940282.pdf, 2018.