Architectures of Intelligence
Assignment 4

Matthijs Prinsen *(S4003365)*
Marinus van den Ende *(s5460484)*
Group 75

December 9, 2024

Introduction

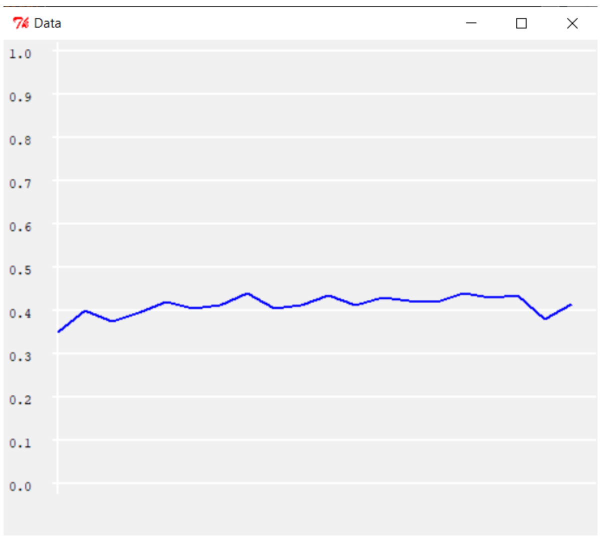
"It's blackjack time - babyyy"   Dunkey Video Games, 2022

The Learning Graph

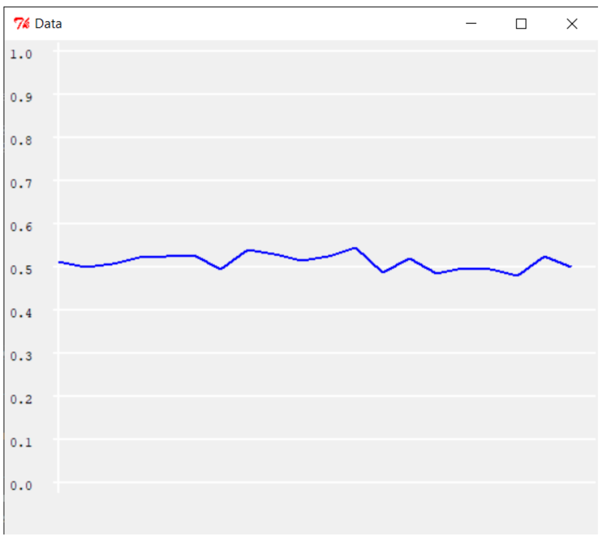
In the learning graph, we see an increase in winning chances from 38% to about 42%. This increase over time demonstrates that the model learns to perform the task better.

Our model run against *game0* (onehit-learning 100) performed well, comfortably clearing the 36% minimum requirement with results often reaching 42%. Figure 1 (left) shows the learning graph for *game0*, where the win rates are detailed for different quarters of the game.

Our model run against *game1* showed significantly better performance, with results exceeding 50%. Figure 1 (right) presents the corresponding learning graph, illustrating consistent improvements over time.



(a) Model performance for *game0*



(b) Model performance for *game1*

Figure 1: Comparison of model performance on *game0* and *game1*

The results for both games are summarized in Table 1, aggregated by game and quarter.

Game	First Quarter	Second Quarter	Third Quarter	Fourth Quarter
game0	38.72%	41.40%	42.36%	41.92%
game1	51.24%	52.06%	51.16%	49.84%

Table 1: Aggregated win rates by game and quarter

Strategy

We need to write a model that decides when to *hit* or *stand* while trying to leverage past experiences. We make this decision (*hit* or *stand*) only once after the game presents the first goal (state start). We aim to retrieve, from our declarative memory, past situations with a similar starting total and the resulting action taken for that total.

Thus, all we need to save in the `learning-info` chunk is the **starting total** and the **resulting action**. Our model can either retrieve a past fact (learning-info chunk) and execute the corresponding action or fail to recall a fact and default to *hit*.

Finally, after the game ends and we are in the goal (state results), we use a few productions to encode learning from the outcome. These productions match specific win or loss/bust conditions. There are five productions in total, each addressing a distinct result condition: *results-hit-win*, *results-stay-win*, *results-stay-bust*, *results-opponent-stay-won*, and *results-lost*. These are explained further in the code comments, but essentially, each contains logic based on the game state to encode the optimal match of the starting total and action.

Storing Facts

When we (implicitly) clear the imaginal buffer, the chunk in the buffer is sent to declarative memory. This model saves the following information into a chunk:

- **Starting total (mstart):** Since decisions are made only after the second card is dealt, we use the total (mstart) to decide whether to *hit* or *stand*. While more information could be leveraged, we limit the data to avoid false or misleading connections.
- **The action:** We encode the best action for the hand dealt.[1]

We chose not to save additional information to prevent the model from forming unexpected connections.

Partial Matching

The use of partial matching is critical for faster learning. Since we have only a hundred rounds to learn from, we aim to use the available information as early and often as possible. Partial Matching helps retrieve chunks from memory even if they don't match perfectly. Without it, the model wouldn't be able to learn efficiently; it would instead default to hitting when a better decision was possible based on prior experience. This would make it less efficient for the model to adapt or improve.

When we use fewer slots in a retrieval, it opens up more possible matches, but they often aren't relevant to the current game. On the other hand, using more slots makes the retrieval more

[1]Our productions aim to encode the best match between the starting total and the action taken. This ensures that correct actions are encoded more frequently, increasing the activation of the corresponding chunks and improving the model's performance.

specific, but then we might not find a result at all. This is where the mismatch penalty (`:mp`) comes in. It allows a chunk that is similar to our request, but not the same, to be retrieved instead of nothing. It allows us to prioritize chunks that are close enough while filtering out anything irrelevant.

This approach feels realistic for blackjack because, like humans, the model relies on past experiences to make decisions, even if the situation isn't identical. Of course, it's not perfect—there's no room for intuition or gut instinct, which are often a big part of human decision-making. Still, it's a reasonable way to model how we learn and adjust over time.

Plausibility of the Model

The model was designed to resemble a player instructed to look at the total of their cards and make a *hit* or *stay* decision based on that information. After learning the game outcome, the player considers both their own and their opponent's actions.

This approach to learning seems plausible because the steps involved are very basic. When running the model, we observe an increase in the win rate, indicating effective learning. Comparing this against test data will help evaluate how well certain aspects of the model fit.